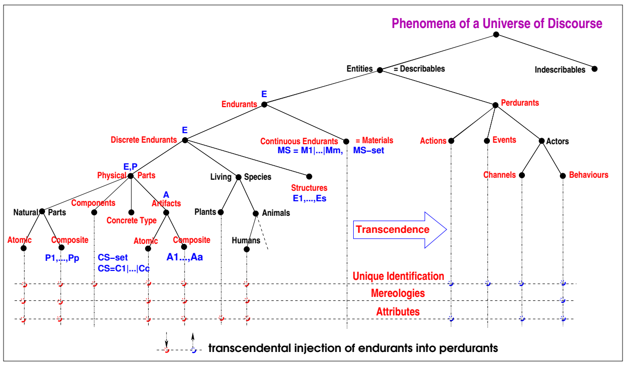
Dines Bjørner

Domain Science & Engineering

A Foundation for Software Development

Version 0

June 17, 2018: 10:12 am



A Monograph of Seven Papers and Six Case Studies

Fredsvej 11, DK–2840 Holte, Denmark
Technical University of Denmark
E-mail: `bjorner@gmail.com`. **URL:** `www.imm.dtu.dk/~dibj`

This compendium was put together in May 2018
It was first released June 17, 2018: 10:12 am

Kari; Charlotte and Nikolaj; Camilla, Marianne, Katrine, Caroline and Jakob

the full meaning of life

Preface

The Triptych Dogma

In order to *specify* **software**,
we must understand its requirements.
In order to *prescribe* **requirements**,
we must understand the **domain**,
so we must **study, analyse** and **describe** it.

General

The thesis of this collection of papers is that domain engineering is a viable, yes, we would claim, necessary initial phase of software development. I mean this rather seriously: How can one think of implementing **software**, preferably satisfying some **requirements**, without demonstrating that one understands the **domain** ? So in this collection of papers I shall explain what domain engineering is, some of the science that goes with it, and how one can 'derive" requirements prescriptions (for computing systems) from domain descriptions. But there is an altogether different reason, also, for presenting these papers: Software houses may not take up the challenge to develop software that satisfies customers expectations, that is, reflects the domain such as these customers know it, and software that is correct with respect to requirements, with proofs of correctness often having to refer to the domain. But computing scientists are shown, in these papers, that domain science and engineering is a field full of interesting problems to be researched. We consider domain descriptions, requirements prescriptions and software design specifications to be mathematical quantities.

A Brief Guide

I have collected seven documents in this monograph:

We urge the reader to study the **Contents** listing and from there to learn that there is a **Bibliography** common to all six chapters, two example appendices, **An RSL Primer**, a set of indexes into definitions, concepts, examples, analysis and description prompts, and an index of **RSL Symbols**.

I have also collected 6 experimental case studies in this compendium:

Some are still in a stage of "development".

•••

This monograph complements [14, 15, 16].

More specifically, this monograph replaces Chapters 10–11 of [16]. Much has happened since 2005 when those chapters were written.

Contents

Part IV Issues of Philosophy

Part V Summing Up

Part **VI** Experimental Case Studies

Part **VII** RSL

0

Domains: Science & Engineering

Dear reader! You are about to embark on a journey. The monograph in front of you is long ! But it is not the number, 230 pages, or duration of your studying that monograph that I am referring to. It is the mind that should be prepared for a journey. It is a journey into a new realm. A realm where we confront the computer & computing scientists with a new universe: a universe in which we build a bridge between the ***informal*** world, that we live in — the context for eventual, ***formal*** software — and that ***formal*** software.

The bridge involves a novel construction, new in computing science: a ***transcendental deduction***. We are going to present you with, we immodestly, claim, a new way of looking at the "origins" of software, the domain in which it is to serve. We shall show a ***method, a set of principles and techniques and a set of languages*** — some formal languages, some "almost" formal languages, and the informal language of usual computing science papers — for a systematic to rigorous way of ***analysing & describing domains.*** We immodestly claim that such a method has not existed before.

0.1 Domains

0.1.1 What Do We Understand by a Domain ?

Definition 1 **Domain:** By a **domain** we shall understand a **rationally describable** segment of a **human assisted** reality, i.e., of the world, its **physical parts**, and **living species**. These are **endurants** ("still"), existing in space, as well as **perdurants** ("alive"), existing also in time. Emphasis is placed on **"human-assistedness"**, that is, that there is ***at least one (man-made)*** **artifact** and that **humans** are a primary cause for change of endurant **states** as well as perdurant **behaviours** Among the ***entities*** we may, in any one specific **domain analysis & description**, include **humans** in so far as their `properties` can be objectively analysed & described ■

0.1.2 What Do We Understand by a Domain Description ?

Definition 2 **Domain Description:** By a **domain description** we shall understand a combination of **narration** and **formalisation** of a domain. A **formal specification** is a collection of **sort**, or **type** definitions, **function** and **behaviour** definitions, together with **axiom**s and **proof obligation**s constraining the definitions. A **specification narrative** is a natural language text which in terse statements introduces the names of (in this case, the domain), and, in cases, also the definitions, of sorts (types), functions, behaviours and axioms; not anthropomorphically, but by emphasizing their properties ■

Domain descriptions are (to be) void of any reference to future, contemplated software, let alone IT systems, that may support entities of the domain. As such ***domain models***[1] can be studied separately, for

[1] We use the terms ***'domain descriptions'*** and ***'domain models'*** interchangeably.

their own sake, for example as a basis for investigating possible domain theories, or can, subsequently, form the basis for requirements engineering with a view towards development of ('future') software, etc. Our aim is to provide a method for the precise analysis and the formal description of domains.

0.2 A Didactice Base

This monograph expresses an approach to software development that has many facets. These will be characterised in terms of the concept of ***didactic base***.

Definition 3 **Didactice Base:** By a *didactic base* we shall understand the knowledge "spheres" within which we operate ■

Our *didactic base* for software development is outlined below. That base is also suggested as the *didactic base* for software engineering.

0.2.1 Philosophy

But there is an even more fundamental issue "at play" here. It is that of philosophy. Let us briefly review some aspects of philosophy.

Definition 4 **Philosophy:** ***Philosophy*** is[2] the study of general and fundamental problems concerning matters such as existence, knowledge, values, reason, mind, and language ■

Definition 5 **Metaphysics:** ***Metaphysics*** is a branch of ***philosophy*** that explores fundamental questions, including the nature of concepts like ***being, existence***, and ***reality*** ■[3]

Traditional metaphysics seeks to answer, in a "suitably abstract and fully general manner", the questions: ***What is there ?*** and ***And what is it like ?*** [4]. Topics of metaphysical investigation include existence, objects and their properties, space and time, cause and effect, and possibility.

Definition 6 **Epistemology:** ***Epistemology*** is the branch of philosophy concerned with the theory of knowledge[5] ■

Epistemology studies the nature of knowledge, justification, and the rationality of belief. Much of the debate in epistemology centers on four areas: (1) the philosophical analysis of the nature of knowledge and how it relates to such concepts as truth, belief, and justification, (2) various problems of skepticism, (3) the sources and scope of knowledge and justified belief, and (4) the criteria for knowledge and justification. A central branch of epistemology is ***ontology***.

Definition 7 **Ontology:** the investigation into the basic categories of being and how they relate to one another.[6]

We shall base some of our modelling decisions of Kai Sørlander's Philosphy [17, 18, 19, 20]. A main contribution of Sørlander is, on the philosophical basis of the ***possibility of truth*** (in contrast to Kant's ***possibility of self-awareness***), to ***rationally*** and ***transcendentally deduce*** the absolutely necessary conditions for describing any world.

[2] From Greek $\phi\iota\lambda\sigma o\phi\iota\alpha$, philosophia, literally ***love of wisdom***.
[3] ■ is used to signal the end of a characterisation, a definition, or an example.
[4] https://en.wikipedia.org/wiki/Metaphysics
[5] https://en.wikipedia.org/wiki/Epistemology
[6] https://en.wikipedia.org/wiki/Metaphysics

These conditions presume a ***principle of contradiction*** and lead to the ***ability*** to ***reason*** using ***logical connectives*** and to ***handle asymmetry, symmetry*** and ***transitivity***. ***Transcendental deductions*** then lead to ***space*** and ***time***, not as priory assumptions, as with Kant, but derived facts of any world. From this basis Sørlander then, by further transcendental deductions arrive at kinematics, dynamics and the bases for Newton's Laws. And so forth.

We build on Sørlander's basis to argue that the domain analysis & description calculi are necessary and sufficient and that a number of relations between domain entities can be understood transcendentally and as "variants" of Newton's Laws !

We thus build on Sørlander's basis to argue that the domain analysis & description calculi are necessary and sufficient and that a number of relations between domain entities can be understood transcendentally and as "variants" of Newton's Laws !

0.2.2 A New Area of Computing Science

Domain science & engineering marks a new area of ***computing science***. Just as we are ***formalising*** the ***syntax and semantics of programming languages***, so we are ***formalising*** the ***syntax and semantics of human-assisted domains***. Just as ***physicists*** are studying ***mother nature***, endowing it with ***mathematical models***, so we, ***computing scientists***, are studying these ***domains***, endowing them with ***mathematical models***, A difference between the endeavours of physicists and ours lies in the models: the physics models are based on ***classical mathematics, differential equations*** and ***integrals,*** etc., our models are based on ***mathematical logic, set theory***, and ***algebra.***

0.2.3 Software As Mathematical Objects

Our base view is that ***computer program***s are ***mathematical object***s. That is, the text that makes up a computer program can be reasoned about. This view entails that computer program specifications can be reasoned about. And that the ***requirements prescription***s upon which these specifications are based can be reasoned about. This base view entails, therefore, that specifications, whether ***software design specification***s, or ***requirements prescription***s, or ***domain description***s, must [also] be ***formal specification***s. This is in contrast to considering ***software design specification***s being artifacts of sociological, or even of psychological "nature".

Since the kind of programs that we are focusing on are developed to solve problems residing in the ***domains*** as we have characterised domains above, the mathematics of software evolves around ***mathematical logic, recursive function theory*** and ***algebra.*** This is in contrast to programs that solve problems in physics: thir mathematics evolves around partial differential equations and, in general, mathematical analysis.

0.2.4 Semiotics

Definition 8 Semiotics: By *semiotics* of a language, or a system of sentential or other structures, we understand a "sum" of the:

- *pragmatics*
- *semantics* and
- *syntax*

of that language or system ■

Definition 9 Syntax: By ***syntax*** we understand (i) the ways in which simple elements, e.g., words or atomic parts, are arranged (cf. Greek: ***syntaxis:*** arrangement) to show meaning (cf. semantics) within and between sentences, and (ii) rules for forming ***syntactically correct*** sentences [21] ■

Definition 10 Semantics: *Semantics* is the study and knowledge (including specification) of meaning in language [21].

By *formal semantics* we understand a semantics, M, such that we can reason about properties of what the syntax describes.

```
type                                  value
  Syntax, Semantics                     M: Syntax → Semantics ■
```

Chapter 3 sketches a semantics of the ***domain analysis & description process*** of Chapter 1.

Definition 11 Pragmatics: (I) *Pragmatics* is the study and practice of the factors that govern our choice of language in social interaction and the effects of our choice on others [21]. By pragmatics we thus understand issues of why we use a special construct, of why we constrain such a construct and of why we endow it with certain properties, and so on. So we summarise: *pragmatics* is the study of language in context, and the context-dependence of various aspects of linguistics interpretation ■

Discussion

In numerous formal specification projects, with students at universities and in industry, we have found that stopping up, now-and-then, reflecting on issues of semiotics: the rôle of syntax, semantics and pragmatics, receptively in the analysis and in the specification efforts, has helped bring a focus on these semiotics issues, and thereby, to us and our colleagues, improved our enjoyment and the results !

[14, 15, 16, Chapters 6–9 of Vol. 2], over 90 pages, elaborates on the linguistics issues of pragmatics, semantics, syntax and semiotics, in that order.

0.2.5 Method, Methodology and Formal Method

Definition 12 Method: By a **method** we shall understand

- a set of **principles**
  - for **selecting**
  - and **applying**
  - a number of **analysis & synthesis**
  - **techniques** and
  - **tools**
- in order to achieve a goal —
- where that goal here is to develop a software specification
- whether that specification be a
  - a domain description,
  - a requirements prescription,
  - a software design and code,
  - or the first or last two, or all of these ■

Definition 13 Methodology: By **methodology** we shall understand the study and knowledge of one or more methods ■

Definition 14 Formal Method: By **formal method** we shall understand

- a method several of whose techniques and tools can be explained **mathematically**, such as, e.g.,
  - refinements,
  - tests, model checks, theorem proofs,
  - specification language syntax, semantics and proof systems ■

This monograph is about a method, in some areas a formal method, for understanding and documenting such understandings of domains.

Definition 15 **Formal Software Development:** By a **formal software development method** we shall understand a formal method where domain descriptions, requirements prescriptions and software designs are expressed in mathematically founded specification languages with the possibility of proving properties of these specifications, of steps and stages of development (refinements within domain descriptions, requirements prescriptions, software designs and between these) — properties such as correctness of software designs with respect to requirements, and satisfaction of user expectations (from software) with respect to domains ■

0.2.6 A Triptych of Software Development

Definition 16 **The Triptych Dogma:**

- Before **software** can be **designed & coded**
- we must have a reasonable grasp of what is **expected & required** from that software,
- and before we can **prescribe** those **expectations & requirements**
- we must have a reasonable grasp of the **domain**, i.e., be able to **describe** it ■

As a consequence we can claim that:

- **Software Systems Development** can be "divided" into three phases:
  - **Domain Science & Engineering**
  - **Requirements Engineering**
  - **Software Design**

Definition 17 **The Triptych Approach to Software Development:** By a **triptych software development** we understand a development which, in principle, starts with either studying an existing or developing a domain description, then proceeds to systematically deriving a requirements prescription from the domain description, and finally designs and codes the software from the requirements prescription ■

This monograph primarily focuses on the domain aspect of the triptych of software development. Chapters 5–6 links domain science & engineering to requirements engineering and to software systems development, respectively.

0.2.7 Informatics & IT

Informatics

- By **informatics** we shall understand a confluence of
  - mathematics: "pure" as well as "applied",
  - computer & computing science, and
  - software.

To us informatics is a universe of quality: correct, fit-for-purpose and pleasing

IT: Information Technology

- By **information technology** we shall understand a confluence of
  - hardware
  - the natural science-based technologies that *"go into making"* hardware:

- electronics,
- mechanics,
- chemistry,
- et cetera.

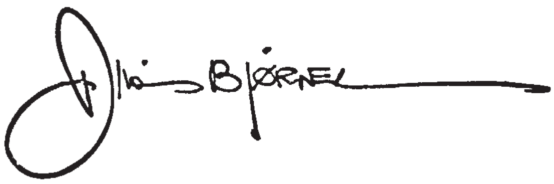
To us IT is a universe of quantity: faster, larger, cheaper, etc.

0.3 Some Editorial Remarks

As has been mentioned, this monograph has been put together essemtially from seven documents [1, 3, 5, 7, 9, 11, 13]. Of these the first six have been extensively edited from earlier publications [2, 4, 6, 8, 10, 12]. Only [13] is new in this monograph. The editing of [1, 3, 5, 7, 9, 11, 13] into Chapters 1–7 includes (i) consolidating (including removing) material from the introductory sections of [1, 3, 5, 7, 9, 11, 13] into this chapter, i.e., Chapter 0; and (ii) similary consolidating (including removing) material from the concluding sections, including the bibliographies of [1, 3, 5, 7, 9, 11, 13] into Chapter 8. (iii) In addition this monograph brings six case studies, Appendices A–F, an RSL primer, Appendix G, and extensive indexes, Appendix H. I urge the reader to carefully study the contents listing and from that “discover” the structure of presentation in this monograph and to make use of the “elaborate” indexes at the very end of the monograph.

0.4 Acknowledgements

I thank colleagues in Austria, China, Denmark, Germany, France, Norway, Russia, Portugal, Singapore, Scotland, Sweden and Ukraine: Bernhard K. Aichernig, Yamine Ait Ameur, Arutyun Avetisyan, Luís Soares Barbosa, Alan Bundy, Chin Wei Ngan, Olivier Danvy, Jin Song Dong, Dominique Méry, Andreas Harmfeldt, Magne Haveraaen, He Ji Feng, Otthein Herzog, Steve McKeever, Jens Knoop, Hans Langmaack, Nikolaj Nikitchenko, Ole N. Oest, José Nuno Oliveira, Wolfgang Paul, Alexander Petrenko, Sun Meng, Wang Shu Lin. Franz Wotawa, Yang ShaoFa, Zhan Nai Jun and Zhu Hui Biao. Their invitations over the last 10 years to present my work, their comments on recent papers, and their acting as sounding boards for the case studies that lead to a number of clarifications, simplifications and solidifications of the ***domain analysis & description*** method of [2] now reported in the present monograph are much appreciated.



Dines Bjørner. June 17, 2018: 10:12 am
Fredsvej 11, DK–2840 Holte, Denmark

Part I

The Domain Analysis & Description Method

1

Domains: Analysis & Description

1.1 Introduction

We present a ***method*** for **analysing and describing domains**.

Emphasis is placed on **"human-assistedness"**, that is, that there is ***at least one (man-made)*** **artifact** and that **humans** are a primary cause for change of endurant **states** as well as perdurant **behaviours**

Domain science & engineering marks a new area of ***computing science***. Just as we are ***formalising*** the ***syntax and semantics of programming languages***, so we are ***formalising*** the ***syntax and semantics of human-assisted domains***. Just as ***physicists*** are studying ***mother nature***, endowing it with ***mathematical models***, so we, ***computing scientists***, are studying these ***domains***, endowing them with ***mathematical models***, A difference between the endeavours of physicists and ours lies in the models: the physics models are based on ***classical mathematics, differential equations*** and ***integrals,*** etc., our models are based on ***mathematical logic, set theory***, and ***algebra.***

1.1.1 Precursor

The present chapter is a revision od [1] which itself is a revision of the published [2]. The revision considerably simplifies and considerably extends the domain analysis & description calculi of [2]. The major revision that prompts this complete rewrite is due to a serious study of Kai Sørlander's Philosophy. As a result we extend [2]'s ontology of endurants: describable phenomena that exists in space, to not only cover those of **physical phenomena**, but also those of **living species**, notably **humans**, and, as a result of that, our understanding of discrete endurants is refined into those of **natural parts** and **artifacts**. A new contribution is that of **intentional "pull"** akin to the ***gravitational pull*** of physics. Both this chpater and [1, 2] are the result of extensive "non-toy" example case studies, see Example 1.1 on Page 14. These were carried out in the years since [2] was first submitted (i.e., 2014). The present chapter omits the extensive introduction and closing of [2], Sects. 1.1 and 1.9, as well as the very many "interwoven" examples of [2]. Instead Sect. 1.8 (Pages 46–55) shows one, rather comprehensive, larger example that illustrates many aspects of the methodology. Most notably, however, is a clarified view on the transition from **parts** to **behaviours**, a **transcendental deduction** from ***domain space*** to ***domain time***.

1.1.2 What is this Chapter About?

We present a ***method*** for ***analysing &***[1] ***describing domains***.

1.1.3 The Four Languages of Domain Analysis & Description

Usually mathematics, in many of its shades and forms are deployed in ***describing*** properties of nature, as when pursuing physics, Usually the formal specification languages of ***computer & computing science***

[1] By *A*&*B* we mean one topic, the confluence of topics *A* and *B*.

have a precise semantics and a consistent proof system. To have these properties those languages must deal with ***computable objects***. ***Domains are not computable.***

So we revert, in a sense, to mathematics as our specification language. Instead of the usual, i.e., the classical style of mathematics, we "couch" the mathematics in a style close to RSL [22, 14]. We shall refer to this language as RSL^+. Main features of RSL^+ evolves in this chapter, mainly in Sect. 1.7.3.

Here we shall make it clear that we need three languages: (i) an **analysis language**, (ii) a **description language**, i.e., RSL^+, and (iii) the language of explaining **domain analysis & description**, (iv) in modeling "the fourth" language, the domain, its syntax and some abstract semantics.

The Analysis Language

Use of the ***analysis language*** is not written down. It consists of a number of single, usually `is_` or `has_` prefixed ***domain analysis prompt*** and ***domain description prompt*** names. The **domain analysis prompts** are:

They apply to phenomena in the domain, that is, to *"the world out there"!* Except for `observe_endurants` and `attribute types` these queries result in truth values; `observe_endurants` results in the ***domain scientist cum engineer*** noting down, in memory or in typed form, suggestive names [of endurant sorts]; and `attribute_types` results in suggestive names [of attribute types]. The truth-valued queries directs, as we shall see, the ***domain scientist cum engineer*** to either further analysis or to "issue" some ***domain description prompt***s. The 'name'-valued queries help the human analyser to formulate the result of ***domain description prompt***s

The **domain description prompts** are:

Again ***they apply to phenomena in the domain,*** that is, to *"the world out there"!* In this case they result in RSL^+Text !

The Description Language

The **description language** is RSL^+. It is a basically applicative subset of RSL [22, 14], that is: no assignable variables. Also we omit RSL's elaborate ***scheme, class, object*** notions. This subset is then "extended" with the following clauses – where E stands for an endurant sort, P for part sort, PoC for part or component sort, PoM for part or material sort, and PU, finally, for part or unique identifier sort.

- ***Structures, Parts, Components and Materials:***
  - **obs_endurant_sorts**_E, dfn. 1, [o] pg. 23
  - **is**_E_i, dfn. 1, [i] pg. 23
  - **obs_part**_T: P, dfn. 2, [t_2] pg. 24
  - **is**_K_i, dfn. 3, [i] pg. 25
  - **obs_part**_T: P, dfn. 4, [o] pg. 16
  - **is**_M_i, dfn. 4, [i] pg. 26
- ***Part and Component Unique Identifiers:***
  - **uid**_P, dfn. 5, [u] pg. 27
  - **uid**_K_i, dfn. 3, [u] pg. 25
- ***Part Mereologies:***
  - **obs_mereo**_P, dfn. 6, [m] pg. 28
- ***Part and Material Attributes:***
  - **attr**_A_i, dfn. 7, [a] pg. 30
  - **obs_attrib_values**_PoM, dfn. 7, [v] pg. 30
  - **is**_A_i: P, dfn. 7, [i] pg. 30
  - **is**_A_{i_j}: M_i, dfn. 4, [a] pg. 26
- ***Endurant and Unique Id. Generic***
  - ηE_i dfn. 1, [η] pg. 23
    if is_part(e_i): η(e_i)≡≪ E_i ≫ i:[1..m]
  - ηS_i, dfn. 2, [t_3] pg. 24
  - ηPI, dfn. 5, [u] pg. 27
- ***Miscellaneous:***
  - **obs_components**_P:P→KS, dfn. 3, [o] pg. 25
  - et cetera.

The Language of Explaining Domain Analysis & Description

In explaining the ***analysis & description prompts*** we use a natural language which contains terms and phrases typical of the technical language of ***computer & computing science***, and the language of ***philosophy***, more specifically ***epistemology*** and ***ontology***. The reason for the former should be obvious. The reason for the latter is given as follows: We are, on one hand, dealing with real, actual segments of domains characterised by their basis in nature, in economics, in technologies, etc., that is, in informal "worlds", and, on the other hand, we aim at a formal understanding of those "worlds". There is, in other words, the task of explaining how we observe those "worlds", and that is what brings us close to some issues well-discussed in ***philosophy***. We shall elaborate further on the ***philosophy*** issues in Sect. 1.9.3.

The Language of Domains

We consider a domain through the ***semiotic looking glass*** of its ***syntax*** and its ***semantics***; we shall not consider here its possible ***pragmatics.*** By ***"its syntax"*** we shall mean the form and "contents", i.e., the ***external*** and ***internal qualities*** of the ***endurant***s of the domain, i.e., those ***entities*** that endure. By ***"its semantics"*** we shall, by a ***transcendental deduction***, mean the ***perdurants***: the ***actions***, the ***events***, and the ***behaviours*** that center on the the endurants and that otherwise characterise the domain.

1.1.4 An Analysis & Description Process

It will transpire that the **domain analysis & description** process can be informally modeled as follows:

```
type
   V = PVAL | KVAL | MVAL
variable
   new:V-set := {uod:UoD} ;
   gen:V-set := {} ;
   txt:Text := {} ;
value
   discover_sorts: Unit → Unit
   discover_sorts() ≡
        while new ≠ {} do
            let v:V • v ∈ new in
            new := new \ {v} || gen := gen ∪ {v} ;
            is_P(v) →
                ( is_atomic(v) → skip ,
                  is_composite(v) →
                     let {e1:E1,e:E2,...,en:En} = observe_endurants(v) in
                     new := new ∪ {e1,e,...,en} ; txt := txt ∪ observe_endurant_sorts(e) end ,
                  has_concrete_type(v) →
                     let {s1,s2,...,sm} = new_sort_values(v) in
                     new := new ∪ {s1,s2,...,sm} ; txt := txt ∪ observe_part_type(v) end ) ,
```

```
        is_K(v) → let {k1:K1,k2:K2,...,kn:Kn} = observe_components(v) in
                  new := new ∪ {k1,k2,...,kn} ; txt := txt ∪ observe_component_sorts(v) end ,
        is_M(v) → txt := txt ∪ observe_material_sorts(v)
        end
    end

discover_uids: Unit → Unit
discover_uids() ≡ for ∀ v:(PVAL|KVAL) • v ∈ gen do txt := txt ∪ observe_unique_identifier(v) end
discover_mereologies: Unit → Unit
discover_mereologies() ≡ for ∀ v:PVAL • v ∈ gen do txt := txt ∪ observe_mereology(v) end
discover_attributes: Unit → Unit
discover_attributes() ≡ for ∀ v:(PVAL|MVAL) • v ∈ gen do txt := txt ∪ observe_attributes(v) end
analysis+description: Unit → Unit
analysis+description() ≡ discover_sorts(); discover_uids(); discover_mereologies(); discover_attributes()
```

Possibly duplicate **text**s “disappear” in the output text, txt.

1.1.5 Structure of this Chapter

Sections 1.2–1.7 form the core of this chapter. Section 1.8 brings a “large” example that is forward-referred to in Sects. 1.2–1.7 and refers (backwards) to Sects. 1.2–1.7. Section 1.2 introduces the first concepts of domain phenomena: ***endurants*** and ***perdurants***. Their characterisation, in the form of “definitions”, cannot be mathematically precise, as is usual in computer science papers. Section 1.3 analyses the so-called ***external qualities*** of ***endurants*** into ***natural parts, structures, components, materials, living species*** and ***artifacts***. In doing so it covers the ***external qualities analysis prompts.*** Section 1.4 covers the ***external qualities description prompts*** Section 1.5 analyses the so-called ***internal qualities*** of ***endurants*** into ***unique identification***, ***mereology*** and ***attributes***. In doing so it covers both the ***internal qualities analysis prompts*** and the ***internal qualities description prompts*** Sections 1.3–1.5 have covered what this chapter has to say about ***endurants.*** Section 1.6 “bridges” Sects. 1.3–1.5 and Sect. 1.7 by introducing the concept of ***transcendental deduction***. These deductions allow us to “transform” ***endurants*** into ***perdurants***: “passive” entities into “active” ones.The essence of Sects. 1.6–1.7 is to ***“translate”*** endurant parts into perdurant behaviours. Section 1.7 – although “only” half as long as the three sections on ***endurants*** – covers the analysis & description method for ***perdurants***. We shall model perdurants, notably ***behaviours***, in the form of CSP [23] Hence we introduce the CSP notions of ***channels*** and channel ***input/output***. Section 1.7 then “derives” the types of the behaviour arguments from the internal endurant qualities. Section 1.9 summarises the achievements and discusses open issues.

1.2 Entities: Endurants and Perdurants

1.2.1 A Generic Domain Ontology

Figure 1.1 on the next page shows a so-called “upper ontology”[2] for manifest domains[3] By ontologies we shall here understand *formal representations of a set of concepts within a domain and the relationships between those concepts*. Kai Sørlander’s Philosphy justifies our organising the ***entities*** of any describable domain, for example[4], as follows: There are ***describable*** phenomena and there are phenomena that we cannot describe. The former we shall call ***entities***. The ***entities*** are either ***endurants***, “still” entities – existing in ***space***, or ***perdurants***, “alive” entities – existing also in ***time***. ***Endurants*** are either ***discrete*** or

[2] An **ontology** encompasses a representation, formal naming, and definition of the categories, properties, and relations of the ... entities that substantiate one, many, or all domains. `https://en.wikipeda.org/wiki/Ontology_(information_science)`. An **upper ontology** (also known as a top-level ontology or foundation ontology) is an ontology which consists of very general terms (such as “entity”, “endurant”, “attribute”) that are common across all domains. `https://en.wikipedia.org/wiki/Upper_ontology`

[3] There are domains that are not ‘manifest’, bit not according to Defn. 1 on Page 1.

[4] We could organise the ontology differently: entities are either naturals, artifacts or living species, et cetera. If an upper node (•) satisfies a predicate $\mathscr{P}$ then all descendant nodes do likewise.

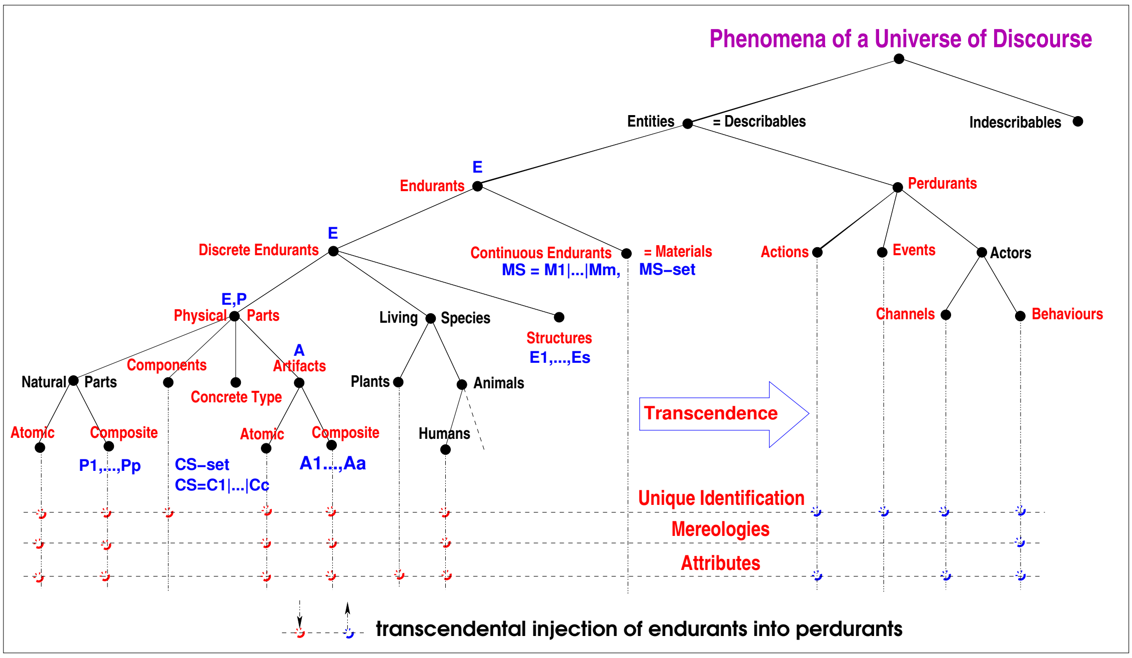


Fig. 1.1. An Upper Ontology for Domains

continuous – in which latter case we call them ***materials***[5]. ***Discrete*** endurants are ***physical parts***, ***living species***, or are ***structures***. ***Structures*** consist of one or more endurants. ***Physical parts*** are either ***naturals***, or ***artifacts***, i.e. man-made, or ***components***[6], or ***sets*** of identically ***typed parts***. ***Living Species*** are either ***plants*** or ***animals***. Among animals we have the ***humans***. ***Naturals*** and ***artifacts*** are either atomic or or composite – consisting of two or more differently typed parts. The categorisation into structures, natural parts, artifactual parts, plants, animals, and components is partly based in Kai Sørlander's Philosphy, partly pragmatic. The distinction between endurants and perdurants, are necessitated by Kai Sørlander's Philosphy as being in space, respectively in space **and** time; discrete and continuous are motivated by arguments of natural sciences; structures and components are purely pragmatic – as we shall later see; plants and animals, including humans, are necessitated by Kai Sørlander's Philosphy. The distinction between natural, physical parts, and artifacts is not necessary in Kai Sørlander's Philosphy, but, we claim, necessary, philosophically, in order to perform the ***intentional "pull"*** transcendental deduction.

Our reference, here, to Kai Sørlander's Philosphy, is very terse. We refer to a detailed research report: **A Philosophy of Domain Science & Engineering**, **http://www.imm.dtu.dk/˜dibj/2018/philosophy/filo.pdf**, for carefully reasoned arguments. That report is under continued revision: It reviews the **domain analysis & description** method; translates many of Sørlander's arguments and relates, in detail, the "options" of the **domain analysis & description** approach to Sørlander's Philosphy.

1.2.2 Universes of Discourse – Sect. 1.8.1 pp. 47

By a **universe of discourse** we shall understand the same as the **domain of interest**, that is, the ***domain*** to be ***analysed & described*** ■

Analysis Prompt 1 `is_universe_of_discourse:` *The domain analyser analyses "things" (θ) into either belonging to a* **universe of discourse** *or not. The method can thus be said to provide the* **domain analysis prompt***:*

[5] Please observe that ***materials*** were either ***natural*** or ***artifactual***, but that we do not "bother" in this paper. You may wish to slightly change the ontology diagram to reflect a distinction.

[6] Whether a discrete endurant as we shall soon see, is treated as a part or a component is a matter of pragmatics. Again cf. Footnote 5.

- *is_universe_of_discourse* – *where* is_universe_of_discourse(θ) *holds if* θ *is an element in the universe of discourse* ■[7]

Example 1.1. **Universes of Discourse**: We refer to a number of Internet accessible experimental reports[8] of descriptions of the following domains:

- **railways** [24, 25, 26],
- **container shipping** [27],
- **stock exchange** [28],
- **document systems** [29],
- **oil pipelines** [30],
- **"The Market"** [31],
- **Web systems** [32],
- **weather information** [33],
- **credit card systems** [34],
- **urban planning** [35],
- **swarms of drones** [36],
- et cetera, et cetera ■

It may be a **"large" domain**, that is, consist of many, as we shall see, ***endurants*** and ***perdurants,*** of many ***parts, components*** and ***materials,*** of many ***humans*** and ***artifacts,*** and of many***actors, actions, events*** and ***behaviours***.

Or it may be a **"small" domain**, that is, consist of a few such entities.

The choice of "boundaries", that is, of how much or little to include, and of how much or little to exclude is entirely the choice of the domain engineer cum scientist: the choice is crucial, and is not always obvious. The choice delineates an ***interface***, that is, that which is within the boundary, i.e., is in the domain, and that which is without, i.e., outside the domain, i.e., is the **context of the domain**, that is, the **external domain interfaces** ■ Experience helps set reasonable boundaries.

There are two "situations": Either a **domain analysis & description** endeavour is pursued in order to prepare for a subsequent development of ***requirements modeling***, in which case one tends to choose a **"narrow" domain**, that is, one that "fits", includes, but not much more, the domain of interest for the requirements ■ Or a **domain analysis & description** endeavour is pursued in order to research a domain. ***Either*** one that can form the basis for subsequent engineering studies aimed, eventually at requirements development; in this case "wider" boundaries may be sought. ***Or*** one that experimentally "throws a larger net", that is, seeks a "large" domain so as to explore interfaces between what is thought of as **internal system interfaces**.

Where, then, to start the ***domain analysis & description*** ? Either one can start "bottom-up", that is, with atomic entities: endurants or perdurants, one-by-one, and work one's way "out", to include composite entities, again endurants or perdurants, to finally reach some satisfaction: ***Eureka***, a goal has been reached. Or one can start "top-down", that is, "casting a wide net". The choice is yours. Our presentation, however, is "top down": most general domain aspects first.

1.2.3 Entities

Characterisation 1 Entity: By an **entity** we shall understand a **phenomenon**, i.e., something that can be ***observe***d, i.e., be seen or touched by humans, ***or*** that can be ***conceive***d as an ***abstraction*** of an entity; alternatively, a phenomenon is an entity, ***if it exists, it is* "being"*, it is that which makes a "thing" what it is: essence, essential nature*** [37, Vol. I, pg. 665] ■

Analysis Prompt 2 *is_entity: The domain analyser analyses "things" (θ) into entities or non-entities. The method can thus be said to provide the* **domain analysis prompt***:*

- *is_entity* – *where* is_entity(θ) *holds if* θ *is an entity*[9] ■

is_entity is said to be a ***prerequisite prompt*** for all other prompts.

The ***entities*** that we are concerned with are those with which Kai Sørlander's Philosphy is likewise concerned. They are the ones that are ***unavoidable*** in any any description of any possible world. And then,

[8] These are **draft** reports, more-or-less complete. The writing of these reports was finished when sufficient evidence, conforming or refuting one or another aspect of the **domain analysis & description method**.

[9] Analysis prompt definitions and description prompt definitions and schemes are delimited by ■

which are those entities ? In both [17] and [20] rationally deduces that these entities must be in ***space*** and ***time***, must satisfy laws of physics – like those of Newton and Einstein, but among them are also ***living species: plants*** and ***animals*** and hence ***humans***. The ***living species***, besides still being in ***space*** and ***time***, and satisfying laws of physics, must satisfy further properties – which we shall outline in Sect. 1.3.4 on Page 19.

1.2.4 Endurants and Perdurants

The concepts of endurants and perdurants are not present in, that is, are not essential to Sørlander's Philosophy. Since our departure point is that of ***computing science*** where, eventually, conventional computing processes data, that is: performs functions on data, we shall, however, introduce these two notion: ***endurant*** and ***perdurant***. The former, in a rough sense, "corresponds" to data; the latter, similarly, to processes.

Characterisation 2 Endurant: By an endurant we shall understand an entity that can be observed or conceived and described as a "complete thing" at no matter which given snapshot of time; alternatively an entity is endurant if it is capable of ***enduring***, that is ***persist***, ***"hold out"*** [37, Vol. I, pg. 656]. Were we to "freeze" time we would still be able to observe the entire endurant ■

Example 1.2. **Geography Endurants**: The geography of an area, like some island, or a country, consists of its geography – "the lay of the land", the geodetics of this land, the meteorology of it, et cetera.

Example 1.3. **Railway System Endurants**: Example railway system endurants are: a railway system, its net, its individual tracks, switch points, trains, their individual locomotives, et cetera.

Analysis Prompt 3 `is_endurant:` *The domain analyser analyses an entity, e, into an endurant as prompted by the* **domain analysis prompt***:*

- `is_endurant` – *ϕ is an endurant if* `is_endurant(e)` *holds.*

`is_entity` *is a* **prerequisite prompt** *for* `is_endurant` ■

Characterisation 3 Perdurant: By a perdurant we shall understand an entity for which only a fragment exists if we look at or touch them at any given snapshot in time, that is, were we to freeze time we would only see or touch a fragment of the perdurant, alternatively an entity is perdurant if it endures continuously, over time, persists, lasting [37, Vol. II, pg. 1552] ■

Example 1.4. **Geography Perdurants**: Example geography perdurants are: the continuous changing of the weather (meteorology); the erosion of coast lines; the rising of some land and the "sinking" of other land areas; volcano eruptions; earth quakes; et cetera.

Example 1.5. **Railway System Perdurants**: Example railway system perdurants are: the ride of a train from one railway station to another; and the stop of a train at a railway station from some arrival time to some departure time.

Analysis Prompt 4 `is_perdurant:` *The domain analyser analyses an entity e into perdurants as prompted by the* **domain analysis prompt***:*

- `is_perdurant` – *e is a perdurant if* `is_perdurant(e)` *holds.*

`is_entity` *is a* **prerequisite prompt** *for* `is_perdurant` ■

1.3 Endurants: Analysis of External Qualities

1.3.1 Discrete and Continuous Endurants

Characterisation 4 **Discrete Endurant:** By a **discrete endurant** we shall understand an endurant which is separate, individual or distinct in form or concept ■

The notion of ***discreteness*** is not extended to ***perdurants***.

Example 1.6. **Discrete Endurants**: The individual endurants of Example 1.3 on the preceding page were all discrete. Here are examples of discrete endurants of pipeline systems. A pipeline and its individual units: pipes, valves, pumps, forks, etc.

Analysis Prompt 5 *is_discrete: The domain analyser analyses endurants e into discrete entities as prompted by the* **domain analysis prompt***:*

- *is_discrete – e is discrete if is_discrete(e) holds* ■

Characterisation 5 **Continuous Endurant:** By a **continuous endurant** we shall understand an endurant which is prolonged, without interruption, in an unbroken series or pattern ■

We shall prefer to refer to continuous endurants as ***materials*** and otherwise cover materials in Sect. 1.3.6. The notion of a ***continuous endurant*** is not extended to ***perdurants***.

Example 1.7. **Materials**: Examples of materials are: water, oil, gas, compressed air, etc. A container, which we consider a discrete endurant, may contain a material, like a gas pipeline unit may contain gas.

Analysis Prompt 6 *is_continuous: The domain analyser analyses endurants e into continuous entities as prompted by the* **domain analysis prompt***:*

- *is_continuous – e is continuous if is_continuous(e) holds* ■

Continuity shall here not be understood in the sense of mathematics. Our definition of 'continuity' focused on ***prolonged, without interruption, in an unbroken series or pattern.*** In that sense materials shall be seen as 'continuous'.

The mathematical notion of 'continuity' is an abstract one.

The endurant notion of 'continuity' is physical one.

1.3.2 Discrete Endurants – Sect. 1.8.1 pp. 47

We analyse discrete endurants into ***physical parts***, ***living species*** and ***structures***.

Physical Parts

Characterisation 6 **Physical Parts:** By a ***physical part*** we shall understand a discrete endurant existing in time and subject to laws of physics, including the ***causality principle*** and ***gravitational pull***[10].

Analysis Prompt 7 *is_physical_part: The domain analyser analyses "things" (η) into physical part. The method can thus be said to provide the* **domain analysis prompt***:*

- *is_physical_part – where is_physical_part(η) holds if η is a physical part* ■

Section 1.3.3 continues our treatment of physical parts.

[10] This characterisation is the result of our study of relations between philosophy and computing science, notably influenced by Kai Sørlander's Philosphy. We refer to our research report [13, www.imm.dtu.dk/~dibj/2018/philosophy/-filo.pdf].

Living Species

Definition 18 **Living Species, I:** By a ***living species*** we shall understand a discrete endurant existing in time, subject to laws of physics, and additionally subject to ***causality of purpose***[11] Definition 24 on Page 19 elaborates.

Analysis Prompt 8 `is_living_species`: *The domain analyser analyses "things" (e) into living species. The method can thus be said to provide the* **domain analysis prompt**:

- `is_living_species` – *where* `is_living_species(e)` *holds if e is a living species* ■

Living species have a ***form*** they can ***develop*** to reach; they are ***causally*** determined to ***maintain*** this form; and they do so by ***exchanging matter*** with an ***environment***. We refer to [13] for details.
Section 1.3.4 continues our treatment of living species.

Structures – Sect. 1.8.1 pp. 47

Definition 19 **Structure:** By a **structure** we shall understand a discrete endurant which the domain engineer chooses to describe as consisting of one or more endurants, whether discrete or continuous, but to **not** endow with ***internal qualities***: unique identifiers, mereology or attributes ■

Structures are "conceptual endurants". A ***structure*** "gathers" one or more endurants under "one umbrella", often simplifying a presentation of some elements of a domain description. Sometimes, in our domain modelling, we choose to model an endurant as a ***structure***, sometimes as a ***physical part***; it all depends on what we wish to focus on in our domain model. As such structures are "compounds" where we are interested only in the (external and internal) qualities of the elements of the compound, but not in the qualities of the structure itself.

Example 1.8. **Structures**:As shown in the main example, Sect. 1.8, a model of transport is structured into a ***road net structure*** and an ***automobile structure***. The ***road net structure*** is then structured as a pair: a ***structure of hubs*** and a ***structure of links***. These latter structures are then modelled as set of hubs, respectively links. We could have modelled the road net ***structure*** as a ***composite part*** with ***unique identity, mereology*** and ***attributes*** which could then serve to model a `road net authority`. We could have modelled the automobile ***structure*** as a ***composite part*** with ***unique identity, mereology*** and ***attributes*** which could then serve to model a `department of vehicles` ■

The concept of ***structure*** is new. That is, it was not present in [2]. Whether to analyse & describe a discrete endurant into a structure or a physical part is a matter of choice. If we choose to analyse a discrete endurant into a ***physical part*** then it is because we are interested in endowing the part with ***qualities***, the unique identifiers, mereology and one or more attributes. If we choose to analyse a discrete endurant into a ***structure*** then it is because we are **not** interested in endowing the endurant with ***qualities***.

Analysis Prompt 9 `is_structure`: *The domain analyser analyse endurants, e, into structure entities as prompted by the* **domain analysis prompt**:

- `is_structure` ■

We shall now treat the external qualities of discrete endurants: ***physical parts*** (Sect. 1.3.3) and ***living species*** (Sect. 1.3.4). After that we cover ***components*** (Sect. 1.3.5), ***materials*** (Sect. 1.3.6) and ***artifacts*** (physical man-made parts, Sect. 1.3.3) . We remind the reader that in this section, i.e. Sect. 1.3, we cover only the ***analysis calculus*** for ***external qualities***; the ***description calculus*** for ***external qualities*** is treated in Sect. 1.4. The analysis and description calculi for internal qualities is covered in Sect. 1.5.

[11] See Footnote 10 on the preceding page.

1.3.3 Physical Parts – Sect. 1.8.1 pp. 47

Physical parts are either ***natural parts***, or ***components***, or ***sets of parts*** of the same type, or are ***artifacts*** i.e. man-made parts. The categorisation of physical parts into these four is pragmatic. ***Physical parts*** follow from Kai Sørlander's Philosphy. ***Natural parts*** are what Sørlander's Philosphy is initially about. ***Artifacts*** follow from ***humans*** acting according to their ***purpose*** in making "physical parts". ***Components*** is a simplification of natural and man-made parts. ***Set of parts*** is a simplification of composite natural and composite man-made parts as will be made clear in Sect. 1.4.2.

Natural Parts

Characterisation 7 **Natural Parts:** Natural parts are in ***space*** and ***time***; are subject to the ***laws of physics***, and also subject to the ***principle of causality*** and ***gravitational pull***.

The above is a factual characterisation of natural parts. The below is our definition – such as we shall model natural parts.

Definition 20 **Natural Part:** By a **natural part** we shall understand a ***physical part*** which the domain engineer chooses to endow with all three ***internal qualities***: unique identification, mereology, and one or more attributes ■

Artifacts

Characterisation 8 **Man-made Parts: Artifacts:** Artifacts are man-made either discrete or continuous endurants. In this section we shall only consider discrete endurants. Man-made continuous endurants are not treated separately but are "lumped" with [natural] materials. Artifacts are are in ***space*** and ***time***; are subject to the ***laws of physics***, and also subject to the ***principle of causality*** and ***gravitational pull***.

The above is a factual characterisation of discrete artifacts. The below is our definition – such as we shall model discrete artifacts.

Definition 21 **Artifact:** By an **artifact** we shall understand a ***man-made physical part*** which, like for ***natural parts***, the domain engineer chooses to endow with all three ***internal qualities***: unique identification, mereology, and one or more attributes ■

We shall assume, cf. Sect. 1.5.3 [***Attributes***], that ***artifacts*** all come with an ***attribute*** of kind ***intent***, that is, a set of purposes for which the artifact was constructed, and for which it is intended to serve. We continue our treatment of artifacts in Sect. 1.3.7 below.

Parts

Example 1.9. **Parts**: The examples of Example 1.2 on Page 15 are all natural parts, and of Example 1.3 on Page 15 are all artifacts ■

Except for the ***intent*** attribute of artifacts, we shall, in the following, treat ***natural*** and ***artifactual*** parts on par, i.e., just as physical parts.

Analysis Prompt 10 `is_part`*: The domain analyser analyse endurants, e, into part entities as prompted by the* **domain analysis prompt**:

- `is_part` *e is a part if* `is_part(e)` *holds* ■

Atomic and Composite Parts

A distinguishing quality of natural and artifactual parts is whether they are atomic or composite. Please note that we shall, in the following, examine the concept of parts in quite some detail. That is, parts become the domain endurants of main interest, whereas components, structures and materials become of secondary interest. This is a choice. The choice is based on pragmatics. It is still the domain analyser cum describers' choice whether to consider a discrete endurant a part or a component, or a structure. If the domain engineer wishes to investigate the details of a discrete endurant then the domain engineer choose to model[12] the discrete endurant as a part otherwise as a component.

Atomic Parts

Definition 22 **Atomic Part: Atomic part**s are those which, in a given context, are deemed to *not* consist of meaningful, separately observable proper *sub-part*s. A **sub-part** is a *part* ■

Analysis Prompt 11 *`is_atomic`: The domain analyser analyses a discrete endurant, i.e., a part p into an atomic endurant:*

- *`is_atomic`: p is an atomic endurant if `is_atomic(p)` holds* ■

Example 1.10. **Atomic Road Net Parts**: From one point of view all of the following can be considered atomic parts: hubs, links[13], and automobiles.

Composite Parts

Definition 23 **Composite Part: Composite part**s are those which, in a given context, are deemed to *indeed* consist of meaningful, separately observable proper *sub-part*s ■

Analysis Prompt 12 *`is_composite`: The domain analyser analyses a discrete endurant, i.e., a part p into a composite endurant:*

- *`is_composite`: p is a composite endurant if `is_composite(p)` holds* ■

`is_discrete` is a prerequisite prompt of both `is_atomic` and `is_composite`.

Example 1.11. **Composite Automobile Parts**: From another point of view all of the following can be considered composites parts: an automobile, consisting of, for example, the following composite parts: the engine train, the chassis the car body, the doors and the wheels. These can again be considered composite parts.

1.3.4 Living Species

We refer to Sect. 1.3.2 for our first characterisation (Page 17) of the concept of ***living species***[14]: a discrete endurant existing in time, subject to laws of physics, and additionally subject to ***causality of purpose***[15]

Definition 24 **Living Species, II:** Living species must have some ***form they can be developed to reach***; which they must be ***causally determined to maintain***. This ***development and maintenance*** must further in an ***exchange of matter with an environment***. It must be possible that living species occur in one of two forms: one form which is characterised by ***development, form and exchange***; another form which, **additionally**, can be characterised by the ***ability to purposeful movement***. The first we call **plants**, the second we call **animals** ■

[12] We use the term ***to model*** interchangeably with the composite term ***to analyse & describe***; similarly ***a model*** is used interchangeably with ***an analysis & description***.
[13] Hub ≡ street intersection; link ≡ street segments with no intervening hubs.
[14] See analysis prompt 8 on Page 17.
[15] See Footnote 10 on Page 16.

Analysis Prompt 13 *is_living_species: The domain analyser analyse discrete endurants, e, into living species entities as prompted by the* **domain analysis prompt***:*

- *is_living_species* ■

Plants

Example 1.12. **Plants**: Although we have not yet come across domains for which the need to model the living species of plants were needed, we give some examples anyway: grass, tulip, rhododendron, oak tree.

Analysis Prompt 14 *is_plant: The domain analyser analyses "things" (ℓ) into a plant. The method can thus be said to provide the* **domain analysis prompt***:*

- *is_plant – where* `is_plant(`ℓ`)` *holds if ℓ is a plant* ■

The predicate `is_living_species(`ℓ`)` is a prerequisite for `is_plant(`ℓ`)`.

Animals

Definition 25 Animal: We refer to the initial definition of ***living species*** above – while ephasizing the following traits: (i) ***form animals can be developed to reach***; (ii) ***causally determined to maintain***. (iii) ***development and maintenance*** in an ***exchange of matter with an environment***, and (iv) ***ability to purposeful movement.***

Analysis Prompt 15 *is_animal: The domain analyser analyses "things" (ℓ) into an animal. The method can thus be said to provide the* **domain analysis prompt***:*

- *is_animal – where* `is_animal(`ℓ`)` *holds if ℓ is an animal* ■

The predicate `is_living_species(`ℓ`)` is a prerequisite for `is_animal(`ℓ`)`.

Example 1.13. **Animals**: Although we have not yet come across domains for which the need to model the living species of animals, in general, were needed, we give some examples anyway: dolphin, goose cow dog, lion, fly.

We have not decided, for this paper, whether to model animals singly or as sets[16] of such.

Humans

Definition 26 Human: A ***human*** (a ***person***) is an ***animal***, cf. Definition 25, with the additional properties of having ***language***, being ***conscious*** of ***having knowledge*** (of its own situation), and ***responsibility***.

Analysis Prompt 16 *is_human: The domain analyser analyses "things" (ℓ) into a human. The method can thus be said to provide the* **domain analysis prompt***:*

- *is_human – where* `is_human(`ℓ`)` *holds if ℓ is a human* ■

The predicate `is_animal(`ℓ`)` is a prerequisite for `is_human(`ℓ`)`.

We refer to [13, Sects. 10.4–10.5] for a specific treatment of living species, animals and humans, and to [13] in general for the philosophy background for rationalising the treatment of living species, animals and humans.

We have not, in our many experimental domain modelling efforts had occasion to model humans; or rather: we have modelled, for example, automobiles as possessing human qualities, i.e., "subsuming humans". We have found, in these experimental domain modelling efforts that we often confer anthropomorphic qualities on artifacts[17], that is, that these artifacts have human characeristics. You, the reader are reminded that when some programmers try to explain their programs they do so using such phrases as ***and here the program does ...*** so-and-so !

[16] school of dolphins, flock of geese, herd of cattle, pack of dogs, pride of lions, swarm of flies,
[17] Cf. Sect. 1.3.7 below.

1.3.5 Components

Definition 27 **Component:** By a **component** we shall understand a discrete endurant which we, the domain analyser cum describer chooses to **not** endow with ***mereology*** ■

Components are discrete endurants. Usually they come in sets. That is, sets of sets of components of different sorts (cf. Sect. 1.4.4 on Page 25). A discrete endurant can (itself) "be" a set of components. But physical parts may contain (`has_components`) components: natural parts may contain natural components, artifacts may contain natural and artifactual components. We leave it to the reader to provide analysis predicates for natural and artifactual "componentry".

Example 1.14. **Components**: A natural part, say a land area may contain gravel pits of sand, clay pits tar pits and other "pits". An artifact, say a postal letter box may contain letters, small parcels, newspapers and advertisement brochures.

Analysis Prompt 17 `has_components`: *The domain analyser analyses discrete endurants e into component entities as prompted by the* **domain analysis prompt**:

- `has_components` ■

We refer to Sect. 1.4.4 on Page 25 for further treatment of the concept of ***components***.

1.3.6 Continuous Endurants ≡ Materials

Definition 28 **Material:** By a **material** we shall understand a continuous endurant ■
Materials are continuous endurants. Usually they come in sets. That is, sets of of materials of different sorts (cf. Sect. 1.4.5 on Page 25). So an endurant can (itself) "be" a set of materials. But physical parts may contain (`has_materials`) materials: natural parts may contain natural materials, artifacts may contain natural and artifactual materials. We leave it to the reader to provide analysis predicates for natural and artifactual "materials".

Example 1.15. **Natural and Man-made Materials**: A **natural part**, say ***a land area***, may contain lakes, rivers, irrigation dams and border seas. An **artifact**, say ***an automobile***, usually contains gasoline, lubrication oil, engine cooler liquid and window screen washer water.

Analysis Prompt 18 `has_materials`: *The* **domain analysis prompt**:

- `has_materials`(p)

yields **true** *if part p:P potentially may contain materials otherwise false* ■

We refer to Sect. 1.4.5 on Page 25 for further treatment of the concept of ***materials***. We shall define the terms unique identification, mereology and attributes in Sects. 1.5.1–1.5.3.

1.3.7 Artifacts

Definition 29 **Artifacts:** By artifacts we shall understand a man-made physical part or a man-made material ■

Example 1.16. **More Artifacts**: We have already, in Example 1.9 on Page 18, referred to some examples of artifacts. Here are some more: ship, container vessels, container, container stack, container terminal port, harbour.

Analysis Prompt 19 `is_artifact`: *The domain analyser analyses "things" (p) into artifacts. The method can thus be said to provide the* **domain analysis prompt**:

- `is_artifact` – *where* `is_artifact(p)` *holds if p is an artifact* ■

1.3.8 States – Sect. 1.8.1 pp. 48

Definition 30 State: By a ***state*** we shall understand any number of physical parts or materials.

Example 1.17. **Artifactual States**: The following endurants are examples of states (including being elements of state compounds): pipe units (pipes, valves, pumps, etc.) of pipe-lines; hubs and links of road nets (i.e., street intersections and street segments); automobiles (of transport systems).

The notion of ***state*** becomes relevant in Sect. 1.7.

1.4 Endurants: The Description Calculus

1.4.1 Parts: Natural or Man-made

The observer functions of this section applies to both natural parts and man-made parts (i.e., artifacts).

On Discovering Endurant Sorts

Our aim now is to present the basic principles that let the domain analyser decide on ***part sort***s. We observe parts one-by-one.

> ***(α) Our analysis of parts concludes when we have "lifted" our examination of a particular part instance to the conclusion that it is of a given sort, that is, reflects a formal concept.***

Thus there is, in this analysis, a "eureka", a step where we shift focus from the concrete to the abstract, from observing specific part instances to postulating a sort: from one to the many

Analysis Prompt 20 *observe_endurant: The* **domain analysis prompt***:*

- *observe_endurants*

directs the domain analyser to observe the sub-endurants of an endurant e and to suggest their sorts. Let *observe_endurants(e)* $= \{e_1{:}E_1, e_2{:}E_2, \ldots, e_m{:}E_m\}$ ■

> ***(β) The analyser analyses, for each of these endurants, e_i, which formal concept, i.e., sort, it belongs to; let us say that it is of sort E_k; thus the sub-parts of p are of sorts $\{E_1,E_2,\ldots,E_m\}$. Some E_k may be natural parts, other artifacts (man-made parts) or structures, and yet others may be components or materials. And parts may be either atomic or composite.***

The domain analyser continues to examine a finite number of other composite parts: $\{p_j, p_\ell, \ldots, p_n\}$. It is then "discovered", that is, decided, that they all consists of the same number of sub-parts $\{e_{i_1},e_{i_2},\ldots,e_{i_m}\}$, $\{e_{j_1},e_{j_2},\ldots,e_{j_m}\}$, $\{e_{\ell_1},e_{\ell_2},\ldots,e_{\ell_m}\}$, ..., $\{e_{n_1},e_{n_2},\ldots,e_{n_m}\}$, of the same, respective, endurant sorts.

> ***(γ) It is therefore concluded, that is, decided, that $\{e_i, e_j, e_\ell, \ldots, e_n\}$ are all of the same endurant sort P with observable part sub-sorts $\{E_1, E_2, \ldots, E_m\}$.***

Above we have ***type-font-highlighted*** three sentences: (α, β, γ). When you analyse what they "prescribe" you will see that they entail a "depth-first search" for part sorts. The β sentence says it rather directly: ***"The analyser analyses, for each of these parts, p_k, which formal concept, i.e., part sort it belongs to."*** To do this analysis in a proper way, the analyser must ("recursively") analyse structures into sub-structures, parts, components and materials, and parts "down" to their atomicity. Components and materials are considered "atomic", i.e., to not contain further analysable endurants. For the structures, parts (whether natural or man-made), components and materials of the structure the analyser cum describer decides on their sort, and work ("recurse") their way "back", through possibly intermediate endurants, to the p_ks. Of course, when the analyser starts by examining atomic parts, components and materials, then their endurant structure and part analysis "recursion" is not necessary.

Endurant Sort Observer Functions

The above analysis amounts to the analyser first "applying" the ***domain analysis*** prompt is_composite(e) to a discrete endurant, e, where we now assume that the obtained truth value is **true**. Let us assume that endurants e:E consist of sub-endurants of sorts $\{E_1,E_2,\ldots,E_m\}$. Since we cannot automatically guarantee that our domain descriptions secure that E and each E_i ($1 \leq i \leq$m) denotes disjoint sets of entities we must prove it.

Domain Description Prompt 1 *observe_endurant_sorts: If is_composite(p) holds, then the analyser "applies" the* **domain description prompt**

- *observe_endurant_sorts(p)*

resulting in the analyser writing down the **endurant sorts and endurant sort observers** *domain description text according to the following schema:*

1. observe_endurant_sorts schema

Narration:
[s] ... narrative text on sorts ...
[o] ... narrative text on sort observers ...
[η] ... narrative text on sort type observers ...
[i] ... narrative text on sort recognisers ...
[p] ... narrative text on proof obligations ...
Formalisation:
type
[s] E,
[s] E_i i:[1..m] **comment:** E_i i:[1..m] abbreviates E_1, E_2, ..., E_m
value
[o] **obs_endurant_sorts**_E_i: E $\to$ E_i i:[1..m]
[η] **if** is_part(e_i): η(e_i) $\equiv$ ⦃ E_i ⦄ i:[1..m]
[i] **is**_E_i: $(E_1|E_2|...|E_m) \to$ **Bool** i[1..m]
proof obligation [Disjointness of endurant sorts]
[p] $\mathscr{PO}$: $\forall$ e:(E_1|E_2|...|E_m) • $\bigwedge$ {**is**_E_i(e) $\equiv$ $\bigwedge$ {∼**is**_E_j(p)|j:[1..m] \ {i}}|i:[1..m]}

is_composite is a **prerequisite prompt** *of observe_endurant_sorts. That is, the composite may satisfy is_natural or is_artifact* ■

We do not here state guidelines for discharging proof obligations.

1.4.2 Concrete Part Types

Analysis Prompt 21 *has_concrete_type: The domain analyser may decide that it is expedient, i.e., pragmatically sound, to render a part sort, P, whether atomic or composite, as a concrete type, T. That decision is prompted by the holding of the* **domain analysis prompt***:*

- *has_concrete_type.*

is_discrete is a ***prerequisite prompt*** *of has_concrete_type* ■

The reader is reminded that the decision as to whether an abstract type is (also) to be described concretely is entirely at the discretion of the domain engineer.

Domain Description Prompt 2 *observe_part_type: Then the domain analyser applies the* **domain description prompt***:*

- *observe_part_type(p)*[18]

[18] has_concrete_type is a ***prerequisite prompt*** of observe_part_type.

to parts p:P which then yield the **part type and part type observers** *domain description text according to the following schema:*

2. observe_part_type schema

Narration:
[t_1] ... narrative text on sorts and types S_i ...
[t_2] ... narrative text on types T ...
[t_3] ... narrative text on type of value observer
[o] ... narrative text on type observers ...

Formalisation:
type
[t_1] S_1, S_2, ..., S_m, ..., S_n,
[t_2] $T = \mathscr{E}(S_1,S_2,...,S_n)$
[t_3] $\eta(s_i) \equiv \lll S \ggg$, i:[1..n],s_i:S_i
value
[o] **obs_part_**T: P → T ■

Here S_1,S_2,...,S_m,...,S_n may be any types, including part sorts, where $0{\le}m{\le}n{\ge}1$, where m is the number of new (atomic or composite) sorts, and where $n-m$ is the number of concrete types (like **Bool, Int, Nat**) or sorts already analysed & described. and $\mathscr{E}(S_1,S_2,...,S_n)$ is a type expression Usually it is wise to restrict the part type definitions, $T_i = \mathscr{E}_i(Q,R,...,S)$, to simple type expressions.[19] The type name, T, of the concrete type, as well as those of the auxiliary types, S_1,S_2,...,S_m, are chosen by the domain describer: they may have already been chosen for other sort–to–type descriptions, or they may be new.

1.4.3 On Endurant Sorts

Derivation Chains

Let E be a composite sort. Let E_1, E_2, ..., E_m be the part sorts "discovered" by means of `observe_endurant_sorts(e)` where e:E. We say that E_1, E_2, ..., E_m are (immediately) **derived** from E. If E_k is derived from E_j and E_j is derived from E_i, then, by transitivity, E_k is **derived** from E_i.

No Recursive Derivations

We "mandate" that if E_k is derived from E_j then there E_j is different from E_k and there can be no E_k derived from E_j, that is, E_k cannot be derived from E_k. That is, we do not "provide for" recursive domain sorts. It is not a question, actually of allowing recursive domain sorts. It is, we claim to have observed, in very many ***analysis & description*** experiments, that there are no recursive domain sorts ![20]

Names of Part Sorts and Types

The **domain analysis & description** text prompts `observe_endurant_sorts`, as well as the below-defined `observe_part_type`, `observe_component_sorts` and `observe_material_sorts`, – as well as the further below defined `attribute_names`, `observe_material_sorts`, `observe_unique_identifier`, `observe_mereology` and `observe_attributes` prompts introduced below – "yield" type names.

[19] T=A-**set** or T=A* or T=ID $\rightarrow\!\!\!\!\!_{m}$ A or T=A_t|B_t|...|C_t where ID is a sort of unique identifiers, T=A_t|B_t|...|C_t defines the disjoint types A_t==mkA_t(s:A_s), B_t==mkB_t(s:B_s), ..., C_t==mkC_t(s:C_s), and where A, A_s, B_s, ..., C_s are sorts. Instead of A_t==mkA_t(a:A_s), etc., we may write A_t::A_s etc.

[20] Some readers may object, but we insist ! If ***trees*** are brought forward as an example of a recursively definable domain, then we argue: Yes, trees can be recursively defined, but it is not recursive. Trees can, as well, be defined as a variant of graphs, and you wouldn't claim, would you, that graphs are recursive ?

That is, it is as if there is a reservoir of an indefinite-size set of such names from which these names are "pulled", and once obtained are never "pulled" again. There may be domains for which two distinct part sorts may be composed from identical part sorts. *In this case the domain analyser indicates so by prescribing a part sort already introduced.*

1.4.4 Components

We refer to Sect. 1.3.5 on Page 21 for our initial treatment of 'components'.

Domain Description Prompt 3 *observe_component_sorts*: *The* **domain description prompt**:

- *observe_component_sorts(p)*

yields the **component sorts and component sort observer** *domain description text according to the following schema – whether or not the actual part p contains any components:*

2. observe_component_sorts schema

Narration:

[s] ... narrative text on component sorts ...
[o] ... narrative text on component observers ...
[i] ... narrative text on component sort recognisers ...
[u] ... narrative text on unique identifier ...
[p] ... narrative text on component sort proof obligations ...

Formalisation:

type
[s] K1, K2, ..., Kn
[s] K = K1| K2 | ... | Kn
[s] KS = K-**set**
value
[o] **obs_components_**P: P → KS
[i] **is_**K_i: (K_1|K_2|...|K_n) → **Bool** i:[1..n]
[u] **uid_**K_i

Proof Obligation: [Disjointness of Component Sorts]

[p] $\mathscr{PO}$: $\forall\, k_i$:(K_1|K_2|...|K_n) • $\bigwedge$ {**is_**$K_i(k_i) \equiv \bigwedge\{\sim$**is_**$K_j(k_j)$|j:[1..n] \ {i}}} i:[1..n] ■

We have presented one way of tackling the issue of describing components. There are other ways. We leave those 'other ways' to the reader. We are not going to suggest techniques and tools for analysing, let alone ascribing qualities to components. We suggest that conventional abstract modeling techniques and tools be applied.

1.4.5 Materials

We refer to Sect. 1.3.6 on Page 21 for our initial treatment of 'materials'. Continuous endurants (i.e., **material**s) are entities, *m*, which satisfy:

- is_material(*e*) ≡ is_continuous(*e*)

If is_material(*e*) holds then we can apply the **domain description prompt**: observe_material_sorts(*e*).

Domain Description Prompt 4 *observe_material_sorts*: *The* **domain description prompt**:

- *observe_material_sorts(e)*

*yields the **material sorts and material sort observers'** domain description text according to the following schema whether or not part p actually contains materials:*

2. observe_material_sorts schema

Narration:
[s] ... narrative text on material sorts ...
[o] ... narrative text on material sort observers ...
[i] ... narrative text on material sort recognisers ...
[p] ... narrative text on material sort proof obligations ...

Formalisation:
type
[s] M1, M2, ..., Mn
[s] M = M1 | M2 | ... | Mn
[s] MS = M-**set**
[a] Ai = A11 | A12 | ... | A1n
value
[o] **obs_mat_sort_M**$_i$: P → M, [i:1..n]
[o] **obs_materials_P**: P → MS
[i] **is_M**$_i$: M → **Bool** [i:1..n]
[a] **attr_A**$_{i_j}$: M$_i$ → A$_{i_j}$ [i:...,j:...]
proof obligation [Disjointness of Material Sorts]
[p] $\mathscr{PO}$: ∀ m_i:M • ∧ {**is_M**$_i(m_i)$ ≡ ∧{∼**is_M**$_j(m_j)$|j ∈ {1..m} \ {i}}|i:[1..n]}

Let us assume that parts p:P embody materials of sorts $\{M_1,M_2,\ldots,M_n\}$. Since we cannot automatically guarantee that our domain descriptions secure that each M_i ([$1 \leq i \leq n$]) denotes disjoint sets of entities we must prove it ■

1.5 Endurants: Analysis & Description of Internal Qualities

We remind the reader that internal qualities cover ***unique Identifiers*** (Sect. 1.5.1), ***mereology*** (Sect. 1.5.2) and ***attributes*** (Sect. 1.5.3).

1.5.1 Unique Identifiers – Sect. 1.8.1 pp. 48

We introduce a notion of unique identification of parts and components. We assume (i) that all parts and components, p, of any domain P, have ***unique identifier***s, (ii) that ***unique identifier***s (of parts and components p:P) are ***abstract value***s (of the ***unique identifier*** sort PI of parts p:P), (iii) such that distinct part or component sorts, P$_i$ and P$_j$, have distinctly named ***unique identifier*** sorts, say PI$_i$ and PI$_j$, (iv) that all π_i:PI$_i$ and π_j:PI$_j$ are distinct, and (v) that the observer function **uid_P** applied to p yields the unique identifier, say π:PI, of p. The description language function **type_name** applies to unique identifiers, p_ui:P_UI, and yield the name of the type, P, of the parts having unique identifiers of type P_UI.

Representation of Unique Identifiers: Unique identifiers are abstractions. When we endow two parts (say of the same sort) with distinct unique identifiers then we are simply saying that these two parts are distinct. We are not assuming anything about how these identifiers otherwise come about.

Domain Description Prompt 5 *observe_unique_identifier*: *We can therefore apply the* **domain description prompt***:*

- *observe_unique_identifier*

to parts p:P resulting in the analyser writing down the ***unique identifier type and observer*** *domain description text according to the following schema:*

2. `observe_unique_identifier` schema

Narration:
[s] ... narrative text on unique identifier sort PI ...
[u] ... narrative text on unique identifier observer **uid_**P ...
[η] ... narrative text on type name, an RSL$^+$Text observer ...
[a] ... axiom on uniqueness of unique identifiers ...
Formalisation:
type
[s] PI
value
[u] **uid_**P: P → PI
[u] η PI → ≪ P ≫
axiom [Disjointness of Domain Identifier Types]
[a] $\mathscr{A}$: $\mathscr{U}$(PI,PI_i,PI_j,...,PI_k) ■

We ascribe, in principle, unique identifiers to all parts whether natural or artifactual, and to all components. We find, from our many experiments, cf. Example 1.1 on Page 14, that we really focus on those domain entities which are artifactual endurants and their behavioural "counterparts".

1.5.2 Mereology – Sect. 1.8.1 pp. 49

Mereology is the study and knowledge of parts and part relations. Mereology, as a logical/philosophical discipline, can perhaps best be attributed to the Polish mathematician/logician Stanisław Leśniewski [38, 39].

Part Relations

Which are the relations that can be relevant for part-hood ? We give some examples. (i) Two otherwise distinct parts may ***"share"*** values.[21] By ***'sharing'*** values we shall, as a generic example, mean that two parts of different sorts has the same attributes but that one ***'defines'*** the attribute, like, for example ***'programming'*** its values, cf. Defn.8 Page31, whereas the other ***'uses'*** these values, like, for example considering them ***'inert'***, cf. Defn.3 Page31. (ii) Two otherwise distinct parts may be said to, for example, be topologically "adjacent" or one "embedded" within the other. These examples are in no way indicative of the "space" of part relations that may be relevant for part-hood. The domain analyser is expected to do a bit of experimental research in order to discover necessary, sufficient and pleasing "mereology-hoods" !

Part Mereology: Types and Functions

Analysis Prompt 22 *`has_mereology`: To discover necessary, sufficient and pleasing "mereology-hoods" the analyser can be said to endow a truth value,* **true**, *to the* **domain analysis prompt***:*

- *`has_mereology`*

When the domain analyser decides that some parts are related in a specifically enunciated mereology, the analyser has to decide on suitable ***mereology type***s and ***mereology observer***s (i.e., part relations).

1 We define a **mereology type** of a part *p*:*P* as a triplet type expression over set of unique [part] identifiers.

[21] For the concept of attribute value see Sect. 1.5.3 on Page 29.

2 There is the identification of all those part types $P_{i_1}, P_{i_2}, ..., P_{i_m}$ where at least one of whose properties "is_of_interest" to parts p:P.
3 There is the identification of all those part types $P_{io_1}, P_{io_2}, ..., P_{io_n}$ where at least one of whose properties "is_of_interest" to parts p:P and vice-versa.
4 There is the identification of all those part types $P_{o_1}, P_{o_2}, ..., P_{o_o}$ for whom properties of p:P "is_of_interest" to parts of types $P_{o_1}, P_{o_2}, ..., P_{o_o}$.
5 The the mereology triplet sets of unique identifiers are disjoint and are all unique identifiers of the universe of discourse.

The three part mereology is just a suggestion. As it is formulated here we mean the three 'sets' to be disjoint. Other forms of expressing a mereology should be considered for the particular domain and for the particular parts of that domain. We leave out further characterisation of the seemingly vague notion "is_of_interest". It is exemplified in Sect. 1.8.1 Pg. 49.

```
type
2   iPI  = iPI1 | iPI2 | ... | iPIm
3   ioPI = ioPI1 | ioPI2 | ... | ioPIn
4   oPI  = oPI1 | oPI2 | ... | oPIo
1   MT = iPI-set × ioPI-set × oPI-set
axiom
5   ∀ (iset,ioset,oset):MT •
5     card iset + card ioset + card oset = card ∪{iset,ioset,oset}
5     ∪{iset,ioset,oset} ⊆ unique_identifiers(uod)
value
5   unique_identifiers: P → UI-set
5   unique_identifiers(p) ≡ ...
```

Domain Description Prompt 6 observe_mereology: *If* has_mereology(p) *holds for parts p of type P, then the analyser can apply the* **domain description prompt**:

- observe_mereology

to parts of that type and write down the **mereology types and observer** *domain description text according to the following schema:*

2. observe_mereology schema

Narration:
[t] ... narrative text on mereology type ...
[m] ... narrative text on mereology observer ...
[a] ... narrative text on mereology type constraints ...
Formalisation:

```
type
[t]   MT[22]
value
[m]   obs_mereo_P: P → MT
axiom [Well−formedness of Domain Mereologies]
[a]   𝒜: 𝒜(MT)
```

𝒜(MT) is a predicate over possibly all unique identifier types of the domain description. To write down the concrete type definition for MT requires a bit of analysis and thinking. has_mereology *is a* **prerequisite prompt** *for* observe_mereology ■

[22] The mereology descriptor, MT will be referred to in the sequel.

Formulation of Mereologies

The `observe_mereology` domain descriptor, Page 28, may give the impression that the mereo type MT can be described "at the point of issue" of the `observe_mereology` prompt. Since the MT type expression may, in general, depend on any part sort the mereo type MT can, for some domains, "first" be described when all part sorts have been dealt with. In [40] we we present a model of one form of evaluation of the `TripTych` analysis and description prompts.

Some Modelling Observations

It is, in principle, possible to find examples of mereologies of natural parts: rivers: their confluence, lakes and oceans; and geography: mountain ranges, flat lands, etc. But in our experimental case studies cf. Example 1.1 on Page 14, we have found no really interesting such cases. All our experimental case studies appears to focus on the mereology of artifacts. And, finally, in modelling humans, we find that their mereology encompass all other humans and all artifactsHumans cannot be tamed to refrain from interacting with everyone and everything.

1.5.3 Attributes – Sect. 1.8.1 pp. 49

To recall: there are three sets of **internal qualities**: unique part identifiers, part mereology and attributes. Unique part identifiers and part mereology are rather definite kinds of internal endurant qualities. Part attributes form more "free-wheeling" sets of **internal qualities**.

Technical Issues

We divide Sect. 1.5.3 into two subsections: ***technical issues***, the present one, and ***modelling issues***, Sect. 1.5.3.

Inseparability of Attributes from Parts and Materials:

Parts and materials are typically recognised because of their spatial form and are otherwise characterised by their intangible, but measurable attributes. That is, whereas endurants, whether discrete (as are parts and components) or continuous (as are materials), are physical, tangible, in the sense of being spatial [or being abstractions, i.e., concepts, of spatial endurants], attributes are intangible: cannot normally be touched[23], or seen[24], but can be objectively measured[25]. Thus, in our quest for describing domains where humans play an active rôle, we rule out subjective "attributes": feelings, sentiments, moods. Thus we shall abstain, in our domain science also from matters of aesthetics. We equate all endurants which, besides possible type of unique identifiers (i.e., excepting materials) and possible type of mereologies (i.e.,, excepting components and materials), have the same types of attributes, with one sort. Thus removing a quality from an endurant makes no sense: the endurant of that type either becomes an endurant of another type or ceases to exist (i.e., becomes a non-entity) !

Attribute Quality and Attribute Value:

We distinguish between an attribute (as a logical proposition, of a name, i.e.) type, and an attribute value, as a value in some value space.

Analysis Prompt 23 *`attribute types`: One can calculate the set of attribute types of parts and materials with the following* **domain analysis prompt***:*

- *`attribute_types`*

Thus for a part p we may have $\mathtt{attribute\_types}(p) = \{A_1,A_2,...,A_m\}$.

[23] One can see the red colour of a wall, but one touches the wall.

[24] One cannot see electric current, and one may touch an electric wire, but only if it conducts high voltage can one know that it is indeed an electric wire.

[25] That is, we restrict our domain analysis with respect to attributes to such quantities which are observable, say by mechanical, electrical or chemical instruments. Once objective measurements can be made of human feelings, beauty, and other, we may wish to include these "attributes" in our domain descriptions.

Attribute Types and Functions:

Let us recall that attributes cover qualities other than unique identifiers and mereology. Let us then consider that parts and materials have one or more attributes. These attributes are qualities which help characterise "what it means" to be a part or a material. Note that we expect every part and material to have at least one attribute. The question is now, in general, how many and, particularly, which.

Domain Description Prompt 7 *observe_attributes*: *The domain analyser experiments, thinks and reflects about part attributes. That process is initiated by the* **domain description prompt***:*

- *observe_attributes*.

The result of that **domain description prompt** *is that the domain analyser cum describer writes down the* ***attribute (sorts or) types and observers*** *domain description text according to the following schema:*

2. observe_attributes **schema**

Narration:
[t] ... narrative text on attribute sorts ...
[o] ... narrative text on attribute sort observers ...
[v] ... narrative text on set **of** attribute **value** observers ...
[i] ... narrative text on attribute sort recognisers ...
[p] ... narrative text on attribute sort proof obligations ...

Formalisation:
type
[t] A_i $[1 \leq i \leq n]$
value
[o] **attr**_A_i:P→A_i i:[1..n]
[v] **obs_attrib_values**_P(p) ≡ { **attr**_A_1(p),**attr**_A_2(p),...,**attr**_A_n(p) }
[i] **is**_A_i:(A_1|A_2|...|A_n)→**Bool** i:[1..n]
proof obligation [Disjointness of Attribute Types]
[p] $\mathscr{PO}$: **let** P be any part sort **in** [the domain description]
[p] **let** a:(A_1|A_2|...|A_n) **in** **is**_A_i(a) ≠ **is**_A_j(a) **end end** [i≠i, i,j:[1..n]]

The **type** (or rather sort) definitions: A_1, A_2, ..., A_n, inform us that the domain analyser has decided to focus on the distinctly named A_1, A_2, ..., A_n attributes.[26] And the **value** clauses **attr**_A_1:P→A_1, **attr**_A_2:P→A_2, ..., **attr**_A_n:P→A_n are then "automatically" given: if a part, p:P, has an attribute A_i then there is postulated, "by definition" [eureka] an attribute observer function **attr**_A_i:P→A_i etcetera ■

We cannot automatically, that is, syntactically, guarantee that our domain descriptions secure that the various attribute types for an emerging part sort denote disjoint sets of values. Therefore we must prove it.

Attribute Categories:

Michael A. Jackson [41] has suggested a hierarchy of attribute categories: static or dynamic values – and within the dynamic value category: inert values or reactive values or active values – and within the dynamic active value category: autonomous values or biddable values or programmable values. We now review these attribute value types. The review is based on [41, M.A. Jackson]. ***Part attributes*** are either constant or varying, i.e., **static** or **dynamic** attributes.

Attribute Category: 1 By a **static attribute**, a:A, is_static_attribute(a), we shall understand an attribute whose values are constants, i.e., cannot change.

[26] The attribute type names are not like type names of, for example, a programming language. Instead they are chosen by the domain analyser to reflect on domain phenomena.

Attribute Category: 2 By a **dynamic attribute**, a:A, `is_dynamic_attribute`(a), we shall understand an attribute whose values are variable, i.e., can change. Dynamic attributes are either ***inert, reactive*** or ***active*** attributes.

Attribute Category: 3 By an **inert attribute**, a:A, `is_inert_attribute`(a), we shall understand a dynamic attribute whose values only change as the result of external stimuli where these stimuli prescribe new values.

Attribute Category: 4 By a **reactive attribute**, a:A, `is_reactive_attribute`(a), we shall understand dynamic attributes whose value, if they vary, change in response to external stimuli, where these stimuli come from outside the domain of interest.

Attribute Category: 5 By an **active attribute**, a:A, `is_active_attribute`(a), we shall understand a dynamic attribute whose values change (also) of its own volition. Active attributes are either ***autonomous, biddable*** or ***programmable*** attributes.

Attribute Category: 6 By an **autonomous attribute**, a:A, `is_autonomous_attribute`(a), we shall understand a dynamic active attribute whose values change value only "on their own volition". The values of an autonomous attributes are a "law onto themselves and their surroundings".

Attribute Category: 7 By a **biddable attribute**, a:A, `is_biddable_attribute`(a) we shall understand a dynamic active attribute whose values ***are prescribed but may fail to be observed as such.***

Attribute Category: 8 By a **programmable attribute**, a:A, `is_programmable_attribute`(a), we shall understand a dynamic active attribute whose values can be prescribed.

Figure 1.2 captures an attribute value ontology.

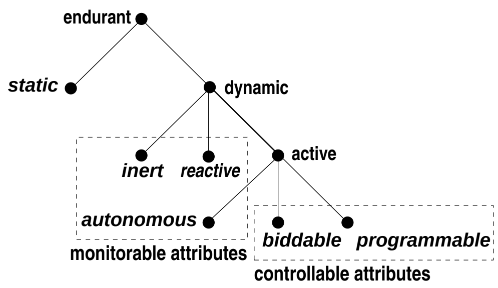


Fig. 1.2. Attribute Value Ontology

Calculating Attributes:

6 Given a part p we can calculate its static attributes.
7 Given a part p we can calculate its controllable, i.e., the biddable and programmable attributes.
8 And given a part p we can calculate its monitor-able attributes, i.e., the inert, reactive and autonomous attributes.
9 These three sets make up all the attributes of part p.

value
6 stat_attr_typs: P → ≪ SA1×SA2×...×SAs ≫
7 ctrl_attr_typs: P → ≪ CA1×CA2×...×CAc ≫
8 mon_attr_typs: P → ≪ MA1×MA2×...×MAm ≫
axiom
9 ∀ p:P •

9 **let** ≪ SA1×SA2×...×SAs ≫ = stat_attr_typs(p),
9 ≪ CA1×CA2×...×CAc ≫ = ctrl_attr_typs(p),
9 ≪ MA1×MA2×...×MAm ≫ = mon_attr_typs(p) **in**
9 **card**{SA1,SA2,...,SAs}+**card**{CA1,CA2,...,CAc}+**card**{MA1,MA2,...,MAm}
9 = **card**{SA1,SA2,...,SAs,CA1,CA2,...,CAc,MA1,MA2,...,MAm} **end**

10 Given a part *p* we can calculate its static attribute values.
11 Given a part *p* we can calculate its controllable, i.e., the biddable and programmable attribute values.

value
10 stat_attr_vals: P → SA1×SA2×...×SAs
10 stat_attr_vals(p) ≡
10 **let** ≪ SA1×SA2×...×SAs ≫ = stat_attr_typs(p) **in**
10 (attr_SA1(p),attr_SA2(p),...,attr_SAs(p)) **end**

11 ctrl_attr_vals: P → CA1×CA2×...×CAc
11 ctrl_attr_vals(p) ≡
11 **let** ≪ CA1×CA2×...×CAc ≫ = ctrl_attr_typs(p) **in**
11 (attr_CA1(p),attr_CA2(p),...,attr_CAc(p)) **end**

Basic Principles for Ascribing Attributes

Section 1.5.3 dealt with technical issues of expressing attributes. This section will indicate some modelling principles.

Natural Parts

are in space and time – and are subject to laws of physics. So basic attributes focus on physical (including chemical) properties. These attributes cover the full spectrum of attribute categories outlined in Sect. 1.5.3.

Materials:

are in space and time – and are subject to laws of physics. So basic attributes focus on physical, especially chemical properties. These attributes cover the full spectrum of attribute categories outlined in Sect. 1.5.3.

•••

The next paragraphs, living species, animate entities and humans, reflect Sørlander's Philospohy [20, pp 14–182].

•••

Causality of Purpose: If there is to be ***the possibility of language and meaning*** then there must exist primary entities which are ***not entirely encapsulated within the physical conditions***; that they are stable and can influence one another. This is only possible if such primary entities are subject to a ***supplementary causality directed at the future***: a ***causality of purpose***. **Living Species:** These primary entities are here called ***living species***. What can be deduced about them ?

Living Species:

Living species are also in space and time – and are subject to laws of physics. Additionally living species ***plants*** and ***animals*** are characterised by ***causality of purpose***: they ***have some form they can be developed to reach;*** and which ***they must be causally determined to maintain;*** this development and maintenance must further in ***an exchange of matter with an environment***. It must be possible that living species occur in one of two forms: one form which is characterised by ***development***, ***form*** and ***exchange***, and another form which, additionally, can be characterised by the ability to ***purposeful movement***s. The first we call ***plant***s, the second we call ***animal***s.

Animate Entities:

For an animal to purposefully move around there must be "additional conditions" for such self-movements to be in accordance with the principle of causality: they must have ***sensory organ***s sensing among others the immediate purpose of its movement; they must have ***means of motion*** so that it can move; and they must have ***instinct***s, ***incentive***s and ***feeling***s as causal conditions that what it senses can drive it to movements. And all of this in accordance with the laws of physics.

Animals, to possess these three kinds of "additional conditions", must be built from special units which have an inner relation to their function as a whole; Their ***purposefulness*** must be built into their physical building units, that is, as we can now say, their ***genome***s. That is, animals are built from genomes which give them the ***inner determination*** to such building blocks for ***instinct***s, ***incentive***s and ***feeling***s. Similar kinds of deduction can be carried out with respect to plants. Transcendentally one can deduce basic principles of evolution but not its details.

Humans:

Consciousness and Learning: The existence of animals is a necessary condition for there being language and meaning in any world. That there can be ***language*** means that animals are capable of ***developing language***. And this must presuppose that animals can ***learn from their experience.*** To learn implies that animals can ***feel*** pleasure and distaste and can ***learn***. ... One can therefore deduce that animals must possess such building blocks whose inner determination is a basis for learning and consciousness. **Language:** Animals with higher social interaction uses ***sign***s, eventually developing a ***language***. These languages adhere to the same system of defined concepts which are a prerequisite for any description of any world: namely the system that philosophy lays bare from a basis of transcendental deductions and the ***principle of contradiction*** and its ***implicit meaning theory***. A ***human*** is an animal which has a ***language***. **Knowledge:** Humans must be ***conscious*** of having ***knowledge*** of its concrete situation, and as such that human can have knowledge about what he feels and eventually that human can know whether what he feels is true or false. Consequently ***a human can describe his situation correctly***. **Responsibility:** In this way one can deduce that humans can thus have ***memory*** and hence can have ***responsibility***, be ***responsible***. Further deductions lead us into ***ethics***.

•••

Intentionality

Intentionality *is a philosophical concept and is defined by the* `Stanford Encyclopedia of Philosophy`[27] *as* ***"the power of minds to be about, to represent, or to stand for, things, properties and states of affairs."***

Definition 31 **Intentional Pull:** Two or more artifactual parts of different sorts, but with overlapping sets of intents may excert an ***intentional "pull"*** on one another ■

This ***intentional "pull"*** may take many forms. Let $p_x : X$ and $p_y : Y$ be two parts of ***different sorts*** (X,Y), and with ***common intent***, ι. ***Manifestations*** of these, their common intent must somehow be ***subject to constraints***, and these must be ***expressed predicatively***. See Sect. 1.8.3, pp. 55–55, for an example.

•••

Artifacts:

Humans create artifacts – for a reason, to serve a purpose, that is, with **intent**. Artifacts are like parts. They satisfy the laws of physics – and serve a ***purpose***, fulfill an ***intent***.

•••

[27] Jacob, P. (Aug 31, 2010). ***Intentionality***. `Stanford Encyclopedia of Philosophy` (`https://seop.illc.uva.nl/entries/intentionality/`) October 15, 2014, retrieved April 3, 2018.

Assignment of Attributes:

So what can we deduce from the above, a little more than a page ?

The attributes of **natural parts** and **natural materials** are generally of such concrete types – expressible as some **real** with a dimension[28] of the International System of Units: `https://physics.nist.gov/cuu/Units/units.html`. Attribute values usually enter ***differential equations*** and ***integrals***, that is, classical calculus.

The attributes of **humans**, besides those of parts, significantly includes one of a usually non-empty set of ***intents***. In directing the creation of artifacts humans create these with an intent.

Examples: These are examples of human intents: they create ***roads*** and ***automobiles*** with the intent of ***transport***. they create ***houses*** with the intents of ***living, offices, production***, etc., and they create ***pipelines*** with the intent of ***oil*** or ***gas transport*** ■

Human attribute values usually enter into ***modal logic*** expressions.

Artifacts, including Man-made Materials: Artifacts, besides those of parts, significantly includes a usually singleton set of ***intents***.

Examples: ***roads*** and ***automobiles*** possess the intent of ***transport***; ***houses*** possess either one of the intents of ***living, offices, production***; and ***pipelines*** possess the intent of ***oil*** or ***gas transport*** ■

Artifact attribute values usually enter into ***mathematical logic*** expressions.

We leave it to the reader to formulate attribute assignment principles for plants and non-human animals.

1.5.4 The Unfolding of an Ontology

We have unfolded an ontology of domain endurants. Figure 1.3 illustrates this "unfolding": The upper left

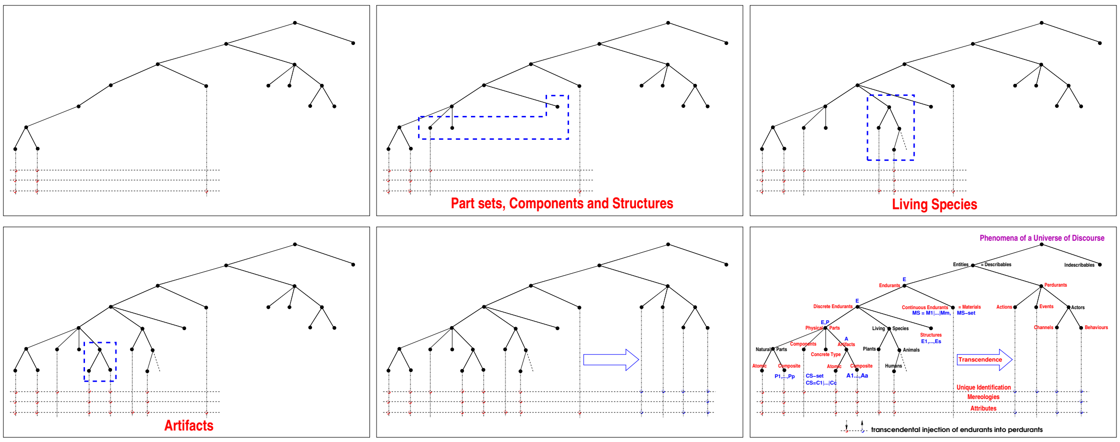


Fig. 1.3. Five Stages of Ontology Development

diagram shows the ontology of ***part*** and ***material*** endurants and of perdurants. The upper middle diagram shows the ontology addition of ***concrete part sets*** and ***structures***. The upper right diagram shows the ontology addition of ***living species***. The lower left diagram shows the ontology addition of ***artifacts***. The lower middle diagram shows the ontology with the ***transcendentally deduced*** "coupling" of ***internal endurant qualities*** with ***perdurant behaviour arguments***. The lower rightmost diagram shows the fully annotated ontology – and that diagram is the same as Fig. 1.1 on Page 13.

[28] Basic units are *m*eter, *k*ilogram, *s*econd, *A*mpere, *K*elvin, *mol*e, and *cand*ela. Some derived units are: *N*ewton: $kg \times m \times s^{-2}$, *Weber*: $kg \times m^2 \times s^{-2} \times A^{-1}$, etc.

1.5.5 Some Axioms and Proof Obligations – Sect. 1.8.1 pp. 50

By an **axiom** we shall – in the ***context*** of **domain analysis & description** – mean a logical expression, usually a predicate, that constrains the types and values, including unique identifiers and mereologies of domain models ■ Axioms, together with the sort, including type definitions, and the unique identifier, mereology and attribute observer functions, define the domain value spaces. We refer to axioms in Item [a] of domain description prompts of ***unique identifiers:*** 5 on Page 27 and of ***mereologies:*** 6 on Page 28.

By a **proof obligation** we shall – in the ***context*** of **domain analysis & description** – mean a logical expression that predicates relations between the types and values, including unique identifiers, mereologies and attributes of domain models, where these predicates must be shown, i.e., proved, to hold ■ Proof obligations supplement axioms. We refer to proof obligations in Item [p] of domain description prompts about ***endurant sorts:*** 1 on Page 23, about ***components sorts:*** 3 on Page 25, about ***materials sorts:*** 4 on Page 26, and about ***attribute types:*** 7 on Page 30.

The difference between expressing axioms and expressing proof obligations is this:

- **We use axioms** when our formula cannot otherwise express it simply, but when physical or other ***properties of the domain***[29] dictates property constraints.
- **We use proof obligations** where necssary constraints are not necessarily physically impossible.
- **Proof obligations** finally arise in the transition from endurants to perdurants where endurant axioms become properties that must be proved to hold.

When considering ***endurants*** we interpret these as stable, i.e., that although they may have, for example, programmable attributes, when we observe them, we observe them at any one moment, but ***we do not consider them over a time***. That is what we turn to next: ***perdurants.*** When considering a part with, for example, a programmable attribute, at two different instances of time we expect the particular programmable attribute to enjoy any expressed well-formedness properties. We shall, in Sect. 1.7, see how these programmable attributes re-occur as explicit behaviour parameters, "programmed" to possibly new values passed on to recursive invocations of the same behaviour. If well-formedness axioms were expressed for the part on which the behaviour is based, then a ***proof obligation*** arises, one that must show that new values of the programmed attribute satisfies the part attribute axiom. This is, but one relation between ***axioms*** and ***proof obligations***. We refer to remarks made in the bullet (•) named **Biddable Access** Page 42.

1.5.6 Discussion of Endurants – Sect. 1.8.1 pp. 50

Domain descriptions are, as we have already shown, formulated, both informally and formally, by means of abstract types, that is, by sorts for which no concrete models are usually given. Sorts are made to denote possibly empty, possibly infinite, rarely singleton, sets of entities on the basis of the qualities defined for these sorts, whether external or internal. By **junk** we shall understand that the domain description unintentionally denotes undesired entities. By **confusion** we shall understand that the domain description unintentionally have two or more identifications of the same entity or type. The question is ***can we formulate a [formal] domain description such that it does not denote junk or confusion*** ? The short answer to this is no ! So, since one naturally wishes "no junk, no confusion" what does one do ? The answer to that is ***one proceeds with great care !***

1.6 A Transcendental Deduction – Sect. 1.8.2 pp. 51

1.6.1 An Explanation

It should be clear to the reader that in **domain analysis & description** we are reflecting on a number of philosophical issues. First and foremost on those of ***epistemology*** and ***ontology***. In this section on a

[29] – examples of such properties are: (i) topologies of the domain makes certain compositions of parts physically impossible, and (ii) conservation laws of the domain usually dictates that endurants cannot suddenly arise out of nothing.

sub-field of epistemology, namely that of a number of issues of ***transcendental*** nature. We refer to [42, pp 878–880] [43, pp 807–810] [44, pp 54–55 (1998)].

Definition 32 **Transcendental:** By **transcendental** we shall understand the philosophical notion: **the a priori or intuitive basis of knowledge, independent of experience.**

A priori knowledge or intuition is central: By ***a priori*** we mean that it not only precedes, but also determines rational thought.

Definition 33 **Transcendental Deduction:** By a **transcendental deduction** we shall understand the philosophical notion: **a transcendental "conversion" of one kind of knowledge into a seemingly different kind of knowledge.**

Definition 34 **Transcendentality:** By **transcendentality** we shall here mean the philosophical notion: the state or condition of being transcendental.

Example 1.18. **Transcendentality**: We can speak of a bus in at least three ***senses:***

(i) The bus as it is being `"maintained, serviced, refueled"`;
(ii) the bus as it `"speeds"` down its route; and
(iii)the bus as it `"appears"` (listed) in a bus time table.

The three ***senses*** are:

(i) as an **endurant** (here a ***part***),
(ii) as a **perdurant** (as we shall see a ***behaviour***), and
(iii)as an **attribute**[30] ■

Example 1.18, we claim, reflects ***transcendentality*** as follows:

(i) We have knowledge of an endurant (i.e., a part) being an endurant.
(ii) We are then to assume that the perdurant referred to in (ii) is an aspect of the endurant mentioned in (i) – where perdurants are to be assumed to represent a different kind of knowledge.
(iii)And, finally, we are to further assume that the attribute mentioned in (iii) is somehow related to both (i) and (ii) – where at least this attribute is to be assumed to represent yet a different kind of knowledge.

In other words: two (i–ii) kinds of different knowledge; that they relate ***must indeed*** be based on ***a priori knowledge***. Someone claims that they relate ! The two statements (i–ii) are claimed to relate transcendentally.[31]

1.6.2 Some Special Notation

The ***transcendentality*** that we are referring to is one in which we **"translate"** endurant descriptions of ***parts*** and their ***unique identifiers, mereologies*** and ***attributes*** into perdurant descriptions, i.e., transcendental interpretations of parts as ***behaviours***, part mereologies as ***channels***, and part attributes as ***attribute value accesses***. The ***translations*** referred to above, ***compile*** endurant descriptions into RSL$^+$Text. We shall therefore first explain some aspects of this translation.

- Where in the function definition bodies
  - we enclose some RSL$^+$Text, e.g., rsl$^+$_text, in ⋘⋙s,
  - i.e., ⋘rsl$^+$_text ⋙
  - we mean that text.
- Where in the function definition bodies

[30] – in this case rather: as a fragment of a bus time table ***attribute***
[31] – the attribute statement was "thrown" in "for good measure", i.e., to highlight the issue !

  - we write ≪rsl$^+$_text ≫function_expression
  - we mean that rsl$^+$_text concatenated to the RSL$^+$Text
  - emanating from function_expression.
- Where in the function definition bodies
  - we write ≪≫function_expression
  - we mean just rsl$^+$_text
  - emanating from function_expression.
  - That is:
    - ≪≫function_expression ≡ function_expression and
    - ≪≫≪≫ ≡ ≪≫.
- Where in the function definition bodies
  - we write { ≪ $f(x)$ ≫| x:RSL$^+$Text}
  - we mean the "expansion" of the RSL$^+$Text $f(x)$,
  - in arbitrary, linear text order,
  - for appropriate RSL$^+$Texts x.

1.7 Perdurants – Sect. 1.8.3 pp. 51

Perdurants can perhaps best be explained in terms of a notion of ***state*** and a notion of ***time***. We shall, in this paper, not detail notions of ***time***, but refer to [45, 46, 47, 48].

1.7.1 States, Actors, Actions, Events and Behaviours: A Preview

States – Sect. 1.8.3 pp. 51

Definition 35 **Domain States:** By a **state** we shall understand any collection of **parts** or **components** or **materials** ■

We refer to Sect. 1.8.1 on Page 48.

Actors, Actions, Events, Behaviours and Channels

To us perdurants are further, pragmatically, analysed into ***action***s, ***event***s, and ***behaviour***s. We shall define these terms below. Common to all of them is that they potentially change a state. Actions and events are here considered atomic perdurants. For behaviours we distinguish between discrete and continuous behaviours.

Time Considerations

We shall, without loss of generality, assume that actions and events are atomic and that behaviours are composite. Atomic perdurants may "occur" during some time interval, but we omit consideration of and concern for what actually goes on during such an interval. Composite perdurants can be analysed into "constituent" actions, events and "sub-behaviours". We shall also omit consideration of temporal properties of behaviours. Instead we shall refer to two seminal monographs: Specifying Systems [49, Leslie Lamport] and Duration Calculus: A Formal Approach to Real-Time Systems [50, Zhou ChaoChen and Michael Reichhardt Hansen] (and [51, Chapter 15]). For a seminal book on "time in computing" we refer to the eclectic [52, Mandrioli et al., 2012]. And for seminal book on time at the epistemology level we refer to [48, J. van Benthem, 1991].

Actors

Definition 36 **Actor:** By an **actor** we shall understand something that is capable of initiating and/or **carrying out** actions, events or behaviours ■

The notion of ***"carrying out"*** will be made clear in this overall section. We shall, in principle, associate an actor with each part[32]. These actors will be described as behaviours. These behaviours evolve around a state. The state is the set of qualities, in particular the dynamic attributes, of the associated parts and/or any possible components or materials of the parts.

Discrete Actions

Definition 37 **Discrete Action:** By a **discrete action** [53, Wilson and Shpall] we shall understand a foreseeable thing which deliberately and potentially changes a well-formed state, in one step, usually into another, still well-formed state, for which an actor can be made responsible ■

An action is what happens when a function invocation changes, or potentially changes a state.

Discrete Events

Definition 38 **Event:** By an **event** we shall understand some unforeseen thing, that is, some 'not-planned-for' "action", one which surreptitiously, non-deterministically changes a well-formed state into another, but usually not a well-formed state, and for which no particular domain actor can be made responsible ■

Events can be characterised by a pair of (before and after) states, a predicate over these and, optionally, a *time* or *time interval*. The notion of event continues to puzzle philosophers [54, 55, 56, 57, 58, 59, 60, 61, 62, 63]. We note, in particular, [57, 59, 60].

Discrete Behaviours

Definition 39 **Discrete Behaviour:** By a **discrete behaviour** we shall understand a set of sequences of potentially interacting sets of discrete actions, events and behaviours ■

Discrete behaviours now become the ***focal point*** of our investigation. To every part we associate, by transcendental deduction, a behaviour. We shall express these behaviours as CSP ***processes*** [23] For those behaviours we must therefore establish their means of ***communication*** via ***channels***; their ***signatures***; and their ***definitions*** – as ***translated*** from endurant parts.

1.7.2 Channels and Communication – Sect. 1.8.3 pp. 51

The CSP Story:

Behaviours sometimes synchronise and usually communicate. We use the CSP [23]notation (adopted by RSL) to introduce and model behaviour communication. Communication is abstracted as the sending (ch ! m) and receipt (ch ?) of messages, m:M, over channels, ch.

```
type M
channel ch:M
```

Communication between (unique identifier) indexed behaviours have their channels modeled as similarly indexed channels:

```
out:     ch[idx]!m
in:      ch[idx]?
channel  {ch[ide]:M|ide:IDE}
```

where IDE typically is some type expression over unique identitifer types.

[32] This is an example of a ***transcendental deduction***.

From Mereologies to Channel Declarations:

The fact that a part, p of sort P with unique identifier p_i, has a mereology, for example the set of unique identifiers $\{q_a, q_b, ..., q_d\}$ identifying parts $\{qa, qb, ..., qd\}$ of sort Q, may mean that parts p and $\{qa, qb, ..., qd\}$ may wish to exchange – for example, attribute – values, one way (from p to the qs) or the other (vice versa) or in both directions. Figure 1.4 shows two dotted rectangle box diagrams. The left fragment of the figure

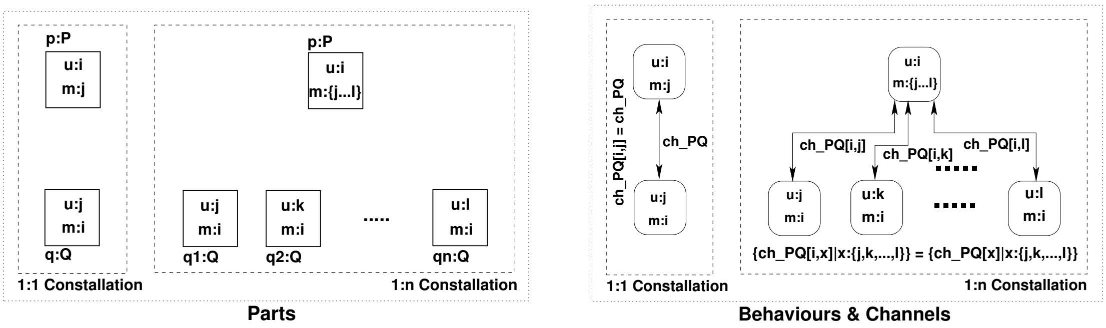


Fig. 1.4. Two Part and Channel Constallations. *u:p unique id. p; m:p mereology p*

intends to show a `1:1 Constallation` of a single p:P box and a single q:Q part, respectively, indicating, within these parts, their unique identifiers and mereologies. The right fragment of the figure intends to show a `1:n Constallation` of a single p:P box and a set of q:Q parts, now with arrowed lines connecting the p part with the q parts. These lines are intended to show channels. We show them with two way arrows. We could instead have chosen one way arrows, in one or the other direction. The directions are intended to show a direction of value transfer. We have given the same channel names to all examples, ch_PQ. We have ascribed channel message types MPQ to all channels.[33] Figure 1.5 shows an arrangement similar to that of Fig. 1.4, but for an `m:n Constallation`.

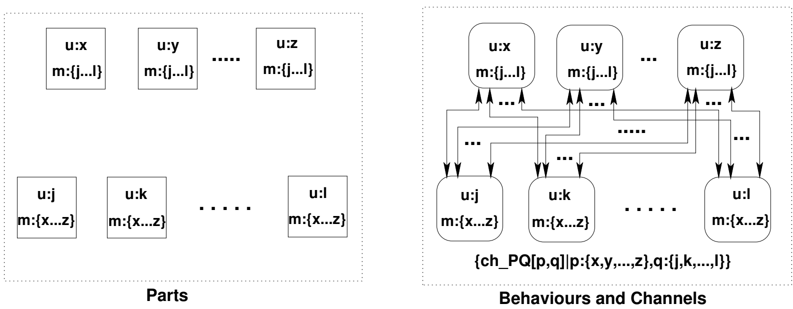


Fig. 1.5. Multiple Part and Channel Arrangements: *u:p unique id. p; m:pmereology p*

The channel declarations corresponding to Figs. 1.4 and 1.5 are:

channel
[1] ch_PQ[i,j]:MPQ
[2] { ch_PQ[i,x]:MPQ | x:{j,k,...,l} }
[3] { ch_PQ[p,q]:MPQ | p:{x,y,...,z}, q:{j,k,...,l} }

Since there is only one index i and j for channel [1], its declaration can be reduced. Similarly there is only one i for declaration [2]:

[33] Of course, these names and types would have to be distinct for any one domain description.

channel
[1] ch_PQ:MPQ
[2] { ch_PQ[x]:MPQ | x:{j,k,...,l} }

12 The following description identities holds:

12 { ch_PQ[x]:MPQ | x:{j,k,...,l} } $\equiv$ ch_PQ[j],ch_PQ[k],...,ch_PQ[l],

12 { ch_PQ[p,q]:MPQ | p:{x,y,...,z}, q:{j,k,...,l} } $\equiv$
12 ch_PQ[x,j],ch_PQ[x,k],...,ch_PQ[x,l],
12 ch_PQ[y,j],ch_PQ[y,k],...,ch_PQ[y,l],
12 ...,
12 ch_PQ[z,j],ch_PQ[z,k],...,ch_PQ[z,l]

We can sketch a diagram similar to Figs. 1.4 on the preceding page and 1.5 on the previous page for the case of composite parts.

Continuous Behaviours

By a continuous behaviour we shall understand a ***continuous time*** sequence of ***state change***s. We shall not go into what may cause these ***state change***s. And we shall not go into continuous behaviours in this paper.

1.7.3 Perdurant Signatures

We shall treat perdurants as function invocations. In our cursory overview of perdurants we shall focus on one perdurant quality: function signatures.

Definition 40 Function Signature: By a function signature we shall understand a ***function name*** and a ***function type expression*** ■

Definition 41 Function Type Expression: By a function type expression we shall understand a pair of ***type expression***s. separated by a ***function type constructor*** either $\rightarrow$ (for total function) or $\stackrel{\sim}{\rightarrow}$ (for partial function) ■

The ***type expression***s are part sort or type, or material sort or type, or component sort or type, or attribute type names, but may, occasionally be expressions over respective type names involving **-set**, $\times$, *, $\overrightarrow{m}$ and | type constructors.

Action Signatures and Definitions

Actors usually provide their initiated actions with arguments, say of type VAL. Hence the schematic function (action) signature and schematic definition:

action: VAL $\rightarrow \Sigma \stackrel{\sim}{\rightarrow} \Sigma$
action(v)(σ) **as** σ'
  pre: $\mathscr{P}$(v,σ)
  post: $\mathscr{Q}$(v,σ,σ')

expresses that a selection of the domain, as provided by the Σ type expression, is acted upon and possibly changed. The partial function type operator $\stackrel{\sim}{\rightarrow}$ shall indicate that action(v)(σ) may not be defined for the argument, i.e., initial state σ and/or the argument v:VAL, hence the precondition $\mathscr{P}$(v,σ). The post condition $\mathscr{Q}$(v,σ,σ') characterises the "after" state, σ':Σ, with respect to the "before" state, σ:Σ, and possible arguments (v:VAL). Which could be the argument values, v:VAL, of actions ? Well, there can basically be only the following kinds of argument values: parts, components and materials, respectively unique part identifiers, mereologies and attribute values. It basically has to be so since there are no other kinds of values in domains. There can be exceptions to the above (Booleans, natural numbers), but they are rare !

Perdurant (action) analysis thus proceeds as follows: identifying relevant actions, assigning names to these, delineating the "smallest" relevant state[34], ascribing signatures to action functions, and determining action pre-conditions and action post-conditions. Of these, ascribing signatures is the most crucial: In the process of determining the action signature one oftentimes discovers that part or component or material attributes have been left ("so far") "undiscovered".

Event Signatures and Definitions

Events are usually characterised by the absence of known actors and the absence of explicit "external" arguments. Hence the schematic function (event) signature:

value
  event: $\Sigma \times \Sigma \stackrel{\sim}{\to}$ **Bool**
  event(σ,σ') **as** tf
    pre: $P(\sigma)$
    post: tf $= Q(\sigma,\sigma')$

The event signature expresses that a selection of the domain as provided by the Σ type expression is "acted" upon, by unknown actors, and possibly changed. The partial function type operator $\stackrel{\sim}{\to}$ shall indicate that event(σ,σ') may not be defined for some states σ. The resulting state may, or may not, satisfy axioms and well-formedness conditions over Σ – as expressed by the post condition $Q(\sigma,\sigma')$. Events may thus cause well-formedness of states to fail. Subsequent actions, once actors discover such "disturbing events", are therefore expected to remedy that situation, that is, to restore well-formedness. We shall not illustrate this point.

Discrete Behaviour Signatures – Sect. 1.8.3 pp. 52

Signatures: We shall only cover behaviour signatures when expressed in RSL/CSP [22]. The behaviour functions are now called processes. That a behaviour function is a never-ending function, i.e., a process, is "revealed" by the "trailing" **Unit**:

  behaviour: ... → ... **Unit**

That a process takes no argument is "revealed" by a "leading" Unit:

  behaviour: **Unit** → ...

That a process accepts channel, viz.: ch, inputs, is "revealed" as follows:

  behaviour: ... → **in** ch ...

That a process offers channel, viz.: ch, outputs is "revealed" as follows:

  behaviour: ... → **out** ch ...

That a process accepts other arguments is "revealed" as follows:

  behaviour: ARG → ...

where ARG can be any type expression:

  T, T→T, T→T→T, etcetera

where T is any type expression.

[34] By "smallest" we mean: containing the fewest number of parts. Experience shows that the domain analyser cum describer should strive for identifying the smallest state.

Attribute Access

We shall only be concerned with part attributes. And we shall here consider them in the context of part behaviours. Part behaviour definitions embody part attributes. In this section we shall suggest how behaviours embody part attributes.

- **Static attributes** designate constants, cf. Defn. 1 Pg. 30. As such they can be "compiled" into behaviour definitions. We choose, instead to list them, in behaviour signatures, as arguments.
- **Inert attributes** designate values provided by external stimuli, cf. Defn. 3 Pg. 31, that is, must be obtained by channel input: attr_Inert_A_ch ?.
- **Reactive attributes** are functions of other attribute values, cf. Defn. 4 Pg. 31.
- **Autonomous attributes** must be input, cf. Defn. 6 Pg. 31, like inert attributes: attr_Autonomous_A_-ch ?.
- **Programmable attribute** values are calculated by their behaviours, cf. Defn. 8 Pg. 31. We list them as behaviour arguments. The behaviour definitions may then specify new values. These are provided in the position of the programmable attribute arguments in ***tail recursive*** invocations of these behaviours.
- **Biddable attributes** are like programmable attributes, but when provided in possibly tail recursive invocations of their behaviour the calculated biddable attribute value is ***modified***, usually by some ***perturbation***[35] of the calculated value – to reflect that although they ***are prescribed*** they ***may fail to be observed as such***, cf. Defn. 7 Pg. 31.

Calculating In/Output Channel Signatures

Given a part p we can calculate the RSL$^+$Text that designates the input channels on which part p behaviour obtains monitorable attribute values. For each monitorable attribute, A, the text ≪ attr_A_ch≫ is to be "generated". One or more such channel declaration contributions is to be preceded by the text ≪**in** ≫ If there are no monitorable attributes then no text is t be yielded.

13 The function calc_i_o_chn_refs apply to parts and yield RSL$^+$Text.
   a From p we calculate its unique identifier value, its mereology value, and its monitorable attribute values.
   b If there the mereology is not void and/or the are monitorable values then a (Currying[36]) right pointing arrow, →, is inserted.[37]
   c If there is an input mereology and/or there are monitorable values then the keyword **in** is inserted in front of the monitorable attribute values and input mereology.
   d Similarly for the input/output mereology;
   e and for the output mereology.

```
value
13   calc_i_o_chn_refs: P → RSL+Text
13   calc_i_o_chn_refs(p) ≡
13a      let ui = uid_P(p),
13a          (ics,iocs,ocs) = obs_mereo_(p),
13a          atrvs = obs_attrib_values_P(p) in
13b      if ics ∪ iocs ∪ ocs ∪ atrvs ≠ {}
13b          then ≪ → ≫ end
13c      if ics ∪ atrvs ≠{}
13c          then ≪in≫ calc_attr_chn_refs(ui,atrvs), calc_chn_refs(ui,ichs) end
13d      if iocs≠{}
13d          then ≪in,out≫  calc_chn_refs(ui,iochs) end
13e      if ocs≠{}
13e          then ≪out≫  calc_chn_refs(ui,ochs) end end
```

[35] – in the sense of https://en.wikipedia.org/wiki/Perturbation_function
[36] https://en.wikipedia.org/wiki/Currying
[37] We refer to the three parts of the mereology value as the input, the input/output and the output mereology (values).

14 The function calc_attr_chn_refs
  a apply to a set, mas, of monitorable attribute types and yield RSL^+Text.
  b If achs is empty no text is generated. Otherwise a channel declaration attr_A_ch is generated for each attribute type whose name, A, which is obtained by applying η to an observed attribute value, ηa.

14a calc_attr_chn_refs: UI × A-**set** → RSL^+Text
14b calc_attr_chn_refs(ui,mas) ≡
14b { ≪ attr_ηa_ch[ui] ≫ | a:A•a ∈ mas }

15 The function calc_chn_refs
  a apply to a pair, (ui,uis) of a unique part identifier and a set of unique part identifiers and yield RSL^+Text.
  b If uis is empty no text is generated. Otherwise an array channel declaration is generated.

15a calc_chn_refs: P_UI × Q_UI-**set** → RSL^+Text
15b calc_chn_refs(pui,quis) ≡ { ≪ η(pui,qui)_ch[pui,qui] ≫ | qui:Q_UI•qui ∈ quis }

16 The function calc_all_chn_dcls
  a apply to a pair, (pui,quis) of a unique part identifier and a set of unique part identifiers and yield RSL^+Text.
  b If quis is empty no text is generated. Otherwise an array channel declaration
    • { ≪ η(pui,qui)_ch[pui,qui]:η(pui,qui)M ≫ | qui:Q_UI•qui ∈ quis }
    is generated.

16a calc_all_chn_dcls: P_UI × Q_UI-**set** → RSL^+Text
16a calc_all_chn_dcls(pui,quis) ≡
16a { ≪η(pui,qui)_ch[pui,qui]:η(pui,qui)M ≫ | qui:Q_UI•qui ∈ quis }

The η(pui,qui) invocation serves to prefix-name both the channel, η(pui,qui)_ch[pui,qui], and the channel message type, η(pui,qui)M.

17 The overloaded η operator is here applied to a pair of unique identifiers.

17 η: (UI → RSL^+Text)|((X_UI×Y_UI) → RSL^+Text)
17 η(x_ui,y_ui) ≡ (≪(η x_ui η y_ui≫))

Repeating these channel calculations over distinct parts $p_1,p_2,...,p_n$ of the same part type P will yield "similar" behaviour signature channel references:

{PQ_ch[$p1_{ui}$,qui]|$p1_{ui}$:P_UI,qui:Q_UI•qui ∈ quis}
{PQ_ch[$p2_{ui}$,qui]|$p2_{ui}$:P_UI,qui:Q_UI•qui ∈ quis}
...
{PQ_ch[pn_{ui},qui]|pn_{ui}:P_UI,qui:Q_UI•qui ∈ quis}

These distinct single channel references can be assembled into one:

{ PQ_ch[pui,qui] | pui:P_UI,qui:Q_UI : −pui ∈ puis,qui ∈ quis }
where puis = { $p1_{ui}$,$p2_{ui}$,...,pn_{ui}}

As an example we have already calculated the array channels for Fig. 1.5 Pg. 39 – cf. the left, the **Parts**, of that figure – cf. Items [1–3] Pages 39–40. The identities Item 12 Pg. 40 apply.

1.7.4 Discrete Behaviour Definitions – Sect. 1.8.3 pp. 52

We associate with each part, $p{:}P$, a behaviour name $\mathcal{M}_P$. Behaviours have as first argument their unique part identifier: **uid**_P(p). Behaviours evolves around a state, or, rather, a set of values: its possibly changing mereology, mt:MT and the attributes of the part.[38] A behaviour signature is therefore:

$\mathcal{M}_P$: ui:UI×me:MT×stat_attr_typs(p) → ctrl_attr_typs(p) → calc_i_o_chn_refs(p) **Unit**

where (i) ui:UI is the unique identifier value and type of part p; (ii) me:MT is the value and type mereology of part p, me = **obs_mereo**_P(p); (iii) stat_attr_typs(p): static attribute types of part $p{:}P$; (iv) ctrl_attr_typs(p): controllable attribute types of part $p{:}P$; (v) calc_i_o_chn_refs(p) calculates references to the **in**put, the **in**put/**out**put and the **out**put channels serving the attributes shared between part p and the parts designated in its mereology me. Let P be a composite sort defined in terms of endurant[39] sub-sorts E_1, E_2, ..., E_n. The behaviour description ***translated*** from p:P, is composed from a behaviour description, $\mathcal{M}_P$, relying on and handling the unique identifier, mereology and attributes of part p to be ***translated*** with behaviour descriptions $\beta_1, \beta_2, \ldots, \beta_n$ where β_1 is ***translated*** from e_1:E_1, β_2 is ***translated*** from e_2:E_2, ..., and β_n is ***translated*** from e_n:E_n. The domain description ***translation*** schematic below "formalises" the above.

Process Schema 1

Abstract `is_composite(p)`

```
value
  Translate_P: P → RSL+Text
  Translate_P(p) ≡
    let ui = uid_P(p), me = obs_mereo_P(p),
        sa = stat_attr_vals(p), ca = ctrl_attr_vals(p),
        MT = mereo_type(p), ST = stat_attr_typs(p), CT = ctrl_attr_typs(p),
        IOR = calc_i_o_chn_refs(p), IOD = calc_all_ch_dcls(p) in
    ⋘ channel
          IOD
       value
          M_P: P_UI × MT × ST  CT  IOR  Unit
          M_P(ui,me,sta)(pa) ≡ B_P(ui,me,sta)ca
           ,⋙ Translate_P1(obs_endurant_sorts_E1(p))
          ⋘⋙ Translate_P2(obs_endurant_sorts_E2(p))
          ⋘⋙ ...
          ⋘⋙ Translate_Pn(obs_endurant_sorts_En(p))
    end
```

Expression $\mathcal{B}_P$(ui,me,sta,pa) stands for the ***behaviour definition body*** in which the names ui, me, sta, pa are bound to the ***behaviour definition head***, i.e., the left hand side of the ≡. Endurant sorts E_1, E_2, ..., E_n are obtained from the `observe_endurant_sorts` prompt, Page 23. We informally explain the Translate$_{P_i}$ function. It takes endurants and produces RSL$^+$Text. Resulting texts are bracketed: ⋘rsl_text⋙ For the case that an endurant is a structure there is only its elements to compile; otherwise Schema 2 is as Schema 1.

Process Schema 2

Abstract `is_structure(e)`

```
value
  Translate_P(p) ≡
```

[38] We leave out consideration of possible components and materials of the part.
[39] – structures or composite

```
          Translate_P1(obs_endurant_sorts_P1(p))
      ⫷⫸ Translate_P2(obs_endurant_sorts_P2(p))
      ⫷⫸ ...
      ⫷⫸ Translate_Pn(obs_endurant_sorts_Pn(p))
```

Let P be a composite sort defined in terms of the concrete type Q-**set**. The process definition compiled from p:P, is composed from a process, $\mathscr{M}_P$, relying on and handling the unique identifier, mereology and attributes of process p as defined by P operating in parallel with processes q:**obs_part**_Qs(p). The domain description "compilation" schematic below "formalises" the above.

Process Schema 3

Concrete is_composite(p)

```
type
   Qs = Q-set
value
   qs:Q-set = obs_part_Qs(p)
   Translate_P(p) ≡
      let ui = uid_P(p), me = obs_mereo_P(p),
            sa = stat_attr_vals(p), ca = ctrl_attr_vals(p)
            ST = stat_attr_typs(p), CT = ctrl_attr_typs(p),
            IOR = calc_i_o_chn_refs(p), IOD = calc_all_ch_dcls(p) in
      ⫷ channel
             IOD
         value
            ℳ_P: P_UI×MT×ST  CT  IOR  Unit
            ℳ_P(ui,me,sa)ca ≡ ℬ_P(ui,me,sa)ca  ⫸
            { ⫷,⫸ Translate_Q(q)|q:Q•q ∈ qs }
      end
```

Process Schema 4

Atomic is_atomic(p)

```
value
   Translate_P(p) ≡
      let ui = uid_P(p), me = obs_mereo_P(p),
         sa = stat_attr_vals(p), ca = ctrl_attr_vals(p),
         ST = stat_attr_typs(p), CT = ctrl_attr_typs(p),
         IOR = calc_i_o_chn_refs(p), IOD = calc_all_chs(p) in
      ⫷ channel
             IOD
         value
            ℳ_P: P_UI×MT×ST  PT  IOR  Unit
            ℳ_P(ui,me,sa)ca ≡ ℬ_P(ui,me,sa)ca  ⫸
      end
```

Process Schema 5

Core Process

The core processes can be understood as never ending, "tail recursively defined" processes:

$\mathscr{B}_P$: uid:P_UI×me:MT×sa:SA
→ ct:CT
→ **in** in_chns(p) **in**,**out** in_out_chns(me) **Unit**
$\mathscr{B}_P$(p)(ui,me,sa)(ca) ≡ **let** (me′,ca′) = $\mathscr{F}_P$(ui,me,sa)ca **in** $\mathscr{M}_P$(ui,me′,sa)ca′ **end**

$\mathscr{F}_P$: P_UI×MT×ST → CT→ in_out_chns(me) → MT×CT

We refer to [2, Process Schema V: Core Process (II), Page 40] for possible forms of $\mathscr{F}_P$.

1.7.5 Running Systems – Sect. 1.8.3 pp. 54

It is one thing to define the behaviours corresponding to all parts, whether composite or atomic. It is another thing to specify an initial configuration of behaviours, that is, those behaviours which "start" the overall system behaviour. The choice as to which parts, i.e., behaviours, are to represent an initial, i.e., a start system behaviour, cannot be "formalised", it really depends on the "deeper purpose" of the system. In other words: requires careful analysis and is beyond the scope of the present chapter. We refer to the example, Sect. 1.8.3 Pages 54–54.

1.7.6 Concurrency: Communication and Synchronisation

Process Schemas I, II, III and V (Pages 44, 44, 45 and 46), reveal that two or more parts, which temporally coexist (i.e., at the same time), imply a notion of ***concurrency***. Process Schema IV, Page 45, through the RSL/CSP language expressions ch ! v and ch ?, indicates the notions of ***communication*** and ***synchronisation***. Other than this we shall not cover these crucial notion related to ***parallelism***.

1.7.7 Summary and Discussion of Perdurants

The most significant contribution of Sect. 1.7 has been to show that for every domain description there exists a normal form behaviour — here expressed in terms of a CSP process expression.

Summary

We have proposed to analyse perdurant entities into actions, events and behaviours – all based on notions of state and time. We have suggested modeling and abstracting these notions in terms of functions with signatures and pre-/post-conditions. We have shown how to model behaviours in terms of CSP (communicating sequential processes). It is in modeling function signatures and behaviours that we justify the endurant entity notions of parts, unique identifiers, mereology and shared attributes.

Discussion

The analysis of perdurants into actions, events and behaviours represents a choice. We suggest skeptical readers to come forward with other choices.

1.8 A Methodology Example: A Transport System

The example of this section defines many concepts and has many formal identifiers. The Index of Sect. H.2 (Pages 432–437) may help you "recall" these definitions.

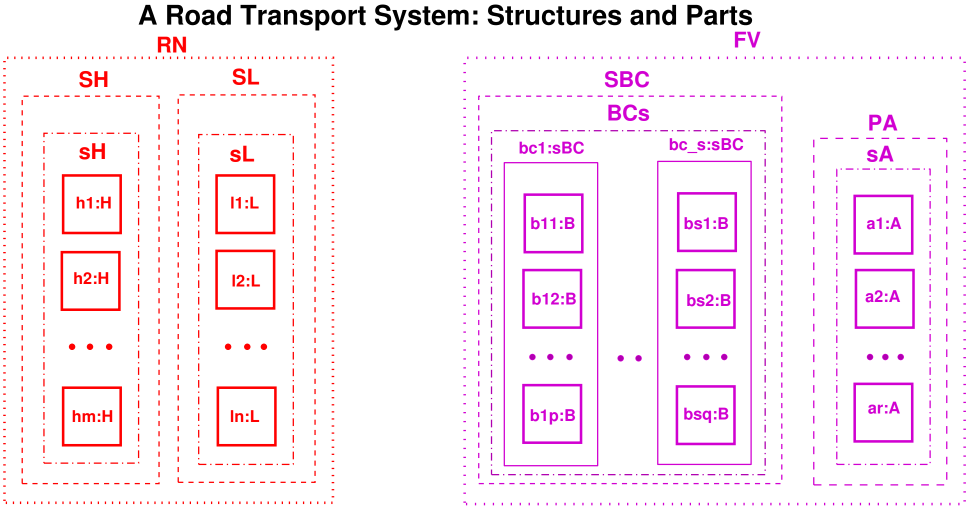


Fig. 1.6. A Road Transport System

1.8.1 Endurants – Sect. 1.3.2 pp. 16

The Discourse – Sect. 1.2.2 pp. 13

The universe of discourse is ***road transport systems***. We analyse & describe not the class of all road transport systems but a representative subclass, UoD, is ***structured*** into such notions as a road net, RN, of hubs, H, (intersections) and links, L, (street segments between intersections); a fleet of vehicles, FV, ***structured*** into companies, BC, of buses, B, and pools, PA, of private automobiles, A (et cetera); et cetera. See Fig. 1.6

Structures & Parts – Sect. 1.3.2 pp. 17

See Description Prompt 1, Pg. 23.

18 There is the ***universe of discourse***, UoD. It is structured into
19 a ***road net***, RN, a structure, and
20 a ***fleet of vehicles***, FV, a structure.

type
18 UoD **axiom** ∀ uod:UoD • is_structure(uod).
19 RN **axiom** ∀ rn:RN • is_strucure(rn).
20 FV **axiom** ∀ fv:FV • is_structure(fv).
value
19 obs_RN: UoD → RN
20 obs_FV: UoD → FV

Parts – Sect. 1.3.3 pp. 19

See Description Prompt 1, Pg. 23.

21 The road net consists of
 a a structure, SH, of hubs and
 b a structure, SL, of links.
22 The fleet of vehicles consists of
 a a structure, SBC, of ***bus companies***, and
 b a structure, PA, a ***pool of automobiles***.

type
21a SH **axiom** ∀ sh:SH • is_structure(sh)
21b SL **axiom** ∀ sl:SL • is_structure(sl)
22a SBC **axiom** ∀ sbc:SBC • is_structure(bc)
22b PA **axiom** ∀ pa:PA • is_structure(pa)
value
21a obs_SH: RN → SH
21b obs_SL: RN → SL
22a obs_BC: FV → BC
22b obs_PA: FV → PA

See Description Prompt 2, Pg. 23.

23 The structure of hubs is a set, sH, of atomic hubs, H.
24 The structure of links is a set, sL, of atomic links, L.
25 The structure of busses is a set, sBC, of composite bus companies, BC.
26 The composite bus companies, BC, are sets of busses, sB.
27 The structure of private automobiles is a set, sA, of atomic automobiles, A.

type
23 H, sH = H-**set** **axiom** ∀ h:H • is_atomic(h)
24 L, sL = L-**set** **axiom** ∀ l:L • is_atomic(l)
25 BC, BCs = BC-**set** **axiom** ∀ bc:BC • is_composite(bc)
26 B, Bs = B-**set** **axiom** ∀ b:B • is_atomic(b)
27 A, sA = A-**set** **axiom** ∀ a:A • is_atomic(a)
value
23 obs_sH: SH → sH
24 obs_sL: SL → sL
25 obs_sBC: SBC → BCs
26 obs_Bs: BCs → Bs
27 obs_sA: SA → sA

Components – Sect. 1.3.5 pp. 21

See Description Prompt 3, Pg. 25.

To illustrate the concept of components we describe timber yards, waste disposal areas, road material storage yards, automobile scrap yards, end the like as special "cul de sac" hubs with components. Here we describe road material storage yards.

28 Hubs may contain components, but only if the hub is connected to exactly one link.
29 These "cul-de-sac" hub components may be such things as Sand, Gravel, Cobble Stones, Asphalt, Cement or other.

value
28 has_components: H → **Bool**
type
29 Sand, Gravel, CobbleStones, Asphalt, Cement, ...
29 KS = (Sand|Gravel|CobbleStones|Asphalt|Cement|...)-**set**
value
28 obs_components_H: H → KS
28 **pre**: obs_components_H(h) ≡ **card** mereo(h) = 1

Materials – Sect. 1.3.6 pp. 21

See Description Prompt 4, Pg. 25.

To illustrate the concept of materials we describe waterways (river, canals, lakes, the open sea) along links as links with material of type water.

30 Links may contain material.
31 That material is water, W.

type
31 W
value
30 obs_material: L → W
30 **pre**: obs_material(l) ≡ has_material(h)

States – Sect. 1.3.8 pp. 22

32 Let there be given a universe of discourse, *rts*. It is an example of a state.

From that state we can calculate other states.

33 The set of all hubs, *hs*.
34 The set of all links, *ls*.
35 The set of all hubs and links, *hls*.
36 The set of all bus companies, *bcs*.
37 The set of all busses, *bs*.
38 The map from the unique bus company identifiers, see Item 44c Pg. 48, to the set of all the identifies bus company's buses, $bc_{ui}bs$.
39 The set of all private automobiles, *as*.
40 The set of all parts, *ps*.

value
32 *rts*:UoD
33 *hs*:H-**set** ≡ ≡ obs_sH(obs_SH(obs_RN(*rts*)))
34 *ls*:L-**set** ≡ ≡ obs_sL(obs_SL(obs_RN(*rts*)))
35 *hls*:(H|L)-**set** ≡ *hs*∪*ls*
36 *bcs*:BC-**set** ≡ obs_BCs(obs_SBC(obs_FV(obs_RN(*rts*))))
37 *bs*:B-**set** ≡ ∪{obs_Bs(bc)|bc:BC•bc ∈ *bcs*}
38 $bc_{ui}bs$:(BC_UI $\overrightarrow{m}$ B-**set**) ≡
38 [uid_BC(bc) ↦ obs_Bs(bc) | bc:BC • bc ∈ *bcs*]
39 *as*:A-**set** ≡ obs_BCs(obs_SBC(obs_FV(obs_RN(*rts*))))
40 *ps*:(H|L|BC|B|A)-**set** ≡ *hls*∪*bcs*∪*bs*∪*as*

Unique Identifiers – Sect. 1.5.1 pp. 26

Part Identifiers:

41 We assign unique identifiers to all parts.
42 By a road identifier we shall mean a link or a hub identifier.
43 By a vehicle identifier we shall mean a bus or an automobile identifier.
44 Unique identifiers uniquely identify all parts.
 a All hubs have distinct [unique] identifiers.
 b All links have distinct identifiers.
 c All bus companies have distinct identifiers.
 d All busses of all bus companies have distinct identifiers.
 e All automobiles have distinct identifiers.
 f All parts have distinct identifiers.

type
41 H_UI, L_UI, BC_UI, B_UI, A_UI
42 R_UI = H_UI | L_UI
43 V_UI = B_UI | A_UI
value
44a uid_H: H → H_UI
44b uid_L: H → L_UI
44c uid_BC: H → BC_UI
44d uid_B: H → B_UI
44e uid_A: H → A_UI

Extract Parts from Their Unique Identifiers:

45 From the unique identifier of a part we can retrieve, ℘, the part having that identifier.

type
45 P = H | L | BC | B | A
value
45 ℘: H_UI→H | L_UI→L | BC_UI→BC | B_UI→B | A_UI→A
45 ℘(ui) ≡ **let** p:(H|L|BC|B|A)•p∈*ps*∧uid_P(p)=ui **in** p **end**

Unique Identifier Constants:

We can calculate:

46 the *s*et, $h_{ui}s$, of *u*nique *h*ub *i*dentifiers;
47 the *s*et, $l_{ui}s$, of *u*nique *l*ink *i*dentifiers;
48 the *m*ap, $hl_{ui}m$, from *u*nique *h*ub *i*dentifiers to the *s*et of *u*nique *l*ink *i*identifiers of the links connected to the zero, one or more identified hubs,
49 the *m*ap, $lh_{ui}m$, from *u*nique *l*ink *i*dentifiers to the *s*et of *u*nique *h*ub *i*identifiers of the two hubs connected to the identified link;
50 the *s*et, $r_{ui}s$, of all *u*nique hub and link, i.e., *r*oad *i*dentifiers;
51 the *s*et, $bc_{ui}s$, of *u*nique *b*us *c*ompany *i*dentifiers;
52 the *s*et, $b_{ui}s$, of *u*nique *b*us *i*dentifiers;
53 the *s*et, $a_{ui}s$, of *u*nique private *a*utomobile *i*dentifiers;
54 the *s*et, $v_{ui}s$, of *u*nique bus and automobile, i.e., *v*ehicle *i*dentifiers;
55 the *m*ap, $bcb_{ui}m$, from *u*nique *b*us *c*ompany *i*dentifiers to the *s*et of its *u*nique *b*us *i*dentifiers; and
56 the (*b*ijective) *m*ap, $bbc_{ui}bm$, from *u*nique *b*us *i*dentifiers to their *u*nique *b*us *c*ompany *i*dentifiers.

value
46. $h_{ui}s$:H_UI-**set** ≡ {uid_H(h)|h:H•h ∈ *hs*}
47. $l_{ui}s$:L_UI-**set** ≡ {uid_L(l)|l:L•l ∈ *ls*}
50. $r_{ui}s$:R_UI-**set** ≡ $h_{ui}s$∪$l_{ui}s$
48. $hl_{ui}m$:(H_UI $\overrightarrow{m}$ L_UI-**set**) ≡
48. [h_ui↦luis|h_ui:H_UI,luis:L_UI-**set**•h_ui∈$h_{ui}s$
48. ∧(_,luis,_)=mereo_H(η(h_ui))] [cf. Item 63]
49. $lh_{ui}m$:(L+UI $\overrightarrow{m}$ H_UI-**set**) ≡
49. [l_ui↦huis [cf. Item 64]
49. | h_ui:L_UI,huis:H_UI-**set** • l_ui∈$l_{ui}s$
49. ∧ (_,huis,_)=mereo_L(η(l_ui))]
51. $bc_{ui}s$:BC_UI-**set** ≡ {uid_BC(bc)|bc:BC•bc ∈ *bcs*}
52. $b_{ui}s$:B_UI-**set** ≡ ∪{uid_B(b)|b:B•b ∈ *bs*}
53. $a_{ui}s$:A_UI-**set** ≡ {uid_A(a)|a:A•a ∈ *as*}
54. $v_{ui}s$:V_UI-**set** ≡ $b_{ui}s$ ∪ $a_{ui}s$
55. $bcb_{ui}m$:(BC_UI $\overrightarrow{m}$ B_UI-**set**) ≡
55. [bc_ui ↦ buis
55. | bc_ui:BC_UI, bc:BC •
55. bc∈*bcs* ∧ bc_ui=uid_BC(bc)
55. ∧ (_,_,buis)=mereo_BC(bc)]
56. $bbc_{ui}bm$:(B_UI $\overrightarrow{m}$ BC_UI) ≡
56. [b_ui ↦ bc_ui
56. | b_ui:B_UI,bc_ui:BC_ui •
56. bc_ui=**dom**$bcb_{ui}m$∧b_ui∈$bcb_{ui}m$(bc_ui)]

Uniqueness of Part Identifiers:

See Sect. 1.5.5 Pg. 35. We must express the following axioms:

57 All hub identifiers are distinct.
58 All link identifiers are distinct.
59 All bus company identifiers are distinct.
60 All bus identifiers are distinct.
61 All private automobile identifiers are distinct.
62 All part identifiers are distinct.

axiom
57 **card** *hs* = **card** $h_{ui}s$
58 **card** *ls* = **card** $l_{ui}s$
59 **card** *bcs* = **card** $bc_{ui}s$
60 **card** *bs* = **card** $b_{ui}s$
61 **card** *as* = **card** $a_{ui}s$
62 **card** {$h_{ui}s$∪$l_{ui}s$∪$bc_{ui}s$∪$b_{ui}s$∪$a_{ui}s$}
62 = **card** $h_{ui}s$+**card** $l_{ui}s$+**card** $bc_{ui}s$+**card** $b_{ui}s$+**card** $a_{ui}s$

We refer to 27.

Mereology – Sect. 1.5.2 pp. 27

63 The mereology of hubs is a triple: (i) the set of all bus and automobile identifiers[40], (ii) the set of unique identifiers of the links that it is connected to and the set of all unique identifiers of all vehicle (buses and private automobiles).[41], and (iii) an empty set.[42]
64 The mereology of links is a triple: (i) the set of all bus and automobile identifiers, (ii) the set of the two distinct hubs they are connected to, and (iii) an empty set.
65 The mereology of of a bus company is a triple: (i) an empty set, (ii) empty set, and (iii) and set the unique identifiers of the buses operated by that company.
66 The mereology of a bus is a triple: (i) the set of the one single unique identifier of the bus company it is operating for, (ii) an empty set, and (iii) the unique identifiers of all links and hubs[43].
67 The mereology of an automobiles is a triple: (i) an empty set, (ii) an empty set, and (iii) the set of the unique identifiers of all links and hubs[44].
68 Empty sets are modeled as empty sets of tokens where tokens are further undefined.

```
type
68  ES = TOKEN-set
68     axiom ∀ es:ES•es={}
63  H_Mer = V_UI-set×L_UI-set×ES
63     axiom ∀ (vuis,luis,__):H_Mer • luis⊆l_ui s ∧ vuis=v_ui s
64  L_Mer = V_UI-set×H_UI-set×ES
64     axiom ∀ (vuis,huis,__):L_Mer •
64              vuis=v_ui s ∧ huis⊆h_ui s ∧ card huis=2
65  BC_Mer = ES×ES×B_UI-set
65     axiom ∀ (__,__,buis):H_Mer • buis = b_ui s
66  B_Mer = BC_UI×ES×R_UI-set
66     axiom ∀ (bc_ui,__,ruis):H_Mer • bc_ui∈bc_ui s ∧ ruis=r_ui s
67  A_Mer = ES×ES×R_UI-set
67     axiom ∀ (__,ruis,__):A_Mer • ruis=r_ui s
```

```
value
63  mereo_H: H → H_Mer
64  mereo_L: L → L_Mer
65  mereo_BC: BC → BC_Mer
66  mereo_B: B → B_Mer
67  mereo_A: A → A_Mer
```

We can express some additional axioms, in this case for relations between hubs and links:

69 If hub, h, and link, l, are in the same road net,
70 and if hub h connects to link l then link l connects to hub h.

```
axiom
69  ∀ h:H,l:L • h ∈ hs ∧ l ∈ ls ⇒
       let (__,luis,__) = mereo_H(h), (__,huis,) = mereo_L(l) in
70     uid_L(l) ∈ luis ⇒ uid_H(h) ∈ huis end
```

More mereology axioms need be expressed – but we leave, to the reader, to narrate and formalise those.

Attributes – Sect. 1.5.3 pp. 29

We treat part attributes, sort by sort. See Description Prompt 7, Pg. 30

Hubs: We show just a few attributes:

71 There is a hub state. It is a set of pairs, (l_f,l_t) of link identifiers, where these link identifiers are in the mereology of the hub. The meaning of the hub state, in which, e.g., (l_f,l_t) is an element, is that the hub is open, **"green"**, for traffic $f$rom link l_f to link l_t. If a hub state is empty then the hub is closed, i.e., **"red"** for traffic from any connected links to any other connected links.
72 There is a hub state space. It is a set of hub states. The meaning of the hub state space is that its states are all those the hub can attain. The current hub state must be in its state space.
73 Since we can think rationally about it, it can be described, hence it can model, as an attribute of hubs a history of its traffic: the recording, per unique bus and automobile identifier, of the time ordered presence in the hub of these vehicles.
74 The link identifiers of hub states must be in the set, $l_{ui}s$, of the road net's link identifiers.

```
type
71  HΣ = (L_UI×L_UI)-set                      [programmable, Df.8 Pg.31]
axiom
71  ∀ h:H • obs_HΣ(h) ∈ obs_HΩ(h)
type
72  HΩ = HΣ-set                                      [static, Df.1 Pg.30]
73  H_Traffic                                 [programmable, Df.8 Pg.31]
73  H_Traffic = (A_UI|B_UI) →m (𝒯 × VPos)*
axiom
73  ∀ ht:H_Traffic,ui:(A_UI|B_UI)•ui ∈ dom ht
73      ⇒ time_ordered(ht(ui))
value
71  attr_HΣ: H → HΣ
72  attr_HΩ: H → HΩ
73  attr_H_Traffic: : → H_Traffic
axiom
74  ∀ h:H • h ∈ hs ⇒
74     let hσ = attr_HΣ(h) in
74     ∀ (l_ui i,li_ui i′):(L_UI×L_UI) • (l_ui i,l_ui i′) ∈ hσ
74          ⇒ {l_ui_i,l′_ui_i} ⊆ l_ui s end
value
73    time_ordered: 𝒯* → Bool
73    time_ordered(tvpl) ≡ ...
```

Links: We show just a few attributes:

75 There is a link state. It is a set of pairs, (h_f,h_t), of distinct hub identifiers, where these hub identifiers are in the mereology of the link. The meaning of a link state in which (h_f,h_t) is an element is that the link is open, **"green"**, for traffic $f$rom hub h_f to hub h_t. Link states can have either 0, 1 or 2 elements.
76 There is a link state space. It is a set of link states. The meaning of the link state space is that its states are all those the which the link can attain. The current link state must be in its state space. If a link state space is empty then the link is (permanently) closed. If it has one element then it is a one-way link. If a one-way link, l, is imminent on a hub whose mereology designates that link, then the link is a "trap", i.e., a "blind cul-de-sac".
77 Since we can think rationally about it, it can be described, hence it can model, as an attribute of links a history of its traffic: the recording, per unique bus and automobile identifier, of the time ordered positions along the link (from one hub to the next) of these vehicles.
78 The hub identifiers of link states must be in the set, $h_{ui}s$, of the road net's hub identifiers.

```
type
75  LΣ = H_UI-set                             [programmable, Df.8 Pg.31]
axiom
75  ∀ lσ:LΣ•card lσ=2
75  ∀ l:L • obs_LΣ(l) ∈ obs_LΩ(l)
type
76  LΩ = LΣ-set                                      [static, Df.1 Pg.30]
77  L_Traffic                                 [programmable, Df.8 Pg.31]
77  L_Traffic = (A_UI|B_UI) →m (𝒯×(H_UI×Frac×H_UI))*
77  Frac = Real, axiom frac:Fract • 0<frac<1
value
75  attr_LΣ: L → LΣ
76  attr_LΩ: L → LΩ
77  attr_L_Traffic: : → L_Traffic
```

[40] This is just another way of saying that the meaning of hub mereologies involves the unique identifiers of all the vehicles that might pass through the hub `is_of_interest` to it
[41] ... its link identifiers designate the links, zero, one or more, that a hub is connected to `is_of_interest` to both the hub and that these links is `interested` in the hub.
[42] ... the hubs are not "proactive", i.e., that the universe of discourse have no parts that are `interested` in the hub.
[43] that the bus might pass through
[44] that the automobile might pass through

```
axiom
77  ∀ lt:L_Traffic,ui:(A_UI|B_UI)•ui ∈ dom ht
77        ⇒ time_ordered(ht(ui))

78  ∀ l:L • l ∈ ls ⇒
78      let lσ = attr_LΣ(l) in
78      ∀ (h_ui i,h_ui i′):(H_UI×K_UI) •
78           (h_ui i,h_ui i′) ∈ lσ ⇒ {h_ui_i,h′_ui_i} ⊆ h_ui s end
```

Bus Companies: Bus companies operate a number of lines that service passenger transport along routes of the road net. Each line being serviced by a number of busses.

79 Bus companies have a physical, i.e., "real, actual" time attribute.
80 Bus companies create, maintain, revise and distribute [to the public (not modeled here), and to busses] bus time tables, not further defined.

```
type
79  𝒯                          [inert, Df.3 Pg.31]
80  BusTimTbl           [programmable, Df.8 Pg.31]
value
79  attr_T: BC → 𝒯
80  attr_BusTimTbl: BC → BusTimTbl
```

There are two notions of time at play here: the inert "real" or "actual" time as an inert attribute provided by some outside "agent"; and the calendar, hour, minute and second time designation occurring in some textual form in, e.g., time tables..

Busses: We show just a few attributes:

79 Buses have a time attribute.
81 Busses run routes, according to their line number, ln:LN, in the
82 bus time table, btt:BusTimTbl obtained from their bus company, and and keep, as inert attributes, their segment of that time table.
83 Busses occupy positions on the road net:
  a either ***at a hub*** identified by some h_ui,
  b or ***on a link***, some ***fraction***, f:Fract, down an ***identified link***, ***l_ui***, from one of its ***identified connecting hub***s, fh_ui, in the direction of the other ***identified hub***, th_ui.
84 Et cetera.

```
type
79   𝒯                             [inert, Df.3 Pg.31]
81   LN                     [programmable, Df.8 Pg.31]
82   BusTimTbl                     [inert, Df.3 Pg.31]
83   BPos   == atHub | onLink [programmable, Df.8 Pg.31]
83a  atHub    :: h_ui:H_UI
83b  onLink   :: fh_ui:H_UI×l_ui:L_UI×frac:Fract×th_ui:H_UI
83b  Fract     = Real, axiom frac:Fract • 0<frac<1
84   ...
value
79   attr_T: B → 𝒯
82   attr_BusTimTbl: B → BusTimTbl
83   attr_BPos: B → BPos
```

Private Automobiles: We show just a few attributes: We illustrate but a few attributes:

79 Automobiles have a time attribute.
85 Automobiles have static number plate registration numbers.
86 Automobiles have dynamic positions on the road net:
  [83a] either ***at a hub*** identified by some h_ui,
  [83b] or ***on a link***, some ***fraction, frac:Fract*** down an ***identified link, l_ui***, from one of its ***identified connecting hub***s, fh_ui, in the direction of the other ***identified hub***, th_ui.
87 Automobiles have a history: a (time ordered) sequence of timed automobile positions;

```
type
79   𝒯                              [inert, Df.3 Pg.31]
85   RegNo                         [static, Df.1 Pg.30]
86   APos  == atHub | onLink [programmable, Df.8 Pg.31]
83a  atHub    :: h_ui:H_UI
83b  onLink   :: fh_ui:H_UI × l_ui:L_UI × frac:Fract × th_ui:H_UI
83b  Fract     = Real, axiom frac:Fract • 0<frac<1
```

```
value
79  attr_T: A → 𝒯
85  attr_RegNo: A → RegNo
86  attr_APos: A → APos
87   A_Hist  = (𝒯 × APos)*
```

Obvious attributes that are not illustrated are those of velocity and acceleration, forward or backward movement, turning right, left or going straight, etc. The ***acceleration, deceleration, even velocity,*** or ***turning right, turning left, moving straight***, or ***forward*** or ***backward*** are seen as ***command actions***. As such they denote actions by the automobile — such as pressing the accelerator, or lifting accelerator pressure or ***braking***, or ***turning the wheel*** in one direction or another, etc. As actions they have a kind of counterpart in the velocity, the acceleration, etc. attributes.

Discussion

Observe that bus companies each have their own distinct ***bus time table***, and that these are modeled as ***programmable***, Item 79, Page 50. Observe then that busses each have their own distinct ***bus time table***, and that these are model-led as ***inert***, Item 82, Page 50. In Items 118–119b Pg. 53 we shall see how the busses communicate with their respective bus companies in order for the busses to obtain the ***programmed*** bus time tables "in lieu" of their ***inert*** one ! In Items 73 Pg. 49 and 77 Pg. 49, we illustrated an aspect of domain analysis & description that may seem, and at least some decades ago would have seemed, strange: namely that if we can think, hence speak, about it, then we can model it "as a fact" in the domain. The case in point is that we include among hub and link attributes their histories of the timed whereabouts of buses and automobiles.[45]

Some Axioms and Proof Obligations – Sect. 1.5.5 pp. 35

Examples of axioms are given in Items 57 – 62 Pg. 48, Items 69 – 70 Pg. 49, Item 73, and in Item 77. We shall give an example of a **proof obligation** expressed as a **post** condition, related to the last two of the above axioms, in Items 110g Pg. 53 and 117 Pg. 53

Those proof obligations reflect an aspect of the concept of **transcendental deduction:** that ***axioms*** over, as here, ***internal qualities*** of ***endurants*** via ***post conditions*** of ***perdurants*** become ***proof obligations !***

Discussion of Endurants – Sect. 1.5.6 pp. 35

- We have chosen to model some discrete endurants
  - as structures
  - others as parts (usually composite).
- Those choices are made mostly to illustrate that the domain analysis & description has a choice.
  - If a choice is made to model a discrete endurant as a structure
    - then it entails that the domain analysis & description does not wish to "implement" that discrete endurant as a behaviour separate from its sub-endurants;
  - If the choice is made to model a discrete endurant as a part
    - then it entails that the domain analysis & description wishes to "implement" that discrete endurant as a behaviour separate from its sub-endurants.
- The following discrete endurants which are modeled as structures above, could, instead, if modeled as parts, have the entailed behaviours reflect the following possibilities:
  - ***road net***, rn:RN: The road net behaviour could be that of a road net authority charged with building, servicing, operating and maintaining the road net. Building and maintaining the road net could mean the insertion of new or removal of old links or hubs. Operating the road net could mean the gathering of bus and automobile traffic statistics, the setting of hub states (traffic signal monitoring and control), etc.
  - ***aggregate of bus companies***, sbc:SBC: The composite aggregate of bus companies could be that of a public transport authority charged with establishing, servicing, operating and maintaining a common bus time table, etc.
  - ***aggregate of private automobiles***, ps:PA: The aggregate of private automobiles could be that of one or more ***automobile clubs***, etc.

[45] In this day and age of road cameras and satellite surveillance these traffic recordings may not appear so strange: We now know, at least in principle, of technologies that can record approximations to the hub and link traffic attributes.

1.8.2 Transcendentality – Sect. 1.6 pp. 35

We refer to Sect. 1.6 on Page 35 Defn. 1.18 Page 36.

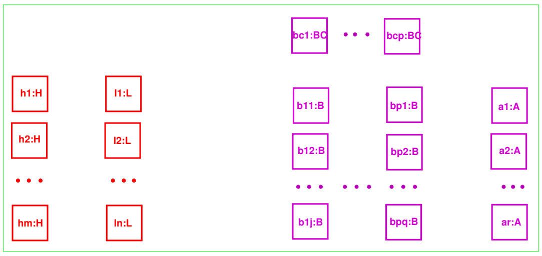
Example 1.19. **A Case of Transcendentality**: We refer to the following example: We can speak of a bus in at least three ***senses:***

- The bus as it is being maintained, serviced, refueled;
- the bus as it "speeds" down its route; and
- the bus as it "appears" (listed) in a bus time table.

The three ***senses*** are:

- as a part,
- as a behaviour, and
- as an attribute[46] ■

1.8.3 Perdurants – Sect. 1.7 pp. 37



In the figure above we "symbolically", i.e., the "...", show the following parts: each individual hub, each individual link, each individual bus company, each individual bus, and each individual automobile – and all of these. The idea is that those are the parts for which we shall define behaviours. That figure, however, and in contrast to Fig. 1.6 Pg. 47, shows the composite parts as not containing their atomic parts, but as if they were "free-standing, atomic" parts. That shall visualise the transcendental interpretation as atomic part behaviours not being somehow embedded in composite behaviours, but operating concurrently, in parallel.

Constants and States – Sect. 1.7.1 pp. 37

Constants:

We refer to Sect. 1.7.1 Pg. 37, and to App. 1.8.1 Pg. 48 We assume, as a constant, an arbitrarily selected universe of discourse, *uod*, and calculate from *uod* all its endurants.

value
32 *rts*:UoD [32]
33 *hs*:H-**set** ≡:H-**set** ≡ obs_sH(obs_SH(obs_RN(*rts*))) [33]
34 *ls*:L-**set** ≡:L-**set** ≡ obs_sL(obs_SL(obs_RN(*rts*))) [34]
35 *hls*:(H|L)-**set** ≡ *hs*∪*ls* [35]
36 *bcs*:BC-**set** ≡ obs_BCs(obs_SBC(obs_FV(obs_RN(*rts*)))) [36]
37 *bs*:B-**set** ≡ ∪{obs_Bs(bc)|bc:BC•bc ∈ *bcs*} [37]
38 *as*:A-**set** ≡ obs_BCs(obs_SBC(obs_FV(obs_RN(*rts*)))) [38]

Indexed States:

We shall

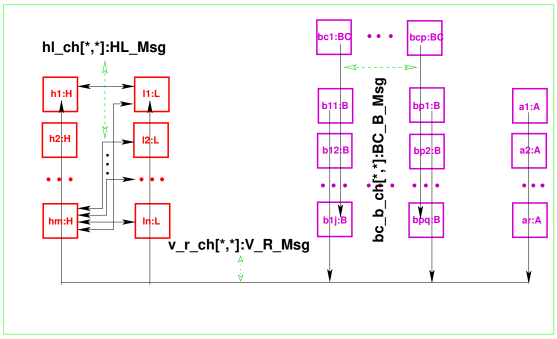
88 index bus companies,
89 index buses, and
90 index automobiles

using the unique identifiers of these parts.

type
88 BC_{ui}
89 B_{ui}
90 A_{ui}
value
88 *ibcs*:BC_{ui}-**set** ≡
88 { bc_{ui} | bc:BC,bc:BC_{ui}:BC_{ui} • bc∈*bcs*∧*ui*=uid_BC(bc) }
89 *ibs*:B_{ui}-**set** ≡
89 { b_{ui} | b:B,b:B_{ui}:B_{ui} • b∈*bs*∧*ui*=uid_B(b) }
90 *ias*:A_{ui}-**set** ≡
90 { a_{ui} | a:A,a:A_{ui}:A_{ui} • a∈*as*∧*ui*=uid_A(a) }

Channels – Sect. 1.7.2 pp. 38

We shall argue for hub-to-link channels based on the mereologies of those parts. Hub parts may be topologically connected to any number, 0 or more, link parts. Only instantiated road nets knows which. Hence there must be channels between any hub behaviour and any link behaviour. Vice versa: link parts will be connected to exactly two hub parts. Hence there must be channels from any link behaviour to two hub behaviours. See the figure below:



Channel Message Types:

We ascribe types to the messages offered on channels.

91 Hubs and links communicate, both ways, with one another, over channels, `hl_ch`, whose indexes are determined by their mereologies.
92 Hubs send one kind of messages, links another.
93 Bus companies offer timed bus time tables to buses, one way.
94 Buses and automobiles offer their current, timed positions to the road element, hub or link they are on, one way.

type
92 H_L_Msg, L_H_Msg
91 HL_Msg = H_L_Msg | L_F_Msg
93 BC_B_Msg = T × BusTimTbl
94 V_R_Msg = T × (BPos|APos)

[46] in this case rather: as a fragment of an attribute

Channel Declarations:

...

95 This justifies the channel declaration which is calculated to be:

channel
95 { hl_ch[h_ui,l_ui]:H_L_Msg
95 | h_ui:H_UI,l_ui:L_UI•i ∈ $h_{ui}s$∧j ∈ $lh_{ui}m$(h_ui) }
95 ∪
95 { hl_ch[h_ui,l_ui]:L_H_Msg
95 | h_ui:H_UI,l_ui:L_UI•l_ui ∈ $l_{ui}s$∧i ∈ $lh_{ui}m$(l_ui) }

We shall argue for bus company-to-bus channels based on the mereologies of those parts. Bus companies need communicate to all its buses, but not the buses of other bus companies. Buses of a bus company need communicate to their bus company, but not to other bus companies.

96 This justifies the channel declaration which is calculated to be:

channel
96 {bc_b_ch[bc_ui,b_ui]:BC_B_Msg
96 | bc_ui:BC_UI, b_ui:B_UI •
96 bc_ui ∈ $bc_{ui}s$ ∧ b_ui ∈ $b_{ui}s$}
96 {bc_b_ch[bc_ui,b_ui] |
96 bc_ui:BC_UI,b_ui:B_UI•bc_ui ∈ $bc_{ui}s$∧j ∈ $b_{ui}s$}: BC_B_MSG
96 {bc_b_ch[bc_ui,b_ui] |
96 bc_ui:BC_UI,b_ui:B_UI•bc_ui ∈ $bc_{ui}s$∧j ∈ $b_{ui}s$}: BC_B_MSG

We shall argue for vehicle to road element channels based on the mereologies of those parts. Buses and automobiles need communicate to all hubs and all links.

97 This justifies the channel declaration which is calculated to be:

channel
97 {v_r_ch[v_ui,r_ui]:V_R_Msg |
97 v_ui:V_UI,r_ui:R_UI•v_ui ∈ $v_{ui}s$∧r_ui ∈ $r_{ui}s$}

The channel calculations are described on Pages 42–43.

Behaviour Signatures – Sect. 1.7.3 pp. 41

We first decide on names of behaviours. In Sect. 1.7.4, Pages 44–45, we gave schematic names to behaviours of the form $\mathscr{M}_P$. We now assign mnemonic names: from part names to names of transcendentally interpreted behaviours and then we assign signatures to these behaviours.

98 hub$_{h_{ui}}$:
 a there is the usual "triplet" of arguments: unique identifier, mereology and static attributes;
 b then there are the programmable attributes;
 c and finally there are the input/output channel references: first those allowing communication between hub and link behaviours,
 d and then those allowing communication between hub and vehicle (bus and automobile) behaviours.

value
98 hub$_{h_{ui}}$:
098a h_ui:H_UI×(vuis,luis,_):H_Mer×HΩ
98b → (HΣ×H_Traffic)
98c → **in**,**out** { h_l_ch[h_ui,l_ui] | l_ui:L_UI:l_ui ∈ luis }
98d { ba_r_ch[h_ui,v_ui] | v_ui:V_UI•v_ui∈vuis } **Unit**
98a **pre**: vuis = $v_{ui}s$ ∧ luis = $l_{ui}s$

99 link$_{l_{ui}}$:
 a there is the usual "triplet" of arguments: unique identifier, mereology and static attributes;
 b then there are the programmable attributes;
 c and finally there are the input/output channel references: first those allowing communication between hub and link behaviours,
 d and then those allowing communication between link and vehicle (bus and automobile) behaviours.

value
99 link$_{l_{ui}}$:
99a l_ui:L_UI×(vuis,huis,_):L_Mer×LΩ
99b → (LΣ×L_Traffic)
99c → **in**,**out** { h_l_ch[h_ui,l_ui] | h_ui:H_UI:h_ui ∈ huis }
99d { ba_r_ch[l_ui,v_ui] | v_ui:(B_UI|A_UI)•v_ui∈vuis } **Unit**
99a **pre**: vuis = $v_{ui}s$ ∧ huis = $h_{ui}s$

100 bus_company$_{bc_{ui}}$:
 a there is here just a "doublet" of arguments: unique identifier and mereology;
 b then there is the one programmable attribute;
 c and finally there are the input/output channel references: first the input time channel,
 d then the input/output allowing communication between the bus company and buses.

value
100 bus_company$_{bc_{ui}}$:
100a bc_ui:BC_UI×(_,_,buis):BC_Mer
100b → BusTimTbl
100c → **in** attr_T_ch
100d **in**,**out** {bc_b_ch[bc_ui,b_ui]|b_ui:B_UI•b_ui∈buis} **Unit**
100a **pre**: buis = $b_{ui}s$ ∧ huis = $h_{ui}s$

101 bus$_{b_{ui}}$:
 a there is here just a "doublet" of arguments: unique identifier and mereology;
 b then there are the programmable attributes;
 c and finally there are the input/output channel references: first the input time channel, and the input/output allowing communication between the bus company and buses,
 d and the input/output allowing communication between the bus and the hub and link behaviours.

value
101 bus$_{b_{ui}}$:
101a b_ui:B_UI×(bc_ui,_,ruis):B_Mer
101b → (LN × BTT × BPOS)
101c → **in** attr_T_ch **in**,**out** bc_b_ch[bc_ui,b_ui],
101d {ba_r_ch[r_ui,b_ui]|r_ui:(H_UI|L_UI)•ui∈$v_{ui}s$} **Unit**
101a **pre**: ruis = $r_{ui}s$ ∧ bc_ui ∈ $bc_{ui}s$

102 automobile$_{a_{ui}}$:
 a there is the usual "triplet" of arguments: unique identifier, mereology and static attributes;
 b then there is the one programmable attribute;
 c and finally there are the input/output channel references: first the input time channel,
 d then the input/output allowing communication between the automobile and the hub and link behaviours.

value
102 automobile$_{a_{ui}}$:
102a a_ui:A_UI×(_,_,ruis):A_Mer×rn:RegNo
102b → apos:APos
102c → **in** attr_T_ch
102d **in**,**out** {ba_r_ch[a_ui,r_ui]|r_ui:(H_UI|L_UI)•r_ui∈ruis} **Unit**
102a **pre**: ruis = $r_{ui}s$ ∧ a_ui ∈ $a_{ui}s$

Behaviour Definitions – Sect. 1.7.4 pp. 44

We define the behaviours in a different order than the treatment of their signatures. We "split" definition of the automobile behaviour into the behaviour of automobiles when positioned at a hub, and into the behaviour automobiles when positioned at on a link. In both cases the behaviours include the "idling" of the automobile, i.e., its "not moving", standing still.

Automobiles:

103 We abstract automobile behaviour at a Hub (hui).
104 The vehicle remains at that hub, "idling",
105 informing the hub behaviour,
106 or, internally non-deterministically,
   a moves onto a link, tli, whose "next" hub, identified by th_ui, is obtained from the mereology of the link identified by tl_ui;
   b informs the hub it is leaving and the link it is entering of its initial link position,
   c whereupon the vehicle resumes the vehicle behaviour positioned at the very beginning (0) of that link,
107 or, again internally non-deterministically,
108 the vehicle "disappears — off the radar" !

```
103   automobile_{a_ui}(a_ui,({},(ruis,vuis),{}),rn)
103         (apos:atH(fl_ui,h_ui,tl_ui)) ≡
104     (ba_r_ch[a_ui,h_ui] ! (attr_T_ch?,atH(fl_ui,h_ui,tl_ui));
105      automobile_{a_ui}(a_ui,({},(ruis,vuis),{}),rn)(apos))
106     ⊓
106a    (let ({fh_ui,th_ui},ruis′)=mereo_L(℘(tl_ui)) in
106a          assert: fh_ui=h_ui ∧ ruis=ruis′
103      let onl = (tl_ui,h_ui,0,th_ui) in
106b      (ba_r_ch[a_ui,h_ui] ! (attr_T_ch?,onL(onl)) ∥
106b       ba_r_ch[a_ui,tl_ui] ! (attr_T_ch?,onL(onl))) ;
106c       automobile_{a_ui}(a_ui,({},(ruis,vuis),{}),rn)
106c             (onL(onl)) end end)
107     ⊓
108      stop
```

109 We abstract automobile behaviour on a Link.
   a Internally non-deterministically, either
      i the automobile remains, "idling", i.e., not moving, on the link,
      ii however, first informing the link of its position,
   b or
      i **if** if the automobile's position on the link ***has not yet reached the hub***, **then**
         1 then the automobile moves an arbitrary small, positive **Real**-valued ***increment*** along the link
         2 informing the hub of this,
         3 while resuming being an automobile ate the new position, or
      ii **else**,
         1 while obtaining a "next link" from the mereology of the hub (where that next link could very well be the same as the link the vehicle is about to leave),
         2 the vehicle informs both the link and the imminent hub that it is now at that hub, identified by th_ui,
         3 whereupon the vehicle resumes the vehicle behaviour positioned at that hub;
   c or
   d the vehicle "disappears — off the radar" !

```
109      automobile_{a_ui}(a_ui,({},ruis,{}),rno)
109                  (vp:onL(fh_ui,l_ui,f,th_ui)) ≡
109(a)ii   (ba_r_ch[thui,aui]!atH(lui,thui,nxt_lui) ;
109(a)i     automobile_{a_ui}(a_ui,({},ruis,{}),rno)(vp))
109b      ⊓
109(b)i   (if not_yet_at_hub(f)
109(b)i      then
109(b)i1       (let incr = increment(f) in
103             let onl = (tl_ui,h_ui,incr,th_ui) in
109(b)i2        ba−r_ch[l_ui,a_ui] ! onL(onl) ;
109(b)i3        automobile_{a_ui}(a_ui,({},ruis,{}),rno)
109(b)i3                    (onL(onl))
109(b)i         end end)
109(b)ii     else
109(b)ii1      (let nxt_lui:L_UI•nxt_lui ∈ mereo_H(℘(th_ui)) in
109(b)ii2       ba_r_ch[thui,aui]!atH(l_ui,th_ui,nxt_lui) ;
109(b)ii3       automobile_{a_ui}(a_ui,({},ruis,{}),rno)
109(b)ii3                   (atH(l_ui,th_ui,nxt_lui)) end)
109(b)i    end)
109c      ⊓
109d       stop
109(b)i1  increment: Fract → Fract
```

Hubs:

We model the hub behaviour vis-a-vis vehicles: buses and automobiles.

110 The hub behaviour
   a non-deterministically, externally offers
   b to accept timed vehicle positions —
   c which will be at the hub, from some vehicle, v_ui.
   d The timed vehicle hub position is appended to the front of that vehicle's entry in the hub's traffic table;
   e whereupon the hub proceeds as a hub behaviour with the updated hub traffic table.
   f The hub behaviour offers to accept from any vehicle.
   g A **post** condition expresses what is really a **proof obligation**: that the hub traffic, ht′ satisfies the **axiom** of the endurant hub traffic attribute Item 73 Pg. 49.

```
value
110   hub_{h_ui}(h_ui,(,(luis,vuis)),hω)(hσ,ht) ≡
110a     ⊓
110b        { let m = ba_r_ch[h_ui,v_ui] ? in
110c              assert: m=(_,atHub(_,h_ui,_))
110d           let ht′ = ht † [h_ui ↦ ⟨m⟩⌢ht(h_ui)] in
110e           hub_{h_ui}(h_ui,(,(luis,vuis)),(hω))(hσ,ht′)
110f         | v_ui:V_UI•v_ui∈vuis end end }
110g     post: ∀ v_ui:V_UI•v_ui ∈ dom ht′⇒time_ordered(ht′(v_ui))
```

Links:

Similarly we model the link behaviour vis-a-vis vehicles.

111 The link behaviour non-deterministically, externally offers
112 to accept timed vehicle positions —
113 which will be on the link, from some vehicle, v_ui.
114 The timed vehicle link position is appended to the front of that vehicle's entry in the link's traffic table;
115 whereupon the link proceeds as a link behaviour with the updated link traffic table.
116 The link behaviour offers to accept from any vehicle.
117 A **post** condition expresses what is really a **proof obligation**: that the link traffic, lt′ satisfies the **axiom** of the endurant link traffic attribute Item 77 Pg. 49.

```
111   link_{l_ui}(l_ui,(_,(huis,vuis),_),lω)(lσ,lt) ≡
111     ⊓
112       { let m = ba_r_ch[l_ui,v_ui] ? in
113             assert: m=(_,onLink(_,l_ui,_,_))
114          let lt′ = lt † [l_ui ↦ ⟨m⟩⌢lt(l_ui)] in
115          link_{l_ui}(l_ui,(huis,vuis),hω)(hσ,lt′)
116       | v_ui:V_UI•v_ui∈vuis end end }
117     post: ∀ v_ui:V_UI•v_ui ∈ dom lt′⇒time_ordered(lt′(v_ui))
```

Bus Companies:

We model bus companies very rudimentary. Bus companies keep a fleet of buses. Bus companies create, maintain, distribute bus time tables. Bus companies deploy their buses to honor obligations of their bus time tables. We shall basically only model the distribution of bus time tables to buses. We shall not cover other aspects of bus company management, etc.

118 Bus companies non-deterministically, internally, chooses among
   a updating their bus time tables
   b whereupon they resume being bus companies, albeit with a new bus time table;
119 "interleaved" with
   a offering the current time-stamped bus time table to buses which offer willingness to received them
   b whereupon they resume being bus companies with unchanged bus time table.

```
100   bus_company_{bc_ui}(bcui,(_,buis,_))(btt) ≡
118a   (let btt′ = update(btt,...) in
118b    bus_company_{bc_ui}(bcui,(_,buis,_))(btt′) end )
119    ⊓
119a   ( ⊓ {bc_b_ch[bc_ui,b_ui] ! btt | b_ui:B_UI•b_ui∈buis
119b         bus_company_{bc_ui}(bcui,(_,buis,_))(attr_T_ch?,btt) } )
```

Buses:

We model the interface between buses and their owning companies — as well as the interface between buses and the road net, the latter by almost "carbon-copying" all elements of the automobile behaviour(s).

120 The bus behaviour chooses to either
  a accept a (latest) time-stamped buss time table from its bus company –
  b where after it resumes being the bus behaviour now with the updated bus time table.
121 or, non-deterministically, internally,
  a based on the bus position
    i if it is at a hub then it behaves as prescribed in the case of automobiles at a hub,
    ii else, it is on a link, and then it behaves as prescribed in the case of automobiles on a link.

```
120      bus_{b_ui}(b_ui,(_,(bc_ui,ruis),_))(ln,btt,bpos) ≡
120a       (let btt′ = b_bc_ch[b_ui,bc_ui] ? in
120b        bus_{b_ui}(b_ui,({},(bc_ui,ruis),{}))(ln,btt′,bpos) end)
121        ⌈⌉
121a       (case bpos of
121(a)i        atH(fl_ui,h_ui,tl_ui) →
121(a)i          atH_bus_{b_ui}(b_ui,(_,(bc_ui,ruis),_))(ln,btt,bpos),
121(a)ii       aonL(fh_ui,l_ui,f,th_ui) →
121(a)ii         onL_bus_{b_ui}(b_ui,(_,(bc_ui,ruis),_))(ln,btt,bpos)
121a       end)
```

The atH_bus$_{b_{ui}}$ behaviour definition is a simple transcription of the automobile$_{a_{ui}}$ (atH) behaviour definition: mereology expressions being changed from to , programmed attributes being changed from atH(fl_ui,h_ui,tl_ui) to (ln,btt,atH(fl_ui,h_ui,tl_ui)), channel references a_ui being replaced by b_ui, and behaviour invocations renamed from automobile$_{a_{ui}}$ to bus$_{b_{ui}}$. So formula lines 104–109d below presents "nothing new" !

```
121(a)i  atH_bus_{b_ui}(b_ui,(_,(bc_ui,ruis),_))
121(a)i                (ln,btt,atH(fl_ui,h_ui,tl_ui)) ≡
104          (ba_r_ch[b_ui,h_ui] ! (attr_T_ch?,atH(fl_ui,h_ui,tl_ui));
105           bus_{b_ui}(b_ui,({},(bc_ui,ruis),{}))(ln,btt,bpos))
120a         ⌈⌉
106a         (let ({fh_ui,th_ui},ruis′)=mereo_L(℘(tl_ui)) in
106a                assert: fh_ui=h_ui ∧ ruis=ruis′
103           let onl = (tl_ui,h_ui,0,th_ui) in
106b           (ba_r_ch[b_ui,h_ui] ! (attr_T_ch?,onL(onl)) ∥
106b            ba_r_ch[b_ui,tl_ui] ! (attr_T_ch?,onL(onl))) ;
106c           bus_{b_ui}(b_ui,({},(bc_ui,ruis),{}))
106c                   (ln,btt,onL(onl)) end end )
109c         ⌈⌉
109d          stop
```

The onL_bus$_{b_{ui}}$ behaviour definition is a similar simple transcription of the automobile$_{a_{ui}}$ (onL) behaviour definition. So formula lines 104–109d below presents "nothing new" !

122 – this is the "almost last formula line" !

```
121(a)ii  onL_bus_{b_ui}(b_ui,(_,(bc_ui,ruis),_))
121(a)ii                (ln,btt,bpos:onL(fh_ui,l_ui,f,th_ui)) ≡
104          (ba_r_ch[b_ui,h_ui] ! (attr_T_ch?,bpos);
105           bus_{b_ui}(b_ui,({},(bc_ui,ruis),{}))(ln,btt,bpos))
120a          ⌈⌉
109(b)i       (if not_yet_at_hub(f)
109(b)i          then
109(b)i1            (let incr = increment(f) in
103                  let onl = (tl_ui,h_ui,incr,th_ui) in
109(b)i2             ba−r_ch[l_ui,b_ui] ! onL(onl) ;
109(b)i3             bus_{b_ui}(b_ui,({},(bc_ui,ruis),{}))
109(b)i3                    (ln,btt,onL(onl))
109(b)i              end end)
109(b)ii         else
109(b)ii1           (let nl_ui:L_UI•nxt_lui∈mereo_H(℘(th_ui)) in
109(b)ii2            ba_r_ch[thui,b_ui]!atH(l_ui,th_ui,nxt_lui) ;
109(b)ii3            bus_{b_ui}(b_ui,({},(bc_ui,ruis),{}))
109(b)ii3                   (ln,btt,atH(l_ui,h_ui,nxt_lui))
109(b)ii1            end)end)
109c          ⌈⌉
122           stop
```

A Running System – Sect. 1.7.5 pp. 46

Preliminaries:

We recall the ***hub, link, bus company, bus*** and the ***automobile states*** first mentioned in Sect. 1.3.8 Page 48.

```
value
33  hs:H-set ≡ ≡ obs_sH(obs_SH(obs_RN(rts)))
34  ls:L-set ≡ ≡ obs_sL(obs_SL(obs_RN(rts)))
36  bcs:BC-set ≡ obs_BCs(obs_SBC(obs_FV(obs_RN(rts))))
37  bs:B-set ≡ ∪{obs_Bs(bc)|bc:BC•bc ∈ bcs}
39  as:A-set ≡ obs_BCs(obs_SBC(obs_FV(obs_RN(rts))))
```

Starting Initial Behaviours:

We are reaching the end of this domain modeling example. Behind us there are narratives and formalisations 18 Pg. 47 – 122 Pg. 54. Based on these we now express the signature and the body of the definition of a ***"system build and execute"*** function.

123 The system to be initialised is
  a the parallel composition (∥) of
  b the distributed parallel composition (∥{...|...}) of
  c all the hub behaviours,
  d all the link behaviours,
  e all the bus company behaviours,
  f all the bus behaviours, and
  g all the automobile behaviours.

```
value
123   initial_system: Unit → Unit
123   initial_system() ≡
123c     ∥ { hub_{h_ui}(h_ui,me,hω)(htrf,hσ)
123c       | h:H•h ∈ hs,
123c         h_ui:H_UI•h_ui=uid_H(h),
123c         me:HMetL•me=mereo_H(h),
123c         hω:HΩ•hω=attr_HΩ(h),
123c         htrf:H_Traffic•htrf=attr_H_Traffic_H(h),
123c         hσ:HΣ•hσ=attr_HΣ(h)∧hσ ∈ hω
123c       }
123a     ∥
123d     ∥ { link_{l_ui}(l_ui,me,lω)(ltrf,lσ)
123d         l:L•l ∈ ls,
123d         l_ui:L_UI•l_ui=uid_L(l),
123d         me:LMet•me=mereo_L(l),
123d         lω:LΩ•lω=attr_LΩ(l),
123d         ltrf:L_Traffic•ltrf=attr_L_Traffic_H(l),
123d         lσ:LΣ•lσ=attr_LΣ(l)∧lσ ∈ lω
123d       }
123a     ∥
123e     ∥ { bus_company_{bc_ui}(bcui,me)(btt)
123e         bc:BC•bc ∈ bcs,
123e         bc_ui:BC_UI•bc_ui=uid_BC(bc),
123e         me:BCMet•me=mereo_BC(bc),
123e         btt:BusTimTbl•btt=attr_BusTimTbl(bc)
123e       }
123a     ∥
123f     ∥ { bus_{b_ui}(b_ui,me)(ln,btt,bpos)
123f         b:B•b ∈ bs,
123f         b_ui:B_UI•b_ui=uid_B(b),
123f         me:BMet•me=mereo_B(b),
123f         ln:LN:pln=attr_LN(b),
123f         btt:BusTimTbl•btt=attr_BusTimTbl(b),
123f         bpos:BPos•bpos=attr_BPos(b)
123f       }
123a     ∥
123g     ∥ { automobile_{a_ui}(a_ui,me,rn)(apos)
123g         a:A•a ∈ as,
123g         a_ui:A_UI•a_ui=uid_A(a),
123g         me:AMet•me=mereo_A(a),
123g         rn:RegNo•rno=attr_RegNo(a),
123g         apos:APos•apos=attr_APos(a)
123g       }
```

Intentional "Pull"

We illustrate the concept of ***intentional "pull"*** cf. definition on Page 33:

124 ***automobiles*** include the ***intent*** of `'transport'`,
125 and so do ***hubs*** and ***links***.

124 **attr**_Intent: A → (`'transport'`|...)-**set**
125 **attr**_Intent: H → (`'transport'`|...)-**set**
125 **attr**_Intent: L → (`'transport'`|...)-**set**

Manifestations of `'transport'` is reflected in ***automobiles*** having the automobile position attribute, APos, Item 86 Pg. 50, ***hubs*** having the ***hub traffic*** attribute, H_Traffic, Item 73 Pg. 49, and in ***links*** having the ***link traffic*** attribute, L_Traffic, Item 77 Pg. 49.

126 Seen from the point of view of an automobile there is its own traffic history, A_Hist, which is a (time ordered) sequence of timed automobile's positions;
127 seen from the point of view of a hub there is its own traffic history, H_Traffic Item 73 Pg. 49, which is a (time ordered) sequence of timed maps from automobile identities into automobile positions; and
128 seen from the point of view of a link there is its own traffic history, L_Traffic Item 77 Pg. 49, which is a (time ordered) sequence of timed maps from automobile identities into automobile positions.

The ***intentional "pull"*** of these manifestations is this:

129 The union, i.e. proper merge of all automobile traffic histories, AllATH, must now be identical to the same proper merge of all hub, AllHTH, and all link traffic histories, AllLTH.

type
126 A_Hist $= (\mathscr{T} \times$ APos$)^*$
73 H_Trf $=$ A_UI $\overrightarrow{m}$ $(\mathscr{T} \times$ APos$)^*$
77 L_Trf $=$ A_UI $\overrightarrow{m}$ $(\mathscr{T} \times$APos$)^*$
129 AllATH$=\mathscr{T}$ $\overrightarrow{m}$(AUI $\overrightarrow{m}$APos)
129 AllHTH$=\mathscr{T}$ $\overrightarrow{m}$(AUI $\overrightarrow{m}$APos)
129 AllLTH $=\mathscr{T}$ $\overrightarrow{m}$(AUI $\overrightarrow{m}$APos)
axiom
129 **let** allA=mrg_AllATH({(a,attr_A_Hi(a))|a:A•a $\in as$}),
129 allH=mrg_AllHTH({attr_H_Trf(h)|h:H•h $\in hs$}),
129 allL =mrg_AllLTH({attr_L_Trf(l)|l:L•h $\in ls$}) **in**
129 allA = mrg_HLT(allH,allL) **end**

We leave the definition of the four merge functions to the reader !

We now discuss the concept of ***intentional "pull"***. We endow each automobile with its history of timed positions and each hub and link with their histories of timed automobile positions. These histories are facts ! They are not something that is laboriously recorded, where such recordings may be imprecise or cumbersome[47]. The facts are there, so we can (but may not necessarily) talk about these histories as facts. It is in that sense that the purpose (`'transport'`) for which man let automobiles, hubs and link be made with their `'transport'` intent are subject to an **intentional "pull"**. It can be no other way: if automobiles "record" their history, then hubs and links must together "record" identically the same history ! ■

1.9 Closing

1.9.1 Review of Ontology and Type Work

This section is "lifted" from [2].

Analysis & Description Calculi for Other Domains

The analysis and description calculus of this chapter appears suitable for manifest domains. For other domains other calculi may be necessary. There is the introvert, composite domain(s) of systems software: operating systems, compilers, database management systems, Internet-related software, etcetera. The classical computer science and software engineering disciplines related to these components of systems software appears to have provided the necessary analysis and description "calculi." There is the domain of financial systems software accounting & bookkeeping, banking systems, insurance, financial instruments handling (stocks, etc.), etcetera. Etcetera. For each domain characterisable by a distinct set of analysis & description calculus prompts such calculi must be identified.

On Domain Description Languages

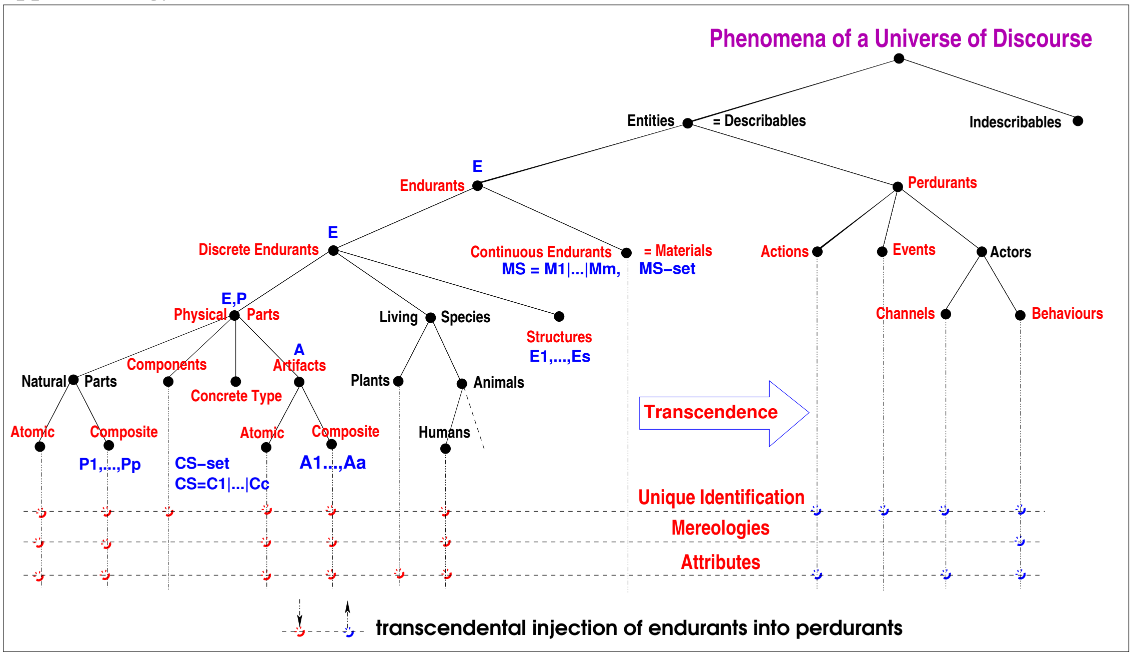
We have in this chapter expressed the domain descriptions in the `RAISE` [64] specification language `RSL` [22]. With what is thought of as minor changes, one can reformulate these domain description texts in either of `Alloy` [65] or `The B-Method` [66] or `VDM` [67, 68, 69] or `Z` [70]. One could also express domain descriptions algebraically, for example in `CafeOBJ` [71, 72]. The analysis and the description prompts remain the same. The description prompts now lead to `Alloy, B-Method, VDM, Z` or `CafeOBJ` texts. We did not go into much detail with respect to perdurants. For all the very many domain descriptions, covered elsewhere, `RSL` (with its `CSP` sub-language) suffices. It is favoured here because of its integrated `CSP` sub-language which both facilitates the 'compilation' of part descriptions into "the dynamics" of parts in terms of `CSP` processes, and the modeling of external attributes in terms of `CSP` process input channels. But there are cases, not documented in this chapter, where, [73], we have conjoined our `RSL` domain descriptions with descriptions in `Petri Nets` [74] or `MSC` [75] (Message Sequence Charts) or `StateCharts` [76].

[47] or thought technologically in-feasible – at least some decades ago!

Comparison to Other Work

Background: The *TripTych* Domain Ontology

We shall now compare the approach of this chapter to a number of techniques and tools that are somehow related — if only by the term 'domain' ! Common to all the "other" approaches is that none of them presents a prompt calculus that help the domain analyser elicit a, or the, domain description. The figure below shows the tree-like structuring of what modern day AI researchers cum ontologists would call ***an upper ontology***.



General

Two related approaches to structuring domain understanding will be reviewed.

0: Ontology Science & Engineering:

Ontologies are *"formal representations of a set of concepts within a domain and the relationships between those concepts"* — expressed usually in some logic. Ontology engineering [77] construct ontologies. Ontology science appears to mainly study structures of ontologies, especially so-called ***upper ontology*** structures, and these studies "waver" between ***philosophy*** and ***information science***[48]. Internet published ontologies usually consists of thousands of logical expressions. These are represented in some, for example, low-level mechanisable form so that they can be interchanged between ontology research groups and processed by various tools. There does not seem to be a concern for "deriving" such ontologies into requirements for software. Usually ontology presentations either start with the presentation of, or makes reference to its reliance on, an *upper ontology*. The term 'ontology' has been much used in connection with automating the design of various aspects WWW applications [78]. Description Logic [79] has been proposed as a language for the Semantic Web [80].

The interplay between endurants and perdurants is studied in [81]. That study investigates axiom systems for two ontologies. One for endurants (`SPAN`), another for perdurants (`SNAP`). No examples of descriptions of specific domains are, however, given, and thus no specific techniques nor tools are given, method components which could help the engineer in constructing specific domain descriptions. [81] is therefore only relevant to the current chapter insofar as it justifies our emphasis on endurant versus perdurant entities.The interplay between endurant and perdurant entities and their qualities is studied in [82]. In our study

[48] We take the liberty of regarding information science as part of ***computer science***.

the term ***quality*** is made specific and covers the ideas of external and internal qualities. External qualities focus on whether endurant or perdurant, whether part, component or material, whether action, event or behaviour, whether atomic or composite part, etcetera. Internal qualities focus on unique identifiers (of parts), the mereology (of parts), and the attributes (of parts, components and materials), that is, of endurants. In [82] the relationship between universals (types), particulars (values of types) and qualities is not "restricted" as in the `TripTych` domain analysis, but is axiomatically interwoven in an almost "recursive" manner. Values [of types ('quantities' [of 'qualities'])] are, for example, seen as sub-ordinated types; this is an ontological distinction that we do not make. The concern of [82] is also the relations between qualities and both endurant and perdurant entities, where we have yet to focus on "qualities", other than signatures, of perdurants. [82] investigates the quality/quantity issue wrt. endurance/perdurance and poses the questions: [b] are non-persisting quality instances enduring, perduring or neither ? and [c] are persisting quality instances enduring, perduring or neither ? and arrives, after some analysis of the endurance/perdurance concepts, at the answers: [b′] non-persisting quality instances are neither enduring nor perduring particulars (i.e., entities), and [c′] persisting quality instances are enduring particulars. Answer [b′] justifies our separating enduring and perduring entities into two disjoint, but jointly "exhaustive" ontologies. The more general study of [82] is therefore really not relevant to our prompt calculi, in which we do not speculate on more abstract, conceptual qualities, but settle on external endurant qualities, on the ***unique identifier***, ***mereology*** and ***attribute*** qualities of endurants, and the simple relations between endurants and perdurants, specifically in the relations between ***signature***s of actions, events and behaviours and the endurant sorts, and especially the relation between parts and behaviours. That is, the `TripTych` approach to ontology, i.e., its domain concept, is not only model-theoretic, but, we risk to say, radically different. The concerns of `TripTych` domain science & engineering is based on that of algorithmic engineering. The domains to which we are applying our analysis & description tools and techniques are spatio-temporal, that is, can be observed, physically; this is in contrast to such conceptual domains as various branches of mathematics, physics, biology, etcetera. Domain science & engineering is not aimed at letting the computer solve problems based on the knowledge it may have stored. Instead it builds models based on knowledge of, but not "in" the domain. The `TripTych` form of domain science & engineering differs from conventional *ontological engineering* in the following, essential ways: The `TripTych` domain descriptions rely essentially on a "built-in" *upper ontology*: types, abstract as well as model-oriented (i.e., concrete) and actions, events and behaviours. Domain science & engineering is not, to a first degree, concerned with modalities, and hence do not focus on the modeling of knowledge and belief, necessity and possibility, i.e., alethic modalities, epistemic modality (certainty), promise and obligation (deontic modalities), etcetera.

The `TripTych` emphasis is on the method for constructing descriptions. It seems that publications on ontological engineering, in contrast, emphasise the resulting ontologies. The papers on ontologies are almost exclusively ***computer science*** (i.e., ***information science***) than ***computing science*** papers.

The next section overlaps with the present section.

1: Knowledge Engineering:

The concept of *knowledge* has occupied philosophers since Plato. No common agreement on what 'knowledge' is has been reached. From [37, 43, 83, 84] we may learn that *knowledge is a familiarity with someone or something; it can include facts, information, descriptions, or skills acquired through experience or education; it can refer to the theoretical or practical understanding of a subject; knowledge is produced by socio-cognitive aggregates (mainly humans) and is structured according to our understanding of how human reasoning and logic works.* The seminal reference here is [85]. The aim of *knowledge engineering* was formulated, in 1983, by an originator of the concept, Edward A. Feigenbaum [86] *knowledge engineering* is an engineering discipline that involves integrating knowledge into computer systems in order to solve complex problems normally requiring a high level of human expertise. *Knowledge engineering* focus on continually building up (acquire) large, shared data bases (i.e., *knowledge bases*), their continued maintenance, testing the validity of the stored 'knowledge', continued experiments with respect to *knowledge representation*, etcetera. *Knowledge engineering* can, perhaps, best be understood in contrast to *algorithmic engineering*: In the latter we seek more-or-less conventional, usually *imperative programming language* expressions of algorithms *whose algorithmic structure embodies the knowledge required to solve the problem being solved by the algorithm.* The former seeks to solve problems based on an interpreter inferring possible solutions from logical data. This logical data has three parts: *a collection that "mimics" the semantics of, say, the imperative programming language, a collection that formulates the problem, and a collection that constitutes the knowledge particular to the problem.* We refer to [87]. Domain science & engineering is not aimed at letting the computer solve problems based on the knowledge it may have stored. Instead it builds models based on knowledge of the domain. Finally, the domains to which we are applying 'our form of' domain analysis are domains which focus on spatio-temporal phenomena. That is, domains which have concrete renditions: air traffic, banks, container lines, manufacturing, pipelines, railways, road transport, stock exchanges, etcetera. In contrast one may claim that the domains described in classical ontologies and knowledge representations are mostly conceptual: mathematics, physics, biology, etcetera.

Specific

2: Database Analysis:

There are different, however related "schools of database analysis". `DSD`: the Bachman (or data structure) diagram model [88]; `RDM`: the relational data model [89]; and `ER`: entity set relationshp model [90] "schools". `DSD` and `ER` aim at graphically specifying database structures. Codd's `RDM` simplifies the data models of `DSD` and `ER` while offering two kinds of languages with which to operate on `RDM` databases: `SQL` and `Relational Algebra`. All three "schools" are focused more on data modeling for databases than on domain modeling both endurant and perdurant entities.

3: Domain Analysis:

Domain analysis, or *product line analysis* (see below), as it was then conceived in the early 1980s by James Neighbors [91], is the analysis of related software systems in a domain to find their common and variable parts. This form of domain analysis turns matters "upside-down": it is the set of software "systems" (or packages) that is subject to some form of inquiry, albeit having some domain in mind, in order to find common features of the software that can be said to represent a named domain.

In this section we shall mainly be comparing the `TripTych` approach to domain analysis to that of Reubén Prieto-Dĩaz's approach [92, 93, 94]. Firstly, our understanding of *domain analysis* basically coincides with Prieto-Dĩaz's. Secondly, in, for example, [92], Prieto-Dĩaz's domain analysis is focused on the very important stages that precede the kind of *domain modeling* that we have described: major concerns are *selection of what appears to be similar, but specific entities, identification of common features, abstraction of entities and classification. Selection* and *identification* is assumed in our approach, but we suggest to follow the ideas of Prieto-Dĩaz. *Abstraction* (from values to types and signatures) and *classification* into parts, materials, actions, events and behaviours is what we have focused on. All-in-all we find Prieto-Dĩaz's work very relevant to our work: relating to it by providing guidance to pre-modeling steps, thereby emphasising issues that are necessarily informal, yet difficult to get started on by most software engineers. Where we might differ is on the following: although Prieto-Dĩaz does mention a need for *domain specific languages*, he does not show examples of *domain descriptions* in such `DSL`s. We, of course, basically use mathematics as the `DSL`. In our approach we do not consider requirements, let alone software components, as do Prieto-Dĩaz, but we find that that is not an important issue.

4: Domain Specific Languages:

Martin Fowler[49] defines a *Domain-specific language* (`DSL`) *as a computer programming language of limited expressiveness focused on a particular domain* [95]. Other references are [96, 97]. Common to [97, 96, 95] is that they define a domain in terms of classes of software packages; that they never really "derive" the `DSL` from a description of the domain; and that they certainly do not describe the domain in terms of that `DSL`, for example, by formalising the `DSL`. In [98] a domain specific language for railway tracks is the basis for verification of the monitoring and control of train traffic on these tracks. Specifications in that domain

[49] **http://martinfowler.com/dsl.html**

specific language, DSL, manifested by track layout drawings and signal interlocking tables, are translated into SystemC [99]. [98] thus takes one very specific DSL and shows how to (informally) translate their "programs", which are not "directly executable", and hence does not satisfy Fowler's definition of DSLs, into executable programs. [98] is a great paper, but it is not solving our problem, that of systematically describing any manifest domain. [98] does, however, point a way to search for — say graphical — DSLs and the possible translation of their programs into executable ones.

5: Feature-oriented Domain Analysis (*FODA*):

Feature oriented domain analysis (FODA) is a domain analysis method which introduced feature modeling to domain engineering. FODA was developed in 1990 following several U.S. Government research projects. Its concepts have been regarded as "critically advancing software engineering and software reuse." The US Government–supported report [100] states: *"FODA is a necessary first step"* for software reuse. To the extent that TripTych *domain engineering* with its subsequent *requirements engineering* indeed encourages reuse at all levels: *domain descriptions* and *requirements prescription*, we can only agree. Another source on FODA is [101]. Since FODA "leans" quite heavily on 'Software Product Line Engineering' our remarks in that section, next, apply equally well here.

6: Software Product Line Engineering:

Software product line engineering, earlier known as domain engineering, is the entire process of *reusing domain knowledge* in the production of new software systems. Key concerns of software product line engineering are *reuse*, the building of repositories of *reusable software components*, and *domain specific languages* with which to more-or-less automatically build software based on *reusable software components*. These are not the primary concerns of TripTych *domain science & engineering*. But they do become concerns as we move from *domain descriptions* to *requirements prescriptions*. But it strongly seems that *software product line engineering* is not really focused on the concerns of *domain description* — such as is TripTych *domain engineering*. It seems that *software product line engineering* is primarily based, as is, for example, FODA: Feature-oriented Domain Analysis, on analysing features of software systems. Our [102] puts the ideas of *software product lines* and *model-oriented software development* in the context of the TripTych approach.

7: Problem Frames:

The concept of *problem frames* is covered in [103] Jackson's prescription for software development focus on the "triple development" of descriptions of the *problem world*, the *requirements* and the *machine* (i.e., the *hardware* and *software*) to be built. Here *domain analysis* means the same as for us: the *problem world analysis*. In the *problem frame* approach the software developer plays three, that is, all the TripTych rôles: *domain engineer*, *requirements engineer* and *software engineer*, "all at the same time", iterating between these rôles repeatedly. So, perhaps belabouring the point, *domain engineering* is done only to the extent needed by the prescription of *requirements* and the *design* of *software*. These, really are minor points. But in "restricting" oneself to consider only those aspects of the domain which are mandated by the *requirements prescription* and *software design* one is considering a potentially smaller fragment [104] of the domain than is suggested by the TripTych approach. At the same time one is, however, sure to consider aspects of the domain that might have been overlooked when pursuing *domain description development* in the "more general" TripTych approach.

8: Domain Specific Software Architectures (*DSSA*):

It seems that the concept of DSSA was formulated by a group of ARPA[50] project "seekers" who also performed a year long study (from around early-mid 1990s); key members of the DSSA project were Will Tracz, Bob Balzer, Rick Hayes-Roth and Richard Platek [105]. The [105] definition of *domain engineering* is *"the process of creating a DSSA: domain analysis and domain modeling followed by creating a software architecture and populating it with software components."* This definition is basically followed also by

[50] ARPA: The US DoD Advanced Research Projects Agency

[106, 107, 108]. Defined and pursued this way, DSSA appears, notably in these latter references, to start with the analysis of software components, "per domain", to identify commonalities within application software, and to then base the idea of *software architecture* on these findings. Thus DSSA turns matter "upside-down" with respect to TripTych *requirements development* by starting with *software components*, assuming that these satisfy some *requirements*, and then suggesting *domain specific software* built using these components. This is not what we are doing: we suggest, [10], that *requirements* can be "derived" systematically from, and formally related back to *domain descriptionss* without, in principle, considering *software components*, whether already existing, or being subsequently developed. Of course, given a *domain description* it is obvious that one can develop, from it, any number of *requirements prescriptions* and that these may strongly hint at shared, (to be) implemented *software components*; but it may also, as well, be the case that two or more *requirements prescriptions* "derived" from the same *domain description* may share no *software components* whatsoever ! It seems to this author that had the DSSA promoters based their studies and practice on also using formal specifications, at all levels of their study and practice, then some very interesting insights might have arisen.

9: Domain Driven Design (*DDD*):

Domain-driven design (DDD)[51] *"is an approach to developing software for complex needs by deeply connecting the implementation to an evolving model of the core business concepts; the premise of domain-driven design is the following: placing the project's primary focus on the core domain and domain logic; basing complex designs on a model; initiating a creative collaboration between technical and domain experts to iteratively cut ever closer to the conceptual heart of the problem."*[52] We have studied some of the DDD literature, mostly only accessible on the Internet, but see also [109], and find that it really does not contribute to new insight into *domains* such as we see them: it is just "plain, good old software engineering cooked up with a new jargon.

10: Unified Modeling Language (*UML*):

Three books representative of UML are [110, 111, 112]. jacobson@Ivar Jacobson The term *domain analysis* appears numerous times in these books, yet there is no clear, definitive understanding of whether it, the *domain*, stands for entities in the domain such as we understand it, or whether it is wrought up, as in several of the 'approaches' treated in this section, to wit, in items [3–5, 7–9] with either *software design* (as it most often is), or *requirements prescription*. Certainly, in UML, in [110, 111, 112] jacobson@Ivar Jacobsons well as in most published papers claiming "adherence" to UML, that domain analysis usuallyis manifested in some UML text which "models" some *requirements* facet. Nothing is necessarily wrong with that, but it is therefore not really the TripTych form of *domain analysis* with its concepts of abstract representations of endurant and perdurants, with its distinctions between *domain* and *requirements*, and with its possibility of "deriving" *requirements prescriptions* from *domain descriptions*. The UML notion of *class diagrams* is worth relating to our structuring of the domain. Class diagrams appear to be inspired by [88, Bachman, 1969] and [90, Chen, 1976]. It seems that (i) each part sort — as well as other than part sorts — deserves a class diagram (box); and (ii) that (assignable) attributes — as well as other non-part types — are written into the diagram box. Class diagram boxes are line-connected with annotations where some annotations are as per the mereology of the part type and the connected part types and others are not part related. The class diagrams are said to be object-oriented but it is not clear how objects relate to parts as many are rather implementation-oriented quantities. All this needs looking into a bit more, for those who care.

11: Requirements Engineering:

There are in-numerous books and published papers on *requirements engineering*. A seminal one is [113]. I, myself, find [114] full of very useful, non-trivial insight. [115] is seminal in that it brings a number or early contributions and views on *requirements engineering*. Conventional text books, notably [116, 117, 118]

[51] Eric Evans: http://www.domaindrivendesign.org/
[52] http://en.wikipedia.org/wiki/Domain-driven_design

all have their "mandatory", yet conventional coverage of *requirements engineering*. None of them "derive" requirements from domain descriptions, yes, OK, from domains, but since their description is not mandated it is unclear what "the domain" is. Most of them repeatedly refer to *domain analysis* but since a written record of that *domain analysis* is not mandated it is unclear what "domain analysis" really amounts to. Axel van Laamsweerde's book [113]s remarkable. Although also it does not mandate descriptions of domains it is quite precise as to the relationships between domains and requirements. Besides, it has a fine treatment of the distinction between *goal*s and *requirements*, also formally. Most of the advices given in [114] can beneficially be followed also in `TripTych` *requirements development*. Neither [113]or [114] preempts `TripTych` *requirements development*.

Summary of Comparisons

We find that there are two kinds of relevant comparisons: the concept of ontology, its science more than its engineering, and the *Problem Frame* work of Michael A. Jackson.

Of all the other "comparison" items ([2]–[12]) basically only Jackson's *problem frames* (Item [8]) and [98] (Item [5]) really take the same view of *domain*s and, in essence, basically maintain similar relations between *requirements prescription* and *domain description*. So potential sources of, we should claim, mutual inspiration ought be found in one-another's work — with, for example, [119, 104, 98], and the present document, being a good starting point.

But none of the referenced works make the distinction between discrete endurants (parts) and their qualities, with their further distinctions between *unique identifiers*, *mereology* and *attributes*. And none of them makes the distinction between *parts*, *components* and *materials*. Therefore our contribution can include the mapping of parts into behaviours interacting as per the part mereologies as highlighted in our ***process schema***s.

1.9.2 What Have We Achieved ?

A step-wise ***method***, its ***principles***, ***techniques***, and a series of ***languages*** for the rigorous development of domain models has been presented. A seemingly large number of domain concepts has been established: ***entities***, ***endurants*** and ***perdurants***, ***discrete*** and ***continuous*** endurants, ***structure, part, component*** and ***material*** endurants, ***living species, plants, animals, humans*** and ***artifacts***, ***unique identifiers, mereology*** and ***attributes***.

A concept of ***transcendental deduction*** has been introduced. It is used to justify the interpretation of ***endurant parts*** as ***perdurant behaviours*** – a la `CSP`. A new concept of ***intentional "pull"*** has been introduced. It applies, in the form of attributes, to humans and artifacts. It "corresponds", in a way, to ***gravitational pull***; that concept invites further study. The pair of gravitational pull and intentional "pull" appears to lie behind the determination of the mereologies of parts; that possibility invites further study.

Finally it is shown how `CSP` ***channels*** can be calculated from endurant mereologies, and how the form of ***behaviour arguments*** can be calculated from respective attribute categorisations.

The domain concepts outlined above form a ***domain ontology*** that applies to a wide variety of domains. An example, Sect. 1.8, is tied, section-by-section to the unfolding of the method and the domain ontology.

1.9.3 Issues of Philosophy

Three issues of philosophy are of concern here: the "nature" of the definition of the analysis prompts; the ***transcendental deduction*** whereby ***parts*** are interpreted as ***behaviours;*** and the ***intentional "pull"*** whereby seemingly "unrelated" parts are indeed "related" ! They all relate to ***what can be described***.

What Can Be Described

As for the first, consider the analysis prompts:

When you read the texts that explain when phenomena can be considered entities, entities can be considered endurants or perdurants, endurants can be considered discrete or continuous, discrete endurants can be considered structures, parts or components, et cetera, then you probably, expecting to read a technical/scientific paper, realise that those explanations are not precise in the sense of such papers.

Many of our definitions are taken from [37, The Oxford Shorter English Dictionary] and from the Internet based [120, The Stanford Encyclopedia of Philosophy].

In technical/scientific papers definitions are expected to be precise, but can be that only if the definer has set up, beforehand, or the reported work is based on a precise, in our case mathematical framework. That can not be done here. There is no, a priori given, model of the domains we are interested in.

This raises the more general question, such as we see it: ***"which are the absolutely necessary and unavoidable bases for describing the world ?"*** This is a question of philosophy. We shall not develop the reasoning here. Instead we refer to the forthcoming [13, Philosophical Issues in Domain Modeling]. That work is based on [17, 18, 19, 20].

The Transcendental Deduction

The interpretation of ***endurant parts*** as ***perdurant behaviours*** represents a ***transcendental deduction*** – and must, somehow, be rationally justified. the justification is here seen as exactly that: a ***transcendental deduction*** It seems that transcendental deductions abound: when compiling program texts into machine code, in transitions from syntax to semantics to pragmatics, and in any abstract interpretation of formal texts. We refer Chapter 7 and to the forthcoming revision of [13, Philosophical Issues in Domain Modeling].

The Intentional "Pull"

This last concept is merely a suggestion. A serious paper cannot solve all issues.

1.9.4 Two Frequently Asked Questions

How much of a* DOMAIN *must or should we* ANALYSE *&* DESCRIBE *? When this question is raised, after a talk of mine over the subject, and by a colleague researcher & scientist I usually reply: *As large a domain as possible !* This reply is often met by this ***comment*** (from the audience) ***Oh ! No, that is not reasonable !*** To me that comment shows either or both of: the questioner was not asking as a researcher/scientist, but as an engineer. Yes, an engineer needs only analyse & describe up to and slightly beyond the "border" of the domain-of-interest for a current software development – but a researcher cum scientist is, of course, interested not only in a possible requirements engineering phase beyond domain engineering, but is also curious about the larger context of the domain, in possibly establishing a proper domain theory, etc.

How, then, should a domain engineer pursue* DOMAIN MODELING *? My answer assumes a "state-of-affairs" of domain science & engineering in which domain modeling is an established subject, i.e., where the domain analysis & description topic, i.e., its methodology, is taught, where there are "text-book" examples from relevant fields – that the domain engineers can rely on, and in whose terminology they can communicate with one another; that is, there is an acknowledged ***body of knowledge***. My answer is therefore: the domain engineer, referring to the relevant ***body of knowledge***, develops a domain model that covers the domain and the context on which the software is to function, just, perhaps covering a little bit more of the context, than possibly necessary — just to be sure. Until such a "state-of-affairs" is reached the domain model developer has to act both as a domain scientist and as a domain engineer, researching and developing models for rather larger domains than perhaps necessary while contributing also to the **domain science & engineering body of knowledge**.

1.9.5 On How to Pursue Domain Science & Engineering

We set up a dogma and discuss a ramification. One thing is the doctrine, the method for **domain analysis & description** outlined in this paper. Another thing is its practice. I find myself, when experimentally pursuing the modeling of domains, as, for example, reported in [24, 25, 26, 121, 122, 123, 30, 27, 124, 36, 33, 35, 29], **not following the doctrine !** That is: (i) in not first, carefully, exploring parts, components and materials, the external properties, (ii) in not then, again carefully settling issues of unique identifiers, (iii) then, carefully, the issues of mereology, (iv) followed by careful consideration of attributes, then the transcendental deduction of behaviours from parts; (v) carefully establishing channels: (v.i) their message types, and (v.ii) declarations, (vi) followed by the careful consideration of behaviour signatures, systematically, one for each transcendentally deduced part, (vii) then the careful definition of each of all the deduced behaviours, and, finally, (iix) the definition of the overall system initialisation. No, instead I faulter, get diverted into exploring ***"this & that"*** in the domain exploration. And I get stuck. When despairing I realise that I must ***"slavically"*** follow the doctrine. When reverting to the strict adherence of the doctrine, I find that I, very quickly, find my way, and the domain modeling get's ***unstruck*** ! I remarked this situation to a dear friend and colleague, Dr. Ole N. Oest. His remark stressed what was going on: the **creative** engineer **took possession**, the **exploring**, sometimes **sceptic** scientist **entered the picture,** the well-trained engineer **lost ground in the realm of imagination**. But perhaps, in the interest of **innovation etc.** it is necessary to be **creative** and **sceptic** and **loose ground** – for a while ! I knew that, but had sort-of-forgotten it ! ***I thank Ole N. Oest for this observation.***

1.9.6 Related Work

The present chapter is but one in a series on the topic of ***domain science & engineering***. With this chapter the author expects to have laid a foundation. With the many experimental case studies, referenced in Example 1.1 on Page 14, the author seriously think that reasonably convincing arguments are given for this ***domain science & engineering***. We comment on some previous publications: [4, 3] explores additional views on analysing & describing domains, in terms of ***domain facets: intrinsics***, ***support technologies***, ***rules & regulations***, ***scripts***, ***management & organisation,*** and ***human behaviour***. [8, 7] explores relations between Stanisław Leśhnieiski's mereology and ours. [10, 9] shows how to rigorously transform domain descriptions into software system requirements prescriptions. [11] discusses various interpretations of domain models: as bases for demos, simulators, real system monitors and real system monitor & controllers. [125] is a compendium of reports around the management and engineering of software development based in domain analysis & description. These reports were the result of a year at JAIST: Japan Institute of Science & Technology, Ishikawa, Japan.

1.9.7 Tony Hoare's Summary on 'Domain Modeling'

In a 2006 e-mail, in response, undoubtedly to my steadfast – perhaps conceived as stubborn – insistence, on domain engineering, Tony Hoare summed up his reaction to domain engineering as follows, and I quote[53]:

"There are many unique contributions that can be made by domain modeling.

1 *The models describe all aspects of the real world that are relevant for any good software design in the area. They describe possible places to define the system boundary for any particular project.*

[53] E-Mail to Dines Bjørner, July 19, 2006

2 *They make explicit the preconditions about the real world that have to be made in any embedded software design, especially one that is going to be formally proved.*
3 *They describe the whole range of possible designs for the software, and the whole range of technologies available for its realisation.*
4 *They provide a framework for a full analysis of requirements, which is wholly independent of the technology of implementation.*
5 *They enumerate and analyse the decisions that must be taken earlier or later in any design project, and identify those that are independent and those that conflict. Late discovery of feature interactions can be avoided.*"

2

Domain Facets: Analysis & Description

2.1 Introduction

In Chapter 1 [1] we outlined a ***method*** for analysing &[1] describing ***domain***s. By a **method** we shall understand a set of ***principles***, ***techniques*** and ***tools*** for analysing and constructing (synthesizing) an artifact, as here a description ■[2] By a **domain** we shall understand a **rationally describable** segment of a **human assisted** reality, i.e., of the world, its **physical parts**, and **living species**. These are **endurants** ("still"), existing in space, as well as **perdurants** ("alive"), existing also in time. Emphasis is placed on **"human-assisted-ness"**, that is, that there is ***at least one (man-made)*** **artifact** and that **humans** are a primary cause for change of endurant **states** as well as perdurant **behaviours** ■ In this chapter we cover domain analysis & description principles and techniques not covered in Chapter 1. That chapter focused on ***manifest domain***s. Here we, on one side, go "outside" the realm of ***manifest domain***s, and, on the other side, cover, what we shall refer to as, ***facet***s, also not covered in [1].

2.1.1 Facets of Domains

By a **domain facet** we shall understand *one amongst a finite set of generic ways of analysing a domain: a view of the domain, such that the different facets cover conceptually different views, and such that these views together cover the domain* ■ Now, the definition of what a ***domain facet*** is can seem vague. It cannot be otherwise. The definition is sharpened by the definitions of the specific facets. You can say, that the definition of ***domain facet*** is the "sum" of the definitions of these specific facets. The specific facets – so far[3] – are: ***intrinsics*** (Sect. 2.2), ***support technology*** (Sect. 2.3), ***rules & regulations*** (Sect. 2.4), ***scripts*** (Sect. 2.5), ***license languages*** (Sect. 2.6), ***management & organisation*** (Sect. 2.7) and ***human behaviour*** (Sect. 2.8). Of these, the ***rules & regulations***, ***scripts*** and ***license languages*** are closely related. Vagueness may "pop up", here and there, in the delineation of facets. It is necessarily so. We are not in a domain of computer science, let alone mathematics, where we can just define ourselves precisely out of any vagueness problems. We are in the domain of (usually) really world facts. And these are often hard to encircle.

2.1.2 Relation to Previous Work

The present chapter (and hence [3]) is a rather complete rewrite of [4]. The reason for the rewriting is the expected publication of [1]. The [4] was finalised already in 2006, 10 years ago, before the analysis & description calculus of [1] had emerged. It was time to revise [4] rather substantially.

---

[1] We use the ampersand (logogram), &, in the following sense: Let A and B be two concepts. By *A and B* we mean to refer to these two concepts. With $A\&B$ we mean to refer to a composite concept "containing" elements of both A and B.

[2] The ■ symbol delimits a definition.

[3] We write: 'so far' in order to "announce", or hint that there may be other specific facets. The one listed are the ones we have been able to "isolate", to identify, in the most recent 10-12 years.

2.1.3 Structure of Chapter

The structure of the chapter follows the seven specific facets, as listed above. Each section, 2.2.–2.8., starts by a definition of the ***specific facet***, Then follows an analysis of the abstract concepts involved usually with one or more examples – with these examples making up most of the section. We then "speculate" on derivable requirements thus relating the present chapter to [9]. We close each of the sections, 2.2.–2.8., with some comments on how to model the specific facet of that section.

•••

Examples 1–22 of sections 2.2.–2.8. present quite a variety. In that, they reflect the wide spectrum of facets.

•••

More generally, domains can be characterised by intrinsically being endurant, or function, or event, or behaviour ***intensive***. Software support for activities in such domains then typically amount to database systems, computation-bound systems, real-time embedded systems, respectively distributed process monitoring and control systems. Other than this brief discourse we shall not cover the "intensity"-aspect of domains in this chapter.

2.2 Intrinsics

- *By domain* **intrinsics** *we shall understand those phenomena and concepts of a domain which are basic to any of the other facets (listed earlier and treated, in some detail, below), with such domain intrinsics initially covering at least one specific, hence named, stakeholder view* ■

2.2.1 Conceptual Analysis

The principles and techniques of domain analysis & description, as unfolded in [1], focused on and resulted in descriptions of the intrinsics of domains. They did so in focusing the analysis (and hence the description) on the basic endurants and their related perdurants, that is, on those parts that most readily present themselves for observation, analysis & description.

Example 1 Railway Net Intrinsics: *We narrate and formalise three railway net intrinsics.*

From the view of potential train passengers a railway net consists of lines, l:L, with names, ln:Ln, stations, s:S, with names sn:Sn, and trains, tn:TN, with names tnm:Tnm. A line connects exactly two distinct stations.

```
scheme N0 =
  class
    type
      N, L, S, Sn, Ln, TN, Tnm
    value
      obs_Ls: N → L-set, obs_Ss: N → S-set
      obs_Ln: L → Ln, obs_Sn: S → Sn
      obs_Sns: L → Sn-set, obs_Lns: S → Ln-set
    axiom
      ...
  end
```

N, L, S, Sn and Ln designate nets, lines, stations, station names and line names. One can observe lines and stations from nets, line and station names from lines and stations, pair sets of station names from lines, and lines names (of lines) into and out from a station from stations. Axioms ensure proper graph properties of these concepts.

From the view of *actual train passengers* a railway net — in addition to the above — allows for several lines between any pair of stations and, within stations, provides for one or more platform tracks, tr:Tr, with names, trn:Trn, from which to embark on or alight from a train.

```
scheme N1 = extend N0 with
   class
      type
         Tr, Trn
      value
         obs_Trs: S → Tr-set, obs_Trn: Tr → Trn
      axiom
         ...
   end
```

The only additions are that of track and track name types, related observer functions and axioms.

From the view of *train operating staff* a railway net — in addition to the above — has lines and stations consisting of suitably connected rail units. A rail unit is either a simple (i.e., linear, straight) unit, or is a switch unit, or is a simple crossover unit, or is a switchable crossover unit, etc. Simple units have two connectors. Switch units have three connectors. Simple and switchable crossover units have four connectors. A path, p:P, (through a unit) is a pair of connectors of that unit. A state, $\sigma : \Sigma$, of a unit is the set of paths, in the direction of which a train may travel. A (current) state may be empty: The unit is closed for traffic. A unit can be in any one of a number of states of its state space, $\omega : \Omega$.

```
scheme N2 = extend N1 with
   class
      type
         U, C
         P′ = U × (C×C)
         P = {| p:P′ • let (u,(c,c′))=p in (c,c′)∈ ∪ obs_Ω(u) end |}
         Σ = P-set
         Ω = Σ-set
      value
         obs_Us: (N|L|S) → U-set
         obs_Cs: U → C-set
         obs_Σ: U → Σ
         obs_Ω: U → Ω
      axiom
         ...
   end
```

Unit and connector types have been added as have concrete types for paths, unit states, unit state spaces and related observer functions, including unit state and unit state space observers. ■

Different stakeholder perspectives, not only of intrinsics, as here, but of any facet, lead to a number of different models. The name of a phenomenon of one perspective, that is, of one model, may coincide with the name of a "similar" phenomenon of another perspective, that is, of another model, and so on. If the intention is that the "same" names cover comparable phenomena, then the developer must state the comparison relation.

Example 2 **Intrinsics of Switches:** *The intrinsic attribute of a rail switch is that it can take on a number of states. A simple switch ($^{c_|}Y^{c_/}_{c}$) has three connectors: $\{c, c_|, c_/\}$. c is the connector of the common rail from which one can either "go straight" $c_|$, or "fork" $c_/$ (Fig. 2.1). So we have that a possible state space of such a switch could be ω_{g_s} :*

$$\{\{\},$$
$$\{(c,c_|)\},\{(c_|,c)\},\{(c,c_|),(c_|,c)\},$$
$$\{(c,c_/)\},\{(c_/,c)\},\{(c,c_/),(c_/,c)\},\{(c_/,c),(c_|,c)\},$$
$$\{(c,c_|),(c_|,c),(c_/,c)\},\{(c,c_/),(c_/,c),(c_|,c)\},\{(c_/,c),(c,c_|)\},\{(c,c_/),(c_|,c)\}\}$$

The above models a general switch ideally. Any particular switch ω_{p_s} may have $\omega_{p_s} \subset \omega_{g_s}$. Nothing is said about how a state is determined: who sets and resets it, whether determined solely by the physical position

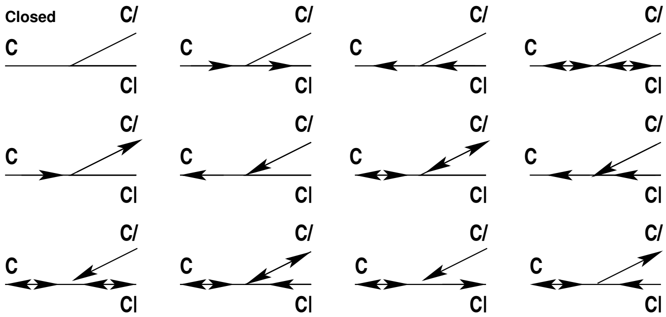


Fig. 2.1. Possible states of a rail switch

of the switch gear, or also by visible or virtual (i.e., invisible, intangible) signals up or down the rail, away from the switch. ■

Example 3 **An Intrinsics of Documents:** *Think of documents, written, by hand, or typed "onto" a computer text processing system. One way of considering such documents is as follows. First we abstract from the syntax that such a document, or set of more-or-less related documents, or just documents, may have: whether they are letters, with sender and receive addressees, dates written, sent and/or received, opening and closing paragraphs, etc., etc.; or they are books, technical, scientific, novels, or otherwise, or they are application forms, tax returns, patient medical records, or otherwise. Then we focus on the operations that one may perform on documents: their creation, editing, reading, copying, authorisation, "transfer"*[4]*, "freezing"*[5]*, and shredding. Finally we consider documents as manifest parts, cf. [1]. Parts, so documents have unique identifications, in this case, changeable mereology, and a number of attributes. The mereology of a document, d, reflects those other documents upon which a document is based, i.e., refers to, and/or refers to d. Among the attributes of a document we can think of (i) a trace of what has happened to a document, i.e., a trace of all the operations performed on "that" document, since and including creation — with that trace, for example, consisting of time-stamped triples of the essence of the operations, the "actor" of the operation (i.e., the operator), and possibly some abstraction of the locale of the document when operated upon; (ii) a synopsis of what the document text "is all about", (iii) and some "rendition" of the document text.* ■

This view of documents, whether "implementable" or "implemented" or not, is at the basis of our view of license languages (for *digital media*, *health-care* (patient medical record), *documents*, and *transport* (contracts) as that facet is covered in Sect. 2.6.

2.2.2 Requirements

[9] illustrated requirements "derived" from the intrinsics of a road transport system – as outlined in [1]. So this chapter has little to add to the subject of requirements "derived" from intrinsics.

2.2.3 On Modeling Intrinsics

[1] outlined basic principles, techniques and tools for modeling the intrinsics of manifest domains. Modeling the domain intrinsics can often be expressed in property-oriented specification languages (like CafeOBJ [126]), model-oriented specification languages (like Alloy [65], B [66], VDM-SL [67, 68, 69], RSL [22], or Z [70]), event-based languages (like Petri nets or [74] or CSP [23], respectively in process-based specification languages (like MSCs [75], LSCs [127], Statecharts [76], or CSP [23]. An area not well-developed is that of modeling continuous domain phenomena like the dynamics of automobile, train and aircraft movements, flow in pipelines, etc. We refer to [128].

[4] to other editors, readers, etc.
[5] i.e., prevention of future operations

2.3 Support Technologies

- *By a domain* **support technology** *we shall understand ways and means of implementing certain observed phenomena or certain conceived concepts* ■

The "ways and means" may be in the form of "soft technologies": human manpower, see, however, Sect. 2.8, or in the form of "hard" technologies: electro-mechanics, etc. The term 'implementing' is crucial. It is here used in the sense that, $\psi\tau$, which is an 'implementation' of a endurant or perdurant, ϕ, is an ***extension*** of ϕ, with ϕ being an ***abstraction*** of $\psi\tau$. We strive for the extensions to be ***proof theoretic conservative extension***s [129].

2.3.1 Conceptual Analysis

There are [always] basically two approaches the task of analysing & describing the support technology facets of a domain. One either stumbles over it, or one tries to tackle the issue systematically. The "stumbling" approach occurs when one, in the midst of analysing & describing a domain realises that one is tackling something that satisfies the definition of a support technology facet. In the systematic approach to the analysis & description of the support technology facets of a domain one usually starts with a basically intrinsics facet-oriented domain description. We then suggest that the domain engineer "inquires" of every endurant and perdurant whether it is an intrinsic entity or, perhaps a support technology.

Example 4 **Railway Support Technology:** *We give a rough sketch description of possible rail unit switch technologies.*

(i) In "ye olde" days, rail switches were "thrown" by manual labour, i.e., by railway staff assigned to and positioned at switches.

(ii) With the advent of reasonably reliable mechanics, pulleys and levers[6] *and steel wires, switches were made to change state by means of "throwing" levers in a cabin tower located centrally at the station (with the lever then connected through wires etc., to the actual switch).*

(iii) This partial mechanical technology then emerged into electro-mechanics, and cabin tower staff was "reduced" to pushing buttons.

(iv) Today, groups of switches, either from a station arrival point to a station track, or from a station track to a station departure point, are set and reset by means also of electronics, by what is known as interlocking (for example, so that two different routes cannot be open in a station if they cross one another). ■

It must be stressed that Example 4 is just a rough sketch. In a proper narrative description the software (cum domain) engineer must describe, in detail, the subsystem of electronics, electro-mechanics and the human operator interface (buttons, lights, sounds, etc.). An aspect of supporting technology includes recording the state-behaviour in response to external stimuli. We give an example.

Example 5 **Probabilistic Rail Switch Unit State Transitions:** *Figure 2.2 indicates a way of formalising this aspect of a supporting technology. Figure 2.2 intends to model the probabilistic (erroneous and correct) behaviour of a switch when subjected to settings (to switched (s) state) and re-settings (to direct (d) state). A switch may go to the switched state from the direct state when subjected to a switch setting s with probability psd.* ■

Example 6 **Traffic Signals:** *We continue Examples 17, 18, 25 and 33 of [1]. This example should, however, be understandable without reference to [1]. A traffic signal represents a technology in support of visualising hub states (transport net road intersection signaling states) and in effecting state changes.*

130 *A traffic signal, ts:TS, is considered a part with observable hub states and hub state spaces. Hub states and hub state spaces are programmable, respectively static attributes of traffic signals.*

131 *A hub state space, $h\omega$, is a set of hub states such that each current hub state is in that hubs' hub state space.*

[6] https://en.wikipedia.org/wiki/Pulley and http://en.wikipedia.org/wiki/Lever

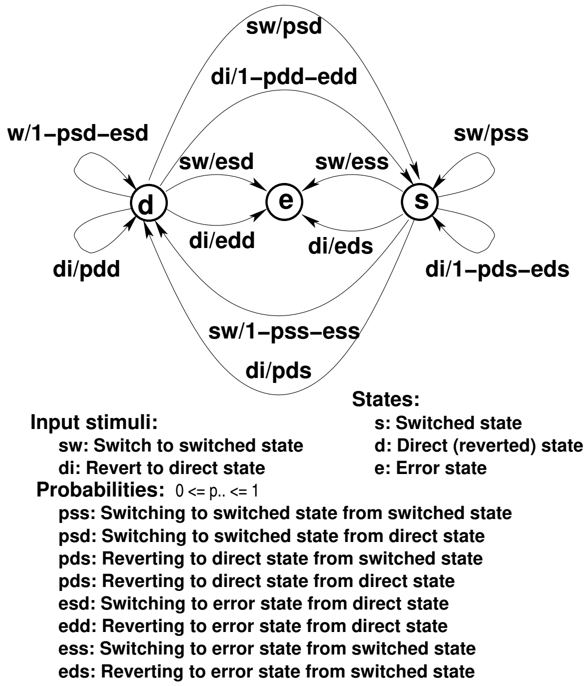


Fig. 2.2. Probabilistic state switching

132 A hub state, $h\sigma$, is now modeled as a set of hub triples.

*133 Each hub triple has a link identifier l_i ("coming from"), a colour (**red**, **yellow** or **green**), and another link identifier l_j ("going to").*

134 Signaling is now a sequence of one or more pairs of next hub states and time intervals, ti:TI, for example: $<(h\sigma_1,ti_1),(h\sigma_2,ti_2),...,(h\sigma_{n-1},ti_{n-1}),(h\sigma_n,ti_n)>$, n>0. The idea of a signaling is to first change the designated hub to state $h\sigma_1$, then wait ti_1 time units, then set the designated hub to state $h\sigma_2$, then wait ti_2 time units, etcetera, ending with final state σ_n and a (supposedly) long time interval ti_n before any decisions are to be made as to another signaling. The set of hub states $\{h\sigma_1,h\sigma_2,...,h\sigma_{n-1}\}$ of $<(h\sigma_1,ti_1),(h\sigma_2,ti_2),...,(h\sigma_{n-1},ti_{n-1}),(h\sigma_n,ti_n)>$, n>0, is called the set of intermediate states. Their purpose is to secure an orderly phase out of green via yellow to red and phase in of red via yellow to green in some order for the various directions. We leave it to the reader to devise proper well-formedness conditions for signaling sequences as they depend on the hub topology.

135 A street signal (a semaphore) is now abstracted as a map from pairs of hub states to signaling sequences. The idea is that given a hub one can observe its semaphore, and given the state, $h\sigma$ (not in the above set), of the hub "to be signaled" and the state $h\sigma_n$ into which that hub is to be signal-led "one looks up" under that pair in the semaphore and obtains the desired signaling.

type
130 TS ≡ H, HΣ, HΩ
value
131 obs_HΣ: H,TS → HΣ
131 obs_HΩ: H,TS → HΩ
type
132 HΣ = Htriple-**set**
132 HΩ = HΣ-**set**
133 Htriple = LI×Colour×LI
axiom
131 ∀ ts:TS • obs_HΣ(ts) ∈ obs_HΩ(ts)
type

133 Colour == **red** | yellow | **green**
134 Signaling = $(\mathsf{H}\Sigma\times\mathsf{TI})^*$
134 TI
135 Sempahore = $(\mathsf{H}\Sigma\times\mathsf{H}\Sigma) \rightarrow_{m}$ Signalling
value
135 obs_Semaphore:TS $\rightarrow$ Sempahore

136 Based on [1] we treat hubs as processes with hub state spaces and semaphores as static attributes and hub states as programmable attributes. We ignore other attributes and input/outputs.
137 We can think of the change of hub states as taking place based the result of some internal, non-deterministic choice.

value
136. hub: HI $\times$ LI-**set** $\times$ $(\mathsf{H}\Omega\times$Semaphore$)$ $\rightarrow$ $\mathsf{H}\Sigma$ **in** ... **out** ... **Unit**
136. hub(hi,lis,(hω,sema))(hσ) $\equiv$
136. ...
137. $\lceil\rceil$ **let** hσ':HI • ... **in** hub(hi,lis,(hω,sema))(signaling(hσ,hσ')) **end**
136. ...
136. **pre**: $\{$hσ,h$\sigma'\}\subseteq$ hω

where we do not bother about the selection of $h\sigma'$.

138 Given two traffic signal, i.e., hub states, $h\sigma_{\text{init}}$ and $h\sigma_{\text{end}}$, where $h\sigma_{\text{init}}$ designates a present hub state and $h\sigma_{\text{end}}$ designates a desired next hub state after signaling.
139 Now *signaling* is a sequence of one or more successful hub state changes.

value
138 signaling: $(\mathsf{H}\Sigma\times\mathsf{H}\Sigma)$ $\times$ Semaphore $\rightarrow$ $\mathsf{H}\Sigma$ $\rightarrow$ $\mathsf{H}\Sigma$
139 signaling(hσ_{init},hσ_{end},sema)(hσ) $\equiv$ **let** sg = sema(hσ_{init},hσ_{end}) **in** signal_sequence(sg)(hσ) **end**
139 **pre** hσ_{init} = hσ $\wedge$ (hσ_{init},hσ_{end}) $\in$ **dom** sema

If a desired hub state change fails (i.e., does not meet the **pre**-condition, or for other reasons (e.g., failure of technology)), then we do not define the outcome of signaling.

139 signal_sequence($\langle\rangle$)(hσ) $\equiv$ hσ
139 signal_sequence($\langle$(hσ‘,ti)$\rangle\widehat{\ }$sg)(hσ) $\equiv$ **wait**(ti); signal_sequence(sg)(hσ‘)

We omit expression of a number of well-formedness conditions, e.g., that the *htriple* link identifiers are those of the corresponding mereology (*lis*), etcetera. The design of the semaphore, for a single hub or for a net of connected hubs has many similarities with the design of interlocking tables for railway tracks [98]. ■

Another example shows another aspect of support technology: Namely that the technology must guarantee certain of its own behaviours, so that software designed to interface with this technology, together with the technology, meets dependability requirements.

Example 7 Railway Optical Gates: *Train traffic (*itf:iTF*), intrinsically, is a total function over some time interval, from time (*t:T*) to continuously positioned (*p:P*) trains (*tn:TN*). Conventional optical gates sample, at regular intervals, the intrinsic train traffic. The result is a sampled traffic (*stf:sTF*). Hence the collection of all optical gates, for any given railway, is a partial function from intrinsic to sampled train traffics (*stf*). We need to express quality criteria that any optical gate technology should satisfy — relative to a necessary and sufficient description of a* closeness predicate*. The following axiom does that:*

- *For all intrinsic traffics, itf, and for all optical gate technologies, og, the following must hold: Let stf be the traffic sampled by the optical gates. For all time points, t, in the sampled traffic, those time points must also be in the intrinsic traffic, and, for all trains, tn, in the intrinsic traffic at that time, the train must be observed by the optical gates, and the actual position of the train and the sampled position must somehow be check-able to be close, or identical to one another.*

Since units change state with time, n:N, the railway net, needs to be part of any model of traffic.

type
  T, TN
  P = U*
  NetTraffic == net:N trf:(TN $\overrightarrow{m}$ P)
  iTF = T → NetTraffic
  sTF = T $\overrightarrow{m}$ NetTraffic
  oG = iTF $\tilde{\to}$ sTF
value
  close: NetTraffic × TN × NetTraffic $\tilde{\to}$ **Bool**
axiom
  ∀ itt:iTF, og:OG • **let** stt = og(itt) **in**
    ∀ t:T • t ∈ **dom** stt ⇒
      ∀ Tn:TN • tn ∈ **dom** trf(itt(t))
        ⇒ tn ∈ **dom** trf(stt(t)) ∧ close(itt(t),tn,stt(t)) **end**

Check-ability is an issue of testing the optical gates when delivered for conformance to the closeness predicate, i.e., to the axiom. ■

2.3.2 Requirements

Section 4.4 [Extension] of [9] illustrates a possible toll-gate, whose behaviour exemplifies a support technology. So do pumps of a pipe-line system such as illustrated in Examples 24, 29 and 42–44 in [1]. A pump of a pipe-line system gives rise to several forms of support technologies: from the Egyptian Shadoof [irrigation] pumps, and the Hellenic Archimedian screw pumps, via the 11th century Su Song pumps of China[7], and the hydraulic "technologies" of Moorish Spain[8] to the centrifugal and gear pumps of the early industrial age, etcetera, The techniques – to mention those that have influenced this author – of [50, 130, 131, 98] appears to apply well to the modeling of support technology requirements.

2.3.3 On Modeling Support Technologies

Support technologies in their relation to the domain in which they reside typically reflect real-time embeddedness. As such the techniques and languages for modeling support technologies resemble those for modeling event and process intensity, while temporal notions are brought into focus. Hence typical modeling notations include event-based languages (like Petri nets [74] or CSP) [23], respectively process-based specification languages (like MSCs, [75], LSCs [127], Statecharts [76], or CSP) [23], as well as temporal languages (like the Duration Calculus and [50] and Temporal Logic of Actions, TLA+) [49]).

2.4 Rules & Regulations

- *By a* **domain rule** *we shall understand some text (in the domain) which prescribes how people or equipment are expected to behave when dispatching their duties, respectively when performing their functions* ■
- *By a* **domain regulation** *we shall understand some text (in the domain) which prescribes what remedial actions are to be taken when it is decided that a rule has not been followed according to its intention* ■

The domain rules & regulations need or may not be explicitly present, i.e., written down. They may be part of the "folklore", i.e., tacitly assumed and understood.

[7] https://en.wikipedia.org/wiki/Su_Song
[8] http://www.islamicspain.tv/Arts-and-Science/The-Culture-of-Al-Andalus/Hydraulic-Technology.htm

2.4.1 Conceptual Analysis

Example 8 Trains at Stations:

- Rule: *In China the arrival and departure of trains at, respectively from, railway stations is subject to the following rule:*

  In any three-minute interval at most one train may either arrive to or depart from a railway station.
- Regulation: *If it is discovered that the above rule is not obeyed, then there is some regulation which prescribes administrative or legal management and/or staff action, as well as some correction to the railway traffic.*

■

Example 9 Trains Along Lines:

- Rule: *In many countries railway lines (between stations) are segmented into blocks or sectors. The purpose is to stipulate that if two or more trains are moving along the line, then:*

  There must be at least one free sector (i.e., without a train) between any two trains along a line.
- Regulation: *If it is discovered that the above rule is not obeyed, then there is some regulation which prescribes administrative or legal management and/or staff action, as well as some correction to the railway traffic.*

■

At a meta-level, i.e., explaining the general framework for describing the syntax and semantics of the human-oriented domain languages for expressing rules and regulations, we can say the following: There are, abstractly speaking, usually three kinds of languages involved wrt. (i.e., when expressing) rules and regulations (respectively when invoking actions that are subject to rules and regulations). Two languages, Rules and Reg, exist for describing rules, respectively regulations; and one, Stimulus, exists for describing the form of the [always current] domain action stimuli. A syntactic stimulus, sy_sti, denotes a function, se_sti:STI: $\Theta \rightarrow \Theta$, from any configuration to a next configuration, where configurations are those of the system being subjected to stimulations. A syntactic rule, sy_rul:Rule, stands for, i.e., has as its semantics, its meaning, rul:RUL, a predicate over current and next configurations, $(\Theta \times \Theta) \rightarrow$ **Bool**, where these next configurations have been brought about, i.e., caused, by the stimuli. These stimuli express: If the predicate holds then the stimulus will result in a valid next configuration.

type
  Stimulus, Rule, Θ
  STI = $\Theta \rightarrow \Theta$
  RUL = $(\Theta \times \Theta) \rightarrow$ **Bool**
value
  meaning: Stimulus $\rightarrow$ STI
  meaning: Rule $\rightarrow$ RUL
  valid: Stimulus $\times$ Rule $\rightarrow \Theta \rightarrow$ **Bool**
  valid(sy_sti,sy_rul)(θ) $\equiv$ meaning(sy_rul)(θ,(meaning(sy_sti))(θ))

A syntactic regulation, sy_reg:Reg (related to a specific rule), stands for, i.e., has as its semantics, its meaning, a semantic regulation, se_reg:REG, which is a pair. This pair consists of a predicate, pre_reg:Pre_REG, where Pre_REG = $(\Theta \times \Theta) \rightarrow$ **Bool**, and a domain configuration-changing function, act_reg:Act_REG, where Act_REG = $\Theta \rightarrow \Theta$, that is, both involving current and next domain configurations. The two kinds of functions express: If the predicate holds, then the action can be applied. The predicate is almost the inverse of the rules functions. The action function serves to undo the stimulus function.

type
  Reg


```
   Rul_and_Reg = Rule × Reg
   REG = Pre_REG × Act_REG
   Pre_REG = Θ × Θ → Bool
   Act_REG = Θ → Θ
value
   interpret: Reg → REG
```

The idea is now the following: Any action (i.e., event) of the system, i.e., the application of any stimulus, may be an action (i.e., event) in accordance with the rules, or it may not. Rules therefore express whether stimuli are valid or not in the current configuration. And regulations therefore express whether they should be applied, and, if so, with what effort. More specifically, there is usually, in any current system configuration, given a set of pairs of rules and regulations. Let (sy_rul,sy_reg) be any such pair. Let sy_sti be any possible stimulus. And let θ be the current configuration. Let the stimulus, sy_sti, applied in that configuration result in a next configuration, θ', where $\theta' = (\text{meaning}(\text{sy\_sti}))(\theta)$. Let θ' violate the rule, $\sim\text{valid}(\text{sy\_sti},\text{sy\_rul})(\theta)$, then if predicate part, pre_reg, of the meaning of the regulation, sy_reg, holds in that violating next configuration, $\text{pre\_reg}(\theta,(\text{meaning}(\text{sy\_sti}))(\theta))$, then the action part, act_reg, of the meaning of the regulation, sy_reg, must be applied, $\text{act\_reg}(\theta)$, to remedy the situation.

```
axiom
   ∀ (sy_rul,sy_reg):Rul_and_Reg •
      let se_rul = meaning(sy_rul),
          (pre_reg,act_reg) = meaning(sy_reg) in
      ∀ sy_sti:Stimulus, θ:Θ •
            ~valid(sy_sti,se_rul)(θ)
                ⇒ pre_reg(θ,(meaning(sy_sti))(θ))
                    ⇒ ∃ nθ:Θ • act_reg(θ)=nθ ∧ se_rul(θ,nθ)
      end
```

It may be that the regulation predicate fails to detect applicability of regulations actions. That is, the interpretation of a rule differs, in that respect, from the interpretation of a regulation. Such is life in the domain, i.e., in actual reality.

2.4.2 Requirements

Implementation of rules & regulations implies ***monitor***ing and partially ***control***ling the states symbolised by Θ in Sect. 2.4.1. Thus some ***partial implementation*** of Θ must be required; as must some monitoring of states $\theta{:}\Theta$ and implementation of the predicates *meaning*, *valid*, *interpret*, *pre_reg* and action(s) *act_reg*. The emerging requirements follow very much in the line of support technology requirements.

2.4.3 On Modeling Rules and Regulations

Usually rules (as well as regulations) are expressed in terms of domain entities, including those grouped into "the state", functions, events, and behaviours. Thus the full spectrum of model-ling techniques and notations may be needed. Since rules usually express properties one often uses some combination of axioms and wellformedness predicates. Properties sometimes include temporality and hence temporal notations (like Duration Calculus or Temporal Logic of Actions) are used. And since regulations usually express state (restoration) changes one often uses state changing notations (such as found in Allard [65], B or event-B [66], RSL [22], VDM-SL [67, 68, 69], and Z [70]). In some cases it may be relevant to model using some constraint satisfaction notation [132] or some Fuzzy Logic notations [133].

2.5 Scripts

- *By a* **domain script** *we shall understand the structured, almost, if not outright, formally expressed, wording of a procedure on how to proceed, one that has legally binding power, that is, which may be contested in a court of law* ■

2.5.1 Conceptual Analysis

Rules & regulations are usually expressed, even when informally so, as predicates. Scripts, in their procedural form, are like instructions, as for an algorithm.

Example 10 **A Casually Described Bank Script:** *Our formulation amounts to just a (casual) rough sketch. It is followed by a series of four large examples. Each of these elaborate on the theme of (bank) scripts. The problem area is that of how repayments of mortgage loans are to be calculated. At any one time a mortgage loan has a balance, a most recent previous date of repayment, an interest rate and a handling fee. When a repayment occurs, then the following calculations shall take place: (i) the interest on the balance of the loan since the most recent repayment, (ii) the handling fee, normally considered fixed, (iii) the effective repayment — being the difference between the repayment and the sum of the interest and the handling fee — and the new balance, being the difference between the old balance and the effective repayment. We assume repayments to occur from a designated account, say a demand/deposit account. We assume that bank to have designated fee and interest income accounts. (i) The interest is subtracted from the mortgage holder's demand/deposit account and added to the bank's interest (income) account. (ii) The handling fee is subtracted from the mortgage holder's demand/deposit account and added to the bank's fee (income) account. (iii) The effective repayment is subtracted from the mortgage holder's demand/deposit account and also from the mortgage balance. Finally, one must also describe deviations such as overdue repayments, too large, or too small repayments, and so on.* ■

Example 11 **A Formally Described Bank Script:** *First we must informally and formally define the bank state: There are clients (c:C), account numbers (a:A), mortgage numbers (m:M), account yields (ay:AY) and mortgage interest rates (mi:MI). The bank registers, by client, all accounts (ρ:A_Register) and all mortgages (μ:M_Register). To each account number there is a balance (α:Accounts). To each mortgage number there is a loan (ℓ:Loans). To each loan is attached the last date that interest was paid on the loan.*

```
value
   r, r′:Real axiom ...
type
   C, A, M, Date
   AY′ = Real, AY = {| ay:AY′ • 0<ay≤r |}
   MI′ = Real, MI = {| mi:MI′ • 0<mi≤r′ |}
   Bank′ = A_Register × Accounts × M_Register × Loans
   Bank = {| β:Bank′ • wf_Bank(β)|}
   A_Register = C →m A-set
   Accounts = A →m Balance
   M_Register = C →m M-set
   Loans = M →m (Loan × Date)
   Loan,Balance = P
   P = Nat
```

Then we must define well-formedness of the bank state:

```
value
   ay:AY, mi:MI
   wf_Bank: Bank → Bool
   wf_Bank(ρ,α,μ,ℓ) ≡ ∪ rng ρ = dom α ∧ ∪ rng μ = dom ℓ
axiom
   ay<mi [ ∧ ... ]
```

We — perhaps too rigidly — assume that mortgage interest rates are higher than demand/deposit account interest rates: ay<mi. Operations on banks are denoted by the commands of the bank script language. First the syntax:

```
type
   Cmd = OpA | CloA | Dep | Wdr | OpM | CloM | Pay
   OpA == mkOA(c:C)
   CloA == mkCA(c:C,a:A)
   Dep == mkD(c:C,a:A,p:P)
   Wdr == mkW(c:C,a:A,p:P)
   OpM == mkOM(c:C,p:P)
   Pay == mkPM(c:C,a:A,m:M,p:P,d:Date)
   CloM == mkCM(c:C,m:M,p:P)
   Reply = A | M | P | OkNok
   OkNok == ok | notok
value
   period: Date × Date → Days [for calculating interest]
   before: Date × Date → Bool [first date is earlier than last date]
```

And then the semantics:

```
   int_Cmd(mkPM(c,a,m,p,d))(ρ,α,μ,ℓ) ≡
      let (b,d′) = ℓ(m) in
      if α(a)≥p
         then
            let i = interest(mi,b,period(d,d′)),
                ℓ′ = ℓ † [m↦ℓ(m)−(p−i)]
                α′ = α † [a↦α(a)−p,a_i↦α(a_i)+i] in
            ((ρ,α′,μ,ℓ′),ok) end
         else
            ((ρ,α′,μ,ℓ),nok)
      end end
      pre c ∈ dom μ ∧ a ∈ dom α ∧ m ∈ μ(c)
      post before(d,d′)

      interest: MI × Loan × Days → P
```

■

The idea about scripts is that they can somehow be objectively enforced: that they can be precisely understood and consistently carried out by all stakeholders, eventually leading to computerisation. But they are, at all times, part of the domain.

2.5.2 Requirements

Script requirements call for the possibly interactive computerisation of algorithms, that is, for rather classical computing problems. But sometimes these scripts can be expressed, computably, in the form of programs in a domain specific language. As an example we refer to [134]. [134] illustrates how the design of pension and life insurance products, and their administration, reserve calculations, and audit, can be based on a common formal notation. The notation is human-readable and machine-processable,and specialised to the actuarial domain, achieving great expressive power combined with ease of use and safety. More specifically (a) product definitions based on standard actuarial models, including arbitrary continuous-time Markov and semi-Markov models, with cyclic transitions permitted; (b) calculation descriptions for reserves and other quantities of interest, based on differential equations; and (c) administration rules.

2.5.3 On Modeling Scripts

Scripts (as are licenses) are like programs (respectively like prescriptions program executions). Hence the full variety of techniques and notations for modeling programming (or specification) languages apply

[135, 136, 137, 138, 139, 140]. [141, Chaps. 6–9] cover pragmatics, semantics and syntax techniques for defining functional, imperative and concurrent programming languages.

2.6 License Languages

License: a right or permission
granted in accordance with law
by a competent authority
to engage in some business or occupation,
to do some act, or to engage in some transaction
which but for such license would be unlawful ■

Merriam Webster Online [83]

2.6.1 Conceptual Analysis

The Settings

A special form of scripts are increasingly appearing in some domains, notably the domain of electronic, or digital media. Here ***license***s express that a ***licensor***, o, ***permit***s a ***licensee***, u, to *render* (i.e., play) works of proprietary nature CD ROM-like music, DVD-like movies, etc. while obligating the licensee to pay the licensor on behalf of the owners of these, usually artistic works. Classical digital rights license languages, [142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160] applied to the electronic "downloading", payment and rendering (playing) of artistic works (for example music, literature readings and movies). In this chapter we generalise such applications languages and we extend the concept of licensing to also cover work authorisation (work commitment and promises) in health care, public government and schedule transport. The digital works for these new application domains are patient medical records, public government documents and bus/train/aircraft transport contracts. Digital rights licensing for artistic works seeks to safeguard against piracy and to ensure proper payments for the rights to render these works. Health care and public government license languages seek to ensure transparent and professional (accurate and timely) health care, respectively 'good governance'. Transport contract languages seeks to ensure timely and reliable transport services by an evolving set of transport companies. Proper mathematical definition of licensing languages seeks to ensure smooth and correct computerised management of licenses and contracts.

On Licenses

The concepts of licenses and licensing express relations between (i) *actors* (licensors (the authority) and licensees), (ii) *entities* (artistic works, hospital patients, public administration, citizen documents) and bus transport contracts and (iii) *functions* (on entities), and as performed by actors. By issuing a license to a licensee, a licensor wishes to express and enforce certain permissions and obligations: which functions on which entities the licensee is allowed (is licensed, is permitted) to perform. In this chapter we shall consider four kinds of entities: (i) digital recordings of artistic and intellectual nature: music, movies, readings ("audio books"), and the like, (ii) patients in a hospital as represented also by their patient medical records, (iii) documents related to public government, and (iv) transport vehicles, time tables and transport nets (of a buses, trains and aircraft).

Permissions and Obligations

The *permissions* and *obligations* issues are, (1) for the owner (agent) of some intellectual property to be paid (an *obligation*) by users when they perform *permitted* operations (rendering, copying, editing, sub-licensing) on their works; (2) for the patient to be professionally treated — by medical staff who are basically *obliged* to try to cure the patient; (3) for public administrators and citizens to enjoy good governance: transparency in law making (national parliaments and local prefectures and city councils), in law enforcement (i.e., the daily administration of laws), and law interpretation (the judiciary) — by agents who are basically *obliged* to produce certain documents while being *permitted* to consult (i.e., read, perhaps copy) other documents; and (4) for bus passengers to enjoy reliable bus schedules — offered by bus transport companies on contract to, say public transport authorities and on sub-contract to other such bus transport companies where these transport companies are *obliged* to honour a contracted schedule.

2.6.2 The Pragmatics

*By **pragmatics** we understand the study and practice of the factors that govern our choice of language in social interaction and the effects of our choice on others.*

In this section we shall rough-sketch-describe pragmatic aspects of the four domains of (1) production, distribution and consumption of artistic works, (2) the hospitalisation of patient, i.e., hospital health care, (3) the handling of law-based document in public government and (4) the operational management of schedule transport vehicles. The emphasis is on the pragmatics of the terms, i.e., the language used in these four domains.

Digital Media

Example 12 Digital Media: *The intrinsic entities of the performing arts are the artistic works: drama or opera performances, music performances, readings of poems, short stories, novels, or jokes, movies, documentaries, newsreels, etc. We shall limit our span to the scope of electronic renditions of these artistic works: videos, CDs or other. In this paper we shall not touch upon the technical issues of "downloading"(whether "streaming" or copying, or other). That and other issues should be analysed in [161].*

Operations on Digital Works:

For a consumer to be able to enjoy these works that consumer must (normally first) usually "buy a ticket" to their performances. The consumer, i.e., the theatre, opera, concert, etc., "goer" (usually) cannot copy the performance (e.g., "tape it"), let alone edit such copies of performances. In the context of electronic, i.e., digital renditions of these performances the above "cannots" take on a new meaning. The consumer may copy digital recordings, may edit these, and may further pass on such copies or editions to others. To do so, while protecting the rights of the producers (owners, performers), the consumer requests permission to have the digital works transferred ("downloaded") from the owner/producer to the consumer, so that the consumer can render ("play") these works on own rendering devices (CD, DVD, etc., players), possibly can copy all or parts of them, then possibly can edit all or parts of the copies, and, finally, possibly can further license these "edited" versions to other consumers subject to payments to "original" licensor.

License Agreement and Obligation:

To be able to obtain these permissions the user agrees with the wording of some license and pays for the rights to operate on the digital works.

Two Assumptions:

Two, related assumptions underlie the pragmatics of the electronics of the artistic works. The first assumption is that the format, the electronic representation of the artistic works is proprietary, that is, that the producer still owns that format. Either the format is publicly known or it is not, that is, it is somehow "secret". In either case we "derive" the second assumption (from the fulfillment of the first). The second assumption is that the consumer is not allowed to, or cannot operate[9] on the works by own means (software, machines). The second assumption implies that acceptance of a license results in the consumer receiving software that supports the consumer in performing all operations on licensed works, their copies and edited versions: rendering, copying, editing and sub-licensing.

[9] render, copy and edit

Protection of the Artistic Electronic Works:

The issue now is: how to protect the intellectual property (i.e., artistic) and financial (exploitation) rights of the owners of the possibly rendered, copied and edited works, both when, and when not further distributed.
■

Health-care

Example 13 **Health-care:** *Citizens go to hospitals in order to be treated for some calamity (disease or other), and by doing so these citizens become patients. At hospitals patients, in a sense, issue a request to be treated with the aim of full or partial restitution. This request is directed at medical staff, that is, the patient authorises medical staff to perform a set of actions upon the patient. One could claim, as we shall, that the patient issues a license.*

Patients and Patient Medical Records:

So patients and their attendant patient medical records (PMRs) are the main entities, the "works" of this domain. We shall treat them synonymously: PMRs as surrogates for patients. Typical actions on patients — and hence on PMRs — involve admitting patients, interviewing patients, analysing patients, diagnosing patients, planning treatment for patients, actually treating patients, and, under normal circumstance, to finally release patients.

Medical Staff:

Medical staff may request ('refer' to) other medical staff to perform some of these actions. One can conceive of describing action sequences (and 'referrals') in the form of hospitalisation (not treatment) plans. We shall call such scripts for licenses.

Professional Health Care:

The issue is now, given that we record these licenses, their being issued and being honoured, whether the handling of patients at hospitals follow, or does not follow properly issued licenses. ■

Government Documents

Example 14 **Documents:** *By public government we shall, following Charles de Secondat, baron de Montesquieu (1689–1755)*[10]*, understand a composition of three powers: the law-making (legislative), the law-enforcing and the law-interpreting parts of public government. Typically national parliament and local (province and city) councils are part of law-making government. Law-enforcing government is called the executive (the administration). And law-interpreting government is called the judiciary [system] (including lawyers etc.).*

Documents:

A crucial means of expressing public administration is through documents.[11] *We shall therefore provide a brief domain analysis of a concept of documents. (This document domain description also applies to patient medical records and, by some "light" interpretation, also to artistic works — insofar as they also are documents.) Documents are created, edited and read; and documents can be copied, distributed, the subject of calculations (interpretations) and be shared and shredded.*

[10] *De l'esprit des lois* (*The Spirit of the Laws*), published 1748

[11] Documents are, for the case of public government to be the "equivalent" of artistic works.

Document Attributes:

With documents one can associate, as attributes of documents, the actors who created, edited, read, copied, distributed (and to whom distributed), shared, performed calculations and shredded documents. With these operations on documents, and hence as attributes of documents one can, again conceptually, associate the location and time of these operations.

Actor Attributes and Licenses:

With actors (whether agents of public government or citizens) one can associate the authority (i.e., the rights) these actors have with respect to performing actions on documents. We now intend to express these authorisations as licenses.

Document Tracing:

An issue of public government is whether citizens and agents of public government act in accordance with the laws — with actions and laws reflected in documents such that the action documents enables a trace from the actions to the laws "governing" these actions. We shall therefore assume that every document can be traced back to its law-origin as well as to all the documents any one document-creation or -editing was based on. ■

Transportation

Example 15 Passenger and Goods Transport:

A Synopsis:

Contracts obligate transport companies to deliver bus traffic according to a timetable. The timetable is part of the contract. A contractor may sub-contract (other) transport companies to deliver bus traffic according to timetables that are sub-parts of their own timetable. Contractors are either public transport authorities or contracted transport companies. Contracted transport companies may cancel a subset of bus rides provided the total amount of cancellations per 24 hours for each bus line does not exceed a contracted upper limit The cancellation rights are spelled out in the contract. A sub-contractor cannot increase a contracted upper limit for cancellations above what the sub-contractor was told (in its contract) by its contractor. Etcetera.

A Pragmatics and Semantics Analysis:

The "works" of the bus transport contracts are two: the timetables and, implicitly, the designated (and obligated) bus traffic. A bus timetable appears to define one or more bus lines, with each bus line giving rise to one or more bus rides. Nothing is (otherwise) said about regularity of bus rides. It appears that bus ride cancellations must be reported back to the contractor. And we assume that cancellations by a sub-contractor is further reported back also to the sub-contractor's contractor. Hence eventually that the public transport authority is notified. Nothing is said, in the contracts, such as we shall model them, about passenger fees for bus rides nor of percentages of profits (i.e., royalties) to be paid back from a sub-contractor to the contractor. So we shall not bother, in this example, about transport costs nor transport subsidies. But will leave that necessary aspect as an exercise. The opposite of cancellations appears to be 'insertion' of extra bus rides, that is, bus rides not listed in the time table, but, perhaps, mandated by special events[12] *We assume that such insertions must also be reported back to the contractor. We assume concepts of acceptable and unacceptable bus ride delays. Details of delay acceptability may be given in contracts, but we ignore further descriptions of delay acceptability. but assume that unacceptable bus ride delays are also to be (iteratively) reported back to contractors. We finally assume that sub-contractors cannot (otherwise) change timetables. (A timetable change can only occur after, or at, the expiration of a license.) Thus we find that contracts have definite period of validity. (Expired contracts may be replaced by new contracts, possibly with new timetables.)*

[12] Special events: breakdown (that is, cancellations) of other bus rides, sports event (soccer matches), etc.

Contracted Operations, An Overview:

The actions that may be granted by a contractor according to a contract are: (i) start: to commence, i.e., to start, a bus ride (obligated); (ii) end: to conclude a bus ride (obligated); (iii) cancel: to cancel a bus ride (allowed, with restrictions); (iv) insert: to insert a bus ride; and (v) subcontract: to sub-contract part or all of a contract. ■

2.6.3 Schematic Rendition of License Language Constructs

There are basically two aspects to licensing languages: (i) the [actual] ***licensing*** [and sub-licensing], in the form of ***license***s, ℓ, by ***licensor***s, o, of ***permission***s and thereby implied ***obligation***s, and (ii) the carrying-out of these obligations in the form of ***licensee***, u, ***action***s. We shall in this chapter treat licensors and licensees on par, that is, some os are also us and vice versa. And we shall think of licenses as not necessarily material entities (e.g., paper documents), but allow licenses to be tacitly established (understood).

Licensing

The granting of a license ℓ by a licensor o, to a set of licensees $u_{u_1}, u_{u_2}, ..., u_{u_u}$ in which ℓ expresses that these may perform actions $a_{a_1}, a_{a_2}, ..., a_{a_a}$ on work items $e_{e_1}, e_{e_2}, ..., e_{e_e}$ can be schematised:

ℓ : **licensor** o **contracts licensees** $\{\mathsf{u}_{u_1},\mathsf{u}_{u_2},...,\mathsf{u}_{u_u}\}$
  to perform actions $\{\mathsf{a}_{a_1},\mathsf{a}_{a_2},...,\mathsf{a}_{a_a}\}$ **on work items** $\{\mathsf{e}_{e_1},\mathsf{e}_{e_2},...,\mathsf{e}_{e_e}\}$
  allowing sub-licensing of actions $\{\mathsf{a}_{a_i},\mathsf{a}_{a_j},...,\mathsf{a}_{a_k}\}$ **to** $\{\mathsf{u}_{u_x},\mathsf{u}_{u_y},...,\mathsf{u}_{u_z}\}$

The two sets of action designators, das :$\{a_{a_1}, a_{a_2}, ..., a_{a_a}\}$ and sas :$\{a_{a_x}, a_{a_y}, ..., a_{a_z}\}$ need not relate. **Sub-licensing:** Line 3 of the above schema, ℓ, expresses that licensees $u_{u_1}, u_{u_2}, ..., u_{u_u}$, may act as licensors and (thereby sub-)license ℓ to licensees $us : \{u_{u_x}, u_{u_y}, ..., u_{u_z}\}$, distinct from $sus : \{u_{u_1}, u_{u_2}, ..., u_{u_u}\}$, that is, $us \cap sus = \{\}$. **Variants:** One can easily "cook up" any number of variations of the above license schema. **Revoke Licenses:** We do not show expressions for revoking part or all of a previously granted license.

Licensors and Licensees

Example 16 **Licensors and Licensees:**

Digital Media:

For digital media the original licensors are the original producers of music, film, etc. The "original" licensees are you and me ! Thereafter some of us may become licensors, etc.

Heath-care:

For health-care the original licensors are, say in Denmark, the Danish governments' National Board of Health[13]*; and the "original" licensees are the national hospitals. These then sub-license their medical clinics (rheumatology, cancer, urology, gynecology, orthopedics, neurology, etc.) which again sub-licenses their medical staff (doctors, nurses, etc.). A medical doctor may, as is the case in Denmark for certain actions, not [necessarily] perform these but may sub-license their execution to nurses, etc.*

Documents:

For government documents the original licensor are the (i) heads of parliament, regional and local governments, (ii) government (prime minister) and the heads of respective ministries, respectively the regional and local agencies and administrations. The "original" licensees are (i′) the members of parliament, regional and local councils charged with drafting laws, rules and regulations, (ii′) the ministry, respectively the regional and local agency department heads. These (the ′s) then become licensors when licensing their staff to handle specific documents.

[13] In the UK: the NHS, etc.

Transport:

For scheduled passenger (etc.) transportation the original licensors are the state, regional and/or local transport authorities. The "original" licensees are the public and private transport firms. These latter then become licensors licensors licensing drivers to handle specific transport lines and/or vehicles. ■

Actors and Actions

Example 17 **Actors and Actions:** *For each of the exemplified domains (**digital media, health care, government documents** and **transport**) we illustrate standard **license schemas:***

Digital Media:

w refers to a digital "work" with w' designating a newly created one; s_i refers to a sector of some work.

- **render** $w(s_i,s_j,...,s_k)$: *sectors $s_i,s_j,...,s_k$ of work w are rendered (played, visualised) in that order.*
- w' **:= copy** $w(s_i,s_j,...,s_k)$: *sectors $s_i,s_j,...,s_k$ of work w are copied and becomes work w'.*
- w' **:= edit** w **with** $\mathscr{E}(w_\alpha(s_a,s_b,...,s_c),...,w_\gamma(s_p,s_q,...,s_r))$: *work w is edited while [also] incorporating references to or excerpts from [other] works $w_\alpha(s_a,s_b,...,s_c),...,w_\gamma(s_p,s_q,...,s_r)$.*
- **read** w: *work w is read, i.e., information about work w is somehow displayed.*
- ℓ **: licensor m**
  - **contracts licensees** $\{\mathbf{u}_{u_1},\mathbf{u}_{u_2},...,\mathbf{u}_{u_u}\}$
  - **to perform actions** {RENDER, COPY, EDIT, READ}
  - **on work items** $\{w_{i_1},w_{i_2},...,w_{i_w}\}$**.**

Etcetera: other forms of actions can be thought of.

Heath-care:

Actors are here limited to the patients and the medical staff. We refer to Fig. 2.3 on the next page. It shows an archetypal hospitalisation plan and identifies a number of actions; π designates patients, t designates treatment (medication, surgery, ...). Actions are performed by medical staff, say h, with h being an implicit argument of the actions.

- **interview** π: *a PMR with name, age, family relations, addresses, etc., is established for patient π.*
- **admit** π: *the PMR records the anamnese (medical history) for patient π.*
- **establish analysis plan** π: *the PMR records which analyses (blood tests, ECG, blood pressure, etc.) are to be carried out.*
- **analyse** π: *the PMR records the results of the analyses referred to previously.*
- **diagnose** π: *medical staff h diagnoses, based on the analyses most recently performed.*
- **plan treatment for** π: *medical staff h sets up a treatment plan for patient π based on the diagnosis most recently performed.*
- **treat** π **wrt.** t: *medical staff h performs treatment t on patient π, observes "reaction" and records this in the PMR.*

Predicate "actions":

- **more analysis** π **?**,
- **more treatment** π **?** *and*
- **more diagnosis** π **?**.
- **release** π: *either the patient dies or is declared ready to be sent 'home'.*
- ℓ **: licensor o**
  - **contracts medical staff** $\{m_{m_1},m_{m_2},...,m_{m_m}\}$
  - **to perform actions** {INTERVIEW, ADMIT, PLAN ANALYSIS, ANALYSE, DIAGNOSE, PLAN TREATMENT, TREAT, RELEASE}
  - **on patients** $\{\pi_{p_1},\pi_{p_2},...,\pi_{p_p}\}$

Etcetera: other forms of actions can be thought of.

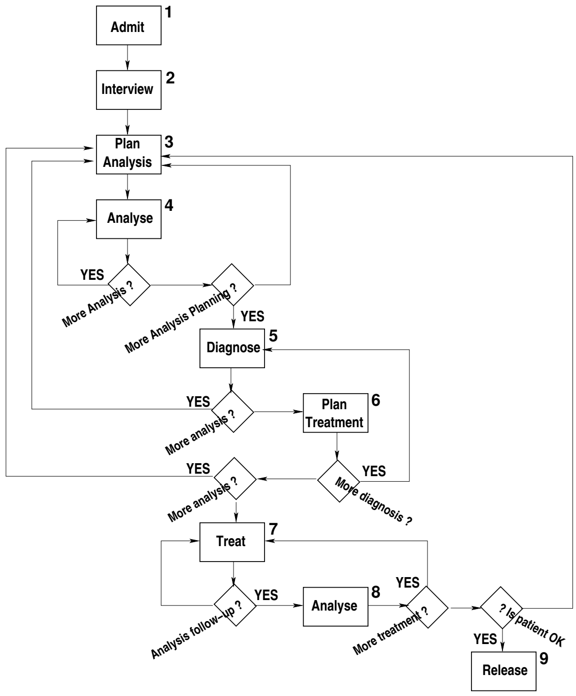


Fig. 2.3. An example single-illness non-fatal hospitalisation plan. States: $\{1,2,3,4,5,6,7,8,9\}$

Documents:

d refer to documents with d' designating new documents.

- **d' := create based on $d_x, d_y, ..., d_z$:**
  A new document, named d', is created, with no information "contents", but referring to existing documents $d_x, d_y, ..., d_z$.
- **edit d with $\mathscr{E}$ based on $d_{n_\alpha}, d_\beta, ..., d_\gamma$:**
  document d is edited with $\mathscr{E}$ being the editing function and $\mathscr{E}^{-1}$ being its "undo" inverse.
- **read d**: *document d is being read.*
- **d' := copy d**: *document d is copied into a new document named d'.*
- **freeze d**: *document d can, from now on, only be read.*
- **shred d**: *document d is shredded. That is, no more actions can be performed on d.*
- **ℓ : licensor o**
  - **contracts civil service staff** $\{c_{c_1}, c_{c_2}, ..., c_{c_c}\}$
  - **to perform actions** {CREATE, EDIT, READ, COPY, FREEZE, SHRED}
  - **on documents** $\{d_{d_1}, d_{d_2}, ..., d_{d_d}\}$

Etcetera: other forms of actions can be thought of.

Transport:

We restrict, without loss of generality, to bus transport. There is a timetable, tt. It records bus lines, l, and specific instances of bus rides, b.

- **start bus ride l, b at time t:**
  Bus line l is recorded in tt and its departure in tt is recorded as τ. Starting that bus ride at t means that the start is either on time, i.e., t=τ, or the start is delayed $\delta_d : \tau$-t or advanced $\delta_a : t$-τ where δ_d and δ_a are expected to be small intervals. All this is to be reported, in due time, to the contractor.

- **end bus ride l,b at time t:**
  Ending bus ride l,b at time t means that it is either ended on time, or earlier, or delayed. This is to be reported, in due time, to the contractor.
- **cancel bus ride l,b at time t:**
  t must be earlier than the scheduled departure of bus ride l,b.
- **insert an extra bus l,b' at time t:**
  t must be the same time as the scheduled departure of bus ride l,b with b' being a "marked" version of b.
- **ℓ : licensor o**
  - **contracts transport staff** $\{b_{b_1}, b_{b_2}, \ldots, b_{b_b}\}$
  - **to perform actions** {START, END, CANCEL, INSERT}
  - **on work items** $\{e_{e_1}, e_{e_2}, \ldots, e_{e_e}\}$

Etcetera: other forms of actions can be thought of. ■

2.6.4 Requirements

Requirements for license language implementation basically amounts to requirements for three aspects. (i) The design of the license language, its abstract and concrete syntax, its interpreter, and its interfaces to distributed licensor and licensee behaviours; (ii) the requirements for a distributed system of licensor and licensee behaviours; and (iii) the monitoring and partial control of the states of licensor and licensee behaviours. The structuring of these distributed licensor and licensee behaviours differ from slightly to somewhat, but not that significant in the four license languages examples. Basically the licensor and licensee behaviours form a set of behaviours. Basically everyone can communicate with everyone. For the case of digital media licensee behaviours communicate back to licensor behaviours whenever a properly licensed action is performed – resulting in the transfer of funds from licensees to licensors. For the case of health care some central authority is expected to validate the granting of licenses and appear to be bound by medical training. For the case of documents such checks appear to be bound by predetermined authorisation rules. For the case of transport one can perhaps speak of more rigid management & organisation dependencies as licenses are traditionally transferred between independent authorities and companies.

2.6.5 On Modeling License Languages

Licensors are expected to maintain a state which records all the licenses it has issued. Whenever at licensee "reports back" (the begin and/or the end) of the performance of a granted action, this is recorded in its state. Sometimes these granted actions are subject to fees. The licensor therefore calculates outstanding fees — etc. Licensees are expected to maintain a state which records all the licenses it has accepted. Whenever an action is to be performed the licensee records this and checks that it is permitted to perform this action. In many cases the licensee is expected to "report back", both the beginning and the end of performance of that action, to the licensor. A typical technique of modeling licensors, licensees and patients, i.e., their PMRs, is to model them as (never ending) processes, a la CSP [23]with input/output, ch ?/ch ! m, communications between licensors, licensees and PMRs. Their states are modeled as programmable attributes.

2.7 Management & Organisation

- *By* **domain management** *we shall understand such people (such decisions) (i) who (which) determine, formulate and thus set standards (cf. rules and regulations, Sect. 2.4) concerning strategic, tactical and operational decisions; (ii) who ensure that these decisions are passed on to (lower) levels of management and to floor staff; (iii) who make sure that such orders, as they were, are indeed carried out; (iv) who handle undesirable deviations in the carrying out of these orders cum decisions; and (v) who "backstops" complaints from lower management levels and from "floor" staff* ■

- *By* **domain organisation** *we shall understand (vi) the structuring of management and non-management staff "overseeable" into clusters with "tight" and "meaningful" relations; (vii) the allocation of strategic, tactical and operational concerns to within management and non-management staff clusters; and hence (viii) the "lines of command": who does what, and who reports to whom, administratively and functionally* ■

The '&' is justified from the interrelations of items *(i–viii)*.

2.7.1 Conceptual Analysis

We first bring some examples.

Example 18 **Train Monitoring, I:** *In China, as an example, till the early 1990s, rescheduling of trains occurs at stations and involves telephone negotiations with neighbouring stations ("up and down the lines"). Such rescheduling negotiations, by phone, imply reasonably strict management and organisation (M&O). This kind of M&O reflects the geographical layout of the rail net.* ■

Example 19 **Railway Management and Organisation: Train Monitoring, II:** *We single out a rather special case of railway management and organisation. Certain (lowest-level operational and station-located) supervisors are responsible for the day-to-day timely progress of trains within a station and along its incoming and outgoing lines, and according to given timetables. These supervisors and their immediate (middle-level) managers (see below for regional managers) set guidelines (for local station and incoming and outgoing lines) for the monitoring of train traffic, and for controlling trains that are either ahead of or behind their schedules. By an incoming and an outgoing line we mean part of a line between two stations, the remaining part being handled by neighbouring station management. Once it has been decided, by such a manager, that a train is not following its schedule, based on information monitored by non-management staff, then that manager directs that staff: (i) to suggest a new schedule for the train in question, as well as for possibly affected other trains, (ii) to negotiate the new schedule with appropriate neighbouring stations, until a proper reschedule can be decided upon, by the managers at respective stations, (iii) and to enact that new schedule.*[14] *A (middle-level operations) manager for regional traffic, i.e., train traffic involving several stations and lines, resolves possible disputes and conflicts.* ■

The above, albeit rough-sketch description, illustrated the following management and organisation issues: (i) There is a set of lowest-level (as here: train traffic scheduling and rescheduling) supervisors and their staff; (ii) they are organised into one such group (as here: per station); (iii) there is a middle-level (as here: regional train traffic scheduling and rescheduling) manager (possibly with some small staff), organised with one such per suitable (as here: railway) region; and (iv) the guidelines issued jointly by local and regional (...) supervisors and managers imply an organisational structuring of lines of information provision and command.

People staff enterprises, the components of infrastructures with which we are concerned, i.e., for which we develop software. The larger these enterprises — these infrastructure components — the more need there is for management and organisation. The role of management is roughly, for our purposes, twofold: first, to perform strategic, tactical and operational work, to set strategic, tactical and operational policies — and to see to it that they are followed. The role of management is, second, to react to adverse conditions, that is, to unforeseen situations, and to decide how they should be handled, i.e., conflict resolution. Policy setting should help non-management staff operate normal situations — those for which no management interference is thus needed. And management "backstops" problems: management takes these problems off the shoulders of non-management staff. To help management and staff know who's in charge wrt. policy setting and problem handling, a clear conception of the overall organisation is needed. Organisation defines lines of communication within management and staff, and between these. Whenever management and staff has to turn to others for assistance they usually, in a reasonably well-functioning enterprise, follow the command line: the paths of organigrams — the usually hierarchical box and arrow/line diagrams.

[14] That enactment may possibly imply the movement of several trains incident upon several stations: the one at which the manager is located, as well as possibly at neighbouring stations.

The *management and organisation* model of a domain is a partial specification; hence all the usual abstraction and modeling principles, techniques and tools apply. More specifically, management is a set of predicate functions, or of observer and generator functions These either parametrise other, the operations functions, that is, determine their behaviour, or yield results that become arguments to these other functions. Organisation is thus a set of constraints on communication behaviours. Hierarchical, rather than linear, and matrix structured organisations can also be modeled as sets (of recursively invoked sets) of equations.

To relate classical organigrams to formal descriptions we first show such an organigram (Fig. 2.4), and then we show schematic processes which — for a rather simple scenario — model managers and the managed! Based on such a diagram, and modeling only one neighbouring group of a manager and the staff

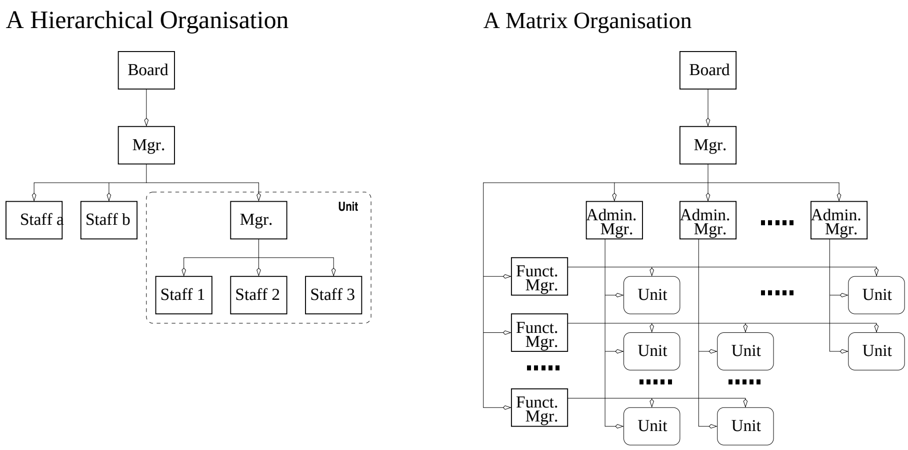


Fig. 2.4. Organisational structures

working for that manager we get a system in which one manager, mgr, and many staff, stf, coexist or work concurrently, i.e., in parallel. The mgr operates in a context and a state modeled by ψ. Each staff, stf(i) operates in a context and a state modeled by sσ(i).

type
  Msg, Ψ, Σ, Sx
  SΣ = Sx $\overrightarrow{m}$ Σ
channel
  { ms[i]:Msg | i:Sx }
value
  sσ:SΣ, ψ:Ψ

  sys: **Unit** $\rightarrow$ **Unit**
  sys() $\equiv$ $\|$ { stf(i)(sσ(i)) | i:Sx } $\|$ mgr(ψ)

In this system the manager, mgr, (1) either broadcasts messages, m, to all staff via message channel ms[i]. The manager's concoction, m_out(ψ), of the message, msg, has changed the manager state. Or (2) is willing to receive messages, msg, from whichever staff i the manager sends a message. Receipt of the message changes, m_in(i,m)(ψ), the manager state. In both cases the manager resumes work as from the new state. The manager chooses — in this model — which of thetwo things (1 or 2) to do by a so-called non-deterministic internal choice ($\sqcap$).

  mg: Ψ $\rightarrow$ **in**,**out** {ms[i]|i:Sx} **Unit**
  mgr(ψ) $\equiv$
(1) **let** (ψ',m)=m_out(ψ) **in** $\|$ {ms[i]!m|i:Sx};mgr(ψ') **end**
  $\sqcap$

(2) **let** ψ' = $\lceil\rceil$ {**let** m=ms[i]? **in** m_in(i,m)(ψ) **end**|i:Sx} **in** mgr(ψ') **end**

m_out: $\Psi \rightarrow \Psi \times$ MSG,
m_in: Sx $\times$ MSG $\rightarrow \Psi \rightarrow \Psi$

And in this system, staff i, stf(i), (1) either is willing to receive a message, msg, from the manager, and then to change, st_in(msg)(σ), state accordingly, or (2) to concoct, st_out(σ), a message, msg (thus changing state) for the manager, and send it ms[i]!msg. In both cases the staff resumes work as from the new state. The staff member chooses — in this model — which of thetwo "things" (1 or 2) to do by a non-deterministic internal choice ($\lceil\rceil$).

stf: i:Sx $\rightarrow \Sigma \rightarrow$ **in**,**out** ms[i] **Unit**
stf(i)(σ) $\equiv$
(1) **let** m = ms[i]? **in** stf(i)(stf_in(m)(σ)) **end**
$\lceil\rceil$
(2) **let** (σ',m) = st_out(σ) **in** ms[i]!m; stf(i)(σ') **end**

st_in: MSG $\rightarrow \Sigma \rightarrow \Sigma$,
st_out: $\Sigma \rightarrow \Sigma \times$ MSG

Both manager and staff processes recurse (i.e., iterate) over possibly changing states. The management process non-deterministically, internal choice, "alternates" between "broadcast"-issuing orders to staff and receiving individual messages from staff. Staff processes likewise non-deterministically, internal choice, alternate between receiving orders from management and issuing individual messages to management. The conceptual example also illustrates modeling stakeholder behaviours as interacting (here CSP-like) processes.

Example 20 Strategic, Tactical and Operations Management: *We think of (i) strategic, (ii) tactic, and (iii) operational managers as well as (iv) supervisors, (v) team leaders and the rest of the (vi) staff (i.e., workers) of a domain enterprise as functions. Each category of staff, i.e., each function, works in state and updates that state according to schedules and resource allocations — which are considered part of the state. To make the description simple we do not detail the state other than saying that each category works on an "instantaneous copy" of "the" state. Now think of six staff category activities, strategic managers, tactical managers, operational managers, supervisors, team leaders and workers as six simultaneous sets of actions. Each function defines a step of collective (i.e., group) (strategic, tactical, operational) management, supervisor, team leader and worker work. Each step is considered "atomic". Now think of an enterprise as the "repeated" step-wise simultaneous performance of these category activities. Six "next" states arise. These are, in the reality of the domain, ameliorated, that is reconciled into one state. however with the next iteration, i.e., step, of work having each category apply its work to a reconciled version of the state resulting from that category's previously yielded state and the mediated "global" state. Caveat: The below is not a mathematically proper definition. It suggests one !*

type
0. Σ, $\Sigma_s, \Sigma_t, \Sigma_o, \Sigma_u, \Sigma_e, \Sigma_w$
value
1. str, tac, opr, sup, tea, wrk: $\Sigma_i \rightarrow \Sigma_i$
2. stra, tact, oper, supr, team, work: $\Sigma \rightarrow (\Sigma_{x_1} \times \Sigma_{x_2} \times \Sigma_{x_3} \times \Sigma_{x_4} \times \Sigma_{x_5}) \rightarrow \Sigma$
3. objective: $(\Sigma_s \times \Sigma_t \times \Sigma_o \times \Sigma_u \times \Sigma_e \times \Sigma_w) \rightarrow$ **Bool**
3. enterprise,ameliorate: $(\Sigma_s \times \Sigma_t \times \Sigma_o \times \Sigma_u \times \Sigma_e \times \Sigma_w) \rightarrow \Sigma$
4. enterprise: $(\sigma_s, \sigma_t, \sigma_u, \sigma_e, \sigma_w) \equiv$
6. **let** σ'_s = stra(str(σ_s))(σ'_t,σ'_o,σ'_u,σ'_e,σ'_w),
7. σ'_t = tact(tac(σ_t))(σ'_s,σ'_o,σ'_u,σ'_e,σ'_w),
8. σ'_o = oper(opr(σ_o))(σ'_s,σ'_t,σ'_u,σ'_e,σ'_w),
9. σ'_u = supr(sup(σ_u))(σ'_s,σ'_t,σ'_o,σ'_e,σ'_w),
10. σ'_e = team(tea(σ_e))(σ'_s,σ'_t,σ'_o,σ'_u,σ'_w),

11. $\sigma'_w = \mathsf{work}(\mathsf{wrk}(\sigma_w))(\sigma'_s,\sigma'_t,\sigma'_o,\sigma'_u,\sigma'_e)$ **in**
12. **if** objective($\sigma'_s,\sigma'_t,\sigma'_o,\sigma'_u,\sigma'_e,\sigma'_w$)
13. **then** ameliorate($\sigma'_s,\sigma'_t,\sigma'_o,\sigma'_u,\sigma'_e,\sigma'_w$)
14. **else** enterprise($\sigma'_s,\sigma'_t,\sigma'_o,\sigma'_u,\sigma'_e,\sigma'_w$)
15. **end end**

0. Σ is a further undefined and unexplained enterprise state space. The various enterprise players view this state in their own way.
1. Six staff group operations, str, tac, opr, sup, tea and wrk, each act in the enterprise state such as conceived by respective groups to effect a resulting enterprise state such as achieved by respective groups.
2. Six staff group state amelioration functions, ame_s,ame _t, ame_o, ame_u, ame_e and ame_w, each apply to the resulting enterprise states such as achieved by respective groups to yield a result state such as achieved by that group.
3. An overall objective function tests whether a state summary reflects that the objectives of the enterprise has been achieved or not.
4. The enterprise function applies to the tuple of six group-biased (i.e., ameliorated) states. Initially these may all be the same state. The result is an ameliorated state.
5. An iteration, that is, a step of enterprise activities, lines 5.–13. proceeds as follows:
6. strategic management operates
   - in its state space, $\sigma_s : \Sigma$;
   - effects a next (un-ameliorated strategic management) state σ'_s;
   - and ameliorates this latter state in the context of all the other player's ameliorated result states.

7.–11. The same actions take place, simultaneously for the other players: tac, opr, sup, tea and wrk.

12. A test, *has* objectives *been met*, is made on the six ameliorated states.
13. If test is successful, then the enterprise terminates in an ameliorated state.
14. Otherwise the enterprise recurses, that is, "repeats" itself in new states.

The above "function" definition is suggestive. It suggests that a solution to the fix-point 6-tuple of equations over "intermediate" states, σ'_x, where x is any of s,t,o,u,e,w, is achieveable by iteration over just these 6 equations. ■

2.7.2 Requirements

Top-level, including strategic management tends to not be amenable to "automation". Increasingly tactical management tends to "divide" time between "bush-fire, stop-gap" actions – hardly automatable and formulating, initiating and monitoring main operations. The initiation and monitoring of tactical actions appear amenable to partial automation. Operational management – with its reliance on rules & regulations, scripts and licenses – is where computer monitoring and partial control has reaped the richest harvests.

2.7.3 On Modeling Management and Organisation

Management and organisation basically spans entity, function, event and behaviour intensities and thus typically require the full spectrum of modeling techniques and notations — summarised in Sect. 2.2.3.

2.8 Human Behaviour

- *By* **domain human behaviour** *we shall understand any of a quality spectrum of carrying out assigned work: from (i) careful, diligent and accurate, via (ii) sloppy dispatch, and (iii) delinquent work, to (iv) outright criminal pursuit* ■

2.8.1 Conceptual Analysis

To model human behaviour "smacks" like modeling human actors, the psychology of humans, etc. ! We shall not attempt to model the psychological side of humans — for the simple reason that we neither know how to do that nor whether it can at all be done. Instead we shall be focusing on the effects on non-human manifest entities of human behaviour.

Example 21 Banking — or Programming — Staff Behaviour: *Let us assume a bank clerk, "in ye olde" days, when calculating, say mortgage repayments (cf. Example 10). We would characterise such a clerk as being diligent, etc., if that person carefully follows the mortgage calculation rules, and checks and double-checks that calculations "tally up", or lets others do so. We would characterise a clerk as being sloppy if that person occasionally forgets the checks alluded to above. We would characterise a clerk as being delinquent if that person systematically forgets these checks. And we would call such a person a criminal if that person intentionally miscalculates in such a way that the bank (and/or the mortgage client) is cheated out of funds which, instead, may be diverted to the cheater. Let us, instead of a bank clerk, assume a software programmer charged with implementing an automatic routine for effecting mortgage repayments (cf. Example 11). We would characterise the programmer as being diligent if that person carefully follows the mortgage calculation rules, and throughout the development verifies and tests that the calculations are correct with respect to the rules. We would characterise the programmer as being sloppy if that person forgets certain checks and tests when otherwise correcting the computing program under development. We would characterise the programmer as being delinquent if that person systematically forgets these checks and tests. And we would characterise the programmer as being a criminal if that person intentionally provides a program which miscalculates the mortgage interest, etc., in such a way that the bank (and/or the mortgage client) is cheated out of funds.* ■

Example 22 A Human Behaviour Mortgage Calculation: *Example 11 gave a semantics to the mortgage calculation request (i.e., command) as would a diligent bank clerk be expected to perform it. To express, that is, to model, how sloppy, delinquent, or outright criminal persons (staff?) could behave we must modify the int_Cmd(mkPM(c,a,m,p,d′))(ρ,α,μ,ℓ) definition.*

```
int_Cmd(mkPM(c,a,m,p,d))(ρ,α,μ,ℓ) ≡
   let (b,d′) = ℓ(m) in
   if q(α(a),p) [α(a)≤p∨α(a)=p∨α(a)≤p∨...]
      then
         let i = f1(interest(mi,b,period(d,d′))),
             ℓ′ = ℓ † [m↦f2(ℓ(m)−(p−i))],
             α′ = α † [a↦f3(α(a)−p),a_i↦f4(α(a_i)+i),a"staff"↦f"staff"(α(a"staff")+i)] in
         ((ρ,α′,μ,ℓ′),ok) end
      else
         ((ρ,α′,μ,ℓ),nok)
   end end
   pre c ∈ dom μ ∧ m ∈ μ(c)

q: P × P ~→ Bool
f1,f2,f3,f4,f"staff": P ~→ P  [typically: f"staff" = λp.p]
```

■

The predicate q and the functions f_1, f_2, f_3, f_4 and $f_{\text{"staff"}}$ of Example 22 are deliberately left undefined. They are being defined by the "staffer" when performing (incl., programming) the mortgage calculation routine. The point of Example 22 is that one must first define the mortgage calculation script precisely as one would like to see the diligent staff (programmer) to perform (incl., correctly program) it before one can "pinpoint" all the places where lack of diligence may "set in". The invocations of q, f_1, f_2, f_3, f_4 and $f_{\text{"staff"}}$ designate those places. The point of Example 22 is also that we must first domain-define, "to the best of our ability" all the places where human behaviour may play other than a desirable role. If we cannot, then we cannot claim that some requirements aim at countering undesirable human behaviour.

Commensurate with the above, humans interpret rules and regulations differently, and, for some humans, not always consistently — in the sense of repeatedly applying the same interpretations. Our final specification pattern is therefore:

type
  Action = $\Theta \stackrel{\sim}{\rightarrow} \Theta$-**infset**
value
  hum_int: Rule → Θ → RUL-**infset**
  action: Stimulus → Θ → Θ
  hum_beha: Stimulus × Rules → Action → $\Theta \stackrel{\sim}{\rightarrow} \Theta$-**infset**
  hum_beha(sy_sti,sy_rul)(α)(θ) **as** θset
    post
      θset = $\alpha(\theta)$ ∧ action(sy_sti)(θ) ∈ θset
      ∧ ∀ θ':Θ•θ' ∈ θset ⇒
        ∃ se_rul:RUL•se_rul ∈ hum_int(sy_rul)(θ)⇒se_rul(θ,θ')

The above is, necessarily, sketchy: There is a possibly infinite variety of ways of interpreting some rules. A human, in carrying out an action, interprets applicable rules and chooses one which that person believes suits some (professional, sloppy, delinquent or criminal) intent. "Suits" means that it satisfies the intent, i.e., yields **true** on the pre/post-configuration pair, when the action is performed — whether as intended by the ones who issued the rules and regulations or not. We do not cover the case of whether an appropriate regulation is applied or not. The above-stated axioms express how it is in the domain, not how we would like it to be. For that we have to establish requirements.

2.8.2 Requirements

Requirements in relation to the human behaviour facet is not requirements about software that "replaces" human behaviour. Such requirements were hinted at in Sects. 2.5.2–2.7.2. Human behaviour facet requirements are about software that checks human behaviour; that its remains diligent; that it does not transgress into sloppy, delinquent, let alone criminal behaviour. When transgressions are discovered, appropriate remedial actions may be prescribed.

2.8.3 On Modeling Human Behaviour

To model human behaviour is, "initially", much like modeling management and organisation. But only 'initially'. The most significant human behaviour modeling aspect is then that of modeling non-determinism and looseness, even ambiguity. So a specification language which allows specifying non-determinism and looseness (like `CafeOBJ` [126] and `RSL` [22]) is to be preferred. To prescribe requirements is to prescribe the monitoring of the human input at the computer interface.

2.9 Conclusion

We have introduced the scientific and engineering concept of domain theories and domain engineering; and we have brought but a mere sample of the principles, techniques and tools that can be used in creating domain descriptions.

2.9.1 Completion

Domain acquisition results in typically up to thousands of units of domain descriptions. Domain analysis subsequently also serves to classify which facet any one of these description units primarily characterises. But some such "compartmentalisations" may be difficult, and may be deferred till the step of "completion". It may then be, "at the end of the day", that is, after all of the above facets have been modeled that some description units are left as not having been described, not deliberately, but "circumstantially". It then behooves the domain engineer to fit these "dangling" description units into suitable parts of the domain description. This "slotting in" may be simple, and all is fine. Or it may be difficult. Such difficulty may be a sign that the chosen model, the chosen description, in its selection of entities, functions, events and behaviours to model — in choosing these over other possible selections of phenomena and concepts is not appropriate. Another attempt must be made. Another selection, another abstraction of entities, functions, etc., may need be chosen. Usually however, after having chosen the abstractions of the intrinsic phenomena and concepts, one can start checking whether "dangling" description units can be fitted in "with ease".

2.9.2 Integrating Formal Descriptions

We have seen that to model the full spectrum of domain facets one needs not one, but several specification languages. No single specification language suffices. It seems highly unlikely and it appears not to be desirable to obtain a single, "universal" specification language capable of "equally" elegantly, suitably abstractly modeling all aspects of a domain. Hence one must conclude that the full modeling of domains shall deploy several formal notations – including plain, good old mathematics in all its forms. The issues are then the following which combinations of notations to select, and how to make sure that the combined specification denotes something meaningful. The ongoing series of "Integrating Formal Methods" conferences [162] is a good source for techniques, compositions and meanings.

2.9.3 The Impossibility of Describing Any Domain Completely

Domain descriptions are, by necessity, abstractions. One can never hope for any notion of complete domain descriptions. The situation is no better for domains such as we define them than for physics. Physicists strive to understand the manifest world around us – the world that was there before humans started creating "their domains". The physicists describe the physical world "in bits and pieces" such that large collections of these pieces "fit together", that is, are based on some commonly accepted laws and in some commonly agreed mathematics. Similarly for such domains as will be the subject of domain science & engineering such as we cover that subject in [1, 9] and in the present and upcoming papers ([3, 163], i.e., this chapter and Chapter 7). Individual such domain descriptions will be emphasizing some clusters of facets, others will be emphasizing other aspects.

2.9.4 Rôles for Domain Descriptions

We can distinguish between a spectrum of rôles for domain descriptions. Some of the issues brought forward below may have been touched upon in [1, 9].

Alternative Domain Descriptions:

It may very well be meaningful to avail oneself of a variety of domain models (i.e., descriptions) for any one domain, that is, for what we may consider basically one and the same domain. In control theory (a science) and automation (an engineering) we develop specific descriptions, usually on the form of a set of differential equations, for any one control problem. The basis for the control problem is typically the science of mechanics. This science has many renditions (i.e., interpretations). For the control problem, say that of keeping a missile carried by a train wagon, erect during train movement and/or windy conditions, one may then develop a "self-contained" description of the problem based on some mechanics theory presentation. Similarly for domains. One may refer to an existing domain description. But one may re-develop a textually "smaller" domain description for any one given, i.e., specific problem.

Domain Science:

A domain description designates a domain theory. That is, a bundle of propositions, lemmas and theorems that are either rather explicit or can be proven from the description. So a domain description is the basis for a theory as well as for the discovery of domain laws, that is, for a domain science. We have sciences of physics (incl. chemistry), biology, etc. Perhaps it is about time to have proper sciences, to the extent one can have such sciences for human-made domains.

Business Process Re-engineering:

Some domains manifest serious amounts of human actions and interactions. These may be found to not be efficient to a degree that one might so desire. A given domain description may therefore be a basis for suggesting other *management & organisation* structures, and/or *rules & regulations* than present ones. Yes, even making explicit *scripts* or a *license language* which have hitherto been tacitly understood – without necessarily computerising any support for such a *script* or *license language*. The given and the resulting domain descriptions may then be the basis for ***operations research*** models that may show desired or acceptable efficiency improvements.

Software Development:

[9] shows one approach to requirements prescription. Domain analysis & description, i.e., domain engineering, is here seen as an initial phase, with requirements prescription engineering being a second phase, and software design being a third phase. We see domain engineering as indispensable, that is, an absolute must, for software development. [102, *Domains: Their Simulation, Monitoring and Control*] further illustrates how domain engineering is a base for the development of domain simulators, demos, monitors and controllers.

2.9.5 Grand Challenges of Informatics[16]

To establish a reasonably trustworthy and believable theory of a domain, say the transportation, or just the railway domain, may take years, possibly 10–15 ! Similarly for domains such as the financial service industry, the market (of consumers and producers, retailers, wholesaler, distribution cum supply chain), health care, and so forth. The current author urges younger scientists to get going! It is about time.

[16] In the early-to-mid 2000s there were a rush of research foundations and scientists enumerating *"Grand Challenges of Informatics"*

3

Formal Models of Processes and Prompts

Chapter 1 [1, 2] introduced a method for analysing and describing manifest domains. In this chapter we formalise the calculus of this method. The formalisation has two aspects: the formalisation of the process of sequencing the prompts of the calculus, and the formalisation of the individual prompts.

3.1 Introduction

The presentation of a calculus for analysing and describing manifest domains, introduced in Chapter 1 [1, 2] and summarised in Sect. 3.2, was and is necessarily informal. The human process of "extracting" a description of a domain, based on analysis, "wavers" between the domain, as it is revealed to our senses, and therefore necessarily informal, and its recorded description, which we present in two forms, an informal narrative and a formalisation. In the present chapter we shall provide a formal, operational semantics formalisation of the analysis and description calculus. There are two aspects to the semantics of the analysis and description calculus. There is the formal explanation of the process of applying the analysis and description prompts, in particular the practical meaning[1] of the results of applying the analysis prompts, and there is the formal explanation of the meaning of the results of applying the description prompts. The former (i.e., the practical meaning of the results of applying the analysis prompts) amounts to a model of the process whereby the domain analyser cum describer navigates "across" the domain, alternating between applying sequences of one or more analysis prompts and applying description prompts. The latter (formal explanation of the meaning of the results of applying the description prompts) amounts to a model of the domain (as it evolves in the mind of the analyser cum describer[2]), the meaning of the evolving description, and thereby the relation between the two.

3.1.1 Related Work

To this author's knowledge there are not many papers, other than the author's own, [1, 2, 3, 9, 11] and the present chapter, which proposes a calculus of analysis and description prompts for capturing a domain, let alone, as this chapter tries, to formalise aspects of this calculus.

There is, however a "school of software engineering", "anchored" in the 1987 publication: [164, Leon Osterweil]. As the title of that paper reveals: *"Software Processes Are Software Too"* the emphasis is on

[1] in contrast to a formal mathematical meaning

[2] By 'domain analyser cum describer' we mean a group of one or more professionals, well-educated and trained in the domain analysis & description techniques outlined in, for example, [2], and where these professionals work closely together. By 'working closely together' we mean that they, together, day-by-day work on each their sections of a common domain description document which they "buddy check", say every morning, then discuss, as a group, also every day, and then revise and further extend, likewise every day. By "buddy checking" we mean that group member $\mathscr{A}$ reviews group member $\mathscr{B}$'s most recent sections – and where this reviewing alternates regularly: $\mathscr{A}$ may first review $\mathscr{B}$'s work, then $\mathscr{C}$'s, etcetera.
We shall, occasionally refer to the 'domain analyser cum describer' as the 'domain engineer'.

considering the software development process as prescribable by a software program. That is not what we are aiming at. We are aiming at an abstract and formal description of a large class of domain analysis & description processes *in terms of possible development calculi*. And in such a way that one can reason about such processes. The Osterweil paper suggests that any particular software development can be described by a program, and, if we wish to reason about the software development process we must reason over that program, but there is no requirement that the "software process programs" be expressed in a language with a proof system.[3] In contrast we can reason over the properties of the development calculi as well as over the resulting description.

There is another "school of programming", one that more closely adheres to the use of a calculus [165, 166]. The calculus here is a set of refinement rules, a *Refinement Calculus*[4], that "drives" the developer from a specification to an executable program. Again, that is not what we are doing here. The proposed calculi of analysis and of description prompts [1, 2] "drives" the domain engineer in developing a domain description. That description may then be 'refined' using a refinement calculus.

3.1.2 Structure of Chapter

Section 3.2 provides a terse summary of the analysis & description of endurants. It is without examples. For such we refer to Chapter 1 [1, 2]. Section 3.3 is informal. It discusses issues of syntax and semantics. The reason we bring this short section is that the current chapter turns "things upside/down": from semantics we extract syntax ! From the real entities of actual domains we extract domain descriptions. Section 3.4 presents a pseudo-formal operational semantics explication of the process of proceeding through iterated sequences of analysis prompts to description prompts. The formal meaning of these prompts are given in Sect. 3.8. But first we must "prepare the ground": The meaning of the analysis and description prompts is given in terms of some formal "context" in which the domain engineer works. Section 3.5 discusses this notion of "image" — an informal aspect of the 'context'. It is a brief discussion. Section 3.6 presents the formal aspect of the 'context': perceived abstract syntaxes of the ontology of domain endurants and of endurant values. Section 3.7 Discusses, in a sense, the mental processes – *from syntax to semantics and back again !* – that the domain engineer appears to undergo while analysing (the semantic) domain entities and synthesizing (the syntactic) domain descriptions. Section 3.8 presents the analysis and description prompts meanings. It represents a high point of this chapter. It so-to-speak justifies the whole "exercise" ! Section 3.9 concludes the chapter. We summarize what we have "achieved". And we discuss whether this "achievement" is a valid one !

3.2 Domain Analysis and Description

In the rest of this chapter we shall consider entities in the context of their being manifest (i.e., spatio-temporal). The restrictions of what we cover with respect to Chapter 1 [1, 2] are: we do not cover ***perdurant***s, only ***endurant***s. These omissions do not affect the main aim of this paper, namely that of presenting a plausible example of how one might wish to operationally formalise the notions of the analysis & description process and of the analysis & description prompts. The presentation is very terse. We refer to Chapter 1 [1, 2] for details.

3.2.1 General

In Chapter 1 [1, 2] we developed an ontology for structuring and a prompt calculus analysing and describing domains. Figure 3.1 on the facing page captures the ontology structure. It is thus a slight simplification of the 'upper ontology' figure given in [2] in that it omits the ***component*** ontology. The rest of this section will summarise the calculus. We refer to [2] for examples.

---

[3] The **RAISE S**pecification **L**anguage [64] does have a proof system.

[4] Ralph–Johan Back appears to be the first to have proposed the idea of refinement calculi, cf. his 1978 PhD thesis *On the Correctness of Refinement Steps in Program Development*, http://users.abo.fi/backrj/index.php?page=-Refinement_calculusall.html&menu=3.

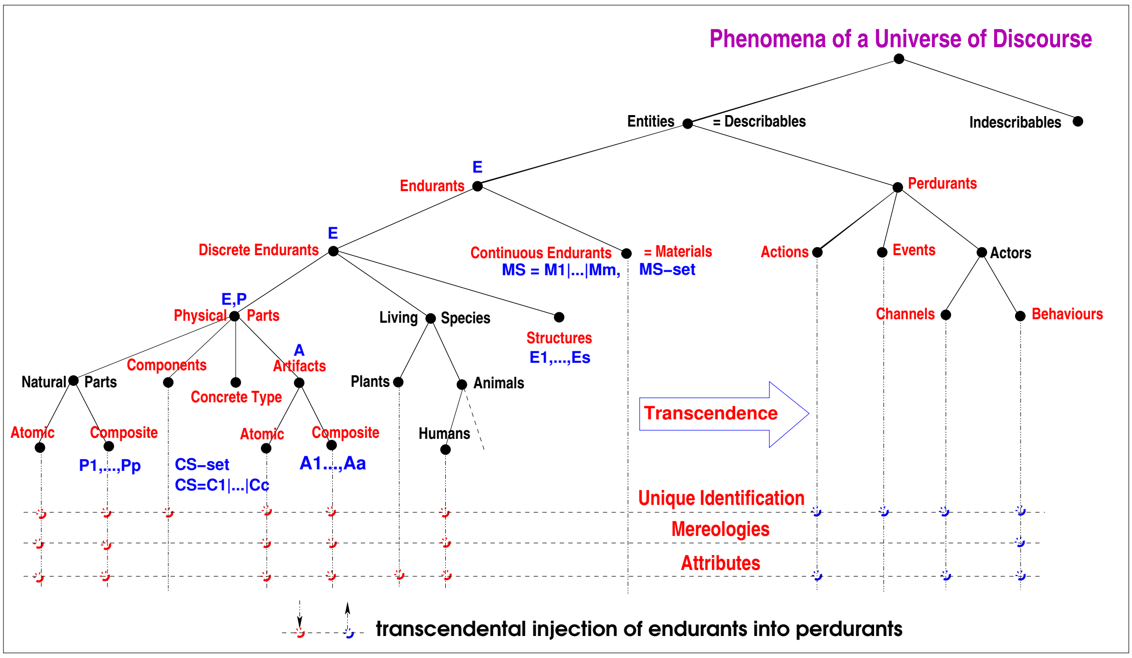


Fig. 3.1. An Upper Ontology for Domains

To the nodes of the upper ontology of Fig. 3.1 we have affixed some names. Names beginning with a capital stand for sub-ontologies. Names starting with a slanted *obs_* stand for description prompts. Other names (starting with an `is_` or a `has_`, or other) stand for analysis prompts.[5]

3.2.2 Entities

Definition 42 Domain Entity: By an **entity** we shall understand a **phenomenon**, i.e., something that can be ***observe***d, i.e., be seen or touched by humans, or that can be ***conceive***d as an ***abstraction*** of an entity. We further demand that an entity can be objectively described ■[6]

Analysis Prompt 24 *is_entity: The domain analyser analyses "things" (θ) into either entities or non-entities. The method can thus be said to provide the* **domain analysis prompt***:*

- *is_entity — where is_entity(θ) holds if θ is an entity* ■[7]

Although "reasonably" precise, the definition of the concept of **entity** is still not precise enough for us to formalise it. In Sect. 3.8.2 we attempt a series of formalisations of the analysis prompts. This is done on the background of some formalisation (Sect. 3.6) of the ontology being unfolded in this section (i.e., Sect. 3.2). A formalisation that covers the notion of phenomena and entities is not offered.

3.2.3 Endurants and Perdurants

Definition 43 Domain Endurant: By an **endurant** we shall understand an entity that can be observed or conceived and described as a "complete thing" at no matter which given snapshot of time. Were we to "freeze" time we would still be able to observe the entire endurant ■

[5] In a coloured version of this document the description prompts are coloured red and the analysis prompts are coloured blue.
[6] **Definitions** and **examples** are delimited by ■ respectively ■
[7] **Analysis** prompt definitions and **description** prompt definitions and schemes are delimited by ■ respectively ■.

Definition 44 Domain Perdurant: By a **perdurant** we shall understand an entity for which only a fragment exists if we look at or touch them at any given snapshot in time, that is, where we to freeze time we would only see or touch a fragment of the perdurant ■

Analysis Prompt 25 `is_endurant:` *The domain analyser analyses an entity, ϕ, into an endurant as prompted by the* **domain analysis prompt***:*

- `is_endurant`*— e is an endurant if* `is_endurant(e)`[8] *holds.*

`is_entity` *is a* **prerequisite prompt***is_entity for* `is_endurant` ■

Analysis Prompt 26 `is_perdurant:` *The domain analyser analyses an entity ϕ into perdurants as prompted by the* **domain analysis prompt***:*

- `is_perdurant`*— e is a perdurant if* `is_perdurant(e)`[9] *holds.*

`is_entity` *is a* **prerequisite prompt***is_entity for* `is_perdurant` ■

3.2.4 Discrete and Continuous Endurants

Definition 45 Discrete Domain Endurant: By a **discrete endurant** we shall understand an endurant which is separate, individual or distinct in form or concept ■

Definition 46 Continuous Domain Endurant: By a **continuous endurant** we shall understand an endurant which is prolonged, without interruption, in an unbroken series or pattern ■

Analysis Prompt 27 `is_discrete:` *The domain analyser analyse endurants e into discrete entities as prompted by the* **domain analysis prompt***:*

- `is_discrete` *— e is discrete if* `is_discrete(e)`[10] *holds* ■

Analysis Prompt 28 `is_continuous:` *The domain analyser analyse endurants e into continuous entities as prompted by the* **domain analysis prompt***:*

- `is_continuous` *— e is continuous if* `is_continuous(e)`[11] *holds* ■

3.2.5 Parts, Components and Materials

General

Definition 47 Part: By a **part** we shall understand a discrete endurant which the domain engineer chooses to endow with ***internal qualities*** such as unique identification, mereology, and one or more attributes ■

Definition 48 Components: By a **component** we shall understand a discrete endurant which the domain engineer chooses to **not** endow with ***internal qualities*** such as unique identification, mereology, and, even perhaps no attributes ■

Definition 49 Material: By a **material** we shall understand a continuous endurant ■

[8] We formalise `is_endurant` in Sect. 3.8.2 on Page 118.
[9] Since we do not cover perdurants in this chapter we shall also refrain from trying to formalise this prompt.
[10] We formalise `is_discrete` in Sect. 3.8.2 on Page 118.
[11] We formalise `is_continuous` in Sect. 3.8.2 on Page 119.

Part, Component and Material Prompts

Analysis Prompt 29 *is_part: The domain analyser analyse endurants e into part entities as prompted by the* **domain analysis prompt***:*

- *is_part — e is a part if is_part(e)*[12] *holds* ■

Analysis Prompt 30 *is_component: The domain analyser analyse endurants e into part entities as prompted by the* **domain analysis prompt***:*

- *is_component — e is a component if is_component(e)*[13] *holds* ■

Analysis Prompt 31 *is_material: The domain analyser analyse endurants e into material entities as prompted by the* **domain analysis prompt***:*

- *is_material — e is a material if is_material(e)*[14] *holds* ■

There is no difference between is_continuous and is_material, that is is_continuous ≡ is_material. We shall henceforth use is_material.

3.2.6 Atomic and Composite Parts

Definition 50 **Atomic Part: Atomic part**s are those which, in a given context, are deemed to *not* consist of meaningful, separately observable proper *sub-parts* ■

A **sub-part** is a *part* ■

Definition 51 **Composite Part: Composite part**s are those which, in a given context, are deemed to *indeed* consist of meaningful, separately observable proper *sub-parts* ■

Analysis Prompt 32 *is_atomic: The domain analyser analyses a discrete endurant, i.e., a part p into an atomic endurant:*

- *is_atomic(p): p is an atomic endurant if is_atomic(p)*[15] *holds* ■

Analysis Prompt 33 *is_composite: The domain analyser analyses a discrete endurant, i.e., a part p into a composite endurant:*

- *is_composite(p): p is a composite endurant if is_composite(p)*[16] *holds* ■

3.2.7 On Observing Part Sorts

Part Sort Observer Functions

Domain Description Prompt 8 *observe_part_sorts: If is_composite(p) holds, then the analyser "applies" the description language observer prompt*

- *observe_part_sorts(p)*[17]

[12] We formalise is_part in Sect. 3.8.2 on Page 119.
[13] We formalise is_component in Sect. 3.8.2 on Page 119.
[14] We formalise is_material in Sect. 3.8.2 on Page 119.
[15] We formalise is_atomic in Sect. 3.8.2 on Page 119.
[16] We formalise is_composite in Sect. 3.8.2 on Page 119.
[17] We formalise observe_part_sorts in Sect. 3.8.3 on Page 121.

resulting in the analyser writing down the **part sorts and part sort observers** *domain description text according to the following schema:*

3. `observe_part_sorts(p:P)` schema

Narration:
[s] ... narrative text on sorts ...
[o] ... narrative text on sort observers ...
[p] ... narrative text on proof obligations ...
Formalisation:
type
[s] P_1, P_2, ..., P_n
value
[o] **obs_part**_P_i: P → P_i [$1 \leq i \leq m$]
proof obligation [Disjointness of part sorts]
[p] $\mathscr{D}$

$\mathscr{D}$ is some predicate over P_1, P_2, ..., P_n. It expresses their disjointedness. *is_composite is a* **prerequisite prompt***is_composite of* *observe_part_sorts* ■

On Discovering Concrete Part Types

Analysis Prompt 34 *has_concrete_type: The domain analyser may decide that it is expedient, i.e., pragmatically sound, to render a part sort, P, whether atomic or composite, as a concrete type, T. That decision is prompted by the holding of the* **domain analysis prompt***:*

- *has_concrete_type(p).*

is_discrete is a **prerequisite prompt** *of has_concrete_type* ■

Many possibilities offer themselves to model a concrete type as: either a set of abstract sorts, or a list of abstract sorts, or any compound of such sorts. Without loss of generality we suggest, as concrete type, as set of sorts. We have modeled many domains. So far, only the set concrete type has been needed.

Domain Description Prompt 9 *observe_concrete_type: Then the domain analyser applies the* **domain description prompt***:*

- *observe_concrete_type(p)*[18]

to parts p:P which then yield the **part type and part type observers** *domain description text according to the following schema:*

3. `observe_concrete_type(p:P)` schema

Narration:
[t_1] ... narrative text on types ...
[t_2] ... narrative text on types ...
[o] ... narrative text on type observers ...
Formalisation:
type
[t_1] Q
[t_2] T = Q-**set**
value
[o] **obs_part**_T: P → T

[18] We formalise `observe_concrete_type` in Sect. 3.8.3 on Page 121.

Q may be any part sort; `has_concrete_type` *is a* **prerequisite prompt** *observe_part_type of* `observe_part_type` ■

External and Internal Qualities of Parts

By an **external part quality** we shall understand the `is_atomic`, `is_composite`, `is_discrete` and `is_continuous` qualities. By an **internal part quality** we shall understand the part qualities to be outlined in the next sections: `unique identification`, `mereology` and `attributes`. By **part qualities** we mean the sum total of `external endurant` and `internal endurant` qualities.

3.2.8 Unique Part Identifiers

We assume that all parts and components have unique identifiers. It may be, however, that we do not always need to define such a part or component identifier.

Domain Description Prompt 10 `observe_unique_identifier`: *We can, however, always apply the* **domain description prompt**:

- `observe_unique_identifier(pk)`[19]

to parts, p:P, or components, k, resulting in the analyser writing down the ***unique identifier type and observer*** *domain description text according to the following schema:*

3. `observe_unique_identifier(pk:(P|K))` schema

Narration:
[s] ... narrative text on unique identifier sort ...
[u] ... narrative text on unique identifier observer ...
[a] ... axiom on uniqueness of unique identifiers ...
Formalisation:
type
[s] PI, KI
value
[u] **uid**_P: P → PI
[u] **uid**_K: K → KI
axiom
[a] $\mathcal{U}$

$\mathcal{U}$ *is a predicate over part sorts and unique part identifier sorts, respectively component sorts and unique component identifiers. The unique part (component) identifier sort, PI (KI), is unique* ■

3.2.9 Mereology

Part Mereology: Types and Functions

Analysis Prompt 35 `has_mereology`: *To discover necessary, sufficient and pleasing "mereology-hoods" the analyser can be said to endow a truth value* **true** *to the* **domain analysis prompt**:

- `has_mereology`.[20]

Domain Description Prompt 11 `observe_mereology`: *If* `has_mereology(p)` *holds for parts p of type P, then the analyser can apply the* **domain description prompt**:

[19] We formalise `observe_unique_identifier` in Sect. 3.8.3 on Page 121.
[20] We formalise `has_mereology` in Sect. 3.8.2 on Page 120.

- *observe_mereology(p)*[21]

to parts of that type and write down the **mereology types and observers** *domain description text according to the following schema:*

3. observe_mereology(p:P) schema

Narration:
[t] ... narrative text on mereology type ...
[m] ... narrative text on mereology observer ...
[a] ... narrative text on mereology type constraints ...

Formalisation:
type
[t] MT = $\mathscr{E}$(PI1,PI2,...,PIm)
value
[m] **obs_mereo_**P: P → MT
axiom [Well−formedness of Domain Mereologies]
[a] $\mathscr{A}$

MT is a type expression over unique part identifiers. $\mathscr{A}$ *is some predicate over unique part identifiers. The* PI_i *are unique part identifier types* ■

3.2.10 Part, Material and Component Attributes

Domain Description Prompt 12 *observe_attributes*: *The domain analyser experiments, thinks and reflects about attributes of endurants (parts p:P, components, k:K, or materials, m:M). That process is initiated by the* **domain description prompt**:

- *observe_part_attributes(e)*.[22]

The result of that **domain description prompt** *is that the domain analyser cum describer writes down the* **attribute (sorts or) types and observers** *domain description text according to the following schema:*

4. observe_part_attributes(e:(P|K|M)) schema

Narration:
[t] ... narrative text on attribute sorts ...
[o] ... narrative text on attribute sort observers ...
[p] ... narrative text on attribute sort proof obligations ...

Formalisation:
type
[t] A_1, A_2, ..., A_n
value
[o] **attr_**A_i:(P|K|M)→A_i [$1 \le i \le n$]
proof obligation [Disjointness of Attribute Types]
[p] $\mathscr{A}$

The **type** *(or rather sort) definitions:* A_1, A_2, ..., A_n *inform us that the domain analyser has decided to focus on the distinctly named* A_1, A_2, ..., A_n *attributes.*[23] $\mathscr{A}$ *is a predicate over attribute types* A_1, A_2, ..., A_n. *It expresses their Disjointness* ■

[21] We formalise `observe_mereology` in Sect. 3.8.3 on Page 121.
[22] We formalise `observe_attributes` in Sect. 3.8.3 on Page 122.
[23] The attribute type names are not like type names of, for example, a programming language. Instead they are chosen by the domain analyser to reflect on domain phenomena.

3.2.11 Components

We now complement the observe_part_sorts (of Sect. 1.4.1). We assume, without loss of generality, that only atomic parts may contain components. Let p:P be some atomic part.

Analysis Prompt 36 *has_components: The* **domain analysis prompt***:*

- *has_components(p)*[24]

yields **true** *if atomic part p potentially contains components otherwise false* ■

Domain Description Prompt 13 *observe_component_sort: The* **domain description prompt***:*

- *observe_component_sort(p)*[25]

yields the **part component sorts and component observers** *domain description text according to the following schema:*

4. observe_component_sort(p:P) schema

Narration:
[s] ... narrative text on component sort ...
[o] ... narrative text on component sort observer ...
Formalisation:
type
[s] K
value
[o] **obs_comps**: P → K-**set**

Components have unique identifiers and attributes, but no mereology ■

3.2.12 Materials

Only atomic parts may contain materials and materials may contain [atomic] parts.

Part Materials

Let p:P be some atomic part.

Analysis Prompt 37 *has_material: The* **domain analysis prompt***:*

- *has_material(p)*[26]

yields **true** *if the atomic part p:P potentially contains a material otherwise false* ■

Domain Description Prompt 14 *observe_material_sorts: The* **domain description prompt***:*

- *observe_material_sorts(p)*[27]

yields the **part material sort and material observer** *domain description text according to the following schema:*

---
[24] We formalise has_components in Sect. 3.8.2 on Page 120.
[25] We formalise observe_component_sort in Sect. 3.8.3 on Page 122.
[26] We formalise has_materials in Sect. 3.8.2 on Page 120.
[27] We formalise observe_material_sorts in Sect. 3.8.3 on Page 122.

4. observe_material_sorts(p:P) schema

Narration:
[s] ... narrative text on material sort ...
[o] ... narrative text on material sort observer ...
Formalisation:
type
[s] M
value
[o] **obs_mat_sort_**M: P → M

Material Parts

Materials may contain parts. We assume that such parts are always atomic and always of the same sort. **Example:** Pipe parts usually contain oil material. And that oil material may contain pigs which are parts whose purpose it is to clean and inspect (i.e., maintain) pipes ■

Analysis Prompt 38 *has_parts: The* **domain analysis prompt***:*

- *has_parts(m)*[28]

yields **true** *if material m:M potentially contains parts otherwise false* ■

Domain Description Prompt 15 *observe_material_sorts: The* **domain description prompt***:*

- *observe_material_sort(e)*[29]

yields the ***material part sorts and material part observers*** *domain description text according to the following schema:*

4. observe_material_sorts(m:M) schema

Narration:
[s] ... narrative text on material part sort ...
[o] ... narrative text on material part sort observer ...
Formalisation:
type
[s] mP
value
[o] **obs_mat_sort_**mP: M → mP

3.2.13 Components and Materials

Experimental evidence[30] appears to justify the following “limitations”: only atomic parts may contain either at most one material, and always of the same sort, or a set of zero, one or more components, all of the same sort; but not both; materials need not be characterised by unique identifiers; and components and materials need not be endowed with mereologies.

[28] We formalise `has_parts` in Sect. 3.8.2 on Page 120.
[29] We formalise `observe_material_part_sort` in Sect. 3.8.3 on Page 123.
[30] — in the form of more than 20 medium-to-large scale domain models

3.2.14 Discussion

We have covered the analysis and description calculi for endurants. We omit covering analysis and description techniques and tools for perdurants.

3.3 Syntax and Semantics

3.3.1 Form and Content

Section 3.2 appears to be expressed in the syntax of the **R**aise [64] **S**pecification **L**anguage, **RSL** [22]. But it only "appears" so. When, in the "conventional" use of **RSL**, we apply meaning functions, we apply them to syntactic quantities. In Sect. 3.2 the "meaning" functions are the analysis, a.–j., and description, 1.–8., prompts:

`attribute_types`, 29
`has_components`, 21
`has_concrete_type`, 23
`has_materials`, 21
`has_mereology`, 27
`is_animal`, 20, 221
`is_artifactual`, 221
`is_artifact`, 21
`is_atomic`, 19
`is_entity`, 14
`is_human`, 20, 221
`is_living_species`, 17
`is_living`, 221
`is_natural`, 221
`is_physical_part`, 16
`is_physical`, 221
`is_plant`, 20, 221
`observe_endurants`, 22
`is_animal`, 20
`is_artifact`, 21
`is_atomic`, 19
`is_composite`, 19
`is_continuous`, 16
`is_discrete`, 16
`is_endurant`, 15
`is_entity`, 14
`is_human`, 20
`is_living_species`, 17, 20
`is_part`, 18
`is_perdurant`, 15
`is_physical_part`, 16
`is_plant`, 20
`is_structure`, 17
`is_universe_of_discourse`, 14

and

`observe_attributes`, 30
`observe_component_sorts`, 25
`observe_endurant_sorts`, 23
`observe_material_sorts`, 25
`observe_mereology`, 28
`observe_part_type`, 23
`observe_unique_identifier`, 26

The quantities that these prompts are "applied to" are semantic ones, in effect, they are the "ultimate" semantic quantities that we deal with: *the real, i.e., actual domain* entities ! The quantities that these prompts "yield" are syntactic ones ! That is, we have "turned matters inside/out". From semantics we "extract" syntax. The arguments of the above-listed 23 prompts are domain entities, i.e., in principle, in-formalisable things. Their types, typically listed as P, denote possibly infinite classes, $\mathscr{P}$, of domain entities. When we write P we thus mean $\mathscr{P}$.

3.3.2 Syntactic and Semantic Types

When we, classically, define a programming language, we first present its syntax, then it semantics. The latter is presented as two – or three – possibly interwoven texts: the static semantics, i.e., the well-formedness of programs, the dynamic semantics, i.e., the mathematical meaning of programs — with a corresponding proof system being the "third texts". We shall briefly comment on the ideas of static and dynamic semantics. In designing a programming language, and therefore also in narrating and formalising it, one is well advised in deciding first on the semantic types, then on the syntactic ones. With describing [f.ex., manifest] domains, matters are the other way around: The semantic domains are given in the form of the endurants and perdurants; and the syntactic domains are given in the form that we, the humans of the domain, mention in our speech acts [167, 168]. That is, from a study of actual life domains, we extract the essentials that speech acts deal with when these speech acts are concerned with performing or talking about entities in some actual world.

3.3.3 Names and Denotations

Above, we may have been somewhat cavalier with the use of names for sorts and names for their meaning. Being so, i.e., "cavalier", is, unfortunately a "standard" practice. And we shall, regrettably, continue to be cavalier, i.e., "loose" in our use of names of syntactic "things" and names for the denotation of these syntactic "things". The context of these uses usually makes it clear which use we refer to: a syntactic use or a semantic one. As from Sect. 3.6 we shall be more careful distinguishing clearly between the names of sorts and the values of sorts, i.e., between syntax and semantics.

3.4 A Model of the Domain Analysis & Description Process

3.4.1 Introduction

A Summary of Prompts

In Sect. 3.3.1 we listed the two classes of prompts: the ***domain [endurant] analysis prompts***: and the ***domain [endurant] description prompts***: These prompts are "imposed" upon the domain by the domain analyser cum describer. They are "figuratively" applied to the domain. Their orderly, sequenced application follows the method hinted at in the previous section, detailed in [2, ***Manifest Domains: Analysis & Description***]. This process of application of prompts will be expressed in a pseudo-formal notation in this section. The notation looks formal but since we have not formalised these prompts it is only pseudo-formal. We formalise these prompts in Sect. 3.8.

Preliminaries

Let P be a sort, that is, a collection of endurants. By P we shall understand both a syntactic quantity: the name of P, and a semantic quantity, the type (of all endurant values of type) P. By ιp:P we shall understand a semantic quantity: an (arbitrarily selected) endurant in P. To guide our analysis & description process we decompose it into steps. Each step "handles" a part sort p:P or a material sort m:M or a component sort k:K. Steps handling discovery of composite part sorts generates a set of part sort names P_1, P_2, ..., P_n:PNm. Steps handling discovery of atomic part sorts may generate a material sort name, m:MNm, or component sort name, k:KNm. The part, material and component sort names are put in a reservoir for *sorts to be inspected*. Once handled, the sort name is removed from that reservoir. Handling of material sorts besides discovering their attributes may involve the discovery of further part sorts — which we assume to be atomic. Each domain description prompt results in domain specification text (here we show only the formal texts, not the narrative texts) being deposited in the domain description reservoir, a global variable τ. We do not formalise this text. Clauses of the form `observe_XXX(p)`, where `XXX` ranges over `part_sorts`, `concrete_type`, `unique_identifier`, `mereology`, `part_attributes`, `part_component_sorts`, `part_material_sorts`, and `material_part_sorts`, stand for ″text″ generating functions. They are defined in Sect. 3.8.3.

Initialising the Domain Analysis & Description Process

We remind the reader that we are dealing only with endurant domain entities. The domain analysis approach covered in Sect. 3.2 was based on decomposing an understanding of a domain from the "overall domain" into its components, and these, if not atomic, into their sub-domains. So we need to initialise the domain analysis & description process by selecting (or choosing) the domain Δ. Here is how we think of that "initialisation" process. The domain analyser & describer spends some time focusing on the domain, maybe at the "white board"[31], rambling, perhaps in an un-structured manner, across its domain, Δ, and its sub-domains. Informally jotting down more-or-less final sort names, building, in the domain analyser & describer's mind an image of that domain. After some time doing this the domain analyser & describer is ready. An image of the domain includes the or a domain endurant, δ:Δ. Let Δnm be the name of the sort Δ. That name may be either a part sort name, or a material sort name, or a component sort name.

3.4.2 A Model of the Analysis & Description Process

A Process State

140 Let Nm denote either a part or a material or a component sort name.
141 A global variable αps will accumulate all the sort names being discovered.
142 A global variable νps will hold names of sorts that have been "discovered", but have yet to be analysed & described.

type
140. Nm = PNm | MNm | KNm
variable
141. αps := [Δnm] **type** Nm-**set**
142. νps := [Δnm] **type** Nm-**set**

We shall explain the use of [...]s and operations on the above variables in Sect. 3.4.3 on Page 108. Each iteration of the "root" function, analyse_and_describe_endurant_sort(Nm,ι:nm), as we shall call it, involves the selection of a sort (value) (which is that of either a part sort or a material sort) with this sort (value) then being removed.

143 The selection occurs from the global state component νps (hence: ()) and changes that state (hence **Unit**).

value
143. sel_and_rem_Nm: **Unit** $\to$ Nm
143. sel_and_rem_Nm() $\equiv$ **let** nm:Nm • nm $\in$ νps **in** νps := νps $\setminus$ {nm} ; nm **end**; **pre**: νps $\neq$ {}

A Technicality

144 The main analysis & description functions of the next sections, except the "root" function, are all expressed in terms of a pair, (nm,val):NmVAL, of a sort name and an endurant value of that sort.

type
144. NmVAL = (PNm$\times$PVAL) | (MNm$\times$MVAL) | (KNm$\times$KVAL)

Analysis & Description of Endurants

145 To analyse and describe endurants means to first
146 examine those endurants which have yet to be so analysed and described
147 by selecting (and removing from νps) a yet un-examined sort nm;

[31] Here 'white board' is a conceptual notion. It could be physical, it could be yellow "post-it" stickers, or it could be an electronic conference "gadget".

148 then analyse and describe an endurant entity (ι:nm) of that sort — this analysis, when applied to composite parts, leads to the insertion of zero[32] or more sort names[33].

As was indicated in Sect. 3.2, the mereology of a part, if it has one, may involve unique identifiers of any part sort, hence must be done after all such part sort unique identifiers have been identified. Similarly for attributes which also may involve unique identifiers,

149 then, if it has a mereology,
150 to analyse and describe the mereology of each part sort,
151 and finally to analyse and describe the attributes of each sort.

value
145. analyse_and_describe_endurants: **Unit** → **Unit**
145. analyse_and_describe_endurants() ≡
146. **while** ∼is_empty(νps) **do**
147. **let** nm = sel_and_rem_Nm() **in**
148. analyse_and_describe_endurant_sort(nm,ι:nm) **end end** ;
149. **for all** nm:PNm • nm ∈ αps **do if** **has_mereology(nm,ι:nm)**[34]
150. **then** **observe_mereology**(nm,ι:nm)[35] **end end**
151. **for all** nm:Nm • nm ∈ αps **do** **observe_attributes**(nm,ι:nm)[36] **end**

The ι:nm of Items 148, 149, 150 and 151 are crucial. The domain analyser is focused on (part or material or component) sort nm and is "directed" (by those items) to choose (select) an endurant (a part or a material or component) ι:nm of that sort.

152 To analyse and describe an endurant
153 is to find out whether it is a part. If so then it is to analyse and describe it.
154 If it instead is a material, then to analyse and describe it as a material.
155 If it instead is a component, then to analyse and describe it as a component.

value
152. analyse_and_describe_endurant_sort: NmVAL → **Unit**
152. analyse_and_describe_endurant_sort(nm,val) ≡
153. **is_part**(nm,val)[37] →[38] analyse_and_describe_part_sorts(nm,val),
154. **is_material**(nm,val)[39] → **observe_material_part_sort**(nm,val)[40],
155. **is_component**(nm,val)[41] → **observe_component_sort**(nm,val)[42]

156 The analysis and description of a part
157 first describe its unique identifier.
158 If the part is atomic it is analysed and described as such;
159 If composite it is analysed and described as such.
160 Part p must be discrete.

[32] If the sub-parts of ι:nm are all either atomic and have no materials or components or have already been analysed, then no new sort names are added to the repository νps).
[33] These new sort names are then "picked-up" for sort analysis &c. in a next iteration of the while loop.
[36] We formalise `has_mereology` in Sect. 3.8.2 on Page 120.
[36] We formalise `observe_mereology` in Sect. 3.8.3 on Page 121.
[36] We formalise `observe_attributes` in Sect. 3.8.3 on Page 122.
[42] We formalise `is_part` in Sect. 3.8.2 on Page 119.
[42] The conditional clause: $\text{cond}_1 \rightarrow \text{clau}_1, \text{cond}_2 \rightarrow \text{clau}_2, \ldots, \text{cond}_n \rightarrow \text{clau}_n$
is same as **if** cond_1 **then** clau_1 **else if** cond_2 **then** clau_2 **else** ... **if** cond_n **then** clau_n **end end** ... **end** .
[42] We formalise `is_material` in Sect. 3.8.2 on Page 119.
[42] We formalise `observe_material_part_sort` in Sect. 3.8.3 on Page 123.
[42] We formalise `is_component` in Sect. 3.8.2 on Page 119.
[42] We formalise `observe_component_sort` in Sect. 3.8.3 on Page 122.

value
156. analyse_and_describe_part_sorts: NmVAL → **Unit**
156. analyse_and_describe_part_sorts(nm,val) ≡
157. observe_unique_identifier(nm,val)[43];
158. is_atomic(nm,val)[44]→ analyse_and_describe_atomic_part(nm,val),
159. is_composite(nm,val)[45]→ analyse_and_describe_composite_parts(nm,val)
160. **pre**: is_discrete(nm,val)[46]

161 To analyse and describe an atomic part is to inquire whether
 a it embodies materials, then we analyse and describe these;
 b and if it further has components, then we describe their sorts.

value
161. analyse_and_describe_atomic_part: NmVAL → **Unit**
161. analyse_and_describe_atomic_part(nm,val) ≡
161a. **if** has_material(nm,val)[47] **then** observe_part_material_sort(nm,val)[48] **end** ;
161b. **if** has_components(nm,val)[49] **then** observe_part_component_sort(nm,val)[50] **end**

162 To analyse and describe a composite endurant of sort nm (and value val)
163 is to analyse if the sort has a concrete type
164 then we analyse and describe that concrete sort type
165 else we analyse and describe the abstract sort.

value
162. analyse_and_describe_composite_endurant: NmVAL → **Unit**
162. analyse_and_describe_composite_endurant(nm,val) ≡
163. **if** has_concrete_type(nm,val)[51]
164. **then** observe_concrete_type(nm,val)[52]
165. **else** observe_abstract_sorts(nm,val)[53]
163. **end**
162. **pre** is_composite(nm,val)[54]

We do not associate materials or components with composite parts.

3.4.3 Discussion of The Process Model

The above model lacks a formal understanding of the individual prompts as listed in Sect. 3.4.1; such an understanding is attempted in Sect. 3.8.

[46] We formalise `observe_unique_identifier` in Sect. 3.8.3 on Page 121.
[46] We formalise `is_atomic` in Sect. 3.8.2 on Page 119.
[46] We formalise `is_composite` in Sect. 3.8.2 on Page 119.
[46] We formalise `is_discrete` in Sect. 3.8.2 on Page 118.
[50] We formalise `has_material` in Sect. 3.8.2 on Page 120.
[50] We formalise `observe_part_material_sort` in Sect. 3.8.3 on Page 122.
[50] We formalise `has_components` in Sect. 3.8.2 on Page 120.
[50] We formalise `observe_part_component_sort` in Sect. 3.8.3 on Page 122.
[51] We formalise `has_concrete_type` in Sect. 3.8.2 on Page 119.
[51] We formalise `observe_concrete_type` in Sect. 3.8.3 on Page 121.
[51] We formalise `observe_part_sorts` in Sect. 3.8.3 on Page 121.
[51] We formalise `is_composite` in Sect. 3.8.2 on Page 119.

Termination

The sort name reservoir ν**ps** is "reduced" by one name in each iteration of the **while** loop of the analyse_and_describe_endurants, cf. Item 147 on Page 105, and is augmented by new part, material and component sort names in some iterations of that loop. We assume that (manifest) domains are finite, hence there are only a finite number of domain sorts. It remains to (formally) prove that the analysis & description process terminates.

Axioms and Proof Obligations

We have omitted, from Sect. 3.2, treatment of axioms concerning well-formedness of parts, materials and attributes and proof obligations concerning disjointedness of observed part and material sorts and attribute types. [2] exemplifies axioms and sketches some proof obligations.

Order of Analysis & Description: A Meaning of '$\oplus$'

The variables αps, νps and τ can be defined to hold either sets or lists. The operator $\oplus$ can be thought of as either set union ($\cup$ and [...]$\equiv$\{...\}) — in which case the domain description text in τ is a set of domain description texts — or as list concatenation ($\widehat{\ }$ and [...]$\equiv\langle$...$\rangle$) of domain description texts. The list operator $\ell_1 \oplus \ \ell_2$ now has at least two interpretations: either $\ell_1\widehat{\ }\ell_2$ or $\ell_2\widehat{\ }\ell_1$. Thus, in the case of lists, the $\oplus$, i.e., $\widehat{\ }$, does not (suffix or prefix) append ℓ_2 elements already in ℓ_1. The sel_and_rem_Nm function on Page 105 applies to the set interpretation. A list interpretation is:

value
147. sel_and_rem_Nm: **Unit** $\rightarrow$ Nm
147. sel_and_rem_Nm() $\equiv$ **let** nm = **hd** ν ps **in** ν ps := **tl** ν ps; nm **end**; **pre**: νps $\neq$<>

In the first case ($\ell_1\widehat{\ }\ell_2$) the analysis and description process proceeds from the root, breadth first, In the second case ($\ell_2\widehat{\ }\ell_1$) the analysis and description process proceeds from the root, depth first. .

Laws of Description Prompts

The domain 'method' outlined in the previous section suggests that many different orders of analysis & description may be possible. But are they ? That is, will they all result in "similar" descriptions ? If, for example, $\mathcal{D}_a$ and $\mathcal{D}_b$ are two domain description prompts where $\mathcal{D}_a$ and $\mathcal{D}_b$ can be pursued in any order will that yield the same description ? And what do we mean by 'can be pursued in any order', and 'same description' ? Let us assume that sort P decomposes into sorts P_a and P_b (etcetera). Let us assume that the domain description prompt $\mathcal{D}_a$ is related to the description of P_a and $\mathcal{D}_b$ to P_b. Here we would expect $\mathcal{D}_a$ and $\mathcal{D}_b$ to commute, that is $\mathcal{D}_a;\mathcal{D}_b$ yields same result as does $\mathcal{D}_b;\mathcal{D}_a$. In [169] we made an early exploration of such laws of domain description prompts. To answer these questions we need a reasonably precise model of domain prompts. We attempt such a model in Sect. 3.8. But we do not prove theorems.

3.5 A Domain Analyser's & Describer's Domain Image

Assumptions: We assume that the domain analysers cum describers are well educated and well trained in the domain analysis & description techniques such as laid out in [2]. This assumption entails that the domain analysis & description development process is structured in sequences of alternating (one or more) analysis prompts and description prompts. We refer to Footnote 2 (Page 93) as well as to the discussion, *"Towards a methodology of manifest domain analysis & description"* of [2, Sect. 1.6]. We further assume that the domain analysers cum describers makes repeated attempts to analyse & describe a domain. We assume, further, that it is "the same domain" that is being analysed & described – two, three or more times, "all-over", before commitment is made to attempt a – hopefully – final analysis & description[52], from

[52] – and if that otherwise planned, final analysis & description is not satisfactory, then yet one more iteration is taken.

"scratch", that is, having "thrown away", previous drafts[53]. We then make the further assumption, as this iterative analysis & description process proceeds, from iteration i to $i+1$, that each and all members of the analysis & description group are forming, in their minds (i.e., brains) an "image" of the domain being analysed. As iterations proceed one can then say that what is being analysed & described increasingly becomes this 'image' as much as it is being the domain — which we assume is not changing across iterations. The iterated descriptions are now postulated to converge: a "final" iteration "differs" only "immaterially." from the description of the "previous" iteration.

• • •

The Domain Engineer's Image of Domains: In the opening ('Assumptions') of this section, i.e., above, we hinted at "an image", in the minds of the domain analysers & describers, of the domain being researched and for which a description document is being engineered. In this paragraph we shall analyse what we mean by such a image. Since the analysis & description techniques are based on applying the analysis and description prompts (reviewed in Sect. 3.2) we can assume that the image somehow relates to the 'ontology' of the domain entities, whether endurants or perdurants, such as graphed in Fig. 3.1. Rather than further investigating (i.e., analysing / arguing) the form of this, until now, vague notion, we simply conjecture that the image is that of an **'abstract syntax of domain types'.**

• • •

The Iterative Nature of The Description Process: Assume that the domain engineers are analysing & describing a particular endurant; that is, as we shall understand it, are examining a given endurant node in the ***domain description tree*** ! The **domain description tree** is defined by the facts that composite parts have sub-parts which may again be composite (tree branches), ending with atomic parts (the leaves of the tree) but not "circularly", i.e. recursively ■

To make this claim: ***the domain analysers cum describers are examining a given endurant node in the domain description tree*** amounts to saying that ***the domain engineers have in their mind a reasonably "stable" "picture" of a domain in terms of a domain description tree.***

We need explain this assumption. In this assumption there is "buried" an understanding that the domain analysers cum describers during the — what we can call "the final" — domain analysis & description process, that leads to a "deliverable" domain description, are not investigating the domain to be described for the first time. That is, we certainly assume that any "final" domain analysis & description process has been preceded by a number of iterations of "trial" domain analysis & description processes.

Hopefully this iteration of experimental domain analysis & description processes converges. Each iteration leads to some domain description, that is, some domain description tree. A first iteration is thus based on a rather incomplete domain description tree which, however, "quickly" emerges into a less incomplete one in that first iteration. When the domain engineers decide that a "final" iteration seems possible then a "final" description emerges If acceptable, OK, otherwise yet an "final" iteration must be performed. Common to all iterations is that the domain analysers cum describers have in mind some more-or-less "complete" domain description tree and apply the prompts introduced in Sect. 3.4.

3.6 Domain Types

There are two kinds of types associated with domains: the syntactic types of endurant descriptions, and the semantic types of endurant values.

3.6.1 Syntactic Types: Parts, Materials and Components

In this section we outline an **'abstract syntax of domain types'.** In Sect. 3.6.1 we introduce the concept of sort names. Then, in Sects. 3.6.1–3.6.1, we describe the syntax of part, material and component types. Finally, in Sects. 3.6.1–3.6.1, we analyse this syntax with respect to a number of well-formedness criteria.

[53] It may be useful, though, to keep a list of the names of all the endurant parts and their attribute names, should the group members accidentally forget such endurants and attributes: at least, if they do not appear in later document iterations, then it can be considered a deliberate omission.

Syntax of Part, Material and Component Sort Names

166 There is a further undefined sort, N, of tokens (which we shall consider atomic and the basis for forming names).
167 From these we form three disjoint sets of sort names:
  a part sort names,
  b material sort names and
  c component sort names,

```
166    N
167a   PNm  :: mkPNm(N)
167b   MNm  :: mkMNm(N)
167c   KNm  :: mkKNm(N)
```

An Abstract Syntax of Domain Endurants

168 We think of the types of parts, materials and components to be a map from their type names to respective type expressions.
169 Thus part types map part sort names into part types;
170 material types map material sort names into material types; and
171 component types map components sort names into component types.
172 Thus we can speak of endurant types to be either part types or material types or component types.
173 A part type expression is either an atomic part type expression or is a composite part type expression or is a concrete composite part type expression.
174 An atomic part type expression consists of a type expression for the qualities of the atomic part and, optionally, a material type name or a component type name (cf. Sect. 3.2.13).
175 An abstract composite part type expression consists of a type expression for the qualities of the composite part and a finite set of one or more part type names.
176 A concrete composite part type expression consists of a type expression for the qualities of the part and a part sort name standing for a set of parts of that sort.
177 A material part type expression consists of of a type expression for the qualities of the material and an optional part type name.
178 We omit consideration of component types.

Endurants: Syntactic Types

```
168        TypDef  =  PTypes ∪ MTypes ∪ KTypes
169        PTypes  =  PNm →m PaTyp
170        MTypes  =  MNm →m MaTyp
171        KTypes  =  KNm →m KoTyp
172       ENDType  =  PaTyp | MaTyp | KoTyp
173         PaTyp ==  AtPaTyp | AbsCoPaTyp | ConCoPaTyp
174       AtPaTyp  :: mkAtPaTyp(s_qs:PQ,s_omkn:({|"nil"|}|MNn|KNm))
175 AbsCoPaTyp     :: mkAbsCoPaTyp(s_qs:PQ,s_pns:PNm-set)
175                   axiom ∀ mkAbsCoPaTyp(pq,pns):AbsCoPaTyp • pns ≠ {}
176 ConCoPaTyp     :: mkConCoPaTyp(s_qs:PQ,s_p:PNm)
177         MaTyp  :: mkMaTyp(s_qs:MQ,s_opn:({|"nil"|}|PNm))
178         KoTyp  :: mkKoTyp(s_qs:KQ)
```

Quality Types

179 There are three aspects to part qualities: the type of the part unique identifiers, the type of the part mereology, and the name and type of attributes.
180 The type unique part identifiers is a not further defined atomic quantity.
181 A part mereology is either "nil" or it is an expression over part unique identifiers, where such expressions are those of either simple unique identifier tokens, or of set, or otherwise over simple unique identifier tokens, or ..., etc.
182 The type of attributes pairs distinct attribute names with attribute types —
183 both of which we presently leave further undefined.
184 Material attributes is the only aspect to material qualities.
185 Components have unique identifiers. Component attribute types are left undefined.

Qualities: Syntactic Types

179 PQ = s_ui:UI×s_me:ME×s_atrs:ATRS}
180 UI
181 ME == "nil"|mkUI(s_ui:UI)|mkUIset(s_uil:UI)|...
182 ATRS = ANm $\overrightarrow{m}$ ATyp
183 ANm, ATyp
184 MQ = s_atrs:ATRS
185 KQ = s_uid:UI × s_atrs:ATRS

It is without loss of generality that we do not distinguish between part and material attribute names and types. Material and component attributes do not refer to any part or any other material and component attributes.

Well-formed Syntactic Types

Well-formed Definitions

186 We need define an auxiliary function, names, which, given an endurant type expression, yields the sort names that are referenced immediately by that type.
- a If the endurant type expression is that of an atomic part type then the sort name is that of its optional component sort.
- b If an abstract composite part type then the sort names of its parts.
- c If a concrete composite part type then the sort name is that of the sort of its set of parts.
- d If a material type then sort name is that of the sort of its optional parts.
- e Component sorts have no references to other sorts.

value
186. names: TypDef → (PNm|MNm|KNm) → (PNm|MNm|KNm)**-set**
186. names(td)(n) ≡
186. ∪ { ns | ns:(PNm|MNm|KNm)**-set** •
186. **case** td(n) **of**
186a. mkAtPaTyp(_,n′) → ns={n′},
186b. mkAbsCoPaTyp(_,ns′) → ns=ns′,
186c. mkConCoPaTyp(_,pn) → ns={pn},
186d. mkMaTyp(_,n′) → ns={n′},
186e. mkKoTyp(_) → ns={}
186. **end** }

187 Endurant sort names being referenced in part types, PaTyp, in material types, MaTyp, and in component types, KoTyp, of the typdef:Typdef definition, *must be defined in* the defining set, **dom** typdef, of the typdef:Typdef definition.

value
187. wf_TypDef_1: TypDef → **Bool**
187. wf_TypDef_1(td) ≡ ∀ n:(PNm|MNm|CNm)•n ∈ **dom** td ⇒ names(td)(n)⊆**dom** td

Perhaps Item 187. should be sharpened:

188 from *"must be defined in"* [187.] to *"must be equal to"*:

188. ∧ ∀ n:(PNm|MNm|CNm) • n ∈ **dom** td ⇒ names(td)(n)=**dom** td

No Recursive Definitions

189 Type definitions must not define types recursively.
 a A type definition, typdef:TypDef, defines, typically composite part sorts, named, say, *n*, in terms of other part (material and component) types. This is captured in the
 - mncs (Item 174),
 - pns (Item 175),
 - p (Item 176) and
 - pns (Item 177),

 selectable elements of respective type definitions. These elements identify type names of materials and components, parts, a part, and parts, respectively. None of these names may be *n*.
 b The identified type names may further identify type definitions none of whose selected type names may be *n*.
 c And so forth.

value
189. wf_TypDef_2: TypDef → **Bool**
189. wf_TypDef_2(typdef) ≡ ∀ n:(PNm|MNm)• n ∈ **dom** typdef ⇒ n ∉ type_names(typdef)(n)

189a. type_names: TypDef → (PNm|MNm) → (PNm|MNm)**-set**
189a. type_names(typdef)(nm) ≡
189b. **let** ns = names(typdef)(nm) ∪ { names(typdef)(n) | n:(PNm|MNm) • n ∈ ns } **in**
189c. nm ∉ ns **end**

ns is the least fix-point solution to the recursive definition of ns.

3.6.2 Semantic Types: Parts, Materials and Components

Part, Material and Component Values

We define the values corresponding to the type definitions of Items 166.–185, structured as per type definition Item 172 on Page 110.

190 An endurant value is either a part value, a material values or a component value.
191 A part value is either the value of an atomic part, or of an abstract composite part, or of a concrete composite part.
192 A atomic part value has a part quality value and, optionally, either a material or a possibly empty set of component values (cf. Sect. 3.2.13).
193 An abstract composite part value has a part quality value and of at least (hence the **axiom**) of
194 one or more (distinct part type) part values.
195 A concrete composite part value has a part quality value and a set of part values.
196 A material value has a material quality value (of material attributes) and a (usually empty) finite set of part values.
197 A component value has a component quality value (of a unique identifier and component attributes).

Endurant Values: Semantic Types

190 ENDVAL = PVAL | MVAL | KVAL
191 PVAL == AtPaVAL|AbsCoPVAL|ConCoPVAL
192 AtPaVAL :: mkAtPaVAL(s_qval:PQVAL,s_omkvals:({|"nil"|}|MVAL|KVAL-**set**))
193 AbsCoPVAL :: mkAbsCoPaVAL(s_qval:PQVAL,s_pvals:(PNm $\overrightarrow{m}$PVAL))
194 **axiom** ∀ mkAbsCoPaVAL(pqs,ppm):AbsCoPVAL • ppm ≠ []
195 ConCoPVAL :: mkConCoPaVAL(s_qval:PQVAL,s_pvals:PVAL-**set**)
196 MVAL :: mkMaVAL(s_qval:MQVAL,s_pvals:PVAL-**set**)
197 KVAL :: mkKoVAL(s_qval:KQVAL)

Quality Values

198 A part quality value consists of three qualities:
199 a unique identifier type name, resp. value, which are both further undefined (atomic value) tokens;
200 a mereology expression, resp. value, which is either a single unique identifier (type, resp.) value, or a set of such unique identifier (types, resp.) values, or ...; and
201 an aggregate of attribute values, modeled here as a map from attribute type names to attribute values.
202 In this chapter we leave attribute type names and attribute values further undefined.
203 A material quality value consists just of an aggregate of attribute values, modeled here as a map from attribute type names to attribute values.
204 A component quality value consists of a pair: a unique identifier value and an aggregate of attribute values, modeled here as a map from attribute type names to attribute values.

Qualities: Semantic Types

198 PQVAL = UIVAL×MEVAL×ATTRVALS
199 UIVAL
200 MEVAL == mkUIVAL(s_ui:UIVAL)|mkUIVALset(s_uis:UIVAL-**set**)|...
201 ATTRVALS = ANm $\overrightarrow{m}$AVAL
202 ANm, AVAL
203 MQVAL = ATTRVALS
204 KQVAL = UIVAL×ATTRVALS

We have left to define the values of attributes. For each part and material attribute value we assume a finite set of values. And for each unique identifier type (i.e., for each UI) we likewise assume a finite set of unique identifiers of that type. The value sets may be large. These assumptions help secure that the set of part, material and component values are also finite.

Type Checking

For part, material and component qualities we postulate an overloaded, simple type checking function, type_of, that applies to unique identifier values, uiv:UIVAL, and yield their unique identifier type name, ui:UI, to mereology values, mev:MEVAL, and yield their mereology expression, me:ME, and to attribute values, AVAL and ATTRSVAL, and yield their types: ATyp, respectively (ANm $\overrightarrow{m}$AVAL)→(ANm $\overrightarrow{m}$ATyp). Since we have let undefined both the syntactic type of attributes types, ATyp, and the semantic type of attribute values, AVAL, we shall leave type_of further unspecified.

value type_of: (UIVAL→UI)|(MEVAL→ME)|(AVAL→ATyp)|((ANm $\overrightarrow{m}$AVAL)→(ANm $\overrightarrow{m}$ATyp))

The definition of the syntactic type of attributes types, ATyp, and the semantic type of attribute values, AVAL, is a simple exercise in a first-year programming language semantics course.

3.7 From Syntax to Semantics and Back Again !

The two syntaxes of the previous section: that of the ***syntactic domain***s, formula Items 166–185 (Pages 110–111), and that of the ***semantic domain***s, formula Items 190–204 (Pages 112–113), are not the syntaxes of domain descriptions, but of some aspects common to all domain descriptions developed according to the calculi of this chapter. The ***syntactic domain*** formulas underlie ("are common to", i.e., "abstracts") aspects of all domain descriptions. The ***semantic domain*** formulas underlay ("are common to", i.e., "abstracts") aspects of the meaning of all domain descriptions. These two syntaxes, hence, are, so-to-speak, in the minds of the domain engineer (i.e., the analyser cum describer) while analysing the domain.

3.7.1 The Analysis & Description Prompt Arguments

The domain engineer analyse & describe endurants on the basis of a sort name i.e., a piece of syntax, nm:Nm, and an endurant value, i.e. a "piece" of semantics, val:VAL, that is, the arguments, (nm,ι:nm), of the analysis and description prompts of Sect. 3.4. Those two quantities are what the domain engineer are "operating" with, i.e., are handling: One is tangible, i.e. can be noted (i.e., "scribbled down"), the other is "in the mind" of the analysers cum describers. We can relate the two in terms of the two syntaxes, the syntactic types, and the meaning of the semantic types. But first some "preliminaries".

3.7.2 Some Auxiliary Maps: Syntax to Semantics and Semantics to Syntax

We define two kinds of map types:

205 Nm_to_ENDVALS are maps from endurant sort names to respective sets of all corresponding endurant values of, and
206 ENDVAL_to_Nm are maps from endurant values to respective sort names.

type
205. Nm_to_ENDVALS = (PNm $\overrightarrow{m}$PVAL-**set**)∪(MNm $\overrightarrow{m}$MVAL-**set**)∪(KNm $\overrightarrow{m}$KVAL-**set**)
206. ENDVAL_to_Nm = (PVAL $\overrightarrow{m}$PNm)∪(MVAL $\overrightarrow{m}$MNm)∪(KVAL $\overrightarrow{m}$KNm)

We can derive values of these map types from type definitions:

207 a function, typval, from type definitions, typdef:TypDef to Nm_to_ENDVALS, and
208 a function valtyp, from Nm_to_ENDVALS, to ENDVAL_to_Nm.

value
207. typval: TypDef $\stackrel{\sim}{\to}$ Nm_to_ENDVALS
208. valtyp: Nm_to_ENDVALS $\stackrel{\sim}{\to}$ ENDVAL_to_Nm

209 The typval function is defined in terms of a meaning function M (let ρ:ENV abbreviate Nm_to_ENDVALS:

209. M: (PaTyp→ENV$\stackrel{\sim}{\to}$PVAL-**set**)|(MaTyp→ENV$\stackrel{\sim}{\to}$MVAL-**set**)|(KoTyp→ENV$\stackrel{\sim}{\to}$KVAL-**set**)

207. typval(td) ≡ **let** ρ = [n↦M(td(n))(ρ)|n:(PNm|MNm|KNm)•n ∈ **dom** td] **in** ρ **end**
208. valtyp(ρ) ≡ [v↦n|n:(PNm|MNm|CNm),v:(PVAL|MVAL|KVAL)•n ∈ **dom** ρ∧v ∈ ρ(n)]

The environment, ρ, of typval, Item 207, is the least fix point of the recursive equation

- 207. **let** ρ = [n↦M(td(n))(ρ)|n:(PNm|MNm|CNm)•n ∈ **dom** td] **in** ...

The M function is defined next.

3.7.3 M: A Meaning of Type Names

Preliminaries

The typval function provides for a homomorphic image from TypDef to TypNm_to_VALS. So, the narrative below, describes, item-by-item, this image. We refer to formula Items 207 and 209 on the preceding page. The definition of M is decomposed into five sub-definitions, one for each kind of endurant type:

- Atomic parts: mkAtPaTyp(s_qs:(UI×ME×ATRS),s_omkn:({|"nil"|}|MNn|KNm)), Items 210;
- Abstract composite parts: mkAbsCoPaTyp(s_qs:PQ,s_pns:PNm-**set**), 211;
- Concrete composite parts: mkConCoPaTyp(s_qs:PQ,s_p:PNm), Items 212 on the following page;
- Materials: mkMaTyp(s_qs:MQ,s_opn:({|"nil"|}|PNm)), Items 213 on the next page; and
- Components: mkKoTyp(s_qs:KQ), Items 214 on Page 117.

We abbreviate, by ENV, the M function argument, ρ, of type: Nm_to_ENDVALS.

Atomic Parts

210 The meaning of an atomic part type expression,
Item 174. mkAtPaTyp((ui,me,attrs),omkn)
in mkAtPaTyp(s_qs:PQ,s_omkn:({|"nil"|}|MNn|KNm)),
is the set of all atomic part values,
Items 192., 198., 201. mkAtPaVAL((uiv,mev,attrvals),omkval)
in mkAtPaVAL(s_qval:(UIVAL×MEVAL×(ANm $\overrightarrow{m}$AVAL)),
s_omkvals:({|"nil"|}|MVAL|KVAL-**set**)).
  a uiv is a value in UIVAL of type ui,
  b mev is a value in MEVAL of type me,
  c attrvals is a value in (ANm $\overrightarrow{m}$AVAL) of type (ANm $\overrightarrow{m}$ATyp), and
  d omkvals is a value in ({|"nil"|}|MVAL|KVAL-**set**):
    i either ''nil'',
    ii or one material value of type MNm,
    iii or a possibly empty set of component values, each of type KNm.

```
210.      M: mkAtPaTyp((UI×ME×(ANm →m ATyp))×({|"nil"|}|MVAL|KVAL-set))→ENV →~ PVAL-set
210.      M(mkAtPaTyp((ui,me,attrs),omkn))(ρ) ≡
210.          { mkATPaVAL((uiv,mev,attrval),omkvals) |
210a.             uiv:UIVAL•type_of(uiv)=ui,
210b.             mev:MEVAL•type_of(mev)=me,
210c.             attrval:(ANm →m AVAL)•type_of(attrval)=attrs,
210d.             omkvals: case omkn of
210(d)i.                 "nil" → "nil",
210(d)ii.                mkMNn(_) → mval:MVAL•type_of(mval)=omkn,
210(d)iii.               mkKNm(_) → kvals:KVAL-set•kvals⊆{kv|kv:KVAL•type_of(kval)=omkn}
210d.             end }
```

Formula terms 210a–210(d)iii express that any applicable uiv is combined with any applicable mev is combined with any applicable attrval is combined with any applicable omkvals.

Abstract Composite Parts

211 The meaning of an abstract composite part type expression,
Item 175. mkAbsCoPaTyp((ui,me,attrs),pns)
in mkAbsCoPaTyp(s_qs:PQ,s_pns:PNm-**set**),
is the set of all abstract, composite part values,
Items 193., 198., 201., mkAbsCoPaVAL((uiv,mev,attrvals),pvals)
in mkAbsCoPaVAL(s_qval:(UIVAL×MEVAL×(ANm $\overrightarrow{m}$AVAL)),s_pvals:(PNm $\overrightarrow{m}$PVAL)).

a uiv is a value in UIVAL of type ui: UI,
b mev is a value in MEVAL of type me: ME,
c attrvals is a value in (ANm $\overrightarrow{m}$ AVAL) of type (ANm $\overrightarrow{m}$ ATyp), and
d pvals is a map of part values in (PNm $\overrightarrow{m}$ PVAL), one for each name, pn:PNm, in pns such that these part values are of the type defined for pn.

211. M: mkAbsCoPaTyp((UI×ME×(ANm $\overrightarrow{m}$ ATyp)),PNm-**set**) → ENV $\xrightarrow{\sim}$ PVAL-**set**
211. M(mkAbsCoPaTyp((ui,me,attrs),pns))(ρ) ≡
211. { mkAbsCoPaVAL((uiv,mev,attrvals),pvals) |
211a. uiv:UIVAL•type_of(uiv)=ui
211b. mev:MEVAL•type_of(mev)=me,
211c. attrvals:(ANm $\overrightarrow{m}$ ATyp)•type_of(attrsval)=attrs,
211d. pvals:(PNm $\overrightarrow{m}$ PVAL)•pvals∈{[pn↦pval|pn:PNm,pval:PVAL•pn∈ pns∧pval∈ρ(pn)]} }

Concrete Composite Parts

212 The meaning of a concrete composite part type expression, Item 176.
mkConCoPaTyp((ui,me,attrs),pn)
in mkConCoPaTyp(s_qs:(UI×ME×(ANm $\overrightarrow{m}$ ATyp)),s_pn:PNm),
is the set of all concrete, composite *set* part values,
Item 195. mkConCoPaVAL((uiv,mev,attrvals),pvals)
in mkConCoPaVAL(s_qval:(UIVAL×MEVAL×(ANm $\overrightarrow{m}$ AVAL)),s_pvals:PVAL-**set**).
a uiv is a value in UIVAL of type ui,
b mev is a value in MEVAL of type me,
c attrvals is a value in (ANm $\overrightarrow{m}$ AVAL) of type attrs, and
d pvals is a[ny] value in PVAL-**set** where each part value in pvals is of the type defined for pn.

212. M: mkConCoPaTyp((UI×ME×(ANm $\overrightarrow{m}$ ATyp))×PNm) → ENV $\xrightarrow{\sim}$ PVAL-**set**
212. M(mkConCoPaTyp((ui,me,attrs),pn))(ρ) ≡
212. { mkConCoPaVAL((uiv,mev,attrvals),pvals) |
212a. uiv:UIVAL•type_of(uiv)=ui,
212b. mev:MEVAL•type_of(mev)=me,
212c. attrsval:(ANm $\overrightarrow{m}$ AVAL)•type_of(attrsval)=attrs,
212d. pvals:PVAL-**set**•pvals⊆ρ(pn) }

Materials

213 The meaning of a material type, 177.,
expression mkMaTyp(mq,pn) in mkMaTyp(s_qs:MQ,s_pn:PNm)
is the set of values mkMaVAL(mqval,ps)
in mkMaVAL(s_qval:MQVAL,s_pvals:PVAL-**set**) such that
a mqval in MQVAL is of type mq, and
b ps is a set of part values all of type pn.

213. M: mkMaTyp(s_mq:(ANm $\overrightarrow{m}$ ATyp),s_pn:PNm) → ENV $\xrightarrow{\sim}$ MVAL-**set**
213. M(mq,pn)(ρ) ≡
213. { mkMVAL(mqval,ps) |
213a. mqval:MVAL•type_of(mqval)=mq,
213b. ps:PVAL-**set**•ps⊆ρ(pn) }

Components

214 The meaning of a component type, 178., expression mkKoType(ui,atrs) in mkKoTyp(s_qs:(s_uid:UI×s_atrs:ATRS)) is the set of values, 177., mkKQVAL(uiv,attrsval) in, 197, mkKoVAL(s_qval:(uiv,attrsval)).
  a uiv is in UIVAL of type ui, and
  b attrsval is in ATTRSVAL of type atrs.

214. M: mkKoTyp(UI×ATRS) → ENV → KVAL-**set**
214. M(mkKoType(ui,atrs))(ρ) ≡
214. { mkKoVAL(uiv,attrsval) |
214a. uiv:UIVAL•type_of(uiv)=ui,
214b. attrsval:ATRSVAL•type_of(attrsval)=atrs }

3.7.4 The ι Description Function

We can now define the meaning of the syntactic clause:

- ιNm:Nm

215 ιNm:Nm "chooses" an arbitrary value from amongst the values of sort Nm:

value
215. ι nm:Nm ≡ iota(nm)
215. iota: Nm → TypDef → VAL
215. iota(nm)(td) ≡ **let** val:(PVAL|MVAL|KVAL)•val ∈ (typval(td))(nm) **in** val **end**

Discussion

From the above two functions, **typval** and **valtyp**, and the type definition "table" td:TypDef and "argument value" val:PVAL|MVAL|KVAL, we can form some expressions. One can understand these expressions as, for example reflecting the following analysis situations:

- **typval**(td): From the type definitions we form a map, by means of function typval, from sort names to the set of all values of respective sorts: Nm_to_ENDVALS.
  That is, whenever we, in the following, as part of some formula, write **typval**(td), then we mean to express that the domain engineer forms those associations, in her mind, from sort names to usually very large, non-trivial sets of endurant values.
- **valtyp**(**typval**(td)): The domain analyser cum describer "inverts", again in his mind, the **typval**(td) into a simple map, ENDVAL_to_Nm, from single endurant values to their sort names.
- (**valtyp**(**typval**(td)))(val): The domain engineer now "applies", in her mind, the simple map (above) to an endurant value and obtains its sort name nm:Nm.
- td((**valtyp**(**typval**(td)))(val)): The domain analyser cum describer then applies the type definition "table" td:TypDef to the sort name nm:Nm and obtains, in his mind, the corresponding type definition, PaTyp|MaTyp|KoTyp.

We leave it to the reader to otherwise get familiarised with these expressions.

3.8 A Formal Description of a Meaning of Prompts

3.8.1 On Function Overloading

In Sect. 3.4 the analysis and description prompt invocations were expressed as

- is_XXX(e), has_YYY(e) and observe_ZZZ(e)

where XXX, YYY, and ZZZ were appropriate entity sorts and e were appropriate endurants (parts, components and materials). The function invocations, is_XXX(e), etcetera, takes place in the context of a type definition, td:TypDef, that is, instead of is_XXX(e), etc. we get

- is_XXX(e)(td), has_YYY(e)(td) and observe_ZZZ(e)(td).

We say that the functions is_XXX, etc., are "lifted".

3.8.2 The Analysis Prompts

The analysis is expressed in terms of the analysis prompts:

The analysis takes place in the context of a type definition "image", td:TypDef, in the minds of the domain engineers.

is_entity

The is_entity predicate is meta-linguistic, that is, we cannot model it on the basis of the type systems given in Sect. 3.6. So we shall just have to accept that.

is_endurant

See analysis prompt definition 25 on Page 96 and Formula Item 153 on Page 106.

value
  is_endurant: Nm×VAL → TypDef $\xrightarrow{\sim}$ **Bool**
  is_endurant(_,val)(td) ≡ val ∈ **dom** valtyp(typval(td)); **pre**: VAL is any value type

is_discrete

See analysis prompt definition 27 on Page 96 and Formula Item 160 on Page 106.

value
  is_discrete: NmVAL → TypDef $\xrightarrow{\sim}$ **Bool**
  is_discrete(_,val)(td) ≡ (is_PaTyp|is_CoTyp)(td((valtyp(typval(td)))(val)))

is_part

See analysis prompt definition 29 on Page 97 and Formula Item 153 on Page 106.

value

is_part: NmVAL → TypDef $\stackrel{\sim}{\to}$ **Bool**
is_part(_,val)(td) ≡ is_PaTyp(td((valtyp(typval(td)))(val)))

is_material [≡ is_continuous]

See analysis prompt definition 31 on Page 97 and Formula Item 154 on Page 106.
We remind the reader that is_continuous≡is_material.

value

is_material: NmVAL → TypDef $\stackrel{\sim}{\to}$ **Bool**
is_material(_,val)(td) ≡ is_MaTyp(td((valtyp(typval(td)))(val)))

is_component

See analysis prompt definition 30 on Page 97 and Formula Item 155 on Page 106.

value

is_component: NmVAL → TypDef $\stackrel{\sim}{\to}$ **Bool**
is_component(_,val)(td) ≡ is_CoTyp(td((valtyp(typval(td)))(val)))

is_atomic

See analysis prompt definition 32 on Page 97 and Formula Item 158 on Page 106.

value

is_atomic: NmVAL → TypDef $\stackrel{\sim}{\to}$ **Bool**
is_atomic(_val)(td) ≡ is_AtPaTyp(td((valtyp(typval(td)))()))

is_composite

See analysis prompt definition 33 on Page 97 and Formula Item 159 on Page 106.

value

is_composite: NmVAL → TypDef $\stackrel{\sim}{\to}$ **Bool**
is_composite(_,val)(td) ≡ (is_AbsCoPaTyp|is_ConCoPaTyp)(td((valtyp(typval(td)))(val)))

has_concrete_type

See analysis prompt definition 34 on Page 98 and Formula Item 163 on Page 107.

value

has_concrete_type: NmVAL → TypDef $\stackrel{\sim}{\to}$ **Bool**
has_concrete_type(_,val)(td) ≡ is_ConCoPaTyp(td((valtyp(typval(td)))(val)))

`has_mereology`

See analysis prompt definition 35 on Page 99 and Formula Item 149 on Page 106.

value
 has_mereology: NmVAL → TypDef $\stackrel{\sim}{\to}$ **Bool**
 has_mereology(__,val)(td) ≡ s_me(td((valtyp(typval(td)))(val)))≠″nil″

`has_materials`

See analysis prompt definition 37 on Page 101 and Formula Item 161a on Page 107.

value
 has_material: NmVAL → TypDef $\stackrel{\sim}{\to}$ **Bool**
 has_material(__,val)(td) ≡ is_MNm(s_omkn(td((valtyp(typval(td)))(val))))
  pre: is_AtPaTyp(td((valtyp(typval(td)))(val)))

`has_components`

See analysis prompt definition 36 on Page 101 and Formula Item 161b on Page 107.

value
 has_components: NmVAL → TypDef $\stackrel{\sim}{\to}$ **Bool**
 has_components(__,val)(td) ≡ is_KNm(s_omkn(td((valtyp(typval(td)))(val))))
  pre: is_AtPaTyp(td((valtyp(typval(td)))(val)))

`has_parts`

See description prompt definition 38 on Page 102.

value
 has_parts: NmVAL → TypDef $\stackrel{\sim}{\to}$ **Bool**
 has_parts(__,val)(td) ≡ is_PNm(s_opn(td((valtyp(typval(td)))(val))))
  pre: is_MaTyp(td((valtyp(typval(td)))(val)))

3.8.3 The Description Prompts

These are the domain description prompts to be defined:

`observe_ attributes`, 30
`observe_ component_ sorts`, 25
`observe_ endurant_ sorts`, 23
`observe_ material_ sorts`, 25
`observe_ mereology`, 28
`observe_ part_ type`, 23
`observe_ unique_ identifier`, 26

A Description State

In addition to the analysis state components αps and νps there is now an additional, the description text state component.

216 Thus a global variable τ will hold the (so far) generated (in this case only) formal domain description text.

variable
216. τ := [] **Text-set**

We shall explain the use of [...]s and the operations of \ and ⊕ on the above variables in Sect. 3.4.3 on Page 108.

`observe_part_sorts`

See description prompt definition 8 on Page 97 and Formula Item 165 on Page 107.

```
value
    observe_part_sorts: NmVAL → TypDef → Unit
    observe_part_sorts(nm,val)(td) ≡
        let mkAbsCoPaTyp(_,{P1,P2,...,Pn}) = td((valtyp(typval(td)))(val)) in
                        τ := τ ⊕ [" type P1,P2,...,Pn;
                                    value
                                      obs_part_P1nm→P1
                                      obs_part_P2:nm→P2
                                      ...,
                                      obs_part_Pn:nm→Pn;
                                    proof obligation
                                      𝒟; " ]
                      ‖ νps := νps ⊕ ([P1,P2,...,Pn] \ αps)
                      ‖ αps := αps ⊕ [P1,P2,...,Pn]
        end
        pre: is_AbsCoPaTyp(td((valtyp(typval(td)))(val)))
```

$\mathscr{D}$ is a predicate expressing the disjointedness of part sorts $P_1,P_2,...,P_n$

`observe_concrete_type`

See description prompt definition 9 on Page 98 and Formula Item 164 on Page 107.

```
value
    observe_concrete_type: NmVAL → TypDef → Unit
    observe_concrete_type(nm,val)(td) ≡
        let mkConCoPaTyp(_,P) = td((valtyp(typval(td)))(val)) in
                τ := τ ⊕ ["type T = P-set ; value obs_part_T: nm→T; " ]
              ‖ νps := νps ⊕ ([P] \ αps)
              ‖ αps := αps ⊕ [P]
        end
        pre: is_ConCoPaTyp(td((valtyp(typval(td)))(val)))
```

`observe_unique_identifier`

See description prompt definition 10 on Page 99 and Formula Item 157 on Page 106.

```
value
    observe_unique_identifier: P → TypDef → Unit
    observe_unique_identifier(nm,val)(td) ≡
        τ := τ ⊕ [" type PI ; value uid_PI: nm → PI ; axiom 𝒰; " ]
```

$\mathscr{U}$ is a predicate expression over unique identifiers.

`observe_mereology`

See description prompt definition 11 on Page 99 and Formula Item 150 on Page 106.

value

```
observe_mereology: NmVAL → TypDef → Unit
observe_mereology(nm,val)(td) ≡
    τ := τ ⊕ ["type MT = ℳ(PI1,PI2,...,PIn) ;
               value obs_mereo_P: nm → MT ;
               axiom ℳℰ; " ]
    pre: has_mereology(nm,val)(td) [54]
```

ℳ(PI1,PI2,...,PIn) is a type expression over unique part identifiers. ℳℰ is a predicate expression over unique part identifiers.

observe_part_attributes

See description prompt definition 12 on Page 100 and Formula Item 151 on Page 106.

value

```
observe_part_attributes: NmVAL → TypDef → Unit
observe_part_attributes(nm,val)(td) ≡
  let {A1,A2,...,Aa} = dom s_attrs(s_qs(val)) in
  τ := τ ⊕ ["  type  A1, A2, ..., Aa
               value attr_A1: nm→Ai
                     attr_A2: nm→A1
                     ...
                     attr_Aa: nm→Ai
               proof obligation [Disjointness of Attribute Types]
                     𝒜 ; " ]
  end
```

𝒜 is a predicate over attribute types A_1, A_2, ..., A_a.

observe_part_material_sort

See description prompt definition 14 on Page 101 and Formula Item 161a on Page 107.

value

```
observe_part_material_sort: NmVAL → TypDef → Unit
observe_part_material_sort(nm,val)(td) ≡
    let M = s_pns(td((valtyp(typval(td)))(val))) in
    τ := τ ⊕ [" type M ; value obs_mat_sort_M:nm→M " ]
    ‖ νps := νps ⊕ ([M]\αps)
    ‖ αps := αps ⊕ [M]
    end
    pre: is_AtPaVAL(val) ∧ is_MNm(s_pns(td((valtyp(typval(td)))(val))))
```

observe_component_sort

See description prompt definition 13 on Page 101 and Formula Item 161b on Page 107.

value

```
observe_component_sort: NmVAL → TypDef → Unit
observe_component_sort(nm,val)(td) ≡
    let K = s_omkn(td((valtyp(typval(td)))(val))) in
```

[54] See analysis prompt definition 35 on Page 99

τ := τ ⊕ [" **type** K ; **value obs-comps**: nm → K-**set**; "]
‖ ν**ps** := ν**ps** ⊕ ([K]\\α**ps**)
‖ α**ps** := α**ps** ⊕ [K]
end
pre: is_AtPaTyp(td((valtyp(typval(td)))(val))) ∧ has_components(nm,val)

`observe_material_part_sort`

See description prompt definition 15 on Page 102 and Formula Item 155 on Page 106.

value
observe_material_part_sort: NmVAL → TypDef → **Unit**
observe_material_part_sort(nm,val)(td) ≡
let P = s_pns(td((valtyp(typval(td)))(val))) **in**
τ := τ ⊕ [" **type** P ; **value obs_part_**P: nm → P "]
‖ ν**ps** := ν**ps** ⊕ ([P] \ α**ps**)
‖ α**ps** := α**ps** ⊕ [P]
end
pre is_MaTyp(td((valtyp(typval(td)))(val))) ∧ is_PNm(s_pns(td((valtyp(typval(td)))(val))))

3.8.4 Discussion of The Prompt Model

The prompt model of this section is formulated so as to reflect a "wavering", of the domain engineer, between syntactic and semantic reflections. The syntactic reflections are represented by the syntactic arguments of the sort names, nm, and the type definitions, td. The semantic reflections are represented by the semantic argument of values, val. When we, in the various prompt definitions, use the expression td((valtyp(typval(td)))(val)) we mean to model that the domain analyser cum describer reflects semantically: "viewing", as it were, the endurant. We could, as well, have written td(nm) — reflecting a syntactic reference to the (emerging) type model in the mind of the domain engineer.

3.9 Conclusion

It is time to summarise, conclude and look forward.

3.9.1 What Has Been Achieved ?

Chapter 1 [1, 2] proposed a set of domain analysis & description prompts – and Sect. 3.2. summarised that language. Sections 3.4. and 3.8. proposed an operational semantics for the process of selecting and applying prompts, respectively a more abstract meaning of of these prompts, the latter based on some notions of an "image" of perceived abstract types of syntactic and of semantic structures of the perceived domain. These notions were discussed in Sects. 3.5. and 3.6. To the best of our knowledge this is the first time a reasonably precise notion of 'method' with a similarly reasonably precise notion of a calculi of tools has been backed up formal definitions.

3.9.2 Are the Models Valid ?

Are the formal descriptions of the process of selecting and applying the analysis & description prompts, Sect. 3.4., and the meaning of these prompts, Sect. 3.8, modeling this process and these meanings realistically ? To that we can only answer the following: The process model is definitely modeling plausible processes. We discuss interpretations of the analysis & description order that this process model imposes in Sect. 3.4.3. There might be other orders, but the ones suggested in Sect. 3.4 can be said to be "orderly" and reflects empirical observations. The model of the meaning of prompts, Sect. 3.8, is more of an hypothesis. This model refers to "images" that the domain engineer is claimed to have in her mind. It must necessarily be a valid model, perhaps one of several valid models. We have speculated, over many years, over the existence of other models. But this is the most reasonable to us.

3.9.3 Future Work

We have hinted at possible 'laws of description prompts' in Sect. 3.4.3. Whether the process and prompt models (Sects. 3.4 and 3.8) are sufficient to express, let alone prove such laws is an open question. If the models are sufficient, then they certainly are valid.

4

To Every Manifest Domain Mereology a CSP Expression

4.1 Introduction

We give an abstract model[1] of parts and part-hood relations, of Stansław Leśniewski's *mereology* [38]. Mereology applies to software application domains such as the financial service industry, railway systems, road transport systems, health care, oil pipelines, secure [IT] systems, etcetera. We relate this model to axiom systems for mereology, showing satisfiability, and show that for every mereology there corresponds a class of `Communicating Sequential Processes` [23], that is: a λ–expression.

4.1.1 Mereology

The term 'mereology' is accredited to the Polish mathematician, philosopher and logician Stansław Leśniewski (1886–1939). In this contribution we shall be concerned with only certain aspects of mereology, namely those that appear most immediately relevant to domain science (a relatively new part of current computer science). Our knowledge of 'mereology' has been through studying, amongst others, [38].

"Mereology (from the Greek $\mu\varepsilon\rho o\varsigma$ 'part') is the theory of parthood relations: of the relations of part to whole and the relations of part to part within a whole"[2]. In this contribution we restrict 'parts' to be those that, firstly, are spatially distinguishable, then, secondly, while "being based" on such spatially distinguishable parts, are conceptually related. We use the term 'part' in a more general sense than in [2]. The relation: "being based", shall be made clear in this chapter. Accordingly two parts, p_x and p_y, (of a same "whole") are are either "adjacent", or are "embedded within", one within the other, as loosely indicated in Fig. 4.1. 'Adjacent' parts are direct parts of a same third part, p_z, i.e., p_x and p_y are "embedded within"

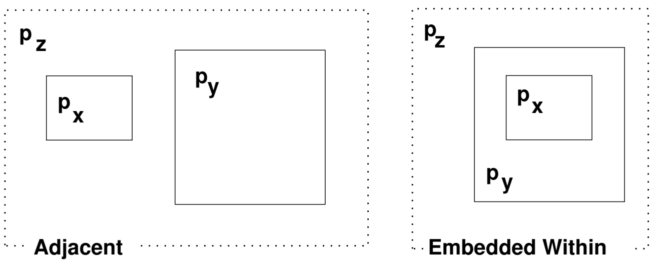


Fig. 4.1. Immediately 'Adjacent' and 'Embedded Within' Parts

p_z; or one (p_x) or the other (p_y) or both (p_x and p_y) are parts of a same third part, p'_z "embedded within"

[1] This paper is a complete rewrite of [170].
[2] Achille Varzi: Mereology, http://plato.stanford.edu/entries/mereology/ 2009 and [38].

p_z; etcetera; as loosely indicated in Fig. 4.2, or one is "embedded within" the other — etc. as loosely indicated in Fig. 4.2. Parts, whether 'adjacent' or 'embedded within', can share properties. For adjacent parts

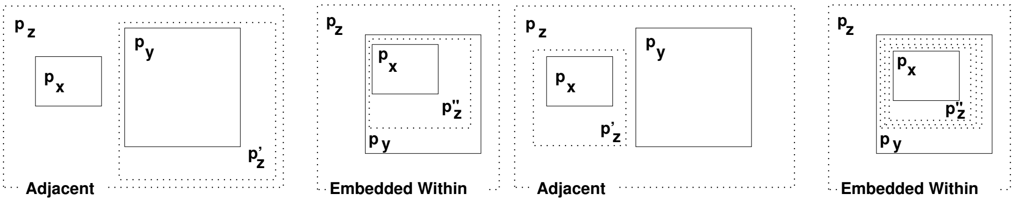


Fig. 4.2. Transitively 'Adjacent' and 'Embedded Within' Parts

this sharing seems, in the literature, to be diagrammatically expressed by letting the part rectangles "intersect". Usually properties are not spatial hence 'intersection' seems confusing. We refer to Fig. 4.3. Instead

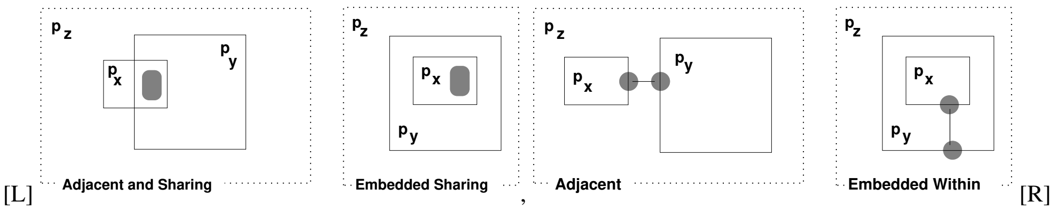


Fig. 4.3. Two models, [L,R], of parts sharing properties

of depicting parts sharing properties as in Fig. 4.3[L]eft, where shaded, dashed rounded-edge rectangles stands for 'sharing', we shall (eventually) show parts sharing properties as in Fig. 4.3[R]ight where •—• connections connect those parts.

4.1.2 From Domains via Requirements to Software

One reason for our interest in mereology is that we find that concept relevant to the modeling of domains. A derived reason is that we find the modeling of domains relevant to the development of software. Conventionally a first phase of software development is that of requirements engineering. To us domain engineering is (also) a prerequisite for requirements engineering [9]. Thus to properly **design** $\mathbb{S}$oftware we need to **understand** its or their $\mathbb{R}$equirements; and to properly **prescribe** $\mathbb{R}$equirements one must **understand** its $\mathbb{D}$omain. To **argue** correctness of $\mathbb{S}$oftware with respect to $\mathbb{R}$equirements one must usually **make assumptions** about the $\mathbb{D}$omain: $\mathbb{D}, \mathbb{S} \models \mathbb{R}$. Thus **description** of $\mathbb{D}$omains become an indispensable part of $\mathbb{S}$oftware development.

4.1.3 Domains: Science and Engineering

Domain Science is the study and knowledge of domains. **Domain Engineering** is the practice of **"walking the bridge"** from domain science to domain descriptions: to **create domain descriptions** on the background of scientific knowledge of domains, the specific domain "at hand", or domains in general; and to **study domain descriptions** with a view to broaden and deepen scientific results about domain descriptions. This contribution is based on the engineering and study of many descriptions, of air traffic, banking, commerce (the consumer/retailer/wholesaler/producer supply chain), container lines, health care, logistics, pipelines, railway systems, secure [IT] systems, stock exchanges, etcetera.

4.1.4 Contributions of This Paper

A general contribution of this paper is that of providing elements of a domain science. Three specific contributions are those of (i) giving a model that satisfies published formal, axiomatic characterisations of mereology; (ii) showing that to every (such modeled) mereology there corresponds a `CSP` [23] program; and (iii) suggesting complementing **syntactic** and **semantic** theories of mereology.

4.1.5 Structure of This Paper

We briefly overview the structure of this contribution. First, in Sect. 4.2, **we loosely characterise how we look at mereologies: "what they are to us !"**. Then, in Sect. 4.3, **we give an abstract, model-oriented specification of a class of mereologies** in the form of composite parts and composite and atomic subparts and their possible connections. In preparation for Sect. 4.4 summarizes some of the part relations introduced by Leśniewski. I The abstract model as well as the axiom system of Sect. 4.5 focuses on the **syntax of mereologies.** Following that, in Sect. 4.5, **we indicate how the model** of Sect. 4.3 **satisfies the axiom system of that Sect. 4.5.** In preparation for Sect. 4.7 we **present characterisations of attributes of parts, whether atomic or composite.** Finally Sect. 4.7 presents **a semantic model of mereologies,** one of a wide variety of such possible models. This one emphasizes the possibility of considering parts and subparts as processes and hence a mereology as a system of processes. Section 4.8 concludes with some remarks on what we have achieved.

4.2 Our Concept of Mereology

4.2.1 Informal Characterisation

Mereology, to us, is the study and knowledge about how physical and conceptual parts relate and what it means for a part to be related to another part: *being disjoint, being adjacent, being neighbours, being contained properly within, being properly overlapped with,* etcetera.

By physical parts we mean such spatial individuals which can be pointed to.

Examples: *a road net (consisting of street segments and street intersections); a street segment (between two intersections); a street intersection; a road (of sequentially neigbouring street segments of the same name); a vehicle; and a platoon (of sequentially neigbouring vehicles).*

By a conceptual part we mean an abstraction with no physical extent, which is either present or not.

Examples: *a bus timetable (not as a piece or booklet of paper, or as an electronic device, but) as an image in the minds of potential bus passengers; and routes of a pipeline, that is, neighbouring sequences of pipes, valves, pumps, forks and joins, for example referred to in discourse: "the gas flows through "such-and-such" a route".* The tricky thing here is that a route may be thought of as being both a concept or being a physical part — in which case one ought give them different names: a planned route and an actual road, for example.

The mereological notion of subpart, that is: *contained within* can be illustrated by **examples:** *the intersections and street segments are subparts of the road net; vehicles are subparts of a platoon; and pipes, valves, pumps, forks and joins are subparts of pipelines.*

The mereological notion of adjacency can be illustrated by **examples.** We consider *the various controls of an air traffic system, cf. Fig. 4.4 on the following page, as well as its aircraft, as adjacent within the air traffic system; the pipes, valves, forks, joins and pumps of a pipeline, cf. Fig. 4.9 on Page 131, as adjacent within the pipeline system; two or more banks of a banking system, cf. Fig. 4.6 on Page 130, as being adjacent.*

The mereo-topological notion of neighbouring can be illustrated by **examples:** *Some adjacent pipes of a pipeline are neighbouring (connected) to other pipes or valves or pumps or forks or joins, etcetera; two immediately adjacent vehicles of a platoon are neighbouring.*

The mereological notion of proper overlap can be illustrated by **examples** some of which are of a general kind: *two routes of a pipelines may overlap; and two conceptual bus timetables may overlap with some, but not all bus line entries being the same;* and some really reflect adjacency: *two adjacent pipe* *overlap in their connection, a wall between two rooms overlap each of these rooms — that is, the rooms overlap each other "in the wall".*

4.2.2 Six Examples

We shall, in Sect. 4.3, present a model that is claimed to abstract essential mereological properties of air traffic, buildings and their installations, machine assemblies, financial service industry, the oil industry and oil pipelines, and railway nets.

Air Traffic

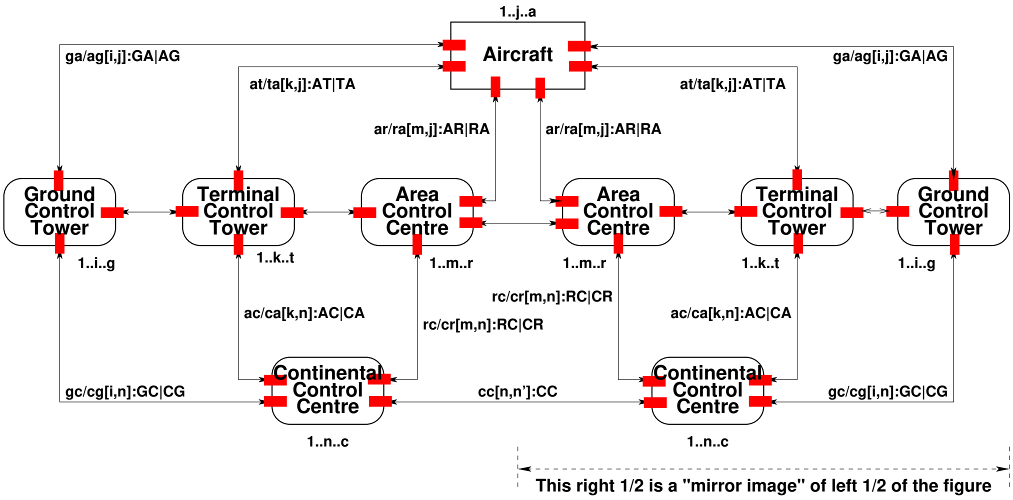


Fig. 4.4. A schematic air traffic system

Figure 4.4 shows nine adjacent (9) boxes and eighteen adjacent (18) lines. Boxes and lines are parts. The line parts "neighbours" the box parts they "connect". Individually boxes and lines represent adjacent parts of the composite air traffic "whole". The rounded corner boxes denote buildings. The sharp corner box denote aircraft. Lines denote radio telecommunication. The "overlap" between neigbouring line and box parts are indicated by "connectors". Connectors are shown as small filled, narrow, either horisontal or vertical "filled" rectangle[3] at both ends of the double-headed-arrows lines, overlapping both the line arrows and the boxes. The index ranges shown attached to, i.e., labeling each unit, shall indicate that there are a multiple of the "single" (thus representative) box or line unit shown. These index annotations are what makes the diagram of Fig. 4.4 schematic. Notice that the 'box' parts are fixed installations and that the double-headed arrows designate the ether where radio waves may propagate. We could, for example, assume that each such line is characterised by a combination of location and (possibly encrypted) radio communication frequency. That would allow us to consider all lines for not overlapping. And if they were overlapping, then that must have been a decision of the air traffic system.

Buildings

Figure 4.5 shows a building plan — as a composite part. The building consists of two buildings, A and H. The buildings A and H are neighbours, i.e., shares a common wall. Building A has rooms B, C, D and E, Building H has roomsI, J and K; Rooms L and M are within K. Rooms F and G are within C. The thick lines labeled N, O, P, Q, R, S, and T models either electric cabling, water supply, air conditioning, or some such

[3] There are 36 such rectangles in Fig. 4.4.

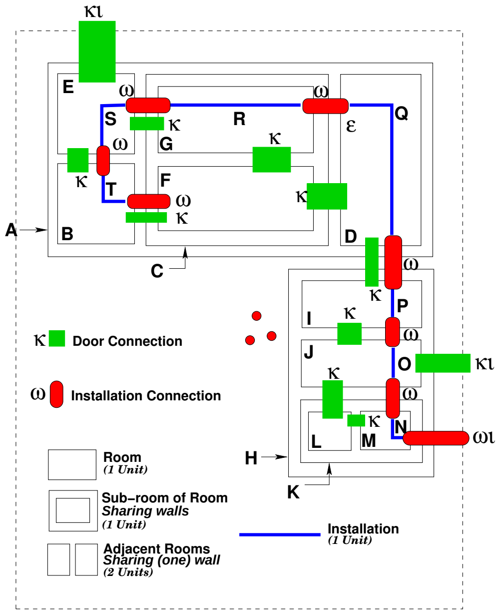


Fig. 4.5. A building plan with installation

"flow" of gases or liquids. Connection $\kappa\iota o$ provides means of a connection between an environment, shown by dashed lines, and B or J, i.e. "models", for example, a door. Connections κ provides "access" between neighbouring rooms. Note that 'neighbouring' is a transitive relation. Connection $\omega\iota o$ allows electricity (or water, or oil) to be conducted between an environment and a room. Connection ω allows electricity (or water, or oil) to be conducted through a wall. Etcetera. Thus "the whole" consists of A and H. Immediate subparts of A are B, C, D and E. Immediate subparts of C are G and F. Etcetera.

Financial Service Industry

Figure 4.6 on the next page is rather rough-sketchy! It shows seven (7) larger boxes [6 of which are shown by dashed lines], six [6] thin lined "distribution" boxes, and twelve (12) double-arrowed lines. Boxes and lines are parts. (We do not described what is meant by "distribution".) Where double-arrowed lines touch upon (dashed) boxes we have connections. Six (6) of the boxes, the dashed line boxes, are composite parts, five (5) of them consisting of a variable number of atomic parts; five (5) are here shown as having three atomic parts each with bullets "between" them to designate "variability". Clients, not shown, access the outermost (and hence the "innermost" boxes, but the latter is not shown) through connections, shown by bullets, •.

Machine Assemblies

Figure 4.7 on the following page shows a machine assembly. Square boxes designate either composite or atomic parts. Black circles or ovals show connections. The full, i.e., the level 0, composite part consists of four immediate parts and three internal and three external connections. The Pump is an assembly of six (6) immediate parts, five (5) internal connections and three (3) external connectors. Etcetera. Some connections afford "transmission" of electrical power. Other connections convey torque. Two connections convey input air, respectively output air.

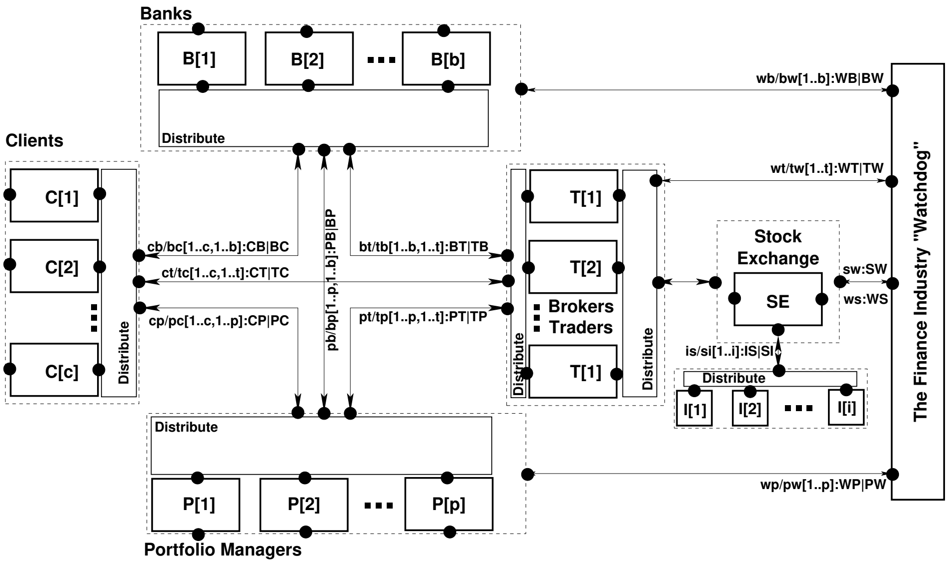


Fig. 4.6. A Financial Service Industry

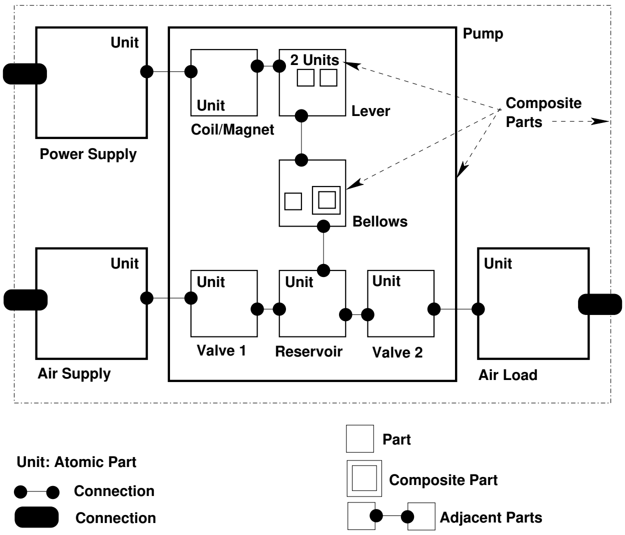


Fig. 4.7. An air pump, i.e., a physical mechanical system

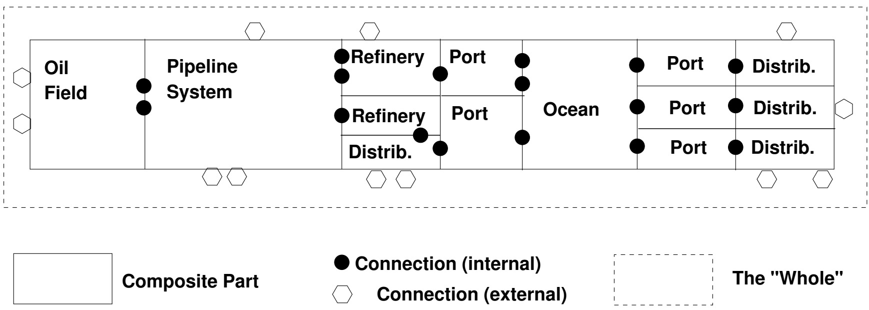


Fig. 4.8. A Schematic of an Oil Industry

Oil Industry

"The" Overall Assembly

Figure 4.8 on the preceding page shows a composite part consisting of fourteen (14) composite parts, left-to-right: one oil field, a crude oil pipeline system, two refineries and one, say, gasoline distribution network, two seaports, an ocean (with oil and ethanol tankers and their sea lanes), three (more) seaports, and three, say gasoline and ethanol distribution networks. Between all of the neighbouring composite parts there are connections, and from some of these composite parts there are connections (to an external environment). The crude oil pipeline system composite part will be concretised next.

A Concretised Composite Pipeline

Figure 4.9 shows a pipeline system. It consists of 32 atomic parts: fifteen (15) pipe units (shown as directed arrows and labeled p1–p15), four (4) input node units (shown as small circles, ∘, and labeled in$i$–inℓ), four (4) flow pump units (shown as small circles, ∘, and labeled fp$a$–fp$d$), five (5) valve units (shown as small circles, ∘, and labeled vx–vw), three (3) join units (shown as small circles, ∘, and labeled jb–jc), two (2) fork units (shown as small circles, ∘, and labeled fb–fc), one (1) combined join & fork unit (shown as small circles, ∘, and labeled jafa), and four (4) output node units (shown as small circles, ∘, and labeled on$p$–on$s$). In this example the routes through the pipeline system start with node units and end with node

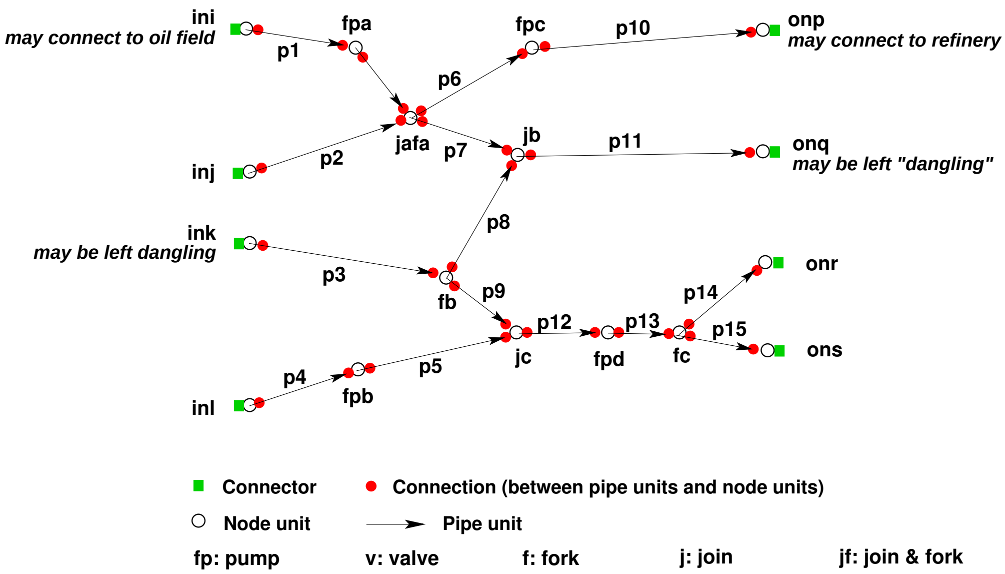


Fig. 4.9. A Pipeline System

units, alternates between node units and pipe units, and are connected as shown by fully filled-out dark coloured disc connections. Input and output nodes have input, respectively output connections, one each, and shown as lighter coloured connections. In [30] we present a description of a class of abstracted pipeline systems.

Railway Nets

The left of Fig. 4.10 on the next page [L] diagrams four rail units, each with two, three or four connectors shown as narrow, somewhat "longish" rectangles. Multiple instances of these rail units can be assembled (i.e., composed) by their connectors as shown on Fig. 4.10 on the following page [L] into proper rail nets. The right of Fig. 4.10 on the next page [R] diagrams an example of a proper rail net. It is assembled from

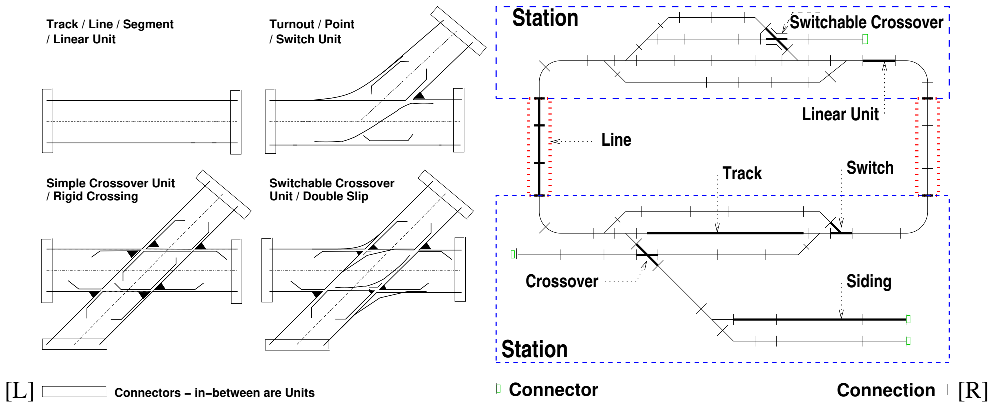


Fig. 4.10. To the left: Four rail units.To the right: A "model" railway net:
An Assembly of four Assemblies: two stations and two lines.
Lines here consist of linear rail units.
Stations of all the kinds of units shown to the left.
There are 66 connections and four "dangling" connectors

the kind of units shown in Fig. 4.10 [L]. In Fig. 4.10 [R] consider just the four dashed boxes: The dashed boxes are assembly units. Two designate stations, two designate lines (tracks) between stations. We refer to the caption four line text of Fig. 4.10 for more "statistics". We could have chosen to show, instead, for each of the four "dangling' connectors, a composition of a connection, a special "end block" rail unit and a connector.

Discussion

We have brought these examples only to indicate the issues of a "whole" and atomic and composite parts, adjacency, within, neighbour and overlap relations, and the ideas of attributes and connections. We shall make the notion of 'connection' more precise in the next section.

4.3 An Abstract, Syntactic Model of Mereologies

4.3.1 Parts and Subparts

217 We distinguish between **atomic** and **composite parts.**
218 Atomic parts do not contain separately distinguishable parts.
219 Composite parts contain at least one separately distinguishable part.

type
217. P == AP | CP[4]
218. AP :: mkAP(...)[5]
219. CP :: mkCP(...,s_sps:P-**set**)[6] **axiom** ∀ mkCP(_,ps):CP • ps≠{}

It is the domain analyser who decides what constitutes "the whole", that is, how parts relate to one another, what constitutes parts, and whether a part is atomic or composite. We refer to the proper parts of a composite part as subparts. Figure 4.11 on the next page illustrates composite and atomic parts. The *slanted sans serif* uppercase identifiers of Fig. 4.11 *A1, A2, A3, A4, A5, A6* and *C1, C2, C3* are meta-linguistic, that is. they stand for the parts they "decorate"; they are not identifiers of "our system".

[4] In the RAISE [64] Specification Languge, RSL [22], writing type definitions X == Y|Z means that Y and Z are to be disjoint types. In Items 218.–219. the identifiers mkAP and mkCP are distinct, hence their types are disjoint.
[5] Y :: mkY(...): *y* values (...) are marked with the "make constructor" mkY, cf. [171, 172].
[6] In Y :: mkY(s_w:W,...) s_w is a "selector function" which when applied to an *y*, i.e., s_w(*y*) identifies the *W* element, cf. [171, 172].

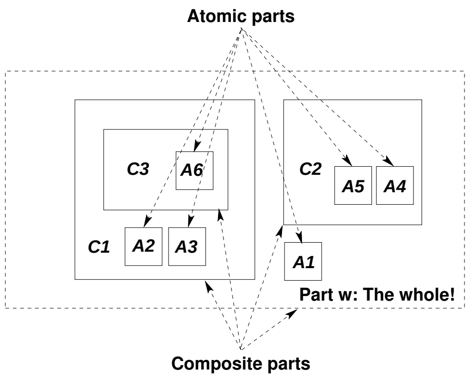


Fig. 4.11. Atomic and Composite Parts

4.3.2 No "Infinitely" Embedded Parts

The above syntax, Items 217–219, does not prevent composite parts, p, to contain composite parts, p', "ad-infinitum" ! But we do not wish such "recursively" contained parts !

220 To express the property that parts are finite we introduce a notion of ***part derivation***.
221 The part derivation of an atomic part is the empty set.
222 The part derivation of a composite part, p, mkC(...,ps) where ... is left undefined, is the set ps of subparts of p.

value
220. pt_der: P → P-**set**
221. pt_der(mkAP(...)) ≡ {}
222. pt_der(mkCP(...,ps)) ≡ ps

223 We can also express the part derivation, pt_der(ps) of a set, ps, of parts.
224 If the set is empty then pt_der({}) is the empty set, {}.
225 Let mkA(pq) be an element of ps, then pt_der({mkA(pq)}∪ps′) is ps′.
226 Let mkC(pq,ps′) be an element of ps, then pt_der(ps′∪ps) is ps′.

223. pt_der: P-**set** → P-**set**
224. pt_der({}) ≡ {}
225. pt_der({mkA(..)}∪ps) ≡ ps
226. pt_der({mkC(..,ps')}∪ps) ≡ ps'∪ps

227 Therefore, to express that a part is finite we postulate
228 a natural number, n, such that a notion of iterated part set derivations lead to an empty set.
229 An iterated part set derivation takes a set of parts and part set derive that set repeatedly, n times.
230 If the result is an empty set, then part p was finite.

value
227. no_infinite_parts: P → **Bool**
228. no_infinite_parts(p) ≡
228. ∃ n:**Nat** • it_pt_der({p})(n)={}
229. it_pt_der: P-**set** → **Nat** → P-**set**
230. it_pt_der(ps)(n) ≡
230. **let** ps′ = pt_der(ps) **in**
230. **if** n=1 **then** ps′ **else** it_pt_der(ps′)(n−1) **end end**

4.3.3 Unique Identifications

Each physical part can be uniquely distinguished for example by an abstraction of its properties at a time of origin. In consequence we also endow conceptual parts with unique identifications.

231 In order to refer to specific parts we endow all parts, whether atomic or composite, with **u**nique **id**entifications.
232 We postulate functions which observe these **u**nique **id**entifications, whether as parts in general or as atomic or composite parts in particular.
233 such that any to parts which are distinct have **u**nique **id**entifications.

```
type
231.  UI
value
232.  uid_UI: P → UI
axiom
233.  ∀ p,p′:P • p≠p′ ⇒ uid_UI(p)≠uid_UI(p′)
```

A model for uid_UI can be given. Presupposing subsequent material (on attributes and mereology) — "lumped" into part qualities, pq:PQ, we augment definitions of atomic and composite parts:

```
type
218.  AP :: mkA(s_pq:(s_uid:UI,...))
219.  CP :: mkC(s_pq:(s_uid:UI,...),s_sps:P-set)
value
232.  uid_UI(mkA((ui,...))) ≡ ui
232.  uid_UI(mkC((ui,...)),...) ≡ ui
```

Figure 4.12 illustrates the unique identifications of composite and atomic parts.

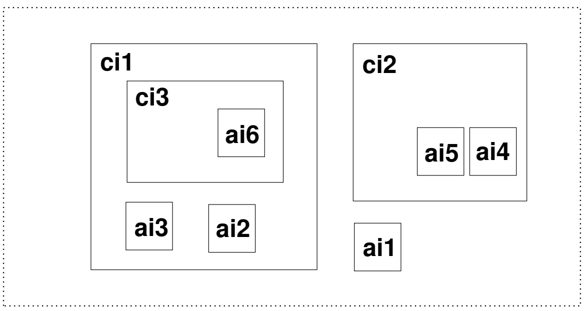


Fig. 4.12. ai_j: atomic part identifiers, ci_k: composite part identifiers

No two parts have the same unique identifier.

234 We define an auxiliary function, no_prts_uis, which applies to a[ny] part, *p*, and yields a pair: the number of subparts of the part argument, and the set of unique identifiers of parts within *p*.
235 no_prts_uis is defined in terms of yet an auxiliary function, sum_no_pts_uis.

```
value
234.  no_prts_uis: P → (Nat × UI-set) → (Nat × UI-set)
234.  no_pts_uis(mkA(ui,...))(n,uis) ≡ (n+1,uis∪{ui})
234.  no_pts_uis(mkC((ui,...),ps))(n,uis) ≡
234.      let (n′,uis′) = sum_no_pts_uis(ps) in
234.      (n+n′,uis∪uis’) end
```

```
234.  pre: no_infinite_parts(p)
235.  sum_no_pts_uis: P-set → (Nat × UI-set) → (Nat × UI-set)
235.  sum_no_pts_uis(ps)(n,uis) ≡
235.    case ps of
235.      {}→(n,uis),
235.      {mkA(ui,...)}∪ps'→sum_no_pts_uis(ps')(n+1,uis∪{ui}),
235.      {mkC((ui,...),ps')}∪ps'' →
235.        let (n'',uis'')=sum_no_pts_uis(ps')(1,{ui}) in
235.        sum_no_pts_uis(ps'')(n+n'',uis∪uis'') end
235.    end
235.    pre: ∀ p:P•p ∈ ps ⇒ no_infinite_parts(p)
```

236 That no two parts have the same unique identifier can now be expressed by demanding that the number of parts equals the number of unique identifiers.

```
axiom
236.  ∀ p:P • let (n,uis)=no_prts_uis(0,{}) in n=card uis end
```

4.3.4 Attributes

Attribute Names and Values

237 Parts have sets of named attribute values, attrs:ATTRS.
238 One can observe attributes from parts.
239 Two distinct parts may share attributes:
  a For some (one or more) attribute name that is among the attribute names of both parts,
  b it is always the case that the corresponding attribute values are identical.

```
type
237.  ANm, AVAL, ATTRS = ANm →m AVAL
value
238.  attr_ATTRS: P → ATTRS
239.  share: P×P → Bool
239.  share(p,p') ≡
239.      p≠p' ∧ ∼trans_adj(p,p') ∧
239a.     ∃ anm:ANm • anm ∈ dom attr_ATTRS(p) ∩ dom attr_ATTRS(p') ⇒
239b.       □ (attr_ATTRS(p))(anm) = (attr_ATTRS(p'))(anm)
```

The function trans_adj is defined in Sect. 4.4.4 on Page 138.

Attribute Categories

One can suggest a hierarchy of part attribute categories: static or dynamic values — and within the dynamic value category: inert values or reactive values or active values — and within the dynamic active value category: autonomous values or biddable values or programmable values. By a **static attribute**, a:A, `is_static_attribute(a)`, we shall understand an attribute whose values are constants, i.e., cannot change. By a **dynamic attribute**, a:A, `is_dynamic_attribute(a)`, we shall understand an attribute whose values are variable, i.e., can change. By an **inert attribute**, a:A, `is_inert_attribute(a)`, we shall understand a dynamic attribute whose values only change as the result of external stimuli where these stimuli prescribe properties of these new values. By a **reactive attribute**, a:A, `is_reactive_attribute(a)`, we shall understand a dynamic attribute whose values, if they vary, change value in response to the change of other attribute values. By an **active attribute**, a:A, `is_active_attribute(a)`, we shall understand a dynamic attribute whose values change (also) of its own volition. By an **autonomous attribute**, a:A, is_autonomous_attribute(a), we shall understand a dynamic active attribute whose values change value only “on their own volition”. The values of an autonomous attributes are a “law onto themselves and their surroundings”. By a **biddable attribute**, a:A, is_biddable_attribute(a), (of a part) we shall understand a dynamic active attribute whose values are prescribed but may fail to be observed as such. By a **programmable attribute**, a:A, is_programmable_attribute(a:A), we shall understand a dynamic active attribute whose values can be prescribed. By an **external attribute** we mean inert, reactive, active or autonomous attribute. By a **controllable attribute** we mean a biddable or programmable attribute. We define some auxiliary functions:

240 $\mathscr{S}_{\mathscr{A}}$ applies to attrs:ATTRS and yields a grouping $(sa_1,sa_2,...,sa_{n_s})$[7], of **static** attribute values.
241 $\mathscr{C}_{\mathscr{A}}$ applies to attrs:ATTRS and yields a grouping $(ca_1,ca_2,...,ca_{n_c})$[8] of **controllable** attribute values.
242 $\mathscr{E}_{\mathscr{A}}$ applies to attrs:ATTRS and yields a set, $\{eA_1,eA_2,...,eA_{n_e}\}$[9] of **external** attribute names.

type
  SA,CA = AVAL*
  EA = ANm−st
value
240. $\mathscr{S}_{\mathscr{A}}$: ATTRS → SA
241. $\mathscr{C}_{\mathscr{A}}$: ATTRS → CA
242. $\mathscr{E}_{\mathscr{A}}$: ATTRS → EA

The attribute names of static, controllable and external attributes do not overlap and together make up the attribute names of attrs.

4.3.5 Mereology

In order to illustrate other than the within and adjacency part relations we introduce the notion of mereology. Figure 4.13 illustrates a mereology between parts. A specific mereology-relation is, visually, a •—• line that connects two distinct parts.

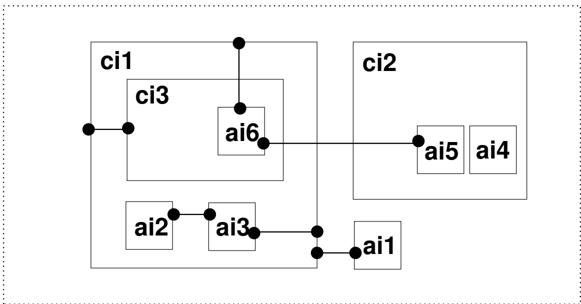


Fig. 4.13. Mereology: Relations between Parts

243 The mereology of a part is a set of unique identifiers of other parts.

type
243. ME = UI-**set**

We may refer to the connectors by the two element sets of the unique identifiers of the parts they connect. For **example** with respect to Fig. 4.13:

[7] – where $\{sa_1,sa_2,...,sa_{n_s}\}\subseteq$**rng** attrs
[8] – where $\{ca_1,ca_2,...,ca_{n_s}\}\subseteq$**rng** attrs
[9] – where $\{eA_1,eA_2,...,eA_{n_e}\}\subseteq$**dom** attrs

- $\{ci_1, ci_3\}$,
- $\{ai_2, ai_3\}$,
- $\{ai_6, ci_1\}$,
- $\{ai_3, ci_1\}$,
- $\{ai_6, ai_5\}$ and
- $\{ai_1, ci_1\}$.

4.3.6 The Model

244 The "whole" is a part.
245 A part value has a part sort name and is either the value of an atomic part or of an abstract composite part.
246 An atomic part value has a part quality value.
247 An abstract composite part value has a part quality value and a set of at least of one or more part values.
248 A part quality value consists of a unique identifier, a mereology, and a set of one or more attribute named attribute values.

```
244   W = P
245    P = AP | CP
246   AP :: mkA(s_pq:PQ)
247   CP :: mkC(s_pq:PQ,s_ps:P-set)
248   PQ = UI×ME×(ANm →m AVAL)
```

We now assume that parts are not "recursively infinite", and that all parts have unique identifiers

4.4 Some Part Relations

4.4.1 'Immediately Within'

249 One part, p, is said to be *immediately within*, imm_within(p,p′), another part, if p′ is a composite part and p is observable in p′.

```
value
249.  imm_within: P × P → Bool
249.  imm_within(p,p′) ≡
249.     case p′ of
249.        (_,mkA(_,ps)) → p ∈ ps,
249.        (_,mkC(_,ps)) → p ∈ ps,
249.        _ → false
249.     end
```

4.4.2 'Transitive Within'

We can generalise the 'immediate within' property.

250 A part, p, is transitively within a part p′, trans_within(p,p′),
  a either if p, is immediately within p′
  b or
  c if there exists a (proper) composite part p″ of p′ such that trans_within(p″,p).

```
value
250.  trans_wihin: P × P → Bool
250.  trans_within(p,p′) ≡
250a.     imm_within(p,p′)
250b.     ∨
250c.     case p′ of
250c.        (_,mkC(_,ps)) → p ∈ ps ∧
250c.            ∃ p″:P• p″ ∈ ps ∧ trans_within(p″,p),
250c.        _ → false
250.     end
```

4.4.3 'Adjacency'

251 Two parts, p,p′, are said to be *immediately adj*acent, imm_adj(p,p′)(c), to one another, in a composite part c, such that p and p′ are distinct and observable in c.

value
251. imm_adj: P × P → P → **Bool**
251. imm_adj(p,p′)(mkA(_,ps)) ≡ p≠p′ ∧ {p,p′}⊆ps
251. imm_adj(p,p′)(mkC(_,ps)) ≡ p≠p′ ∧ {p,p′}⊆ps
251. imm_adj(p,p′)(mkA(_)) ≡ **false**

4.4.4 Transitive 'Adjacency'

We can generalise the immediate 'adjacent' property.

252 Two parts, p′,p″, of a composite part, p, are trans_adj(p′, p″) in p
  a either if imm_adj(p′,p″)(p),
  b or if there are two p‴ and p⁗ such that
    i p‴ and p⁗ are immediately adjacent parts of p and
    ii p is equal to p‴ or p‴ is properly within p and p′ is equal to p⁗ or p⁗ is properly within p′

We leave the formalisation to the reader.

4.5 An Axiom System

Classical axiom systems for mereology focus on just one sort of "things", namely 𝒫arts. Leśniewski had in mind, when setting up his mereology to have it supplant set theory. So parts could be composite and consisting of other, the sub-parts — some of which would be atomic; just as sets could consist of elements which were sets — some of which would be empty.

4.5.1 Parts and Attributes

In our axiom system for mereology we shall avail ourselves of two sorts: 𝒫arts, and 𝒜ttributes.[10]

- **type** 𝒫, 𝒜

𝒜ttributes are associated with 𝒫arts. We do not say very much about attributes: We think of attributes of parts to form possibly empty sets. So we postulate a primitive predicate, ∈, relating 𝒫arts and 𝒜ttributes.

- ∈: 𝒜 × 𝒫 → **Bool**.

4.5.2 The Axioms

The axiom system to be developed in this section is a variant of that in [38]. We introduce the following relations between parts:

part_of:	$\mathbb{P}: \mathscr{P} \times \mathscr{P} \to$ **Bool**	Page 139
proper_part_of:	$\mathbb{PP}: \mathscr{P} \times \mathscr{P} \to$ **Bool**	Page 139
overlap:	$\mathbb{O}: \mathscr{P} \times \mathscr{P} \to$ **Bool**	Page 139
underlap:	$\mathbb{U}: \mathscr{P} \times \mathscr{P} \to$ **Bool**	Page 139
over_crossing:	$\mathbb{OX}: \mathscr{P} \times \mathscr{P} \to$ **Bool**	Page 139
under_crossing:	$\mathbb{UX}: \mathscr{P} \times \mathscr{P} \to$ **Bool**	Page 139
proper_overlap:	$\mathbb{PO}: \mathscr{P} \times \mathscr{P} \to$ **Bool**	Page 139
proper_underlap:	$\mathbb{PU}: \mathscr{P} \times \mathscr{P} \to$ **Bool**	Page 139

[10] Identifiers P and A stand for model-oriented types (parts and atomic parts), whereas identifiers 𝒫 and 𝒜 stand for property-oriented types (parts and attributes).

Let $\mathbb{P}$ denote **part-hood**; p_x is part of p_y, is then expressed as $\mathbb{P}(p_x, p_y)$.[11] (4.1) Part p_x is part of itself (reflexivity). (4.2) If a part p_x is part p_y and, vice versa, part p_y is part of p_x, then $p_x = p_y$ (anti-symmetry). (4.3) If a part p_x is part of p_y and part p_y is part of p_z, then p_x is part of p_z (transitivity).

$$\forall p_x : \mathscr{P} \bullet \mathbb{P}(p_x, p_x) \tag{4.1}$$

$$\forall p_x, p_y : \mathscr{P} \bullet (\mathbb{P}(p_x, p_y) \wedge \mathbb{P}(p_y, p_x)) \Rightarrow p_x = p_y \tag{4.2}$$

$$\forall p_x, p_y, p_z : \mathscr{P} \bullet (\mathbb{P}(p_x, p_y) \wedge \mathbb{P}(p_y, p_z)) \Rightarrow \mathbb{P}(p_z, p_z) \tag{4.3}$$

Let $\mathbb{PP}$ denote **proper part-hood**. p_x is a proper part of p_y is then expressed as $\mathbb{PP}(p_x, p_y)$. $\mathbb{PP}$ can be defined in terms of $\mathbb{P}$. $\mathbb{PP}(p_x, p_y)$ holds if p_x is part of p_y, but p_y is not part of p_x.

$$\mathbb{PP}(p_x, p_y) \triangleq \mathbb{P}(p_x, p_y) \wedge \neg\mathbb{P}(p_y, p_x) \tag{4.4}$$

Overlap, $\mathbb{O}$, expresses a relation between parts. Two parts are said to overlap if they have “something” in common. In classical mereology that ‘something’ is parts. To us parts are spatial entities and these cannot “overlap”. Instead they can ‘share’ attributes.

$$\mathbb{O}(p_x, p_y) \triangleq \exists a : \mathscr{A} \bullet a \in p_x \wedge a \in p_y \tag{4.5}$$

Underlap, $\mathbb{U}$, expresses a relation between parts. Two parts are said to underlap if there exists a part p_z of which p_x is a part and of which p_y is a part.

$$\mathbb{U}(p_x, p_y) \triangleq \exists p_z : \mathscr{P} \bullet \mathbb{P}(p_x, p_z) \wedge \mathbb{P}(p_y, p_z) \tag{4.6}$$

Think of the underlap p_z as an “umbrella” which both p_x and p_y are “under”.
Over-cross, $\mathbb{OX}$, p_x and p_y are said to over-cross if p_x and p_y overlap and p_x is not part of p_y.

$$\mathbb{OX}(p_x, p_y) \triangleq \mathbb{O}(p_x, p_y) \wedge \neg\mathbb{P}(p_x, p_y) \tag{4.7}$$

Under-cross, $\mathbb{UX}$, p_x and p_y are said to under cross if p_x and p_y underlap and p_y is not part of p_x.

$$\mathbb{UX}(p_x, p_y) \triangleq \mathbb{U}(p_x, p_z) \wedge \neg\mathbb{P}(p_y, p_x) \tag{4.8}$$

Proper Overlap, $\mathbb{PO}$, expresses a relation between parts. p_x and p_y are said to properly overlap if p_x and p_y over-cross and if p_y and p_x over-cross.

$$\mathbb{PO}(p_x, p_y) \triangleq \mathbb{OX}(p_x, p_y) \wedge \mathbb{OX}(p_y, p_x) \tag{4.9}$$

Proper Underlap, $\mathbb{PU}$, p_x and p_y are said to properly underlap if p_x and p_y under-cross and p_y and p_x under-cross.

$$\mathbb{PU}(p_x, p_y) \triangleq \mathbb{UX}(p_x, p_y) \wedge \mathbb{UX}(p_y, p_x) \tag{4.10}$$

4.6 Satisfaction

We shall sketch a proof that the *model* of Sect. 4.3, *satisfies*, i.e., is a model of, the *axioms* of Sect. 4.5.

4.6.1 Some Definitions

To that end we first define the notions of *interpretation*, *satisfiability*, *validity* and *model*. **Interpretation:** By an interpretation of a predicate we mean an assignment of a truth value to the predicate where the assignment may entail an assignment of values, in general, to the terms of the predicate. **Satisfiability:** By the satisfiability of a predicate we mean that the predicate is true for some interpretation. **Valid:** By the validity of a predicate we mean that the predicate is true for all interpretations. **Model:** By a model of a predicate we mean an interpretation for which the predicate holds.

[11] Our notation now is not RSL but a conventional first-order predicate logic notation.

4.6.2 A Proof Sketch

We assign

253 P as the meaning of $\mathscr{P}$
254 ATR as the meaning of $\mathscr{A}$,
255 imm_within as the meaning of $\mathbb{P}$,
256 trans_within as the meaning of $\mathbb{PP}$,
257 $\in$: ATTR$\times$ATTRS-**set**$\rightarrow$**Bool** as the meaning of $\in$: $\mathscr{A} \times \mathscr{P} \rightarrow$**Bool** and
258 sharing as the meaning of $\mathbb{O}$.

With the above assignments it is now easy to prove that the other axiom-operators $\mathbb{U}$, $\mathbb{PO}$, $\mathbb{PU}$, $\mathbb{OX}$ and $\mathbb{UX}$ can be modeled by means of imm_within, within, ATTR$\times$ATTRS-**set**$\rightarrow$**Bool** and sharing.

4.7 A Semantic CSP Model of Mereology

The model of Sect. 4.3 can be said to be an abstract model-oriented definition of the syntax of mereology. Similarly the axiom system of Sect. 4.5 can be said to be an abstract property-oriented definition of the syntax of mereology. We show that to every mereology there corresponds a program of communicating sequential processes CSP. We assume that the reader has practical knowledge of Hoare's CSP [23].

4.7.1 Parts $\simeq$ Processes

The model of mereology presented in Sect. 4.3 focused on (i) parts, (ii) unique identifiers and (iii) mereology. To parts we associate CSP processes. Part processes are indexed by the unique part identifiers. The mereology reveals the structure of CSP channels between CSP processes.

4.7.2 Channels

We define a general notion of a vector of channels. One vector element for each "pair" of distinct unique identifiers. Vector indices are set of two distinct unique identifiers.

259 Let w be the "whole" (i.e., a part).
260 Let uis be the set of all unique identifiers of the "whole".
261 Let M be the type of messages sent over channels.
262 Channels provide means for processes to synchronise and communicate.

value
259. w:P
260. uis = **let** (_,uis′)=no_prts_uis(w) **in** uis′ **end**
type
261. M
channel
262. $\{$ch[$\{$ui,ui′$\}$]:M|ui,ui′:UI•ui$\neq$ui′ $\wedge$ $\{$ui,ui′$\}\subseteq$uis$\}$

263 We also define channels for access to external attribute values.

Without loss of generality we do so for all possible parts and all possible attributes.

channel
263. $\{$xch[ui,an]:AVAL|ui:UI• ui $\in$ uis,an:ANm$\}$

4.7.3 Compilation

We now show how to compile "real-life, actual" parts into **RSL-Text**. That is, turning "semantics" into syntax !

value
  comp_P: P → **RSL-Text**
  comp_P(mkA(ui,me,attrs)) ≡ "$\mathscr{M}_a$(ui,me,attrs)"
  comp_P(mkC((ui,me,attrs),{p_1,p_2,...,p_n})) ≡
    "$\mathscr{M}_c$(ui,me,attrs) ∥
    " comp_process(p_1)"∥"comp_process(p_2)"∥"..."∥"comp_process(p_n)

The so-called core process expressions $\mathscr{M}_a$ and $\mathscr{M}_c$ relate to *a*tomic and *c*omposite parts. They are defined, schematically, below as just $\mathscr{M}$. The compilation expressions have two elements: (i) those embraced by double quotes: "...", and (ii) those that invoke further compilations, The first texts, (i), shall be understood as **RSL-Text**s. The compilation invocations, (ii), as expending into **RSL-Text**s. We emphasize the distinction between 'usages' and 'definitions'. The expressions between double quotes: "..." designate usages. We now show how some of these usages require "definitions". These 'definitions' are not the result of 'parts-to-processes' compilations. They are shown here to indicate, to the domain engineers, what must be further described, beyond the 'mere' compilations.

value
  $\mathscr{M}$: ui:UI×me:ME×attrs:ATTRS → ca:$\mathscr{C}_{\mathscr{A}}$(attrs) → **RSL-Text**
  $\mathscr{M}$(ui,me,attrs)(ca) ≡
    let (me′,ca′) = $\mathscr{F}$(ui,me,attrs)(ca) **in** $\mathscr{M}$(ui,me′,attrs)(ca′) **end**
  $\mathscr{F}$: ui:UI×me:ME×attrs:ATTRS→ca:CA→
    in in_chs(ui,attrs) **in**,**out** in_out_chs(ui,me)→ME×CA ′

Recall (Page 136) that $\mathscr{C}_{\mathscr{A}}$(attrs) is a grouping, ($ca_1$,$ca_2$,...,$ca_{n_c}$), of controlled attribute values.

264 The in_chs function applies to a set of uniquely named attributes and yields some **RSL**-Text, in the form of **in**put channel declarations, one for each external attribute.

264. in_chs: ui:UI × attrs:ATTRS → **RSL−Text**
264. in_chs(ui,attrs) ≡ "**in** { xch[ui,xa_i] | xa_i:ANm • xa_i∈$\mathscr{E}_{\mathscr{A}}$(attrs) }"

265 The in_out_chs function applies to a pair, a unique identifier and a mereology, and yields some **RSL**-Text, in the form of **input**/**out**put channel declarations, one for each unique identifier in the mereology.

265. in_out_chs: ui:UI × me:ME → **RSL−Text**
265. in_out_chs(ui,me) ≡ "**in**,**out** { xch[ui,ui′]|ui:UI • ui'∈me }"

$\mathscr{F}$ is an action: it returns a possibly updated mereology and possibly updated controlled attribute values. We present a rough sketch of $\mathscr{F}$. The $\mathscr{F}$ action non-deterministically internal choice chooses between

- either [1,2,3,4]
  - [1] accepting input from
  - [4] a suitable ("offering") part process,
  - [2] optionally offering a reply;
  - [3] leading to an updated state;
- or [3,4]
  - [5] finding a suitable "order" (val)
  - [8] to a suitable ("inquiring") behaviour,
  - [6] offering that value,
  - [7] leading to an updated state;
- or [9] doing own work leading to an new state.

value
  $\mathscr{F}$(ui,me,attrs)(ca) ≡
[1]   ⌈⌉ {**let** val=ch[{ui,ui′}]? **in**
[2]   (ch[{ui,ui′}]!in_reply(val,(ui,me,attrs))(ca)) ;

```
          in_update(val,(ui,me,attrs))(ca) end
        | ui′:UI • ui′ ∈ me}
  ⌈⌉ ⌈⌉ {let val=await_reply(ui′,me,attrs)(ca) in
          ch[{ui,ui′}]!val ;
          out_update(val,(ui,me,attrs))(ca) end
        | ui′:UI • ui′ ∈ me}
  ⌈⌉     (me,own_work(ui,attrs)(ca))

in_reply:  VAL×(ui:UI×me:ME×attrs:ATTRS)→ca:CA→
                         in in_chs(attrs) in,out in_out_chs(ui,me)→VAL
in_update: VAL × (ui:UI×me:ME×attrs:ATTRS)→ca:CA→
                         in,out in_out_chs(ui,me)→ME×CA
await_reply: (ui:UI,me:ME)→ca:CA→in,out in_out_chs(ui,me:ME)→VAL
out_update: (VAL×(ui:UI×me:ME<>attrs:ATTRS))→ca:CA→
                         in,out in_out_chs(ui,me)→ME×CA
own_work: (ui:UI×attrs:ATTRS)→CA→in,out in_out_chs(ui,me) CA
```

The above definitions of channels and core functions $\mathscr{M}$ and $\mathscr{F}$ are not examples of what will be compiled but of what the domain engineer must, after careful analysis, "create".

4.7.4 Discussion

General

A little more meaning has been added to the notions of parts and their mereology. The within and adjacent to relations between parts (composite and atomic) reflect a phenomenological world of geometry, and the mereological relation between parts reflect both physical and conceptual world understandings: physical world in that, for example, radio waves cross geometric "boundaries", and conceptual world in that ontological classifications typically reflect lattice orderings where *overlaps* likewise cross geometric "boundaries".

Specific

The notion of parts is far more general than that of [2]. We have been able to treat Stansław Leśniewski's notion of mereology sôlely based on parts, that is, their semantic values, without introducing the notion of the syntax of parts. Our compilation functions are (thus) far more general than defined in [2].

4.8 Concluding Remarks

4.8.1 Relation to Other Work

The present contribution has been conceived in the following context.

My first awareness of the concept of 'mereology' was from listening to many presentations by **Douglas T. Ross** (1929–2007) at IFIP working group WG 2.3 meetings over the years 1980–1999. In [173] **Henry S. Leonard** and **Henry Nelson Goodman:** *A Calculus of Individuals and Its Uses* present the American Pragmatist version of Leśniewski's mereology. It is based on a single primitive: *discreet.* The idea of the calculus of individuals is, as in Leśniewski's mereology, to avoid having to deal with the empty sets while relying on explicit reference to classes (or parts).

[38] **R. Casati** and **A. Varzi:** *Parts and Places: the structures of spatial representation* has been the major source for this paper's understanding of mereology. Although our motivation was not the spatial or topological mereology, [174], and although the present paper does not utilize any of these concepts' axiomatisation in [38, 174] it is best to say that it has benefited much from these publications.

Domain descriptions, besides mereological notions, also depend, in their successful form. on FCA: Formal Concept Analysis. Here a main inspiration has been drawn, since the mid 1990s, from **B. Ganter** and **R. Wille's** *Formal Concept Analysis — Mathematical Foundations* [175].

> *The approach takes as input a matrix specifying a set of objects and the properties thereof, called attributes, and finds both all the "natural" clusters of attributes and all the "natural" clusters of objects in the input data, where a "natural" object cluster is the set of all objects that share a common subset of attributes, and a "natural" property cluster is the set of all attributes shared by one of the natural object clusters. Natural property clusters correspond one-for-one with natural object clusters, and a concept is a pair containing both a natural property cluster and its corresponding natural object cluster. The family of these concepts obeys the mathematical axioms defining a lattice, a Galois connection).*

Thus the choice of adjacent and embedded ('within') parts and their connections is determined after serious formal concept analysis.

4.8.2 What Has Been Achieved ?

We have given a model-oriented specification of mereology. We have indicated that the model satisfies a widely known axiom system for mereology. We have suggested that (perhaps most) work on mereology amounts to syntactic studies. So we have suggested one of a large number of possible, schematic semantics of mereology. And we have shown that to every mereology there corresponds a set of communicating sequential process (CSP).

Part II

A Requirements Engineering Method

5

From Domain Descriptions to Requirements Prescriptions

5.1 Introduction

[1, 2, Domains Analysis & Description] introduce a method for analysing and describing manifest domains. In this chapter we show how to systematically, but, of course, not automatically, "derive" requirements prescriptions from domain descriptions.

5.1.1 The Contribution of This Chapter

We claim that the present chapter content contributes to our understanding and practice of ***software engineering*** as follows: (1) it shows how the new phase of engineering, domain engineering, as introduced in [2], forms a prerequisite for requirements engineering; (2) it endows the "classical" form of requirements engineering with a structured set of development stages and steps: (a) first a domain requirements stage, (b) to be followed by an interface requirements stages, and (c) to be concluded by a machine requirements stage; (3) it further structures and gives a reasonably precise contents to the stage of domain requirements: (i) first a projection step, (ii) then an instantiation step, (iii) then a determination step, (iv) then an extension step, and (v) finally a fitting step — with these five steps possibly being iterated; and (4) it also structures and gives a reasonably precise contents to the stage of interface requirements based on a notion of shared entities, Each of the steps (i–v) open for the possibility of ***simplification***s. Steps (a–c) and (i-v), we claim, are new. They reflect a serious contribution, we claim, to a logical structuring of the field of requirements engineering and its very many otherwise seemingly diverse concerns.

5.1.2 Some Comments

This chapter is, perhaps, unusual in the following respects: (i) It is a methodology chapter, hence there are no "neat" theories about development, no succinctly expressed propositions, lemmas nor theorems, and hence no proofs[1]. (ii) As a consequence the chapter is borne by many, and by extensive examples. (iii) The examples of this chapter are all focused on a generic road transport net. (iv) To reasonably fully exemplify the requirements approach, illustrating how our method copes with a seeming complexity of interrelated method aspects, the full example of this chapter embodies very many description and prescription elements: hundreds of concepts (types, axioms, functions). (v) This methodology chapter covers a "grand" area of software engineering: Many textbooks and papers are written on *Requirements Engineering*. We postulate, in contrast to all such books (and papers), that ***requirements engineering*** should be founded on ***domain engineering***. Hence we must, somehow, show that our approach relates to major elements of what the *Requirements Engineering* books put forward. (vi) As a result, this chapter is long.

[1] — where these proofs would be about the development theories. The example development of requirements do imply properties, but formulation and proof of these do not constitute specifically new contributions — so are left out.

5.1.3 Structure of Chapter

The structure of the chapter is as follows: Section 5.2 provides a fair-sized, hence realistic example. Sections 5.3–5.5 covers our approach to requirements development. Section 5.3 overviews the issue of 'requirements'; relates our approach (i.e., Sects. 5.4–5.5) to ***systems***, ***user and external equipment*** and ***functional requirements***; and Sect. 5.3 also introduces the concepts of the ***machine*** to be requirements prescribed, the ***domain***, the ***interface*** and the ***machine requirements***. Section 5.4 covers the ***domain requirements*** stages of ***projection*** (Sect. 5.4.1), ***instantiation*** (Sect. 5.4.2), ***determination*** (Sect. 5.4.3), ***extension*** (Sect. 5.4.4) and ***fitting*** (Sect. 5.4.5). Section 5.5 covers key features of ***interface requirements***: ***shared phenomena*** (Sect. 5.5.1), ***shared endurant***s (Sect. 5.5.1) and ***shared action***s, ***shared event***sand ***shared behaviour***s (Sect. 5.5.1). Section 5.5.1 further introduces the notion of ***derived requirements***. Section 5.7 concludes the chapter.

5.2 An Example Domain: Transport

In order to exemplify the various stages and steps of requirements development we first bring a domain description example. The example follows the steps of an idealised domain description. First we describe the endurants, then we describe the perdurants. Endurant description initially focus on the composite and atomic parts. Then on their "internal" qualities: unique identifications, mereologies, and attributes. The descriptions alternate between enumerated, i.e., labeled narrative sentences and correspondingly "numbered" formalisations. The narrative labels cum formula numbers will be referred to, frequently in the various steps of domain requirements development.

5.2.1 Endurants

Since we have chosen a manifest domain, that is, a domain whose endurants can be pointed at, seen, touched, we shall follow the analysis & description process as outlined in [2] and formalised in [40]. That is, we first identify, analyse and describe (manifest) parts, composite and atomic, abstract (Sect. 5.2.2) or concrete (Sect. 5.2.2). Then we identify, analyse and describe their unique identifiers (Sect. 5.2.2), mereologies (Sect. 5.2.2), and attributes (Sects. 5.2.2–5.2.2).

The example fragments will be presented in a small type-font.

5.2.2 Domain, Net, Fleet and Monitor

The root domain, Δ, is that of a composite traffic system (266a.) with a road net, (266b.) with a fleet of vehicles and (266c.) of whose individual position on the road net we can speak, that is, monitor.[2]

266 We analyse the traffic system into
  a a composite road net,
  b a composite fleet (of vehicles), and
  c an atomic monitor.

type
266 Δ
266a N
266b F
266c M

value
266a **obs_part**_N: $\Delta \to$ N
266b **obs_part**_F: $\Delta \to$ F
266c **obs_part**_M: $\Delta \to$ M

Applying `ch5observe_part_sorts` [2, Sect. 3.1.6] to a net, *n*:*N*, yields the following.

267 The road net consists of two composite parts,
  a an aggregation of hubs and
  b an aggregation of links.

[2] The monitor can be thought of, i.e., conceptualised. It is not necessarily a physically manifest phenomenon.

type
267a HA
267b LA

value
267a **obs_part**_HA: N $\rightarrow$ HA
267b **obs_part**_LA: N $\rightarrow$ LA

Hubs and Links

Applying `ch5observe_part_types` [2, Sect. 3.1.7] to hub and link aggregates yields the following.

268 Hub aggregates are sets of hubs.
269 Link aggregates are sets of links.
270 Fleets are set of vehicles.

type
268 H, HS = H-**set**
269 L, LS = L-**set**
270 V, VS = V-**set**

value
268 **obs_part**_HS: HA $\rightarrow$ HS
269 **obs_part**_LS: LA $\rightarrow$ LS
270 **obs_part**_VS: F $\rightarrow$ VS

271 We introduce some auxiliary functions.
  a links extracts the links of a network.
  b hubs extracts the hubs of a network.

value
271a links: $\Delta \rightarrow$ L-**set**
271b hubs: $\Delta \rightarrow$ H-**set**
271a links(δ) $\equiv$
271a   **obs_part**_LS(**obs_part**_LA(**obs_part**_N(δ)))
271b hubs(δ) $\equiv$
271b   **obs_part**_HS(**obs_part**_HA(**obs_part**_N(δ)))

Unique Identifiers

Applying `ch5observe_unique_identifier` [2, Sect. 3.2] to the observed parts yields the following.

272 Nets, hub and link aggregates, hubs and links, fleets, vehicles and the monitor all
  a have unique identifiers
  b such that all such are distinct, and
  c with corresponding observers.

type
272a NI, HAI, LAI, HI, LI, FI, VI, MI
value
272c **uid**_NI: N $\rightarrow$ NI
272c **uid**_HAI: HA $\rightarrow$ HAI
272c **uid**_LAI: LA $\rightarrow$ LAI
272c **uid**_HI: H $\rightarrow$ HI
272c **uid**_LI: L $\rightarrow$ LI
272c **uid**_FI: F $\rightarrow$ FI
272c **uid**_VI: V $\rightarrow$ VI
272c **uid**_MI: M $\rightarrow$ MI
axiom
272b NI$\cap$HAI=$\emptyset$, NI$\cap$LAI=$\emptyset$, NI$\cap$HI=$\emptyset$, etc.

where axiom 272b. is expressed semi-formally, in mathematics. We introduce some auxiliary functions:

273 xtr_lis extracts all link identifiers of a traffic system.
274 xtr_his extracts all hub identifiers of a traffic system.
275 Given an appropriate link identifier and a net get_link 'retrieves' the designated link.
276 Given an appropriate hub identifier and a net get_hub 'retrieves' the designated hub.

value
273 xtr_lis: $\Delta \rightarrow$ LI-**set**
273 xtr_lis(δ) $\equiv$
273   **let** ls = links(δ) **in** {**uid**_LI(l)|l:L•l $\in$ ls} **end**
274 xtr_his: $\Delta \rightarrow$ HI-**set**
274 xtr_his(δ) $\equiv$
274   **let** hs = hubs(δ) **in** {**uid**_HI(h)|h:H•k $\in$ hs} **end**
275 get_link: LI $\rightarrow \Delta \xrightarrow{\sim}$ L
275 get_link(li)(δ) $\equiv$
275   **let** ls = links(δ) **in**
275   **let** l:L • l $\in$ ls $\wedge$ li=**uid**_LI(l) **in** l **end end**
275   **pre**: li $\in$ xtr_lis(δ)
276 get_hub: HI $\rightarrow \Delta \xrightarrow{\sim}$ H
276 get_hub(hi)(δ) $\equiv$
276   **let** hs = hubs(δ) **in**
276   **let** h:H • h $\in$ hs $\wedge$ hi=**uid**_HI(h) **in** h **end end**
276   **pre**: hi $\in$ xtr_his(δ)

Mereology

We cover the mereologies of all part sorts introduced so far. We decide that nets, hub aggregates, link aggregates and fleets have no mereologies of interest.Applying `ch5observe_mereology` [2, Sect. 3.3.2] to hubs, links, vehicles and the monitor yields the following.

277 Hub mereologies reflect that they are connected to zero, one or more links.
278 Link mereologies reflect that they are connected to exactly two distinct hubs.
279 Vehicle mereologies reflect that they are connected to the monitor.
280 The monitor mereology reflects that it is connected to all vehicles.
281 For all hubs of any net it must be the case that their mereology designates links of that net.
282 For all links of any net it must be the case that their mereologies designates hubs of that net.
283 For all transport domains it must be the case that
  a the mereology of vehicles of that system designates the monitor of that system, and that
  b the mereology of the monitor of that system designates vehicles of that system.

```
value
277   obs_mereo_H: H → LI-set
278   obs_mereo_L: L → HI-set
axiom
278   ∀ l:L•card obs_mereo_L(l)=2
value
279   obs_mereo_V: V → MI
280   obs_mereo_M: M → VI-set
axiom
281   ∀ δ:Δ, hs:HS•hs=hubs(δ), ls:LS•ls=links(δ) •
281      ∀ h:H•h ∈ hs•obs_mereo_H(h)⊆xtr_lis(δ) ∧
282      ∀ l:L•l ∈ ls•obs_mereo_L(l)⊆xtr_his(δ) ∧
283a     let f:F•f=obs_part_F(δ) ⇒
283a        let m:M•m=obs_part_M(δ),
283a            vs:VS•vs=obs_part_VS(f) in
283a         ∀ v:V•v ∈ vs⇒uid_V(v) ∈ obs_mereo_M(m)
283b       ∧ obs_mereo_M(m) = {uid_V(v)|v:V•v ∈ vs}
283b     end end
```

Attributes, I

We may not have shown all of the attributes mentioned below — so consider them informally introduced !

- **Hubs:** ***location*s**[3] are considered static, ***hub state*s** and ***hub state space*s** are considered programmable;
- **Links:** ***length*s** and ***location*s** are considered static, ***link state*s** and ***link state space*s** are considered programmable;
- **Vehicles:** ***manufacturer name***, ***engine type*** (whether diesel, gasoline or electric) and ***engine power*** (kW/horse power) are considered static; ***velocity*** and ***acceleration*** may be considered reactive (i.e., a function of gas pedal position, etc.), ***global position*** (informed via a `GNSS: Global Navigation Satellite System`) and ***local position*** (calculated from a global position) are considered biddable

Applying `ch5observe_attributes` [2, Sect. 3.4.3] to hubs, links, vehicles and the monitor yields the following.

First hubs.

284 Hubs
  a have geodetic locations, GeoH,
  b have ***hub states*** which are sets of pairs of identifiers of links connected to the hub[4],
  c and have ***hub state spaces*** which are sets of hub states[5].
285 For every net,
  a link identifiers of a hub state must designate links of that net.
  b Every hub state of a net must be in the hub state space of that hub.
286 We introduce an auxiliary function: xtr_lis extracts all link identifiers of a hub state.

```
type
284a  GeoH
284b  HΣ = (LI×LI)-set
284c  HΩ = HΣ-set
value
284a  attr_GeoH: H → GeoH
284b  attr_HΣ: H → HΣ
284c  attr_HΩ: H → HΩ
axiom
285     ∀ δ:Δ,
285      let hs = hubs(δ) in
285      ∀ h:H • h ∈ hs •
285a         xtr_lis(h)⊆xtr_lis(δ)
285b       ∧ attr_Σ(h) ∈ attr_Ω(h)
285      end
value
286  xtr_lis: H → LI-set
286  xtr_lis(h) ≡
286     {li | li:LI,(li′,li″):LI×LI •
286        (li′,li″)∈attr_HΣ(h)∧li∈{li′,li″}}
```

[3] By location we mean a geodetic position.
[4] A hub state "signals" which input-to-output link connections are open for traffic.
[5] A hub state space indicates which hub states a hub may attain over time.

Then links.

287 Links have lengths.
288 Links have geodetic location.
289 Links have states and state spaces:
  a States modeled here as pairs, (hi', hi''), of identifiers the hubs with which the links are connected and indicating directions (from hub h' to hub h''.) A link state can thus have 0, 1, 2, 3 or 4 such pairs.
  b State spaces are the set of all the link states that a link may enjoy.

```
type
287   LEN
288   GeoL
289a  LΣ = (HI×HI)-set
289b  LΩ = LΣ-set
value
287   attr_LEN: L → LEN
288   attr_GeoL: L → GeoL
289a  attr_LΣ: L → LΣ
289b  attr_LΩ: L → LΩ
axiom
289   ∀ n:N •
289     let ls = xtr−links(n), hs = xtr_hubs(n) in
289     ∀ l:L•l ∈ ls ⇒
289a      let lσ = attr_LΣ(l) in
289a      0≤card lσ≤4
289a      ∧ ∀ (hi′,hi″):(HI×HI)•(hi′,hi″) ∈ lσ
289a          ⇒ {hi′,hi″}=obs_mereo_L(l)
289b      ∧ attr_LΣ(l) ∈ attr_LΩ(l)
289     end end
```

Then vehicles.

290 Every vehicle of a traffic system has a position which is either 'on a link' or 'at a hub'.
  a An 'on a link' position has four elements: a unique link identifier which must designate a link of that traffic system and a pair of unique hub identifiers which must be those of the mereology of that link.
  b The 'on a link' position real is the fraction, thus properly between 0 (zero) and 1 (one) of the length from the first identified hub "down the link" to the second identifier hub.
  c An 'at a hub' position has three elements: a unique hub identifier and a pair of unique link identifiers — which must be in the hub state.

```
type
290   VPos = onL | atH
290a  onL :: LI HI HI R
290b  R = Real     axiom ∀ r:R • 0≤r≤1
290c  atH :: HI LI LI
value
290   attr_VPos: V → VPos
axiom
290a  ∀ n:N, onL(li,fhi,thi,r):VPos •
290a    ∃ l:L•l ∈obs_part_LS(obs_part_N(n))
290a      ⇒ li=uid_L(l)∧{fhi,thi}=obs_mereo_L(l),
290c  ∀ n:N, atH(hi,fli,tli):VPos •
290c    ∃ h:H•h ∈obs_part_HS(obs_part_N(n))
290c      ⇒ hi=uid_H(h)∧(fli,tli)∈attr_LΣ(h)
```

291 We introduce an auxiliary function `distribute`.
  a `distribute` takes a net and a set of vehicles and
  b generates a map from vehicles to distinct vehicle positions on the net.
  c We sketch a "formal" `distribute` function, but, for simplicity we omit the technical details that secures distinctness — and leave that to an axiom !
292 We define two auxiliary functions:
  a xtr_links extracts all links of a net and
  b xtr_hub extracts all hubs of a net.

```
type
291b  MAP = VI →m VPos
axiom
291b  ∀ map:MAP • card dom map = card rng map
value
291   distribute: VS → N → MAP
291   distribute(vs)(n) ≡
291a    let (hs,ls) = (xtr_hubs(n),xtr_links(n)) in
291a      let vps = {onL(uid_(l),fhi,thi,r) |
291a                 l:L•l ∈ls∧{fhi,thi}
291a                 ⊆obs_mereo_L(l)∧0≤r≤1}
291a               ∪ {atH(uid_H(h),fli,tli)|
291a                 h:H•h ∈hs∧{fli,tli}
291a                 ⊆obs_mereo_H(h)} in
291b      [uid_V(v)↦vp|v:V,vp:VPos•v ∈vs∧vp∈vps]
291     end end

292a  xtr_links: N → L-set
292a  xtr_links(n)≡
292a      obs_part_LS(obs_part_LA(n))

292b  xtr_hubs: N → H-set
292a  xtr_hubs(n)≡
292a      obs_part_H(obs_part_HA_Δ(n))
```

And finally monitors. We consider only one monitor attribute.

293 The monitor has a vehicle traffic attribute.
  a For every vehicle of the road transport system the vehicle traffic attribute records a possibly empty list of time marked vehicle positions.
  b These vehicle positions are alternate sequences of 'on link' and 'at hub' positions
    i such that any sub-sequence of 'on link' positions record the same link identifier, the same pair of ''to' and 'from' hub identifiers and increasing fractions,
    ii such that any sub-segment of 'at hub' positions are identical,
    iii such that vehicle transition from a link to a hub is commensurate with the link and hub mereologies, and
    iv such that vehicle transition from a hub to a link is commensurate with the hub and link mereologies.

```
type
293    Traffic = VI →m (T × VPos)*
value
293    attr_Traffic: M → Traffic
axiom
293b    ∀ δ:Δ •
293b        let m = obs_part_M(δ) in
293b        let tf = attr_Traffic(m) in
293b        dom tf ⊆ xtr_vis(δ) ∧
293b        ∀ vi:VI • vi ∈ dom tf •
293b            let tr = tf(vi) in
293b            ∀ i,i+1:Nat • {i,i+1}⊆dom tr •
293b                let (t,vp)=tr(i),(t′,vp′)=tr(i+1) in
293b                t<t′
293(b)i              ∧ case (vp,vp′) of
293(b)i                  (onL(li,fhi,thi,r),onL(li′,fhi′,thi′,r′))
293(b)i                     → li=li′∧fhi=fhi′∧thi=thi′∧r≤r′ ∧ li ∈ xtr_lis(δ) ∧
293(b)i                        {fhi,thi} = obs_mereo_L(get_link(li)(δ)),
293(b)ii                 (atH(hi,fli,tli),atH(hi′,fli′,tli′))
293(b)ii                    → hi=hi′∧fli=fli′∧tli=tli′ ∧ hi ∈ xtr_his(δ) ∧
293(b)ii                       (fli,tli) ∈ obs_mereo_H(get_hub(hi)(δ)),
293(b)iii                (onL(li,fhi,thi,1),atH(hi,fli,tli))
293(b)iii                   → li=fli∧thi=hi ∧ {li,tli} ⊆ xtr_lis(δ) ∧
293(b)iii                      {fhi,thi}=obs_mereo_L(get_link(li)(δ)) ∧ hi ∈ xtr_his(δ) ∧
293(b)iii                      (fli,tli) ∈ obs_mereo_H(get_hub(hi)(δ)),
293(b)iv                 (atH(hi,fli,tli),onL(li′,fhi′,thi′,0))
293(b)iv                    → etcetera,
293b                  _ → false
293b        end end end end end
```

5.2.3 Perdurants

Our presentation of example perdurants is not as systematic as that of example endurants. Give the simple basis of endurants covered above there is now a huge variety of perdurants, so we just select one example from each of the three classes of perdurants (as outline in [2]): a simple hub insertion ***action*** (Sect. 5.2.3), a simple link disappearance ***event*** (Sect. 5.2.3) and a not quite so simple ***behaviour***, that of road traffic (Sect. 5.2.3).

Hub Insertion Action

294 Initially inserted hubs, h, are characterised
  a by their unique identifier which not one of any hub in the net, n, into which the hub is being inserted,
  b by a mereology, {}, of zero link identifiers, and
  c by — whatever — attributes, $attrs$, are needed.

295 The result of such a hub insertion is a net, n',
  a whose links are those of n, and
  b whose hubs are those of n augmented with h.

```
value
294    insert_hub: H → N → N
```

295 insert_hub(h)(n) **as** n′
294a **pre**: **uid**_H(h)∉ xtr_his(n)
294b ∧ **obs_mereo**_H={}
294c ∧ ...
295a **post**: **obs_part**_Ls(n)=**obs_part**_Ls(n′)
295b ∧ **obs_part**_Hs(n)∪{h}=**obs_part**_Hs(n′)

Link Disappearance Event

We formalise aspects of the link disappearance event:

296 The result net, n':N', is not well-formed.
297 For a link to disappear there must be at least one link in the net;
298 and such a link may disappear such that
299 it together with the resulting net makes up for the "original" net.

value
296 link_diss_event: N × N′ × **Bool**
296 link_diss_event(n,n′) **as** tf
297 **pre**: **obs_part**_Ls(**obs_part**_LS(n))≠{}
298 **post**: ∃ l:L•l∈ **obs_part**_Ls(**obs_part**_LS(n))⇒
299 l∉ **obs_part**_Ls(**obs_part**_LS(n′))
299 ∧ n′union{l}=**obs_part**_Ls(**obs_part**_LS(n))

Road Traffic

The analysis & description of the road traffic behaviour is composed (i) from the description of the global values of nets, links and hubs, vehicles, monitor, a clock, and an initial distribution, ***map***, of vehicles, "across" the net; (ii) from the description of channels between vehicles and the monitor; (iii) from the description of behaviour signatures, that is, those of the overall road traffic system, the vehicles, and the monitor; and (iv) from the description of the individual behaviours, that is, the overall road traffic system, ***rts***, the individual vehicles, ***veh***, and the monitor, ***mon***.

Global Values:

There is given some globally observable parts.

300 besides the domain, δ:Δ,
301 a net, n:N,
302 a set of vehicles, vs:V-**set**,
303 a monitor, m:M, and
304 a clock, clock, behaviour.
305 From the net and vehicles we generate an initial distribution of positions of vehicles.

The n:N, vs:V-**set** and m:M are observable from any road traffic system domain δ.

value
300 δ:Δ
301 n:N = **obs_part**_N(δ),
301 ls:L-**set**=links(δ),hs:H-**set**=hubs(δ),
301 lis:LI-**set**=xtr_lis(δ),his:HI-**set**=xtr_his(δ)
302 va:VS=**obs_part**_VS(**obs_part**_F(δ)),
302 vs:Vs-**set**=**obs_part**_Vs(va),
302 vis:VI-**set** = {**uid**_VI(v)|v:V•v ∈ vs},
303 m:**obs_part**_M(δ),
303 mi=**uid**_MI(m),
303 ma:**attributes**(m)
304 clock: $\mathbb{T}$ → **out** {clk_ch[vi|vi:VI•vi ∈ vis]} **Unit**
305 vm:MAP•vpos_map = distribute(vs)(n);

Channels:

306 We additionally declare a set of vehicle-to-monitor-channels indexed
a by the unique identifiers of vehicles
b and the (single) monitor identifier.[6]

and communicating vehicle positions.

channel
306 {v_m_ch[vi,mi]|vi:VI•vi ∈ vis}:VPos

Behaviour Signatures:

[6] Technically speaking: we could omit the monitor identifier.

307 The road traffic system behaviour, rts, takes no arguments (hence the first **Unit**)[7]; and "behaves", that is, continues forever (hence the last **Unit**).

308 The vehicle behaviour
- a is indexed by the unique identifier, uid_V(v):VI,
- b the vehicle mereology, in this case the single monitor identifier mi:MI,
- c the vehicle attributes, obs__attribs(v)
- d and — factoring out one of the vehicle attributes — the current vehicle position.
- e The vehicle behaviour offers communication to the monitor behaviour (on channel vm_ch[vi]); and behaves "forever".

309 The monitor behaviour takes
- a the monitor identifier,
- b the monitor mereology,
- c the monitor attributes,
- d and — factoring out one of the vehicle attributes — the discrete road traffic, drtf:dRTF, being repeatedly "updated" as the result of **in**put communications from (all) vehicles;
- e the behaviour otherwise behaves forever.

value
307 rts: **Unit** → **Unit**
308 veh$_{vi:VI}$: mi:MI→vp:VPos→
308   **out** vm_ch[vi,mi] **Unit**
309 mon$_{mi:MI}$: vis:VI-**set**→RTF→
309   **in** {v_m_ch[vi,mi]|vi:VI•vi ∈ vis},clk_ch **Unit**

The Road Traffic System Behaviour:

310 Thus we shall consider our **road traffic system**, rts, as
- a the concurrent behaviour of a number of vehicles and, to "observe", or, as we shall call it, to monitor their movements,
- b the monitor behaviour.

value
310 rts() =
310a   ∥ {veh$_{\mathbf{uid_}VI(v)}$(mi)(vm(**uid**_VI(v)))
310a | v:V•v ∈ vs}
310b   ∥ mon$_{mi}$(vis)([vi↦⟨⟩|vi:VI•vi ∈ vis])

where, wrt, the monitor, we dispense with the mereology and the attribute state arguments and instead just have a monitor traffic argument which records the discrete road traffic, MAP, initially set to "empty" traces (⟨⟩, of so far "no road traffic"!).

In order for the monitor behaviour to assess the vehicle positions these vehicles communicate their positions to the monitor via a vehicle to monitor channel. In order for the monitor to time-stamp these positions it must be able to "read" a clock.

311 We describe here an abstraction of the vehicle behaviour **at** a Hub (hi).
- a Either the vehicle remains at that hub informing the monitor of its position,
- b or, internally non-deterministically,
  - i moves onto a link, tli, whose "next" hub, identified by thi, is obtained from the mereology of the link identified by tli;
  - ii informs the monitor, on channel vm[vi,mi], that it is now at the very beginning (0) of the link identified by tli, whereupon the vehicle resumes the vehicle behaviour positioned at the very beginning of that link,
- c or, again internally non-deterministically, the vehicle "disappears — off the radar" !

311 veh$_{vi}$(mi)(vp:atH(hi,fli,tli)) ≡
311a   v_m_ch[vi,mi]!vp ; veh$_{vi}$(mi)(vp)
311b   ⊓
311(b)i   **let** {hi′,thi}=**obs_mereo**_L(get_link(tli)(n))
311(b)i   **assert:** hi′=hi
311(b)ii   **in** v_m_ch[vi,mi]!onL(tli,hi,thi,0) ;
311(b)ii   veh$_{vi}$(mi)(onL(tli,hi,thi,0)) **end**
311c   ⊓ **stop**

312 We describe here an abstraction of the vehicle behaviour **on** a Link (ii). Either
- a the vehicle remains at that link position informing the monitor of its position,
- b or, internally non-deterministically, if the vehicle's position on the link has not yet reached the hub,
  - i then the vehicle moves an arbitrary increment ℓ_ε (less than or equal to the distance to the hub) along the link informing the monitor of this, or
  - ii else,
    - 1 while obtaining a "next link" from the mereology of the hub (where that next link could very well be the same as the link the vehicle is about to leave),
    - 2 the vehicle informs the monitor that it is now at the hub identified by thi, whereupon the vehicle resumes the vehicle behaviour positioned at that hub.
- c or, internally non-deterministically, the vehicle "disappears — off the radar" !

312 veh$_{vi}$(mi)(vp:onL(li,fhi,thi,r)) ≡
312a   v_m_ch[vi,mi]!vp ; veh$_{vi}$(mi,va)(vp)
312b   ⊓ **if** r + ℓ_ε≤1
312(b)i   **then**
312(b)i   v_m_ch[vi,mi]!onL(li,fhi,thi,r+ℓ_ε) ;
312(b)i   veh$_{vi}$(mi)(onL(li,fhi,thi,r+ℓ_ε))
312(b)ii   **else**

[7] The **Unit** designator is an RSL technicality.

```
312(b)ii1    let li′:LI•li′ ∈ obs_mereo_H(get_hub(thi)(n)) in  ⌈⌉ stop   312a
312(b)ii2     v_m_ch[vi,mi]!atH(li,thi,li′);
312(b)ii2     veh_vi(mi)(atH(li,thi,li′)) end end
```

The Monitor Behaviour

313 The monitor behaviour evolves around
  a the monitor identifier,
  b the monitor mereology,
  c and the attributes, ma:ATTR
  d — where we have factored out as a separate arguments — a table of traces of time-stamped vehicle positions,
  e while accepting messages
    i about time
    ii and about vehicle positions
  f and otherwise progressing "in[de]finitely".

314 Either the monitor "does own work"
315 or, internally non-deterministically accepts messages from vehicles.
  a A vehicle position message, vp, may arrive from the vehicle identified by vi.
  b That message is appended to that vehicle's movement trace – prefixed by time (obtained from the time channel),
  c whereupon the monitor resumes its behaviour —
  d where the communicating vehicles range over all identified vehicles.

```
313    mon_mi(vis)(trf) ≡
314        mon_mi(vis)(trf)
315     ⌈⌉
315a     ⌈⌉{let tvp = (clk_ch?,v_m_ch[vi,mi]?) in
315b        let trf′ = trf † [vi ↦ trf(vi)^<tvp>] in
315c        mon_mi(vis)(trf′)
315d        end end | vi:VI • vi ∈ vis}
```

We are about to complete a long, i.e., a 6.3 page example (!). We can now comment on the full example: The domain, $\delta : \Delta$ is a manifest part. The road net, $n : N$ is also a manifest part. The fleet, $f : F$, of vehicles, $vs : VS$, likewise, is a manifest part. But the monitor, $m : M$, is a concept. One does not have to think of it as a manifest "observer". The vehicles are on — or off — the road (i.e., links and hubs). We know that from a few observations and generalise to all vehicles. They either move or stand still. We also, similarly, know that. Vehicles move. Yes, we know that. Based on all these repeated observations and generalisations we introduce the concept of vehicle traffic. Unless positioned high above a road net — and with good binoculars — a single person cannot really observe the traffic. There are simply too many links, hubs, vehicles, vehicle positions and times. Thus we conclude that, even in a richly manifest domain, we can also "speak of", that is, describe concepts over manifest phenomena, including time !

5.2.4 Domain Facets

The example of this section, i.e., Sect. 5.2, focuses on the ***domain facet*** [4, 2008] of (i) ***instrinsics***. It does not reflect the other ***domain facet***s: (ii) domain support technologies, (iii) domain rules, regulations & scripts, (iv) organisation & management, and (v) human behaviour. The requirements examples, i.e., the rest of this chapter, thus builds only on the ***domain instrinsics***. This means that we shall not be able to cover principles, technique and tools for the prescription of such important requirements that handle failures of support technology or humans. We shall, however point out where we think such, for example, fault tolerance requirements prescriptions "fit in" and refer to relevant publications for their handling.

5.3 Requirements

This and the next three sections, Sects. 5.4.–5.5., are the main sections of this chapter. Section 5.4. is the most detailed and systematic section. It covers the ***domain requirements*** operations of ***projection***, ***instantiation***, ***determination***, ***extension*** and, less detailed, ***fitting***. Section 5.5. surveys the ***interface requirements*** issues of ***shared phenomena***: ***shared endurant***s, ***shared action***s, ***shared event***s and ***shared behaviour***, and "completes" the exemplification of the detailed ***domain extension*** of our requirements into a ***road pricing system***. Section 5.5. also covers the notion of ***derived requirements***.

5.3.1 The Three Phases of Requirements Engineering

There are, as we see it, three kinds of design assumptions and requirements: (i) ***domain requirements***, (ii) ***interface requirements*** and (iii) ***machine requirements***. (i) **Domain requirements** are those requirements which can be expressed sôlely using terms of the domain ■ (ii) **Interface requirements** are those requirements which can be expressed only using technical terms of both the domain and the machine ■ (iii) **Machine requirements** are those requirements which, in principle, can be expressed sôlely using terms of the machine ■

Definition 52 Verification Paradigm: Some preliminary designations: let $\mathscr{D}$ designate the the domain description; let $\mathscr{R}$ designate the requirements prescription, and let $\mathscr{S}$ designate the system design. Now $\mathscr{D},\mathscr{S} \models \mathscr{R}$ shall be read: it must be verified that the $\mathscr{S}$ystem design satisfies the $\mathscr{R}$equirements prescription in the context of the $\mathscr{D}$omain description ■

The "in the context of $\mathscr{D}$..." term means that proofs of $\mathscr{S}$oftware design correctness with respect to $\mathscr{R}$equirements will often have to refer to $\mathscr{D}$omain requirements assumptions. We refer to [119, Gunter, Jackson and Zave, 2000] for an analysis of a varieties of forms in which $\models$ relate to variants of $\mathscr{D}$, $\mathscr{R}$ and $\mathscr{S}$.

5.3.2 Order of Presentation of Requirements Prescriptions

The ***domain requirements development*** stage — as we shall see — can be sub-staged into: ***projection***, ***instantiation***, ***determination***, ***extension*** and ***fitting***. The ***interface requirements development*** stage — can be sub-staged into ***shared: endurant, action, event and behaviour*** developments, where "sharedness" pertains to phenomena shared between, i.e., "present" in, both the domain (concretely, manifestly) and the machine (abstractly, conceptually). These development stages need not be pursued in the order of the three stages and their sub-stages. We emphasize that one thing is the stages and steps of development, as for example these: projection, instantiation, determination, extension, fitting, shared endurants, shared actions, shared events, shared behaviours, etcetera, another thing is the requirements prescription that results from these development stages and steps. The further software development, after and on the basis of the requirements prescription starts only when all stages and steps of the requirements prescription have been fully developed. The domain engineer is now free to rearrange the final prescription, irrespective of the order in which the various sections were developed, in such a way as to give a most pleasing, pedagogic and cohesive reading (i.e., presentation). From such a requirements prescription one can therefore not necessarily see in which order the various sections of the prescription were developed.

5.3.3 Design Requirements and Design Assumptions

A crucial distinction is between ***design requirements*** and ***design assumptions***. The **design requirements** are those requirements for which the system designer **has to** implement hardware or software in order satisfy system user expectations ■ The **design assumptions** are those requirements for which the system designer **does not** have to implement hardware or software, but whose properties the designed hardware, respectively software relies on for proper functioning ■

Example 5.1. . **Road Pricing System — Design Requirements**: The design requirements for the road pricing calculator of this chapter are for the design (ii) of that part of the vehicle software which interfaces the `GNSS` receiver and the road pricing calculator (cf. Items 394–397), (iii) of that part of the toll-gate software which interfaces the toll-gate and the road pricing calculator (cf. Items 402–404) and (i) of the road pricing calculator (cf. Items 433–446) ■

Example 5.2. . **Road Pricing System — Design Assumptions**: The design assumptions for the road pricing calculator include: (i) that *vehicles* behave as prescribed in Items 393–397, (ii) that the `GNSS` regularly offers vehicles correct information as to their global position (cf. Item 394), (iii) that *toll-gates* behave as prescribed in Items 399–404, and (iv) that the *road net* is formed and well-formed as defined in Examples 5.7–5.9 ■

Example 5.3. . **Toll-Gate System — Design Requirements**: The design requirements for the toll-gate system of this chapter are for the design of software for the toll-gate and its interfaces to the road pricing system, i.e., Items 398–399 ■

Example 5.4. . **Toll-Gate System — Design Assumptions**: The design assumptions for the toll-gate system include (i) that the vehicles behave as per Items 393–397, and (ii) that the road pricing calculator behave as per Items 433–446 ■

5.3.4 Derived Requirements

In building up the domain, interface and machine requirements a number of machine concepts are introduced. These machine concepts enable the expression of additional requirements. It is these we refer to as derived requirements. Techniques and tools espoused in such classical publications as [176, 103, 50, 114, 113] can in those cases be used to advantage.

5.4 Domain Requirements

Domain requirements primarily express the assumptions that a design must rely upon in order that that design can be verified. Although domain requirements firstly express assumptions it appears that the software designer is well-advised in also implementing, as data structures and procedures, the endurants, respectively perdurants expressed in the domain requirements prescriptions. Whereas domain endurants are "real-life" phenomena they are now, in domain requirements prescriptions, abstract concepts (to be represented by a machine).

Definition 53 **Domain Requirements Prescription:** A **domain requirements prescription** is that subset of the requirements prescription whose technical terms are defined in a domain description ■

To determine a relevant subset all we need is collaboration with requirements, cum domain stake-holders. Experimental evidence, in the form of example developments of requirements prescriptions from domain descriptions, appears to show that one can formulate techniques for such developments around a few domain-description-to-requirements-prescription operations. We suggest these: ***projection***, ***instantiation***, ***determination***, ***extension*** and ***fitting***. In Sect. 5.3.2 we mentioned that the order in which one performs these domain-description-to-domain-requirements-prescription operations is not necessarily the order in which we have listed them here, but, with notable exceptions, one is well-served in starting out requirements development by following this order.

5.4.1 Domain Projection

Definition 54 **Domain Projection:** By a **domain projection** we mean *a subset of the domain description, one which* ***projects out*** *all those endurants: parts, materials and components, as well as perdurants: actions, events and behaviours that the stake-holders do not wish represented or relied upon by the machine* ■

The resulting document is a ***partial domain requirements prescription***. In determining an appropriate subset the requirements engineer must secure that the final "projection prescription" is complete and consistent — that is, that there are no "dangling references", i.e., that all entities and their internal properties that are referred to are all properly defined.

Domain Projection — Narrative

We now start on a series of examples that illustrate domain requirements development.

Example 5.5. . **Domain Requirements. Projection: A Narrative Sketch**: We require that the road pricing system shall [at most] relate to the following domain entities – and only to these[8]: the net, its links and hubs, and their properties (unique identifiers, mereologies and some attributes), the vehicles, as endurants, and the general vehicle behaviours, as perdurants. We treat projection together with a concept of **simplification**. The example simplifications are vehicle positions and, related to the simpler vehicle position, vehicle behaviours. To prescribe and formalise this we copy the domain description. From that domain description we remove all mention of the hub insertion action, the link disappearance event, and the monitor ■

As a result we obtain $\Delta_{\mathscr{P}}$, the projected version of the domain requirements prescription[9].

[8] By 'relate to ... these' we mean that the required system does not rely on domain phenomena that have been "projected away".

[9] Restrictions of the net to the toll road nets, hinted at earlier, will follow in the next domain requirements steps.

Domain Projection — Formalisation

The requirements prescription hinges, crucially, not only on a systematic narrative of all the projected, instantiated, determinated, extended and fitted specifications, but also on their formalisation. In the formal domain projection example we, regretfully, omit the narrative texts. In bringing the formal texts we keep the item numbering from Sect. 5.2, where you can find the associated narrative texts.

Example 5.6. . **Domain Requirements — Projection**: **Main Sorts**

```
type
266    Δ_𝒫
266a   N_𝒫
266b   F_𝒫
value
266a   obs_part_N_𝒫: Δ_𝒫→N_𝒫
266b   obs_part_F_𝒫: Δ_𝒫→F_𝒫

type
267a   HA_𝒫
267b   LA_𝒫
value
267a   obs_part_HA: N_𝒫 → HA
267b   obs_part_LA: N_𝒫 → LA
```

Concrete Types

```
type
268    H_𝒫, HS_𝒫 = H_𝒫-set
269    L_𝒫, LS_𝒫 = L_𝒫-set
270    V_𝒫, VS_𝒫 = V_𝒫-set
value
268    obs_part_HS_𝒫: HA_𝒫 → HS_𝒫

269    obs_part_LS_𝒫: LA_𝒫 → LS_𝒫
270    obs_part_VS_𝒫: F_𝒫 → VS_𝒫
271a   links: Δ_𝒫 → L-set
271a   links(δ_𝒫) ≡ obs_part_LS_ℛ(obs_part_LA_ℛ(δ_ℛ))
271b   hubs: Δ_𝒫 → H-set
271b   hubs(δ_𝒫) ≡ obs_part_HS_𝒫(obs_part_HA_𝒫(δ_𝒫))
```

Unique Identifiers

```
type
272a   HI, LI, VI, MI
value
272c   uid_HI: H_𝒫 → HI
272c   uid_LI: L_𝒫 → LI

272c   uid_VI: V_𝒫 → VI
272c   uid_MI: M_𝒫 → MI
axiom
272b   HI∩LI=Ø, HI∩VI=Ø, HI∩MI=Ø,
272b   LI∩VI=Ø, LI∩MI=Ø, VI∩MI=Ø
```

Mereology

```
value
277    obs_mereo_H_𝒫: H_𝒫 → LI-set
278    obs_mereo_L_𝒫: L_𝒫 → HI-set
278        axiom ∀ l:L_𝒫 • card obs_mereo_L_𝒫(l)=2
279    obs_mereo_V_𝒫: V_𝒫 → MI
280    obs_mereo_M_𝒫: M_𝒫 → VI-set
axiom
281    ∀ δ_𝒫:Δ_𝒫, hs:HS•hs=hubs(δ), ls:LS•ls=links(δ_𝒫)
281        ∀ h:H_𝒫•h ∈ hs ⇒
281            obs_mereo_H_𝒫(h)⊆xtr_his(δ_𝒫) ∧
282        ∀ l:L_𝒫•l ∈ ls •
281            obs_mereo_L_𝒫(l)⊆xtr_lis(δ_𝒫) ∧
283a       let f:F_𝒫•f=obs_part_F_𝒫(δ_𝒫) ⇒
283a               vs:VS_𝒫•vs=obs_part_VS_𝒫(f) in
283a           ∀ v:V_𝒫•v ∈ vs ⇒
283a               uid_V_𝒫(v) ∈ obs_mereo_M_𝒫(m) ∧
283b           obs_mereo_M_𝒫(m)
283b               = {uid_V_𝒫(v)|v:V•v ∈ vs}
283b       end
```

Attributes: We project attributes of hubs, links and vehicles.
First **hubs:**

```
type
284a   GeoH
284b   HΣ_𝒫 = (LI×LI)-sett
284c   HΩ_𝒫 = HΣ_𝒫-set
value
284b   attr_HΣ_𝒫: H_𝒫 → HΣ_𝒫
284c   attr_HΩ_𝒫: H_𝒫 → HΩ_𝒫

axiom
285    ∀ δ_𝒫:Δ_𝒫,
285        let hs = hubs(δ_𝒫) in
285        ∀ h:H_𝒫 • h ∈ hs •
285a           xtr_lis(h)⊆xtr_lis(δ_𝒫)
285b         ∧ attr_Σ_𝒫(h) ∈ attr_Ω_𝒫(h)
285        end
```

Then **links:**

```
type
288    GeoL
289a   LΣ𝒫 = (HI×HI)-set
289b   LΩ𝒫 = LΣ𝒫-set
value
288    attr_GeoL: L → GeoL
289a   attr_LΣ𝒫: L𝒫 → LΣ𝒫
289b   attr_LΩ𝒫: L𝒫 → LΩ𝒫
axiom
       289a– 289b on Page 151.
```

Finally **vehicles:** For 'road pricing' we need vehicle positions. But, for "technical reasons", we must abstain from the detailed description given in Items 290–290c[10] We therefore **simplify** vehicle positions.

316 A simplified vehicle position designates
  a either a link
  b or a hub,

```
type
316    SVPos = SonL | SatH
316a   SonL :: LI
316b   SatH :: HI
axiom
290a'    ∀ n:N, SonL(li):SVPos •
290a'       ∃ l:L•l ∈obs_part_LS(obs_part_N(n)) ⇒ li=uid_L(l)
290c'    ∀ n:N, SatH(hi):SVPos •
290c'       ∃ h:H•h ∈obs_part_HS(obs_part_N(n)) ⇒ hi=uid_H(h)
```

Global Values

```
value
300   δ𝒫:Δ𝒫,
301   n:N𝒫 = obs_part_N𝒫(δ𝒫),
301   ls:L𝒫-set = links(δ𝒫),
301   hs:H𝒫-set = hubs(δ𝒫),
301   lis:LI-set = xtr_lis(δ𝒫),
301   his:HI-set = xtr_his(δ𝒫)
```

Behaviour Signatures: We omit the monitor behaviour.

317 We leave the vehicle behaviours' attribute argument undefined.

```
type
317   ATTR
value
307   trs𝒫: Unit → Unit
308   veh𝒫: VI×MI×ATTR → ... Unit
```

The System Behaviour: We omit the monitor behaviour.

```
value
310a  trs𝒫()=‖{veh𝒫(uid_VI(v),obs_mereo_V(v),_) | v:V𝒫•v ∈ vs}
```

The Vehicle Behaviour: Given the simplification of vehicle positions we *simplify* the vehicle behaviour given in Items 311–312

```
311'      veh_vi(mi)(vp:SatH(hi)) ≡
311a'         v_m_ch[vi,mi]!SatH(hi) ;
311a'         veh_vi(mi)(SatH(hi))
311(b)i'    ⌈⌉ let li:LI•li∈obs_mereo_H(get_hub(hi)(n))
311(b)ii'     in v_m_ch[vi,mi]!SonL(li) ;
311(b)ii'        veh_vi(mi)(SonL(li)) end
311c'       ⌈⌉ stop

312'      veh_vi(mi)(vp:SonL(li)) ≡
312a'         v_m_ch[vi,mi]!SonL(li) ;
312a'         veh_vi(mi)(SonL(li))
312(b)ii1'  ⌈⌉ let hi:HI•hi∈obs_mereo_L(get_link(li)(n))
312(b)ii2'      in v_m_ch[vi,mi]!SatH(hi) ;
312(b)ii2'         veh_vi(mi)(atH(hi)) end
312c'       ⌈⌉ stop
```

We can simplify Items 311′–312c′ further.

```
318   veh_vi(mi)(vp) ≡
319       v_m_ch[vi,mi]!vp ; veh_vi(mi)(vp)
320     ⌈⌉ case vp of
320         SatH(hi) →
321           let li:LI•li∈obs_mereo_H(get_hub(hi)(n))
322           in v_m_ch[vi,mi]!SonL(li) ;
322              veh_vi(mi)(SonL(li)) end,
320         SonL(li) →
323           let hi:HI•hi∈obs_mereo_L(get_link(li)(n))
324           in v_m_ch[vi,mi]!SatH(hi) ;
324              veh_vi(mi)(atH(hi)) end end
325     ⌈⌉ stop
```

318 This line coalesces Items 311′ and 312′.

[10] The 'technical reasons' are that we assume that the *GNSS* cannot provide us with direction of vehicle movement and therefore we cannot, using only the *GNSS* provide the details of 'offset' along a link (*onL*) nor the "from/to link" at a hub (*atH*).

319 Coalescing Items 311a′ and 312′.
320 Captures the distinct parameters of Items 311′ and 312′.
321 Item 311(b)i′.
322 Item 311(b)ii′.
323 Item 312(b)ii1′.
324 Item 312(b)ii2′.
325 Coalescing Items 311c′ and 312c′.

The above vehicle behaviour definition will be transformed (i.e., further "refined") in Sect. 5.5.1's Example 5.15; cf. Items 393– 397 on Page 169 ■

Discussion

Domain projection can also be achieved by developing a "completely new" domain description — typically on the basis of one or more existing domain description(s) — where that "new" description now takes the rôle of being the project domain requirements.

5.4.2 Domain Instantiation

Definition 55 Domain Instantiation: By **domain instantiation** we mean *a* ***refinement*** *of the partial domain requirements prescription (resulting from the projection step) in which the refinements aim at rendering the endurants: parts, materials and components, as well as the perdurants: actions, events and behaviours of the domain requirements prescription more concrete, more specific* ■ Instantiations usually render these concepts less general.

Properties that hold of the projected domain shall also hold of the (therefrom) instantiated domain.

Refinement of endurants can be expressed (i) either in the form of concrete types, (ii) or of further "delineating" axioms over sorts, (iii) or of a combination of concretisation and axioms. We shall exemplify the third possibility. Example 5.7 express requirements that the road net (on which the road-pricing system is to be based) must satisfy. Refinement of perdurants will not be illustrated (other than the simplification of the ***vehicle*** projected behaviour).

Domain Instantiation

Example 5.7. . **Domain Requirements. Instantiation Road Net**: We now require that there is, as before, a road net, $n_{\mathscr{I}}$:$N_{\mathscr{I}}$, which can be understood as consisting of two, "connected sub-nets". A toll-road net, $trn_{\mathscr{I}}$:$TRN_{\mathscr{I}}$, cf. Fig. 5.1, and an ordinary road net, $n_{\mathscr{P}'}$. The two are connected as follows: The toll-road net, $trn_{\mathscr{I}}$, borders some toll-road plazas, in Fig. 5.1 shown by white filled circles (i.e., hubs). These toll-road plaza hubs are proper hubs of the 'ordinary' road net, $n'_{\mathscr{P}}$.

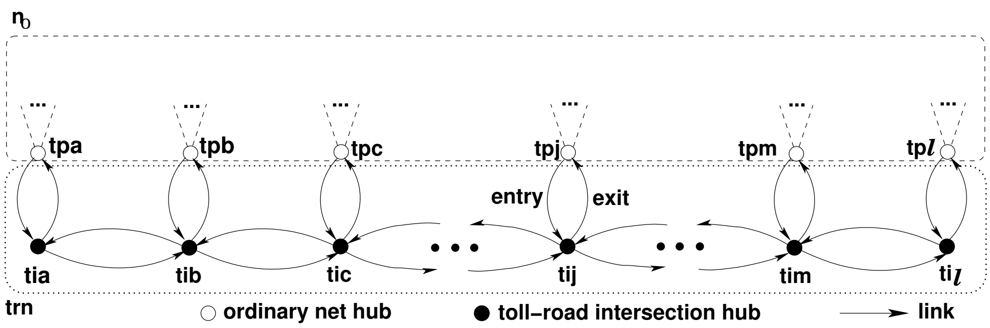


Fig. 5.1. A simple, linear toll-road net trn. tp_j: toll plaza j, ti_j: toll road intersection j.
Upper dashed sub-figure hint at an ordinary road net n_o.
Lower dotted sub-figure hint at a toll-road net trn.
Dash-dotted (- - -) "V"-images above tp_js hint at links to remaining "parts" of n_o.

326 The instantiated domain, $\delta_{\mathscr{I}}$:$\Delta_{\mathscr{I}}$ has just the net, $\mathsf{n}_{\mathscr{I}}$:$\mathsf{N}_{\mathscr{I}}$ being instantiated.
327 The road net consists of two "sub-nets"
 a an "ordinary" road net, n_o:$\mathsf{N}_{\mathscr{P}'}$ and
 b a toll-road net proper, trn:$\mathsf{TRN}_{\mathscr{I}}$ —
 c "connected" by an interface hil:HIL:
  i That interface consists of a number of toll-road plazas (i.e., hubs), modeled as a list of hub identifiers, hil:HI*.
  ii The toll-road plaza interface to the toll-road net, trn:$\mathsf{TRN}_{\mathscr{I}}$[11], has each plaza, hil[i], connected to a pair of toll-road links: an entry and an exit link: $(l_e{:}L, l_x{:}L)$.
  iii The toll-road plaza interface to the 'ordinary' net, n_o:$\mathsf{N}_{\mathscr{P}'}$, has each plaza, i.e., the hub designated by the hub identifier hil[i], connected to one or more ordinary net links, $\{l_{i_1}, l_{i_2}, \cdots, l_{i_k}\}$.
327b The toll-road net, trn:$\mathsf{TRN}_{\mathscr{I}}$, consists of three collections (modeled as lists) of links and hubs:
  i a list of pairs of toll-road entry/exit links: $\langle(l_{e_1}, l_{x_1}), \cdots, (l_{e_\ell}, l_{x_\ell})\rangle$,
  ii a list of toll-road intersection hubs: $\langle h_{i_1}, h_{i_2}, \cdots, h_{i_\ell}\rangle$, and
  iii a list of pairs of main toll-road ("up" and "down") links: $\langle(ml_{i_{1u}}, ml_{i_{1d}}), (m_{i_{2u}}, m_{i_{2d}}), \cdots, (m_{i_{\ell u}}, m_{i_{\ell d}})\rangle$.
 d The three lists have commensurate lengths (ℓ).

ℓ is the number of toll plazas, hence also the number of toll-road intersection hubs and therefore a number one larger than the number of pairs of main toll-road ("up" and "down") links

type
326 $\Delta_{\mathscr{I}}$
327 $\mathsf{N}_{\mathscr{I}} = \mathsf{N}_{\mathscr{P}'} \times \mathsf{HIL} \times \mathsf{TRN}$
327a $\mathsf{N}_{\mathscr{P}'}$
327b $\mathsf{TRN}_{\mathscr{I}} = (\mathsf{L}\times\mathsf{L})^* \times \mathsf{H}^* \times (\mathsf{L}\times\mathsf{L})^*$
327c HIL = HI*

axiom
327d ∀ $\mathsf{n}_{\mathscr{I}}$:$\mathsf{N}_{\mathscr{I}}$ •
327d **let** (n_Δ,hil,(exll,hl,lll)) = $\mathsf{n}_{\mathscr{I}}$ **in**
327d **len** hil = **len** exll = **len** hl = **len** lll + 1
327d **end**

We have named the "ordinary" net sort (primed) $\mathsf{N}_{\mathscr{P}'}$. It is "almost" like (unprimed) $\mathsf{N}_{\mathscr{P}}$ — except that the interface hubs are also connected to the toll-road net entry and exit links.

The partial concretisation of the net sorts, $\mathsf{N}_{\mathscr{P}}$, into $\mathsf{N}_{\mathscr{I}}$ requires some additional well-formedness conditions to be satisfied.

328 The toll-road intersection hubs all[12] have distinct identifiers.

```
328 wf_dist_toll_road_isect_hub_ids: H*→Bool
328 wf_dist_toll_road_isect_hub_ids(hl) ≡
328     len hl = card xtr_his(hl)
```

329 The toll-road links all have distinct identifiers.

```
329 wf_dist_toll_road_u_d_link_ids: (L×L)*→Bool
329 wf_dist_toll_road_u_d_link_ids(lll) ≡
329     2 × len lll = card xtr_lis(lll)
```

330 The toll-road entry/exit links all have distinct identifiers.

```
330 wf_dist_e_x_link_ids: (L×L)*→Bool
330 wf_dist_e_x_link_ids(exll) ≡
330     2 × len exll = card xtr_lis(exll)
```

331 Proper net links must not designate toll-road intersection hubs.

```
331 wf_isoltd_toll_road_isect_hubs: HI*×H*→N_I→Bool
331 wf_isoltd_toll_road_isect_hubs(hil,hl)(n_I) ≡
331     let ls=xtr_links(n_I) in
331     let his = ∪ {obs_mereo_L(l)|l:L•l ∈ ls} in
331     his ∩ xtr_his(hl) = {} end end
```

332 The plaza hub identifiers must designate hubs of the 'ordinary' net.

```
332 wf_p_hubs_pt_of_ord_net: HI*→N'_Δ→Bool
332 wf_p_hubs_pt_of_ord_net(hil)(n'_Δ) ≡
332     elems hil ⊆ xtr_his(n'_Δ)
```

[11] We (sometimes) omit the subscript $\mathscr{I}$ when it should be clear from the context what we mean.
[12] A 'must' can be inserted in front of all 'all's,

333 The plaza hub mereologies must each,
 a besides identifying at least one hub of the ordinary net,
 b also identify the two entry/exit links with which they are supposed to be connected.

333 wf_p_hub_interf: N'_Δ→**Bool**
333 wf_p_hub_interf(n_o,hil,(exll,_,_)) ≡
333 ∀ i:**Nat** • i ∈ **inds** exll ⇒
333   **let** h = get_H(hil(i))(n'_Δ) **in**
333   **let** lis = **obs_mereo_H**(h) **in**
333   **let** lis′ = lis \ xtr_lis(n′) **in**
333   lis′ = xtr_lis(exll(i)) **end end end**

334 The mereology of each toll-road intersection hub must identify
 a the entry/exit links
 b and exactly the toll-road 'up' and 'down' links
 c with which they are supposed to be connected.

334 wf_toll_road_isect_hub_iface: $N_{\mathscr{I}}$→**Bool**
334 wf_toll_road_isect_hub_iface(_,_,(exll,hl,lll)) ≡
334 ∀ i:**Nat** • i ∈ **inds** hl ⇒
334   **obs_mereo_H**(hl(i)) =
334a   xtr_lis(exll(i)) ∪
334   **case** i **of**
334b   1 → xtr_lis(lll(1)),
334b   **len** hl → xtr_lis(lll(**len** hl−1))
334b   _ → xtr_lis(lll(i)) ∪ xtr_lis(lll(i−1))
334   **end**

335 The mereology of the entry/exit links must identify exactly the
 a interface hubs and the
 b toll-road intersection hubs
 c with which they are supposed to be connected.

335 wf_exll: (L×L)*×HI*×H*→**Bool**
335 wf_exll(exll,hil,hl) ≡
335 ∀ i:**Nat** • i ∈ **len** exll
335   **let** (hi,(el,xl),h) = (hil(i),exll(i),hl(i)) **in**
335   **obs_mereo_L**(el) = **obs_mereo_L**(xl)
335   = {hi} ∪ {**uid_H**(h)} **end**
335 **pre**: **len** eell = **len** hil = **len** hl

336 The mereology of the toll-road 'up' and 'down' links must
 a identify exactly the toll-road intersection hubs
 b with which they are supposed to be connected.

336 wf_u_d_links: (L×L)*×H*→**Bool**
336 wf_u_d_links(lll,hl) ≡
336 ∀ i:**Nat** • i ∈ **inds** lll ⇒
336   **let** (ul,dl) = lll(i) **in**
336   **obs_mereo_L**(ul) = **obs_mereo_L**(dl) =
336a   **uid_H**(hl(i)) ∪ **uid_H**(hl(i+1)) **end**
336 **pre**: **len** lll = **len** hl+1

We have used some additional auxiliary functions:

 xtr_his: H*→**HI-set**
 xtr_his(hl) ≡ {**uid_HI**(h)|h:H•h ∈ **elems** hl}
 xtr_lis: (L×L)→**LI-set**
 xtr_lis(l′,l″) ≡ {**uid_LI**(l′)}∪{**uid_LI**(l″)}
 xtr_lis: (L×L)*− **LI-set**
 xtr_lis(lll) ≡
 ∪{xtr_lis(l′,l″)|(l′,l″):(L×L)•(l′,l″)∈ **elems** lll}

337 The well-formedness of instantiated nets is now the conjunction of the individual well-formedness predicates above.

337 wf_instantiated_net: $N_{\mathscr{I}}$ → **Bool**
337 wf_instantiated_net(n'_Δ,hil,(exll,hl,lll))
328 wf_dist_toll_road_isect_hub_ids(hl)
329 ∧ wf_dist_toll_road_u_d_link_ids(lll)
330 ∧ wf_dist_e_e_link_ids(exll)
331 ∧ wf_isolated_toll_road_isect_hubs(hil,hl)(n′)
332 ∧ wf_p_hubs_pt_of_ord_net(hil)(n′)
333 ∧ wf_p_hub_interf(n'_Δ,hil,(exll,_,_))
334 ∧ wf_toll_road_isect_hub_iface(_,_,(exll,hl,lll))
335 ∧ wf_exll(exll,hil,hl)
336 ∧ wf_u_d_links(lll,hl)

Domain Instantiation — Abstraction

Example 5.8. . **Domain Requirements. Instantiation Road Net, Abstraction**: Domain instantiation has refined an abstract definition of net sorts, $n_{\mathscr{P}}$:$N_{\mathscr{P}}$, into a partially concrete definition of nets, $n_{\mathscr{I}}$:$N_{\mathscr{I}}$. We need to show the refinement relation:

- abstraction($n_{\mathscr{I}}$) = $n_{\mathscr{P}}$.

value
338 $\mathsf{abstraction}: \mathsf{N}_{\mathscr{I}} \to \mathsf{N}_{\mathscr{P}}$
339 $\mathsf{abstraction}(\mathsf{n}'_{\Delta},\mathsf{hil},(\mathsf{exll},\mathsf{hl},\mathsf{lll})) \equiv$
340 $\quad$ **let** $\mathsf{n}_{\mathscr{P}}:\mathsf{N}_{\mathscr{P}} \bullet$
340 $\quad\quad$ **let** $\mathsf{hs}=\mathbf{obs\_part\_HS}_{\mathscr{P}}(\mathbf{obs\_part\_HA}_{\mathscr{P}}(\mathsf{n}'_{\mathscr{P}})),$
340 $\quad\quad\quad \mathsf{ls}=\mathbf{obs\_part\_LS}_{\mathscr{P}}(\mathbf{obs\_part\_LA}_{\mathscr{P}}(\mathsf{n}'_{\mathscr{P}})),$
340 $\quad\quad\quad \mathsf{ths}=$ **elems** $\mathsf{hl},$
340 $\quad\quad\quad \mathsf{eells}=\mathsf{xtr\_links}(\mathsf{eell}),\ \mathsf{llls}=\mathsf{xtr\_links}(\mathsf{lll})$ **in**
341 $\quad\quad\quad \mathsf{hs} \cup \mathsf{ths} =$
341 $\quad\quad\quad\quad \mathbf{obs\_part\_HS}_{\mathscr{P}}(\mathbf{obs\_part\_HA}_{\mathscr{P}}(\mathsf{n}_{\mathscr{P}}))$
342 $\quad\quad \wedge\ \mathsf{ls} \cup \mathsf{eells} \cup \mathsf{llls} =$
342 $\quad\quad\quad\quad \mathbf{obs\_part\_LS}_{\mathscr{P}}(\mathbf{obs\_part\_LA}_{\mathscr{P}}(\mathsf{n}_{\mathscr{P}}))$
343 $\quad$ **in** $\mathsf{n}_{\mathscr{P}}$ **end end**

338 The abstraction function takes a concrete net, $\mathsf{n}_{\mathscr{I}}:\mathsf{N}_{\mathscr{I}}$, and yields an abstract net, $\mathsf{n}_{\mathscr{P}}:\mathsf{N}_{\mathscr{P}}$.
339 The abstraction function doubly decomposes its argument into constituent lists and sub-lists.
340 There is postulated an abstract net, $\mathsf{n}_{\mathscr{P}}:\mathsf{N}_{\mathscr{P}}$, such that
341 the hubs of the concrete net and toll-road equals those of the abstract net, and
342 the links of the concrete net and toll-road equals those of the abstract net.
343 And that abstract net, $\mathsf{n}_{\mathscr{P}}:\mathsf{N}_{\mathscr{P}}$, is postulated to be an abstraction of the concrete net.

Discussion

Domain descriptions, such as illustrated in [2, ***Manifest Domains: Analysis & Description***] and in this chapter, model families of concrete, i.e., specifically occurring domains. Domain instantiation, as exemplified in this section (i.e., Sect. 5.4.2), "narrow down" these families. Domain instantiation, such as it is defined, cf. Definition 55 on Page 160, allows the requirements engineer to instantiate to a concrete instance of a very specific domain, that, for example, of the toll-road between *Bolzano Nord* and *Trento Sud* in Italy (i.e., n=7)[13].

5.4.3 Domain Determination

Definition 56 **Determination:** By **domain determination** we mean *a refinement of the partial domain requirements prescription, resulting from the instantiation step, in which the refinements aim at rendering the endurants: parts, materials and components, as well as the perdurants: functions, events and behaviours of the partial domain requirements prescription* ***less non-determinate, more determinate*** ■

Determinations usually render these concepts less general. That is, the value space of endurants that are made more determinate is "smaller", contains fewer values, as compared to the endurants before determination has been "applied".

Domain Determination: Example

We show an example of 'domain determination'. It is expressed sôlely in terms of axioms over the concrete toll-road net type.

Example 5.9. . **Domain Requirements. Determination Toll-roads**: We focus only on the toll-road net. We single out only two 'determinations':

All Toll-road Links are One-way Links

344 *The entry/exit and toll-road links*
- a are always all one way links,
- b as indicated by the arrows of Fig. 5.1 on Page 160,
- c such that each pair allows traffic in opposite directions.

344 $\mathsf{opposite\_traffics}: (\mathsf{L}\times\mathsf{L})^* \times (\mathsf{L}\times\mathsf{L})^* \to$ **Bool**
344 $\mathsf{opposite\_traffics}(\mathsf{exll},\mathsf{lll}) \equiv$
344 $\quad \forall\ (\mathsf{lt},\mathsf{lf}):(\mathsf{L}\times\mathsf{L}) \bullet (\mathsf{lt},\mathsf{lf}) \in$ **elems** $\mathsf{exll}^\frown\mathsf{lll} \Rightarrow$
344a $\quad\quad$ **let** $(\mathsf{lt}\sigma,\mathsf{lf}\sigma) = (\mathbf{attr\_L\Sigma}(\mathsf{lt}),\mathbf{attr\_L\Sigma}(\mathsf{lf}))$ **in**
344a′. $\quad\quad\quad \mathbf{attr\_L\Omega}(\mathsf{lt})=\{\mathsf{lt}\sigma\} \wedge \mathbf{attr\_L\Omega}(\mathsf{ft})=\{\mathsf{ft}\sigma\}$
344a″. $\quad\quad \wedge$ **card** $\mathsf{lt}\sigma = 1 =$ **card** $\mathsf{lf}\sigma$
344 $\quad\quad \wedge$ **let** $(\{(\mathsf{hi},\mathsf{hi}')\},\{(\mathsf{hi}'',\mathsf{hi}''')\}) = (\mathsf{lt}\sigma,\mathsf{lf}\sigma)$ **in**
344c $\quad\quad\quad \mathsf{hi}=\mathsf{hi}''' \wedge \mathsf{hi}'=\mathsf{hi}''$
344 $\quad\quad$ **end end**

Predicates 344a′. and 344a″. express the same property.

[13] Here we disregard the fact that this toll-road does not start/end in neither *Bolzano Nord* nor *Trento Sud*.

All Toll-road Hubs are Free-flow

345 *The hub state spaces* are singleton sets of the toll-road hub states which always allow exactly these (and only these) crossings:
  a from *e*ntry links back to the paired e*x*it *l*ink*s*,
  b from *e*ntry links to emanating *t*oll-road *l*ink*s*,
  c from incident *t*oll-road links to e*x*it *l*ink*s*, and
  d from incident *t*oll-road link to emanating *t*oll-road *l*ink*s*.

```
345   free_flow_toll_road_hubs: (L×L)*×(L×L)*→Bool
345   free_flow_toll_road_hubs(exl,ll) ≡
345       ∀ i:Nat•i ∈ inds hl ⇒
345           attr_HΣ(hl(i)) =
345a              hσ_ex_ls(exl(i))
345b            ∪ hσ_et_ls(exl(i),(i,ll))
345c            ∪ hσ_tx_ls(exl(i),(i,ll))
345d            ∪ hσ_tt_ls(i,ll)
```

345a: from *e*ntry links back to the paired e*x*it *l*ink*s*:

```
345a  hσ_ex_ls: (L×L)→LΣ
345a  hσ_ex_ls(e,x) ≡ {(uid_LI(e),uid_LI(x))}
```

345b: from *e*ntry links to emanating *t*oll-road *l*ink*s*:

```
345b  hσ_et_ls: (L×L)×(Nat×(em:L×in:L)*)→LΣ
345b  hσ_et_ls((e,_),(i,ll)) ≡
345b      case i of
345b          2 → {(uid_LI(e),uid_LI(em(ll(1))))},
345b          len ll+1 →
345b                  {(uid_LI(e),uid_LI(em(ll(len ll))))},
345b          _ → {(uid_LI(e),uid_LI(em(ll(i−1)))),
345b                (uid_LI(e),uid_LI(em(ll(i))))}
345b      end
```

The *em* and *in* in the toll-road link list (em:L×in:L)* designate selectors for *em*anating, respectively *in*cident links. 345c: from incident *t*oll-road links to e*x*it *l*ink*s*:

```
345c  hσ_tx_ls: (L×L)×(Nat×(em:L×in:L)*)→LΣ
345c  hσ_tx_ls((_,x),(i,ll)) ≡
345c      case i of
345c          2 → {(uid_LI(in(ll(1))),uid_LI(x))},
345c          len ll+1 →
345c                  {(uid_LI(in(ll(len ll))),uid_LI(x))},
345c          _ → {(uid_LI(in(ll(i−1))),uid_LI(x)),
345c                (uid_LI(in(ll(i))),uid_LI(x))}
345c      end
```

345d: from incident *t*oll-road link to emanating *t*oll-road *l*ink*s*:

```
345d  hσ_tt_ls: Nat×(em:L×in:L)*→LΣ
345d  hσ_tt_ls(i,ll) ≡
345d      case i of
345d          2 → {(uid_LI(in(ll(1))),
345d                uid_LI(em(ll(1))))},
345d          len ll+1 →
345d                  {(uid_LI(in(ll(len ll))),
345d                   uid_LI(em(ll(len ll))))},
345d          _ → {(uid_LI(in(ll(i−1))),
345d                uid_LI(em(ll(i−1)))),
345d               (uid_LI(in(ll(i))),
345d                uid_LI(em(ll(i))))}
345d      end
```

The example above illustrated 'domain determination' with respect to endurants. Typically "endurant determination" is expressed in terms of axioms that limit state spaces — where "endurant instantiation" typically "limited" the mereology of endurants: how parts are related to one another. We shall not exemplify domain determination with respect to perdurants.

Discussion

The borderline between instantiation and determination is fuzzy. Whether, as an example, fixing the number of toll-road intersection hubs to a constant value, e.g., n=7, is instantiation or determination, is really a matter of choice !

5.4.4 Domain Extension

Definition 57 Extension: By **domain extension** we understand *the **introduction of endurants (see Sect. 5.4.4) and perdurants (see Sect. 5.5.2) that were not feasible in the original domain,** but for which, with computing and communication, and with new, emerging technologies, for example, sensors, actuators and satellites, there is the possibility of feasible implementations, hence the requirements, that what is introduced becomes part of the unfolding requirements prescription* ■

Endurant Extensions

Definition 58 **Endurant Extension:** By an **endurant extension** we understand the introduction of one or more endurants into the projected, instantiated and determined domain $\mathscr{D}_{\mathscr{R}}$ resulting in domain $\mathscr{D}_{\mathscr{R}}'$, such that these form a ***conservative extension*** of the theory, $\mathscr{T}_{\mathscr{D}_{\mathscr{R}}}$ denoted by the domain requirements $\mathscr{D}_{\mathscr{R}}$ (i.e., "before" the extension), that is: every theorem of $\mathscr{T}_{\mathscr{D}_{\mathscr{R}}}$ is still a theorem of $\mathscr{T}_{\mathscr{D}_{\mathscr{R}}'}$.

Usually domain extensions involve one or more of the already introduced sorts. In Example 5.10 we introduce (i.e., "extend") vehicles with GPSS-like sensors, and introduce toll-gates with entry sensors, vehicle identification sensors, gate actuators and exit sensors. Finally road pricing calculators are introduced.

Example 5.10. . **Domain Requirements — Endurant Extension**: We present the extensions in several steps. Some of them will be developed in this section. Development of the remaining will be deferred to Sect. 5.5.1. The reason for this deferment is that those last steps are examples of **interface requirements**. The initial extension-development steps are: [a] vehicle extension, [b] sort and unique identifiers of road price calculators, [c] vehicle to road pricing calculator channel, [d] sorts and dynamic attributes of toll-gates, [e] road pricing calculator attributes, [f] "total" system state, and [g] the overall system behaviour. This decomposition establishes system interfaces in "small, easy steps".

[a] Vehicle Extension:

346 There is a domain, $\delta_{\mathscr{E}}:\Delta_{\mathscr{E}}$, which contains

347 a fleet, $f_{\mathscr{E}}:F_{\mathscr{E}}$, that is,

348 a set, $vs_{\mathscr{E}}:VS_{\mathscr{E}}$, of

349 extended vehicles, $v_{\mathscr{E}}:V_{\mathscr{E}}$ — their extension amounting to

350 a dynamic reactive attribute, whose value, tigpos:TiGpos, at any time, reflects that vehicle's **time-stamped global position.**[14]

351 The vehicle's GNSS receiver calculates, *loc_pos*, its local position, lpos:LPos, based on these signals.

352 Vehicles access these **external attributes** via the **external attribute** channel, attr_TiGPos_ch.

```
type
346   Δℰ
347   Fℰ
348   VSℰ = Vℰ-set
349   Vℰ
350   TiGPos = 𝕋 × GPos
351   GPos, LPos
value
346   δℰ:Δℰ
347   obs_part_Fℰ: Δℰ → Fℰ
347   f = obs_part_Fℰ(δℰ)
348   obs_part_VSℰ: Fℰ → VSℰ
348   vs = obs_part_VSℰ(f)
348   vis = xtr_vis(vs)
350   attr_TiGPos_ch[vi]?
351   loc_pos: GPos → LPos
channel
351   {attr_TiGPos_ch[vi]|vi:VI•vi ∈ vis}:TiGPos
```

We define two auxiliary functions,

353 xtr_vs, which given a domain, or a fleet, extracts its set of vehicles, and

354 xtr_vis which given a set of vehicles generates their unique identifiers.

```
value
353   xtr_vs: (Δℰ|Fℰ|VSℰ) → Vℰ-set
353   xtr_vs(arg) ≡
353     is_Δℰ(arg) →
353       obs_part_VSℰ(obs_part_Fℰ(arg)),
353     is_Fℰ(arg) → obs_part_VSℰ(arg),
353     is_VSℰ(arg) → arg
354   xtr_vis: (Δℰ|Fℰ|VSℰ) → VI-set
354   xtr_vis(arg) ≡ {uid_VI(v)|v ∈ xtr_vs(arg)}
```

[b] Road Pricing Calculator: Basic Sort and Unique Identifier:

355 The domain $\delta_{\mathscr{E}}:\Delta_{\mathscr{E}}$, also contains a pricing calculator, $c:C_{\delta_{\mathscr{E}}}$, with unique identifier ci:CI.

```
type
355   C, CI
value
355   obs_part_C: Δℰ → C
```

[14] We refer to literature on GNSS, ***global navigation satellite systems***. The simple vehicle position, vp:SVPos, is determined from three to four time-stamped signals received from a like number of GNSS satellites [177].

355 **uid_CI**: C → CI

355 c = **obs_part_C**$(\delta_\mathscr{E})$
355 ci = **uid_CI**(c)

[c] Vehicle to Road Pricing Calculator Channel:

356 Vehicles can, on their own volition, offer the timed local position, viti-lpos:VITiLPos
357 to the pricing calculator, c:$C_\mathscr{E}$ along a vehicles-to-calculator channel, v_c_ch.

type
356 VITiLPos = VI × ($\mathbb{T}$ × LPos)
channel
357 {v_c_ch[vi,ci]|vi:VI,ci:CI•vi∈vis∧ci=**uid**_C(c)}:VITiLPos

[d] Toll-gate Sorts and Dynamic Types:

We extend the domain with toll-gates for vehicles entering and exiting the toll-road entry and exit links. Figure 5.2 illustrates the idea of gates.

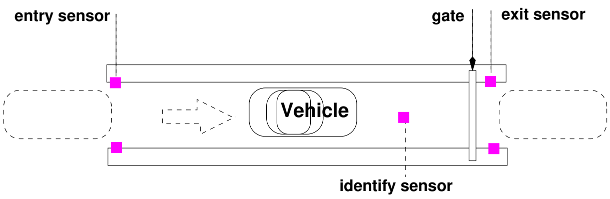


Fig. 5.2. A toll plaza gate

Figure 5.2 is intended to illustrate a vehicle entering (or exiting) a toll-road arrival link. The toll-gate is equipped with three sensors: an arrival sensor, a vehicle identification sensor and an departure sensor. The arrival sensor serves to prepare the vehicle identification sensor. The departure sensor serves to prepare the gate for closing when a vehicle has passed. The vehicle identify sensor identifies the vehicle and "delivers" a pair: the current time and the vehicle identifier. Once the vehicle identification sensor has identified a vehicle the gate opens and a message is sent to the road pricing calculator as to the passing vehicle's identity and the identity of the link associated with the toll-gate (see Items 374- 375 on the facing page).

358 The domain contains the extended net, n:$N_\mathscr{E}$,
359 with the net extension amounting to the toll-road net, $TRN_\mathscr{E}$, that is, the instantiated toll-road net, trn:$TRN_\mathscr{I}$, is extended, into trn:$TRN_\mathscr{E}$, with **entry,** eg:EG, and **exit,** xg:XG, toll-gates.

From entry- and exit-gates we can observe

360 their unique identifier and
361 their mereology: pairs of entry-, respectively exit link and calculator unique identifiers; further
362 a pair of gate entry and exit sensors modeled as **external attribute** channels, (ges:ES,gls:XS), and
363 a time-stamped vehicle identity sensor modeled as **external attribute** channels.

type
358 $N_\mathscr{E}$
359 $TRN_\mathscr{E}$ = (EG×XG)* × $TRN_\mathscr{I}$
360 GI
value
358 **obs_part_N**$_\mathscr{E}$: $\Delta_\mathscr{E}$ → $N_\mathscr{E}$
359 **obs_part_TRN**$_\mathscr{E}$: $N_\mathscr{E}$ → $TRN_\mathscr{E}$
360 **uid_G**: (EG|XG) → GI
361 **obs_mereo_G**: (EG|XG) → (LI×CI)
359 trn:$TRN_\mathscr{E}$ = **obs_part_TRN**$_\mathscr{E}(\delta_\mathscr{E})$
channel
362 {attr_entry_ch[gi]|gi:GI•xtr_eGIds(trn)} ″`enter`″
362 {attr_exit_ch[gi]|gi:GI•xtr_xGIds(trn)} ″`exit`″
363 {attr_identity_ch[gi]|gi:GI•xtr_GIds(trn)} TIVI
type
363 TIVI = $\mathbb{T}$ × VI

We define some **auxiliary functions** over toll-road nets, trn:$TRN_\mathscr{E}$:

364 xtr_eGℓ extracts the ℓist of entry gates,
365 xtr_xGℓ extracts the ℓist of exit gates,
366 xtr_eGIds extracts the set of entry gate identifiers,
367 xtr_xGIds extracts the set of exit gate identifiers,
368 xtr_Gs extracts the set of all gates, and
369 xtr_GIds extracts the set of all gate identifiers.

```
value
364   xtr_eGℓ: TRN_ℰ → EG*
364   xtr_eGℓ(pgl,_) ≡ {eg|(eg,xg):(EG,XG)•(eg,xg)∈ elems pgl}
365   xtr_xGℓ: TRN_ℰ → XG*
365   xtr_xGℓ(pgl,_) ≡ {xg|(eg,xg):(EG,XG)•(eg,xg)∈ elems pgl}
366   xtr_eGIds: TRN_ℰ → GI-set
366   xtr_eGIds(pgl,_) ≡ {uid_GI(g)|g:EG•g ∈ xtr_eGs(pgl,_)}
367   xtr_xGIds: TRN_ℰ → GI-set
367   xtr_xGIds(pgl,_) ≡ {uid_GI(g)|g:EG•g ∈ xtr_xGs(pgl,_)}
368   xtr_Gs: TRN_ℰ → G-set
368   xtr_Gs(pgl,_) ≡ xtr_eGs(pgl,_) ∪ xtr_xGs(pgl,_)
369   xtr_GIds: TRN_ℰ → GI-set
369   xtr_GIds(pgl,_) ≡ xtr_eGIds(pgl,_) ∪ xtr_xGIds(pgl,_)
```

370 A **well-formedness condition** expresses
  a that there are as many entry end exit gate pairs as there are toll-plazas,
  b that all gates are uniquely identified, and
  c that each entry [exit] gate is paired with an entry [exit] link and has that link's unique identifier as one element of its mereology, the other elements being the calculator identifier and the vehicle identifiers.

The well-formedness relies on awareness of

371 the unique identifier, ci:CI, of the road pricing calculator, c:C, and

372 the unique identifiers, vis:VI-**set**, of the fleet vehicles.

```
axiom
370    ∀ n:N_ℛ3, trn:TRN_ℛ3 •
370      let (exgl,(exl,hl,lll)) = obs_part_TRN_ℛ3(n) in
370a       len exgl = len exl = len hl = len lll + 1
370b     ∧ card xtr_GIds(exgl) = 2 ∗ len exgl
370c     ∧ ∀ i:Nat•i ∈ inds exgl•
370c          let ((eg,xg),(el,xl)) = (exgl(i),exl(i)) in
370c          obs_mereo_G(eg) = (uid_U(el),ci,vis)
370c        ∧ obs_mereo_G(xg) = (uid_U(xl),ci,vis)
370      end end
```

[e] Toll-gate to Calculator Channels:

373 We distinguish between **entry** and **exit** gates.

374 Toll road entry and exit gates offers the road pricing calculator a pair: whether it is an entry or an exit gates, and pair of the passing vehicle's identity and the time-stamped identity of the link associated with the toll-gate

375 to the road pricing calculator via a (gate to calculator) channel.

```
type
373    EE = ″entry″|″exit″
374    EEVITiLI = EE×(VI×(𝕋×SonL))
channel
375    {g_c_ch[gi,ci]|gi:GI•gi ∈ gis}:EETiVILI
```

[f] Road Pricing Calculator Attributes:

376 The road pricing attributes include a programmable traffic map, trm:TRM, which, for each vehicle inside the toll-road net, records a chronologically ordered list of each vehicle's timed position, (τ,lpos), and

377 a static (total) road location function, vplf:VPLF. The *v*ehicle *p*osition *l*ocation *f*unction, vplf:VPLF, which, given a local position, lpos:LPos, yields *either* the simple vehicle position, svpos:SVPos, designated by the GNSS-provided position, *or* yields the response that the provided position is off the toll-road net The vplf:VPLF function is constructed, construct_vplf,

378 from awareness, of a geodetic road map, GRM, of the topology of the extended net, $n_{\mathscr{E}}$:$N_{\mathscr{E}}$, including the mereology and the geodetic attributes of links and hubs.

```
type
376       TRM=VI →m (𝕋×SVPos)*
377       VPLF=GRM→LPos→(SVPos|″off_N″)
378       GRM
value
376       attr_TRM: C_ℰ → TRM
377       attr_VPLF: C_ℰ → VPLF
```

The geodetic road map maps geodetic locations into hub and link identifiers.

288 Geodetic link locations represent the set of point locations of a link.

284a Geodetic hub locations represent the set of point locations of a hub.

379 A geodetic road map maps geodetic link locations into link identifiers and geodetic hub locations into hub identifiers.

380 We sketch the construction, *geo_GRM*, of geodetic road maps.

type
379 GRM = (GeoL $\overrightarrow{m}$ LI) ∪ (GeoH $\overrightarrow{m}$ HI)
value
380 geo_GRM: N → GRM
380 geo_GRM(n) ≡
380 **let** ls = xtr_links(n), hs = xtr_hubs(n) **in**
380 [**attr**_GeoL(l)↦**uid**_LI(l)|l:L•l ∈ ls]
380 ∪
380 [**attr**_GeoH(h)↦**uid**_HI(h)|h:H•h ∈ hs] **end**

381 The vplf:VPLF function obtains a simple vehicle position, svpos, from a geodetic road map, grm:GRM, and a local position , lpos:

value
381 obtain_SVPos: GRM → LPos → SVPos
381 obtain_SVPos(grm)(lpos) **as** svpos
381 **post**: **case** svpos **of**
381 SatH(hi) → within(lpos,grm(hi)),
381 SonL(li) → within(lpos,grm(li)),
381 ″off_N″ → **true end**

where *within* is a predicate which holds if its first argument, a local position calculated from a GNSS-generated global position, falls within the point set representation of the geodetic locations of a link or a hub. The design of the *obtain_SVPos* represents an interesting challenge.

[g] "Total" System State:

Global values:

382 There is a given domain, $\delta_{\mathscr{E}}$:$\Delta_{\mathscr{E}}$;
383 there is the net, $n_{\mathscr{E}}$:$N_{\mathscr{E}}$, of that domain;
384 there is toll-road net, $trn_{\mathscr{E}}$:$TRN_{\mathscr{E}}$, of that net;
385 there is a set, $egs_{\mathscr{E}}$:$EG_{\mathscr{E}}$**-set**, of entry gates;
386 there is a set, $xgs_{\mathscr{E}}$:$XG_{\mathscr{E}}$**-set**, of exit gates;
387 there is a set, $gis_{\mathscr{E}}$:$GI_{\mathscr{E}}$**-set**, ofgate identifiers;
388 there is a set, $vs_{\mathscr{E}}$:$V_{\mathscr{E}}$**-set**, of vehicles;
389 there is a set, $vis_{\mathscr{E}}$:$VI_{\mathscr{E}}$**-set**, of vehicle identifiers;
390 there is the road-pricing calculator, $c_{\mathscr{E}}$:$C_{\mathscr{E}}$ and
391 there is its unique identifier, $ci_{\mathscr{E}}$:CI.

value
382 $\delta_{\mathscr{E}}$:$\Delta_{\mathscr{E}}$
383 $n_{\mathscr{E}}$:$N_{\mathscr{E}}$ = **obs_part**_$N_{\mathscr{E}}$($\delta_{\mathscr{E}}$)
384 $trn_{\mathscr{E}}$:$TRN_{\mathscr{E}}$ = **obs_part**_$TRN_{\mathscr{E}}$($n_{\mathscr{E}}$)
385 $egs_{\mathscr{E}}$:EG-**set** = xtr_egs($trn_{\mathscr{E}}$)
386 $xgs_{\mathscr{E}}$:XG-**set** = xtr_xgs($trn_{\mathscr{E}}$)
387 $gis_{\mathscr{E}}$:XG-**set** = xtr_gis($trn_{\mathscr{E}}$)
388 $vs_{\mathscr{E}}$:$V_{\mathscr{E}}$-**set** = **obs_part**_VS(**obs_part**_$F_{\mathscr{E}}$($\delta_{\mathscr{E}}$))
389 $vis_{\mathscr{E}}$:VI-**set** = {**uid**_VI($v_{\mathscr{E}}$)|$v_{\mathscr{E}}$:$V_{\mathscr{E}}$•$v_{\mathscr{E}}$ ∈ $vs_{\mathscr{E}}$}
390 $c_{\mathscr{E}}$:$C_{\mathscr{E}}$ = **obs_part**_$C_{\mathscr{E}}$($\delta_{\mathscr{E}}$)
391 $ci_{\mathscr{E}}$:$CI_{\mathscr{E}}$ = **uid**_CI($c_{\mathscr{E}}$)

In the following we shall omit the cumbersome $_{\mathscr{E}}$ subscripts.

[h] "Total" System Behaviour:

The signature and definition of the system behaviour is sketched as are the signatures of the vehicle, toll-gate and road pricing calculator. We shall model the behaviour of the road pricing system as follows: we shall not model behaviours nets, hubs and links; thus we shall model only the behaviour of vehicles, veh, the behaviour of toll-gates, gate, and the behaviour of the road-pricing calculator, calc, The behaviours of vehicles and toll-gates are presented here. But the behaviour of the road-pricing calculator is "deferred" till Sect. 5.5.1 since it reflects an interface requirements.

392 The road pricing system behaviour, sys, is expressed as
  a the parallel, ‖, (distributed) composition of the behaviours of all vehicles,
  b with the parallel composition of the parallel (likewise distributed) composition of the behaviours of all entry gates,
  c with the parallel composition of the parallel (likewise distributed) composition of the behaviours of all exit gates,
  d with the parallel composition of the behaviour of the road-pricing calculator,

value
392 sys: **Unit** → **Unit**
392 sys() ≡
392a ‖{$veh_{\mathbf{uid}\_V(v)}$(**obs_mereo**_V(v))
392a | v:V•v ∈ vs}
392b ‖ ‖{$gate_{\mathbf{uid}\_EG(eg)}$(**obs_mereo**_G(eg),"entry")
392b | eg:EG•eg ∈ egs}
392c ‖ ‖{$gate_{\mathbf{uid}\_XG(xg)}$(**obs_mereo**_G(xg),"exit")
392c | xg:XG•xg ∈ xgs}
392d ‖ $calc_{\mathbf{uid}\_C(c)}$(vis,gis)(rlf)(trm)

393 veh_{vi}: (ci:CI×gis:GI-**set**) →
393 **in** attr_TiGPos[vi] **out** v_c_ch[vi,ci] **Unit**
399 $gate_{gi}$: (ci:CI×VI-**set**×LI)×ee:EE →
399 **in** attr_entry_ch[gi,ci],attr_id_ch[gi,ci],attr_exit_ch[gi,ci]

```
399      out attr_barrier_ch[gi],g_c_ch[gi,ci]  Unit
433     calc_ci: (vis:VI-set×gis:GI-set)×VPLF→TRM→
433        in {v_c_ch[vi,ci]|vi:VI•vi ∈ vis},{g_c_ch[gi,ci]
433            | gi:GI•gi ∈ gis}  Unit
```

We consider "entry" or "exit" to be a static attribute of toll-gates. The behaviour signatures were determined as per the techniques presented in [2, Sect. 4.1.1 and 4.5.2].

Vehicle Behaviour: We refer to the vehicle behaviour, in the domain, described in Sect. 5.2's The Road Traffic System Behaviour Items 311 and Items 312, Page 154 and, projected, Page 159.

393 Instead of moving around by explicitly expressed internal non-determinism[15] vehicles move around by unstated internal non-determinism and instead receive their current position from the global positioning subsystem.

394 At each moment the vehicle receives its time-stamped global position, (τ,gpos):TiGPos,

395 from which it calculates the local position, lpos:VPos

396 which it then communicates, with its vehicle identification, (vi,(τ,lpos)), to the road pricing subsystem —

397 whereupon it resumes its vehicle behaviour.

```
value
393    veh_vi: (ci:CI×gis:GI-set) →
393          in attr_TiGPos_ch[vi] out v_c_ch[vi,ci] Unit
393    veh_vi(ci,gis) ≡
394         let (τ,gpos) = attr_TiGPos_ch[vi]? in
395         let lpos = loc_pos(gpos) in
396         v_c_ch[vi,ci] ! (vi,(τ,lpos)) ;
397         veh_vi(ci,gis) end end
393         pre vi ∈ vis
```

The *vehicle* signature has *attr_TiGPos_ch[vi]* model an external vehicle attribute and *v_c_ch[vi,ci]* the **embedded attribute sharing** [2, Sect. 4.1.1 and 4.5.2] between vehicles (their position) and the price calculator's road map. The above behaviour represents an assumption about the behaviour of vehicles. If we were to design software for the monitoring and control of vehicles then the above vehicle behaviour would have to be refined in order to serve as a proper interface requirements. The refinement would include handling concerns about the drivers' behaviour when entering, passing and exiting toll-gates, about the proper function of the GNSS equipment, and about the safe communication with the road price calculator. The above concerns would already have been addressed in a model of **domain facets** such as *human behaviour*, *technology support*, proper tele-communications *scripts*, etcetera. We refer to [4].

Gate Behaviour: The entry and the exit gates have "vehicle enter", "vehicle exit" and "timed vehicle identification" sensors. The following assumption can now be made: during the time interval between a gate's vehicle "entry" sensor having first sensed a vehicle entering that gate and that gate's "exit" sensor having last sensed that vehicle leaving that gate that gate's vehicle time and "identify" sensor registers the time when the vehicle is entering the gate and that vehicle's unique identification. We sketch the toll-gate behaviour:

398 We parameterise the toll-gate behaviour as either an entry or an exit gate.

399 Toll-gates operate autonomously and cyclically.

400 The attr_enter_ch event "triggers" the behaviour specified in formula line Item 401–403 starting with a "Raise" barrier action.

401 The time-of-passing and the identity of the passing vehicle is sensed by attr_passing_ch channel events.

402 Then the road pricing calculator is informed of time-of-passing and of the vehicle identity vi and the link li associated with the gate – and with a "Lower" barrier action.

403 And finally, after that vehicle has left the entry or exit gate the barrier is again "Lower"ered and

404 that toll-gate's behaviour is resumed.

```
type
398    EE = "enter" | "exit"
```

```
value
399    gate_gi: (ci:CI×VI-set×LI)×ee:EE →
399         in attr_enter_ch[gi],
399             attr_passing_ch[gi],
399             attr_leave_ch[gi]
399         out attr_barrier_ch[gi],
399              g_c_ch[gi,ci] Unit
399    gate_gi((ci,vis,li),ee) ≡
400         attr_enter_ch[gi] ? ;
400         attr_barrier_ch[gi] ! "Lower"
401         let (τ,vi) = attr_passing_ch[gi] ? in
401             assert vi ∈ vis
402         (attr_barrier_ch[gi] ! "Raise"
402          ‖ g_c_ch[gi,ci] ! (ee,(vi,(τ,SonL(li))))) ;
403         attr_leave_ch[gi] ? ;
403         attr_barrier_ch[gi] ! "Lower"
404         gate_gi((ci,vis,li),ee)
399         end
399         pre li ∈ lis
```

The *gate* signature's *attr_enter_ch[gi]*, *attr_passing_ch[gi]*, *attr_barrier_ch[gi]* and *attr_leave_ch[gi]* model respective **external attributes** [2, Sect. 4.1.1 and 4.5.2] (the *attr_barrier_ch[gi]* models reactive (i.e., output) attribute), while *g_c_ch[gi,ci]* models the **embedded attribute sharing** between gates

[15] We refer to Items 311b, 311c on Page 154 and 312b, 312(b)ii, 312c on Page 154

(their identification of vehicle positions) and the calculator road map. The above behaviour represents an assumption about the behaviour of toll-gates. If we were to design software for the monitoring and control of toll-gates then the above gate behaviour would have to be refined in order to serve as a proper interface requirements. The refinement would include handling concerns about the drivers' behaviour when entering, passing and exiting toll-gates, about the proper function of the entry, passing and exit sensors, about the proper function of the gate barrier (opening and closing), and about the safe communication with the road price calculator. The above concerns would already have been addressed in a model of **domain facets** such as *human behaviour*, *technology support*, proper tele-communications *scripts*, etcetera. We refer to [4] ■

We shall define the ***calc***ulator behaviour in Sect. 5.5.1 on Page 174. The reason for this deferral is that it exemplifies ***interface requirements***.

Discussion

The requirements assumptions expressed in the specifications of the vehicle and gate behaviours assume that these behave in an orderly fashion. But they seldom do ! The attr_TiGPos_ch sensor may fail. And so may the attr_enter_ch, attr_passing_ch, and attr_leave_ch sensors and the attr_barrier_ch actuator. These attributes represent ***support technology*** facets. They can fail. To secure fault tolerance one must prescribe very carefully what counter-measures are to be taken and/or the safety assumptions. We refer to [50, 130, 131]. They cover three alternative approaches to the handling of fault tolerance. Either of the approaches can be made to fit with our approach. First one can pursue our approach to where we stand now. Then we join the approaches of either of [50, 130, 131]. [130] likewise decompose the requirements prescription as is suggested here.

5.4.5 Requirements Fitting

Often a domain being described "fits" onto, is "adjacent" to, "interacts" in some areas with, another domain: ***transportation*** with ***logistics***, ***health-care*** with ***insurance***, ***banking*** with ***securities trading*** and/or ***insurance***, and so on. The issue of requirements fitting arises when two or more software development projects are based on what appears to be the same domain. The problem then is to harmonise the two or more software development projects by harmonising, if not too late, their requirements developments.

We thus assume that there are n domain requirements developments, d_{r_1}, d_{r_2}, ..., d_{r_n}, being considered, and that these pertain to the same domain — and can hence be assumed covered by a same domain description.

Definition 59 **Requirements Fitting:** By **requirements fitting** we mean a ***harmonisation*** of $n > 1$ domain requirements that have overlapping (shared) not always consistent parts and which results in n ***partial domain requirement***s', $p_{d_{r_1}}, p_{d_{r_2}}, \ldots, p_{d_{r_n}}$, and m ***shared domain requirement***s, $s_{d_{r_1}}, s_{d_{r_2}}, \ldots, s_{d_{r_m}}$, that "fit into" two or more of the partial domain requirements ■ The above definition pertains to the result of 'fitting'. The next definition pertains to the act, or process, of 'fitting'.

Definition 60 **Requirements Harmonisation:** By **requirements harmonisation** we mean a number of alternative and/or co-ordinated prescription actions, one set for each of the domain requirements actions: ***Projection, Instantiation, Determination*** and ***Extension.*** They are – we assume n separate software product requirements: ***Projection:*** If the n product requirements do not have the same projections, then identify a common projection which they all share, and refer to it as the ***common projection***. Then develop, for each of the n product requirements, if required, a ***specific projection*** of the common one. Let there be m such specific projections, $m \leq n$. ***Instantiation:*** First instantiate the common projection, if any instantiation is needed. Then for each of the m specific projections instantiate these, if required. ***Determination:*** Likewise, if required, "perform" "determination" of the possibly instantiated common projection, and similarly, if required, "perform" "determination" of the up to m possibly instantiated projections. ***Extension:*** Finally "perform extension" likewise: First, if required, of the common projection (etc.), then, if required, on the up m specific projections (etc.). These harmonization developments may possibly interact and may need to be iterated ■

By a **partial domain requirement**s we mean a domain requirements which is short of (that is, is missing) some prescription parts: text and formula ■ By a **shared domain requirement**s we mean a domain requirements ■ By **requirements fitting** m *s*hared *d*omain *r*equirements texts, $sdrs$, into n partial domain requirements we mean that there is for each *p*artial *d*omain *r*equirements, pdr_i, an identified, non-empty *s*ubset of $sdrs$ (could be all of $sdrs$), $ssdrs_i$, such that textually conjoining $ssdrs_i$ to pdr_i, i.e., $ssdrs_i \oplus pdr_i$ can be claimed to yield the "original" d_{r_i}, that is, $\mathscr{M}(ssdrs_i \oplus pdr_i) \subseteq \mathscr{M}(d_{r_i})$, where $\mathscr{M}$ is a suitable meaning function over prescriptions ■

5.4.6 Discussion

Facet-oriented Fittings: An altogether different way of looking at domain requirements may be achieved when also considering domain facets — not covered in neither the example of Sect. 5.2 nor in this section (i.e., Sect. 5.4) nor in the following two sections. We refer to [4].

Example 5.11. . **Domain Requirements — Fitting**: Example 5.10 hints at three possible sets of interface requirements: (i) for a road pricing [sub-]system, as will be illustrated in Sect. 5.5.1; (ii) for a vehicle monitoring and control [sub-]system, and (iii) for a toll-gate monitoring and control [sub-]system. The vehicle monitoring and control [sub-]system would focus on implementing the vehicle behaviour, see Items 393- 397 on Page 169. The toll-gate monitoring and control [sub-]system would focus on implementing the calculator behaviour, see Items 399- 404 on Page 169. The fitting amounts to (a) making precise the (narrative and formal) texts that are specific to each of of the three (i–iii) separate sub-system requirements are kept separate; (b) ensuring that (meaning-wise) shared texts that have different names for (meaning-wise) identical entities have these names renamed appropriately; (c) that these texts are subject to commensurate and ameliorated further requirements development; etcetera ■

5.5 Interface and Derived Requirements

We remind the reader that **interface requirements** can be expressed only using terms from both the domain and the machine ■ Users are not part of the machine. So no reference can be made to users, such as *"the system must be user friendly"*, and the like ![16] By **interface requirements** we [also] mean *requirements prescriptions which refines and extends the domain requirements by considering those requirements of the domain requirements whose endurants (parts, materials) and perdurants (actions, events and behaviours) are* **"shared"** *between the domain and the machine (being requirements prescribed)* ■ The two ***interface requirements*** definitions above go hand–in–hand, i.e., complement one-another.

By **derived requirements** we mean *requirements prescriptions which are expressed in terms of the machine concepts and facilities introduced by the emerging requirements* ■

5.5.1 Interface Requirements

Shared Phenomena

By **sharing** we mean (a) that ***some or all properties*** of an **endurant** is represented both in the domain and "inside" the machine, and that their machine representation must at suitable times reflect their state in the domain; and/or (b) that an **action** requires a sequence of several "on-line" interactions between the machine (being requirements prescribed) and the domain, usually a person or another machine; and/or (c) that an **event** arises either in the domain, that is, in the environment of the machine, or in the machine, and need be communicated to the machine, respectively to the environment; and/or (d) that a **behaviour** is manifested

[16] So how do we cope with the statement: *"the system must be user friendly"* ? We refer to Sect. 5.5.3 on Page 178 for a discussion of this issue.

both by actions and events of the domain and by actions and events of the machine ■ So a systematic reading of the domain requirements shall result in an identification of all shared endurants, parts, materials and components; and perdurants actions, events and behaviours. Each such shared phenomenon shall then be individually dealt with: **endurant sharing** shall lead to interface requirements for data initialisation and refreshment as well as for access to endurant attributes; **action sharing** shall lead to interface requirements for interactive dialogues between the machine and its environment; **event sharing** shall lead to interface requirements for how such event are communicated between the environment of the machine and the machine; and **behaviour sharing** shall lead to interface requirements for action and event dialogues between the machine and its environment.

Environment–Machine Interface:

Domain requirements extension, Sect. 5.4.4, usually introduce new endurants into (i.e., 'extend' the) domain. Some of these endurants may become elements of the domain requirements. Others are to be projected "away". Those that are let into the domain requirements either have their endurants represented, somehow, also in the machine, or have (some of) their properties, usually some attributes, accessed by the machine. Similarly for perdurants. Usually the machine representation of shared perdurants access (some of) their properties, usually some attributes. The interface requirements must spell out which domain extensions are shared. Thus domain extensions may necessitate a review of domain projection, instantiations and determination. In general, there may be several of the projection–eliminated parts (etc.) whose dynamic attributes need be accessed in the usual way, i.e., by means of attr_XYZ_ch channel communications (where XYZ is a projection–eliminated part attribute).

Example 5.12. . **Interface Requirements — Projected Extensions**: We refer to Fig. 5.2 on Page 166.We do not represent the GNSS system in the machine: only its "effect": the ability to record global positions by accessing the GNSS attribute (channel):

channel
352 {attr_TiGPos_ch[vi]|vi:VI•vi ∈ xtr_VIs(vs)}: TiGPos

And we do not really represent the gate nor its sensors and actuator in the machine. But we do give an idealised description of the gate behaviour, see Items 399–404 Instead we represent their dynamic gate attributes:

- (362) the vehicle entry sensors (leftmost ■s),
- (362) the vehicle identity sensor (center ■), and
- (363) the vehicle exit sensors (rightmost ■s)

by channels — we refer to Example 5.10 (Sect. 5.5.1, Page 166):

channel
362 {attr_entry_ch[gi]|gi:GI•xtr_eGIds(trn)} ″enter″
362 {attr_exit_ch[gi]|gi:GI•xtr_xGIds(trn)} ″exit″
363 {attr_identity_ch[gi]|gi:GI•xtr_GIds(trn)} TIVI ■

Shared Endurants

Example 5.13. . **Interface Requirements. Shared Endurants**: The main shared endurants are the vehicles, the net (hubs, links, toll-gates) and the price calculator. As domain endurants hubs and links undergo changes, all the time, with respect to the values of several attributes: **length, geodetic information, names, wear and tear** (where-ever applicable), **last/next scheduled maintenance** (where-ever applicable), **state** and **state space**, and many others. Similarly for vehicles: their position, velocity and acceleration, and many other attributes. We then come up with something like hubs and links are to be represented as tuples of relations; each net will be represented by a pair of relations a hubs relation and a links relation; each hub and each link may or will be represented by several tuples; etcetera. In this database modeling effort it must be secured that "standard" operations on nets, hubs and links can be supported by the chosen relational database system

Data Initialisation:

In general, one must prescribe data initialisation, that is provision for an interactive user interface dialogue with a set of proper display screens, one for establishing net, hub or link attributes names and their types, and, for example, two for the input of hub and link attribute values. Interaction prompts may be prescribed: next input, on-line vetting and display of evolving net, etc. These and many other aspects may therefore need prescriptions.

Example 5.14. . **Interface Requirements. Shared Endurant Initialisation**: The domain is that of the road net, n:N. By 'shared road net initialisation' we mean the "ab initio" establishment, "from scratch", of a data base recording the properties of all links, l:L, and hubs, h:H, their unique identifications, **uid_L**(l) and **uid_H**(h), their mereologies, **obs_mereo_L**(l) and **obs_mereo_H**(h), the initial values of all their static and programmable attributes and the access values, that is, channel designations for all other attribute categories.

405 There are r_l and r_h "recorders" recording link, respectively hub properties – with each recorder having a unique identity.
406 Each recorder is charged with the recording of a set of links or a set of hubs according to some partitioning of all such.
407 The recorders inform a central data base, net_db, of their recordings (ri,hol,(u$_j$,m$_j$,attrs$_j$)) where
408 ri is the identity of the recorder,
409 hol is either a `hub` or a `link` literal,
410 u$_j$ = **uid_L**(l) or **uid_H**(h) for some link or hub,
411 m$_j$ = **obs_mereo_L**(l) or **obs_mereo_H**(h) for that link or hub and
412 attrs$_j$ are *attributes* for that link or hub — where *attributes* is a function which "records" all respective static and dynamic attributes (left undefined).

```
type
405  RI
value
405  rl,rh:NAT axiom rl>0 ∧ rh>0
type
407  M = RI×"link"×LNK | RI×"hub"×HUB
407  LNK = LI × HI-set × LATTRS
407  HUB = HI × LI-set × HATTRS
```

```
value
406  partitioning: L-set→Nat→(L-set)*
406                  | H-set→Nat→(H-set)*
406  partitioning(s)(r) as sl
406  post: len sl = r ∧ ∪ elems sl = s
406         ∧ ∀ si,sj:(L-set|H-set) •
406           si≠{}∧sj≠{}∧{si,sj}⊆elems ss⇒si ∩ sj={}
```

413 The $r_l + r_h$ recorder behaviours interact with the one net_db behaviour

```
channel
413  r_db: RI×(LNK|HUB)
```

```
value
413  link_rec: RI → L-set → out r_db Unit
413  hub_rec: RI → H-set → out r_db Unit
413  net_db: Unit → in r_db Unit
```

414 The data base behaviour, net_db, offers to receive messages from the link and hub recorders.
415 The data base behaviour, net_db, deposits these messages in respective variables.
416 Initially there is a net, $n : N$,
417 from which is observed its links and hubs.
418 These sets are partitioned into r_l, respectively r_h length lists of non-empty links and hubs.
419 The ab-initio data initialisation behaviour, ab_initio_data, is then the parallel composition of link recorder, hub recorder and data base behaviours with link and hub recorder being allotted appropriate link, respectively hub sets.
420 We construct, for technical reasons, as the reader will soon see, disjoint lists of link, respectively hub recorder identities.

```
value
414  net_db:
variable
415  lnk_db: (RI×LNK)-set
415  hub_db: (RI×HUB)-set
value
416  n:N
417  ls:L-set = obs_Ls(obs_LS(n))
417  hs:H-set = obs_Hs(obs_HS(n))
```

```
418  lsl:(L-set)* = partitioning(ls)(rl)
418  lhl:(H-set)* = partitioning(hs)(rh)
420  rill:RI* axiom len rill = rl = card elems rill
420  rihl:RI* axiom len rihl = rh = card elems rihl
419  ab_initio_data: Unit → Unit
419  ab_initio_data() ≡
419    ∥ {lnk_rec(rill[i])(lsl[i])|i:Nat•1≤i≤rl} ∥
419    ∥ {hub_rec(rihl[i])(lhl[i])|i:Nat•1≤i≤rh}
419    ∥ net_db()
```

421 The link and the hub recorders are near-identical behaviours.
422 They both revolve around an imperatively stated **for all ... do ... end**. The selected link (or hub) is inspected and the "data" for the data base is prepared from
423 the unique identifier,
424 the mereology, and
425 the attributes.
426 These "data" are sent, as a message, prefixed the senders identity, to the data base behaviour.
427 We presently leave the ... unexplained.

```
value
413   link_rec: RI → L-set → Unit
421   link_rec(ri,ls) ≡
422     for ∀ l:L•l ∈ ls do uid_L(l)
423       let lnk = (uid_L(l),
424                  obs_mereo_L(l),
425                  attributes(l)) in
426       rdb ! (ri,"link",lnk);
427       ... end
422   end
```

```
413   hub_rec: RI × H-set → Unit
421   hub_rec(ri,hs) ≡
422     for ∀ h:H•h ∈ hs do uid_H(h)
423       let hub = (uid_L(h),
424                  obs_mereo_H(h),
425                  attributes(h)) in
426       rdb ! (ri,"hub",hub);
427       ... end
422   end
```

428 The net_db data base behaviour revolves around a seemingly "never-ending" cyclic process.
429 Each cycle "starts" with acceptance of some,
430 either link or hub data.
431 If link data then it is deposited in the link data base,
432 if hub data then it is deposited in the hub data base.

```
value
428   net_db() ≡
429     let (ri,hol,data) = r_db ? in
430     case hol of
431       "link" → ... ;
431         lnk_db:=lnk_db∪(ri,data),
432       "hub"  → ... ;
432         hub_db:=hub_db∪(ri,data)
430     end end ;
428′    ... ;
428     net_db()
```

The above model is an idealisation. It assumes that the link and hub data represent a well-formed net. Included in this well-formedness are the following issues: (a) that all link or hub identifiers are communicated exactly once, (b) that all mereologies refer to defined parts, and (c) that all attribute values lie within an appropriate value range. If we were to cope with possible recording errors then we could, for example, extend the model as follows: (i) when a link or a hub recorder has completed its recording then it increments an initially zero counter (say at formula Item 427); (ii) before the net data base recycles it tests whether all recording sessions has ended and then proceeds to check the data base for well-formedness issues (a–b–c) (say at formula Item 428′) ■

The above example illustrates the 'interface' phenomenon: In the formulas, for example, we show both manifest domain entities, viz., n, l, h etc., and abstract (required) software objects, viz., $(ui, me, attrs)$.

Data Refreshment:

One must also prescribe data refreshment: an interactive user interface dialogue with a set of proper display screens one for selecting the updating of net, of hub or of link attribute names and their types and, for example, two for the respective update of hub and link attribute values. Interaction-prompts may be prescribed: next update, on-line vetting and display of revised net, etc. These and many other aspects may therefore need prescriptions.

Shared Perdurants

We can expect that for every part in the domain that is shared with the machine and for which there is a corresponding behaviour of the domain there might be a corresponding process of the machine. If a projected, instantiated, 'determinated' and possibly extended domain part is dynamic, then it is definitely a candidate for being shared and having an associated machine process. We now illustrate the concept of shared perdurants via the domain requirements extension example of Sect. 5.4.4, i.e. Example 5.10 Pages 165–170.

Example 5.15. . **Interface Requirements — Shared Behaviours**: **Road Pricing Calculator Behaviour:**

433 The road-pricing calculator alternates between offering to accept communication from
434 either any vehicle
435 or any toll-gate.

```
433   calc: ci:CI×(vis:VI-set×gis:GI-set)→RLF→TRM→
434       in {v_c_ch[ci,vi]|vi:VI•vi ∈ vis},
435          {g_c_ch[ci,gi]|gi:GI•gi ∈ gis}  Unit
433   calc(ci,(vis,gis))(rlf)(trm) ≡
434       react_to_vehicles(ci,(vis,gis))(rlf)(trm)
433    ⌷
435       react_to_gates(ci,(vis,gis))(rlf)(trm)
433       pre ci = ci_ℰ ∧ vis = vis_ℰ ∧ gis = gis_ℰ
```

The *calc*ulator signature's *v_c_ch*[*ci,vi*] and *g_c_ch*[*ci,gi*] model the ***embedded attribute sharing*** between vehicles (their position), respectively gates (their vehicle identification) and the calculator road map [2, Sect. 4.1.1 and 4.5.2].

436 If the communication is from a vehicle inside the toll-road net
437 then its toll-road net position, vp, is found from the road location function, rlf,
438 and the calculator resumes its work with the traffic map, trm, suitably updated,
439 otherwise the calculator resumes its work with no changes.

```
434 react_to_vehicles(ci,(vis,gis),vplf)(trm) ≡
434   let (vi,(τ,lpos))=⌷{v_c_ch[ci,vi]?|vi:VI•vi∈ vis}
434   in
436   if vi ∈ dom trm
437     then
437       let vp = vplf(lpos) in
438       calc(ci,(vis,gis),vplf)
438            (trm†[vi↦trm⌢⟨(τ,vp)⟩])
438       end
439     else calc(ci,(vis,gis),vplf)(trm) end end
```

440 If the communication is from a gate,
441 then that gate is either an entry gate or an exit gate;
442 if it is an entry gate
443 then the calculator resumes its work with the vehicle (that passed the entry gate) now recorded, afresh, in the traffic map, trm.
444 Else it is an exit gate and
445 the calculator concludes that the vehicle has ended its to-be-paid-for journey inside the toll-road net, and hence to be billed;
446 then the calculator resumes its work with the vehicle now removed from the traffic map, trm.

```
435   react_to_gates(ci,(vis,gis),vplf)(trm) ≡
435     let (ee,(τ,(vi,li))) =
435         ⌷{g_c_ch[ci,gi]?|gi:GI•gi∈ gis} in
441     case ee of
442       "Enter" →
443          calc(ci,(vis,gis),vplf)
443               (trm∪[vi↦⟨(τ,SonL(li))⟩]),
444        "Exit"  →
445          billing(vi,trm(vi)⌢⟨(τ,SonL(li))⟩);
446          calc(ci,(vis,gis),vplf)(trm\{vi})
435     end end
```

The above behaviour is the one for which we are to design software ∎

5.5.2 Derived Requirements

Definition 61 Derived Perdurant: By a **derived perdurant** we shall understand a perdurant which is not shared with the domain, but which focus on exploiting facilities of the software or hardware of the machine ∎

"Exploiting facilities of the software", to us, means that requirements, imply the presence, in the machine, of concepts (i.e., hardware and/or software), and that it is these concepts that the **derived requirements** "rely" on. We illustrate all three forms of perdurant extensions: derived actions, derived events and derived behaviours.

Derived Actions

Definition 62 Derived Action: By a **derived action** we shall understand (a) a conceptual action (b) that calculates a usually non-Boolean valued property from, and possibly changes to (c) a machine behaviour state (d) as instigated by some actor ∎

Example 5.16. . **Domain Requirements. Derived Action: Tracing Vehicles**: The example is based on the *Road Pricing Calculator Behaviour* of Example 5.15 on the facing page. The "external" actor, i.e., a user of the *Road Pricing Calculator* system wishes to trace specific vehicles "cruising" the toll-road. That user (a

Road Pricing Calculator staff), issues a command to the *Road Pricing Calculator* system, with the identity of a vehicle not already being traced. As a result the *Road Pricing Calculator* system augments a possibly void trace of the timed toll-road positions of vehicles. We augment the definition of the *cal*culator definition Items 433–446, Pages 175–175.

447 Traces are modeled by a pair of dynamic attributes:
  a as a programmable attribute, *tra:TRA*, of the set of identifiers of vehicles being traced, and
  b as a reactive attribute, *vdu:VDU*[17], that maps vehicle identifiers into time-stamped sequences of simple vehicle positions, i.e., as a subset of the *trm:TRM* programmable attribute.

448 The actor-to-calculator *begin* or *end* trace command, *cmd:Cmd*, is modeled as an autonomous dynamic attribute of the *cal*culator.

449 The *cal*culator signature is furthermore augmented with the three attributes mentioned above.

450 The occurrence and handling of an actor trace command is modeled as a non-deterministic external choice and a *react_to_trace_cmd* behaviour.

451 The reactive attribute value (`attr_vdu_ch?`) is that subset of the traffic map (*trm*) which records just the time-stamped sequences of simple vehicle positions being traced (*tra*).

type

```
447a   TRA = VI-set
447b   VDU = TRM
448      Cmd = BTr | ETr
448      BTr :: VI
448      ETr :: VI
```

value

```
449      calc: ci:CI×(vis:VI-set×gis:GI-set)
449            → RLF → TRM → TRA
434,435        in {v_c_ch[ci,vi]|vi:VI•vi ∈ vis},
434,435              {g_c_ch[ci,gi]|gi:GI•gi ∈ gis},
450,451              attr_cmd_ch,attr_vdu_ch  Unit
433      calc(ci,(vis,gis))(rlf)(trm)(tra) ≡
434            react_to_vehicles(ci,(vis,gis),)(rlf)(trm)(tra)
435        ⌷ react_to_gates(ci,(vis,gis))(rlf)(trm)(tra)
450        ⌷ react_to_trace_cmd(ci,(vis,gis))(rlf)(trm)(tra)
433            pre ci = ci_ℰ ∧ vis = vis_ℰ ∧ gis = gis_ℰ
451            axiom □ attr_vdu_ch[ci]? = trm|tra
```

The *450,451 attr_cmd_ch,attr_vdu_ch* of the *cal*culator signature models the *cal*culator's external *command* and *visual display unit* attributes.

452 The *react_to_trace_cmd* alternative behaviour is either a "Begin" or an "End" request which identifies the affected vehicle.

453 If it is a "Begin" request

454 and the identified vehicle is already being traced then we do not prescribe what to do!

455 Else we resume the calculator behaviour, now recording that vehicle as being traced.

456 If it is an "End" request

457 and the identified vehicle is already being traced then we do not prescribe what to do!

458 Else we resume the calculator behaviour, now recording that vehicle as no longer being traced.

```
452            react_to_trace_cmd(ci,(vis,gis))(vplf)(trm)(tra) ≡
452                case attr_cmd_ch[ci]? of
453,454,455          mkBTr(vi) → if vi ∈ tra then chaos else calc(ci,(vis,gis))(vplf)(trm)(tra ∪ {vi}) end
456,457,458          mkETr(vi) → if vi ∉ tra then chaos else calc(ci,(vis,gis))(vplf)(trm)(tra\{vi}) end
452                end
```

The above behaviour, Items 433–458, is the one for which we are to design software ■

Example 5.16 exemplifies an action requirement as per definition 62: (a) the action is conceptual, it has no physical counterpart in the domain; (b) it calculates (451) a visual display (vdu); (c) the vdu value is based on a conceptual notion of traffic road maps (trm), an element of the calculator state; (d) the calculation is triggered by an actor (`attr_cmd_ch`).

Derived Events

Definition 63 Derived Event: By a **derived event** we shall understand (a) a conceptual event, (b) that calculates a property or some non-Boolean value (c) from a machine behaviour state change ■

[17] VDU: visual display unit

Example 5.17. . **Domain Requirements. Derived Event: Current Maximum Flow**: The example is based on the *Road Pricing Calculator Behaviour* of Examples 5.16 and 5.15 on Page 174. By “the current maximum flow” we understand a time-stamped natural number, the number representing the highest number of vehicles which at the time-stamped moment cruised or now cruises around the toll-road net. We augment the definition of the *cal*culator definition Items 433–458, Pages 175–176.

459 We augment the *cal*culator signature with
460 a time-stamped natural number valued dynamic programmable attribute, *(t:𝕋,max:Max)*.
461 Whenever a vehicle enters the toll-road net, through one of its [entry] gates,
 a it is checked whether the resulting number of vehicles recorded in the *road traffic map* is higher than the hitherto *max*imum recorded number.
 b If so, that programmable attribute has its number element “upped” by one.
 c Otherwise not.
462 No changes are to be made to the react_to_gates behaviour (Items 435–446 Page 175) when a vehicle exits the toll-road net.

```
type
460      MAX = 𝕋 × NAT
value
449,459   calc: ci:CI×(vis:VI-set×gis:GI-set) → RLF → TRM → TRA → MAX
434,435        in {v_c_ch[ci,vi]|vi:VI•vi ∈ vis}, {g_c_ch[ci,gi]|gi:GI•gi ∈ gis}, attr_cmd_ch,attr_vdu_ch  Unit
435      react_to_gates(ci,(vis,gis))(vplf)(trm)(tra)(t,m) ≡
435         let (ee,(τ,(vi,li))) = ⌈⌉{g_c_ch[ci,gi]|gi:GI•gi∈ gis} in
441         case ee of
461            "Enter" →
461               calc(ci,(vis,gis))(vplf)(trm∪[vi↦⟨(τ,SonL(li))⟩])(tra)(τ,if card dom trm=m then m+1 else m end),
462            "Exit" →
462               billing(vi,trm(vi)⌢⟨(τ,SonL(li))⟩); calc(ci,(vis,gis))(vplf)(trm\{vi})(tra)(t,m) end
441         end
```

The above behaviour, Items 433 on Page 175 through 461c, is the one for which we are to design software ■

Example 5.17 exemplifies a derived event requirement as per Definition 63: (a) the event is conceptual, it has no physical counterpart in the domain; (b) it calculates (461b) the max value based on a conceptual notion of traffic road maps (trm), (c) which is an element of the calculator state.

No Derived Behaviours

There are no derived behaviours. The reason is as follows. Behaviours are associated with parts. A possibly ‘derived behaviour’ would entail the introduction of an ‘associated’ part. And if such a part made sense it should – in all likelihood – already have been either a proper domain part or become a domain extension. If the domain–to-requirements engineer insist on modeling some interface requirements as a process then we consider that a technical matter, a choice of abstraction.

5.5.3 Discussion

Derived Requirements

Formulation of derived actions or derived events usually involves technical terms not only from the domain but typically from such conceptual ‘domains’ as mathematics, economics, engineering or their visualisation. Derived requirements may, for some requirements developments, constitute “sizable” requirements compared to “all the other” requirements. For their analysis and prescription it makes good sense to first having developed “the other” requirements: domain, interface and machine requirements. The treatment of the present chapter does not offer special techniques and tools for the conception, &c., of derived requirements. Instead we refer to the seminal works of [176, 114, 113].

Introspective Requirements

Humans, including human users are, in this chapter, considered to never be part of the domain for which a requirements prescription is being developed. If it is necessary to involve humans in the domain description or the requirements prescription then their prescription is to reflect assumptions upon whose behaviour the machine rely. It is therefore that we, above, have stated, in passing, that we cannot accept requirements of the kind: *"the machine must be user friendly"*, because, in reality, it means *"the user must rely upon the machine being 'friendly'"* whatever that may mean. We are not requirements prescribing humans, nor their sentiments !

5.6 Machine Requirements

Other than listing a sizable number of ***machine requirement facet***s we shall not cover machine requirements in this chapter. The reason for this is as follows. We find, cf. [51, Sect. 19.6], that when the individual machine requirements are expressed then references to domain phenomena are, in fact, abstract references, that is, they do not refer to the semantics of what they name. Hence ***machine requirement***s "fall" outside the scope of this chapter — with that scope being *"derivation" of requirements from domain specifications* with emphasis on derivation techniques that relate to various aspects of the domain.

(A) There are the ***technology requirements*** of (1) ***performance*** and (2) ***dependability***. Within ***dependability requirements*** there are (a) ***accessibility***, (b) ***availability***, (c) ***integrity***, (d) ***reliability***, (e) ***safety***, (f) ***security*** and (g) ***robustness*** requirements. A proper treatment of dependability requirements need a careful definition of such terms as *failure, error, fault,* and, from these *dependability.* (B) And there are the ***development requirements*** of (i) ***process***, (ii) ***maintenance***, (iii) ***platform***, (iv) ***management*** and (v) ***documentation*** requirements. Within ***maintenance requirements*** there are (ii.1) ***adaptive***, (ii.2) ***corrective***, (ii.3) ***perfective***, (ii.4) ***preventive***, and (ii.5) ***extensional*** requirements. Within ***platform requirements*** there are (iii.1) ***development***, (iii.2) ***execution***, (iii.3) ***maintenance***, and (iii.4) ***demonstration*** platform requirements. We refer to [51, Sect. 19.6] for an early treatment of ***machine requirements***.

5.7 Conclusion

Conventional requirements engineering considers the domain only rather implicitly. Requirements gathering ('acquisition') is not structured by any pre-existing knowledge of the domain, instead it is "structured" by a number of relevant techniques and tools [103, 113, 104] which, when applied, "fragment-by-fragment" "discovers" such elements of the domain that are immediately relevant to the requirements. The present chapter turns this requirements prescription process "up-side-down". Now the process is guided ("steered", "controlled") almost exclusively by the domain description which is assumed to be existing before the requirements development starts. In conventional requirements engineering many of the relevant techniques and tools can be said to take into account *sociological* and *psychological* facets of gathering the requirements and *linguistic* facets of expressing these requirements. That is, the focus is rather much on the *process.* In the present chapter's requirements "derivation" from domain descriptions the focus is all the time on the descriptions and prescriptions, in particular on their formal expressions and the "transformation" of these. That is (descriptions and) prescriptions are considered formal, *mathematical* objects. That is, the focus is rather much on the *objects.*

• • •

We conclude by briefly reviewing what has been achieved, present shortcomings & possible research challenges, and a few words on relations to "classical requirements engineering".

5.7.1 What has been Achieved ?

We have shown how to systematically "derive" initial aspects of requirements prescriptions from domain descriptions. The stages[18] and steps[19] of this "derivation"[20] are new. We claim that current requirements engineering approaches, although they may refer to a or the 'domain', are not really 'serious' about this: they do not describe the domain, and they do not base their techniques and tools on a reasoned understanding of the domain. In contrast we have identified, we claim, a logically motivated decomposition of requirements into three phases, cf. Footnote 18., of domain requirements into five steps, cf. Footnote 19., and of interface requirements, based on a concept of shared entities, tentatively into (α) shared endurants, (β) shared actions, (γ) shared events, and (δ) shared behaviours (with more research into the (α-δ) techniques needed).

5.7.2 Present Shortcomings and Research Challenges

We see three shortcomings: (1) The "derivation" techniques have yet to consider "extracting" requirements from ***domain facet description***s. Only by including ***domain facet description***s can we, in "deriving" ***requirements prescription***s, include failures of, for example, support technologies and humans, in the design of fault-tolerant software. (2) The "derivation" principles, techniques and tools should be given a formal treatment. (3) There is a serious need for relating the approach of the present chapter to that of the seminal text book of [113, Axel van Lamsweerde]. [113] is not being "replaced" by the present work. It tackles a different set of problems. We refer to the penultimate paragraph before the Acknowledgment closing.

5.7.3 Comparison to "Classical" Requirements Engineering:

Except for a few, represented by two, we are not going to compare the contributions of the present chapter with published journal or conference papers on the subject of requirements engineering. The reason for this is the following. The present chapter, rather completely, we claim, reformulates requirements engineering, giving it a 'foundation', in ***domain engineering***, and then developing ***requirements engineering*** from there, viewing requirements prescriptions as "derived" from domain descriptions. We do not see any of the papers, except those reviewed below [130] and [176], referring in any technical sense to 'domains' such as we understand them.

[130, Deriving Specifications for Systems That Are Connected to the Physical World]

The paper that comes closest to the present chapter in its serious treatment of the [problem] domain as a precursor for requirements development is that of [130, Jones, Hayes & Jackson]. A purpose of [130] (Sect. 1.1, Page 367, last §) is to see "how little can one say" (about the problem domain) when expressing assumptions about requirements. This is seen by [130] (earlier in the same paragraph) as in contrast to our form of domain modeling. [130] reveals assumptions about the domain when expressing ***rely guarantee***s in tight conjunction with expressing the ***guarantee*** (requirements). That is, analysing and expressing requirements, in [130], goes hand-in-hand with analysing and expressing fragments of the domain. The current chapter takes the view that since, as demonstrated in [2], it is possible to model sizable aspects of domains, then it would be interesting to study how one might "derive" — and which — requirements prescriptions from domain descriptions; and having demonstrated that (i.e., the "how much can be derived") it seems of scientific interest to see how that new start (i.e., starting with a priori given domain descriptions or starting with first developing domain descriptions) can be combined with existing approaches, such as [130]. We do appreciate the "tight coupling" of rely–guarantees of [130]. But perhaps one looses understanding the domain due to its fragmented presentation. If the 'relies' are not outright, i.e., textually directly expressed in our domain descriptions, then they obviously must be provable properties of what our domain descriptions

[18] (a) domain, (b) interface and (c) machine requirements

[19] For domain requirements: (i) projection, (ii) instantiation, (iii) determination, (iv) extension and (v) fitting; etc.

[20] We use double quotation marks: "..." to indicate that the derivation is not automatable.

express. Our, i.e., the present, chapter — with its background in Chapter 1, [2, Sect. 4.7] — develops — with a background in [41, M.A. Jackson] — a set of principles and techniques for the access of attributes. The "discovery" of the `CM` and `SG` channels of [130] and of the type of their messages, seems, compared to our approach, less systematic. Also, it is not clear how the [130] case study "scales" up to a larger domain. The *sluice gate* of [130] is but part of a large ('irrigation') system of reservoirs (water sources), canals, sluice gates and the fields (water sinks) to be irrigated. We obviously would delineate such a larger system and research & develop an appropriate, both informal, a narrative, and formal domain description for such a class of irrigation systems based on assumptions of precipitation and evaporation. Then the users' requirements, in [130], that the sluice gate, over suitable time intervals, is open 20% of the time and otherwise closed, could now be expressed more pertinently, in terms of the fields being appropriately irrigated.

[176, Goal-directed Requirements Acquisition]

outlines an approach to requirements acquisition that starts with fragments of domain description. The domain description is captured in terms of predicates over *actors*, *actions*, *events*, *entities* and (their) *relations*. Our approach to domain modeling differs from that of [176] as follows: Agents, actions, entities and relations are, in [176], seen as specialisations of a concept of *objects*. The nearest analogy to relations, in Chapter 1 [2], as well as in this chapter, is the signatures of perdurants. Our 'agents' relate to discrete endurants, i.e., parts, and are the behaviours that evolve around these parts: one agent per part ! [176] otherwise include describing parts, relations between parts, actions and events much like Chapter 1 [2] and this chapter does. [176] then introduces a notion of ***goal***. *A* **goal**, *in [176], is defined as ″a nonoperational objective to be achieved by the desired system. Nonoperational means that the objective is not formulated in terms of objects and actions "available" to some agent of the system* ■[21]″ [176] then goes on to exemplify goals. In this, the current chapter, we are not considering *goal*s, also a major theme of [113].[22] Typically the expression of goals of [176, 113], are "within" computer & computing science and involve the use of temporal logic.[23] ″***Constraint****s are* ***operational objective****s to be achieved by the desired (i.e., required) system, ..., formulated in terms of objects and actions "available" to some agents of the system. ... Goals are made operational through constraints. ... A constraint operationalising a goal amounts to some abstract "implementation" of this goal*″ [176]. [176] then goes on to express goals and constraints operationalising these. [176] is a fascinating paper[24] as it shows how to build goals and constraints on domain description fragments.

•••

These papers, [130] and [176], as well as the current chapter, together with such seminal monographs as [50, 131, 113], clearly shows that there are many diverse ways in which to achieve precise requirements prescriptions. The [50, 131] monographs primarily study the $\mathscr{D},\mathscr{S} \models \mathscr{R}$ specification and proof techniques from the point of view of the specific tools of their specification languages[25]. Physics, as a natural science, and its many engineering 'renditions', are manifested in many separate sub-fields: Electricity, mechanics, statics, fluid dynamics — each with further sub-fields. It seems, to this author, that there is a need to study the [50, 131, 113] approaches and the approach taken in this chapter in the light of identifying sub-fields of requirements engineering. The title of the present chapter suggests one such sub-field.

[21] We have reservations about this definition: Firstly, it is expressed in terms of some of the "things" it is not ! (To us, not a very useful approach.) Secondly, we can imagine goals that are indeed formulated in terms of objects and actions 'available' to some agent of the system. For example, wrt. the ongoing library examples of [176], *the system shall automate the borrowing of books*, etcetera. Thirdly, we assume that by " 'available' to some agent of the system" is meant that these agents, actions, entities, etc., are also required.

[22] An example of a goal — for the road pricing system — could be that of *shortening travel times of motorists, reducing gasoline consumption and air pollution, while recouping investments on toll-road construction*. We consider techniques for ensuring the above kind of goals "outside" the realm of computer & computing science but "inside" the realm of operations research (`OR`) — while securing that the `OR` models are commensurate with our domain models.

[23] In this chapter we do not exemplify goals, let alone the use of temporal logic. We cannot exemplify all aspects of domain description and requirements prescription, but, if we were, would then use the temporal logic of [50, `The Duration Calculus`].

[24] — that might, however, warrant a complete rewrite.

[25] The Duration Calculus [`DC`], respectively `DC, Timed Automata` and `Z`

5.8 Bibliographical Notes

I have thought about domain engineering for more than 20 years. But serious, focused writing only started to appear since [51, Part IV] — with [178, 179] being exceptions: [180] suggests a number of domain science and engineering research topics; [4] covers the concept of domain facets; [181] explores compositionality and Galois connections. [10, 182] show how to systematically, but, of course, not automatically, "derive" requirements prescriptions from domain descriptions; [183] takes the triptych software development as a basis for outlining principles for believable software management; [8, 39] presents a model for Stanisław Leśniewski's [38] concept of mereology; [184, 169] present an extensive example and is otherwise a precursor for the present chapter; [102] presents, based on the `TripTych` view of software development as ideally proceeding from domain description via requirements prescription to software design, concepts such as software demos and simulators; [185] analyses the `TripTych`, especially its domain engineering approach, with respect to [186, 187, Maslow]'s and [188, Peterson's and Seligman's]'s notions of humanity: how can computing relate to notions of humanity; the first part of [40] is a precursor for [2] with the second part of [40] presenting a first formal model of the elicitation process of analysis and description based on the prompts more definitively presented in the current chapter; and with [189] focus on domain safety criticality. The present chapter, [9], marks, for me, a high point, with Chapter 1 [2] now constituting the base introduction to domain science & engineering.

Part **III**

Some Implications for Software

6

Demos, Simulators, Monitors and Controllers

6.1 Introduction

We sketch some observations of the concepts of domain, requirements and modeling – where abstract interpretations of these models cover both a priori, a posteriori and real-time aspects of the domain as well as 1–1 (i.e., real-time), microscopic and macroscopic simulations, real-time monitoring and real-time monitoring & control of that domain. The reference frame for these concepts are domain models: carefully narrated and formally described domains. On the basis of a familiarising example[1] of a domain description, we survey more-or-less standard ideas of verifiable software developments and conjecture software product families of demos, simulators, monitors and monitors & controllers – but now these "standard ideas" are recast in the context of core requirements prescriptions being "derived" from domain descriptions.

A background setting for this chapter is the concern for (α) professionally developing the right software, i.e., software which satisfies users expectations, and (ω) software that is right: i.e., software which is correct with respect to user requirements and thus has no "bugs", no "blue screens". The present chapter must be seen on the background of a main line of experimental research around the topics of domain science & engineering and requirements engineering and their relation. For details I refer to [2, 3, 9].

"Confusing Demos":

This author has had the doubtful honour, on his many visits to computer science and software engineering laboratories around the world, to be presented, by his colleagues' aspiring PhD students, so-called demos of "systems" that they were investigating. There always was a tacit assumption, namely that the audience, i.e., me, knew, a priori, what the domain "behind" the "system" being "demo'ed" was. Certainly, if there was such an understanding, it was brutally demolished by the "demo" presentation. My questions, such as *"what are you demo'ing"* (etcetera) went unanswered. Instead, while we were waiting to see "something interesting" to be displayed on the computer screen we were witnessing frantic, sometimes failed, input of commands and data, "nervous" attempts with "mouse" clickings, etc. – before something intended was displayed. After a, usually 15 minute, grace period, it was time, luckily, to proceed to the next "demo".

Aims & Objectives:

The aims of this chapter is to present (a) some ideas about software that either "demo", simulate, monitor or monitor & control domains; (b) some ideas about "time scaling": demo and simulation time versus domain time; and (c) how these kinds of software relate. The (undoubtedly very naïve) objectives of the chapter is also to improve the kind of demo-presentations, alluded to above, so as to ensure that the basis for such demos is crystal clear from the very outset of research & development, i.e., that domains be well-described. The chapter, we think, tackles the issue of so-called 'model-oriented (or model-based) software development' from altogether different angles than usually promoted.

[1] Instead of bringing this example, as an appendix, as it was done in [12] we now refer to Sect. 1.8 (Pages 46–55) of Chapter 1.

An Exploratory Chapter:

The chapter is exploratory. There will be no theorems and therefore there will be no proofs. We are presenting what might eventually emerge into (α) a theory of domains, i.e., a domain science [180, 181, 190, 169], and (β) a software development theory of domain engineering versus requirements engineering [183, 10, 191, 184].

The chapter is not a "standard" research chapter: it does not compare its claimed achievements with corresponding or related achievements of other researchers – simply because we do not claim "achievements" which have been reasonably well formalised. But we would suggest that you might find some of the ideas of the chapter (in Sect. 6.3) worthwhile.

Structure of The Chapter:

The structure of the chapter is as follows. We refer to Sect. 1.8 (Pages 46–55) of Chapter 1. In Sect. 6.3 we then outline a series of interpretations of domain descriptions. These arise, when developed in an orderly, professional manner, from requirements prescriptions which are themselves orderly developed from the domain description[2], cf. [9].

The essence of Sect. 6.3 is (i) the (albeit informal) presentation of such tightly related notions as *demos* (Sect. 6.3.1), *simulators* (Sect. 6.3.2), *monitors* (Sect. 6.3.3) and *monitors & controllers* (Sect. 6.3.3) (these notions can be formalised), and (ii) the conjectures on a product family of domain-based software developments (Sect. 6.3.5). A notion of *script-based simulation* extends demos and is the basis for monitor and controller developments and uses. The scripts used in our examples are related to time, but one can define non-temporal scripts – so the "carrying idea" of Sect. 6.3 extends to a widest variety of software. We claim that Sect. 6.3 thus brings these new ideas: a tightly related software engineering concept of *demo-simulator-monitor-controller* machines, and an extended notion of *reference models for requirements and specifications* [119].

6.2 Domain Descriptions

By a domain description we shall mean a combined narrative, that is, precise, but informal, and a formal description of the application domain ***as it is:*** no reference to any possible requirements let alone software that is desired for that domain. Thus a requirements prescription is a likewise combined precise, but informal, narrative, and a formal prescription of what we expect from a machine (hardware + software) that is to support endurants, actions, events and behaviours of a possibly business process re-engineered application domain. Requirements expresses a domain ***as we would like it to be.***

We further refer to the literature for examples: [24, *railways* (2000)], [31, *the 'market'* (2000)], [191, *public government, IT security, hospitals* (2006) chapters 8–10], [10, *transport nets* (2008)] and [184, *pipelines* (2010)]. On the net you may find technical reports covering "larger" domain descriptions. "Older" publications on the concept of domain descriptions are [184, 169, 8, 181, 10, 180, 4] all summarised in [2, 3, 9].

Domain descriptions do not necessarily describe computable objects. They relate to the described domain in a way similar to the way in which mathematical descriptions of physical phenomena stand to "the physical world".

6.3 Interpretations

In this main section of the chapter we present a number of interpretations of rôles of domain descriptions.

[2] We do not show such orderly "derivations" but outline their basics in Sect. 6.3.4.

6.3.1 What Is a Domain-based Demo?

A *domain-based demo* is a software system which *"present"* endurants and perdurants[3]: actions, events and behaviours of a domain. The *"presentation"* abstracts these phenomena and their related concepts in various computer generated forms: visual, acoustic, etc.

Examples

There are two main examples. One was given in Sect. 1.8 (Pages 46–55) of Chapter 1. The other is summarised below. It is from our paper on "deriving requirements prescriptions from domain descriptions" [9]. The summary follows.

The domain description of Sect. 2. of [9], outlines an abstract concept of transport nets (of hubs [street intersections, train stations, harbours, airports] and links [road segments, rail tracks, shipping lanes, airlanes]), their development, traffic [of vehicles, trains, ships and aircraft], etc. We shall assume such a transport domain description below.

Endurants are, for example, presented as follows: (a) transport nets by two dimensional (2D) road, railway or air traffic maps, (b) hubs and links by highlighting parts of 2D maps and by related photos – and their unique identifiers by labeling hubs and links, (c) routes by highlighting sequences of paths (hubs and links) on a 2D map, (d) buses by photographs and by dots at hubs or on links of a 2D map, and (e) bus timetables by, well, indeed, by showing a 2D bus timetable.

Actions are, for example, presented as follows: (f) The insertion or removal of a hub or a link by showing "instantaneous" triplets of "before", "during" and "after" animation sequences. (g) The start or end of a bus ride by showing flashing animations of the appearance, respectively the flashing disappearance of a bus (dot) at the origin, respectively the destination bus stops.

Events are, for example, presented as follows: (h) A mudslide [or fire in a road tunnel, or collapse of a bridge] along a (road) link by showing an animation of part of a (road) map with an instantaneous sequence of (α) the present link , (β) a gap somewhere on the link, (γ) and the appearance of two ("symbolic") hubs "on either side of the gap". (i) The congestion of road traffic "grinding to a halt" at, for example, a hub, by showing an animation of part of a (road) map with an instantaneous sequence of the massive accumulation of vehicle dots moving (instantaneously) from two or more links into a hub.

Behaviours are, for example, presented as follows: (k) A bus tour: from its start, on time, or "thereabouts", from its bus stop of origin, via (all) intermediate stops, with or without delays or advances in times of arrivals and departures, to the bus stop of destination (ℓ) The composite behaviour of "all bus tours", meeting or missing connection times, with sporadic delays, with cancellation of some bus tours, etc. – by showing the sequence of states of all the buses on the net.

We say that behaviours ((j)–(ℓ)) are *script-based* in that they (try to) satisfy a bus timetable ((e)).

Towards a Theory of Visualisation and Acoustic Manifestation

The above examples shall serve to highlight the general problem of visualisation and acoustic manifestation. Just as we need sciences of visualising scientific data and of diagrammatic logics, so *we need more serious studies of visualisation and acoustic manifestation — so amply, but, this author thinks, inconsistently demonstrated by current uses of interactive computing media.*

6.3.2 Simulations

"Simulation is the imitation of some real thing, state of affairs, or process; the act of simulating something generally entails representing certain key characteristics or behaviours of a selected physical or abstract system" [Wikipedia] for the purposes of testing some hypotheses usually stated in terms of the model being simulated and pairs of statistical data and expected outcomes.

[3] The concepts of 'endurants' and 'perdurants' were defined in [2].

Explication of Figure 6.1

Figure 6.1 attempts to indicate four things: (i) Left top: the rounded edge rectangle labeled "The Domain" alludes to some specific domain ("out there"). (ii) Left middle: the small rounded rectangle labeled "A Domain Description" alludes to some document which narrates and formalises a description of "the domain". (iii) Left bottom: the medium sized rectangle labeled "A Domain Demo based on the Domain Description" (for short "Demo") alludes to a software system that, in some sense (to be made clear later) "simulates" "The Domain." (iv) Right: the large rectangle (a) shows a horisontal time axis which basically "divides" that large rectangle into two parts: (b) Above the time axis the **"fat"** rounded edge rectangle alludes to the time-wise behaviour, a *domain trace*, of "The Domain" (i.e., the actual, the real, domain). (c) Below the time axis there are eight **"thin"** rectangles. These are labels S1, S2, S3, S4, S5, S6, S7 and S8. (d) Each of these denote a "run", i.e., a time-stamped "execution", a *program trace*, of the "Demo". Their "relationship" to the time axis is this: their execution takes place in the real time as related to that of "The Domain" behaviour.

A *trace* (whether a domain or a program execution trace) is a time-stamped sequence of states: domain states, respectively demo, simulator, monitor and monitor & control states.

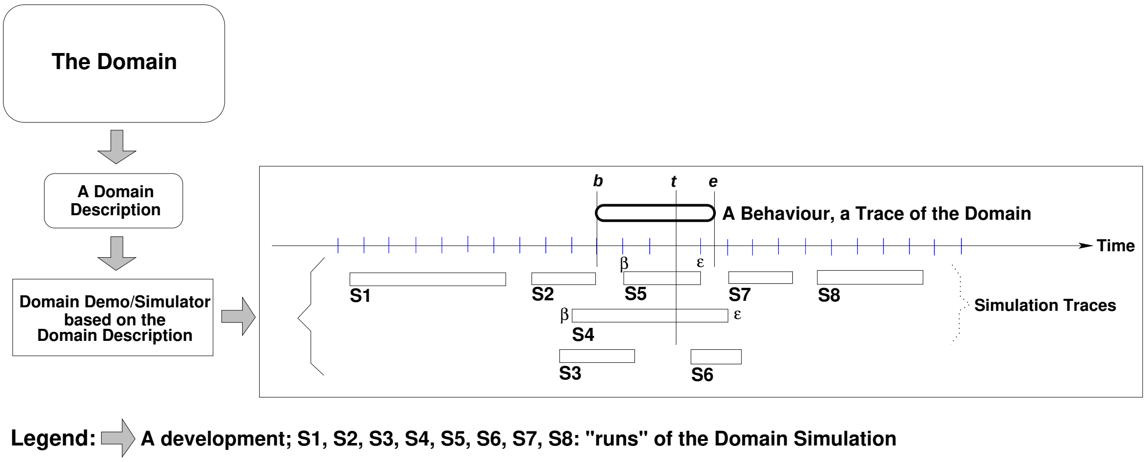


Fig. 6.1. Simulations

From Fig. 6.1 and the above explication we can conclude that "executions" S4 and S5 each share exactly one time point, t, at which "The Domain" and "The Simulation" "share" time, that is, the time-stamped execution S4 and S5 reflect a "Simulation" state which at time t should reflect (some abstraction of) "The Domain" state.

Only if the domain behaviour (i.e., trace) fully "surrounds" that of the simulation trace, or, vice-versa (cf. Fig. 6.1[S4,S5]), is there a "shared" time. Only if the 'begin' and 'end' times of the domain behaviour are identical to the 'start' and 'finish' times of the simulation trace, is there an infinity of shared 1–1 times. Only then do we speak of a real-time simulation.

In Fig 6.2 on Page 190 we show "the same" "Domain Behaviour" (three times) and a (1) simulation, a (2) monitoring and a (3) monitoring & control, all of whose 'begin/start' (b/β) and 'end/finish' (e/ε) times coincide. In such cases the "Demo/Simulation" takes place in real-time throughout the 'begin$\cdots\cdots$end' interval.

Let β and ε be the 'start' and 'finish' times of either S4 or S5. Then the relationship between t, β, ε, b and e is $\frac{t-\mathsf{b}}{\mathsf{e}-\mathsf{t}} = \frac{t-\beta}{\varepsilon-t}$ — which leads to a second degree polynomial in t which can then be solved in the usual, high school manner.

Script-based Simulation

A script-based simulation is the behaviour, i.e., an execution, of, basically, a demo which, step-by-step, follows a script: that is a prescription for highlighting endurants, actions, events and behaviours.

Script-based simulations where the script embodies a notion of time, like a bus timetable, and unlike a route, can be thought of as the execution of a demos where "chunks" of demo operations take place in accordance with "chunks"[4] of script prescriptions. The latter (i.e., the script prescriptions) can be said to represent simulated (i.e., domain) time in contrast to "actual computer" time. The actual times in which the script-based simulation takes place relate to domain times as shown in Simulations S1 to S8 in Fig. 6.1 and in Fig. 6.2(1–3). Traces Fig. 6.2(1–3) and S8 Fig. 6.1 are said to be *real-time:* there is a one-to-one mapping between computer time and domain time. S1 and S4 Fig. 6.1 are said to be *microscopic:* disjoint computer time intervals map into distinct domain times. S2, S3, S5, S6 and S7 are said to be *macroscopic:* disjoint domain time intervals map into distinct computer times.

In order to concretise the above "vague" statements let us take the example of simulating bus traffic as based on a bus timetable script. A simulation scenario could be as follows. Initially, not relating to any domain time, the simulation "demos" a net, available buses and a bus timetable. The person(s) who are requesting the simulation are asked to decide on the ratio of the domain time interval to simulation time interval. If the ratio is 1 a real-time simulation has been requested. If the ratio is less than 1 a microscopic simulation has been requested. If the ratio is larger than 1 a microscopic simulation has been requested. A chosen ratio of, say 48 to 1 means that a 24 hour bus traffic is to be simulated in 30 minutes of elapsed simulation time. Then the person(s) who are requesting the simulation are asked to decide on the starting domain time, say 6:00am, and the domain time interval of simulation, say 4 hours – in which case the simulation of bus traffic from 6am till 10am is to be shown in 5 minutes (300 seconds) of elapsed simulation time. The person(s) who are requesting the simulation are then asked to decide on the "*sampling times*" or "*time intervals*": If '*sampling times*' 6:00 am, 6:30 am, 7:00 am, 8:00 am, 9:00 am, 9:30 am and 10:00 am are chosen, then the simulation is stopped at corresponding simulation times: 0 sec., 37.5 sec., 75 sec., 150 sec., 225 sec., 262.5 sec. and 300 sec. The simulation then shows the state of selected endurants and actions at these domain times. If '*sampling time interval*' is chosen and is set to every 5 min., then the simulation shows the state of selected endurants and actions at corresponding domain times. The simulation is resumed when the person(s) who are requesting the simulation so indicates, say by a "resume" icon click. The time interval between adjacent simulation stops and resumptions contribute with 0 time to elapsed simulation time – which in this case was set to 5 minutes. Finally the requestor provides some statistical data such as numbers of potential and actual bus passengers, etc.

Then two clocks are started: a domain time clock and a simulation time clock. The simulation proceeds as driven by, in this case, the bus time table. To include "unforeseen" events, such as the wreckage of a bus (which is then unable to complete a bus tour), we allow any number of such events to be randomly scheduled. Actually scheduled events "interrupts" the "programmed" simulation and leads to thus unscheduled stops (and resumptions) where the unscheduled stop now focuses on showing the event.

The Development Arrow

The arrow, ⇨, between a pair of boxes (of Fig. 6.1 on the preceding page) denote a step of development: (i) from the domain box to the domain description box, ⇩, it denotes the development of a domain description based on studies and analyses of the domain; (ii) from the domain description box to the domain demo box, ⇩, it denotes the development of a software system — where that development assumes an intermediate requirements box which has not been show; (iii) from the domain demo box to either of a simulation traces, ⇨, it denotes the development of a simulator as the related demo software system, again depending on whichever special requirements have been put to the simulator.

6.3.3 Monitoring & Control

Figure 6.2 on the following page shows three different kinds of uses of software systems (where (2) [Monitoring] and (3) [Monitoring & Control] represent further) developments from the demo or simu-

[4] We deliberately leave the notion of chunk vague so as to allow as wide an spectrum of simulations.

lation software system mentioned in Sect. 6.3.1 and Sect. 6.3.2 on the previous page. We have added some

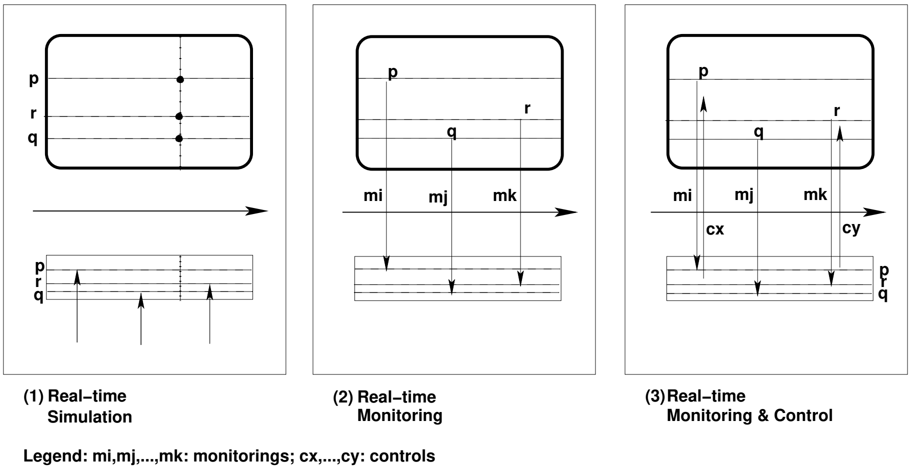


Fig. 6.2. Simulation, Monitoring and Monitoring & Control

(three) horisontal and labeled (p, q and r) lines to Fig. 6.2(1,2,3) (with respect to the traces of Fig. 6.1 on Page 188). They each denote a trace of a endurant, an action or an event, that is, they are traces of values of these phenomena or concepts. A (named) endurant value entails a description of the endurant, whither atomic ('hub', 'link', 'bus timetable') or composite ('net', 'set of hubs', etc.): of its unique identity, its mereology and a selection of its attributes. A (named) action value could, for example, be the pair of the before and after states of the action and some description of the function ('insertion of a link', 'start of a bus tour') involved in the action. A (named) event value could, for example, be a pair of the before and after states of the endurants causing, respectively being effected by the event and some description of the predicate ('mudslide', 'break-down of a bus') involved in the event. A cross section, such as designated by the vertical lines (one for the domain trace, one for the "corresponding" program trace) of Fig. 6.2(1) denotes a state: a domain, respectively a program state.

Figure 6.2(1) attempts to show a real-time demo or simulation for the chosen domain. Figure 6.2(2) purports to show the deployment of real-time software for monitoring (chosen aspects of) the chosen domain. Figure 6.2(3) purports to show the deployment of real-time software for monitoring as well as controlling (chosen aspects of) the chosen domain.

Monitoring

By *domain monitoring* we mean *"to be aware of the state of a domain"*, its endurants, actions, events and behaviour. Domain monitoring is thus a process, typically within a distributed system for collecting and storing state data. In this process "observation" points — i.e., endurants, actions and where events may occur — are identified in the domain, cf. points p, q and r of Fig. 6.2. Sensors are inserted at these points. The "downward" pointing vertical arrows of Figs. 6.2(2–3), from "the domain behaviour" to the "monitoring" and the "monitoring & control" traces express communication of what has been sensed (measured, photographed, etc.) [as directed by and] as input data (etc.) to these monitors. The monitor (being "executed") may store these "sensings" for future analysis.

Control

By *domain control* we mean *"the ability to change the value"* of endurants and the course of actions and hence behaviours, including prevention of events of the domain. Domain control is thus based on domain monitoring. Actuators are inserted in the domain "at or near" monitoring points or at points related to these, viz. points p and r of Fig. 6.2 on the preceding page(3). The "upward" pointing vertical arrows of Fig. 6.2 on the facing page(3), from the "monitoring & control" traces to the "domain behaviour" express communication, to the domain, of what has been computed by the controller as a proper control reaction in response to the monitoring.

6.3.4 Machine Development

Machines

By a *machine* we shall understand a combination of hardware and software. For demos and simulators the machine is "mostly" software with the hardware typically being graphic display units with tactile instruments. For monitors the "main" machine, besides the hardware and software of demos and simulators, additionally includes *sensors* distributed throughout the domain and the technological machine means of *communicating* monitored signals from the sensors to the "main" machine and the processing of these signals by the main machine. For monitors & controllers the machine, besides the monitor machine, further includes actuators placed in the domain and the machine means of computing and communicating control signals to the actuators.

Requirements Development

Essential parts of $\mathbb{R}$equirements to a $\mathbb{M}$achine can be systematically "derived" from a $\mathbb{D}$omain description. These essential parts are the *domain requirements* and the *interface requirements*. Domain requirements are those requirements which can be expressed, say in narrative form, by mentioning technical terms only of the domain. These technical terms cover only phenomena and concepts (endurants, actions, events and behaviours) of the domain. Some domain requirements are *projected*, *instantiated*, made more *deterministic* and *extended*[5]. We bring examples that are taken from Sect. 2. of [9], cf. Sect. 6.3.1 on Page 187 of this paper. (a) By *domain projection* we mean a sub-setting of the domain description: parts are left out which the requirements stake-holders, collaborating with the requirements engineer, decide is of no relevance to the requirements. For our example it could be that our domain description had contained models of road net attributes such as "the wear & tear" of road surfaces, the length of links, states of hubs and links (that is, [dis]allowable directions of traffic through hubs and along links), etc. Projection might then omit these attributes. (b) By *domain instantiation* we mean a specialisation of endurants, actions, events and behaviours, refining them from abstract simple entities to more concrete such, etc. For our example it could be that we only model freeways or only model road-pricing nets – or any one or more other aspects. (c) By *domain determination* we mean that of making the domain description cum domain requirements prescription less non-deterministic, i.e., more deterministic (or even the other way around !). For our example it could be that we had domain-described states of street intersections as not controlled by traffic signals – where the determination is now that of introducing an abstract notion of traffic signals which allow only certain states (of red, yellow and green). (d) By *domain extension* we basically mean that of extending the domain with phenomena and concepts that were not feasible without information technology. For our examples we could extend the domain with bus mounted GPS gadgets that record and communicate (to, say a central bus traffic computer) the more-or-less exact positions of buses – thereby enabling the observation of bus traffic. Interface requirements are those requirements which can be expressed, say in narrative form, by mentioning technical terms both of the domain and of the machine. These technical terms thus cover shared phenomena and concepts, that is, phenomena and concepts of the domain which are, in some sense, also (to be) represented by the machine. Interface requirements represent (i) the initialisation and "on-the-fly" update of machine endurants on the basis of *shared* domain endurants; (ii) the interaction between the machine and the domain while the machine is carrying out a (previous domain) action; (iii) machine responses, if any, to domain events — or domain responses, if any, to machine events cum "outputs"; and (iv) machine monitoring and machine control of domain phenomena. Each of these four (i–iv) interface requirement facets themselves involve projection, instantiation, determination, extension and fitting. Machine requirements are those requirements which can be expressed, say in narrative form, by mentioning technical terms only of the machine. (An example is: visual display units.)

[5] We omit consideration of *fitting*.

6.3.5 Verifiable Software Development

An Example Set of Conjectures

We illustrate some conjectures.

(A) From a domain, $\mathscr{D}$, one can develop a domain description $\mathbb{D}$. $\mathbb{D}$ cannot be [formally] verified. It can be [informally] validated "against" $\mathscr{D}$. Individual properties, $\mathbb{P}_{\mathbb{D}}$, of the domain description $\mathbb{D}$ and hence, purportedly, of the domain, $\mathscr{D}$, can be expressed and possibly proved $\mathbb{D} \models \mathbb{P}_{\mathbb{D}}$ and these may be validated to be properties of $\mathscr{D}$ by observations in (or of) that domain.

(B) From a domain description, $\mathbb{D}$, one can develop requirements, $\mathbb{R}_{DE}$, for, and from $\mathbb{R}_{DE}$ one can develop a domain demo machine specification $\mathbb{M}_{DE}$ such that $\mathbb{D},\mathbb{M}_{DE} \models \mathbb{R}_{DE}$. The formula $\mathbb{D},\mathbb{M} \models \mathbb{R}$ can be read as follows: in order to prove that the $\mathbb{M}$achine satisfies the $\mathbb{R}$equirements, assumptions about the $\mathbb{D}$omain must often be made explicit in steps of the proof.

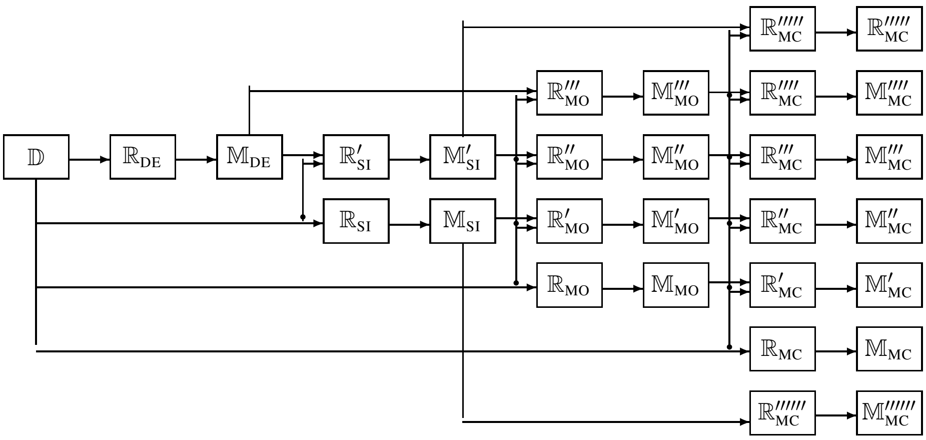
(C) From a domain description, $\mathbb{D}$, and a domain demo machine specification, $\mathbb{S}_{DE}$, one can develop requirements, $\mathbb{R}_{SI}$, for, and from such a $\mathbb{R}_{SI}$ one can develop a domain simulator machine specification $\mathbb{M}_{SI}$ such that $(\mathbb{D};\mathbb{M}_{DE}),\mathbb{M}_{SI} \models \mathbb{R}_{SI}$. We have "lumped" $(\mathbb{D};\mathbb{M}_{DE})$ as the two constitute the extended domain for which we, in this case of development, suggest the next stage requirements and machine development to take place.

(D) From a domain description, $\mathbb{D}$, and a domain simulator machine specification, $\mathbb{M}_{SI}$, one can develop requirements, $\mathbb{R}_{MO}$, for, and from such a $\mathbb{R}_{MO}$ one can develop a domain monitor machine specification $\mathbb{M}_{MO}$ such that $(\mathbb{D};\mathbb{M}_{SI}),\mathbb{M}_{MO} \models \mathbb{R}_{MO}$.

(E) From a domain description, $\mathbb{D}$, and a domain monitor machine specification, $\mathbb{M}_{MO}$, one can develop requirements, $\mathbb{R}_{MC}$, for, and from such a $\mathbb{R}_{MC}$ one can develop a domain monitor & controller machine specification $\mathbb{M}_{MC}$ such that $(\mathbb{D};\mathbb{M}_{MO}),\mathbb{M}_{MC} \models \mathbb{R}_{MC}$.

Chains of Verifiable Developments

The above illustrated just one chain (A–E) of developments. There are others. All are shown in Fig. 6.3.



Legend: $\mathbb{D}$ domain, $\mathbb{R}$ requirements, $\mathbb{M}$ machine
DE:DEMO, SI: SIMULATOR, MO: MONITOR, MC: MONITOR & CONTROLLER

Fig. 6.3. Chains of Verifiable Developments

Figure 6.3 can also be interpreted as prescribing a widest possible range of machine cum software products [192, 193] for a given domain. One domain may give rise to many different kinds of DEmo machines, SImulators, MOnitors and Monitor & Controllers (the unprimed versions of the $\mathbb{M}_T$ machines (where T ranges over DE, SI, MO, MC)). For each of these there are similarly, "exponentially" many variants of successor machines (the primed versions of the $\mathbb{M}_T$ machines). What does it mean that a machine is a primed version? Well, here it means, for example, that $\mathbb{M}'_{SI}$ embodies facets of the demo machine $\mathbb{M}_{DE}$, and that $\mathbb{M}'''_{\text{MC}}$ embodies facets of the demo machine $\mathbb{M}_{\text{DE}}$, of the simulator $\mathbb{M}'_{\text{SI}}$, and the monitor $\mathbb{M}''_{\text{MO}}$. Whether such requirements are desirable is left to product customers and their software providers [192, 193] to decide.

6.4 Conclusion

Our divertimento is almost over. It is time to conclude.

6.4.1 Discussion

The $\mathbb{D},\mathbb{M} \models \mathbb{R}$ ('correctness' of) development relation appears to have been first indicated in the Computational Logic Inc. `Stack` [194, 195] and the EU ESPRIT `ProCoS` [196, 197] projects; [119] presents this same idea with a purpose much like ours, but with more technical discussions.

The term 'domain engineering' appears to have at least two meanings: the one used here [180, 4] and one [198, 199, 200] emerging out of the Software Engineering Institute at CMU where it is also called *product line engineering*[6]. Our meaning, is, in a sense, more narrow, but then it seems to also be more highly specialised (with detailed description and formalisation principles and techniques). Fig. 6.3 on the facing page illustrates, in capsule form, what we think is the CMU/SEI meaning. The relationship between, say Fig. 6.3 and *model-based software development* seems obvious but need be explored. An extensive discussion of the term 'domain', as it appears in the software engineering literature is found in [2, Sect. 5.3].

What Have We Achieved

We have characterised a spectrum of strongly domain-related as well as strongly inter-related (cf. Fig. 6.3) software product families: *demos*, *simulators*, *monitors* and *monitor & controllers*. We have indicated varieties of these: simulators based on demos, monitors based on simulators, monitor & controllers based on monitors, in fact any of the latter ones in the software product family list as based on any of the earlier ones. We have sketched temporal relations between simulation traces and domain behaviours: *a priori*, *a posteriori*, *macroscopic* and *microscopic*, and we have identified the real-time cases which lead on to monitors and monitor & controllers.

What Have We Not Achieved — Some Conjectures

We have not characterised the software product family relations other than by the $\mathbb{D},\mathbb{M} \models \mathbb{R}$ and $(\mathbb{D};\mathbb{M}_{\text{XYZ}}),\mathbb{M} \models \mathbb{R}$ clauses. That is, we should like to prove conjectured type theoretic inclusion relations like:

$$\wp([\![\mathscr{M}_{\text{X}_{\underline{\text{mod}}\ ext.}}]\!]) \sqsupseteq \wp([\![\mathscr{M}'^{\ldots'}_{\text{X}_{\underline{\text{mod}}\ ext.}}]\!]), \quad \wp([\![\mathscr{M}'^{\ldots'}_{\text{X}_{\underline{\text{mod}}\ ext.}}]\!]) \sqsupseteq \wp([\![\mathscr{M}''^{\ldots'}_{\text{X}_{\underline{\text{mod}}\ ext.}}]\!])$$

where X and Y range appropriately, where $[\![\mathscr{M}]\!]$ expresses the meaning of $\mathscr{M}$, where $\wp([\![\mathscr{M}]\!])$ denote the space of all machine meanings and where $\wp([\![\mathscr{M}_{x_{\underline{\text{mod}}\ ext.}}]\!])$ is intended to denote that space modulo ("free of") the y facet (here *ext.*, for extension).

That is, it is conjectured that the set of more specialised, i.e., n primed, machines of kind x is type theoretically "contained" in the set of m primed (unprimed) x machines ($0 \leq m < n$).

There are undoubtedly many such interesting relations between the DEMO, SIMULATOR, MONITOR and MONITOR & CONTROLLER machines, unprimed and primed.

[6] http://en.wikipedia.org/wiki/Domain_engineering.

What Should We Do Next

This paper has the subtitle: *A Divertimento of Ideas and Suggestions.* It is not a proper theoretical paper. It tries to throw some light on families and varieties of software, i.e., their relations. It focuses, in particular, on so-called DEMO, SIMULATOR, MONITOR and MONITOR & CONTROLLER software and their relation to the "originating" domain, i.e., that in which such software is to serve, and hence that which is being *extended* by such software, cf. the compounded 'domain' $(\mathbb{D};\mathbb{M}_i)$ of in $(\mathbb{D};\mathbb{M}_i),\mathbb{M}_j \models \mathbb{D}$. These notions should be studied formally. All of these notions: requirements projection, instantiation, determination and extension can be formalised; and the specification language, in the form used here (without CSP processes, [23] has a formal semantics and a proof system — so the various notions of development, $(\mathbb{D};\mathbb{M}_i),\mathbb{M}_j \models \mathbb{R}$ and $\wp(\mathbb{M})$ can be formalised.

Part IV

Issues of Philosophy

7

Philosophical Issues

We show how the domain analysis & description calculi of Chapter 1 [1, 2] satisfy Kai Sørlander's Philosophy, but also that Sørlander's Philosophy, notably [201] and [202] mandates extensions to the calculi of [2] in order to form a more consistent "whole". Where, in [2], discrete parts were just that, we must now distinguish between three kinds of parts: (i) **physical parts**, (ii) **living species parts**, and (iii) **artifacts**.

(i) The **physical parts** are not made by man, but are in ***space*** and ***time***; these are ***endurants*** that are subject to the ***laws*** of physics as formulated by for example ***Newton*** and ***Einstein***, and also subject to the ***principle of causality*** and ***gravitational pull*** – but were not so explicated in [2] – hence the need for the revision of [2] into [1], i.e., Chapter 1.

(ii) The **living species parts** are ***plants*** and ***animals***; they are still subject to the laws and principles of physics, but additionally ***unavoidably*** endowed with such properties as ***causality of purpose***. Animals have ***sensory organs***, ***means of motion***, ***instincts***, ***incentives*** and ***feelings***. Among animals we single out **humans** as parts that are further characterisable: possessing ***language***, ***learning skills***, being ***consciousness***, and having ***knowledge***. These aspects were somehow, by us, subsumed in our analysis & description by partially endowing ***physical part***s with such properties.

(iii) Then there are the parts made by humans, i.e., **artifacts**. ***Artifacts*** have a usual set of attributes of the kind ***physical parts*** can have; but in addition they have a ***distinguished attribute:*** **attr_Intent** – expressed as a set of intents by the ***humans*** who constructed them according to some ***purpose***. This more-or-less "standard" ***property of intents*** determines a form of ***counterpart*** to the ***gravitational pull*** of ***physical parts*** namely, what we shall refer to as ***intentional "pull"***. Also these were subsumed in [2] – by either partially endowing ***physical part***s with such properties, or by ***ignoring*** them !

We thus suggest a **philosophy basis** for **domain science & engineering**. This chapter is based on recent research [1, 2, 3, 4, 5, 6, 7, 170, 9, 10, 11, 12] into methods for analysing and describing human-centered universes of discourses such as ***transport nets, container lines, credit cards*** (Appendix A), ***weather information*** (Appendix B), ***pipelines*** (Appendix C), ***documents*** (Appendix D), ***urban planning*** (Appendix E), ***drones*** (Appendix F), etc. The present paper is motivated by speculations about possible "interfaces" between domain analysis & description methods and the reality they model.

This chapter builds strongly on Chapter 1's calculi: one for **analysing** manifest "worlds" and one for **describing** those "realities". We "interpret" ***manifest endurant entities*** as ***behaviours*** i.e., as ***perdurants***. This interpretation is, from the point-of-view of post-Kantian philosophy, a **transcendental deduction**, i.e. cannot be logically explained, but can be understood extra-logically. In a more-or-less summary section we shall then show that the calculi are necessary and sufficient, in that they have a basis in philosophical reasoning. But, what is to us more interesting, we show how the Sørlander Philosophy "kicks back" and either mandates or requires domain properties not covered in my earlier paper on the **domain analysis & description** [2].

7.1 Introduction

Definition 64 Domains: By a **domain** we shall understand a **rationally describable** segment of a **human assisted** reality, i.e., of the world, its **physical parts**, and **living species**. These are **endurants**

("still"), existing in space, as well as **perdurants** ("alive"), existing also in time. Emphasis is placed on **"human-assistedness"**, that is, that there is ***at least one (man-made)*** **artifact** and that **humans** are a primary cause for change of endurant **states** as well as perdurant **behaviours** ■

The ***science and engineering of domain analysis & description*** is ***different*** from the science of physics and the core of its derived engineerings: building (civil), chemical, mechanical, electrical, electronics, et cetera. All of these engineerings emerged out of ***practical constructing*** and the ***the natural sciences***. The classical engineering disciplines have increasingly included many facets of ***man-machine interface*** concerns, but their core is still in the ***the natural sciences***.

The core of ***domain science & engineering*** such as we shall pursue it, is in two disciplines: ***mathematics***, notably ***mathematical logic*** and ***abstract algebra***, and ***philosophy***, notably ***meta physics*** and ***epistemology***. We assume that the readers are familiar with the above-mentioned notions of mathematics. Definitions 5 on Page 2 and 6 on Page 2 characterized the concepts of ***meta physics*** and ***epistemology***.

Topics of metaphysical investigation include existence, objects and their properties, space and time, cause and effect, and possibility. The philosophy aspect of our study is primarily epistemological, not metaphysical.

Epistemology studies the nature of knowledge, justification, and the rationality of belief. Much of the debate in epistemology centers on four areas: (1) the philosophical analysis of the nature of knowledge and how it relates to such concepts as truth, belief, and justification, (2) various problems of skepticism, (3) the sources and scope of knowledge and justified belief, and (4) the criteria for knowledge and justification.[1] A central branch of epistemology is ***ontology***, the investigation into the basic categories of being and how they relate to one another.[2]

Observe the distinction in the definitions of metaphysics and epistemology between [metaphysics] ***"explores fundamental questions, including the nature of concepts like being, existence, and reality"*** and [epistemology] ***"the philosophical analysis of the nature of knowledge and how it relates to such concepts as truth, belief, and justification, etc."***. Epistemology addresses such questions as ***What makes justified beliefs justified ?"***; ***"What does it mean to say that we know something ?"*** and, fundamentally, ***"How do we know that we know ?"***[3]

7.1.1 Two Views of Domains

There are two aspects to this chapter: (i) the analysis & description of fragments of the context in which software, to be developed, is to serve, (ii) and the general, basically philosophical, problem of the absolutely necessary conditions for describing the world.

The Computing Science View

In twelve papers, six pairs, now collected in the six preceding chapters of this monograph, we have put forward a method for analysing and describing the domains for which software is developed:

- Chapter 1 [1, 2] Manifest Domains: Analysis & Description
- Chapter 2 [3, 4] Domain Facets: Analysis & Description
- Chapter 3 [5, 6] Formal Models of Processes and Prompts
- Chapter 4 [7, 8] To Every Manifest Domain Mereology a CSP Expression
- Chapter 5 [9, 10] From Domain Descriptions to Requirements Prescriptions
- Chapter 6 [11, 12] Domains: Their Simulation, Monitoring and Control

These methods involve new principles, techniques and tools – the ***calculi***. The calculi has been applied in around 20+ experimental researches to as diverse domains as

[1] https://en.wikipedia.org/wiki/Epistemology
[2] https://en.wikipedia.org/wiki/Metaphysics
[3] https://en.wikipedia.org/wiki/Epistemology

- railways,
- IT security,
- container shipping lines,
- "the market",
- road transport systems,
- stock exchanges,
- credit card systems (Appendix A),
- weather information (Appendix B),
- pipelines (Appendix C),
- documents (Appendix D),
- urban planning (Appendix E) and
- swarms of drones (Appendix F).

The calculi, we claim, has withstood some severe "tests".

The Philosophy View

In four books the Danish philosopher Kai Sørlander has investigated the philosophical issues alluded to above.

- [17] Kai Sørlander. *Det Uomgængelige – Filosofiske Deduktioner [The Inevitable – Philosophical Deductions] Forord/Foreword: Georg Henrik von Wright*. Munksgaard · Rosinante, 1994. 168 pages.
- [18] Kai Sørlander. *Under Evighedens Synsvinkel [Under the viewpoint of eternity]*. Munksgaard · Rosinante, 1997. 200 pages.
- [19] Kai Sørlander. *Den Endegyldige Sandhed [The Final Truth]*. Rosinante, 2002. 187 pages.
- [20] Kai Sørlander. *Indføring i Filosofien [Introduction to The Philosophy]*. Informations Forlag, 2016. 233 pages.

A main contribution of Sørlander is, ***on the philosophical basis of the possibility of truth*** (in contrast to Kant's ***possibility of self-awareness cognition***), to ***rationally*** and ***transcendentally deduce*** **the absolutely necessary conditions for describing any world**. These conditions presume a ***principle of contradiction*** and lead to the ***ability*** to ***reason*** using ***logical connectives*** and to ***handle asymmetry, symmetry*** and ***transitivity***. ***Transcendental deductions*** then lead to ***space*** and ***time***, not as priory assumptions, as with Kant, but derived facts of any the world. From this basis Sørlander then, by further transcendental deductions arrive at kinematics, dynamics and the bases for Newton's Laws. And so forth. We build on Sørlander's basis to argue ***that the domain analysis & description calculi are necessary and sufficient*** and that a number of relations between domain entities can be understood transcendentally and as "variants" of Newton's Laws !

First Two Independent Treatments, then An Interpretation

In Chapter 1 we presented one, a new computing science approach, to the analysis & description of "reality": the ***principles, techniques*** and ***tools***, Sects. 1.2–1.7 of the **domain analysis & description**, and a ***substantial example***, Sect. 1.8, to ***support understanding*** the **domain analysis & description**.

In Sects. 7.3–7.5 we then present a philosophy-based foundation for describing "reality". In Sect. 7.3 a brief motivation of the task of philosophy; in Sect. 7.4 an extensive review is presented of metaphysical and epistemological issues in philosophy, from the ancient Greeks up til the mid 1900's; and in Sect. 7.5 an extensive review is then given of Sørlander's Philosophy.

Then, in Sect. 7.6, we bring the two studies — the ***domain analysis & description calculi*** and the ***Kai Sørlander Philosophy*** — together: It is here that, as a consequence of Sørlander's Philosophy, we modify the **domain analysis & description**, of Chapter 1, in suggesting extensions.

The Main Contribution

With Sects. 7.5–7.6 the ***the main contribution*** of this monograph is achieved:

- establishing ***a basis*** for ***domain science & engineering*** in ***philosophy***; and
- the ***specific modifications*** required by
  and the ***founding*** of the domain analysis & description calculi in ***philosophy***.

7.2 Space Time

Editorial Remark

YOU MAY SKIP THIS SECTION, I.E., THIS AND THE NEXT 3+ PAGES.

The presentation of the domain analysis & description calculi avoided, in principle, references to space and time; but these concepts are there: "buried" as follows: endurants can be said to "exist" in space and perdurants to "exist" in time. We shall briefly examine these two concepts as they have been the concern of mathematicians. We shall not be interested in the physicists' ***spacetime*** mathematical model that fuses the three dimensions of space and the one dimension of time into a single four-dimensional continuum.

7.2.1 Space

> ***Space is the boundless three-dimensional extent in which objects and events have relative position and direction[4]. Physical space is often conceived in three linear dimensions, although modern physicists usually consider it, with time, to be part of a boundless four-dimensional continuum known as space-time. The concept of space is considered to be of fundamental importance to an understanding of the physical universe. However, disagreement continues between philosophers over whether it is itself an entity, a relationship between entities, or part of a conceptual framework[5].***

To us ***space*** is a conceptual framework. That is, it is not an entity, hence neither an endurant nor a perdurant. Here we shall primarily look at space as a mathematical construction. In Sect. 7.5 we shall widen that consideration considerably.

Topological Space

One notion of space, in mathematics, is that of a Hausdorf (or topological) space:

Definition 65 **Topological Space:** A **topological space** is an ordered pair (X,τ), where X is a set and τ is a collection of subsets of X, satisfying the following axioms:[6]

- The empty set and X itself belong to τ.
- Any (finite or infinite) union of members ofτ still belongs to τ.
- The intersection of any finite number of members of τ still belongs to τ ■

The elements of τ are called **open set**s and the collection τ is called a **topology** on X.

Metric Space

A metric spaces is a set for which distances between all members of the set are defined. Those distances, taken together, are called a metric on the set. A metric on a space induces topological properties like open and closed sets, which lead to the study of more abstract topological spaces.

Definition 66 **Metric Space:** A **metric space** is an ordered pair (M,d) where M is a set and d is a metric on M, i.e., a function

- $d : M \times M \to \mathbb{R}$

such that for any $x,y,z : M$, the following holds:[7]

[4] https://www.britannica.com/science/space-physics-and-metaphysics
[5] https://en.wikipedia.org/wiki/Space
[6] Armstrong, M. A. (1983) [1979]. Basic Topology. Undergraduate Texts in Mathematics. Springer. ISBN 0-387-90839-0.
[7] B. Choudhary (1992). The Elements of Complex Analysis. New Age International. p.20. ISBN 978-81-224-0399-2.

- 1. $d(x,y) \geq 0$ non-negativity or separation axiom
- 2. $d(x,y) = 0 \Leftrightarrow x = y$ identity of indiscernibles
- 3. $d(x,y) = d(y,x)$ symmetry
- 4. $d(x,z) \leq d(x,y) + d(y,z)$ subadditivity or triangle inequality ■

Euclidian Space

The notion of ***Euclidian Space*** is due to ***Euclid of Alexandria*** [325–265]. Euclid postulated

Example 7.1. **Euclid's Postulates**:

- To draw a straight line from any point to any point.
- To produce [extend] a finite straight line continuously in a straight line.
- To describe a circle with any centre and distance [radius].
- That all right angles are equal to one another.
- [The parallel postulate] That, if a straight line falling on two straight lines make the interior angles on the same side less than two right angles, the two straight lines, if produced indefinitely, meet on that side on which are the angles less than the two right angles ■

We refer to Euclidean space. Encyclopedia of Mathematics. URL: http://www.encyclopediaofmath.-org/index.php?title=Euclidean_space&oldid=38673 The European Mathematical Society and Springer.

Example 7.2. **Euclid's Plane Geometry**: The Euclidean geometry informally described in Example 7.1 can be formally axiomatised by first introducing the sorts P and L:

```
type
  P, L
value
  [0] obs_Ps: L → P-infset
      parallel: L × L → Bool
```

Observe how the informal axiom in Example 7.1 has been modelled by the *observer function* obs_Ps. It applies to lines and yields possibly infinite sets of points.

Now we can introduce the axioms proper:

```
axiom
  [1] ∃ p,q:P • p ≠ q,
  [2] ∀ p,q:P • p ≠ q ⇒
            ∃! l:L • p ∈ obs_Ps(l) ∧ q ∈ obs_Ps(l),
  [3] ∀ l:L • ∃ p:P • p ∉ obs_Ps(l),
  [4] ∀ l:L • ∃ p:P • p ∉ obs_Ps(l) ⇒
            ∃ l′:L • l≠l′ ∧ p ∈ obs_Ps(l′) ∧ parallel(l,l′)
```

The concept of being parallel is modelled by the predicate symbol of the same name, by its signature and by axiom [4] ■

We leave it to the reader to reconcile the models of topological space, Defn. 65 on the preceding page, and metric space, Defn. 66 on the facing page, with the axioms systems of examples 7.1 and 7.2.

7.2.2 Time

(i) A moving image of eternity;
(ii) The number of the movement
in respect of the before and the after;
(iii) The life of the soul in movement as it passes

```
from one stage of act or experience to another;
        (iv) A present of things past: memory,
             a present of things present: sight,
  and a present of things future: expectations.
```

[43, (i) Plato, (ii) Aristotle, (iii) Plotinus, (iv) Augustine].

Time — General Issues

In the next sections we shall focus on various models of time, and we shall conclude with a simple view of the operations we shall assume when claiming that an abstract type models time. These sections are far from complete. They are necessary, but, as a general treatment of notions of time, they are not sufficient. We refer the interested reader to special monographs: [48, 46, 203, 204, 205, 206, 207, 208, 209, 210, 211].

"A-Series" and "B-Series" Models of Time

Colloquially, in ordinary, everyday parlance, we think of time as a dense series of time points. We often illustrate time by a usually horizontal line with an arrow pointing towards the right. Sometimes that line arrowhead is labeled with either a t or the word *time*, or some such name. J.M.E. McTaggart (1908, [203, 46, 211]) discussed theories of time around two notions:

- **"A-series":** has terms like "past", "present" and "future".
- **"B-series":** has terms like "precede", "simultaneous" and "follow".

McTaggart argued that the B-series presupposes the A-series: If t precedes t' then there must be a "thing" t'' at which t is past and t' is present. He argued that the A-series is incoherent: What was once 'future', becomes 'present' and then 'past'; and thus events 'will be events', 'are events' and 'were events', that is, will have all three properties.

A Continuum Theory of Time

The following is taken from Johan van Benthem [48]: Let P be a point structure (for example, a set). Think of time as a continuum; the following axioms characterise ordering $(<, =, >)$ relations between (i.e., aspects of) time points. The axioms listed below are not thought of as an axiom system, that is, as a set of independent axioms all claimed to hold for the time concept, which we are encircling. Instead van Benthem offers the individual axioms as possible "blocks" from which we can then "build" our own time system — one that suits the application at hand, while also fitting our intuition.

Time is transitive: If $p{<}p'$ and $p'{<}p''$ then $p{<}p''$. Time may not loop, that is, is not reflexive: $p \not< p$. Linear time can be defined: Either one time comes before, or is equal to, or comes after another time. Time can be left-linear, i.e., linear "to the left" of a given time. The following is taken from Johan van Benthem [48]: Let P be a point structure (for example, a set). Think of time as a continuum; the following axioms characterise ordering $(<, =, >)$ relations between (i.e., aspects of) time points. The axioms listed below are not thought of as an axiom system, that is, as a set of independent axioms all claimed to hold for the time concept, which we are encircling. Instead van Benthem offers the individual axioms as possible "blocks" from which we can then "build" our own time system — one that suits the application at hand, while also fitting our intuition.

Time is transitive: If $p{<}p'$ and $p'{<}p''$ then $p{<}p''$. Time may not loop, that is, is not reflexive: $p \not< p$. Linear time can be defined: Either one time comes before, or is equal to, or comes after another time. Time can be left-linear, i.e., linear "to the left" of a given time. One could designate a time axis as beginning at some time, that is, having no predecessor times. And one can designate a time axis as ending at some time, that is, having no successor times. General, past and future successors (predecessors, respectively successors in daily talk) can be defined. Time can be dense: Given any two times one can always find a time between them. Discrete time can be defined.

axiom

[TRANS: Transitivity] $\forall$ p,p′,p″:P • p < p′ < p″ ⇒ p < p″
[IRREF: Irreflexitivity] $\forall$ p:P • p ≮ p
[LIN: Linearity] $\forall$ p,p′:P • (p=p′ ∨ p<p′ ∨ p>p′)
[L−LIN: Left Linearity] $\forall$ p,p′,p″:P • (p′<p ∧ p″<p) ⇒ (p′<p″ ∨ p′=p″ ∨ p″<p′)
[BEG: Beginning] ∃ p:P • ∼∃ p′:P • p′<p
[END: Ending] ∃ p:P • ∼∃ p′:P • p<p′
[SUCC: Successor]
    [PAST: Predecessors] $\forall$ p:P,∃ p′:P • p′<p
    [FUTURE: Successor] $\forall$ p:P,∃ p′:P • p<p′
[DENS: Dense] $\forall$ p,p′:P (p<p′ ⇒ ∃ p″:P • p<p″<p′)
[DENS: Converse Dense] ≡ [TRANS: Transitivity]
[DISC: Discrete]
    $\forall$ p,p′:P • (p<p′ ⇒ ∃ p″:P • (p<p″ ∧ ∼∃ p‴:P • (p<p‴<p″))) ∧
    $\forall$ p,p′:P • (p<p′ ⇒ ∃ p″:P • (p″<p′ ∧ ∼∃ p‴:P • (p″<p‴<p′)))

A strict partial order, SPO, is a point structure satisfying TRANS and IRREF. TRANS, IRREF and SUCC imply infinite models. TRANS and SUCC may have finite, "looping time" models.

7.2.3 Wayne D. Blizard's Theory of Space–Time

We now bring space and time together in an axiom system (Wayne D. Blizard, 1980 [47]) which relate abstracted entities to spatial points and time. Let $A,B,\ldots$ stand for entitites, $p,q,\ldots$ for spatial points, and t,τ for times. 0 designates a first, a begin time. Let t' stand for the discrete time successor of time t. Let $N(p,q)$ express that p and q are spatial neighbours. Let $=$ be an overloaded equality operator applicable, pairwise to entities, spatial locations and times, respectively. A^t_p expresses that entity A is at location p at time t. The axioms — where we omit (obvious) typings (of A, B, P, Q, and T): $'$ designates the time successsor function: t'.

(I)	$\forall A \forall t \exists p \;:\; A^t_p$	
(II)	$(A^t_p \wedge A^t_q) \supset p=q$	
(III)	$(A^t_p \wedge B^t_p) \supset A=B$	
$(IV)(?)$	$(A^t_p \wedge A^{t'}_p) \supset t=t'$	
$(V\ i)$	$\forall p,q \;:\; N(p,q) \supset p \neq q$	Irreflexivity
$(V\ ii)$	$\forall p,q \;:\; N(p,q) = N(q,p)$	Symmetry
$(V\ iii)$	$\forall p \exists q,r \;:\; N(p,q) \wedge N(p,r) \wedge q \neq r$	No isolated locations
$(VI\ i)$	$\forall t \;:\; t \neq t'$	
$(VI\ ii)$	$\forall t \;:\; t' \neq 0$	
$(VI\ iii)$	$\forall t \;:\; t \neq 0 \supset \exists \tau : t = \tau'$	
$(VI\ iv)$	$\forall t,\tau \;:\; \tau' = t' \supset \tau = t$	
(VII)	$A^t_p \wedge A^{t'}_q \supset N(p,q)$	
$(VIII)$	$A^t_p \wedge B^t_q \wedge N(p,q) \supset \sim (A^{t'}_q \wedge B^{t'}_p)$	

We comment on these axioms:

- II–IV,VII–VIII: The axioms are universally 'closed'; that is: We have omitted the usual $\forall A,B,p,q,t$s.
- (I): For every entity, A, and every time, t, there is a location, p, at which A is located at time t.
- (II): An entity cannot be in two locations at the same time.
- (III): Two distinct entities cannot be at the same location at the same time.
- (IV): Entities always move: An entity cannot be at the same location at different times. *This is more like a conjecture: Could be questioned.*
- (V): These three axioms define N.
- (V i): Same as $\forall p : \sim N(p,p)$. "Being a neighbour of", is the same as "being distinct from".
- (V ii): If p is a neighbour of q, then q is a neighbour of p.
- (V iii): Every location has at least two distinct neighbours.
- (VI): The next four axioms determine the time successor function $'$.
- (VI i): A time is always distinct from its successor: time cannot rest. There are no time fix points.
- (VI ii): Any time successor is distinct from the begin time. Time 0 has no predecessor.

- (VI iii): Every non–begin time has an immediate predecessor.
- (VI iv): The time successor function $'$ is a one–to–one (i.e., a bijection) function.
- (VII): The *continuous path axiom:* If entity A is at location p at time t, and it is at location q in the immediate next time (t'), then p and q are neighbours.
- (VIII): *No "switching":* If entities A and B occupy neighbouring locations at time t them it is not possible for A and B to have switched locations at the next time (t').

Except for Axiom (IV) the system applies both to systems of entities that "sometimes" rests, i.e., do not move. These entities are spatial and occupy at least a point in space. If some entities "occupy more" space volume than others, then we may suitably "repair" the notion of the point space P (etc.). We do not show so here.

7.3 A Task of Philosophy

Philosophy is the study of general and fundamental problems concerning matters such as ***existence, knowledge***[8]***, values, reason, mind,*** and ***language.***

7.3.1 Epistemology

We shall focus on ***existence***, specifically on ***epistemology*** – meaning 'knowledge' and 'logical discourse' – it is the branch of philosophy concerned with the theory of knowledge. Epistemology studies the nature of knowledge, justification, and the rationality of belief. Much of the debate in epistemology centers on four areas: (1) the philosophical analysis of the nature of knowledge and how it relates to such concepts as truth, belief, and justification, (2) various problems of skepticism, (3) the sources and scope of knowledge and justified belief, and (4) the criteria for knowledge and justification. Epistemology addresses such questions as "What makes justified beliefs justified?", "What does it mean to say that we know something ?", and fundamentally "How do we know that we know ?"

7.3.2 Ontology

A ***"corollary"*** of epistemology is ***ontology***: the philosophical study of the nature of ***being, becoming, existence,*** or ***reality,*** as well as the ***basic categories of being and their relations.***

7.3.3 The Quest

The ***quest*** is now threefold.

(i) First to prepare the ground for a discussion of possible philosophical issues of the domain analysis & description calculi. We do so by a review of philosophy (Pages 205–210) focusing on epistemology and ontology problems – from the ancient Greek philosophers till Bertrand Russell.

(ii) Then to follow that up with a review of the Philosophy of Kai Sørlander as it is, most recently, expressed in [20], and as refined from earlier works: [17, 18, 19]. This is done in Sect. 7.5, Pages 210–217.

(iii) Finally to show, issue-by-issue how concepts of the domain analysis & description calculi more have a basis in philosophy than in mathematics and computer science. This is done in Sect. 7.6, Pages 217–224.

7.3.4 Schools of Philosophy

We shall only cover Western Philosophy, and only to some depth. A seven line summary will be give, in Sect. 7.3.4, of a possibly relevant aspect of Indian Philosophy. We'll leave it at that. The fact is that Indian Philosophy has not, it appears, influenced Western Philosophy. That short summary are in line the choice of issues that we seek to uncover.

[8] including Scientific Knowledge: Mathematics, Physics, Computer Science, etc.

Western Philosophy

Section 7.4 presents a "capsule" summary of Western Philosophy. It is, at present, a "tour de force", seven pages. One purpose of presenting it is that we are then able to enumerate and date the issues relevant to our quest while discarding some of the proposed theories. Another purpose is to remind the reader of the depth, breadth and plurality of issues of Western Philosophy.

Indian Philosophy

***Pramana**, literally means "proof" and "means of knowledge", refers to epistemology in Indian philosophies, The focus of Pramana is how correct knowledge can be acquired, how one knows, how one doesn't, and to what extent knowledge pertinent about someone or something can be acquired. Ancient and medieval Indian texts identify six pramanas as correct means of accurate knowledge and to truths: (1) perception, (2) inference, (3) comparison and analogy, (4) postulation, (5) derivation from circumstances, non-perception, negative/cognitive proof, and (6) word, testimony of past or present reliable experts*[9].

7.4 From Ancient to Kantian Philosophy and Beyond !

The review of this section, i.e., Sect. 7.4, is based primarily on [17]. It is exclusively "slanted" towards those aspects of the thinking of these philosophers with respect to the ***task of philosophy*** as we defined it in Sect. 7.3. In this review we reject the contributions of these great philosophers that is contradictory. This presentational "bias" should in no way stand in way of our general admiration for their otherwise profound thinking.

7.4.1 Pre-Socrates

A number of pre-Socratian thinkers speculated on how the world was "constructed". The earlier thinkers were pre-occupied with ***matter***, that is, ***substance***; what did the world consist of, how was it constructed ? In doing that these thinkers were trying to be scientists, they were not, in this philosophers. We briefly review some of the pre-Socratian thinkers and philosophers.

Thales of Miletus, 624–546 BC [20, pp 35] *"claimed*[10] *that all existing, i.e., **base matter**, derived from **water**"*; **Anaximander of Miletus, 610–546 BC** [20, pp 35-36] *"that **base matter** all came from **apeiron**, some further unspecified substance"*; **Anaximenes of Miletus, 585–528 BC** [20, pp 36] *"that **base matter** was **air**"*; **Heraklit of Efesos, a. 500 BC** [20, pp 37] *"claimed that **fire** was the **base matter**; and extended the concern from **substance** to **permanence** and based the thinking not only on (**empirical**) **observations** but also on **logical reasoning** claiming that everything in the world was in a constant struggle, all the time changing – so since all is **changing**, i.e., that nothing is **stable**, he concludes that **nothing exists**."* In that Heraklit was a philosopher.

And, from now, philosophy reigned.

Parmenides of Elea, 501–470 BC [20, pp 37-38, 48-49] *"counterclaimed that that which actually exists is **eternal** and **unchanging** – is logically impossible"*; **Zeno of Elea, 490–430 BC** [20, pp 38-39] *"supported Parmenindes' claim by claiming some paradox, i.e., the well-known Achilles and the tortoise – thereby introducing **dialectic reasoning** and **proof by contradiction** (**reductio ad absurdum**)"*; **Demokrit, 460–370 BC** [20, pp 40-42] *"tried to unify Heraklit's concept of **changeability** and Parmenides' concept of **permanence** in a new way; everything in the world is built from, consists of **atoms** and change is due to movement of atoms"*. **The Sophists, 5th Century BC** [20, pp 43-44] *"doubted, or even refuted, that we can arrive at universal truths about the world purely through reasoning. They refute that there is an objectively true reality which we can obtain knowledge about. So, instead, skepticism reigned"*.

What is interesting, to us, is that, the thinking of even the early Greek thinkers delineates the realms of religion and mythology on one side, and those of science and philosophy, on the other side.

[9] https://en.wikipedia.org/wiki/Pramana
[10] [20, pp 35] refers to Sørlander's book [20] Page 35.

7.4.2 Plato, Socrates and Aristotle

Socrates, 470–399 BC [20, pp 44-45] *"protested against the sophists' refusal of reason, common sense, sanity and prudence"*. We know of Socrates' thinking almost exclusively through **Plato, 427–347 BC**: [20, pp 46-49] *"We shall focus on Plato's **theory of ideas**. His argument is that non-physical (but substantial) **ideas** represent the most accurate reality. Abstract and common concepts obtain meaning through standing for ideas that are eternal and unchangeable. In contrast to **ideas** Plato considers the concept of a **phenomenon**. **Phenomena are instances of ideas. We recognize a phenomenon because it embodies an idea.** So, according to Plato, the changeable world that surrounds us, one which we experience through our senses, is only a reflection of a, or the, real world. That real world is unchangeable and "consists" of ideas"*.[11] **Aristotle, 384–322 BC**. [20, pp 50-53] *"For Aristotle it was not Plato's **abstract ideas** that "existed" but the **concrete world** of which we are a part of with our body. The abstract ideas, however, in Aristotle's thinking, constitute a system for describing the world.[12] We shall very briefly list two of the concept clusters that Aristotle made to our thinking of the world: (i) **modalities** and (ii) **explanations** – the latter also referred to as **causes**. The **modalities** are: (i.1) **necessity**, that which is unavoidably so; (i.2) **reality**, that which we observe; and (i.3) **possibility**, that which might be. The **causes** (or **explanations**) are: (ii.1) **matter** or **material cause**, (ii.2) **form cause** or **formal cause** (ii.3) **agent cause** and (ii.4) **end cause** or **purpose cause** (ii.1) By **material cause** Aristotle means the aspect of the change or movement which is determined by the material that composes the moving or changing things. (ii.2) By **form** or **formal cause** Aristotle means a change or movement's **formal cause**, is a change or movement caused by the arrangement, shape or appearance of the thing changing or moving. (ii.3) By **agent cause** Aristotle means a change or movement's efficient or moving cause, consists of things apart from the thing being changed or moved, which interact so as to be an agency of the change or movement. (ii.4) By **end cause** or **purpose cause** Aristotle means a change or movement's final cause, is that for the sake of which a thing is what it is. Aristotle's contributions are, for us, decisive. Aristotle reveals how **being** is by revealing the **irreducible types of predicates** which we can actually use when **describing the world**. Aristotle thus examines the **categories**: **substance** (human, horse), **quantity** (6 feet tall), **quality** (white, red), **relation** (larger, shorter), **location** (in Athens), **time** (yesterday, last year), **position** (lying, sitting), **posture** (wearing shoes), **action** (running, singing), and **suffering** (being cut). This enumeration[13] is certainly not definitive. Kant, two thousand years later, revives this idea: a **system of unavoidable basic concepts** for the description of the world and our situation in it."*[14]

7.4.3 The Stoics: 300 BC–200 AD

We shall just focus on one aspect of their contribution to logic and philosophy, that of logic. [212, pp 22-23] *"They distinguish between **simple propositions** and **composite propositions**. They also distinguish between three kinds of **propositions**. **implication**, **conjunction** and **disjunction**. They had a special understanding of **implication**: A **proposition** is, to the Stoics, of the composite form:* $\mathbf{A \Rightarrow B; A; B}$. *For example:* If it is day then it is light; it is day; therefore it is light. *In this and many other ways they contributed to the philosophy of logic (from which, it seems Gottlob Frege was inspired)"*. **Chrysippus of Soli: 279–206 BC** was a prominent early Stoic.

[11] One may, rather crudely, interpret Plato's concept of ideas with that of types. A value of some type is then a 'phenomenon'.

[12] It should be quite clear, to the reader, that, in this, we follow Aristotle: A main descriptional, in fact, specificational, tool is that of ***type definitions***.

[13] *"Of things said without any combination, each signifies either substance or quantity or qualification or a relative or where or when or being-in-a-position or having or doing or being-affected. To give a rough idea, examples of substance are man, horse; of quantity: four-foot, five-foot; of qualification: white, grammatical; of a relative: double, half, larger; of where: in the Lyceum, in the market-place; of when: yesterday, last-year; of being-in-a-position: is-lying, is-sitting; of having: has-shoes-on, has-armour-on; of doing: cutting, burning; of being-affected: being-cut, being-burned."* Ackrill, John (1963). Aristotle, Categories and De Interpretatione. Oxford: At the Clarendon Press. ISBN 0198720866.

[14] It should likewise be obvious to the reader that the notion of ***categories*** is central to our ontological structuring of domain entities.

•••

Almost two thousand years passed before philosophy again flourished. ***Christianity***, in Europe, in a sense, "monopolised" critical thinking. With the ***Renaissance*** and ***Martin Luther's Protestantism*** thinkers again turned to philosophy.

7.4.4 The Rational Tradition: Descartes,

René Descartes: 1596–1650 [20, pp 72–74]*"rejected the splitting of* **corporeal substance** *into* **matter** *and* **form***. His main focus was on the relations between* **mind** *and* **form***: as thinking substance we recognize material substance"*. **Baruch Spinoza: 1632–1677** [20, pp 74-78]*"rejected Descartes's two substances: there is, he claims, is only one substance; for Spinoza God and nature was one and the same"*. **Gottfried Wilhelm Leibniz: 1646–1716** [20, pp 78-79]*"introduced the* **Law of the Indiscernability of Identicals***, It is still in wide use today. It states that if some object x is identical to some object y, then any property that x has, y will have as well"*.[15]

7.4.5 The Empirical Tradition: Locke, Berkeley and Hume

John Locke: 1632–1704. We focus on Locke's ideas of ***sensing***. He defines himself[16]:

as that conscious thinking thing,
(whatever substance, made up of whether spiritual,
or material, simple, or compounded, it matters not)
which is sensible, or conscious of pleasure and pain,
capable of happiness or misery,
and so is concerned for itself,
as far as that consciousness extends.

[20, pp 80-82]*"According to Locke, humans obtain their knowledge about the world through sensory perception. At one level, he claims, the world is "mechanical", so our sensory apparatus is influenced mechanically, for example through tactile or visual means. This sense information is then communicated to our brains. First the mechanical sense data become* **sense ideas***, The sense ideas then become* **reflection ideas***."* In the "jargon" of our **domain analysis & description** the ***sense ideas*** are ***values*** and the ***reflection ideas*** become ***types***. So a central idea in Locke's theory is that all ***cognition*** builds on our ***reflection*** over ***sense ideas***. In other words: *"Can we conclude anything from our sense ideas to knowledge about those "outer" things which cause the sense ideas ?"* [20, pg. 85] To answer that question Locke goes on to distinguish[17] between *"**primary qualities**[18] and **secondary qualities**[19]*. In the jargon of domain analysis & description the primary qualities correspond to "our" ***external qualities***, the secondary qualities to "our" ***internal qualities***, but not quite ! *"Locke views primary qualities as measurable aspects of physical reality and secondary qualities as subjective aspects of physical reality, where "our" domain analysis & description takes both to be somehow measurable. We must therefore claim that our distinction is purely pragmatic"*. Locke now claims: *"(i) that we can, with respect to the primary qualities, deduce from our sense ideas to the reality, the world behind these; (ii) that the primary qualities exist in reality independent*

[15] We refer, forward, to Sect. 7.5.2 on Page 212, and, 'backward', to Chapter 1's Sect. 1.2.2 on Page 13 [***unique identifiers***], for our "response" to Leibniz's ***Law of the Indiscernability of Identicals***.

[16] Locke, John (1997), Woolhouse, Roger, ed., ***An Essay Concerning Human Understanding***, New York: Penguin Books

[17] https://en.wikipedia.org/wiki/Primary/secondary_quality_distinction

[18] Primary qualities are thought to be properties of objects that are independent of any observer, such as solidity, extension, motion, number and figure. These characteristics convey facts. They exist in the thing itself, can be determined with certainty, and do not rely on subjective judgments. For example, if an object is spherical, no one can reasonably argue that it is triangular.

[19] Secondary qualities are thought to be properties that produce sensations in observers, such as color, taste, smell, and sound. They can be described as the effect things have on certain people. Knowledge that comes from secondary qualities does not provide objective facts about things.

of whether we "experience" them or not; and (iii) that this is not the case for the secondary qualities which exist only in our **consciousness***".* **George Berkeley: 1685–1753** [20, pp 82-84] *"points out a problem in Locke's theory: namely that Locke's distinction between primary qualities as being* **objective** *and secondary qualities as being* **subjective** *does not hold. He argues that primary qualities can be subjective. To solve that problem Berkeley denied the existence of a reality "behind" the sense ideas: there is no material reality; reality is our sense ideas:* **esse est precipi**[20] *! The material reality is there because it is continuously experienced by 'God'. The problem now is can we, at all, determine fundamental characteristics about the world and our situation as humans in that world without assuming the concept of independently existing substance".* **David Hume, 1711–1776.** Hume's major work was ***An Enquiry Concerning Human Understanding*** [213]. [20, pp 85-87] *"Where Berkeley eliminated* **material substance** *Hume also eliminated Berkeley's concepts of 'God' and 'Consciousness'. He claimed that the basic sense-impressions, which to Hume were the basis for all valid human recognition, made it impossible to arrive at a valid recognition of 'God' and a substantial 'I'. They must therefore be eliminated when trying to describe the world and our situation in it. According to Hume all that we know are* **sense impressions** *and the* **conceptions** *derived from these. Hume further distinguishes between* **composite** *and* **simple** *(not-composite)* **sense impressions***. Correspondingly Hume distinguishes between* **composite** *and* **simple** *(non-composite)* **ideas***. As a consequence there is no* **necessity** *in the world, nor in possible relations between* **cause** *and* **effect** *This renders Hume's thinking in this area very problematic".*

7.4.6 Immanuel Kant: 1720–1804

[212, pp 280-282] *"Kant was "shaken" by Hume's critique of causality. As a response – along one line of thought – Kant introduced two notions:* ***"Das Ding an sich"*** *is the world that we know, that we sense, and* ***"Das Ding für uns"*** *is a world prior to, outside our cognition. Along another line of thought Kant claimed that there is our cognition.By means of the cognitive tools with which our reason is equipped we reach out for* ***"Das Ding an sich"*** *and forms it according to our cognition. The result is the world as we know it. This means that* **reality** *never means the* ***"Das Ding an sich"****, the world "outside" us, "independent" of us. We are excluded from that world".*

[20, pp 88-92] *"Kant turns the reasoning around. What we empirically observe is determined by our "reasoning apparatus". We do not observe "things" as they are in themselves (****"Das Ding an sich"****), but we "recognize" them as they are formed by our own reasoning apparatus. This "reasoning apparatus" includes some* **intuition forms***:* **space** *and* **time***. These, space and time, are therefore, to Kant, not characteristics of the world as it is, but are some* **intuition forms** *that determine our view of the world. How can it now be possible that we can have* **self-awareness** *on the basis of what we are confronted with – what we see ? Here Kant introduces what he termsthe* **transcendental deduction***. We can only have* **self awareness** *under the assumption that we experience our* **views** *(***outlook***) as expression of objects, "things", that exist independent of our experiencing them !"*

[20, pp 90-91] *"But Kant's concept of* ***"Das Ding an sich"*** *is inconsistent. It is in* **contradiction***, because it itself is* **knowable** *as being* **unknowable***; and it is in contradiction, because it, in a mystical sense, is the* **cause** *of the thing which we know as a phenomenon, but (we) cannot apply the* **cause effect category** *outside the world of phenomena".*

A main contribution of Kant however, is his concept of ***Transcendental Schemata***[21]. *"If pure concepts of the understanding (categories) and sensations are radically different, what common quality allows them to relate?"* Kant wrote the chapter on Schemata in his ***Critique of Pure Reason*** to solve the problem of *"... how we can ensure that categories have 'sense and significance' "*. Transcendental schema are not related to empirical concepts or to mathematical concepts. These schemata connect pure concepts of the understanding, or categories, to the phenomenal appearance of objects in general, that is, objects as such,

[20] ***"to-be-is-to-be-perceived"***

[21] In Kantian philosophy, a transcendental schema (plural: schemata; from Greek: $\sigma\chi\eta\mu\alpha$, "form, shape, figure") is the procedural rule by which a category or pure, non-empirical concept is associated with a sense impression. A private, subjective intuition is thereby discursively thought to be a representation of an external object. Transcendental schemata are supposedly produced by the imagination in relation to time `https://en.wikipedia.org/wiki/Schema_(Kant)#Transcendental_schemata`.

or all objects[22]. Example ***categorical schemas*** are: The categories of quantity all share the schema of number. The categories of quality all have degrees of reality as their schema. "The schema of the category of relation is the order of time"[23]. "The schema of the category of modality is time itself as related to the existence of the object"[24].

7.4.7 Post-Kant

Johann Gottlieb Fichte, 1752–1824 [20, pp 93-94] *"tried to avoid Kant's **Das Ding an sich/Das Ding für uns** dualism by letting the subject, the I, determine the object, the not-I, but ends up in contradiction"*. **Georg Wilhelm Friedrich Hegel, 1770–1831** [20, pp 94-97] *"also dissolves the Kantian dualism. He builds an impressive theory. The basis for this theory is the assumption of a deep-seated identity between **reason** (**sense**) and **reality**: **"the reasonable is real"** and **"the real is reasonable"**. Hegel saw his understanding of this duality in the light of **history**. Hegel thus saw truth, reason and reality historically. "Modern" dialectism was born. Now two contradictory philosophies could now be both true. From this Hegel developed an impressive "apparatus": From "nothingness" via "creation", "quality", quantity" to "essence", "cause", "reality", "causality", and on to "concept", "life" and "cognition" ending with the "absolute""* ! And there we end ! We must reject Hegel's ***thesis, antithesis, synthesis***. By relativising philosophy wrt. history Hegel has removed necessity. By thus postulating that ***"it is an eternal truth that we cannot achieve eternal truths"***. Hegel's main contribution ends up in contradiction. **Friedrich Schelling, 1775–1854**, [20, pp 94] *"goes further by removing the **subject/object** distinction claiming an underlying identity between these, that is, between mind and matter: nature is the visible mind, and mind is the invisible nature. Again this attempt brings Schelling's work into contradictions"*. **Friedrich Ludwig Gottlob Frege, 1848–1925**. Although primarily a mathematician and logician, Frege contributed to Philosophy. Amongst his contributions were the distinction between ***"sinn"*** (***sense***), and ***"bedeutung"*** (***reference***). The distinction[25] is: the reference (or "referent"; bedeutung) of a proper name is the object it means or indicates (bedeuten), its sense (Sinn) is what the name expresses. The reference of a sentence is its truth value, its sense is the thought that it expresses. **Edmund Husserl, 1859–1938**, [20, pp 115-116] *"founded a school of **phenomenology**. To Husserl our conscience is characterised by **intentionality**. Cognition is an **act** which is **directed** at something. When I see, I see something. When I think, I think something. Philosophy, to Husserl, should build on this insight. It should investigate that which conscience is directed at from "within", and without prejudice of what it might be. Husserl expressed clearly the difference between **meaning** and **object**"*. But as [17, pp 115-116] shows, Husserl thereby ends up in an inconsistent theory. **Bertrand Russell, 1872–1970**, [20, pp 117-118] *"amongst very many contributions put forward a **Philosophy of Logical Atomism** [214]. It is based on the formal logic developed Russell and Whitehead in [215, Principia Mathematica]. That formal logic distinguishes between simple and complex propositions; the latter being truth functions over simple propositions. **Logical Atomism** now claims that the world must be describable by independent simple propositions. This requires that simple empirical propositions must be logically independent of one another. This again requires that the meaning of a simple empirical proposition alone must depend on a relation between the simple proposition and that which it stands for in reality. The meaning of a word is that "object" which the word "denotes". This is similar to Wittgenstein's theory. The problem is that the requirement that the simple, elementary propositions must be logically independent of one another makes it impossible to find such elementary propositions. It is therefore impossible to find those "objects" that the elementary propositions are supposed to denote. The whole of **Logical Atomism** thus builds on an erroneous extrapolation from formal logic"*. **Logical Positivism: 1920s–1936** was a "circle" if philosophers, mostly based in Vienna, cf. **Wiener Kreis**. [20, pp 119-121] *"They did not adopt Russell's **Logical Atomism**. Instead they claimed that the meaning of a sentence is its conditions for being true: i.e., a description of all facts that must be the case in order for the sentence to be judged true; that is, the **verification conditions**. But the problem here is that if the **verification conditions** are a valid **mean-***

[22] Körner, S., Kant, Penguin Books, 1990. p. 72

[23] William Henty Stanley Monck, Introduction to the Critical Philosophy. Publ. Dublin, W. McGee, 1874, p.44.

[24] See footnote 23 above.

[25] ***On Sense and Reference*** [***"Über Sinn und Bedeutung"***], Zeitschrift für Philosophie und philosophische Kritik, vol. 100 (1892), pp. 25–50

ing criterion**, then its own formulation cannot be meaningful! So logical positivism ends up in contradiction"*. Some philosophers of the Vienna Circle were **Moritz Schlick, 1882–1936; Alfred Jules Ayer, 1910–1989; Rudolf Carnap, 1891–1970** and **Otto Neurath, 1882–1945**. **Ludwig Wittgenstein, 1889–1951** was not a member of the ***Vienna Circle, but his early work was much discussed in the Circle. [20, pp 121-124] *"This work of Wittgenstein was Tractatus Logico-Philosophicus [216, 1921]. Tractatus, as did Logical Positivism, basically takes language as a departure point for a philosophical analysis of the world and our situation in it. But both these theories build on self-refusing bases. Wittgenstein understood that his Tractatus was built on a too simple **meaning theory**, i.e., a theory of how meaning is ascribed to sentences. In Philosophische Untersuchungen [217] Wittgenstein explores new directions – which have no bearing on our quest."*

7.4.8 Bertrand Russell – Again !

We bring an excerpt from Russell's [218, ***History of Western Philosophy***, Chap. XXXI: The Philosophy of Logical Analysis, pp 786–788]. The excerpt that we bring reflects Russell's thinking, around 1945, as influenced, no doubt, by developments in quantum physics. *From all this it seems to follow that events, not particles, must be the 'stuff' of physics. What has been thought of as a particle will have to be thought of as a series of events. The series of events that replaces a particle has certain important physical properties, and therefore demands our attention; but it has no more substantiality than any other series of events that we might arbitrarily single out. Thus **'matter'** is not part of the ultimate material of the world, but merely a convenient way of collecting events into bundles."*

We cannot, but point out, the "similarity" of these observations to our transcendental deduction of behaviours from parts.

•••

We have surveyed ideas of 32 philosophers – ideas relevant to our quest: that of understanding borderlines between philosophical arguments and formal, mathematical arguments as they relate to domain analysis & description. We shall now turn to elucidate these.

7.5 The Kai Sørlander Philosophy

We shall review an essence of [17, 20]. Kai Sørlander's objective [20, pp 131] *"is to investigate the philosophical question: **'what are the necessary characteristics of each and every possible world and our situation in it'**. We can reformulate this question into the task of determining the necessary logical conditions for every possible description of the world and our situation in it"*.

7.5.1 The Basis

In this section we shall mostly quote from [17]. *"The world is all that is the case. All that can be described in true propositions." "In science we investigate how the world is factually." "Philosophy puts forward another question. We ask of what could not consistently be otherwise."* [26 :1,2,3] **The Inescapable Meaning Assignment**: *"It is thus the task of philosophy to determine the inescapable characteristics of the world and our situation in it."* In determining these inescapable characteristic *"we cannot refer to our experience ... since the experience cannot tell us anything that could not consistently be otherwise." "Two demands must be satisfied by the philosophical basis. The first is that it must not be based on empirical premises. The other is that it cannot consistently be refuted by anybody under any conceivable circumstances. These demands can only be satisfied by one assumption."* We shall refer to this assumption as:

The Inescapable Meaning Assignment

- The ***The Inescapable Meaning Assignment*** is[27] the recognition of the mutual dependency between

[26] [17], : [1] pg. 13, ℓ 2–3, [2] pg. 13, ℓ 7–8, [3] pg. 13, ℓ 11–12

- ***the meaning of designations and***
- ***the consistency relations between propositions.***

As an example of what “goes into” *the inescapable meaning assignment* we bring, albeit from the world of computer science, that of the description of the ***stack*** data type (its entities and operations).

The Meaning of Designations

Stacks - A Narrative

463 Stacks, s:S, have elements, e:E;
464 the empty_S operation takes no arguments and yields a result stack;
465 the is_empty_S operation takes an argument stack and yields a Boolean value result.
466 the stack operation takes two arguments: an element and a stack and yields a result stack.
467 the unstack operation takes an non-empty argument stack and yields a stack result.
468 the top operation takes an non-empty argument stack and yields an element result.

The consistency relations:

469 an empty_S stack is_empty, and a stack with at least one element is not;
470 unstacking an argument stack, stack(e,s), results in the stack s; and
471 inquiring as to the top of a non-empty argument stack, stack(e,s), yields e.

The meaning of designations:

type
463. E, S
value
464. empty_S: **Unit** $\rightarrow$ S
465. is_empty_S: S $\rightarrow$ **Bool**
466. stack: E $\times$ S $\rightarrow$ S
467. unstack: S $\xrightarrow{\sim}$ S
468. top: S $\xrightarrow{\sim}$ E

The consistency relations:

469. is_empty(empty_S()) = **true**
469. is_empty(stack(e,s)) = **false**
470. unstack(stack(e,s)) = s
471. top(stack(e,s)) = e

Necessary and Empirical Propositions: *“That the inescapable meaning assignment is required in order to answer the question of how the world must necessarily be can be seen from the following.” “It makes it possible to distinguish between necessary and empirical propositions.” “**A proposition is necessary** if its truth value depends only on the meaning of the designators by means of which it is expressed.” “**A proposition is empirical** if its truth value does not so depend.” “An empirical proposition must therefore refer to something ... which exists independently of its designators, and it must predicate something about the thing to which it refers.”* The definition *“the world is all that is the case. All that can be described in true propositions.”*[28 :1,2,3,4,5] satisfies *the inescapable meaning assignment*. *“That which is described in `necessary` propositions is that which is common to [all] possible worlds. A concrete world is all that can be described in true `empirical` propositions.”*[29] **Primary Objects**: *“an empirical proposition must refer to an independently existing thing and must predicate something about that thing. On that basis it is then possible to deduce how those objects that can be directly referred to in simple empirical propositions must necessarily be. Those things are referred to as `primary objects`. A deduction of the `inevitable characteristics` of a possible world is thus identical to a deduction of how primary objects must necessarily be.”*[30] **Two Requirements to the Philosophical Basis**: *“Two demands have been put to the philosophical basis for our quest. It must not contain empirical preconditions; and the foundation must*

[27] [17], pg. 13-14, ℓ13-ℓ1
[28] [17], : [1] pg. 13, ℓ 16–17; [2] pg. 13, ℓ 17–18; [3] pg. 13, ℓ 20–21; [4] pg. 14, ℓ 26–30; [5] pg. 13, ℓ 2–3
[29] [17], pg.15, ℓ15-18
[30] [17], pg.15, ℓ23-30

not consistently be refuted. It must not consistently be false.” [31] **The inescapable meaning assignment:** ***‘the meaning of designations and the consistency relations between propositions’*** [32] … satisfies this basis.[33] **The Possibility of Truth**: Where Kant builds on the ***contradictory*** dichotomy of ***Das Ding an sich*** and ***Das Ding für uns***, that is, the possibility of ***self-awareness***, Kai Sørlander builds on the ***possibility of truth***: [20, pp 136] *“since the possibility of truth cannot in a consistent manner be denied we can hence assume the* **contradiction principle:** *‘a proposition and its negation cannot both be true’. We assume that the contradiction principle is a* ***necessary truth***[34]*”* **The Logical Connectives**: Sørlander now deduces the logical connectives: ***conjunction*** (‘and’ $\wedge$), ***disjunction*** (‘or’, $\vee$), and ***implication*** ($\Rightarrow$ or $\supset$). **Necessity and Possibility**: [20, pp 142] *“A* ***proposition*** *is* ***necessarily true****, if its truth follows from the definition of of the designations by means of which it is expressed; then it must be true under all circumstances. A* ***proposition*** *is* ***possibly true****, if its* ***negation*** *is not* ***necessarily true****”*. **Empirical Propositions**: An ***empirical proposition*** refers to an independently existing entities and predicates something that can be either true or false about the referenced entity. The entities that are referenced in empirical propositions have not been completely characterised by these propositions; they are simply those that can be referenced in empirical propositions.

7.5.2 Logical Conditions for Describing Physical Worlds

So which are the logical conditions of descriptions of any world ? In [17] and [20] Kai Sørlander, through a series of transcendental deductions “unravels” the following logical conditions: (i) symmetry and asymmetry (ii) transitivity and intransitivity, (iii) space: direction, distance, etc., (iv) time: before, after, in-between etc., (v) states and causality, (vi) kinematics, dynamics, etc., and (vii) Newton’s laws, et cetera. We shall summarise Sørlander’s deductions. To remind the reader: the issue is that of deducing how the ***primary entities*** must necessarily be.

Symmetry and Asymmetry

[20, pp 152] *“There can be different* ***primary entities****. Entity A is* ***different*** *from entity B if A can be ascribed a predicate in-commensurable with a predicate ascribed to B.* ***‘Different from’*** *is a* ***symmetric predicate****. If entity A is* ***identical*** *to entity B then A cannot be ascribed a predicate which is in-commensurable with any predicate that can be ascribed to B; and then B is identical to A.* ***‘Equal to’*** *is a* ***symmetric predicate****”*.

Transitivity and Intransitivity

[20, pp 148] *“If A is identical to B and B is identical to C then A is identical to C with* ***identity*** *then being a* ***transitive relation****. The relation* ***different from*** *is not transitive it is an* ***transitive relation****”*.

Space

[20, pp 154] *“The two relations* ***asymmetric*** *and* ***symmetric****, by a transcendental deduction, can be given an interpretation: The relation (spatial)* ***direction*** *is asymmetric; and the relation (spatial)* ***distance*** *is symmetric. Direction and distance can be understood as spatial relations. From these relations are derived the relation* ***in-between****. Hence we must conclude that* ***primary entities exist in space. Space*** *is therefore an unavoidable characteristic of any possible world”*. From the direction and distance relations one can derive ***Euclidean Geometry***.

[31] [17], pg. 30, ℓ 6–12
[32] [17], pg. 13-14, ℓ13-ℓ1
[33] [17], pg. 30, ℓ 16–28
[34] [20, pp 136] *“A* ***necessary truth****, on one side, follows from the meaning of the designations by means of which it is expressed, and, on the other side and at the same instance, define these designations and their mutual meaning.”*

States

[20, pp 158-159] *"We must assume that primary entities may be ascribed predicates which are not logically required. That is, they may be ascribed predicates incompatible with predicates which they actually satisfy. For it to be logically possible, that one-and-the-same* ***primary entity*** *can be ascribed incompatible predicates, is only logically possible if any primary entity can exist in different* ***state****s. A* ***primary entity*** *may be in one state where it can be ascribed one predicate, and in another state where it can be ascribed another incompatible predicate"*.

Time

[20, pp 159] *"Two such different states must necessarily be ascribed different incompatible predicates. But how can we ensure so ? Only if states stand in an asymmetric relation to one another. This state relation is also transitive. So that is an indispensable property of any world. By a transcendental deduction we say that* ***primary entities exist in time****.* ***So every possible world must exist in time****"*.

Causality

[20, pp 162-163] *"States are related by the* ***time relation****s "before" and "after". These are asymmetric and transitive relations. But how can it be so ? Propositions about primary entities at different times must necessarily be logically independent of one another. This follows from the possibility that a primary entity necessarily be ascribed different, incompatible predicates at different times. It is therefore logically impossible from the primary entities alone to deduce how a primary entity is at on time point to how it is at another time point. How, therefore, can these predicates supposedly of one and the same entity at different time points be about the* ***same entity ?*** *There can be no logical implication about this !* ***Transcendentally*** *therefore there must be a* ***non-logical implicative*** *between propositions about properties of a primary entity at different times. Such an* ***non-logical implicative*** *must depend on* ***empirical circumstances*** *subject to which the primary entity exists. There are no other circumstances. If the state on a primary entity changes then there must be changes in its "circumstances" whose consequences are that the primary entity changes state. And such "circumstance"–changes will imply primary entity state changes. We shall use the term 'cause' for a preceding "circumstance"–change that implies a state change of a primary entity. So now we can conclude* ***that every change of state of a primary entity must have a cause,*** *and* ***that "equivalent circumstances" must have "equivalent effects".*** *This form of implication is called* **causal implication**. *And the principle of implication for* **causal principle**. *So every possible world enjoys the* ***causal principle****. Kant's transcendental deduction is fundamentally built on the the* ***possibility of self-awareness****. Sørlander's transcendental deduction is fundamentally built on the* ***possibility of truth****. In Kant's thinking the* ***causal principle*** *is a prerequisite for possibility of self-awareness"*. In this way Sørlander avoids Kant's ***solipsism***, i.e., ***"that only one's own mind is sure to exist"*** a solipsism that, however, flaws Kant's otherwise great thinking.

Kinematics

[20, pp 164–165] *"So* ***primary entities*** *exist in* ***space*** *and* ***time****. They must have* ***spatial extent*** *and* ***temporal extent****. They must therefore be able to* ***change*** *their* ***spatial properties****. Both as concerns* ***form*** *and* ***location****. But a spatial change in form presupposes a change in location – as the more fundamental. A* ***primary entity*** *which changes location is said to be* ***in movement****. If a* ***primary entity*** *which does not change location is said to be* ***at rest****. The* ***velocity***[35] *of a primary entity expresses the distance and direction it moves in a given time interval. Change in velocity of a primary entity is called its* ***acceleration****. Acceleration involves either change in velocity, or change in direction of movement, or both."* So far we have reasoned us to fundamental concepts of ***kinematics***.

[35] Velocity has a ***speed*** and a ***vectorial direction***. ***Speed*** is a scalar, for example of type `kilometers per hour`. ***Vectorial direction*** is a scalar structure, for example for a spatial direction consisting of geographical elements: x `degrees North,` y `degrees East` $(x+y=90)$`, and` z `degrees Up or Down` $(0 \leq z \leq 90$`, where, if` $z=90$ `we have that both` x `and` y `are` $0)$.

Dynamics

[20, pp 165-165] *"When we "add" causality" to kinematics we obtain* **dynamics***. We can do so, because primary entities are in time. Kinematics imply that that a primary entity changes when it goes from being* **at rest** *to* **be moving.** *Likewise when it goes from movement to rest. And similarly, when it accelerates (decelerates). So a primary entity has same* **state of movement** *if it has same velocity and moves in the same direction. Primary entities change state of movement if they change velocity or direction. So, combining kinematics and the principle of causality, we can deduce that if a primary entity changes state of movement then there must be a cause, and we call that cause a* **force***"*.

Newton's Laws

Newton's First Law: [20, pp 165-166] *"Combining* **kinematics** *and the* **principle of causality***, and the therefrom deduced concept of* **force***, we can deduce that any* **change of movement** *is proportional*[36] *to the* **force***. This implies that a primary entity which is not under the influence of an external force will continue in the same state of movement – that is, be at rest or conduction a linear movement at constant velocity. This is* **Newton's First Law***"*. **Newton's Second Law:** [20, pp 166] *"That a certain, non-zero force implies change of movement, imply that the primary entity must excert a certain* **resistance** *to that change. It must have what we shall call a certain* **mass***.*[37] *From this it follows that* **the change in the state of movement of a primary entity** *not only is proportional to the excerted force, but also inversely proportional*[38] *to the mass of that entity. This is* **Newton's Second Law***"*. **Newton's Third Law:** [20, pp 166-167] *"In a possible world, the forces that affects primary entities must come from "other" primary entities. Primary entities are located in different volumes of space. Their location may interfere with one another in the sense at least of "obstructing" their mutual movements – leading to clashes. In principle we must assume that even primary entities "far away from one another" obstruct. If they clash it must be with* **oppositely directed and equal forces.** *This is* **Newton's Third Law***"*.

7.5.3 Gravitation and Quantum Mechanics

Mutual Attraction: [20, pp 167-168] *"How can primary entities possibly be the* **source** *of* **force***s that* **influence** *one another? How can primary entities at all have a* **mass**[39] *such that it requires* **force***s to change their* **state of movement***? The answer must be that primary entities excert a* **mutual influence** *on one another – that is there is a* **mutual attraction***"* **Gravitation:** [20, pp 168] *"This must be the case for all primary entities. This must mean that all primary entities can be characterised by a* **universal mutual attraction***: a* **universal gravitation** *"* **Finite Propagation – A Gravitational Constant:** [20, pp 168] *"Thus* **mutual attraction** *must* **propagate** *at a certain, finite, velocity. If that velocity was infinite, then it is everywhere and cannot therefore have its source in concretely existing primary entities. But having a finite velocity implies that there must be a* **propagational speed limit***. It must be a* **constant of nature***."*[40]
Gravitational "Pull": [20, pp 169-170] *"The nature of* **gravitational "pull"** *can be deduced, basically as follows:* Primary entities must basically consist of elements that attract one another, but which are **stable**, and that is only possible if it is, in principle, **impossible to describe these elementary particles precisely**. If there is a fundamental limit to how these basic particles can be described, then it is also precluded that they can undergo continuous change. *Hence there is a basis for stability despite*

[36] Observe that we have "only" said: ***proportional***, meaning also directly proportional, not whether it is logarithmically, or linearly, or polynomially, or exponentially, etc., so.

[37] ***Mass*** refers loosely to the amount of ***matter*** in an entity. This is in contrast to ***weight*** which refers to the ***force*** exerted on an entity by ***gravity***.

[38] Cf. Footnote 36.

[39] cf. Footnote 37 Pg. 214

[40] Let two entities have respective masses m_1 and m_2. Let the forces with which they attract each other be f_1, respectively f_2. Then the ***law of gravitation*** – *as it can be deduced by philosophical arguments* – can be expressed as $f_1 = f_2$. The specific force, expressed using Newton's constant G is $f = G \times m_1 \times m_2 \times r^{-2}$ where r is the distance between the two entities and $G = 6.674 \times 10^{-11} \times m^3 \times kg^{-1} \times s^{-2}$ [m:meter, kg:kilogam s:second] – *as derived by physicists.*

mutual attraction. There must be a foundational limit for how precise these descriptions can be. which implies that the elementary particle as a whole can be described statistically" **Quantum Mechanics:** The rest is physics: unification of quantum mechanics and Einstein's special relativity has been done; unification of gravitation with Einstein's general theory of relativity is still to be done. **A Summary:** [20, pp 170-173] *"Philosophy lends to physics its results a necessity that physics cannot give them. Experiments have shown that Einstein's results – with propagation limits – indeed hold for this world. Philosophy shows that every possible world is subject to a fixed propagation limit. Philosophy also shows that for a possible world to exist it must be built from elementary particles which cannot be individually described (with Newton's theory) "*

7.5.4 The Logical Conditions for Describing Living Species

Purpose, Life and Evolution

Causality of Purpose: [20, pp 174] *"If there is to be* ***the possibility of language and meaning*** *then there must exist primary entities which are* ***not entirely encapsulated within the physical conditions****; that they are stable and can influence one another. This is only possible if such primary entities are subject to a* ***supplementary causality directed at the future****: a* ***causality of purpose****"* **Living Species:** [20, pp 174-175] *"These primary entities are here called* ***living species****. What can be deduced about them ? They must* ***have some form they can be*** **developed** ***to reach;*** *and which* ***they must be causally determined to maintain.*** *This development and maintenance must further in* ***an exchange of matter with an environment****. . . . It must be possible that living species occur in one of two forms: one form which is characterised by* ***development****,* ***form*** *and* ***exchange****, and another form which, additionally, can be characterised by the ability to* ***purposeful movement****. The first we call* ***plant****s, the second we call* ***animal****s"* **Animate Entities:** [20, pp 176] *"For an animal to purposefully move around there must be "additional conditions" for such self-movements to be in accordance with the principle of causality: they must have* ***sensory organ****s sensing among others the immediate purpose of its movement; they must have* ***means of motion*** *so that it can move; and they must have* ***instinct****s,* ***incentive****s and* ***feeling****s as causal conditions that what it senses can drive it to movements"* And all of this in accordance with the laws of physics. **Animal Structure:** [20, pp 177-178] *"Animals, to possess these three kinds of "additional conditions", must be built from special units which have an inner relation to their function as a whole: their* ***purposefulness*** *must be built into their physical building units; that is, as we can now say, their* ***genome****s; that is, animals are built from genomes which give them the* ***inner determination*** *to such building blocks for* ***instinct****s,* ***incentive****s and* ***feeling****s. Similar kinds of deduction can be carried out with respect to plants. Transcendentally one can deduce* ***basic principles of evolution*** *but not its details"*

Consciousness, Learning and Language

Consciousness and Learning: [20, pp 180-181] *"The existence of animals is a necessary condition for there being language and meaning in any world. That there can be* ***language*** *means that animals are capable of* ***developing language****. And this must presuppose that animals can* ***learn from their experience.*** *To learn implies that animals can* ***feel*** *pleasure and distaste and can* ***learn****. . . . One can therefore deduce that animals must possess such building blocks whose inner determination is a basis for learning and consciousness "* **Language:** [20, pp 181-182] *"Animals with higher social interaction uses* ***sign****s, eventually developing a* ***language****. These languages adhere to the same system of defined concepts which are a prerequisite for any description of any world: namely the system that philosophy lays bare from a basis of transcendental deductions and the* ***principle of contradiction*** *and its* ***implicit meaning theory****"*

7.5.5 Humans, Knowledge, Responsibility

Humans: [20, pp 184] *"A* ***human*** *is an animal which has a* ***language****"* **Knowledge:** [20, pp 184] *"Humans must be* ***conscious*** *of having* ***knowledge*** *of its concrete situation, and as such that humans can have knowledge about what they feel, and eventually that humans can know whether what they feel is true or false.*

*Consequently **humans can describe their situation correctly*"** **Responsibility:** [20, pp 184] *"In this way one can deduce that humans can thus have **memory** and hence can have **responsibility**, be **responsible**. Further deductions lead us into **ethics**"*

7.5.6 An Augmented Upper Ontology

We now augment our upper-ontology, to include ***living species***, from that of Fig. 1.1 Pg. 13 to that of Fig. 7.1 Pg. 216. We leave it to the reader to "fill in the details !"

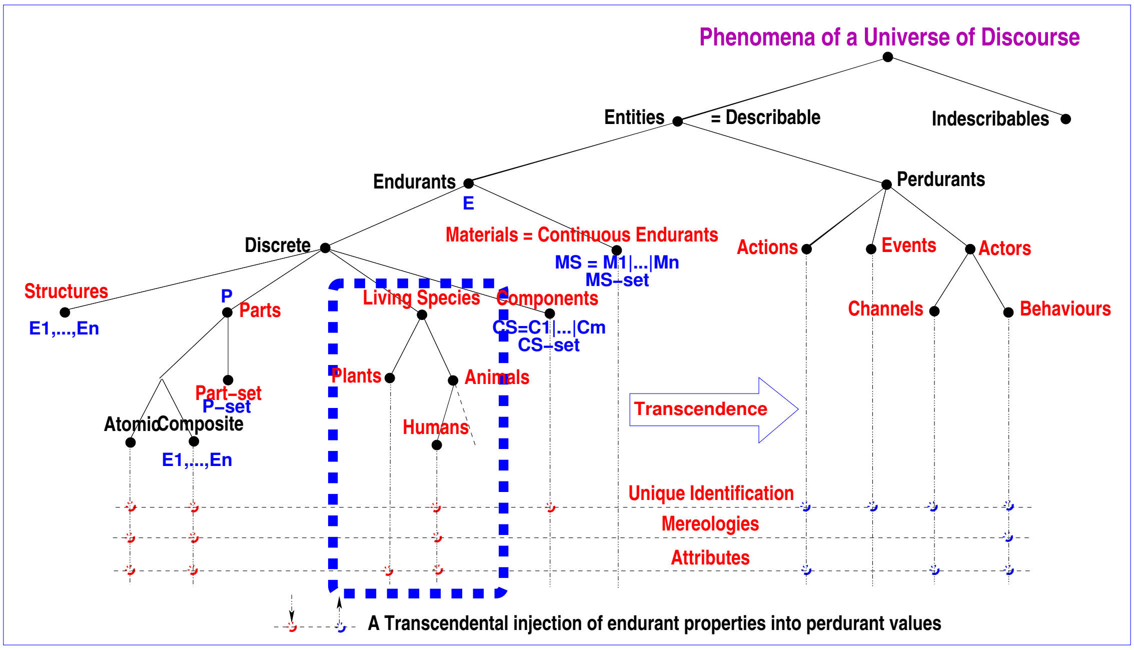


Fig. 7.1. An Upper Ontology for Domains – with **Living Species**

7.5.7 Artifacts: Man-made Entities

By an ***artifact*** we shall understand a ***man-made entity***: usually an ***endurant*** in ***space***, one that satisfies the laws of physics, and sometimes one that, by a ***transcendental deduction***, can take on the rôle of a ***perdurant***; but the artifact can also, for example, by ***intended*** as a ***piece of art***, something for our enjoyment and reflection.
We then augment our upper-ontology, to include ***artifacts***, from that of Fig. 7.1 Pg. 216 to that of Fig. 7.2 Pg. 217. We leave it to the reader to "fill in the details !"

7.5.8 Intentionality

We have ended our presentation of Sørlander's Philosophy. Before going into justifications of our ***domain analysis & description calculi*** with respect to this philosophy we shall briefly comment on the concept of ***intentionality***.

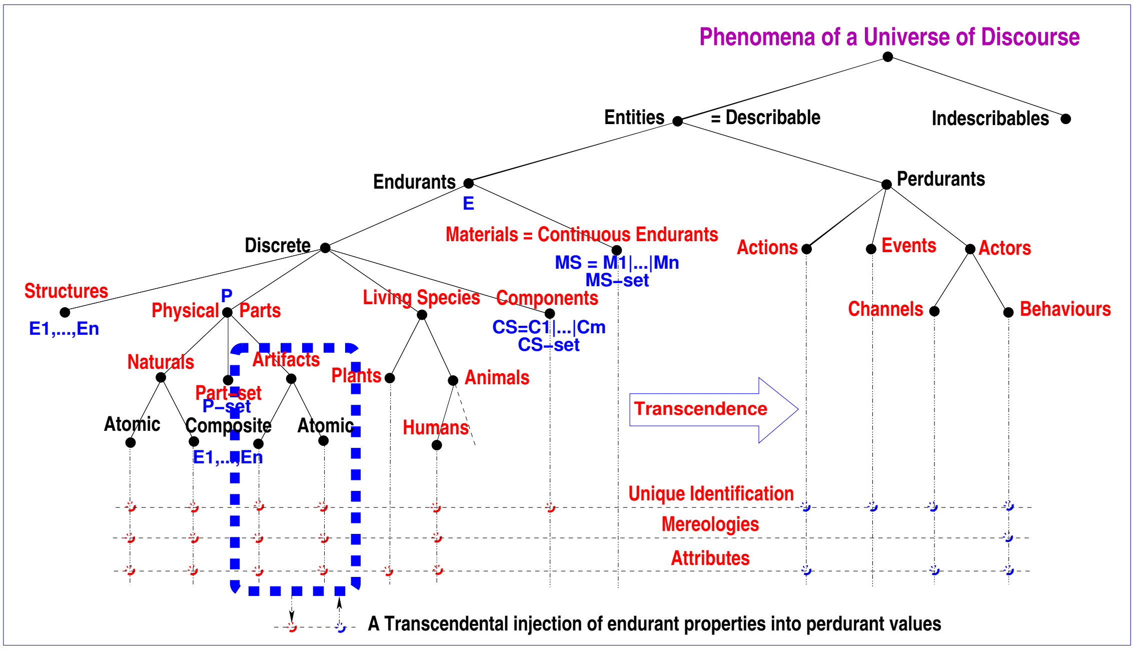


Fig. 7.2. An Upper Ontology Extended with **Artifacts**

Intentionality *is a philosophical concept and is defined by the* `Stanford Encyclopedia of Philosophy`[41] *as* **"the power of minds to be about, to represent, or to stand for, things, properties and states of affairs."** *The puzzles of intentionality lie at the interface between the philosophy of mind and the philosophy of language. The word itself, which is of medieval Scholastic origin, was rehabilitated by the philosopher Franz Brentano towards the end of the nineteenth century. and adopted by Edmund Husserl. 'Intentionality' is a philosopher's word. It derives from the Latin word* intentio, *which in turn derives from the verb intendere, which means being directed towards some goal or thing. The earliest theory of intentionality is associated with St. Anselm's ontological argument for the existence of God, and with his tenets distinguishing between objects that exist in the understanding and objects that exist in reality.*

We shall here endow the concept of 'intentionality' with the following interpretation. Man-made artifacts are made for specific purposes. Often two or more artifacts are intended to serve a purpose, that is, to represent an intent. We speculate as follows:

Definition 67 **On Intentional Pull:** Two or more artifactual parts of different sorts, but with overlapping sets of intents may excert an ***intentional "pull"*** on one another ■

This ***intentional "pull"*** may take many forms. Let p_x:X and p_y:Y be two parts of ***different sorts*** (X,Y), and with ***common intent***, ι. ***Manifestations*** of these, their common intent must somehow be ***subject to constraints***, and these must be ***expressed predicatively***.

We return, in Sect. 7.6.1 on Page 220, with an ***example of*** what we claim to be an ***intentional "pull"***, that is, Example 7.6 on Page 221.

7.6 Philosophical Issues of The Domain Calculi

We now interpret the ***domain analysis & description analysis calculus*** of Chapter 1 in the light of Sørlander's ***Philosophy*** of Sect. 7.5.

[41] Jacob, P. (Aug 31, 2010). ***Intentionality***. `Stanford Encyclopedia of Philosophy` (`https://seop.illc.uva.nl/entries/intentionality/`) October 15, 2014, retrieved April 3, 2018.

We re-examine all analysis calculus prompts with references to their prompt number or the section – and the page on which their definition is given.

7.6.1 The Analysis Calculus Prompts

External Qualities

- Item 1, pp. 13: `is_universe_of_discourse:` After a rough sketch narrative of the contemplated domain, the informal justification to be given for this query should be along these lines: the chosen universe-of-discourse is one that can be described in true propositions; that is, one that is based in ***space*** and ***time***; subject to ***Laws of Newton***; etc., and, indispensably so, involves ***persons*** with ***language***, ***responsibility*** and ***intents.***
- Item 2, pp. 14: `is_entity:` So entities are just that: describable, based in either space (as are endurants) or in both space and time (as are perdurants), and involving persons. That is, entities are the "stuff" that philosophy cares about in its quest to understand the world. What lies outside may be in the realm of superstition, "mumbo-jumbo", et cetera; "things" that are neither in space nor time; figments of the mind.
- Item 3, pp. 15: `is_endurant:` An endurant is an entity which we characterise in propositions without reference to (actual, i.e. "real") time. There is no notion of state changes in describing entities. Endurants are either based in physics or based in living species: plants and animals including persons, or are artifacts which build on endurants. Endurants are, in the words of Whitehead, [219], ***continuants***.
- Item 4, pp. 15: `is_perdurant:` And, consequently, a perdurant is an entity which we characterise in propositions with more-or-less explicit reference to (actual, i.e. "real") time, focusing on state-changes and/or interaction between perdurants. Perdurants are either ***action***s or ***event***s or ***behaviour***s. **Definition**: ***Behaviours*** are defined as sets of sequences of actions, events and behaviours ■ Philosophical treatments are given of the notions of ***time*** in [45, 46, 47, 48], [discrete] ***action***s in [53], ***event***s in [54, 55, 56, 57, 58, 59, 60, 61, 62, 63], and ***behaviour***s in, for example, the Internet based articles on `plato.stanford.edu/entries/behaviorism/` and `www.behavior.org/search.php?q=behavior+and+philosophy`. Most of the literature on behaviours focus on psychological aspects which we consider outside the realm of our form of domain analysis & description,
The interplay between endurants and perdurants is studied in [81].
- Item 5, pp. 16: `is_discrete:` [We re-emphasize that the notion of ***discreteness*** of ***endurants*** such as we "need" it here, is not related to the notion of ***discreteness*** in physics or mathematics.] The terms ***separate, individual*** and ***distinct*** characterise ***discreteness***. It is up to the ***domain analysis & description scientist cum engineer*** to decide whether en entity should be characterised as primarily distinguished by these 'qualities' – or not.
- Item 6, pp. 16: `is_continuous:` [We re-emphasize that the notion of ***continuity*** of ***endurants*** such as we "need" it here, is not related to the notion of ***continuity*** in physics or mathematics.] The terms: ***prolonged, without interruption,*** and ***unbroken series or pattern*** characterise ***continuity*** of endurants. It is up to the ***domain analysis & description scientist cum engineer*** to decide whether en entity should be characterised as primarily distinguished by these 'qualities', or not.
- Item 9, pp. 17: `is_structure:` Whether a discrete endurant is considered a ***structure***, or a ***part***, or ***a set of components*** is a ***pragmatic*** decision. So has no bearings in the Sørlander Philosophy outside its possible bearings in language where the notion of language can be motivated philosophically.
- Item 10, pp. 18: `is_part`; Item 17, pp. 21: `has_components`; and Item 18, pp. 21: `has_materials:` All entities, whether non-living species, including artifactual, or living species (plants and animals, incl. humans) are subject to **the inescapable meaning assignment**, the ***principle of contradiction*** and its ***implicit meaning theory***. They are also subject to the notions of ***space*** and ***time*** and to the ***Laws of Newton***, etc. The living species entities are ***additionally*** subject to ***causality of purpose*** with humans having ***language***, ***memory*** and ***responsibility***. These notions can be assumed, but we do not, at present, i.e., in this report, suggest any means of modelling language, memory and responsibility. Following Sørlander's Philosophy there are the (atomic, see below) part p living species: **is**_LIVE_SPECIES(p), of which there are plants, **is**_PLANT(p), and there are animals, **is**_ANIMAL(p),

of which (latter) some are humans, **is**_HUMAN(p), and some are not; and there are the non-living-species parts, p, of which some are made by man (or by other artifacts), **is**_ARTIFACT(p), and some are not, we refer to them as ***physical part***s. We therefore now, as a consequence of Sørlander's Philosophy, suggest the domain analysis prompts: **is**_LIVE_SPECIES, **is**_PLANT, **is**_ANIMAL, **is**_HUMAN and **is**_ARTIFACT.

All this means that the Sørlander Philosophy, in a sense, mandates us to introduce the following ***new analysis prompts:***

Analysis Prompt 39 `is_physical:` *The domain analyser analyses discrete endurants (d) into physical parts:*

- `is_physical` *– where* `is_physical(d)` *holds if d is a physical part* ■

Analysis Prompt 40 `is_living:` *The domain analyser analyses discrete endurants (d) into living species:*

- `is_living` *– where* `is_living(d)` *holds if θ is a living species.* ■

Analysis Prompt 41 `is_natural:` *The domain analyser analyses physical parts (p) into natural:*

- `is_natural` *– where* `is_natural(p)` *holds if p is a natural part* ■

Analysis Prompt 42 `is_artifactual:` *The domain analyser analyses physical parts (p) into artifactual physical parts:*

- `is_artifactual` *– where* `is_artifactual(p)` *holds if p is a man-made part* ■

Analysis Prompt 43 `is_plant:` *The domain analyser analyses living species (ℓ) into plants:*

- `is_plant` *– where* `is_plant(ℓ)` *holds if ℓ is a plant* ■

Analysis Prompt 44 `is_animal:` *The domain analyser analyses living species (ℓ) into animals:*

- `is_animal` *– where* `is_animal(ℓ)` *holds if ℓ is an animal* ■

Analysis Prompt 45 `is_human:` *The domain analyser analyses animals (α) into humans:*

- `is_human` *– where* `is_human(α)` *holds if α is a human* ■

Analysis prompts, `is_XXX`, similar to `is_human`, can be devised for other animal species.

- Item 11, pp. 19: `is_atomic:` and Item 12, pp. 19: `is_composite:` The notion of atomicity here has nothing to do with that of the the Greeks [Demokrit, pp. 205]. Here it is a rather pragmatic issue, void, it seems, of philosophical challenge. It is a purely pragmatic issue with respect to any chose domain whether the domain scientist cum engineer decides to analyse & describe a part into being atomic or composite.

  Example 7.3. **Automobile: Atomic or Composite**:Thus, ***for example***, you the reader may consider your automobile as atomic, whereas your mechanic undoubtedly considers it composite ■

Unique Identifiers

Sect. 1.5.1, pp. 26–27: `unique identifiers:`

Uniqueness of entities follows from the basic logic of symmetry etc. Uniqueness or rather ***identity***, is an thus important philosophical notion [cf. Sect. 7.5.2 on Page 212]. Notice that we are not concerned with any representation of unique part and component identifiers. So please, dear reader, do not speculate on that ! The uniqueness of part or component identifiers "follows" the part and component, irrespective of the spatial location and time of the possibly "movable" part or component, i.e., irrespective of its state !

Mereology

Sect. 1.5.2, pp. 27–29: `mereology:`

There are some new aspects of the concept of mereology – which, in light of the Sørlander Philosophy, were not considered in Chapter 1, and which it is now high time to consider, and, for some of these aspects, ***to include in the* domain analysis & description*.***

- **Philosophy:** Mereology, such as we use it, derives from ***Stanisław Leśniewski***, Polish mathematician, logician, philosopher (1886–1939) [220, 221, 222, 223, 224, 225]. Wikipedia presents an overview of aspects of mereology.[42]. Related to our "use" of the concept of mereology are the studies of Henry S. Leonard and Nelson Goodman [173, 226, 227, 1940–2008], Bowman L. Clarke [228, 229, 1981–1985], Douglass T. Ross [230, 1976], Mario Bunge [231, 232, 1977–1979], Peter Simons [233, 1987], Barry Smith [234, 235, 236, 174, 237, 238, 1993–2004] and Roberto Casati and Achille C. Varzi [239, 240, 38, 1993–1999].
- **Topologies and Intents:** To us mereology, in light of Sørlander's Philosophy, now becomes either of two relations (or possibly both): (i) spatial relations, as for ***Stanisław Leśniewski*** and the cited references, and (ii) ***intensional*** relations. We characterise the latter as follows:

  Definition 68 Intentional Relations: By an ***intensional relation*** we shall understand a relation between distinct endurants which manifests two (or more) ***designation***s and at least one ***meaning*** ■

  Example 7.4. **Transport**:Automobiles and roads, i.e. hubs and links, have distinct sorts and designations, but share the ***intent*** (***meaning***) of technologically ***supporting traffic*** ■

  We refer to [3, ***Domain Facets: Analysis & Description***].
- **Part Mereologies:** Thus the mereology of parts shall be sought in either their topological, i.e., spatial, arrangements, or their intents – with parts of same intent being mereologically related, or possibly some combination of both.

  Example 7.5. **Traffic**: Hence, in reference to the example of Chapter 1's Sect. 1.8, we have that the mereologies of each automobile include the set of unique identifiers of all hubs and links, and the mereologies of each hub and link include the set of unique identifiers of all automobiles ■
- **Further Studies:** It appears that the concept of mereology, in light of Sørlander's Philosophy, warrants further scrutiny, philosophically well as from the point of view of domain analysis & description. Should discrete endurants be further analysed into structures, parts and components, as now, and ***natural discrete endurants*** or ***artifact discrete endurants*** or should discrete endurants have attribute values of ***natural discrete endurant values*** or ***artifact discrete endurant values***.

Attributes

Sect. 1.5.3, pp. 29–34: `attributes:`

Attributes, their type and value, are the main means for ***expressing propositions about primary entities.***[43] Let us first recall: ***parts*** and ***components*** have unique identifiers, ***parts*** have mereologies and ***parts*** and ***materials*** have attributes. Let us also "remember" that these differences are purely pragmatic. All endurants are subject to being in ***space*** and ***time***, and being subject to the ***principle of causality***. Three sets of attributes follow from the Sørlander's Philosophy: (i) attributes of non-life-specifies entities; (ii) attributes of life-specifies entities, but additionally subject to ***purpose***, ***language***, ***responsibility***, and ***causality of principle***; and those (iii) attributes that are additional and more individually determined by the kind of the part. We shall now summarise these.

Non-Species Parts: These are the parts that were actually treated in Chapter 1. To them, as a consequence of Sørlander's Philosophy, one can ascribe the following attribute observers: **attr_**$\mathbb{SPACE}$ and **attr_**$\mathbb{TIME}$. No explanation seems necessary here. Attribute observers related to the above could be:

[42] https://en.wikipedia.org/wiki/Mereology#Metaphysics

[43] ***The world is all that is the case. All that can be described in true propositions.*** [17, pp.13, ℓ 2–3]

attr_LOCATION where the ***location*** to be yielded is some spatial point within the space yielded by the SPACE observer. **attr_VOLUME** where the ***volume*** is the volume (in some units) of the space yielded by the SPACE observer. **attr_MASS**(p) where the ***mass*** is the mass (in some units) of the part p. Et cetera. We leave it to the reader to "think up" Boolean and other algebraic operators over time, space, location, mass, etc.

Artifacts: To remind, ***artifacts*** are parts made by man and/or other artifacts. They have all the same attributes (i.e. attribute observers) as has non-species parts. In addition they may have such attribute observes as **attr_Intent**, **attr**_Maker, **attr**_Brand_Name, **attr**_Production_Year, **attr**_Owner, **attr**_Purchase_Price, **attr**_Current_Value and **attr**_Condition. The idea of the **attr_Intent** attribute observer is to yield a token that somehow identifies the ***purpose*** of the artifact: `transport`, `"measurement-of-this"`, `"measurement-of-that"`, `"food-stuff"`, etc. We leave it to the reader to figure out the idea of the other attributes. **Artifactual Intents:** In the world of physics, since Isaac Newton, the mutual attraction of bodies (with mass) and in the context of gravitation leads to the **gravitational pull**, cf. Sect. 7.5.3 pp. 214. Now, in the context of artifactual parts with intents we may speak of **intentional "pull"**.

Definition 69 **Intentional "Pull":** Two or more artifactual parts of different sorts, but with overlapping sets of intents may excert an ***intentional "pull"*** on one another ■

This ***intentional "pull"*** may take many forms. Let $p_x : X$ and $p_y : Y$ be two parts of ***different sorts*** (X,Y), and with ***common intent***, ι. ***Manifestations*** of these, their common intent must somehow be ***subject to constraints***, and these must be ***expressed predicatively***.

Example 7.6. **Automobile and Road Transport**: For the main example, Chapter 1, Sect. 1.8;

472 ***automobiles*** shall now include the intent of `'transport'`,
473 and so shall ***hubs*** and ***links***.

472 **attr**_Intent: A → (`'transport'`|...)-**set**
473 **attr**_Intent: H → (`'transport'`|...)-**set**
473 **attr**_Intent: L → (`'transport'`|...)-**set**

Manifestations of `'transport'` is reflected in ***automobiles*** having the automobile position attribute, APos, Item 86 Pg. 50, ***hubs*** having the ***hub traffic*** attribute, H_Traffic, Item 73 Pg. 49, and in ***links*** having the ***link traffic*** attribute, L_Traffic, Item 77 Pg. 49.

474 Seen from the point of view of an automobile there is its own traffic history, which is a (time ordered) sequence of timed automobile positions;
73 seen from the point of view of a hub there is its own traffic history, H_Traffic Item 73 Page 49, which is a (time ordered) sequence of timed maps from automobile identities into automobile positions; and
77 seen from the point of view of a link there is its own traffic history, L_Traffic Item 77 Page 49, which is a (time ordered) sequence of timed maps from automobile identities into automobile positions.

The ***intentional "pull"*** of these manifestations is this:

475 The union, i.e. proper merge of all automobile traffic histories, AllATH, must now be identical to the same proper merge of all hub, AllHTH, and all link traffic histories, AllLTH.

type
474 A_Hist = $(\mathscr{T} \times \text{APos})^*$
73, pp.49 H_Traffic = (A_UI|B_UI) $\overrightarrow{m}$ $(\mathscr{T} \times \text{APos})^*$
77, pp.49 L_Traffic = (A_UI|B_UI) $\overrightarrow{m}$ $(\mathscr{T} \times \text{APos})^*$
475 AllATH = $\mathscr{T}$ $\overrightarrow{m}$ (AUI $\overrightarrow{m}$ APos)
475 AllHTH = $\mathscr{T}$ $\overrightarrow{m}$ (AUI $\overrightarrow{m}$ APos)
475 AllLTH = $\mathscr{T}$ $\overrightarrow{m}$ (AUI $\overrightarrow{m}$ APos)
axiom
475 **let** allA = proper_merge_into_AllATH({(a,attr_A_Hist(a)∩ias)|a:A•a ∈ as}),
475 allH = proper_merge_into_AllHTH({attr_H_Traffic(h)∩ias|h:H•h ∈ hs}),
475 allL = proper_merge_into_AllLTH({attr_L_Traffic(l)∩ias|l:L•h ∈ ls}) **in**
475 allA = H_and_L_Traffic_merge(allH,allL) **end**

We leave the definition of the four merge functions to the reader !

We now discuss the concept of ***intentional "pull"***. We endow each automobile with its history of timed positions and each hub and link with their histories of timed automobile positions. These histories are facts ! They are not something that is laboriously recorded, where such recordings may be imprecise or cumbersome[44]. The facts are there, so we can (but may not necessarily) talk about these histories as facts. It is in that sense that the purpose (`'transport'`) for which man let automobiles, hubs and link be made with their `'transport'` intent are subject to an **intentional "pull"**. **It can be no other way: if automobiles "record" their history, then hubs and links must together "record" identically the same history !** ■

We have tentatively proposed a concept of ***intentional "pull"***. That proposal is in the form, I think, of a transcendental deduction; it has to be further studied.

Humans[45]**:** ***Humans*** have ***sensory organs*** and ***means of motion***; ***inner determination*** for ***instincts***, ***incentives*** and ***feelings***; ***purpose***; and ***language***; and can ***learn***[46]. We leave it, to the reader, as a ***research topic***: to suggest means for expressing analysis prompts that cover these kinds of attributes.

For this compendium we have little to say on the issue of ***humans***. Rather much more work has to be done for any meaningful writing. So, here is a challenge to the readers !

A Summary of Domain Analysis Prompts

7.6.2 The Description Calculus Prompts

MORE TO COME

- Item ??, pp. ??: `observe_universe_of_discourse:`
- Item 1, pp. 23: `observe_endurant_sorts:`
- Item 2, pp. 23: `observe_part_type:`
- Item 3, pp. 25: `observe_component_sorts:`
- Item 4, pp. 25: `observe_material_sorts:`
- Item 5, pp. 26: `observe_unique_identifier:`
- Item 6, pp. 28: `observe_mereology:`
- Item 7, pp. 30: `observe_attributes:`

MORE TO COME

A Summary of Domain Description Prompts

MORE TO COME

[44] or thought technologically in-feasible – at least some decades ago!
[45] We focus on humans, but the discussion can be "repeated", in modified form, for plants and animals in general.
[46] cf. Sect. 7.5.4 on Page 215

MORE TO COME

7.6.3 The Behaviour Schemata

TO BE WRITTEN

7.6.4 Wrapping Up

We summarise the above in a revision of the ***ontology diagram*** first given in Fig. 1.1 Pg. 13 and used, in more-or-less that form, in several publications: [2, 1, 5, 241]. The revision is shown in Fig. 7.3:

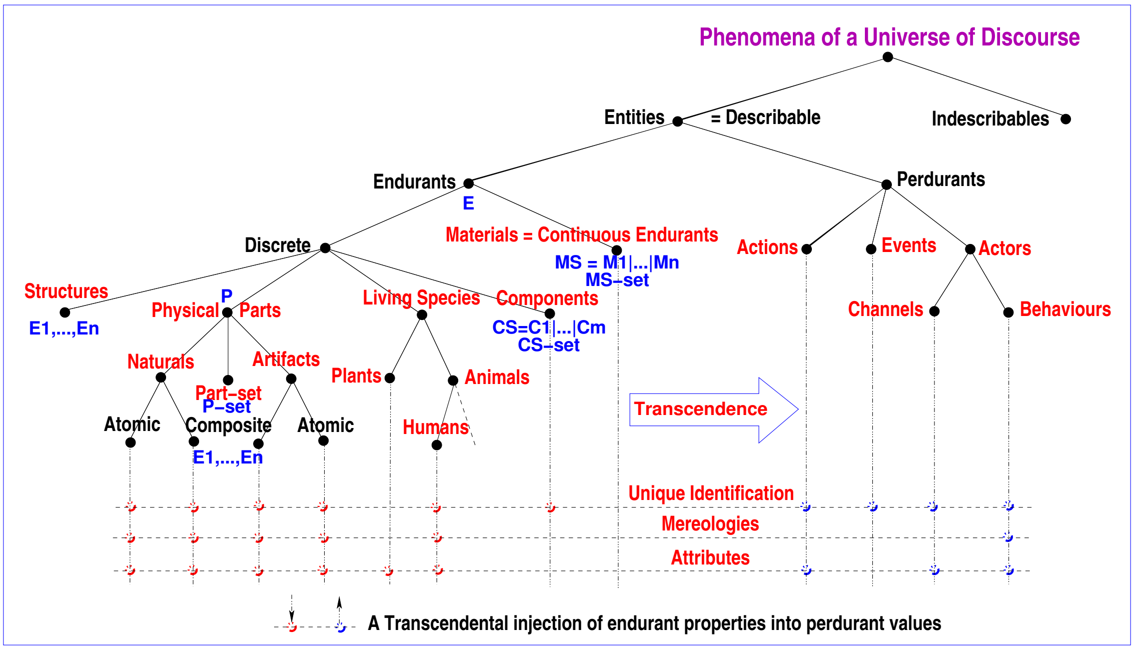


Fig. 7.3. A Revised Upper Ontology for Domains

Figure 7.3 emphasises the analytic, "upper" structure of domains and emphasises endurants: **Black** names attached to diagram nodes designate "upper" categories of entities. **Red** names similarly attached designate manifest categories of entities. **Blue** names also so attached are the sort names of values of manifest endurants. Both naturals and artifacts have atomic and composite values. We only hint (⋱) at other (than human) animal species. The lower dashed horizontal lines with pairs of **-o- - -o-** hint at the internal endurant qualities that are "transferred"

7.6.5 Discussion

Review of Revisions

We have related a number of the **domain analysis & description**'s analysis prompts to Sørlander's Philosophy – and have found that a number of corrections has to be made to the understanding of these:

the basis for ***unique identifiers*** and the categories of endurants and attributes. With [2] endurants came in three forms: ***structures***, ***parts*** (atomic and composite), and ***materials***. Now we must ***refine*** the notion of parts into: ***physical parts*** (as assumed in [2]), ***artifactual parts*** and ***living species parts***. We must further articulate the notion of attributes: as before, for ***physical parts***, to necessarily include the in-avoidable classical physics attributes[47] and be subject to the ***principle of causality*** and ***gravitational pull***; but now additionally also to ***artifactual parts***, still subject to the attributes of physical parts but now additionally subject to additional in-avoidable attributes such as ***intent*** and to both ***gravitational pull*** and ***intentional "pull"***; and to ***living species parts***, notably, in this compendium, ***humans*** with their attributes.

General

It is only of interest to study the **domain analysis & description** ***analysis calculus*** with respect to Sørlander's Philosophy. The corresponding ***description calculus*** and schemata are not analytic. They represent our "response" to the domain analysis. So our "quest" has ended. It is time to "sum up".

Although there is obviously a lot more to study we stop here, for a while, to wrap up this compendium. With what we have presented we can, however, make several conclusions – and that will now be done !

7.7 Conclusion

7.7.1 General Remarks

When I have informed my colleagues of this work their reactions have been mixed. *Oh yes, philosophy, yes, I referred to Plato in one of my papers, ages ago !*, or – *does it relate to the recent Facebook scandal ?*, and other such deeply committing and understanding uttering. Philosophy is actually hard. Anyone can claim to reflect philosophically, and many do, and some even refer, in their newspaper columns, to being philosophers, but it does take some practice to actually do philosophy. Good schooling, up to senior high, is required. Having learned to reason, in classical disciplines like mathematics and physics; being able to read in two or more foreign languages; having learned history, real history, for us, in the Western world, from before the ancient Greeks, and on-wards; these seems to be prerequisites for a serious study of philosophy.

In grammar school I passed the little test in Greek and the "large" test in Latin at the age of 14–15. I had wonderful teachers. I learned about the ***history of ideas*** from Johs. Sløk [242]. My university did not offer courses in philosophy. Over the years I acquired many [and browsed some additional] philosophy books: Karl Jaspers [243], Bertrand Russell [244, 245, 218], [Alfred North Whitehead [246, 219, 247],] Willard van Orme Quine [248, 249, 250], [Martin Heidegger [45],] Ludwig Johan Josef Wittgenstein [251, 217], Karl Popper [252, 253, 254, 255, 256, 257], Imre Lakatos [258], David Favrholdt [259, 260], John Sowa [261], as well as some dictionaries: [43, 42, 262, 44, Cambridge, Oxford, Blackwell] and [263]. In this century I started looking at a number of epistemological essays: [264, Logic and Ontology], [231, 232, 236, 265, 266, Objects], [233, 234, 235, 267, 238, Ontology], [268, 53, 57, Actions], [54, 55, 59, 269, 61, 63, 62, 58, 57, Events], [221, 222, 228, 229, 173, 239, 240, 174, 62, 38, Mereology], [270, 271, 272, 273, Qualities, Properties] and [56, SpaceTime]. But although wonderful "reads", it was not until Sørlander's [17, 18, 201, 19, 274, 275, 202, 20] that philosophy really started meaning something. ***'Philosophy is useless'*** it is said. ***' "Results" of philosophy are not meant to solve problems '***, it is said. But Sørlander's Philosophy, [17, 20], have definitely helped shape the ***domain analysis & description analysis calculus*** into a form that makes it rather definitive !

Before my study of Kai Sørlander's Philosophy the upper ontology – like shown in Fig. 1.1 Pg. 13 – was based on empirical observations.

After my study the upper ontology – now shown in Fig. 7.2 Pg. 217 – is based on philosophical reasoning and is definite, is unavoidable !

[47] ***space, time, mass, velocity,*** etc.

7.7.2 Revisions to the Calculi and Further Studies

Yes, our study of Sørlander's Philosophy, [17, 20], has led to the following modifications of the ***domain analysis & description analysis calculus***: (i) a more refined view of ***discrete endurants***; (ii) "refinements" of ***attributes*** need be studied further; (iii) the ***intentional "pull"*** between ***artifactual parts*** need be studied further; and (iv) the ***transcendental deduction*** that "translates" ***endurants*** into ***behaviours*** need be studied further see, however, below.

(i) Refined View of Discrete Endurants: Where ***discrete endurants*** before were (i.1) ***parts*** and (i.2) ***components***, they are now (i.1a) ***physical***, (i.2) ***components***, (i.3) ***live species parts*** and (i.1b) ***artifacts***. of which the ***live species parts*** are (i.3a) ***plants*** and (i.3b) ***animals***, (i.3c) for which latter we focus on ***humans***,

(iv) Which Endurants are Candidates for Perdurancy ? (iv.1) **Naturals:** It seems that if we only focus on transcendentally deducing ***natural endurants*** into behaviours then we are really studying or doing **physics:** ***mechanics***, ***chemistry***, ***electricity***, et cetera. (iv.2) **Living Species:** It seems that if we only focus on transcendentally deducing (iv.2.1) ***living species*** into behaviours then we are really studying or doing **life sciences:** ***botanics***, ***zoology***, ***biology***, et cetera. (iv.2.2) or if we just focus on ***humans***, then we are really studying or doing **behavioral sciences**. (iv.3) **Artifacts:** (iv.3.1) We have seen that it makes sense to "transmogrify" many artifacts into behaviours. But how characterise those for which that deduction makes, or does not make sense ? (iv.3.2) It seems that if we only focus on transcendentally deducing ***artifacts*** into behaviours then we are really studying or doing **engineering:** ***mechanical***, ***chemical***, ***electrical***, ***electronics***, et cetera, engineering.

7.7.3 Remarks on Classes of Artifactual Perdurants

We can rather immediately identify the following "classes" of ***artifactual perdurants***:

- **Computerised Command & Control Systems:** Here we have several, i.e. more than just a few distinct artifacts, interacting with human operators for the purpose of command, monitoring and controlling some of these artifacts and humans. Examples are ***pipelines*** [30] and ***swarms of drones*** [36].
- **Logistics: Planning & Monitoring:** Here again we have several, i.e. more than just a few distinct artifacts, but the emphasis is on operational planning and the monitoring of plan fulfillment. Examples are ***container lines*** [27] and ***railways*** [24, 25, 26, 121, 122].
- **Monitoring:** Usually the systems here are just monitoring a single endurant. Examples are ***weather forecast*** [33] and ***health care***.
- **Mechanics:** Here we are dealing with the operation of just one artifact: a ***lathe*** a ***machine saw***, etc., an ***automobile***, et cetera.
- **The "End" Result:** Here we are dealing with computers being the artifacts – "final" instruments in achieving some purpose ! Examples are ***urban planning*** [35] ***stock exchange*** [28] ***credit card system*** [34] ***documents*** [29] ***Web systems*** [32] ***E-market*** [31]

We refer to [12] for a discussion of domain models as a basis for software demos, software simulators, software monitoring and software monitoring and control.

7.7.4 Acknowledgements

First and foremost I acknowledge the deep inspiration drawn from the study of Sørlander's Philosophy, notably [201] and [202]. Several people have commented, in various more-or-less spurious ways, not knowing really, what I was up to, when I informed them of my current study and writing on "applying" Sørlander's Philosophy, notably [201] and [202] to my work on domain analysis & description. Several of these comments, however uncommitted, have, however – strangely enough, upon reflection, helped me to even better grasp what it was I was trying to unravel. Let my acknowledgments to them remain anonymous.

7.8 Bibliographical Notes

We list a number of reports all of which document descriptions of domains. These descriptions were carried out in order to research and develop the domain analysis and description concepts now summarised in the present paper. These reports ought now be revised, some slightly, others less so, so as to follow all of the prescriptions of the current paper. Except where a URL is given in full, please prefix the web reference with: `http://www2.compute.dtu.dk/~dibj/`.

1 *A Railway Systems Domain:* `racosy/domains.ps` (2003)
2 *Models of IT Security:* `it-security.pdf` (2006)
3 *A Container Line Industry Domain:* `container-paper.pdf` (2007)
4 *The "Market": Buyers, Sellers, Traders:* `themarket.pdf` (2007)
5 *What is Logistics ?:* `logistics.pdf` (2009)
6 *A Domain Model of Oil Pipelines:* `pipeline.pdf` (2009)
7 *Transport Systems:* `comet/comet1.pdf` (2010)
8 *The Tokyo Stock Exchange:* `todai/tse-1.pdf` and `todai/tse-2.pdf` (2010)
9 *On Development of Web-based Software:* `wfdftp.pdf` (2010)
10 *A Credit Card System:* `/2016/uppsala/accs.pdf` (2016)
11 *Documents:* `/2017/docs.pdf` (2017)
12 *A Context for Swarms of Drones:* `/2016/uppsala/accs.pdf` (2017)
13 *A Framework for Urban Planning:* `/2018/urban-planning.pdf`urban (2018)
http://www.imm.dtu.dk/ dibj/2017/urban-planning.pdf

Part V

Summing Up

8

Conclusion

8.1 The Thesis

The ***thesis*** is this:

8.2 What Has Been Achieved ?

8.3 Future Work

8.4 The Beauty of Informatics

To create domain descriptions, or requirements prescriptions, or software designs, properly, at least such as this author sees it, is a joy to behold. The beauty of carefully selected and balanced abstractions, their interplay with other such, the relations between phases, stages and steps, and many more conceptual constructions make software engineering possibly the most challenging intellectual pursuit today. For this and more consult [14, 15, 16].

9

Bibliography

1. Dines Bjørner. A Domain Analysis & Description Method – Principles, Techniques and Modeling Languages. Research Note based on [2], Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, February 20 2018. http://www.imm.dtu.dk/ dibj/2018/tosem/Bjorner-TOSEM.pdf.
2. Dines Bjørner. Manifest Domains: Analysis & Description. *Formal Aspects of Computing*, 29(2):175–225, Online: July 2016.
3. Dines Bjørner. Domain Facets: Analysis & Description. May 2018. Extensive revision of [4]. http://-www.imm.dtu.dk/˜dibj/2016/facets/faoc-facets.pdf.
4. Dines Bjørner. Domain Engineering. In Paul Boca and Jonathan Bowen, editors, *Formal Methods: State of the Art and New Directions*, Eds. Paul Boca and Jonathan Bowen, pages 1–42, London, UK, 2010. Springer.
5. Dines Bjørner. Domain Analysis and Description – Formal Models of Processes and Prompts. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, 2016. Extensive revision of [6]. http://www.imm.dtu.dk/˜dibj/2016/process/process-p.pdf.
6. Dines Bjørner. Domain Analysis: Endurants – An Analysis & Description Process Model. In Shusaku Iida, José Meseguer, and Kazuhiro Ogata, editors, *Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi*. Springer, May 2014.
7. Dines Bjørner. To Every Manifest Domain a CSP Expression — A Rôle for Mereology in Computer Science. *Journal of Logical and Algebraic Methods in Programming*, (94):91–108, January 2018.
8. Dines Bjørner. On Mereologies in Computing Science. In *Festschrift: Reflections on the Work of C.A.R. Hoare*, History of Computing (eds. Cliff B. Jones, A.W. Roscoe and Kenneth R. Wood), pages 47–70, London, UK, 2009. Springer.
9. Dines Bjørner. From Domain Descriptions to Requirements Prescriptions – A Different Approach to Requirements Engineering. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, 2016. Extensive revision of [10].
10. Dines Bjørner. From Domains to Requirements. In *Montanari Festschrift*, volume 5065 of *Lecture Notes in Computer Science (eds. Pierpaolo Degano, Rocco De Nicola and José Meseguer)*, pages 1–30, Heidelberg, May 2008. Springer.
11. Dines Bjørner. Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, 2016. Extensive revision of [12]. http://www.imm.dtu.dk/˜dibj/2016/demos/faoc-demo.pdf.
12. Dines Bjørner. Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions. In *Rainbow of Computer Science, Festschrift for Hermann Maurer on the Occasion of His 70th Anniversary.*, Festschrift (eds. C. Calude, G. Rozenberg and A. Saloma), pages 167–183. Springer, Heidelberg, Germany, January 2011.
13. Dines Bjørner. A Philosophy of Domain Science & Engineering – An Interpretation of Kai Sørlander's Philosophy. Research Note, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, Spring 2018. http://www.imm.dtu.dk/˜dibj/2018/philosophy/filo.pdf.
14. Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. See [276, 277].
15. Dines Bjørner. *Software Engineering, Vol. 2: Specification of Systems and Languages*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. Chapters 12–14 are primarily authored by Christian Krog Madsen. See [278, 279].

16. Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. See [280, 281].
17. Kai Sørlander. *Det Uomgængelige – Filosofiske Deduktioner [The Inevitable – Philosophical Deductions, with a foreword by Georg Henrik von Wright]*. Munksgaard · Rosinante, 1994. 168 pages.
18. Kai Sørlander. *Under Evighedens Synsvinkel [Under the viewpoint of eternity]*. Munksgaard · Rosinante, 1997. 200 pages.
19. Kai Sørlander. *Den Endegyldige Sandhed [The Final Truth]*. Rosinante, 2002. 187 pages.
20. Kai Sørlander. *Indføring i Filosofien [Introduction to The Philosophy]*. Informations Forlag, 2016. 233 pages.
21. David Crystal. *The Cambridge Encyclopedia of Language*. Cambridge University Press, 1987, 1988.
22. Chris W. George, Peter Haff, Klaus Havelund, Anne Elisabeth Haxthausen, Robert Milne, Claus Bendix Nielsen, Søren Prehn, and Kim Ritter Wagner. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.
23. C.A.R. Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985. Published electronically: http://www.usingcsp.com/cspbook.pdf (2004).
24. Dines Bjørner. Formal Software Techniques in Railway Systems. In Eckehard Schnieder, editor, *9th IFAC Symposium on Control in Transportation Systems*, pages 1–12, Technical University, Braunschweig, Germany, 13–15 June 2000. VDI/VDE-Gesellschaft Mess– und Automatisieringstechnik, VDI-Gesellschaft für Fahrzeug– und Verkehrstechnik. Invited talk.
25. Dines Bjørner, Chris W. George, and Søren Prehn. Computing Systems for Railways — A Rôle for Domain Engineering. Relations to Requirements Engineering and Software for Control Applications. In *Integrated Design and Process Technology. Editors: Bernd Kraemer and John C. Petterson*, P.O.Box 1299, Grand View, Texas 76050-1299, USA, 24–28 June 2002. Society for Design and Process Science. Extended version[1].
26. Dines Bjørner. Dynamics of Railway Nets: On an Interface between Automatic Control and Software Engineering. In *CTS2003: 10th IFAC Symposium on Control in Transportation Systems*, Oxford, UK, August 4-6 2003. Elsevier Science Ltd. Symposium held at Tokyo, Japan. Editors: S. Tsugawa and M. Aoki. Final version[2].
27. Dines Bjørner. A Container Line Industry Domain. Techn. report, Fredsvej 11, DK-2840 Holte, Denmark, June 2007. Extensive Draft[3].
28. Dines Bjørner. The Tokyo Stock Exchange Trading Rules. R&D Experiment, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, January and February, 2010. Version 1[4], Version 2[5].
29. Dines Bjørner. What are Documents? Research Note, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, July 2017. http://www.imm.dtu.dk/˜dibj/2017/docs/docs.pdf.
30. Dines Bjørner. Pipelines – a Domain Description[6]. Experimental Research Report 2013-2, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013.
31. Dines Bjørner. Domain Models of "The Market" — in Preparation for E–Transaction Systems. In *Practical Foundations of Business and System Specifications (Eds.: Haim Kilov and Ken Baclawski)*, The Netherlands, December 2002. Kluwer Academic Press. Final draft version[7].
32. Dines Bjørner. On Development of Web-based Software: A Divertimento of Ideas and Suggestions. Technical, Technical University of Vienna, August–October 2010. http://www.imm.dtu.dk/˜dibj/wfdftp.pdf.
33. Dines Bjørner. Weather Information Systems: Towards a Domain Description. Technical Report: Experimental Research, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, November 2016. http://www.imm.dtu.dk/˜dibj/2016/wis/wis-p.pdf.
34. Dines Bjørner. A Credit Card System: Uppsala Draft. Technical Report: Experimental Research, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, November 2016. http://www.imm.dtu.dk/˜dibj/2016/credit/accs.pdf.
35. Dines Bjørner. Urban Planning Processes. Research Note, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, July 2017. http://www.imm.dtu.dk/˜dibj/2017/up/urban-planning.pdf.
36. Dines Bjørner. A Space of Swarms of Drones. Research Note, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, November–December 2017. http://www.imm.dtu.dk/˜dibj/2017/docs/-docs.pdf.

[1] http://www2.imm.dtu.dk/ db/pasadena-25.pdf
[2] http://www2.imm.dtu.dk/ db/ifac-dynamics.pdf
[3] http://www2.imm.dtu.dk/ db/container-paper.pdf
[4] http://www2.imm.dtu.dk/ db/todai/tse-1.pdf
[5] http://www2.imm.dtu.dk/ db/todai/tse-2.pdf
[6] http://www.imm.dtu.dk/˜dibj/pipe-p.pdf
[7] http://www2.imm.dtu.dk/ db/themarket.pdf

37. W. Little, H.W. Fowler, J. Coulson, and C.T. Onions. *The Shorter Oxford English Dictionary on Historical Principles*. Clarendon Press, Oxford, England, 1973, 1987. Two vols.
38. Roberto Casati and Achille C. Varzi. *Parts and Places: the structures of spatial representation*. MIT Press, 1999.
39. Dines Bjørner. *A Rôle for Mereology in Domain Science and Engineering*. Synthese Library (eds. Claudio Calosi and Pierluigi Graziani). Springer, Amsterdam, The Netherlands, October 2014.
40. Dines Bjørner. Domain Analysis: Endurants – An Analysis & Description Process Model. In Shusaku Iida, José Meseguer, and Kazuhiro Ogata, editors, *Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi*. Springer, May 2014.
41. Michael A. Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. ACM Press. Addison-Wesley, Reading, England, 1995.
42. Ted Honderich. *The Oxford Companion to Philosophy*. Oxford University Press, Walton St., Oxford OX2 6DP, England, 1995.
43. Rober Audi. *The Cambridge Dictionary of Philosophy*. Cambridge University Press, The Pitt Building, Trumpington Street, Cambridge CB2 1RP, England, 1995.
44. Nicholas Bunnin and E.P. Tsui-James, editors. *The Blackwell Companion to Philosophy*. Blackwell Companions to Philosophy. Blackwell Publishers, 108 Cowley Road, Oxford OX4 1JF, UK, 1996.
45. Martin Heidegger. *Sein und Zeit (Being and Time)*. Oxford University Press, 1927, 1962.
46. David John Farmer. *Being in time: The nature of time in light of McTaggart's paradox*. University Press of America, Lanham, Maryland, 1990. 223 pages.
47. Wayne D. Blizard. A Formal Theory of Objects, Space and Time. *The Journal of Symbolic Logic*, 55(1):74–89, March 1990.
48. Johan van Benthem. *The Logic of Time*, volume 156 of *Synthese Library: Studies in Epistemology, Logic, Methhodology, and Philosophy of Science (Editor: Jaakko Hintika)*. Kluwer Academic Publishers, P.O.Box 17, NL 3300 AA Dordrecht, The Netherlands, second edition, 1983, 1991.
49. Leslie Lamport. *Specifying Systems*. Addison–Wesley, Boston, Mass., USA, 2002.
50. Chao Chen Zhou and Michael R. Hansen. *Duration Calculus: A Formal Approach to Real–time Systems*. Monographs in Theoretical Computer Science. An EATCS Series. Springer–Verlag, 2004.
51. Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
52. Carlo A. Furia, Dino Mandrioli, Angelo Morzenti, and Matteo Rossi. *Modeling Time in Computing*. Monographs in Theoretical Computer Science. Springer, 2012.
53. George Wilson and Samuel Shpall. Action. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2012 edition, 2012.
54. F. Dretske. Can Events Move? *Mind*, 76(479-492), 1967. Reprinted in [61, 1996], pp. 415-428.
55. A. Quinton. Objects and Events. *Mind*, 88:197–214, 1979.
56. D.H. Mellor. Things and Causes in Spacetime. *British Journal for the Philosophy of Science*, 31:282–288, 1980.
57. Donald Davidson. *Essays on Actions and Events*. Oxford University Press, 1980.
58. P.M.S. Hacker. Events and Objects in Space and Time. *Mind*, 91:1–19, 1982. reprinted in [61], pp. 429-447.
59. Alain Badiou. *Being and Event*. Continuum, 2005. (Lêtre et l'événements, Edition du Seuil, 1988).
60. Jaegwon Kim. *Supervenience and Mind*. Cambridge University Press, 1993.
61. Roberto Casati and Achille C. Varzi, editors. *Events*. Ashgate Publishing Group – Dartmouth Publishing Co. Ltd., Wey Court East, Union Road, Farnham, Surrey, GU9 7PT, United Kingdom, 23 March 1996.
62. Chia-Yi Tony Pi. *Mereology in Event Semantics*. Phd, McGill University, Montreal, Canada, August 1999.
63. Roberto Casati and Achille Varzi. Events. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2010 edition, 2010.
64. Chris W. George, Anne Elisabeth Haxthausen, Steven Hughes, Robert Milne, Søren Prehn, and Jan Storbank Pedersen. *The RAISE Development Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.
65. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-10114-9.
66. Jean-Raymond Abrial. The B Book: Assigning Programs to Meanings *and* Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge, England, 1996 and 2009.
67. Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.
68. Dines Bjørner and Cliff B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.

69. John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. ISBN 0-521-62348-0.
70. J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.
71. Kokichi Futatsugi and Ataru Nakagawa. An overview of CAFE specification environment - an algebraic approach for creating, verifying, and maintaining formal specifications over networks. In *Proc. of 1st International Conference on Formal Engineering Methods (ICFEM '97), November 12-14, 1997, Hiroshima, JAPAN*, pages 170–182. IEEE, 1997.
72. Kokichi Futatsugi, Daniel Gâlinâ, and Kazuhiro Ogata. Principles of proof scores in CafeOBJ. *Theor. Comput. Science*, 464:90–112, 2012.
73. Dines Bjørner, Chris W. George, Anne Eliabeth Haxthausen, Christian Krog Madsen, Steffen Holmslykke, and Martin Pěnička. "UML"–ising Formal Techniques. In *INT 2004: Third International Workshop on Integration of Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*, pages 423–450. Springer–Verlag, 28 March 2004, ETAPS, Barcelona, Spain. Final Version[8].
74. Wolfang Reisig. *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*. Leitfäden der Informatik. Vieweg+Teubner, 1st edition, 15 June 2010. 248 pages; ISBN 978-3-8348-1290-2.
75. ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992, 1996, 1999.
76. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
77. V. Richard Benjamins and Dieter Fensel. The Ontological Engineering Initiative (KA)2. Internet publication + Formal Ontology in Information Systems, University of Amsterdam, SWI, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands and University of Karlsruhe, AIFB, 76128 Karlsruhe, Germany, 1998. http://www.aifb.uni-karlsruhe.de/WBS/broker/KA2.htm.
78. H. Wang, J. S. Dong, and J. Sun. Reasoning Support for Semantic Web Ontology Family Languages Using Alloy. *International Journal of Multiagent and Grid Systems, IOS Press*, 2(4):455–471, 2006.
79. Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. . Cambrige University Press, January 2003. 570 pages, 14 tables, 53 figures; ISBN: 0521781760.
80. Franz Baader, Ian Horrocks, and Ulrike Sattler. Description Logics as Ontology Languages for the Semantic Web. In Dieter Hutter and Werner Stephan, editors, *Mechanizing Mathematical Reasoning*, pages 228–248. Springer, Heidelberg, 2005.
81. Thomas Bittner, Maureen Donnelly, and Barry Smith. Endurants and Perdurants in Directly Depicting Ontologies. *AI Communications*, 17(4):247–258, December 2004. IOS Press, in [282].
82. Ingvar Johansson. Qualities, Quantities, and the Endurant-Perdurant Distinction in Top-Level Ontologies. In K.D. Althoff, A. Dengel, R. Bergmann, M. Nick, and Th. Roth-Berghofer, editors, *Professional Knowledge Management WM 2005*, volume 3782 of *Lecture Notes in Artificial Intelligence*, pages 543–550. Springer, 2005. 3rd Biennial Conference, Kaiserslautern, Germany, April 10-13, 2005, Revised Selected Papers.
83. Merriam Webster Staff. Online Dictionary: `http://www.m-w.com/home.htm`, 2004. Merriam–Webster, Inc., 47 Federal Street, P.O. Box 281, Springfield, MA 01102, USA.
84. Staff of Encyclopædia Brittanica. Encyclopædia Brittanica. Merriam Webster/Brittanica: Access over the Web: http://www.eb.com:180/, 1999.
85. Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning about Knowledge*. The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts 02142, 1996. 2nd printing.
86. Edward A. Feigenbaum and Pamela McCorduck. *The fifth generation*. Addison-Wesley, Reading, MA, USA, 1st ed. edition, 1983.
87. Dines Bjørner and Jørgen Fischer Nilsson. Algorithmic & Knowledge Based Methods — Do they "Unify" ? In *International Conference on Fifth Generation Computer Systems: FGCS'92*, pages 191–198. ICOT, June 1–5 1992.
88. C. Bachman. Data structure diagrams. *Data Base, Journal of ACM SIGBDP*, 1(2), 1969.
89. E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.
90. Peter P. Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Trans. Database Syst*, 1(1):9–36, 1976.

[8] http://www.imm.dtu.dk/ db/fmuml.pdf

91. James M. Neighbors. The Draco Approach to Constructing Software from Reusable Components. *IEEE Transactions of Software Engineering*, SE-10(5), September 1984.
92. Rubén Prieto-Díaz. Domain Analysis for Reusability. In *COMPSAC 87*. ACM Press, 1987.
93. Rubén Prieto-Díaz. Domain analysis: an introduction. *Software Engineering Notes*, 15(2):47–54, 1990.
94. Rubén Prieto-Díaz and Guillermo Arrango. *Domain Analysis and Software Systems Modelling*. IEEE Computer Society Press, 1991.
95. Martin Fowler. *Domain Specific Languages*. Signature Series. Addison Wesley, October 20120.
96. Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, December 2005.
97. Diomidis Spinellis. Notable design patterns for domain specific languages. *Journal of Systems and Software*, 56(1):91–99, February 2001.
98. A.E. Haxthausen, J. Peleska, and S. Kinder. A formal approach for the construction and and verification of railway control systems. *Formal Aspects of Computing*, 23:191–219, 2011.
99. T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer, Dordrecht, 2002.
100. K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S Peterson. FODA: Feature-Oriented Domain Analysis. Feasibility Study CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, November 1990. http://www.sei.cmu.edu/library/abstracts/reports/90tr021.cfm.
101. Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, 2000.
102. Dines Bjørner. Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions. In *Rainbow of Computer Science, Festschrift for Hermann Maurer on the Occasion of His 70th Anniversary.*, Festschrift (eds. C. Calude, G. Rozenberg and A. Saloma), pages 167–183. Springer, Heidelberg, Germany, January 2011.
103. Michael A. Jackson. *Problem Frames — Analyzing and Structuring Software Development Problems*. ACM Press, Pearson Education. Addison-Wesley, England, 2001.
104. Michael A. Jackson. Program Verification and System Dependability. In Paul Boca and Jonathan Bowen, editors, *Formal Methods: State of the Art and New Directions*, pages 43–78, London, UK, 2010. Springer.
105. Will Tracz. Domain-specific software architecture (DSSA) frequently asked questions (FAQ). *Software Engineering Notes*, 19(2):52–56, 1994.
106. Erik Mettala and Marc H. Graham. The Domain Specific Software Architecture Program. Project Report CMU/SEI-92-SR-009, Software Engineering Institute Carnegie Mellon University Pittsburgh, Pennsylvania 15213, June 1992.
107. Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
108. Neno Medvidovic and Edward Colbert. Domain-Specific Software Architectures (DSSA). Power Point Presentation, found on The Internet, Absolute Software Corp., Inc.: `Abs[S/W]`, 5 March 2004.
109. Dan Haywood. *Domain-Driven Design Using Naked Objects*. The Pragmatic Bookshelf (an imprint of 'The Pragmatic Programmers, LLC.'), http://pragprog.com/, 2009.
110. Grady Booch, Jim Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
111. Jim Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
112. Ivar Jacobson, Grady Booch, and Jim Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
113. Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.
114. Søren Lauesen. *Software Requirements - Styles and Techniques*. Addison-Wesley, UK, 2002.
115. Merlin Dorfman and Richard H. Thayer, editors. *Software Requirements Engineering*. IEEE Computer Society Press, 1997.
116. Shari Lawrence Pfleeger. *Software Engineering, Theory and Practice*. Prentice–Hall, 2nd edition, 2001.
117. Roger S. Pressman. *Software Engineering, A Practitioner's Approach*. International Edition, Computer Science Series. McGraw–Hill, 5th edition, 1981–2001.
118. Ian Sommerville. *Software Engineering*. Pearson, 8th edition, 2006.
119. Carl A. Gunter, Elsa L. Gunter, Michael A. Jackson, and Pamela Zave. A Reference Model for Requirements and Specifications. *IEEE Software*, 17(3):37–43, May–June 2000.
120. Edward N. Zalta. The Stanford Encyclopedia of Philosophy. 2016. Principal Editor: https://plato.stanford.edu/.

121. Martin Pěnička, Albena Kirilova Strupchanska, and Dines Bjørner. Train Maintenance Routing. In *FORMS'2003: Symposium on Formal Methods for Railway Operation and Control Systems*. L'Harmattan Hongrie, 15–16 May 2003. Conf. held at Techn.Univ. of Budapest, Hungary. Editors: G. Tarnai and E. Schnieder, Germany. Final version[9].
122. Albena Kirilova Strupchanska, Martin Pěnička, and Dines Bjørner. Railway Staff Rostering. In *FORMS2003: Symposium on Formal Methods for Railway Operation and Control Systems*. L'Harmattan Hongrie, 15–16 May 2003. Conf. held at Techn.Univ. of Budapest, Hungary. Editors: G. Tarnai and E. Schnieder, Germany. Final version[10].
123. Dines Bjørner. Road Transportation – a Domain Description[11]. Experimental Research Report 2013-4, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013.
124. Dines Bjørner. Software Systems Engineering — From Domain Analysis to Requirements Capture: An Air Traffic Control Example. In *2nd Asia-Pacific Software Engineering Conference (APSEC '95)*. IEEE Computer Society, 6–9 December 1995. Brisbane, Queensland, Australia.
125. Dines Bjørner. *Domain Engineering: Technology Management, Research and Engineering*. A JAIST Press Research Monograph # 4, 536 pages, March 2009.
126. K. Futatsugi, A.T. Nakagawa, and T. Tamai, editors. *CAFE: An Industrial–Strength Algebraic Formal Method*, Sara Burgerhartstraat 25, P.O. Box 211, NL–1000 AE Amsterdam, The Netherlands, 2000. Elsevier. Proceedings from an April 1998 Symposium, Numazu, Japan.
127. David Harel and Rami Marelly. *Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
128. Ernst Rüdiger Olderog, Anders Peter Ravn, and Rafael Wisniewski. Linking Discrete and Continuous Models, Applied to Traffic Maneuvers. In Jonathan Bowen, Michael Hinchey, and Ernst Rüdiger Olderog, editors, *BCS FACS – ProCoS Workshop on Provably Correct Systems*, Lecture Notes in Computer Science. Springer, 2016.
129. Tom Maibaum. Conservative Extensions, Interpretations Between Theories and All That. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development*, volume 1214 of *LNCS*, pages 40–66, 1997.
130. Cliff B. Jones, Ian Hayes, and Michael A. Jackson. Deriving Specfications for Systems That Are Connected to the Physical World. In Cliff Jones, Zhiming Liu, and James Woodcock, editors, *Formal Methods and Hybrid Real-Time Systems: Essays in Honour of Dines Bjørner and Zhou Chaochen on the Occasion of Their 70th Birthdays*, volume 4700 of *Lecture Notes in Computer Science*, pages 364–390. Springer, 2007.
131. Ernst-Rüdiger Olderog and Henning Dierks. *Real-Time Systems: Formal Specification and Automatic Verification*. Cambridge University Press, UK, 2008.
132. Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, August 2003. ISBN 0521825830.
133. F. Van der Rhee, H.R. Van Nauta Lemke, and J.G. Dukman. Knowledge based fuzzy control of systems. *IEEE Trans. Autom. Control*, 35(2):148–155, February 1990.
134. David R. Christiansen, Klaus Grue, Henning Niss, Peter Sestoft, and Kristján S. Sigtryggsson. Actulus Modeling Language - An actuarial programming language for life insurance and pensions. Technical Report, http://www.edlund.dk/sites/default/files/Downloads/paper_actulus-modeling-language.pdf, Edlund A/S, Denmark, Bjerregårds Sidevej 4, DK-2500 Valby. (+45) 36 15 06 30. edlund@edlund.dk, http://www.edlund.dk/en/insights/scientific-papers. This paper illustrates how the design of pension and life insurance products, and their administration, reserve calculations, and audit, can be based on a common formal notation. The notation is human-readable and machine-processable, and specialised to the actuarial domain, achieving great expressive power combined with ease of use and safety.
135. Jaco W. de Bakker. *Control Flow Semantics*. The MIT Press, Cambridge, Mass., USA, 1995.
136. C.A. Gunther. *Semantics of Programming Languages*. The MIT Press, Cambridge, Mass., USA, 1992.
137. John C. Reynolds. *The Semantics of Programming Languages*. Cambridge University Press, 1999.
138. David A. Schmidt. *Denotational Semantics: a Methodology for Language Development*. Allyn & Bacon, 1986.
139. Robert Tennent. *The Semantics of Programming Languages*. Prentice–Hall Intl., 1997.
140. G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, Cambridge, Mass., USA, 1993.
141. Dines Bjørner. *Software Engineering, Vol. 2: Specification of Systems and Languages*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. Chapters 12–14 are primarily authored by Christian Krog Madsen.

[9] http://www2.imm.dtu.dk/ db/martin.pdf
[10] http://www2.imm.dtu.dk/ db/albena.pdf
[11] http://www.imm.dtu.dk/˜dibj/road-p.pdf

142. Yochai Benkler. Coase's Penguin, or Linux and the Nature of the Firm. *The Yale Law Journal*, 112, 2002.
143. Alapan Arnab and Andrew Hutchison. Fairer Usage Contracts for DRM. In *Proceedings of the Fifth ACM Workshop on Digital Rights Management (DRM'05)*, pages 65–74, Alexandria, Virginia, USA, Nov 2005.
144. C. N. Chong, R. J. Corin, J. M. Doumen, S. Etalle, P. H. Hartel, Y. W. Law, and A. Tokmakoff. LicenseScript: a logical language for digital rights management. *Annals of telecommunications special issue on Information systems security*, 2006.
145. C. N. Chong, S. Etalle, and P. H. Hartel. Comparing Logic-based and XML-based Rights Expression Languages. In *Confederated Int. Workshops: On The Move to Meaningful Internet Systems (OTM)*, number 2889 in LNCS, pages 779–792, Catania, Sicily, Italy, 2003. Springer.
146. Cheun Ngen Chong, Ricardo Corin, and Sandro Etalle. LicenseScript: A novel digital rights languages and its semantics. In *Proc. of the Third International Conference WEB Delivering of Music (WEDELMUSIC'03)*, pages 122–129. IEEE Computer Society Press, 2003.
147. ContentGuard Inc. XrML: Extensible rights Markup Language. http://www.xrml.org, 2000.
148. C.E.C. Digital Rights: Background, Systems, Assessment. Commission of The European Communities, Staff Working Paper, 2002. Brussels, 14.02.2002, SEC(2002) 197.
149. Carl A. Gunter, Stephen T. Weeks, and Andrew K. Wright. Models and Languages for Digtial Rights. In *Proc. of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)*, pages 4034–4038, Maui, Hawaii, USA, January 2001. IEEE Computer Society Press.
150. Joseph Y. Halpern and Vicky Weissman. A Formal Foundation for XrML. In *Proc. of the 17th IEEE Computer Security Foundations Workshop (CSFW'04)*, 2004.
151. IPR Systems Pty Ltd. Open Digital Rights Language (ODRL). http://odrl.net, 2001.
152. D. Mulligan and A. Burstein. Implementing copyright limitations in rights expression languages. In *Proc. of 2002 ACM Workshop on Digital Rights Management*, volume 2696 of *Lecture Notes in Computer Science*, pages 137–154. Springer-Verlag, 2002.
153. S. Michiels, K. Verslype, W. Joosen, and B. De Decker. Towards a Software Architecture for DRM. In *Proceedings of the Fifth ACM Workshop on Digital Rights Management (DRM'05)*, pages 65–74, Alexandria, Virginia, USA, Nov 2005.
154. Gordon E. Lyon. Information Technology: A Quick-Reference List of Organizations and Standards for Digital Rights Management. NIST Special Publication 500-241, National Institute of Standards and Technology, Technology Administration, U.S. Department of Commerce, Oct 2002.
155. R.H. Koenen, J. Lacy, M. Mackay, and S. Mitchell. The long march to interoperable digital rights management. *Proceedings of the IEEE*, 92(6):883–897, June 2004.
156. Pamela Samuelson. Digital rights management {and, or, vs.} the law. *Communications of ACM*, 46(4):41–45, Apr 2003.
157. Riccardo Pucella and Vicky Weissman. A Formal Foundation for ODRL. In *Proc. of the Workshop on Issues in the Theory of Security (WIST'04)*, 2004.
158. Riccardo Pucella and Vicky Weissman. A Logic for Reasoning about Digital Rights. In *Proc. of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, pages 282–294. IEEE Computer Society Press, 2002.
159. Open Mobile Alliance. OMA DRM V2.0 Candidate Enabler. http://www.openmobilealliance.org/-release_program/drm_v2_0.html, Sep 2005.
160. Deirdre K. Mulligan, John Han, and Aaron J. Burstein. How DRM-Based Content Delivery Systems Disrupt Expectations of "Personal Use". In *Proc. of The 3rd International Workshop on Digital Rights Management*, pages 77–89, Washington DC, USA, Oct 2003. ACM.
161. JianWen Xiang and Dines Bjørner. The Electronic Media Industry: A Domain Analysis and a License Language. Technical note, JAIST, School of Information Science, 1-1, Asahidai, Tatsunokuchi, Nomi, Ishikawa, Japan 923-1292, Summer 2006.
162. K. Araki et al., editors. *IFM 1999–2013: Integrated Formal Methods*, LNCS Vols. 1945, 2335, 2999, 3771, 4591, 5423, 6496, 7321 and 7940. Springer, 1999–2013.
163. Dines Bjørner. Manifest Domains: Analysis & Description – A Philosophical Basis. , 2016–2017. http://-www.imm.dtu.dk/~dibj/2016/apb/daad-apb.pdf.
164. Leon Osterweil. Software processes are software too. In *Proceedings of the 9th International Conference on Software Engineering*, ICSE '87, pages 2–13, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.
165. Ralph-Johan Back, Abo Akademi, J. von Wright, and F. B. Schneider. *Refinement Calculus: A Systematic Introduction*. Springer–Verlag New York, Inc., 1998.
166. C. Carroll Morgan. *Programming from Specifications*. International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK, 1990.
167. John R. Searle. *Speech Act*. CUP, 1969.

168. John Longshaw Austin. *How To Do Things With Words*. Oxford University Press, second edition, 1976.
169. Dines Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics Part II of II: The Science Part*. *Kibernetika i sistemny analiz*, (2):100–120, May 2011.
170. Dines Bjørner. *A Rôle for Mereology in Domain Science and Engineering*. Synthese Library (eds. Claudio Calosi and Pierluigi Graziani). Springer, Amsterdam, The Netherlands, May 2014.
171. John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machines, Part I. *Communications of the ACM*, 3(4):184–195, 1960.
172. John McCarthy. Towards a Mathematical Science of Computation. In C.M. Popplewell, editor, *IFIP World Congress Proceedings*, pages 21–28, 1962.
173. Henry S. Leonard and Nelson Goodman. The Calculus of Individuals and its Uses. *Journal of Symbolic Logic*, 5:45–44, 1940.
174. Barry Smith. Mereotopology: A Theory of Parts and Boundaries. *Data and Knowledge Engineering*, 20:287–303, 1996.
175. Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis — Mathematical Foundations*. Springer-Verlag, January 1999.
176. Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Sci. Comput. Program.*, 20(1-2):3–50, April 1993.
177. ESA. Global Navigation Satellite Systems. Web, European Space Agency. http://en.wikipedia.org/wiki/Satellite_navigation.
178. Dines Bjørner. Domain Engineering: A "Radical Innovation" for Systems and Software Engineering ? In *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, Heidelberg, October 7–11 2003. Springer–Verlag. The Zohar Manna International Conference, Taormina, Sicily 29 June – 4 July 2003. [12].
179. Dines Bjørner. Michael Jackson's Problem Frames: Domains, Requirements and Design. In Li ShaoYang and Michael Hinchley, editors, *ICFEM'97: International Conference on Formal Engineering Methods*, Los Alamitos, November 12–14 1997. IEEE Computer Society. Final Version[13].
180. Dines Bjørner. Domain Theory: Practice and Theories, Discussion of Possible Research Topics. In *ICTAC'2007*, volume 4701 of *Lecture Notes in Computer Science (eds. J.C.P. Woodcock et al.)*, pages 1–17, Heidelberg, September 2007. Springer.
181. Dines Bjørner and Asger Eir. Compositionality: Ontology and Mereology of Domains. Some Clarifying Observations in the Context of Software Engineering in July 2008, eds. Martin Steffen, Dennis Dams and Ulrich Hannemann. In *Festschrift for Prof. Willem Paul de Roever Concurrency, Compositionality, and Correctness*, volume 5930 of *Lecture Notes in Computer Science*, pages 22–59, Heidelberg, July 2010. Springer.
182. Dines Bjørner. The Rôle of Domain Engineering in Software Development. Why Current Requirements Engineering Seems Flawed! In *Perspectives of Systems Informatics*, volume 5947 of *Lecture Notes in Computer Science*, pages 2–34, Heidelberg, Wednesday, January 27, 2010. Springer.
183. Dines Bjørner. Believable Software Management. *Encyclopedia of Software Engineering*, 1(1):1–32, 2011.
184. Dines Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics, Part I of II: The Engineering Part*. *Kibernetika i sistemny analiz*, (4):100–116, May 2010.
185. Dines Bjørner. *Domain Science and Engineering as a Foundation for Computation for Humanity*, chapter 7, pages 159–177. Computational Analysis, Synthesis, and Design of Dynamic Systems. CRC [Francis & Taylor], 2013. (eds.: Justyna Zander and Pieter J. Mosterman).
186. Abraham Maslow. A Theory of Human Motivation. *Psychological Review*, 50(4):370–96, 1943. http://-psychclassics.yorku.ca/Maslow/motivation.htm.
187. Abraham Maslow. *Motivation and Personality*. Harper and Row Publishers, 3rd ed., 1954.
188. Christopher Peterson and Martin E.P. Seligman. *Character strengths and virtues: A handbook and classification*. Oxford University Press, 2004.
189. Dines Bjørner. Domain Engineering – A Basis for Safety Critical Software. Invited Keynote, ASSC2014: Australian System Safety Conference, Melbourne, 26–28 May, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, December 2014.
190. Dines Bjørner. An Emerging Domain Science – A Rôle for Stanisław Leśniewski's Mereology and Bertrand Russell's Philosophy of Logical Atomism. *Higher-order and Symbolic Computation*, 2009.
191. Dines Bjørner. *Domain Engineering: Technology Management, Research and Engineering*. Research Monograph (# 4); JAIST Press, 1-1, Asahidai, Nomi, Ishikawa 923-1292 Japan, This Research Monograph contains the following main chapters:

[12] http://www2.imm.dtu.dk/ db/zohar.pdf
[13] http://www.imm.dtu.dk/ db/.pdf

1 *On Domains and On Domain Engineering – Prerequisites for Trustworthy Software – A Necessity for Believable Management*, pages 3–38.
2 *Possible Collaborative Domain Projects – A Management Brief*, pages 39–56.
3 *The Rôle of Domain Engineering in Software Development*, pages 57–72.
4 *Verified Software for Ubiquitous Computing – A VSTTE Ubiquitous Computing Project Proposal*, pages 73–106.
5 *The Triptych Process Model – Process Assessment and Improvement*, pages 107–138.
6 *Domains and Problem Frames – The Triptych Dogma and M.A.Jackson's PF Paradigm*, pages 139–175.
7 *Documents – A Rough Sketch Domain Analysis*, pages 179–200.
8 *Public Government – A Rough Sketch Domain Analysis*, pages 201–222.
9 *Towards a Model of IT Security — – The ISO Information Security Code of Practice – An Incomplete Rough Sketch Analysis*, pages 223–282.
10 *Towards a Family of Script Languages – – Licenses and Contracts – An Incomplete Sketch*, pages 283–328.

2009.

192. Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. ACM Press/Addison-Wesley, New York, NY, 2000.
193. K. Pohl, G. Bockle, and F. van der Linden. *Software Product Line Engineering*. Springer, Berlin, Heidelberg, New York, 2005.
194. W.R. Bevier, W.A. Hunt Jr., J Strother Moore, and W.D. Young. An approach to system verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989. Special Issue on System Verification.
195. Don I. Good and William D. Young. Mathematical Methods for Digital Systems Development. In *VDM '91: Formal Software Development Methods*, pages 406–430. Springer-Verlag, October 1991. Volume 2.
196. Dines Bjørner. A ProCoS Project Description. *Published in two slightly different versions: (1) EATCS Bulletin, October 1989, (2) (Ed. Ivan Plander:) Proceedings: Intl. Conf. on AI & Robotics, Strebske Pleso, Slovakia, Nov. 5-9, 1989, North-Holland, Publ.*, Dept. of Computer Science, Technical University of Denmark, October 1989.
197. Dines Bjørner. Trustworthy Computing Systems: The ProCoS Experience. In *14'th ICSE: Intl. Conf. on Software Eng., Melbourne, Australia*, pages 15–34. ACM Press, May 11–15 1992.
198. M. Harsu. A Survey on Domain Engineering. Review, Institute of Software Systems, Tampere University of Technology, Finland, December 2002.
199. R. Falbo, G. Guizzardi, and K.C. Duarte. An Ontological Approach to Domain Engineering. In *Software Engineering and Knowledge Engineering*, Proceedings of the 14th international conference SEKE'02, pages 351–358, Ischia, Italy, July 15-19 2002. ACM.
200. F. Buschmann, K. Henney, and D.C. Schmidt. *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*. John Wiley & Sons Ltd., England, 2007.
201. Kai Sørlander. *Om Menneskerettigheder*. Rosinante, 2000. 171 pages.
202. Kai Sørlander. *Fornuftens Skæbne – Tanker om Menneskets Vilkår*. Informations Forlag, 2014. 238 pages.
203. J. M. E. McTaggart. The Unreality of Time. *Mind*, 18(68):457–84, October 1908. New Series. See also: [204].
204. Robin Le Poidevin and Murray MacBeath, editors. *The Philosophy of Time*. Oxford University Press, 1993.
205. Arthur Prior. *Changes in Events and Changes in Things*, chapter in [204]. Oxford University Press, 1993.
206. Arthur N. Prior. *Logic and the Basis of Ethics*. Clarendon Press, Oxford, UK, 1949.
207. Arthur N. Prior. *Formal Logic*. Clarendon Press, Oxford, UK, 1955.
208. Arthur N. Prior. *Time and Modality*. Oxford University Press, Oxford, UK, 1957.
209. Arthur N. Prior. *Past, Present and Future*. Clarendon Press, Oxford, UK, 1967.
210. Arthur N. Prior. *Papers on Time and Tense*. Clarendon Press, Oxford, UK, 1968.
211. Gerald Rochelle. *Behind time: The incoherence of time and McTaggart's atemporal replacement*. Avebury series in philosophy. Ashgate, Brookfield, Vt., USA, 1998. vii + 221 pages.
212. Johannes Sløk. *Stoikerne*. Berlingske Filosofi Bibliotek, 1966.
213. David Hume. *An Enquiry Concerning Human Understanding*. Washington Square Press, 1963 (1748, 1750). Ed.Ernest C. Mossner.
214. Bertrand Russell. The Philosophy of Logical Atomism. *The Monist: An International Quarterly Journal of General Philosophical Inquiry,*, xxxviii–xxix:495–527, 32–63, 190–222, 345–380, 1918–1919.
215. Alfred North Whitehead and Bertrand Russell. *Principia Mathematica, 3 vols.* Cambridge University Press, 1910, 1912, and 1913. Second edition, 1925 (Vol. 1), 1927 (Vols 2, 3), also Cambridge University Press, 1962.

216. Ludwig Wittgenstein. *Tractatus Logico–Philosophicus*. Taylor & Francis Books Ltd, 1921 (1975).
217. Ludwig Johan Josef Wittgenstein. *Philosophical Investigations*. Oxford Univ. Press, 1958.
218. Bertrand Russell. *History of Western Philosophy*. George Allen and Unwin Ltd, Unwin University Books, London, 1974 (1945, 1961).
219. A.N. Whitehead. *The Concept of Nature*. Cambridge University Press, Cambridge, 1920.
220. . Technical report. .
221. E.C. Luschei. *The Logical Systems of Leśniewksi*. North Holland, Amsterdam, The Netherlands, 1962.
222. C. Lejewski. A note on Leśniewksi's Axiom System for the Mereological Notion of Ingredient or Element. *Topoi*, 2(1):63–71, June, 1983.
223. J.T.J. Srzednicki and Z. Stachniak, editors. *Leśniewksi's Lecture Notes in Logic*. Dordrecht, 1988.
224. J.T.J. Srzednicki and Z. Stachniak. *Leśniewksi's Systems Protothetic*. . Dordrecht, 1998.
225. S. J. Surma, J. T. Srzednicki, D. I. Barnett, and V. F. Rickey, editors. *Stanisław Leśniewksi: Collected works (2 Vols.)*. Dordrecht, Boston – New York, 1988.
226. Nelson Goodman. *The Structure of Appearance*. 1st.: Cambridge, Mass., Harvard University Press; 2nd.: Indianapolis, Bobbs-Merrill, 1966; 3rd.: Dordrecht, Reidel, 1977, 1951.
227. Marcus Rossberg. *Leonard, Goodman, and the Development of the Calculus of Individuals*. Ontos, Frankfurt, Germany, 2009.
228. Bowman L. Clarke. A Calculus of Individuals Based on 'Connection'. *Notre Dame J. Formal Logic*, 22(3):204–218, 1981.
229. Bowman L. Clarke. Individuals and Points. *Notre Dame J. Formal Logic*, 26(1):61–75, 1985.
230. Douglas T. Ross. Toward foundations for the understanding of type. In *Proceedings of the 1976 conference on Data: Abstraction, definition and structure*, pages 63–65, New York, NY, USA, 1976. ACM. http://doi.acm.org/10.1145/800237.807120.
231. M. Bunge. *Treatise on Basic Philosophy: Ontology I: The Furniture of the World*, volume 3. Reidel, Boston, Mass., USA, 1977.
232. M. Bunge. *Treatise on Basic Philosophy: Ontology II: A World of Systems*, volume 4. Reidel, Boston, Mass., USA, 1979.
233. Peter M. Simons. *Parts: A Study in Ontology*. Clarendon Press, 1987.
234. Barry Smith. Ontology and the Logistic Analysis of Reality. In N. Guarino and R. Poli, editors, *International Workshop on Formal Ontology in Conceptual Analysis and Knowledge Representation*. Institute for Systems Theory and Biomedical Engineering of the Italian National Research Council, Padua, Italy, 1993. Revised version in G. Haefliger and P. M. Simons (eds.), Analytic Phenomenology, Dordrecht/Boston/London: Kluwer.
235. Barry Smith. Ontology and the Logistic Analysis of Reality. In G. Haefliger and P. M. Simons, editors, *Analytic Phenomenology*. Dordrecht/Boston/London: Kluwer, Padua, Italy, 1993.
236. Barry Smith. Fiat Objects. In L. Vieu N. Guarino and S. Pribbenow, editors, *Parts and Wholes: Conceptual Part-Whole Relations and Formal Mereology*, 11th European Conference on Artificial Intelligence, pages 15–23. European Coordinating Committee for Artificial Intelligence, Amsterdam, 8 August 1994. (Revised version: Topoi, 2001, vol.20, no.2, pp.131-148).
237. Pierre Grenon and Barry Smith. SNAP and SPAN: Towards Dynamic Spatial Ontology. *Spatial Cognition & Computation*, 4(1):69 – 104, 2004.
238. Pierre Grenon and Barry Smith. SNAP and SPAN: Towards Dynamic Spatial Ontology. *Spatial Cognition and Computation*, 4(1):69–104, 2004.
239. Achille C. Varzi. *Spatial Reasoning in a Holey*[14] *World*, volume 728 of *Lecture Notes in Artificial Intelligence*, pages 326–336. Springer, 1994.
240. Achille C. Varzi. *On the Boundary between Mereology and Topology*, pages 419–438. Hölder-Pichler-Tempsky, Vienna, 1994.
241. Dines Bjørner. The Manifest Domain Analysis & Description Approach to Implicit and Explicit Semantics. *EPTCS: Electronic Proceedings in Theoretical Computer Science, Yasmine Ait-Majeur, Paul J. Gibson and Dominique Méry*, 2018. First International Workshop on Handling IMPlicit and EXplicit Knowledge in Formal Fystem Development, 17 November 2017, Xi'an, China.
242. Erik Lund, Mogens Pihl, and Johannes Sløk. *De europæiske ideeres historie*. Gyldendal, 1962.
243. Karl Jaspers. *Philosophy, Vols.1–3*. The University of Chicago Press, 1969. Translated by E.B. Ashton.
244. Bertrand Russell. *The Problems of Philosophy*. Home University Library, London, 1912. Oxford University Press paperback, 1959 Reprinted, 1971-2.
245. Bertrand Russell. *Introduction to Mathematical Philosophy*. George Allen and Unwin, London, 1919.

[14] holey: something full of holes

246. A.N. Whitehead. *An Enquiry Concerning the Principles of Natural Knowledge*. Cambridge University Press, Cambridge, 1929.
247. Alfred North Whitehead. *Science and the Modern World*. The Free Press, New York, 1925.
248. Willard van Orman Quine. *From a Logical Point of View*. Harvard Univ. Press, Cambridge, Mass., USA, 1953, 1980. Collection of papers: Language and Ontology.
249. Willard van Orman Quine. *Word and Object*. The MIT Press, Cambridge, Mass., USA, 1960. Naturalism: Philosophy as part of Natural Science.
250. Willard van Orman Quine. *Pursuit of Truth*. Harvard Univ. Press, Cambridge, Mass., USA, paperback edition, 1992. Clear, concise formulation of Quine's philosophical position.
251. Ludwig Johan Josef Wittgenstein. *Tractatu Logico–Philosophicus*. Oxford Univ. Press, London, (1921) 1961.
252. Karl R. Popper. *The Logic of Scientific Dicovery*. Hutchinson of London, 3 Fitzroy Square, London W1, England, 1959,. . . ,1979. Translated from [283].
253. Karl R. Popper. *Conjectures and Refutations. The Growth of Scientific Knowledge*. Routledge and Kegan Paul Ltd. (Basic Books, Inc.), 39 Store Street, WC1E 7DD, London, England (New York, NY, USA), 1963,. . . ,1981.
254. Karl R. Popper. *Autobiography of Karl Popper*. in The Philosohy of Karl Popper in *The Library of Living Philosophers*, Ed. Paul Arthur Schilpp. Open Court Publishing Co., Illionos, USA, 1976. See also [255].
255. Karl R. Popper. *Unended Quest: An Intellectual Autobiography*. Philosophy and Autobiography. Fontana/Collins, England, 1976–1982. Originally in [254].
256. Karl R. Popper. *A Pocket Popper*. Fontana Pocket Readers. Fontana Press, England, 1983. An edited collection, Ed. David Miller.
257. Karl R. Popper. *The Myth of the Framework. In defence of science and rationality*. Routledge, 11 New Fetter Lane, London EC4P 4EE, England, 1994, 1996. An edited collection of essays, Ed. M.A. Notturno.
258. Imre Lakatos. *Proofs and Refutations: The Logic of Mathematical Discovery (Eds.: J. Worrall and E. G. Zahar)*. Cambridge University Press, The Edinburgh Building, Shaftesbury Road, Cambridge CB2 2RU, England, 2 September 1976. ISBN: 0521290384. Published in 1963-64 in four parts in the British Journal for Philosophy of Science. (Originally Lakatos' name was Imre Lipschitz.).
259. David Favrholdt. *Filosofisk Codex — Om begrundelsen af den menneskelige erkendelse*. Gyldendal, Nordisk Forlag, Klareboderne, Copenhagen K, Denmark, 1999. In Danish. English/German 'translation': Philosophical Codex — On motivating human 'erkenntniss'. 361 pages.
260. David Favrholdt. *Æstetik og filosofi*. Høst Humaniora. Høst & Søn, Købmagergade 62, DK–1150 Copenhagen K, Denmark, Fall 2000.
261. John F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Brooks/Cole Thompson Learning, August 17, 1999.
262. Jonathan Dancy and Ernest Sosa, editors. *The Blackwell Companion to Epistemology*. Blackwell Companions to Philosophy. Blackwell Publishers, 108 Cowley Road, Oxford OX4 1JF, UK, 1994.
263. Arne Naess. *Philosophy*. Oslo, Norway, 1980.
264. Thomas Hofweber. Logic and ontology. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2012 edition, 2012.
265. Henry Laycock. Object. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2011 edition, 2011.
266. Henry Laycock. Object. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2011 edition, 2011.
267. Thomas Bittner. Ontology, Vagueness, and Indeterminacy — Extended Abstract. Technical report, Centre de recherche en geomatique, Laval University, Quebec, Canada. Thomas.Bittner@scg.ulaval.ca.
268. George Wilson and Samuel Shpall. Action. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2012 edition, 2012.
269. Roberto Casati and Achille Varzi. Events. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2010 edition, 2010.
270. Nelson Goodman. *A Study of Qualities*. Garland, New York, 1990. Ph.D. dissertation thesis, Harvard, 1940.
271. D. H. Mellor and Alex Oliver. *Properties*. Oxford Readings in Philosophy. Oxford Univ Press, May 1997. ISBN: 0198751761, 320 pages.
272. D. H. Mellor and Alex Oliver. *Properties*. Oxford Readings in Philosophy. Oxford Univ Press, May 1997. ISBN: 0198751761, 320 pages.
273. Chris Fox. *The Ontology of Language: Properties, Individuals and Discourse*. CSLI Publications, Center for the Study of Language and Information, Stanford University, California, ISA, 2000.
274. Kai Sørlander. *Forsvaret for Rationaliteten*. Informations Forlag (Publ.), 2008. 232 pages.

275. Kai Sørlander. *Den Politiske Forpligtelse – Det Filosofsike Fundament for Demokratisk Stillingtagen*. Informations Forlag, 2011. 280 pages.
276. Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Qinghua University Press, 2008.
277. Dines Bjørner. **Chinese:** *Software Engineering, Vol. 1: Abstraction and Modelling*. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.
278. Dines Bjørner. *Software Engineering, Vol. 2: Specification of Systems and Languages*. Qinghua University Press, 2008.
279. Dines Bjørner. **Chinese:** *Software Engineering, Vol. 2: Specification of Systems and Languages*. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.
280. Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Qinghua University Press, 2008.
281. Dines Bjørner. **Chinese:** *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.
282. Jochen Renz and Hans W. Guesgen, editors. *Spatial and Temporal Reasoning*, volume 14, vol. 4, Journal: AI Communications, Amsterdam, The Netherlands, Special Issue. IOS Press, December 2004.
283. Karl R. Popper. *Logik der Forschung*. Julius Springer Verlag, Vienna, Austria, 1934 (1935). English version [252].
284. C.A.R. Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985.
285. Dines Bjørner. Documents – a Domain Description[15]. Experimental Research Report 2013-3, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013.
286. Dines Bjørner. *[125] Chap. 10: Towards a Family of Script Languages – – Licenses and Contracts – Incomplete Sketch*, pages 283–328. JAIST Press, March 2009.
287. Usama Mehmood, Radu Grosu, Ashish Tiwari, Nicola Paoletti, Shan Lin, Yang JunXing, Dung Phan, Scott D. Stoller, and Scott A. Smolka. *Declarative vs Rule-based Control for Flocking Dynamics*. In *Proceedings of ACM/SIGAPP Symposium on Applied Computing (SACC 2018)*. ACM Press, April 9–13, 2018. 8 pages.
288. C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8), Aug. 1978.
289. C.A.R. Hoare. Communicating Sequential Processes. Published electronically: `http://www.usingcsp.-com/cspbook.pdf`, 2004. Second edition of [284]. See also `http://www.usingcsp.com/`.
290. A. W. Roscoe. *Theory and Practice of Concurrency*. C.A.R. Hoare Series in Computer Science. Prentice-Hall, 1997. Now available on the net: http://www.comlab.ox.ac.uk/people/bill.roscoe/publications/-68b.pdf.
291. Steve Schneider. *Concurrent and Real-time Systems — The CSP Approach*. Worldwide Series in Computer Science. John Wiley & Sons, Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England, January 2000.
292. Craig Reynolds. *Flocks, Herds and Schools: A Distributed Behavioral Model*. *SIGGRAPH Computer Graphics*, 21(4), August 1987. https://doi.org/10.1145/37402.37406.
293. Craig Reynolds. *Steering Behaviors for Autonomous Characters*. In *Proceedings of Game Developers Conference*, pages 763–782, 1999.
294. Craig Reynolds. OpenSteer, *Steering Behaviours for Autonomous Characters*, 2004. http://opensteer.sourceforge.net.
295. Reza Olfati-Saber. Flocking for Multi-agent Dynamic Systems: Algorithms and Theory. *IEEE Transactions on Automatic Control*, 51(3):401–420, 13 March 2006. http://ieeexplore.ieee.org/document/1605401/; DOI: 10.1109/TAC.2005.864190; Thayer School of Engineering, Dartmouth College, Hanover, NH, USA.

[15] http://www.imm.dtu.dk/˜dibj/doc-p.pdf

Part VI

Experimental Case Studies

A

Credit Card Systems

This appendix presents an attempt at a model of a credit card system.

A.1 Introduction

We present a domain description of an abstracted credit card system. The narrative part of the description is terse, perhaps a bit too terse. I might "repair" this shortness if told so. A reference is made to my paper: [2, Manifest Domains: Analysis & Description]. That paper can be found on the Internet: **http://www2.compute.dtu.dk/~dibj/2015/faoc/faoc-bjorner.pdf**.

Credit cards are moving from simple plastic cards to smart phones. Uses of credit cards move from their mechanical insertion in credit card terminals to being swiped. Authentication (hence not modelled) moves from keying in security codes to eye iris "prints", and/or finger prints or voice prints or combinations thereof.

This document abstracts from all that in order to understand a bare, minimum essence of credit cards and their uses. Based on a model, such as presented here, the reader should be able to extend/refine the model into any future technology – for requirements purposes.

A.2 Endurants

A.2.1 Credit Card Systems

476 Credit card systems, *ccs*:*CCS*, [1]consists of three kinds of parts:
477 an assembly, *cs*:*CS*, of credit cards[3],
478 an assembly, *bs*:*BS*, of banks, and
479 an assembly, *ss*:*SS*, of shops.

type
476 CCS
477 CS
478 BS
479 SS
value
477 **obs_part_**CS: CCS → CS

[1] The composite part *CS* can be thought of as a credit card company, say VISA[2]. The composite part *BS* can be thought of as a bank society, say BBA: British Banking Association. The composite part *SS* can be thought of as the association of retailers, say bira: British Independent Retailers Association. The model does not prevent "shops" from being airlines, or car rental agencies, or dentists, or consultancy firms. In this case *SS* would be some appropriate association.

[3] We "equate" credit cards with their holders.

478 **obs_part**_BS: CCS → BS
479 **obs_part**_SS: CCS → SS

480 There are credit cards, *c*:*C*, banks *b*:*B*, and shops *s*:*S*.
481 The credit card part, *cs*:*CS*, abstracts a set, *soc*:*Cs*, of card.
482 The bank part, *bs*:*BS*, abstracts a set, *sob*:*Bs*, of banks.
483 The shop part, *ss*:*SS*, abstracts a set, *sos*:*Ss*, of shops.

type
480 C, B, S
481 Cs = C-**set**
482 Bs = B-**set**
483 Ss = S-**set**
value
481 **obs_part**_CS: CS → Cs, **obs_part**_Cs: CS → Cs
482 **obs_part**_BS: BS → Bs, **obs_part**_Bs: BS → Bs
483 **obs_part**_SS: SS → Ss, **obs_part**_Ss: SS → Ss

484 Assembliers of credit cards, banks and shops have unique identifiers, *csi*:*CSI*, *bsi*:*BSI*, and *ssi*:*SSI*.
485 Credit cards, banks and shops have unique identifiers, *ci*:*CI*, *bi*:*BI*, and *si*:*SI*.
486 One can define functions which extract all the
487 unique credit card,
488 bank and
489 shop identifiers from a credit card system.

484 CSI, BSI, SSI
485 CI, BI, SI
value
484 **uid**_CS: CS→CSI, **uid**_BS: BS→BSI, **uid**_SS: SS→SSI,
485 **uid**_C: C→CI, **uid**_B: B→BI, **uid**_S: S→SI,
487 xtr_CIs: CCS → CI-**set**
487 xtr_CIs(ccs) ≡ {**uid**_C(c)|c:C•c ∈ **obs_part**_Cs(**obs_part**_CS(ccs))}
488 xtr_BIs: CCS → BI-**set**
488 xtr_BIs(ccs) ≡ {**uid**_B(s)|b:B•b ∈ **obs_part**_Bs(**obs_part**_BS(ccs))}
489 xtr_SIs: CCS → SI-**set**
489 xtr_SIs(ccs) ≡ {**uid**_S(s)|s:S•s ∈ **obs_part**_Ss(**obs_part**_SS(ccs))}

490 For all credit card systems it is the case that
491 all credit card identifiers are distinct from bank identifiers,
492 all credit card identifiers are distinct from shop identifiers,
493 all shop identifiers are distinct from bank identifiers,

axiom
490 ∀ ccs:CCS •
490 **let** cis=xtr_CIs(ccs), bis=xtr_BIs(ccs), sis = xtr_SIs(ccs) **in**
491 cis ∩ bis = {}
492 ∧ cis ∩ sis = {}
493 ∧ sis ∩ bis = {} **end**

A.2.2 Credit Cards

494 A credit card has a mereology which "connects" it to any of the shops of the system and to exactly one bank of the system,
495 and some attributes — which we shall presently disregard.
496 The wellformedness of a credit card system includes the wellformedness of credit card mereologies with respect to the system of banks and shops:
497 The unique shop identifiers of a credit card mereology must be those of the shops of the credit card system; and
498 the unique bank identifier of a credit card mereology must be of one of the banks of the credit card system.

```
type
494.    CM = SI-set × BI
value
494.    obs_mereo_CM: C → CM
496     wf_CM_of_C: CCS → Bool
496     wf_CM_of_C(ccs) ≡
494         let bis=xtr_BIs(ccs), sis=xtr_SIs(ccs) in
494          ∀ c:C•c ∈ obs_part_Cs(obs_part_CS(ccs)) ⇒
494             let (ccsis,bi)=obs_mereo_CM(c) in
497             ccsis ⊆ sis
498           ∧ bi ∈ bis
494         end end
```

A.2.3 Banks

Our model of banks is (also) very limited.

499 A bank has a mereology which "connects" it to a subset of all credit cards and a subset of all shops,
500 and, as attributes:
501 a cash register, and
502 a ledger.
503 The ledger records for every card, by unique credit card identifier,
504 the current balance: how much money, credit or debit, i.e., plus or minus, that customer is owed, respectively has borrowed from the bank,
505 the dates-of-issue and -expiry of the credit card, and
506 the name, address, and other information about the credit card holder.
507 The wellformedness of the credit card system includes the wellformedness of the banks with respect to the credit cards and shops:
508 the bank mereology's
509 must list a subset of the credit card identifiers and a subset of the shop identifiers.

```
type
499     BM = CI-set × SI-set
501     CR = Bal
502     LG = CI →m (Bal×DoI×DoE×...)
504     Bal = Int
value
499     obs_mereo_B: B → BM
501     attr_CR: B → CR
502     attr_LG: B → LG
507         wf_BM_B: CCS → Bool
507         wf_BM_B(ccs) ≡
```

```
507        let allcis = xtr_CIs(ccs), allsis = xtr_SIs(ccs) in
507        ∀ b:B • b ∈ obs_part_Bs(obs_part_BS(ccs)) in
508           let (cis,sis) = obs_mereo_B(b) in
509           cis ⊆ ∀ cis ∧ sis ⊆ allsis end end
```

A.2.4 Shops

510 The mereology of a shop is a pair: a unique bank identifiers, and a set of unique credit card identifiers.
511 The mereology of a shop
512 must list a bank of the credit card system,
513 band a subset (or all) of the unique credit identifiers.

We omit treatment of shop attributes.

```
type
510   SM = CI-set × BI
value
510   obs_mereo_S: S → SM
511   wf_SM_S: CCS → Bool
511   wf_SM_S(ccs) ≡
511     let allcis = xtr_CIs(ccs), allbis = xtr_BIs(ccs) in
511     ∀ s:S • s ∈ obs_part_Ss(obs_part_SS(ccs)) ⇒
511         let (cis,bi) obs_mereo_S(s) in
512         bi ∈ allbis
513       ∧ cis ⊆ allcis
511     end end
```

A.3 Perdurants

A.3.1 Behaviours

514 We ignore the behaviours related to the *CCS*, *CS*, *BS* and *SS* parts.
515 We therefore only consider the behaviours related to the *Cs*, *Bs* and *Ss* parts.
516 And we therefore compile the credit card system into the parallel composition of the parallel compositions of all the credit card, *crd*, all the bank, *bnk*, and all the shop, *shp*, behaviours.

```
value
514   ccs:CCS
514   cs:CS = obs_part_CS(ccs),
514   uics:CSI =uid_CS(cs),
514   bs:BS = obs_part_BS(ccs),
514   uibs:BSI =uid_BS(bs),
514   ss:SS = obs_part_SS(ccs),
514   uiss:SSI =uid_SS(ss),
515   socs:Cs = obs_part_Cs(cs),
515   sobs:Bs = obs_part_Bs(bs),
515   soss:Ss = obs_part_Ss(ss),
```

```
value
516   sys: Unit → Unit,
514   sys() ≡
516       cards_uics(obs_mereo_CS(cs),...)
```

516 ∥ ∥ {crd$_{uid_C(c)}$(**obs_mereo_**C(c))|c:C•c ∈ socs}
516 ∥ banks$_{uibs}$(**obs_mereo_**BS(bs),...)
516 ∥ ∥ {bnk$_{uid_B(b)}$(**obs_mereo_**B(b))|b:B•b ∈ sobs}
516 ∥ shops$_{uiss}$(**obs_mereo_**SS(ss),...)
516 ∥ ∥ {shp$_{uid_S(s)}$(**obs_mereo_**S(s))|s:S•s ∈ soss},
514 cards$_{uics}$(...) ≡ **skip**,
514 banks$_{uibs}$(...) ≡ **skip**,
514 shops$_{uiss}$(...) ≡ **skip**

axiom **skip** ∥ behaviour(...) ≡ behaviour(...)

A.3.2 Channels

517 Credit card behaviours interact with bank (each with one) and many shop behaviours.
518 Shop behaviours interact with bank (each with one) and many credit card behaviours.
519 Bank behaviours interact with many credit card and many shop behaviours.
The inter-behaviour interactions concern:
520 between credit cards and banks: withdrawal requests as to a sufficient, mk_Wdr(am), balance on the credit card account for buying am:AM amounts of goods or services, with the bank response of either is_OK() or is_NOK(), or the revoke of a card;
521 between credit cards and shops: the buying, for an amount, am:AM, of goods or services: mk_Buy(am), or the refund of an amount;
522 between shops and banks: the deposit of an amount, am:AM, in the shops' bank account: mk_Depost(ui,am) or the removal of an amount, am:AM, from the shops' bank account: mk_Removl(bi,si,am)

channel
517 {ch_cb[ci,bi]|ci:CI,bi:BI•ci ∈ cis ∧ bi ∈ bis}:CB_Msg
518 {ch_cs[ci,si]|ci:CI,si:SI•ci ∈ cis ∧ si ∈ sis}:CS_Msg
519 {ch_sb[si,bi]|si:SI,bi:BI•si ∈ sis ∧ bi ∈ bis}:SB_Msg
520 CB_Msg == mk_Wdrw(am:aM) | is_OK() | is_NOK() | ...
521 CS_Msg == mk_Buy(am:aM) | mk_Ref(am:aM) | ...
522 SB_Msg == Depost | Removl | ...
522 Depost == mk_Dep((ci:CI|si:SI),am:aM) |
522 Removl == mk_Rem(bi:BI,si:SI,am:aM)

A.3.3 Behaviour Interactions

523 The credit card initiates
  a buy transactions
    i [1.Buy] by enquiring with its bank as to sufficient purchase funds (am:aM);
    ii [2.Buy] if NOK then there are presently no further actions; if OK
    iii [3.Buy] the credit card requests the purchase from the shop – handing it an appropriate amount;
    iv [4.Buy] finally the shop requests its bank to deposit the purchase amount into its bank account.
  b refund transactions
    i [1.Refund] by requesting such refunds, in the amount of am:aM, from a[ny] shop; whereupon
    ii [2.Refund] the shop requests its bank to move the amount am:aM from the shop's bank account
    iii [3.Refund] to the credit card's account.

Thus the three sets of behaviours, crd, bnk and shp interact as sketched in Fig. A.1 on the next page.

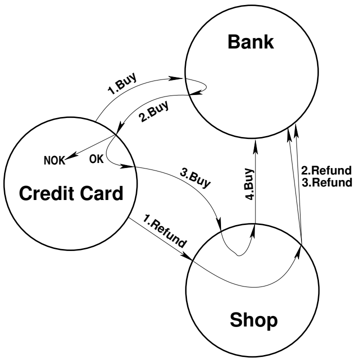


Fig. A.1. Credit Card, Bank and Shop Behaviours

[1.Buy]	Item 529, Pg.250	card ch_cb[ci,bi]!mk_Wdrw(am) (shown as ... three lines down) and
	Item 538, Pg.251	bank mk_Wdrw(ci,am)=⌈⌉⌊⌋{ch_cb[bi,bi]?|ci:CI•ci ∈ cis}.
[2.Buy]	Items 531-532, Pg.251	bank ch_cb[ci,bi]!is_[N]OK() and
	Item 529, Pg.250	shop (...;ch_cb[ci,bi]?).
[3.Buy]	Item 531, Pg.250	card ch_cs[ci,si]!mk_Buy(am) and
	Item 553, Pg.253	shop mk_Buy(am)=⌈⌉⌊⌋{ch_cs[ci,si]?|ci:CI•ci∈ cis}.
[4.Buy]	Item 554, Pg.253	shop ch_sb[si,bi]!mk_Dep(si,am) and
	Item 543, Pg.252	bank mk_Dep(si,am)=⌈⌉⌊⌋{ch_cs[ci,si]?|si:SI•si∈sis}.
[1.Refund]	Item 535, Pg.251	card ch_cs[ci,si]!mk_Ref((ci,si),am) and
	Item 554, Pg.253	shop (si,mk_Ref(ci,am))=⌈⌉⌊⌋{si′,ch_sb[si,bi]?|si,si′:SI•{si,si′}⊆sis∧si=si′}.
[2.Refund]	Item 558, Pg.253	shop ch_sb[si,cbi]!mk_Ref(cbi,(ci,si),am and
	Item 547, Pg.252	bank (si,mk_Ref(cbi,(ci,am)))=⌈⌉⌊⌋{(si′,ch_sb[si,bi]?)|si,si′:SI•{si,si′}⊆sis∧si=si′}.
[3.Refund]	Item 559, Pg.253	shop ch_sb[si,sbi]!mk_Wdr(si,am)) **end** and
	Item 548, Pg.252	bank (si,mk_Wdr(ci,am))=⌈⌉⌊⌋{(si′,ch_sb[si,bi]?)|si,si′:SI•{si,si′}⊆sis∧si=si′}

A.3.4 Credit Card

524 The credit card behaviour, crd, takes the credit card unique identifier, the credit card mereology, and attribute arguments (omitted). The credit card behaviour, crd, accepts inputs from and offers outputs to the bank, bi, and any of the shops, si∈sis.

525 The credit card behaviour, crd, non-deterministically, internally “cycles” between buying and getting refunds.

value

524 $\mathsf{crd}_{ci:CI}$: (bi,sis):CM → **in**,**out** ch_cb[ci,bi],{ch_cs[ci,si]|si:SI•si ∈ sis} **Unit**

524 crd_{ci}(bi,sis) ≡ (buy(ci,(bi,sis)) ⌈⌉ ref(ci,(bi,sis))) ; crd_{ci}(ci,(bi,sis))

526 By am:AM we mean an amount of money, and by si:SI we refer to a shop in which we have selected a number or goods or services (not detailed) costing am:AM.

527 The buyer action is simple.

528 The amount for which to buy and the shop from which to buy are selected (arbitrarily).

529 The credit card (holder) withdraws am:AM from the bank, if sufficient funds are available[4].

530 The response from the bank

531 is either OK and the credit card [holder] completes the purchase by buying the goods or services offered by the selected shop,

[4] First the credit card [holder] requests a withdrawal. If sufficient funds are available, then the withdrawal takes place, otherwise not – and the credit card holder is informed accordingly.

532 or the response is "not OK", and the transaction is skipped.

```
type
526  AM = Int
value
527  buy: ci:CI × (bi,sis):CM →
527     in,out ch_cb[ci,bi] out {ch_cs[ci,si]|si:SI•si ∈ sis}  Unit
527  buy(ci,(bi,sis)) ≡
528     let am:aM • am>0, si:SI • si ∈ sis in
529     let msg = (ch_cb[ci,bi]!mk_Wdrw(am);ch_cb[ci,bi]?) in
530     case msg of
531         is_OK() → ch_cs[ci,si]!mk_Buy(am),
532         is_NOK() → skip
527     end end end
```

533 The refund action is simple.
534 The credit card [handler] requests a refund am:AM
535 from shop si:SI.
This request is handled by the shop behaviour's sub-action ref, see lines 551.–560. page 253.

```
value
533  rfu: ci:CI × (bi,sis):CM → out {ch_cs[ci,si]|si:SI•si ∈ sis}  Unit
533  rfu(ci,(bi,sis)) ≡
534       let am:AM • am>0, si:SI • si ∈ sis in
535       ch_cs[ci,si]!mk_Ref(bi,(ci,si),am)
533       end
```

A.3.5 Banks

536 The bank behaviour, bnk, takes the bank's unique identifier, the bank mereology, and the programmable attribute arguments: the ledger and the cash register. The bank behaviour, bnk, accepts inputs from and offers outputs to the any of the credit cards, $ci \in cis$, and any of the shops, $si \in sis$.

537 The bank behaviour non-deterministically externally chooses to accept either 'withdraw'al requests from credit cards or 'deposit' requests from shops or 'refund' requests from credit cards.

```
value
536  bnk_{bi:BI}: (cis,sis):BM → (LG×CR) →
536       in,out {ch_cb[ci,bi]|ci:CI•ci ∈ cis} {ch_sb[si,bi]|si:SI•si ∈ sis}  Unit
536  bnk_{bi}((cis,sis))(lg:(bal,doi,doe,...),cr) ≡
537       wdrw(bi,(cis,sis))(lg,cr)
537       ⌈⌉⌊⌋ depo(bi,(cis,sis))(lg,cr)
537       ⌈⌉⌊⌋ refu(bi,(cis,sis))(lg,cr)
```

538 The 'withdraw' request, wdrw, (an action) non-deterministically, externally offers to accept input from a credit card behaviour and marks the only possible form of input from credit cards, mk_Wdrw(ci,am), with the identity of the credit card.

539 If the requested amount (to be withdrawn) is not within balance on the account

540 then we, at present, refrain from defining an outcome (**chaos**), whereupon the bank behaviour is resumed with no changes to the ledger and cash register;

541 otherwise the bank behaviour informs the credit card behaviour that the amount can be withdrawn; whereupon the bank behaviour is resumed notifying a lower balance and 'withdraws' the monies from the cash register.

value
```
537 wdrw: bi:BI × (cis,sis):BM → (LG×CR) → in,out {ch_cb[bi,ci]|ci:CI•ci ∈ cis} Unit
537 wdrw(bi,(cis,sis))(lg,cr) ≡
538     let mk_Wdrw(ci,am) = ⌈⌉⌊⌋ {ch_cb[ci,bi]?|ci:CI•ci ∈ cis} in
537     let (bal,doi,doe) = lg(ci) in
539     if am>bal
540        then (ch_cb[ci,bi]!is_NOK(); bnk_bi(cis,sis)(lg,cr))
541        else (ch_cb[ci,bi]!is_OK(); bnk_bi(cis,sis)(lg†[ci↦(bal−am,doi,doe)],cr−am)) end
536     end end
```

The ledger and cash register attributes, lg,cr, are programmable attributes. Hence they are modeled as separate function arguments.

542 The deposit action is invoked, either by a shop behaviour, when a credit card [holder] buy's for a certain amount, am:AM, or requests a refund of that amount. The deposit is made by shop behaviours, either on behalf of themselves, hence am:AM, is to be inserted into the shops' bank account, si:SI, or on behalf of a credit card [i.e., a customer], hence am:AM, is to be inserted into the credit card holder's bank account, si:SI.
543 The message, ch_cs[ci,si]?, received from a credit card behaviour is either concerning a buy [in which case *i* is a ci:CI, hence sale, or a refund order [in which case *i* is a si:SI].
544 In either case, the respective bank account is "upped" by am:AM – and the bank behaviour is resumed.

value
```
542 deposit: bi:BI × (cis,sis):BM → (LG×CR) →
543      in,out {ch_sb[bi,si]|si:SI•si ∈ sis} Unit
542 deposit(bi,(cis,sis))(lg,cr) ≡
543      let mk_Dep(si,am) = ⌈⌉⌊⌋ {ch_cs[ci,si]?|si:SI•si ∈ sis} in
542      let (bal,doi,doe) = lg(si) in
544      bnk_bi(cis,sis)(lg†[si↦(bal+am,doi,doe)],cr+am)
542      end end
```

545 The refund action
546 non-deterministically externally offers to either
547 non-deterministically externally accept a mk_Ref(ci,am) request from a shop behaviour, si, or
548 non-deterministically externally accept a mk_Wdr(ci,am) request from a shop behaviour, si.
The bank behaviour is then resumed with the
549 credit card's bank balance and cash register incremented by am and the
550 shop' bank balance and cash register decremented by that same amount.

value
```
545 rfu: bi:BI × (cis,sis):BM → (LG×CR) → in,out {ch_sb[bi,si]|si:SI•si ∈ sis} Unit
545 rfu(bi,(cis,sis))(lg,cr) ≡
547      (let (si,mk_Ref(cbi,(ci,am))) = ⌈⌉⌊⌋ {(si′,ch_sb[si,bi]?)|si,si′:SI•{si,si′}⊆sis∧si=si′} in
545       let (balc,doic,doec) = lg(ci) in
549       bnk_bi(cis,sis)(lg†[ci↦(balc+am,doic,doec)],cr+am)
545       end end)
546     ⌈⌉⌊⌋
548      (let (si,mk_Wdr(ci,am)) = ⌈⌉⌊⌋ {(si′,ch_sb[si,bi]?)|si,si′:SI•{si,si′}⊆sis∧si=si′} in
545       let (bals,dois,does) = lg(si) in
550       bnk_bi(cis,sis)(lg†[si↦(bals−am,dois,does)],cr−am)
545       end end)
```

A.3.6 Shops

551 The shop behaviour, shp, takes the shop's unique identifier, the shop mereology, etcetera.
552 The shop behaviour non-deterministically, externally
either
553 offers to accept a Buy request from a credit card behaviour,
554 and instructs the shop's bank to deposit the purchase amount.
555 whereupon the shop behaviour resumes being a shop behaviour;
556 or
557 offers to accept a refund request in this amount, am, from a credit card [holder].
558 It then proceeds to inform the shop's bank to withdraw the refund from its ledger and cash register,
559 and the credit card's bank to deposit the refund into its ledger and cash register.
560 Whereupon the shop behaviour resumes being a shop behaviour.

```
value
551   shp_si:SI: (CI-set×BI)×...→in,out: {ch_cs[ci,si]|ci:CI•ci ∈ cis},{ch_sb[si,bi′]|bi′:BI•bi′isin bis} Unit
551   shp_si((cis,bi),...) ≡
553         (sal(si,(bi,cis),...)
556          ⌈⌉⌊⌋
557         ref(si,(cis,bi),...)):

551   sal: SI×(CI-set×BI)×...→in,out: {cs[ci,si]|ci:CI•ci ∈ cis},sb[si,bi] Unit
551   sal(si,(cis,bi),...) ≡
553       let mk_Buy(am) = ⌈⌉⌊⌋{ch_cs[ci,si]?|ci:CI•ci ∈ cis} in
554       ch_sb[si,bi]!mk_Dep(si,am) end ;
555       shp_si((cis,bi),...)

551   ref: SI×(CI-set×BI)×...→in,out: {ch_cs[ci,si]|ci:CI•ci ∈ cis},{ch_sb[si,bi′]|bi′:BI•bi′isin bis} Unit
557   ref(si,(cis,sbi),...) ≡
557       let mk_Ref((ci,cbi,si),am) = ⌈⌉⌊⌋{ch_cs[ci,si]?|ci:CI•ci ∈ cis} in
558       (ch_sb[si,cbi]!mk_Ref(cbi,(ci,si),am)
559     ∥ ch_sb[si,sbi]!mk_Wdr(si,am)) end ;
560       shp_si((cis,sbi),...)
```

A.4 Discussion

TO BE WRITTEN

B

Weather Information Systems

Summary

This document reports *work in progress*. We show an example domain description. It is developed and presented as outlined in [2]. The domain being described is that of a generic weather information system. Four main endurants (i.e., aspects) of a generic weather information system are those of the weather, weather stations (collecting weather data), weather data interpretation (i.e., metereological institute[s]), and weather forecast consumers. There are, correspondingly, two kinds of weather information: the weather data, and the weather forecasts. These forms of weather information are acted upon: the weather data interpreter (i.e., a metereological institute) is gathering weather data; based on such interpretations the metereological institute is "calculating" weather forecasts; and and weather forecast consumers are requesting and further "interpreting" (i.e., rendering) such forecasts. Thus weather data is communicated from weather stations to the weather data interpreter; and weather forecasts are communicated from the weather data interpreter to the weather forecast consumers. It is the dual purpose of this technical report to present a domain description of the essence of generic weather information systems, and to add to the "pile" [28, 32, 30, 285, 123, 11, 34, 33] of technical reports that illustrate the use[fulness] of the principles, techniques and tools of [2].

B.1 On Weather Information Systems

B.1.1 On a Base Terminology

From Wikipedia:

561 **Weather** is the state of the atmosphere, to the degree that it is hot or cold, wet or dry, calm or stormy, clear or cloudy, atmospheric (barometric) pressure: high or low.

562 So weather is characterized by **temperature, humidity** (incl. **rain**, **wind** (direction, velocity, center, incl. its possible mobility), **atmospheric pressure**, etcetera.

563 By **weather information** we mean
- either weather data that characterizes the weather as defined above (Item 561),
- or weather forecast, i.e., a prediction of the state of the atmosphere for a given location and time or time interval.

564 Weather data are collected by **weather stations**. We shall here not be concerned with technical means of weather data collection.

565 **Weather forecasts** are used by forecast consumers, anyone: you and me.

566 Weather data interpretation (i.e., **forecasting**) is the science and technology of creating weather forecasts based on **time-** or **time interval-stamped weather data** and **locations**. Weather data interpretation is amongst the charges of meteorological institutes.

567 **Meteorology** is the interdisciplinary scientific study of the atmosphere.

568 An **atmosphere** (from Greek $\alpha\tau\mu o\zeta$ (atmos), meaning “vapour”, and $\sigma\phi\alpha\iota\rho\alpha$ (sphaira), meaning “sphere”) is a layer of gases surrounding a planet or other material body, that is held in place by the gravity of that body.

569 Meteorological institutes work together with the World Meteorological Organization (WMO). Besides weather forecasting, meteorological institutes (and hence WMO) are concerned also with aviation, agricultural, nuclear, maritime, military and environmental meteorology, hydrometeorology and renewable energy.

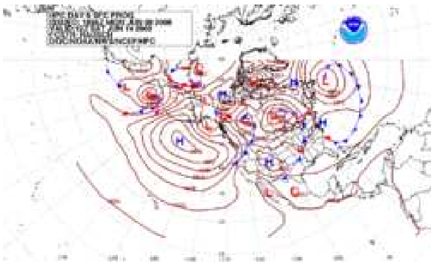
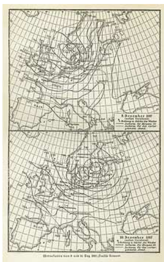
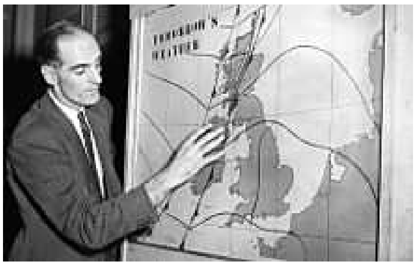
570 Agricultural meteorologists, soil scientists, agricultural hydrologists, and agronomists are persons concerned with studying the effects of weather and climate on plant distribution, crop yield, water-use efficiency, phenology of plant and animal development, and the energy balance of managed and natural ecosystems. Conversely, they are interested in the rôle of vegetation on climate and weather.

B.1.2 Some Illustrations

Weather Stations



Weather Forecasts

Forecast Consumers



B.2 Major Parts of a Weather Information System

We think of the following parts as being of concern in the kind of weather information systems that we shall analyse and describe: Figure B.1 shows one **weather** (dashed rounded corner all embracing rectangle), one central **weather data interpreter** (cum meteorological institute) seven **weather stations** (rounded corner squares), nineteen **weather forecast consumers**, and one global **clock**. All are distributed, as hinted at, in some geographical space. Figure B.2 on the next page shows "an orderly diagram" of "the same"

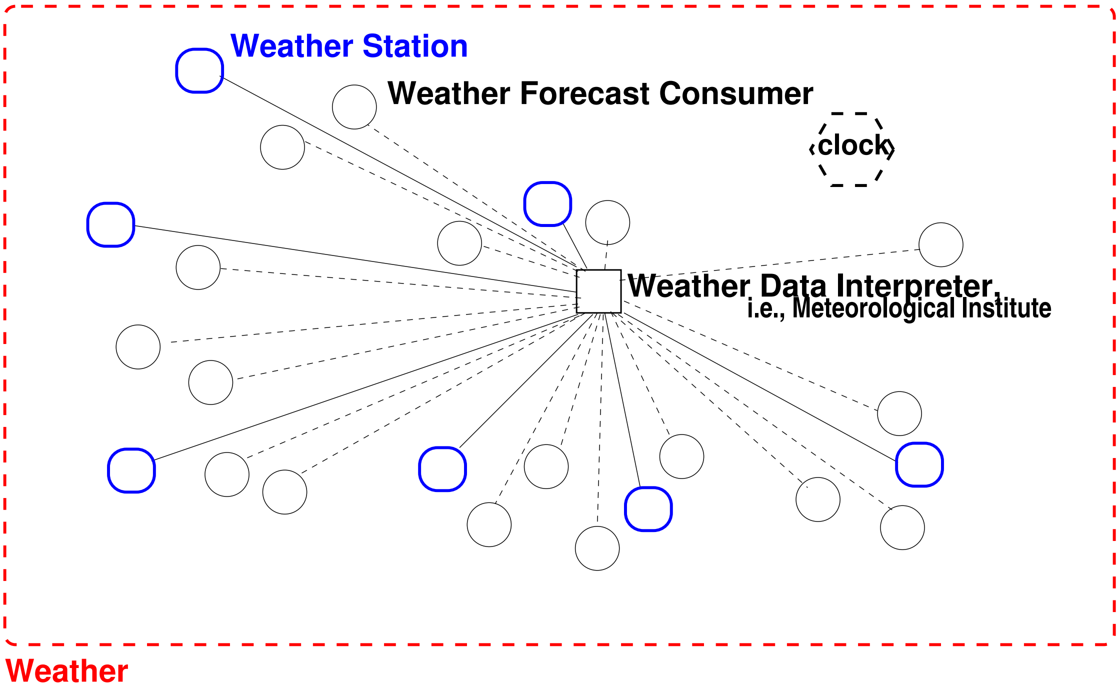


Fig. B.1. A Weather Information System

weather information system as Figure B.1. The lines between pairs of the various parts shall indicate means communication between the pairs of (thus) connected parts. Dashed lines "crossing" bundles of these communication lines are labeled ch_xy. These labels, ch_xy, designated CSP-like channels. An input, by a weather station (wsi), of weather data from the weather (wi), is designated by the CSP expression ch_ws[wi,wsi] ?. An output, say from the weather data interpreter (wdi) to a weather forecast consumer (fci), of a forecast f, is designated by ch_ic[wdii,fci] ! f

B.3 Endurants

B.3.1 Parts and Materials

571 The WIS domain contains a number of sub-domains:
  a the weather, W, which we consider a material,
  b the weather stations sub-domain, WSS (a composite part),
  c the weather data interpretation sub-domain, WDIS (an atomic part),
  d the weather forecast consumers sub-domain, WFCS (a composite part), and
  e the ("global") clock (an atomic part).

type

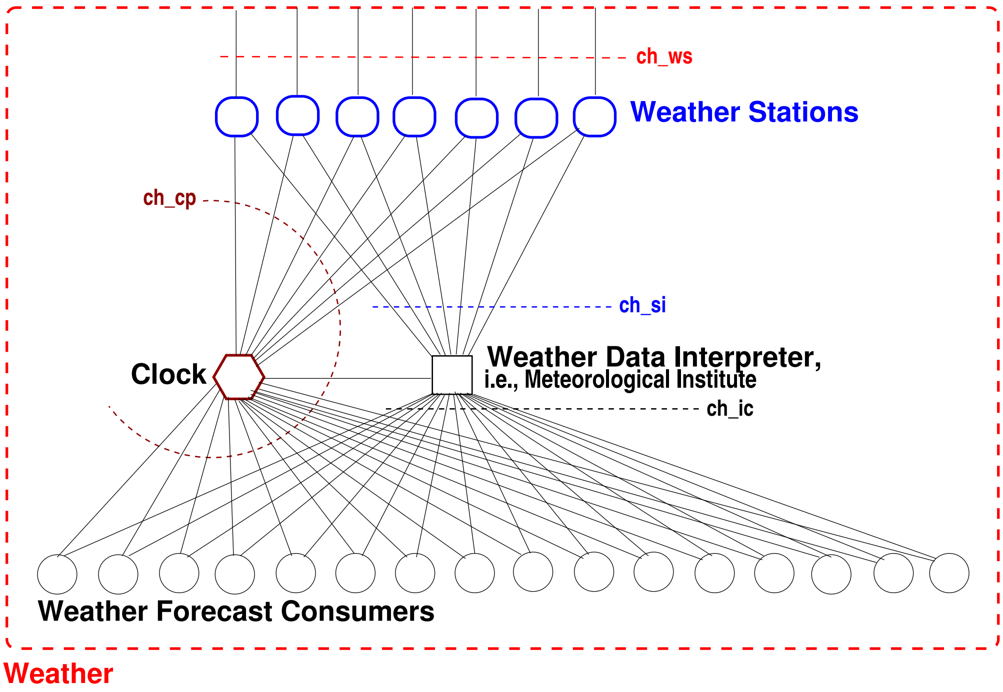


Fig. B.2. A Weather Information System Diagram

571 WIS
571a W
571b WSS
571c WDIS
571d WFCS
571e CLK
value
571a obs_material_W: WIS → W
571b obs_part_WSS: WIS → WSS
571c obs_part_WDIS: WIS → WDIS
571d obs_part_WFCS: WIS → WFCS
571e obs_part_CLK: WIS → CLK

572 The weather station sub-domain, WSS, consists of a set, WSs,
573 of atomic weather stations, WS.
574 The weather forecast consumers sub-domain, WFCS, consists of a set, WFCs,
575 of atomic weather forecast consumers, WFC.

type
572 WSs = WS-**set**
573 WS
574 WFCs = WFC-**set**
575 WFC
value
572 obs_part_WSs: WSS → WSs
574 obs_part_WFCs: WFCS → WFCs

B.3.2 Unique Identifiers

We shall consider only atomic parts.

576 Every single weather station has a unique identifier.
577 The weather data interpretation (i.e., the weather forecast “creator”) has a unique identifier.
578 Every single weather forecast consumer has a unique identifier.
579 The global clock has a unique identifier.

type
576 WSI
577 WDII
578 WFCI
579 CLKI
value
576 uid_WSI: WS → WSI
577 uid_WDII: WDIS → WDII
578 uid_WFCI: WFC → WFCI
578 uid_CLKI: CLK → CLKI

B.3.3 Mereologies

We shall restrict ourselves to consider the mereologies only of the atomic parts.

580 The mereology of weather stations is the pair of the unique clock identifier and the unique identifier of the weather data interpreter.
581 The mereology of weather data interpreter is the triple of the unique clock identifier, set of unique identifiers of all the weather stations and the set of unique identifiers of all the weather forecast consumers.
582 The mereology of weather forecast consumer is the the pair of the unique clock identifier and the unique identifier of the weather data interpreter.
583 The mereology of the global clock is the triple of the set of all the unique identifiers of weather stations, the unique identifier of the weather data interpreter, and the set of all the unique identifiers of weather forecast consumers.

type
580 WSM = CLKI × WDII
581 WDIM = CLKI × WSI-**set** × WFCI-**set**
582 WFCM = CLKI × WDII
583 CLKM = CLKI × WDGI-**set** × WDII × WFCI-**set**
value
580 mereo_WSM: WS → WSM
581 mereo_WDI: WDI → WDIM
582 mereo_WFC: WFC → WFCM
583 mereo_CLK: CLK → CLKM

B.3.4 Attributes

Clock, Time and Time-intervals

584 The global clock has an autonomous time attribute.
585 Time values are further undefined, but times are considered absolute in the sense as representing some intervals since “the birth of time”, an example, concrete time could be JUNE 17, 2018: 10:12 AM.

586 Time intervals are further undefined, but time intervals can be considered relative in the sense of representing a quantity elapsed between two times, examples are: 1 day 2 hours and 3 minutes, etc. When a time interval, ti, is specified it is always to be understood to designate the times from now, or from a specified time, t, until the time $t+ti$.
587 We postulate $\oplus$, $\ominus$, and can postulate further "arithmetic" operators, and
588 we can postulate relational operators.

type
584 TIME
585 TI
value
584 attr_TIME: CLK $\rightarrow$ TIME
587 $\oplus$: TIME$\times$TI$\rightarrow$TIME, TI$\times$TI$\rightarrow$TI
587 $\ominus$: TIME$\times$TI$\rightarrow$TIME, TIME$\times$TIME$\rightarrow$TI
588 $=, \neq, <, \leq, \geq, >$: TIME$\times$TIME$\rightarrow$**Bool**, TI$\times$TI$\rightarrow$**Bool**, ...

We do not here define these operations and relations.

Locations

589 Locations are metric, topological spaces and can thus be considered dense spaces of three dimensional points.
590 We can speak of one location properly contained ($\subset$) within, or contained or equal ($\subseteq$), or equal ($=$), or not equal ($\neq$) to another location.

type
589. LOC
value
590. $\subset, \subseteq, =, \neq$: LOC $\times$ LOC $\rightarrow$ **Bool**

Weather

591 The weather material is considered a dense, infinite set of weather point volumes WP. Some dense, infinite subsets (still proper volumes) of such points may be liquid, i.e., rain, water in rivers, lakes and oceans. Other dense, infinite subsets (still proper volumes) of such points may be gaseous, i.e., the air, or atmosphere. These two forms of proper volumes "border" along infinite subsets (curved planes, surfaces) of weather points.
592 From the material weather one can observe its location.

type
591 W = WP-**infset**
591 WP
value
592 attr_LOC: W $\rightarrow$ LOC

593 Some meteorological quantities are:

a *Hu*midity,
b *Te*mperature,
c *Wi*nd and
d *Ba*rometric pressure.

594 The weather has an indefinite number of attributes at any one time.
a Humidity distribution, at level (above sea) and by location,
b Temperature distribution, at level (above sea) and by location,

c Wind direction, velocity and mobility of wind center, and by location,
d Barometric pressure, and by location,
e etc., etc.

type
593a Hu
593b Te
593c Wi
593d Ba
594a HDL = LOC $\overrightarrow{m}$ Hu
594b TDL = LOC $\overrightarrow{m}$ Te
594c WDL = LOC $\overrightarrow{m}$ Wi
594d BPL = LOC $\overrightarrow{m}$ Ba
594e ...
value
594a attr_HDL: W → HDL
594b attr_TDL: W → TDL
594c attr_WDL: W → WDL
594d attr_APL: W → BPL
594e ...

Weather Stations

595 Weather stations have static location attributes.
596 Weather stations sample the weather gathering humidity, temperature, wind, barometric pressure, and possibly other data, into time and location stamped weather data.

value
595 attr_LOC: WS → LOC
type
596 WD :: mkWD((TIME×LOC)×(TDL×HDL×WDL×BPL×...))

Weather Data Interpreter

597 There is a programmable attribute: weather data repository, wdr:WDR, of weather data, wd:WD, collected from weather stations.
598 And there is programmable attribute: weather forecast repository, wfr:WFR, of forecasts, wf:WF, disseminate-able to weather forecast consumers.
These repositories are updated when
599 received from the weather stations, respectively when
600 calculated by the weather data interpreter.

type
597 WDR
598 WFR
value
599 update_wdr: TIME × WD → WDR → WDR
600 update_wfr: TIME × WF → WFR → WFR

It is a standard exercise to define these two functions (say algebraically).

Weather Forecasts

601 Weather forecasts are weather forecast format-, time- and location-stamped quantities, the latter referred to as wefo:WeFo.
602 There are a definite number ($n \geq 1$) of weather forecast formats.
603 We do not presently define these various weather forecast formats.
604 They are here thought of as being requested, mkWFReq, by weather forecast consumers.

```
type
601  WF = WFF × (TIME×TI) × LOC × WeFo
602  WFF = WFF1 | WFF2 | ... | WFFn
603  WFF1, WFF2, ..., WFFn
604  WFReq :: mkWFReq(s_wff:WFF,s_ti:(TIME×TI),s_loc:LOC)
```

Weather Forecast Consumer

605 There is a programmable attribute, d:D, D for display (!).
606 Displays can be rendered (RND): visualized, tabularised, made audible, translated (between languages and language dialects, ...), etc.
607 A rendered display can be "abstracted back" into its basic form.
608 Any abstracted rendered display is identical to its abstracted form.

```
type
605  D
606  RND
value
605  attr_D: WFC → D
606  rndr_D: RND × D → D
607  abs_D: D → D
axiom
608  ∀ d:D, r:RND • abs_D(rndr(r,d)) = d
```

B.4 Perdurants

B.4.1 A WIS Context

609 We postulate a given system, wis:WIS. That system is characterized by
610 a dynamic weather
611 and its unique identifier,
612 a set of weather stations
613 and their unique identifiers,
614 a single weather data interpreter
615 and its unique identifier,
616 a set of weather forecast consumers
617 and their unique identifiers, and
618 a single clock
619 and its unique identifier.
620 Given any specific wis:WIS there is [therefore] a full set of part identifiers, is, of weather, clock, all weather stations, the weather data interpreter and all weather forecast consumers.

We list the above-mentioned values. They will be referenced by the channel declarations and the behaviour definitions of this section.

```
value
609  wis:WIS
610  w:W = obs_material_W(wis)
611  wi:WI = uid_WI(w)
612  wss:WSs = obs_part_WSs(obs_part_WSS(wis))
613  wsis:WDGI-set = {uid_WSI(ws)|ws:WS•ws ∈ wss}
614  wdi:WDI = obs_part_WDIS(wis)
615  wdii:WDII = uid_WDII(wdi)
616  wfcs:WFCs = obs_part_WFCs(obs_part_WFCS(wis))
```

```
617 wfcis:WFI-set = {uid_WFCI(wfc)|wfc:WFC•wfc ∈ wfcs}
618 clk:CLK = obs_part_CLK(wis)
619 clki:CLKI = uid_CLKI(clk)
620 is:(WI|WSI|WDII|WFCI)-set = {wi}∪wsis∪{wdii}∪wfcis
```

B.4.2 Channels

621 Weather stations share weather data, WD, with the weather data interpreter — so there is a set of channels, one each, "connecting" weather stations to the weather data interpreter.
622 The weather data interpreter shares weather forecast requests, WFReq, and interpreted weather data (i.e., forecasts), WF, with each and every forecast consumer — so there is a set of channels, one each, "connecting" the weather data interpreter to the interpreted weather data (i.e., forecast) consumers.
623 The clock offers its current time value to each and every part, except the weather, of the WIS system.

```
channel
621 { ch_si[wsi,wdii]:WD | wsi:WSI•wsi ∈ wsis }
622 { ch_ic[wdii,fci]:(WFReq|WF) | fci:FCI•fci ∈ fcis }
623 { ch_cp[clki,i]:TIME | i:(WI|CLKI|WSI|WDII|WFCI)•i ∈ is }
```

B.4.3 WIS Behaviours

624 WIS behaviour, wis_beh, is the
625 parallel composition of all the weather station behaviours, in parallel with the
626 weather data interpreter behaviour, in parallel with the
627 parallel composition of all the weather forecast consumer behaviours, in parallel with the
628 clock behaviour.

```
value
624 wis_beh: Unit → Unit
624 wis_beh() ≡
625    ‖ { ws_beh(uid_WSI(ws),mereo_WS(ws),...) | ws:WS•ws ∈ wss } ‖
626    ‖ wdi_beh(uid_WDI(wdi),mereo_WDI(wdi),...)(wd_rep,wf_rep) ‖
627    ‖ { wfc_beh(uid_WFCI(wfc),mereo_WDG(wfc),...) | wfc:WFC•wfc ∈ wfcs } ‖
628      clk_beh(uid_CLKI(clk),mereo_CLK(clk),...)("June 17, 2018: 10:12 am")
```

B.4.4 Clock

629 The clock behaviour has a programmable attribute, t.
630 It repeatedly offers its current time to any part of the WIS system.
It nondeterministically internally "cycles" between
631 retaining its current time, or
632 increment that time with a "small" time interval, δ, or
633 offering the current time to a requesting part.

```
value
629. clk_beh: clki:CLKI × clkm:CLKM → TIME →
630.      out {ch_cp[clki,i]|i:(WSI|WDII|WFCI)•i ∈wsis∪{wdii}∪wfcis }  Unit
629. clk_beh(clki,is)(t) ≡
631.        clk_beh(clki,is)(t)
632.     ⌈⌉ clk_beh(clki,is)(t⊕δ)
633.     ⌈⌉ ( ⌈⌉⌊⌋{ ch_cp[clki,i] ! t | i:(WSI|WDII|WFCI)•i ∈ is } ; clk_beh(clki,is)(t) )
```

B.4.5 Weather Station

634 The weather station behaviour communicates with the global clock and the weather data interpreter.
635 The weather station behaviour simply “cycles” between sampling the weather, reporting its findings to the weather data interpreter and resume being that overall behaviour.
636 The weather station time-stamp “sample’ the weather (i.e., meteorological information).
637 The meteorological information obtained is analysed with respect to temperature (distribution etc.),
638 humidity (distribution etc.),
639 wind (distribution etc.),
640 barometric pressure (distribution etc.), etcetera,
641 and this is time-stamp and location aggregated (mkWD) and “sent” to the (central ?) weather data interpreter,
642 whereupon the weather data generator behaviour resumes.

```
value
634  ws_beh: wsi:WSI × (clki,wi,wdii):WDGM × (LOC × ...) →
634      in ch_cp[clki,wsi] out ch_gi[wsi,wdii]  Unit
635  ws_beh(wsi,(clki,wi,wdii),(loc,...)) ≡
637      let tdl = attr_TDL(w),
638          hdl = attr_HDL(w),
639          wdl = attr_WDL(w),
640          bpl = attr_BPL(w), ... in
641      ch_gi[wsi,wdii] ! mkWD((ch_cp[clki,wsi] ?,loc),(tdl,hdl,wdl,bpl,...)) end ;
642      wdg_beh(wsi,(clki,wi,wdii),(loc,...))
```

B.4.6 Weather Data Interpreter

643 The weather data interpreter behaviour communicates with the global clock, all the weather stations and all the weather forecast consumers.
644 The weather data interpreter behaviour non-deterministically internally (⊓) chooses to
645 either collect weather data,
646 or calculate some weather forecast,
647 or disseminate a weather forecast.

```
value
643  wdi_beh: wdii:WDII×(clki,wsis,wfcis):WDIM×...→(WD_Rep×WF_Rep)→
643          in ch_cp[clki,wdii], { ch_si[wsi,wdii] | wsi:WSI•wsi ∈ wsis },
643          out { ch_ic[wdii,wfci] | wfci:WFCI•wfci ∈ wfcis }  Unit
643  wdi_beh(wdii,(clki,wsis,wfcis),...)(wd_rep,wf_rep) ≡
645      collect_wd(wdii,(clki,wsis,wfcis),...)(wd_rep,wf_rep)
644      ⊓
646      calculate_wf(wdii,(clki,wsis,wfcis),...)(wd_rep,wf_rep)
644      ⊓
647      disseminate_wf(wdii,(clki,wsis,wfcis),...)(wd_rep,wf_rep)
```

collect_wd

648 The collect weather data behaviour communicates with the global clock and all the weather stations – but “passes-on” the capability to communicate with all of the weather forecast consumers.
649 The collect weather data behaviour
650 non-deterministically externally offers to accept weather data from some weather station,
651 updates the weather data repository with a time-stamped version of that weather data,

652 and resumes being a weather data interpreter behaviour, now with an updated weather data repository.

```
value
648  collect_wd: wdii:WDII×(clki,wsis,wfcis):WDIM×...
648           → (WD_Rep×WF_Rep) →
648           in ch_cp[clki,wdii], { ch_si[wsi,wdii] | wsi:WSI•wsi ∈ wsis },
648           out { ch_ic[wdii,wfci] | wfci:WFCI•wfci ∈ wfcis }  Unit
649  collect_wd(wdii,(clki,wsis,wfcis),...)(wd_rep,wf_rep) ≡
650      let ((ti,loc),(hdl,tdl,wdl,bpl,...)) = ⨅⨆{wsi[wsi,wdii]?|wsi:WSI•wsi∈wsis} in
651      let wd_rep′ = update_wdr(ch_cp[clki,wdii]?,((ti,loc),(hdl,tdl,wdl,bpl,...)))(wd_rep) in
652      wdi_beh(wdii,(clki,wsis,wfcis),...)(wd_rep′,wf_rep) end end
```

calculate_wf

653 The calculate forecast behaviour communicates with the global clock – but "passes-on" the capability to communicate with all of weather stations and the weather forecast consumers.
654 The calculate forecast behaviour
655 non-deterministically internally chooses a forecast type from among a indefinite set of such,
656 and a current or "future" time-interval,
657 whereupon it calculates the weather forecast and updates the weather forecast repository,
658 and then resumes being a weather data interpreter behaviour now with the weather forecast repository updated with the calculated forecast.

```
value
653  calculate_wf: wdii:WDII×(clki,wsis,wfcis):WDIM×...→(WD_Rep×WF_Rep)→
653           in ch_cp[clki,wdii], { ch_si[wsi,wdii] | wsi:WSI•wsi ∈ wsis },
653           out { ch_ic[wdii,wfci] | wfci:WFCI•wfci ∈ wfcis }  Unit
654  calculate_wf(wdii,(clki,wsis,wfcis),...)(wd_rep,wf_rep) ≡
655       let tf:WWF = ft1 ⨅ ft2 ⨅ ... ⨅ ftn,
656           ti:(TIME×TIVAL) • toti≥ch_cp[clki,wdii] ? in
657       let wf_rep′ = update_wfr(calc_wf(tf,ti)(wf_rep)) in
658       wdi_beh(wdii,(clki,wsis,wfcis),...)(wd_rep,wf_rep′) end end
```

659 The calculate_weather forecast function is, at present, further undefined.

```
value
659.  calc_wf: WFF × (TIME×TI) → WFRep → WF
659.  calc_wf(tf,ti)(wf_rep) ≡ ,,,
```

disseminate_wf

660 The disseminate weather forecast behaviour communicates with the global clock and all the weather forecast consumers – but "passes-on" the capability to communicate with all of weather stations.
661 The disseminate weather forecast behaviour non-deterministically externally offers to received a weather forecast request from any of the weather forecast consumers, wfci, that request is for a specific format forecast, tf, and either for a specific time or for a time-interval, toti, as well as for a specific location, loc.
662 The disseminate weather forecast behaviour retrieves an appropriate forecast and
663 sends it to the requesting consumer –
664 whereupon the disseminate weather forecast behaviour resumes being a weather data interpreter behaviour

```
value
660 disseminate_wf: wdii:WDII×(clki,wsis,wfcis):WDIM×...→(WD_Rep×WF_Rep)→
660         in ch_cp[clki,wdii] in,out { ch_ic[wdii,wfci] | wfci:WFCI•wfci ∈ wfcis }  Unit
660 disseminate_wf(wdii,(clki,wsis,wfcis),...)(wd_rep,wf_rep) ≡
661     let mkReqWF((tf,toti,loc),wfci) = ⌈⌉⌊⌋{ch_ic[wdii,wfci] ? | wfci:WFCI•wfci∈wfcis} in
662     let wf = retr_WF((tf,toti,loc),wf_rep) in
663     ch_ic[wdii,wfci] ! wf ;
664     disseminate_wf(wdii,(clki,wsis,wfcis),...)(wd_rep,wf_rep) end end
```

665 The retr_WF((tf,toti,loc),wf_rep) function invocation retrieves the weather forecast from the weather forecast repository most "closely" matching the format, tf, time, toti, and location of the request received from the weather forecast consumer. We do not define this function.

```
665. retr_WF: (WFF×(TIME×TI)×LOC) × WFRep → WF
665. retr_WF((tf,toti,loc),wf_rep) ≡ ...
```

We could have included, in our model, the time-stamping of receipt (formula Item 661) of requests, and the time-stamping of delivery of requested forecast in which case we would insert ch_cp[clki,wdii]? at respective points in formula Items 661 and 663.

B.4.7 Weather Forecast Consumer

666 The weather forecast consumer communicates with the global clock and the weather data interpreter.
667 The weather forecast consumer behaviour
668 nondeterministically internally either
669 selects a suitable weather cast format, tf,
670 selects a suitable location, loc′, and
671 selects, toti, a suitable time (past, present or future) or a time interval (that is supposed to start when forecast request is received by the weather data interpreter.
672 With a suitable formatting of this triple, mkReqWF(tf,loc′,toti), the weather forecast consumer behaviour "outputs" a request for a forecast to the weather data interpreter (first "half" of formula Item 671) whereupon it awaits (;) its response (last "half" of formula Item 671) which is a weather forecast, wf,
673 whereupon the weather forecast consumer behaviour resumes being that behaviour with it programmable attribute, d, being replaced by the received forecast suitably annotated;
668 or the weather forecast consumer behaviour
674 edits a display
675 and resumes being a weather forecast consumer behaviour with the edited programmable attribute, d′.

```
value
666 wfc_beh: wfci:WFCI × (clki,wdii):WFCM × (LOC × ...) → D →
666       in  ch_cp[clki,wfci],
666       in,out { ch_ic[wdii,wfci] | wfci:WFCI•wfci ∈ wfcis }  Unit
667 wfc_beh(wfci,(clki,wdii),(loc,...))(d) ≡
669     let tf = tf1 ⌈⌉ tf2 ⌈⌉ ... ⌈⌉ tfn,
670         loc′:LOC • loc′=loc∨loc′≠loc,
671         (t,ti):(TIME×TI) • ti≥0 in
672     let wf = (ch_ic[wdii,wfci] ! mkReqWF(tf,loc′,(t,ti))) ; ch_ic[wdii,wfci] ? in
673     wfc_beh(wfci,(clki,wdii),(loc,...))((tf,loc′,(t,ti)),wf) end end
668   ⌈⌉
674     let d′:D {\EQ} rndr\_D(d,{\DOTDOTDOT}) in
675     wfc_beh(wfci,(clki,wdii),(loc,...))(d′) end
```

The choice of location may be that of the weather forecast consumer location, or it may be one different from that. The choice of time and time-interval is likewise a non-deterministic internal choice.

B.5 Conclusion

B.5.1 Reference to Similar Work

As far as I know there are no published literature nor, to our knowledge, institutional or private works on the subject of modelling weather data collection, interpretaion and weather forecast delivery systems.

B.5.2 What Have We Achieved ?

TO BE WRITTEN

B.5.3 What Needs to be Done Next ?

TO BE WRITTEN

B.5.4 Acknowledgements

This technical cum experimental research report was begun in Bergen, Wednesday, November 9, 2016 – inspired by a presentation by Ms. Doreen Tuheirwe, Makarere University, Kampala, Uganda. I thank her, and Profs. Magne Haveraaen and Jaakko Järvi of BLDL: the Bergen Language Design Laboratory, Dept. of Informatics, University of Bergen (Norway), for their early comments, and Prof. Haveraaen for inviting me to give PhD lectures there in the week of Nov. 6–12, 2016.

C

Pipeline Systems

Summary

Fig. C.1. The Planned Nabucco Pipeline: http://en.wikipedia.org/wiki/Nabucco_Pipeline

- Named after Verdi's opera
- Gas pipeline
- 3300 kms
- 2011–2014, first gas flow: 2014; 2017–2019, more pipes
- 8 billion Euros
- Max flow: 31 bcmy: billion cubic meters a year
- **http://www.nabucco-pipeline.com/**

C.1 Photos of Pipeline Units and Diagrams of Pipeline Systems

When combining joins and forks we can construct sitches. Figure C.7 on Page 273 shows some actual switches.

Figure C.8 on Page 274 diagrams a generic switch.

[0] See http://en.wikipedia.org/wiki/Nabucco_Pipeline

Fig. C.2. The Planned Nabucco Pipeline: http://en.wikipedia.org/wiki/Nabucco_Pipeline

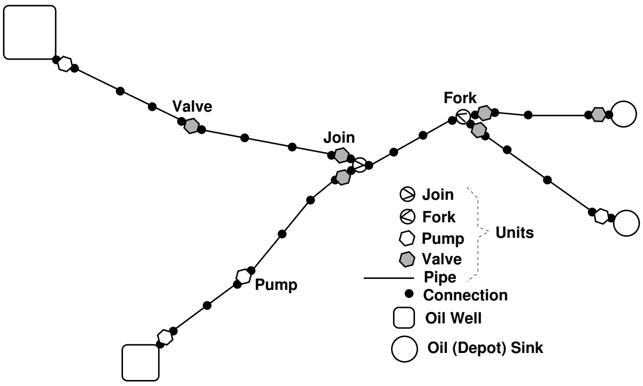


Fig. C.3. An oil pipeline system

C.2 Non-Temporal Aspects of Pipelines

These are some non-temporal aspects of pipelines. nets and units: wells, pumps, pipes, valves, joins, forks and sinks; net and unit attributes; and units states, but not state changes. We omit, in early (i.e., next) chapters, consideration of "pigs" and "pig"-insertion and "pig"-extraction units.

C.2.1 Nets of Pipes, Valves, Pumps, Forks and Joins

676 We focus on nets, $n : N$, of pipes, $\pi : \Pi$, valves, $v : V$, pumps, $p : P$, forks, $f : F$, joins, $j : J$, wells, $w : W$ and sinks, $s : S$.
677 Units, $u : U$, are either pipes, valves, pumps, forks, joins, wells or sinks.
678 Units are explained in terms of disjoint types of PIpes, VAlves, PUmps, FOrks, JOins, WElls and SKs.[1]

type
```
676 N, PI, VA, PU, FO, JO, WE, SK
677 U = Π | V | P | F | J | S| W
677 Π == mkΠ(pi:PI)
677 V == mkV(va:VA)
677 P == mkP(pu:PU)
677 F == mkF(fo:FO)
677 J == mkJ(jo:JO)
677 W == mkW(we:WE)
677 S == mkS(sk:SK)
```

[1] This is a mere specification language technicality.

Fig. C.4. Pipes

Fig. C.5. Valves

C.2.2 Unit Identifiers and Unit Type Predicates

679 We associate with each unit a unique identifier, $ui : UI$.
680 From a unit we can observe its unique identifier.
681 From a unit we can observe whether it is a pipe, a valve, a pump, a fork, a join, a well or a sink unit.

type
  679 UI
value

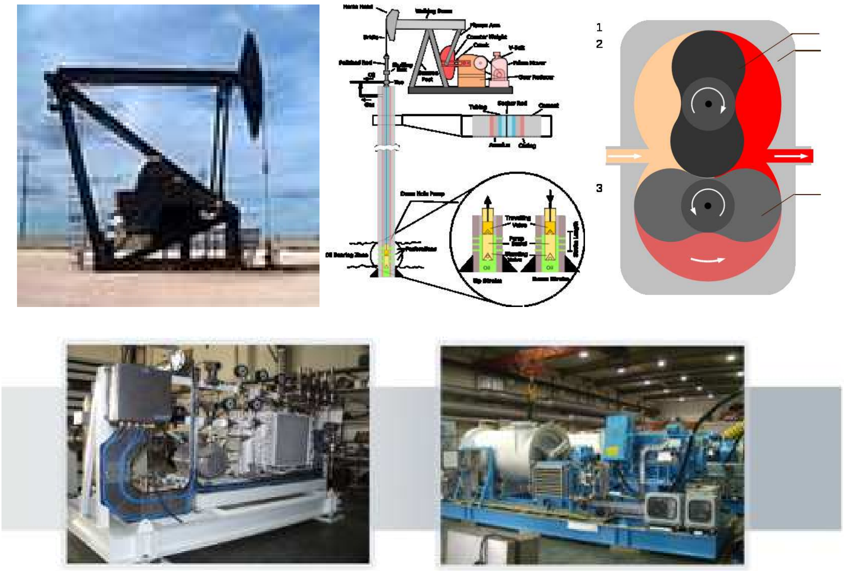


Fig. C.6. Oil Pumps and Gas Compressors

680 obs_UI: U → UI
681 is_Π: U → **Bool**, is_V: U → **Bool**, ..., is_J: U → **Bool**
  is_Π(u) ≡ **case** u **of** mkPI(_) → **true**, _ → **false end**
  is_V(u) ≡ **case** u **of** mkV(_) → **true**, _ → **false end**
  ...
  is_S(u) ≡ **case** u **of** mkS(_) → **true**, _ → **false end**

C.2.3 Unit Connections

A connection is a means of juxtaposing units. A connection may connect two units in which case one can observe the identity of connected units from "the other side".

682 With a pipe, a valve and a pump we associate exactly one input and one output connection.
683 With a fork we associate a maximum number of output connections, *m*, larger than one.
684 With a join we associate a maximum number of input connections, *m*, larger than one.
685 With a well we associate zero input connections and exactly one output connection.
686 With a sink we associate exactly one input connection and zero output connections.

value
  682 obs_InCs,obs_OutCs: Π|V|P → {|1:**Nat**|}
  683 obs_inCs: F → {|1:**Nat**|}, obs_outCs: F → **Nat**
  684 obs_inCs: J → **Nat**, obs_outCs: J → {|1:**Nat**|}
  685 obs_inCs: W → {|0:**Nat**|}, obs_outCs: W → {|1:**Nat**|}
  686 obs_inCs: S → {|1:**Nat**|}, obs_outCs: S → {|0:**Nat**|}
axiom
  683 ∀ f:F • obs_outCs(f) ≥ 2
  684 ∀ j:J • obs_inCs(j) ≥ 2

If a pipe, valve or pump unit is input-connected [output-connected] to zero (other) units, then it means that the unit input [output] connector has been sealed. If a fork is input-connected to zero (other) units, then it means that the fork input connector has been sealed. If a fork is output-connected to *n* units less than the

Fig. C.7. Oil and Gas Switches

maximum fork-connectability, then it means that the unconnected fork outputs have been sealed. Similarly for joins: “the other way around”.

C.2.4 Net Observers and Unit Connections

687 From a net one can observe all its units.
688 From a unit one can observe the the pairs of disjoint input and output units to which it is connected:
  a Wells can be connected to zero or one output unit — a pump.
  b Sinks can be connected to zero or one input unit — a pump or a valve.
  c Pipes, valves and pumps can be connected to zero or one input units and to zero or one output units.
  d Forks, f, can be connected to zero or one input unit and to zero or n, $2 \leq n \leq$obs_Cs(f) output units.
  e Joins, j, can be connected to zero or n, $2 \leq n \leq$obs_Cs(j) input units and zero or one output units.

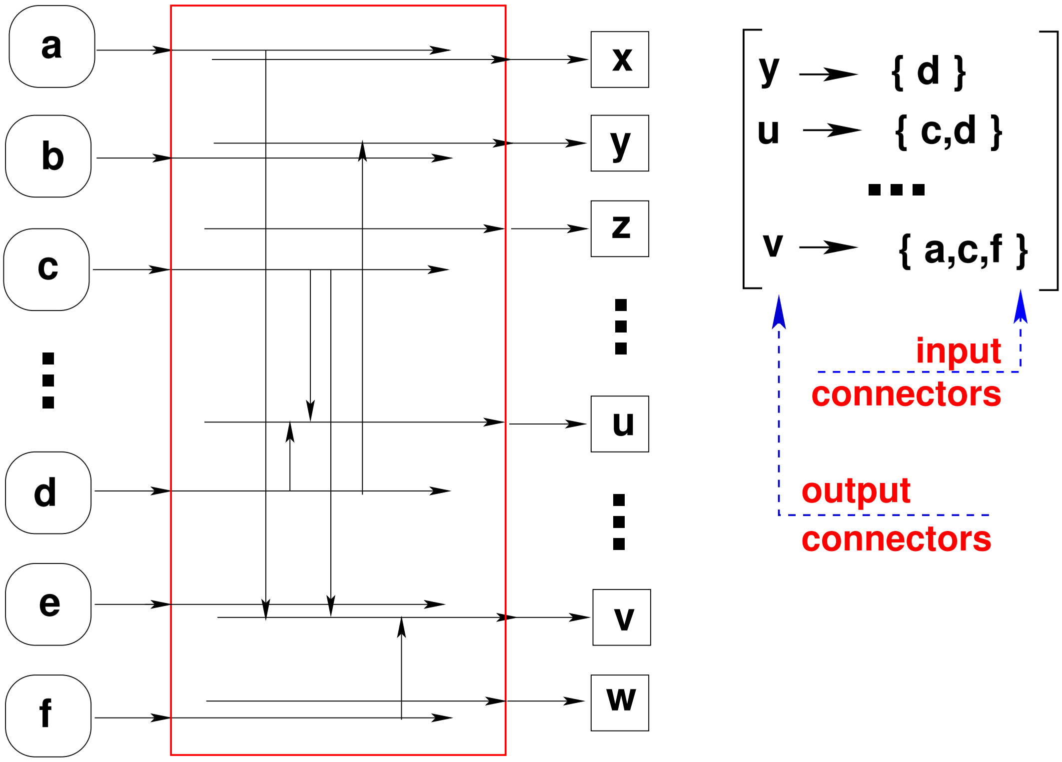


Fig. C.8. A Switch Diagram

value
687 obs_Us: N → U-**set**
688 obs_cUIs: U → UI-**set** × UI-**set**
wf_Conns: U → **Bool**
wf_Conns(u) ≡
let (iuis,ouis) = obs_cUIs(u) **in** iuis ∩ ouis = {} ∧
case u **of**
688a mkW(_) → **card** iuis ∈ {0} ∧ **card** ouis ∈ {0,1},
688b mkS(_) → **card** iuis ∈ {0,1} ∧ **card** ouis ∈ {0},
688c mk*Π*(_) → **card** iuis ∈ {0,1} ∧ **card** ouis ∈ {0,1},
688c mkV(_) → **card** iuis ∈ {0,1} ∧ **card** ouis ∈ {0,1},
688c mkP(_) → **card** iuis ∈ {0,1} ∧ **card** ouis ∈ {0,1},
688d mkF(_) → **card** iuis ∈ {0,1} ∧ **card** ouis ∈ {0}∪{2..obs_inCs(j)},
688e mkJ(_) → **card** iuis ∈ {0}∪{2..obs_inCs(j)} ∧ **card** ouis ∈ {0,1}
end end

C.2.5 Well-formed Nets, Actual Connections

689 The unit identifiers observed by the obs_cUIs observer must be identifiers of units of the net.

axiom
689 ∀ n:N,u:U • u ∈ obs_Us(n) ⇒
689 **let** (iuis,ouis) = obs_cUIs(u) **in**
689 ∀ ui:UI • ui ∈ iuis ∪ ouis ⇒
689 ∃ u′:U • u′ ∈ obs_Us(n) ∧ u′≠u ∧ obs_UI(u′)=ui **end**

Fig. C.9. To be treated in a later version of this report: Pig Launcher, Receiver and New and Old Pigs

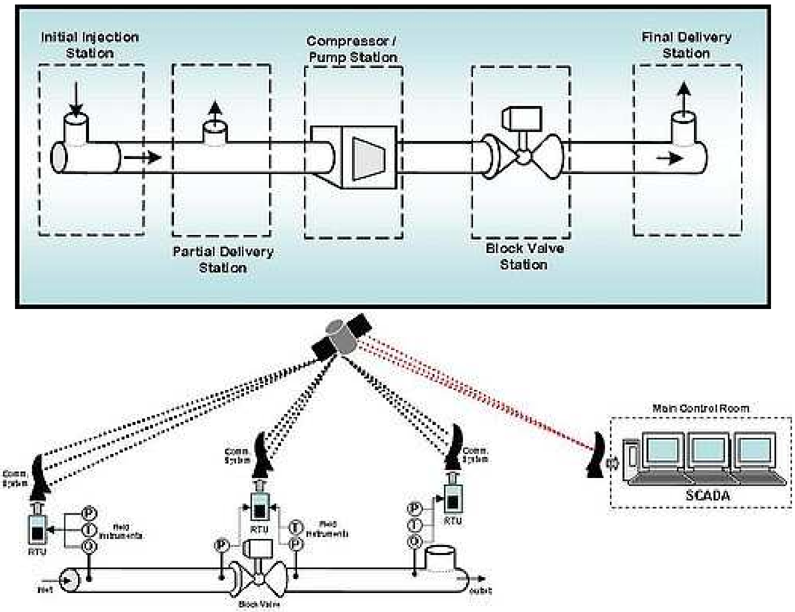


Fig. C.10. Pipeline Diagrams

C.2.6 Well-formed Nets, No Circular Nets

690 By a route we shall understand a sequence of units.
691 Units form routes of the net.

type
  690 $\mathsf{R} = \mathsf{UI}^\omega$
value
  691 routes: N → R-**infset**

```
691 routes(n) ≡
691   let us = obs_Us(n) in
691   let rs = {⟨u⟩|u:U•u ∈ us} ∪ {r⌢r′|r,r′:R• {r,r′}⊆rs∧adj(r,r′)} in
691   rs end end
```

692 A route of length two or more can be decomposed into two routes
693 such that the least unit of the first route “connects” to the first unit of the second route.

value

```
692   adj: R × R → Bool
692   adj(fr,lr) ≡
692     let (lu,fu)=(fr(len fr),hd lr) in
693     let (lui,fui)=(obs_UI(lu),obs_UI(fu)) in
693     let ((_,luis),(fuis,_))=(obs_cUIs(lu),obs_cUIs(fu)) in
693     lui ∈ fuis ∧ fui ∈ luis end end end
```

694 No route must be circular, that is, the net must be acyclic.

value

```
694 acyclic: N → Bool
694   let rs = routes(n) in
694   ∼∃ r:R•r ∈ rs⇒∃ i,j:Nat•{i,j}⊆inds r∧i≠j∧r(i)=r(j) end
```

C.2.7 Well-formed Nets, Special Pairs, wfN_SP

695 We define a “special-pairs” well-formedness function.
 a Fork outputs are output-connected to valves.
 b Join inputs are input-connected to valves.
 c Wells are output-connected to pumps.
 d Sinks are input-connected to either pumps or valves.

value

```
695  wfN_SP: N → Bool
695  wfN_SP(n) ≡
695    ∀ r:R • r ∈ routes(n) in
695      ∀ i:Nat • {i,i+1}⊆inds r ⇒
695        case r(i) of ∧
695a         mkF(_) → ∀ u:U•adj(⟨r(i)⟩,⟨u⟩) ⇒ is_V(u),_→true end ∧
695        case r(i+1) of
695b         mkJ(_) → ∀ u:U•adj(⟨u⟩,⟨r(i)⟩) ⇒ is_V(u),_→true end ∧
695        case r(1) of
695c         mkW(_) → is_P(r(2)),_→true end ∧
695        case r(len r) of
695d         mkS(_) → is_P(r(len r−1))∨is_V(r(len r−1)),_→true end
```

The **true** clauses may be negated by other **case** distinctions’ is_V or is_V clauses.

C.2.8 Special Routes, I

696 A pump-pump route is a route of length two or more whose first and last units are pumps and whose intermediate units are pipes or forks or joins.
697 A simple pump-pump route is a pump-pump route with no forks and joins.

698 A pump-valve route is a route of length two or more whose first unit is a pump, whose last unit is a valve and whose intermediate units are pipes or forks or joins.
699 A simple pump-valve route is a pump-valve route with no forks and joins.
700 A valve-pump route is a route of length two or more whose first unit is a valve, whose last unit is a pump and whose intermediate units are pipes or forks or joins.
701 A simple valve-pump route is a valve-pump route with no forks and joins.
702 A valve-valve route is a route of length two or more whose first and last units are valves and whose intermediate units are pipes or forks or joins.
703 A simple valve-valve route is a valve-valve route with no forks and joins.

```
value
  696-703 ppr,sppr,pvr,spvr,vpr,svpr,vvr,svvr: R → Bool
          pre {ppr,sppr,pvr,spvr,vpr,svpr,vvr,svvr}(n): len n≥2

  696 ppr(r:⟨fu⟩⁀ℓ⁀⟨lu⟩) ≡ is_P(fu) ∧ is_P(lu) ∧ is_πfjr(ℓ)
  697 sppr(r:⟨fu⟩⁀ℓ⁀⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)
  698 pvr(r:⟨fu⟩⁀ℓ⁀⟨lu⟩) ≡ is_P(fu) ∧ is_V(r(len r)) ∧ is_πfjr(ℓ)
  699 sppr(r:⟨fu⟩⁀ℓ⁀⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)
  700 vpr(r:⟨fu⟩⁀ℓ⁀⟨lu⟩) ≡ is_V(fu) ∧ is_P(lu) ∧ is_πfjr(ℓ)
  701 sppr(r:⟨fu⟩⁀ℓ⁀⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)
  702 vvr(r:⟨fu⟩⁀ℓ⁀⟨lu⟩) ≡ is_V(fu) ∧ is_V(lu) ∧ is_πfjr(ℓ)
  703 sppr(r:⟨fu⟩⁀ℓ⁀⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)

  is_πfjr,is_πr: R → Bool
  is_πfjr(r) ≡ ∀ u:U•u ∈ elems r⇒is_Π(u)∨is_F(u)∨is_J(u)
  is_πr(r) ≡ ∀ u:U•u ∈ elems r⇒is_Π(u)
```

C.2.9 Special Routes, II

Given a unit of a route,
704 if they exist (∃),
705 find the nearest pump or valve unit,
706 “upstream” and
707 “downstream” from the given unit.

```
value
  704 ∃UpPoV: U × R → Bool
  704 ∃DoPoV: U × R → Bool
  706 find_UpPoV: U × R ~→ (P|V), pre find_UpPoV(u,r): ∃UpPoV(u,r)
  707 find_DoPoV: U × R ~→ (P|V), pre find_DoPoV(u,r): ∃DoPoV(u,r)
  704 ∃UpPoV(u,r) ≡
  704   ∃ i,j Nat•{i,j}⊆inds r∧i≤j∧{is_V|is_P}(r(i))∧u=r(j)
  704 ∃DoPoV(u,r) ≡
  704   ∃ i,j Nat•{i,j}⊆inds r∧i≤j∧u=r(i)∧{is_V|is_P}(r(j))
  706 find_UpPoV(u,r) ≡
  706   let i,j:Nat•{i,j}⊆indsr∧i≤j∧{is_V|is_P}(r(i))∧u=r(j) in r(i) end
  707 find_DoPoV(u,r) ≡
  707   let i,j:Nat•{i,j}⊆indsr∧i≤j∧u=r(i)∧{is_V|is_P}(r(j)) in r(j) end
```

C.3 State Attributes of Pipeline Units

By a state attribute of a unit we mean either of the following three kinds: (i) the open/close states of valves and the pumping/not_pumping states of pumps; (ii) the maximum (laminar) oil flow characteristics of all units; and (iii) the current oil flow and current oil leak states of all units.

708 Oil flow, $\phi : \Phi$, is measured in volume per time unit.
709 Pumps are either pumping or not pumping, and if not pumping they are closed.
710 Valves are either open or closed.
711 Any unit permits a maximum input flow of oil while maintaining laminar flow. We shall assume that we need not be concerned with turbulent flows.
712 At any time any unit is sustaining a current input flow of oil (at its input(s)).
713 While sustaining (even a zero) current input flow of oil a unit leaks a current amount of oil (within the unit).

```
type
  708  Φ
  709  PΣ == pumping | not_pumping
  709  VΣ == open | closed
value
          −,+: Φ × Φ → Φ, <,=,>: Φ × Φ → Bool
  709       obs_PΣ: P → PΣ
  710       obs_VΣ: V → VΣ
  711–713  obs_LamiΦ.obs_CurrΦ,obs_LeakΦ: U → Φ
  is_Open: U → Bool
    case u of
      mkΠ(_)→true,mkF(_)→true,mkJ(_)→true,mkW(_)→true,mkS(_)→true,
      mkP(_)→obs_PΣ(u)=pumping,
      mkV(_)→obs_VΣ(u)=open
    end
  acceptable_LeakΦ, excessive_LeakΦ: U → Φ
axiom
  ∀ u:U • excess_LeakΦ(u) > accept_LeakΦ(u)
```

C.3.1 Flow Laws

The sum of the current flows into a unit equals the the sum of the current flows out of a unit minus the (current) leak of that unit. This is the same as the current flows out of a unit equals the current flows into a unit minus the (current) leak of that unit. The above represents an interpretation which justifies the below laws.

714 When, in Item 712, for a unit u, we say that at any time any unit is sustaining a current input flow of oil, and when we model that by obs_CurrΦ(u) then we mean that obs_CurrΦ(u) - obs_LeakΦ(u) represents the flow of oil from its outputs.

```
value
  714    obs_inΦ: U → Φ
  714    obs_inΦ(u) ≡ obs_CurrΦ(u)
  714    obs_outΦ: U → Φ
law:
  714    ∀ u:U • obs_outΦ(u) = obs_CurrΦ(u)−obs_LeakΦ(u)
```

715 Two connected units enjoy the following flow relation:
  a If

i two pipes, or
ii a pipe and a valve, or
iii a valve and a pipe, or
iv a valve and a valve, or
v a pipe and a pump, or
vi a pump and a pipe, or
vii a pump and a pump, or
viii a pump and a valve, or
ix a valve and a pump

are immediately connected

b then
  i the current flow out of the first unit's connection to the second unit
  ii equals the current flow into the second unit's connection to the first unit

law:

715a ∀ u,u′:U • {is_Π,is_V,is_P,is_W}(u′|u″) ∧ adj(⟨u⟩,⟨u′⟩)
715a   is_Π(u)∨is_V(u)∨is_P(u)∨is_W(u) ∧
715a   is_Π(u′)∨is_V(u′)∨is_P(u′)∨is_S(u′)
715b   ⇒ obs_outΦ(u)=obs_inΦ(u′)

A similar law can be established for forks and joins. For a fork output-connected to, for example, pipes, valves and pumps, it is the case that for each fork output the out-flow equals the in-flow for that output-connected unit. For a join input-connected to, for example, pipes, valves and pumps, it is the case that for each join input the in-flow equals the out-flow for that input-connected unit. We leave the formalisation as an exercise.

C.3.2 Possibly Desirable Properties

716 Let r be a route of length two or more, whose first unit is a pump, *p*, whose last unit is a valve, *v* and whose intermediate units are all pipes: if the pump, *p* is pumping, then we expect the valve, *v*, to be open.

717 Let r be a route of length two or more, whose first unit is a pump, *p*, whose last unit is another pump, p' and whose intermediate units are all pipes: if the pump, *p* is pumping, then we expect pump p'', to also be pumping.

718 Let r be a route of length two or more, whose first unit is a valve, *v*, whose last unit is a pump, *p* and whose intermediate units are all pipes: if the valve, *v* is closed, then we expect pump *p*, to not be pumping.

719 Let r be a route of length two or more, whose first unit is a valve, v', whose last unit is a valve, v'' and whose intermediate units are all pipes: if the valve, v' is in some state, then we expect valve v'', to also be in the same state.

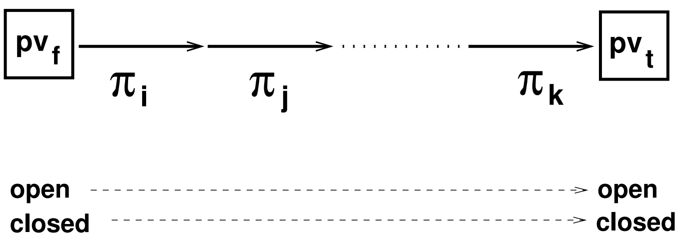


Fig. C.11. pv: Pump or valve, π: pipe

desirable properties:

716 ∀ r:R • spvr(r) ∧
716   **spvr_prop(r):** obs_PΣ(**hd** r)=pumping ⇒ obs_PΣ(r(**len** r))=open

```
717 ∀ r:R • sppr(r) ∧
717   sppr_prop(r): obs_PΣ(hd r)=pumping⇒obs_PΣ(r(len r))=pumping

718 ∀ r:R • svpr(r) ∧
718   svpr_prop(r): obs_PΣ(hd r)=open⇒obs_PΣ(r(len r))=pumping

719 ∀ r:R • svvr(r) ∧
719   svvr_prop(r): obs_PΣ(hd r)=obs_PΣ(r(len r))
```

C.4 Pipeline Actions

C.4.1 Simple Pump and Valve Actions

720 Pumps may be set to pumping or reset to not pumping irrespective of the pump state.
721 Valves may be set to be open or to be closed irrespective of the valve state.
722 In setting or resetting a pump or a valve a desirable property may be lost.

```
value
  720 pump_to_pump, pump_to_not_pump: P → N → N
  721 valve_to_open, valve_to_close: V → N → N
```

```
value
  720 pump_to_pump(p)(n) as n′
  720   pre p ∈ obs_Us(n)
  720   post let p′:P•obs_UI(p)=obs_UI(p′) in
  720          obs_PΣ(p′)=pumping∧else_equal(n,n′)(p,p′) end
  720 pump_to_not_pump(p)(n) as n′
  720   pre p ∈ obs_Us(n)
  720   post let p′:P•obs_UI(p)=obs_UI(p′) in
  720          obs_PΣ(p′)=not_pumping∧else_equal(n,n′)(p,p′) end
  721 valve_to_open(v)(n) as n′
  720   pre v ∈ obs_Us(n)
  721   post let v′:V•obs_UI(v)=obs_UI(v′) in
  720          obs_VΣ(v′)=open∧else_equal(n,n′)(v,v′) end
  721 valve_to_close(v)(n) as n′
  720   pre v ∈ obs_Us(n)
  721   post let v′:V•obs_UI(v)=obs_UI(v′) in
  720          obs_VΣ(v′)=close∧else_equal(n,n′)(v,v′) end
```

```
value
  else_equal: (N×N) → (U×U) → Bool
  else_equal(n,n′)(u,u′) ≡
     obs_UI(u)=obs_UI(u′)
   ∧ u ∈ obs_Us(n)∧u′ ∈ obs_Us(n′)
   ∧ omit_Σ(u)=omit_Σ(u′)
   ∧ obs_Us(n)\{u}=obs_Us(n)\{u′}
   ∧ ∀ u″:U•u″ ∈ obs_Us(n)\{u} ≡ u″ ∈ obs_Us(n′)\{u′}

  omit_Σ: U → U_no_state −−− "magic" function
```

```
    =: U_no_state × U_no_state → Bool
axiom
    ∀ u,u′:U•omit_Σ(u)=omit_Σ(u′) ≡ obs_UI(u)=obs_UI(u′)
```

C.4.2 Events

Unit Handling Events

723 Let n be any acyclic net.
723. If there exists p, p', v, v', pairs of distinct pumps and distinct valves of the net,
723. and if there exists a route, r, of length two or more of the net such that
724 all units, u, of the route, except its first and last unit, are pipes, then
725 if the route "spans" between p and p' and the *simple desirable property*, sppr(r), does not hold for the route, then we have a possibly undesirable event — that occurred as soon as sppr(r) did not hold;
726 if the route "spans" between p and v and the *simple desirable property*, spvr(r), does not hold for the route, then we have a possibly undesirable event;
727 if the route "spans" between v and p and the *simple desirable property*, svpr(r), does not hold for the route, then we have a possibly undesirable event; and
728 if the route "spans" between v and v' and the *simple desirable property*, svvr(r), does not hold for the route, then we have a possibly undesirable event.

```
events:
    723  ∀ n:N • acyclic(n) ∧
    723    ∃ p,p′:P,v,v′:V • {p,p′,v,v′}⊆obs_Us(n)⇒
    723      ∧ ∃ r:R • routes(n) ∧
    724         ∀ u:U • u ∈ elems(r)\{hd r,r(len r)} ⇒ is_Π(i) ⇒
    725           p=hd r∧p′=r(len r) ⇒ ∼sppr_prop(r) ∧
    726           p=hd r∧v=r(len r) ⇒ ∼spvr_prop(r) ∧
    727           v=hd r∧p=r(len r) ⇒ ∼svpr_prop(r) ∧
    728           v=hd r∧v′=r(len r) ⇒ ∼svvr_prop(r)
```

Foreseeable Accident Events

A number of foreseeable accidents may occur.

729 A unit ceases to function, that is,
  a a unit is clogged,
  b a valve does not open or close,
  c a pump does not pump or stop pumping.
730 A unit gives rise to excessive leakage.
731 A well becomes empty or a sunk becomes full.
732 A unit, or a connected net of units gets on fire.
733 Or a number of other such "accident".

C.4.3 Well-formed Operational Nets

734 A well-formed operational net
735 is a well-formed net
  a with at least one well, w, and at least one sink, s,
  b and such that there is a route in the net between w and s.

value

```
734   wf_OpN: N → Bool
734   wf_OpN(n) ≡
735     satisfies axiom 689 on Page 274 ∧ acyclic(n): Item 694 on Page 276 ∧
735     wfN_SP(n): satisfies flow laws, 714 on Page 278 and 715 on Page 278 ∧
735a     ∃ w:W,s:S • {w,s}⊆obs_Us(n) ⇒
735b       ∃ r:R• ⟨w⟩^r^⟨s⟩ ∈ routes(n)
```

C.4.4 Orderly Action Sequences

Initial Operational Net

736 Let us assume a notion of an initial operational net.
737 Its pump and valve units are in the following states
  a all pumps are not_pumping, and
  b all valves are closed.

value

```
736   initial_OpN: N → Bool
737   initial_OpN(n) ≡ wf_OpN(n) ∧
737a     ∀ p:P • p ∈ obs_Us(n) ⇒ obs_PΣ(p)=not_pumping ∧
737b     ∀ v:V • v ∈ obs_Us(n) ⇒ obs_VΣ(p)=closed
```

Oil Pipeline Preparation and Engagement

738 We now wish to prepare a pipeline from some well, $w : W$, to some sink, $s : S$, for flow.
  a We assume that the underlying net is operational wrt. w and s, that is, that there is a route, r, from w to s.
  b Now, an orderly action sequence for engaging route r is to "work backwards", from s to w
  c setting encountered pumps to pumping and valves to open.

In this way the system is well-formed wrt. the desirable sppr, spvr, svpr and svvr properties. Finally, setting the pump adjacent to the (preceding) well starts the system.

value

```
738    prepare_and_engage: W × S → N ~→ N
738    prepare_and_engage(w,s)(n) ≡
738a     let r:R • ⟨w⟩^r^⟨s⟩ ∈ routes(n) in
738b     action_sequence(⟨w⟩^r^⟨s⟩)(len⟨w⟩^r^⟨s⟩)(n) end
738     pre ∃ r:R • ⟨w⟩^r^⟨s⟩ ∈ routes(n)

738c   action_sequence: R → Nat → N → N
738c   action_sequence(r)(i)(n) ≡
738c     if i=1 then n else
738c     case r(i) of
738c       mkV(_) → action_sequence(r)(i−1)(valve_to_open(r(i))(n)),
738c       mkP(_) → action_sequence(r)(i−1)(pump_to_pump(r(i))(n)),
738c       _ → action_sequence(r)(i−1)(n)
738c     end end
```

C.4.5 Emergency Actions

739 If a unit starts leaking excessive oil
  a then nearest up-stream valve(s) must be closed,
  b and any pumps in-between this (these) valves and the leaking unit must be set to not_pumping — following an orderly sequence.

740 If, as a result, for example, of the above remedial actions, any of the desirable properties cease to hold
  a then — a ha !
  b Left as an exercise.

C.5 Connectors

The interface , that is, the possible "openings", between adjacent units have not been explored. Likewise the for the possible "openings" of "begin" or "end" units, that is, units not having their input(s), respectively their "output(s)" connected to anything, but left "exposed" to the environment. We now introduce a notion of connectors: abstractly you may think of connectors as concepts, and concretely as "fittings" with bolts and nuts, or "weldings", or "plates" inserted onto "begin" or "end" units.

741 There are connectors and connectors have unique connector identifiers.
742 From a connector one can observe its uniwue connector identifier.
743 From a net one can observe all its connectors
744 and hence one can extract all its connector identifiers.
745 From a connector one can observe a pair of "optional" (distinct) unit identifiers:
  a An optional unit identifier is
  b either a unit identifier of some unit of the net
  c or a ``nil'' "identifier".
746 In an observed pair of "optional" (distinct) unit identifiers
- there can not be two ``nil'' "identifiers".
- or the possibly two unit identifiers must be distinct

type
  741 K, KI
value
  742 obs_KI: K → KI
  743 obs_Ks: N → K-**set**
  744 xtr_KIS: N → KI-**set**
  744 xtr_KIs(n) ≡ {obs_KI(k)|k:K•k ∈ obs_Ks(n)}
type
  745 oUIp′ = (UI|{|nil|})×(UI|{|nil|})
  745 oUIp = {|ouip:oUIp′•wf_oUIp(ouip)|}
value
  745 obs_oUIp: K → oUIp
  746 wf_oUIp: oUIp′ → **Bool**
  746 wf_oUIp(uon,uon′) ≡
  746   uon=nil⇒uon′≠nil∨uon′=nil⇒uon≠nil∨uon≠uon′

747 Under the assumption that a fork unit cannot be adjacent to a join unit
748 we impose the constraint thet no two distinct connectors feature the same pair of actual (distinct) unit identifiers.
749 The first proper unit identifier of a pair of "optional" (distinct) unit identifiers must identify a unit of the net.
750 The second proper unit identifier of a pair of "optional" (distinct) unit identifiers must identify a unit of the net.

axiom

747 $\forall$ n:N,u,u′:U•{u.u′}⊆obs_Us(n)∧adj(u,u′)⇒ ∼(is_F(u)∧is_J(u′))

748 $\forall$ k,k′:K•obs_KI(k)≠obs_KI(k′)⇒
  case (obs_oUIp(k),obs_oUIp(k′)) **of**
    ((nil,ui),(nil,ui′)) → ui≠ui′,
    ((nil,ui),(ui′,nil)) → **false**,
    ((ui,nil),(nil,ui′)) → **false**,
    ((ui,nil),(ui′,nil)) → ui≠ui′,
    _ → **false**
  end

$\forall$ n:N,k:K•k ∈ obs_Ks(n) ⇒
  case obs_oUIp(k) **of**
749 (ui,nil) → ∃UI(ui)(n)
750 (nil,ui) → ∃UI(ui)(n)
749-750 (ui,ui′) → ∃UI(ui)(n)∧∃UI(ui′)(n)
  end

value

∃UI: UI → N → **Bool**
∃UI(ui)(n) ≡ ∃ u:U•u ∈ obs_Us(n)∧obs_UI(u)=ui

C.6 On Temporal Aspects of Pipelines

The else_qual(u,u′)(n,n′) function definition represents a gross simplification. It ignores the actual flow which changes as a result of setting alternate states, and hence the net state. We now wish to capture the dynamics of flow. We shall do so using the Duration Calculus — a continuous time, integral temporal logic that is semantically and proof system "integrated" with RSL:

> Zhou ChaoChen and Michael Reichhardt Hansen
> Duration Calculus: A Formal Approach to Real-time Systems
> Monographs in Theoretical Computer Science
> The EATCS Series
> Springer 2004

C.7 A CSP Model of Pipelines

We recapitulate Sect. C.5 — now adding connectors to our model:

751 From an oil pipeline system one can observe units and connectors.
752 Units are either well, or pipe, or pump, or valve, or join, or fork or sink units.
753 Units and connectors have unique identifiers.
754 From a connector one can observe the ordered pair of the identity of the two from-, respectively to-units that the connector connects.

type
751 OPLS, U, K
753 UI, KI
value
751 obs_Us: OPLS → U-**set**, obs_Ks: OPLS → K-**set**

752 is_WeU, is_PiU, is_PuU, is_VaU,
752 is_JoU, is_FoU, is_SiU: U → **Bool** [mutually exclusive]
753 obs_UI: U → UI, obs_KI: K → KI
754 obs_UIp: K → (UI|{nil}) × (UI|{nil})

Above, we think of the types OPLS, U, K, UI and KI as denoting semantic entities. Below, in the next section, we shall consider exactly the same types as denoting syntactic entities !

755 There is given an oil pipeline system, opls.
756 To every unit we associate a CSP behaviour.
757 Units are indexed by their unique unit identifiers.
758 To every connector we associate a CSP channel.
Channels are indexed by their unique "k" onnector identifiers.
759 Unit behaviours are cyclic and over the state of their (static and dynamic) attributes, represented by u.
760 Channels, in this model, have no state.
761 Unit behaviours communicate with neighbouring units — those with which they are connected.
762 Unit functions, $\mathcal{U}_i$, change the unit state.
763 The pipeline system is now the parallel composition of all the unit behaviours.

Editorial Remark: Our use of the term unit and the RSL literal **Unit** may seem confusing, and we apologise. The former, unit, is the generic name of a well, pipe, or pump, or valve, or join, or fork, or sink. The literal **Unit**, in a function signature, before the → "announces" that the function takes no argument.[2] The literal **Unit**, in a function signature, after the → "announces", as used here, that the function never terminates.

value
755 opls:OPLS
channel
758 {ch[ki]|k:KI,k:K•k ∈ obs_Ks(opls)∧ki=obs_KI(k)} M
value
763 pipeline_system: **Unit** → **Unit**
763 pipeline_system() ≡
756 ‖ {unit(ui)(u)|u:U•u ∈ obs_Us(opls)∧ui=obs_UI(u)}

757 unit: ui:UI → U →
761 **in**,**out** {ch[ki]|k:K,ki:KI•k ∈ obs_Ks(opls)∧ki=obs_KI(k)∧
761 **let** (ui′,ui″)=obs_UIp(k) **in** ui ∈{ui′,ui″}\{nil} **end**} **Unit**
759 unit(ui)(u) ≡ **let** u′ = $\mathcal{U}_i$(ui)(u) **in** unit(ui)(u′) **end**

762 $\mathcal{U}_i$: ui:UI → U →
762 **in**,**out** {ch[ki]|k:K,ki:KI•k ∈ obs_Ks(opls)∧ki=obs_KI(k)∧
762 **let** (ui′,ui″)=obs_UIp(k) **in** ui ∈{ui′,ui″}\{nil} **end**} U

C.8 Conclusion

We have shown draft sketches of aspects of gas/oil pipelines. From a comprehensive such domain description we can systematically "derive" a set of complementary or alternative requirements prescriptions for the monitoring and control of individual pipe units, as well as of consolidated pipelines. Etcetera !

[2] **Unit** is a type name; () is the only value of type **Unit**.

D

A Document System

We domain analyse and suggest a description of a domain of documents. We emphasize that the model is one of several possible. Common to these models is that we model "all" we can say about documents – irrespective of whether it can also be "implemented" ! The model(s) are not requirements prescriptions – but we can develop such from our domain description.

Yiu may find that the model is overly detailed with respect to a number of "operations" and properties of documents. We find that these operations must be part of the very basis of a document domain in order to cope with documents such as they occur in, for example, public government, see Appendix sect. D.17, or in urban planning, see Appendix Sect. D.18.

D.1 Introduction

We analyse a notion of documents. Documents such as they occur in daily life. What can we say about documents – regardless of whether we can actually provide compelling evidence for what we say ! That is: we model documents, not as electronic entities — which they are becoming, more-and-more, but as if they were manifest entities. When we, for example, say that *"this document was recently edited by such-and-such and the changes of that editing with respect to the text before is such-and-such"*, then we can, of course, always claim so, even if it may be difficult or even impossible to verify the claim. It is a fact, although maybe not demonstrably so, that there was a version of any document before an edit of that document. It is a fact that some handler did the editing. It is a fact that the editing took place at (or in) exactly such-and-such a time (interval), etc. We model such facts.

This research note unravels its analysis &[1] description in stages.

D.2 A System for Managing, Archiving and Handling Documents

The title of this section: *A System for Managing, Archiving and Handling Documents* immediately reveals the major concepts: That we are dealing with a *system* that **manages**, **archives** and **handles documents.** So what do we mean by **managing, archiving** and **handling** documents, and by **documents** ? We give an ultra short survey. The survey relies on your prior knowledge of what you think documents are ! **Management** decides[2] to direct **handler**s to work on **document**s. **Management** first directs the document archive to **create document**s. The document **archive creates document**s, as requested by **management**, and informs management of the **unique document identifiers** (by means of which handlers can handle these documents). **Management** then **grant**s its designated **handler**(s) **access rights** to **document**s, these access rights enable handlers to **edit, read** and **copy** documents. The **handler**s' **edit**ing and **read**ing of **document**s is accomplished by the **handler**s "working directly" with the **document**s (i.e., synchronising

[1] We use the logogram & between two terms, A & B, when we mean to express one meaning.

[2] How these decisions come about is not shown in this research note – as it has nothing to do with the essence of document handling, but, perhaps, with 'management'.

and communicating with **document behaviours**). The **handlers**' **copy**ing of **document**s is accomplished by the **handler**s requesting **management**, in collaboration with the **archive** behaviour, to do so.

D.3 Principal Endurants

By an *endurant* we shall understand *"an entity that can be observed or conceived and described as a "complete thing" at no matter which given snapshot of time."* Were we to "freeze" time we would still be able to observe the entire endurant. This characterisation of what we mean by an 'endurant' is from [2, Manifest Domains: Analysis & Description]. We begin by identifying the principal endurants.

764 From document handling systems one can observe aggregates of handlers and documents.
We shall refer to 'aggregates of handlers' by M, for management, and to 'aggregates of documents' by A, for archive.
765 From aggregates of handlers (i.e., M) we can observe sets of handlers (i.e., H).
766 From aggregates of documents (i.e., A) we can observe sets of documents (i.e., D).

type
764 S, M, A
value
764 obs_M: S → M
764 obs_A: S → A
type
765 H, Hs = H-**set**
766 D, Ds = D-**set**
value
765 obs_Hs: M → Hs
766 obs_Ds: A → Ds

D.4 Unique Identifiers

The notion of unique identifiers is treated, at length, in [2, Manifest Domains: Analysis & Description].

767 We associate unique identifiers with aggregate, handler and document endurants.
768 These can be observed from respective parts[3].

type
767 MI[4], AI[5], HI, DI
value
768 uid_MI[6]: M → MI
768 uid_AI[7]: A → AI
768 uid_HI: H → HI
768 uid_DI: D → DI

As reasoned in [2, Manifest Domains: Analysis & Description], the unique identifiers of endurant parts are indeed unique: No two parts, whether composite, as are the aggregates, or atomic, as are handlers and documents, can have the same unique identifiers.

[3] [2, Manifest Domains: Analysis & Description] explains how 'parts' are the discrete endurants with which we associate the full complement of properties: unique identifiers, mereology and attributes.
[4] We shall not, in this research note, make use of the (one and only) management identifier.
[5] We shall not, in this research note, make use of the (one and only) archive identifier.
[6] Cf. Footnote 4: hence we shall not be using the uid_MI observer.
[7] Cf. Footnote 5: hence we shall not be using the uid_AI observer.

D.5 Documents: A First View

A document is a written, drawn, presented, or memorialized representation of thought. The word originates from the Latin documentum, which denotes a "teaching" or "lesson".[8] We shall, for this research note, take a document in its written and/or drawn form. In this section we shall survey the concept a documents.

D.5.1 Document Identifiers

Documents have *unique identifiers*. If two or more documents have the same document identifier then they are the same, one (and not two or more) document(s).

D.5.2 Document Descriptors

With documents we associate *document descriptors*. We do not here stipulate what document descriptors are other than saying that when a document is **create**d it is provided with a descriptor and this descriptor "remains" with the document and never changes value. In other words, it is a static attribute.[9] We do, however, include, in document descriptors, that the document they describe was initially based on a set of zero, one or more documents – identified by their unique identifiers.

D.5.3 Document Annotations

With documents we also associate *document annotations*. By a document annotation we mean a programmable attribute, that is, an attribute which can be 'augmented' by document handlers. We think of document annotations as "incremental", that is, as "adding" notes "on top of" previous notes. Thus we shall model document annotations as a repository: notes are added, i.e., annotations are augmented, previous notes are not edited, and no notes are deleted. We suggest that notes be time-stamped. The notes (of annotations) may be such which record handlers work on documents. Examples could be: *"June 17, 2018: 10:12 am: This is version V."*, *"This document was released on June 17, 2018: 10:12 am."*, *"June 17, 2018: 10:12 am: Section X.Y.Z of version III was deleted."*, *"June 17, 2018: 10:12 am: References to documents* doc_i *and* doc_j *are inserted on Pages* p *and* q*, respectively."* and *"June 17, 2018: 10:12 am: Final release."*

D.5.4 Document Contents: Text/Graphics

The main idea of a document, to us, is the *written* (i.e., text) and/or *drawn* (i.e., graphics) *contents*. We do not characterise any format for this *contents*. We may wish to insert, in the *contents*, references to locations in the *contents* of other documents. But, for now, we shall not go into such details. The main operations on documents, to us, are concerned with: their **creation, editing, reading, copying** and **shredding**. The **editing** and **reading** operations are mainly concerned with document *annotations* and *text/graphics*.

D.5.5 Document Histories

So documents are **create**d, **edit**ed, **read**, **copied** and **shred**ed. These operations are initiated by the management (**create**), by the archive (**create**), and by handlers (**edit, read, copy**), and at specific times.

[8] From: https://en.wikipedia.org/wiki/Document

[9] You may think of a document descriptor as giving the document a title; perhaps one or more authors; perhaps a physical address (of, for example, these authors); an initial date; as expressing whether the document is a research, or a technical report, or other; who is issuing the document (a public institution, a private firm, an individual citizen, or other); etc.

D.5.6 A Summary of Document Attributes

769 As separate attributes of documents we have document descriptors, document annotations, document contents and document histories.
770 Document annotations are lists of document notes.
771 Document histories are lists of time-stamped document operation designators.
772 A document operation designator is either a create, or an edit, or a read, or a copy, or a shred designator.
773 A create designator identifies
  a a handler and a time (at which the create request first arose), and presents
  b elements for constructing a document descriptor, one which
    i besides some further undefined information
    ii refers to a set of documents (i.e., embeds reference to their unique identifiers),
  c a (first) document note, and
  d an empty document contents.
774 An edit designator identifies a handler, a time, and specifies a pair of edit/undo functions.
775 A read designator identifies a handler.
776 A copy designator identifies a handler, a time, the document to be copied (by its unique identifier, and a document note to be inserted in both the master and the copy document.
777 A shred designator identifies a handler.
778 An edit function takes a triple of a document annotation, a document note and document contents and yields a pair of a document annotation and a document contents.
779 An undo function takes a pair of a document note and document contents and yields a triple of a document annotation, a document note and a document contents.
780 Proper pairs of (edit,undo) functions satisfy some inverse relation.

There is, of course, no need, in any document history, to identify the identifier of that document.

```
type
769     DD, DA, DC, DH
value
769     attr_DD: D → DD
769     attr_DA: D → DA
769     attr_DC: D → DC
769     attr_DH: D → DH
type
770     DA = DN*
771     DH = (TIME × DO)*
772     DO == Crea | Edit | Read | Copy | Shre
773    Crea :: (HI × TIME) × (DI-set × Info) × DN × {|"empty_DC"|}
773(b)i    Info = ...
value
773(b)ii    embed_DIs_in_DD: DI-set × Info → DD
axiom
773d    "empty_DC" ∈ DC
type
774    Edit :: (HI × TIME) × (EDIT × UNDO)
775    Read :: (HI × TIME) × DI
776    Copy :: (HI × TIME) × DI × DN
777    Shre :: (HI × TIME) × DI
778    EDIT = (DA × DN × DC) → (DA × DC)
779    UNDO = (DA × DC) → (DA × DN × DC)
axiom
780    ∀ mkEdit(_,(e,u)):Edit •
780        ∀ (da,dn,dc):(DA×DN×DC) •
780            u(e(da,dn,dc))=(da,dn,dc)
```

D.6 Behaviours: An Informal, First View

In [2, Manifest Domains: Analysis & Description] we show that we can associate behaviours with parts, where parts are such discrete endurants for which we choose to model all its observable properties: unique identifiers, mereology and attributes, and where behaviours are sequences of actions, events and behaviours.

- The overall document handler system behaviour can be expressed in terms of the parallel composition of the behaviours
  781 of the system core behaviour,
  782 of the handler aggregate (the management) behaviour
  783 and the document aggregate (the archive) behaviour,
  with the (distributed) parallel composition of
  784 all the behaviours of handlers and,
  the (distributed) parallel composition of
  785 at any one time, zero, one or more behaviours of documents.
- To express the latter
  786 we need introduce two "global" values: an indefinite set of handler identifiers and an indefinite set of document identifiers.

value
786 his:HI-**set**, dis:DI-**set**

781 sys(...)
782 ‖ mgtm(...)
783 ‖ arch(...)
784 ‖ ‖{hdlr_i(...)|i:HI•i∈his}
785 ‖ ‖{docu_i(dd)(da,dc,dh)|i:DI•i∈dis}

For now we leave undefined the arguments, (...) etc., of these behaviours. The arguments of the document behaviour, (dd)(da,dc,dh), are the static, respectively the three programmable (i.e., dynamic) attributes: *document descriptor*, *document annotation*, *document contents* and *document history*. The above expressions, Items 782–785, do not define anything, they can be said to be "snapshots" of a "behaviour state". Initially there are no document behaviours, docu_i(dd)(da,dc,dh), Item 785. Document behaviours are "started" by the archive behaviour (on behalf of the management and the handler behaviours). Other than mentioning the system (core) behaviour we shall not model that behaviour further.

D.7 Channels, A First View

Channels are means for behaviours to synchronise and communicate values (such as unique identifiers, mereologies and attributes).

787 The management behaviour, mgtm, need to (synchronise and) communicate with the archive behaviour, arch, in order, for the management behaviour, to request the archive behaviour
- to **create** (ab initio or due to **copy**ing)
- or **shred** document behaviours, docu_j,

and for the archive behaviour
- to inform the management behaviour of the identity of the document(behaviour)s that it has created.

channel
787 mgtm_arch_ch:MA

788 The management behaviour, mgtm, need to (synchronise and) communicate with all handler behaviours, hdlr_i and they, in turn, to (synchronised) communicate with the handler management behaviour, mgtm. The management behaviour need to do so in order

- to inform a handler behaviour that it is granted access rights to a specific document, subsequently these access rights may be modified, including revoked.

channel
788 {mgtm_hdlr_ch[i]:MH|i:HI•i ∈ his}

789 The document archive behaviour, arch, need (synchronise and) communicate with all document behaviours, docu_j and they, in turn, to (synchronise and) communicate with the archive behaviour, arch.

channel
789 {arch_docu_ch[j]:AD|h:DI•j ∈ dis}

790 Handler behaviours, hdlr_i, need (synchronise and) communicate with all the document behaviours, docu_j, with which it has operational allowance to so do so[10], and document behaviours, docu_j, need (synchronise and) communicate with potentially all handler behaviours, hdlr_i, namely those handler behaviours, hdlr_i with which they have ("earlier" synchronised and) communicated.

channel
790 {hdlr_docu_ch[i,j]:HD|i:HI,j:DI•i ∈ his∧j ∈ dis}

791 At present we leave undefined the type of messages that are communicated.

type
791 MA, MH, AD, HD

D.8 An Informal Graphical System Rendition

Figure D.1 is an informal rendition of the "state" of a number of behaviours: a single management behaviour, a single archive behaviour, a fixed number, n_h, of one or more handler behaviours, and a variable, initially zero number of document behaviours, with a maximum of these being n_d. The figure also indicates, again rather informally, the channels between these behaviours: one channel between the management and the archive behaviours; n_h channels (n_h is, again, informally indicated) between the management behaviour and the n_h handler behaviours; n_d channels (n_d is, again, informally indicated) between the archive behaviour and the n_d document behaviours; and $n_h \times n_d$ channels ($n_d \times n_d$ is, again, informally indicated) between the n_h handler behaviours and the n_d document behaviours

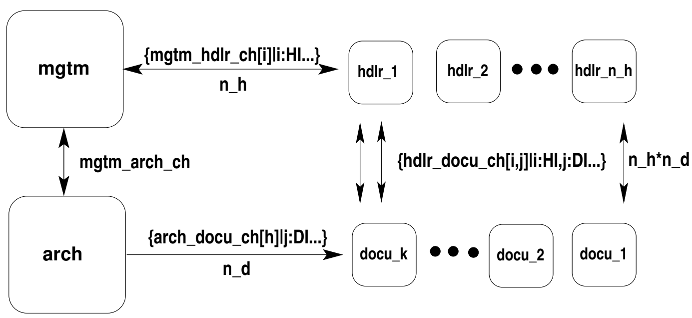


Fig. D.1. An Informal Snapshot of System Behaviours

[10] The notion of operational allowance will be explained below.

D.9 Behaviour Signatures

792 The mgtm behaviour (synchronises and) communicates with the archive behaviour and with all of the handler behaviours, hdlr_i.
793 The archive behaviour (synchronises and) communicates with the mgtm behaviour and with all of the document behaviours, docu_j.
794 The signature of the generic handler behaviours, hdlr_i expresses that they [occasionally] receive "orders" from management, and otherwise [regularly] interacts with document behaviours.
795 The signature of the generic document behaviours, docu_j expresses that they [occasionally] receive "orders" from the archive behaviour and that they [regularly] interacts with handler behaviours.

value
792 mgtm: ... → **in**,**out** mgtm_arch_ch, {mgtm_hdlr_ch[i]|i:HI•i ∈ his} **Unit**
793 arch: ... → **in**,**out** mgtm_arch_ch, {arch_docu_ch[j]|j:DI•j ∈ dis} **Unit**
794 hdlr_i: ... → **in** mgtm_hdlr_ch[i], **in**,**out** {hdlr_docu_ch[i,j]|j:DI•j∈dis} **Unit**
795 docu_j: ... → **in** mgtm_arch_ch, **in**,**out** {hdlr_docu_ch[i,j]|i:HI•i ∈ his} **Unit**

D.10 Time

D.10.1 Time and Time Intervals: Types and Functions

796 We postulate a notion of time, one that covers both a calendar date (from before Christ up till now and beyond). But we do not specify any concrete type (i.e., format such as: YY:MM:DD, HH:MM:SS).
797 And we postulate a notion of (signed) time interval — between two times (say: ±YY:MM:DD:HH:MM:SS).
798 Then we postulate some operations on time: Adding a time interval to a time obtaining a time; subtracting one time from another time obtaining a time interval, multiplying a time interval with a natural number; etc.
799 And we postulate some relations between times and between time intervals.

type
796 TIME
797 TIME_INTERVAL
value
798 add: TIME_INTERVAL × TIME → TIME
798 sub: TIME × TIME → TIME_INTERVAL
798 mpy: TIM_INTERVALE × **Nat** → TIME_INTERVAL
799 <,≤,=,≠,≥,>: ((TIME×TIME)|(TIME_INTERVAL×TIME_INTERVAL)) → **Bool**

D.10.2 A Time Behaviour and a Time Channel

800 We postulate a[n "ongoing"] time behaviour: it either keeps being a time behaviour with unchanged time, t, or – internally non-deterministically – chooses being a time behaviour with a time interval incremented time, t+ti, or – internally non-deterministically – chooses to [first] offer its time on a [global] channel, time_ch, then resumes being a time behaviour with unchanged time., t
801 The time interval increment, ti, is likewise internally non-deterministically chosen. We would assume that the increment is "infinitesimally small", but there is no need to specify so.
802 We also postulate a channel, time_ch, on which the time behaviour offers time values to whoever so requests.

value
800 time: TIME → time_ch TIME **Unit**
800 time(t) ≡ (time(t) ⌈⌉ time(t+ti) ⌈⌉ time_ch!t ; time(t))

801 ti:TIME_INTERVAL ...
channel
802 time_ch:TIME

D.10.3 An Informal RSL Construct

The formal-looking specifications of this report appear in the style of the RAISE [64] Specification Language, RSL [22]. We shall be making use of an informal language construct:

- **wait** ti.

wait is a keyword; ti designates a time interval. A typical use of the wait construct is:

- ... *ptA* ; **wait** ti; *ptB* ; ...

If at specification text point *ptA* we may assert that time is t, then at specification text point *ptB* we can assert that time is t+ti.

D.11 Behaviour "States"

We recall that the endurant parts, Management, Archive, Handlers, and Documents, have properties in the form of *unique identifiers*, *mereologies* and *attributes*. We shall not, in this research note, deal with possible mereologies of these endurants. In this section we shall discuss the endurant attributes of mgtm (management), arch (archive), hdlrs (handlers), and docus (documents). Together the values of these properties, notably the attributes, constitute states – and, since we associate behaviours with these endurants, we can refer to these states also a behaviour states. Some attributes are static, i.e., their value never changes. Other attributes are dynamic.[11] Document handling systems are rather conceptual, i.e., abstract in nature. The dynamic attributes, therefore, in this modeling "exercise", are constrained to just the *programmable* attributes. Programmable attributes are those whose value is set by "their" behaviour. For a behaviour β we shall show the static attributes as one set of parameters and the programmable attributes as another set of parameters.

value β: Static → Program → ... **Unit**

803 For the management endurant/behaviour we focus on one programmable attribute. The management behaviour needs keep track of all the handlers it is charged with, and for each of these which zero, one or more documents they have been granted access to (cf. Sect. D.12.3 on Page 296). Initially that management directory lists a number of handlers, by their identifiers, but with no granted documents.
804 For the archive behaviour we similarly focus on one programmable attribute. The archive behaviour needs keep track of all the documents it has used (i.e., created), those that are avaliable (and not yet used), and of those it has shredded. Initially all these three archive directory sets are empty.
805 For the handler behaviour we similarly focus on one programmable attribute. The handler behaviour needs keep track of all the documents it has been charged with and its access rights to these.
806 Document attributes we mentioned above, cf. Items 769–772.

type
803 MDIR = HI $\overrightarrow{m}$ (DI $\overrightarrow{m}$ ANm-**set**)
804 ADIR = avail:DI-**set** × used:DI-**set** × gone:DI-**set**
805 HDIR = DI $\overrightarrow{m}$ ANm-**set**
806 SDATR = DD, PDATR = DA × DC × DH
axiom
804 ∀ (avail,used,gone):ADIR • avail ∩ used = {} ∧ gone ⊆ used

[11] We refer to Sect. 3.4 of [2], and in particular its subsection 3.4.4.

We can now "complete" the behaviour signatures. We omit, for now, static attributes.

value
792 mgtm: MDIR → **in**,**out** mgtm_arch_ch, {mgtm_hdlr_ch[i]|i:HI•i ∈ his} **Unit**
793 arch: ADIR → **in**,**out** mgtm_arch_ch, {arch_docu_ch[j]|j:DI•j ∈ dis} **Unit**
794 hdlr_i: HDIR → **in** mgtm_hdlr_ch[i], **in**,**out** {hdlr_docu_ch[i,j]|j:DI•j∈dis} **Unit**
795 docu_j: SDATR → PDATR → **in** mgtm_arch_ch, **in**,**out** {hdlr_docu_ch[i,j]|i:HI•i ∈ his} **Unit**

D.12 Inter-Behaviour Messages

Documents are not "fixed, innate" entities. They embody a "history", they have a "past". Somehow or other they "carry a trace of all the "things" that have happened/occurred to them. And, to us, these things are the manipulations that management, via the archive and handlers perform on documents.

D.12.1 Management Messages with Respect to the Archive

807 Management **create** documents. It does so by requesting the archive behaviour to allocate a document identifier and initialize the document "state" and start a document behaviour, with initial information, cf. Item 773 on Page 290:
  a the identity of the initial handler of the document to be created,
  b the time at wich the request is being made,
  c a document descriptor which embodies a (finite) set of zero or more (used) document identifiers (dis),
  d a document annotation note dn, and
  e an initial, i.e., "empty" contents, `"empty_DC"`.

type
773. Crea :: (HI × TIME) × (DI-**set** × Info) × DN × {|″`empty_DC`″|} [cf. formula Item 773, Page 290]

808 The management behaviour passes on to the archive behaviour, requests that it accepts from handlers behaviours, for the copying of document:

808 Copy :: DI × HI × TIME × DN [cf. Item 818 on Page 297]

809 Management **schred**s documents by informing the archive behaviour to do so.

type
809 Shred :: TIME × DI

D.12.2 Management Messages with Respect to Handlers

810 Upon receiving, from the archive behaviour, the "feedback" the identifier of the created document (behaviour):

type
810. Create_Reply :: NewDocID(di:DI)

811 the management behaviour decides to **grant** access rights, acrs:ACRS[12], to a document handler, hi:HI.

type
811 Gran :: HI × TIME × DI × ACRS

[12] For the concept of access rights see Sect. D.12.3 on the next page.

D.12.3 Document Access Rights

Implicit in the above is a notion of document access rights.

812 By document access rights we mean a set of action names.
813 By an action name we mean such tokens that indicate either of the document handler operations indicate above.

type
812 ACRS = ANm-**set**
813 ANm = {|″edit″,″read″,″copy″|}

D.12.4 Archive Messages with Respect to Management

To create a document management provides the archive with some initial information. The archive behaviour selects a document identifier that has not been used before.

814 The archive behaviour informs the management behaviour of the identifier of the created document.

type
814 NewDocID :: DI

D.12.5 Archive Message with Respect to Documents

815 To shred a document the archive behaviour must access the designated document in order to **stop** it. No "message", other than a symbolic "stop", need be communicated to the document behaviour.

type
815 Shred :: {|″stop″|}

D.12.6 Handler Messages with Respect to Documents

Handlers, generically referred to by hdlr_i, may perform the following operations on documents: **edit, read** and **copy**. (Management, via the archive behaviour, **create**s and **shred**s documents.)

816 To perform an **edit** action handler hdlr_i must provide the following:
- the document identity – in the form of a (i:HI,j:DI) channel hdlr_docu_ch index value,
- the handler identity, *i*,
- the time of the edit request,
- and a pair of functions: one which performs the editing and one which un-does it !

type
816 Edit :: DI × HI × TIME × (EDIT × UNDO)

817 To perform a **read** action handler hdlr_i must provide the following information:
- the document identity – in the form of a di:DI channel hdlr_docu_ch index value,
- the handler identity and
- the time of the read request.

type
817 Read :: DI × HI × TIME

D.12.7 Handler Messages with Respect to Management

818 To perform a **copy** action, a handler, hdlr_i, must provide the following information to the management behaviour, mgtm:

- the document identity,
- the handler identity – in the form of an hi:HI channel mgtm_hdlr_ch index value,
- the time of the copy request, and
- a document note (to be affixed both the master and the copy documents).

818 Copy :: DI × HI × TIME × DN [cf. Item 808 on Page 295]

How the handler, the management, the archive and the "named other" handlers then enact the copying, etc., will be outlined later.

D.12.8 A Summary of Behaviour Interactions

Figure D.2 summarises the sources, **out**, resp. !, and the targets, **in**, resp. ?, of the messages covered in the previous sections.

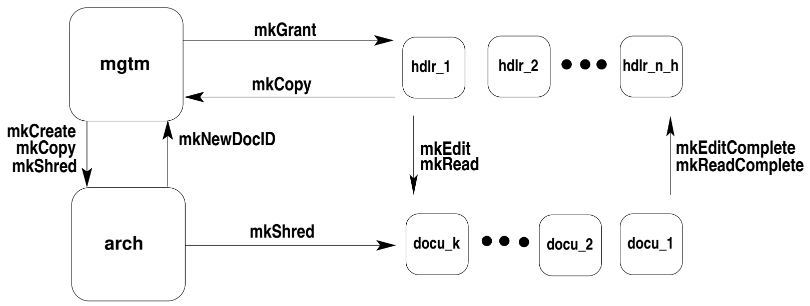


Fig. D.2. A Summary of Behaviour Interactions

D.13 A General Discussion of Handler and Document Interactions

We think of documents being manifest. Either a document is in paper form, or it is in electronic form. In paper form we think of a document as being in only one – and exactly one – physical location. In electronic form a document is also in only one – and exactly one – physical location. No two handlers can access the same document at the same time or in overlapping time intervals. If your conventional thinking makes you think that two or more handlers can, for example, read the same document "at the same time", then, in fact, they are reading either a master and a copy of that master, or they are reading two copies of a common master.

D.14 Channels: A Final View

We can now summarize the types of the various channel messages first referred to in Items 787, 788, 789 and 790.

type

787 MA = Create (Item 807 on Page 295) | Shred (Item 807d on Page 295) | NewDocID (Item 814 on Page 296)
788 MH = Grant (Item 807c on Page 295) | Copy (Item 818 on the preceding page) |
789 AD = Shred (Item 815 on Page 296)
790 HD = Edit (Item 816 on Page 296) | Read (Item 817 on Page 296) | Copy (Item 818 on the preceding page)

D.15 An Informal Summary of Behaviours

D.15.1 The Create Behaviour: Left Fig. D.3

819 [1] The management behaviour, at its own volition, initiates a create document behaviour. It does so by offering a create document message to the archive behaviour.
  a [1.1] That message contains a meaningful document descriptor,
  b [1.2] an initial document annotation,
  c [1.3] an "empty" document contents and
  d [1.4] a single element document history.
(We refer to Sect. D.12.1 on Page 295, Items 807–807e.)
820 [2] The archive behaviour offers to accept that management message. It then selects an available document identifier (here shown as *k*), henceforth marking *k* as used.
821 [3] The archive behaviour then "spawns off" document behaviour docu_k – here shown by the "dash–dotted" rounded edge square.
822 [4] The archive behaviour then offers the document identifier *k* message to the management behaviour. (We refer to Sect. D.12.4 on Page 296, Item 814.)
823 [5] The management behaviour then
  a [5.1] selects a handler, here shown as *i*, i.e., hdlr_i,
  b [5.2] records that that handler is granted certain access rights to document *k*,
  c [5.3] and offers that granting to handler behaviour *i*.
(We refer to Sect. D.12.2 on Page 295, Item 811 on Page 295.)
824 [6] Handler behaviour *i* records that it now has certain access rights to doccument *i*.

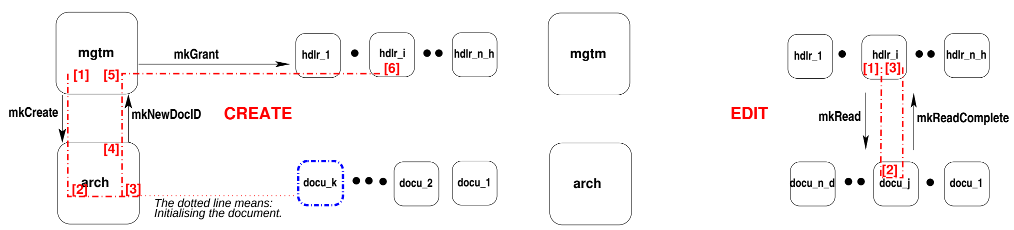


Fig. D.3. Informal Snapshots of Create and Edit Document Behaviours

D.15.2 The Edit Behaviour: Right Fig. D.3

1 Handler behaviour *i*, at its own volition, initiates an edit action on document *j* (where *i* has editing rights for document *j*). Handler *i*, optionally, provides document *j* with a(annotation) note. While editing document *j* handler *i* also "selects" an appropriate pair of *edit/undo* functions for document *j*.
2 Document behaviour *j* accepts the editing request, enacts the editing, optionally appends the (annotation) note, and, with handler *i*, completes the editing, after some time interval ti.
3 Handler behaviour *i* completes its edit action.

D.15.3 The Read Behaviour: Left Fig. D.4

1 Handler behaviour i, at its own volition, initiates a read action on document j (where i has reading rights for document j). Handler i, optionally, provides document j with a(annotation) note.
2 Document behaviour j accepts the reading request, enacts the reading by providing the handler, i, with the document contents, and optionally appends the (annotation) note, and, with handler i, completes the reading, after some time interval ti.
3 Handler behaviour i completes its read action.

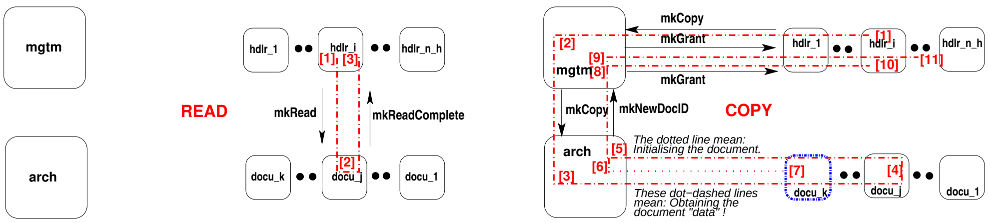


Fig. D.4. Informal Snapshots of Read and Copy Document Behaviours

D.15.4 The Copy Behaviour: Right Fig. D.4

1 Handler behaviour i, at its own volition, initiates a copy action on document j (where i has copying rights for document j). Handler i, optionally, provides master document j as well as the copied document (yet to be identified) with respective (annotation) notes.
2 The management behaviour offers to accept the handler message. As for the create action, the management behaviour offers a combined *copy and create* document message to the archive behaviour.
3 The archive behaviour selects an available document identifier (here shown as k), henceforth marking k as used.
4 The archive behaviour then obtains, from the master document j its *d*ocument *d*escriptor, dd_j, its *d*ocument *a*nnotations, da_j, its *d*ocument *c*ontents, dc_j, and its document history, dh_j.
5 The archice behaviour informs the management behaviour of the identifier, k, of the (new) document copy,
6 while assembling the attributes for that (new) document copy: its *d*ocument *d*escriptor, dd_k, its *d*ocument *a*nnotations, da_k, its *d*ocument *c*ontents, dc_k, and its document history, dh_k, from these "similar" attributes of the master document j,
7 while then "spawning off" document behaviour docu_k – here shown by the "dash–dotted" rounded edge square.
8 The management behaviour accepts the identifier, k, of the (new) document copy, recording the identities of the handlers and their access rights to k,
9 while informing these handlers (informally indicated by a "dangling" dash-dotted line) of their grants,
10 while also informing the master copy of the copy identity (etcetera).
11 The handlers granted access to the copy record this fact.

D.15.5 The Grant Behaviour: Left Fig. D.5 on the next page

This behaviour has its

1 Item [1] correspond, in essence, to Item [9] of the copy behaviour – see just above – and
2 Item [2] correspond, in essence, to Item [11] of the copy behaviour.

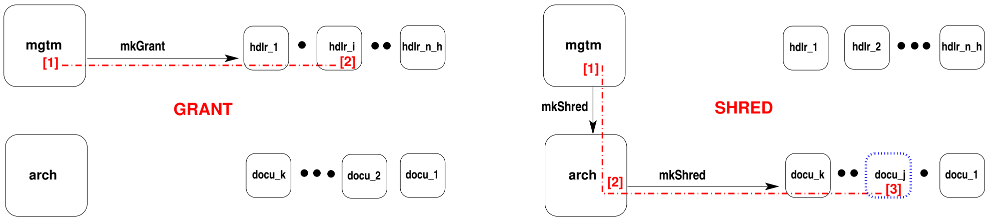


Fig. D.5. Informal Snapshots of Grant and Shred Document Behaviours

D.15.6 The Shred Behaviour: Right Fig. D.5

1 The management, at its own volition, selects a document, j, to be shredded. It so informs the archive behaviour.
2 The archive behaviour records that document j is to be no longer in use, but shredded, and informs document j's behaviour.
3 The document j behaviour accepts the shred message and **stop**s (indicated by the dotted rounded edge box).

D.16 The Behaviour Actions

To properly structure the definitions of the four kinds of (management, archive, handler and document) behaviours we single each of these out "across" the six behaviour traces informally described in Sects. D.15.1–D.15.6. The idea is that if behaviour β is involved in τ traces, $\tau_1, \tau_2, ..., \tau_\tau$, then behaviour β shall be defined in terms of τ non-deterministic alternative behaviours named $\beta_{\tau_1}, \beta_{\tau_2}, ..., \beta_{\tau_\tau}$.

D.16.1 Management Behaviour

825 The management behaviour is involved in the following action traces:
- a **create** — Fig. D.3 on Page 298 Left
- b **copy** — Fig. D.4 on the preceding page Right
- c **grant** — Fig. D.5 Left
- d **shred** — Fig. D.5 Right

value
825 mgtm: MDIR → **in**,**out** mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} **Unit**
825 mgtm(mdir) ≡
825a mgtm_create(mdir)
825b ⌈⌉ mgtm_copy(mdir)
825c ⌈⌉ mgtm_grant(mdir)
825d ⌈⌉ mgtm_shred(mdir)

Management Create Behaviour: Left Fig. D.3 on Page 298

826 The **management create** behaviour
827 initiates a create document behaviour (i.e., a request to the archive behaviour),
828 and then awaits its response.

value

```
826  mgtm_create: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
826  mgtm_create(mdir) ≡
827  [1]  let hi = mgtm_create_initiation(mdir) ;            [Left Fig. D.3 on Page 298]
828  [5]  mgtm_create_awaits_response(mdir)(hi) end  [Left Fig. D.3 on Page 298]
```

The **management create initiation** behaviour

829 selects a handler on behalf of which it requests the document creation,
830 assembles the elements of the create message:
- by embedding a set of zero or more document references, dis, with some information, info, into a document descriptor, adding
- a document note, dn, and
- and initial, that is, empty document contents, "empty_DC",

831 offers such a create document message to the archive behaviour, and
832 yields the identifier of the chosen handler.

value

```
827  mgtm_create_initiation: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} HI
827  mgtm_create_initiation(mdir) ≡
829            let hi:HI • hi ∈ dom mdir,
830  [1.2−.4]      (dis,info):(DI-set×Info),dn:DN • is_meaningful(embed_DIs_in_DD(dis,info))(mdir) in
831  [1.1]      mgtm_arch_ch ! mkCreate(embed_DIs_in_DD(ds,info),dn,″empty_DC″)
832            hi end

830  is_meaningful: DD → MDIR → Bool [left further undefined]
```

The **management create awaits response** behaviour

833 starts by awaiting a reply from the archive behaviour with the identity, *di*, of the document (that that behaviour has created).
834 It then selects suitable access rights,
835 with which it updates its handler/document directory
836 and offers to the chosen handler
837 whereupon it resumes, with the updated management directory, being the management behaviour.

value

```
828  mgtm_create_awaits_response: MDIR → HI → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
828  mgtm_create_awaits_response(mdir) ≡
833  [5]        let mkNewDocID(di) = mgtm_arch_ch ? in
834  [5.1]      let acrs:ANm-set in
835  [5.2]      let mdir′ = mdir † [hi ↦ [di ↦ acrs]] in
836  [5.3]      mgtm_hdlr_ch[hi] ! mkGrant(di,acrs)
837             mgtm(mdir′) end end end
```

Management Copy Behaviour: Right Fig. D.4 on Page 299

838 The **management copy** behaviour
839 accepts a copy document request from a handler behaviour (i.e., a request to the archive behaviour),
840 and then awaits a response from the archive behaviour;
841 after which it grants access rights to handlers to the document copy.

value

```
838  mgtm_copy: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
838  mgtm_copy(mdir) ≡
```

```
839  [2]  let hi = mgtm_accept_copy_request(mdir) in
840  [8]  let di = mgtm_awaits_copy_response(mdir)(hi) in
841  [9]  mgtm_grant_access_rights(mdir)(di) end end
```

842 The **management accept copy** behaviour non-deterministically externally (⌈⌉) awaits a copy request from a[ny] handler (*i*) behaviour –
843 with the request identifying the master document, *j*, to be copied.
844 The management accept copy behaviour forwards (!) this request to the archive behaviour –
845 while yielding the identity of the requesting handler.

```
842. mgtm_accept_copy_request: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} HI
842. mgtm_accept_copy_request(mdir) ≡
843.      let mkCopy(di,hi,t,dn) = ⌈⌉{mgtm_hdlr_ch[i]?|i:HI•i ∈ his} in
844.      mgtm_arch_ch ! mkCopy(di,hi,t,dn) ;
844.      hi end
```

The **management awaits copy response** behaviour

846 awaits a reply from the archive behaviour as to the identity of the newly created copy (*di*) of master document *j*.
847 The management awaits copy response behaviour then informs the 'copying-requesting' handler, *hi*, that the copying has been completed and the identity of the copy (*di*) –
848 while yielding the identity, *di*, of the newly created copy.

```
825b. mgtm_awaits_copy_response: MDIR → HI → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} DI
825b. mgtm_awaits_copy_response(mdir)(hi) ≡
846.  [8]  let mkNewDocID(di) = mgtm_arch_ch ? in
847.       mgtm_hdlr_ch[hi] ! mkCopy(di) ;
848.       di end
```

The **management grants access rights** behaviour

849 selects suitable access rights for a suitable number of selected handlers.
850 It then offers these to the selected handlers.

```
841. mgtm_grant_access_rights: MDIR → DI → in,out {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
841. mgtm_grant_access_rights(mdir)(di) ≡
849.      let diarm = [hi↦acrs|hi:HI,acrs:ANm-set• hi ∈ dom mdir∧acrs⊆(diarm(hi))(di)] in
850.      ‖ {mgtm_hdlr_ch[hi]!mkGrant(hi,time_ch?,di,acrs) |
850.         hi:HI,acrs:ANm-set•hi ∈ dom diarm∧acrs⊆(diarm(hi))(di)} end
```

Management Grant Behaviour: Left Fig. D.5 on Page 300

The **management grant** behaviour

851 is a variant of the mgtm_grant_access_rights function, Items 849–850.
852 The management behaviour selects a suitable subset of known handler identifiers, and
853 for these a suitable subset of document identifiers from which
854 it then constructs a map from handler identifiers to subsets of access rights.
855 With this the management behaviour then issues appropriate grants to the chosen handlers.

```
type
    MDIR = HI →m (DI →m ANm-set)
value
851  mgtm_grant: MDIR → in,out {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
```

```
851  mgtm_grant(mdir) ≡
852      let his ⊆ dom dir in
853      let dis ⊆ ∪{dom mdir(hi)|hi:HI•hi ∈ his} in
854      let diarm = [hi↦acrs|hi:HI,di:DI,acrs:ANm-set• hi ∈ his∧di ∈ dis∧acrs⊆(diarm(hi))(di)] in
855      ∥{mgtm_hdlr_ch[hi]!mkGrant(di,acrs) |
855         hi:HI,di:DI,acrs:ANm-set•hi ∈ dom diarm∧di ∈ dis∧acrs⊆(diarm(hi))(di)}
851      end end end
```

Management Shred Behaviour: Right Fig. D.5 on Page 300

The **management shred** behaviour

856 initiates a request to the archive behaviour.
857 First the management shred behaviour selects a document identifier (from its directory).
858 Then it communicates a shred document message to the archive behaviour;
859 then it notes the (to be shredded) document in its directory
860 whereupon the management shred behaviour resumes being the management behaviour.

```
value
856  mgtm_shred: MDIR → out mgtm_arch_ch  Unit
856  mgtm_shred(mdir) ≡
857      let di:DI • is_suitable(di)(mdir) in
858  [1] mgtm_arch_ch ! mkShred(time_ch?,di) ;
859      let mdir′ = [hi↦mdir(hi)\{di}|hi:HI•hi ∈ dom mdir] in
860      mgtm(mdir′) end end
```

D.16.2 Archive Behaviour

861 The archive behaviour is involved in the following action traces:
- a **create** — Fig. D.3 on Page 298 Left
- b **copy** — Fig. D.4 on Page 299 Right
- c **shred** — Fig. D.5 on Page 300 Right

```
type
804  ADIR = avail:DI-set × used:DI-set × gone:DI-set
axiom
804  ∀ (avail,used,gone):ADIR • avail ∩ used = {} ∧ gone ⊆ used
value
861  arch: ADIR → in,out mgmt_arch_ch, {arch_docu_ch[di]|di:DI•di ∈ dis} Unit
861a arch(adir) ≡
861a      arch_create(adir)
861b    ⌈⌉ arch_copy(adir)
861c    ⌈⌉ arch_shred(adir)
```

The Archive Create Behaviour: Left Fig. D.3 on Page 298

The **archive create** behaviour

862 accepts a request, from the management behaviour to create a document;
863 it then selects an available document identifier;
864 communicates this new document identifier to the management behaviour;

865 while initiating a new document behaviour, docu_{di}, with the document descriptor, *dd*, the initial document annotation being the singleton list of the note, *an*, and the initial document contents, *dc* – all received from the management behaviour – and an initial document history of just one entry: the date of creation, all
866 in parallel with resuming the archive behaviour with updated programmable attributes.

```
861a.  arch_create: AATTR → in,out mgmt_arch_ch, {arch_docu_ch[di]|di:DI•di ∈ dis} Unit
861a.  arch_create(avail,used,gone) ≡
862.   [2]  let mkCreate((hi,t),dd,an,dc) = mgmt_arch_ch ? in
863.        let di:DI•di ∈ avail in
864.   [4]  mgmt_arch_ch ! mkNewDocID(di) ;
865.   [3]  docu_di(dd)(⟨an⟩,dc,<(date_of_creation)>)
866.        ‖ arch(avail\{di},used∪{di},gone)
861a.       end end
```

The Archive Copy Behaviour: Right Fig. D.4 on Page 299

The **archive copy** behaviour

867 accepts a copy document request from the management behaviour with the identity, *j*, of the master document;
868 it communicates (the request to obtain all the attribute values of the master document, *j*) to that document behaviour;
869 whereupon it awaits their communication (i.e., (*dd*,*da*,*dc*,*dh*));
870 (meanwhile) it obtains an available document identifier,
871 which it communicates to the management behaviour,
872 while initiating a new document behaviour, docu_{di}, with the master document descriptor, *dd*, the master document annotation, and the master document contents, *dc*, and the master document history, *dh* (all received from the master document),
873 in parallel with resuming the archive behaviour with updated programmable attributes.

```
861b.  arch_copy: AATTR → in,out mgmt_arch_ch, {arch_docu_ch[di]|di:DI•di ∈ dis} Unit
861b.  arch_copy(avail,used,gone) ≡
867.   [3]  let mkDocID(j,hi) = mgtm_arch_ch ? in
868.        arch_docu_ch[j] ! mkReqAttrs() ;
869.        let mkAttrs(dd,da,dc,dh) = arch_docu_ch[j] ? in
870.        let di:DI • di ∈ avail in
871.        mgtm_arch_ch ! mkCopyDocID(di) ;
872.  [6,7] docu_di(augment(dd,"copy",j,hi),augment(da,"copy",hi),dc,augment(dh,("copy",date_and_time,j,hi)))
873.      ‖ arch(avail\{di},used∪{di},gone)
861b.     end end end
```

where we presently leave the [overloaded] augment functions undefined.

The Archive Shred Behaviour: Right Fig. D.5 on Page 300

The **archive shred** behaviour

874 accepts a shred request from the management behaviour.
875 It communicates this request to the identified document behaviour.
876 And then resumes being the archive behaviour, noting however, that the shredded document has been shredded.

```
861c.  arch_shred: AATTR → in,out mgmt_arch_ch, {arch_docu_ch[di]|di:DI•di ∈ dis} Unit
861c.  arch_shred(avail,used,gone) ≡
874.   [2] let mkShred(j) = mgmt_arch_ch ? in
875.       arch_docu_ch[j] ! mkShred() ;
876.       arch(avail,used,gone∪{j})
861c.      end
```

D.16.3 Handler Behaviours

877 The handler behaviour is involved in the following action traces:

- a **create** — Fig. D.3 on Page 298 Left
- b **edit** — Fig. D.3 on Page 298 Right
- c **read** — Fig. D.4 on Page 299 Left
- d **copy** — Fig. D.4 on Page 299 Right
- e **grant** — Fig. D.5 on Page 300 Left

value

```
877   hdlr_hi: HATTRS → in,out mgtm_hdlr_ch[hi],{hdlr_docu_ch[hi,di]|di:DI•di∈dis} Unit
877   hdlr_hi(hattrs) ≡
877a        hdlr_create_hi(hattrs)
877b      ⌈⌉ hdlr_edit_hi(hattrs)
877c      ⌈⌉ hdlr_read_hi(hattrs)
877d      ⌈⌉ hdlr_copy_hi(hattrs)
877e      ⌈⌉ hdlr_grant_hi(hattrs)
```

The Handler Create Behaviour: Left Fig. D.3 on Page 298

878 The **handler create** behaviour offers to accept the granting of access rights, acrs, to document di.
879 It according updates its programmable hattrs attribute;
880 and resumes being a handler behaviour with that update.

```
877a  hdlr_create_hi: HATTRS × HHIST → in,out mgtm_hdlr_ch[hi] Unit
877a  hdlr_create_hi(hattrs,hhist) ≡
878     let mkGrant(di,acrs) = mgtm_hdlr_ch[hi] ? in
879     let hattrs′ = hattrs † [hi ↦ acrs] in
880     hdlr_create_hi(hattrs′,augment(hhist,mkGrant(di,acrs))) end end
```

The Handler Edit Behaviour: Right Fig. D.3 on Page 298

881 The handler behaviour, on its own volition, decides to edit a document, *di*, for which it has editing rights.
882 The handler behaviour selects a suitable (...) pair of *e*dit/*u*ndo functions and a suitable (annotation) note.
883 It then communicates the desire to edit document *di* with (*e*,*u*) (at time *t*=time_ch?).
884 Editing take some time, *ti*.
885 We can therefore assert that the time at which editing has completed is *t*+*ti*.
886 The handler behaviour accepts the edit completion message from the document handler.
887 The handler behaviour can therefore resume with an updated document history.

```
877b  hdlr_edit_hi: HATTRS × HHIST → in,out {hdlr_docu_ch[hi,di]|di:DI•di∈dis} Unit
877b  hdlr_edit_hi(hattrs,hhist) ≡
881 [1]  let di:DI • di ∈ dom hattrs ∧ "edit" ∈ hattrs(di) in
882 [1]  let (e,u):(EDIT×UNDO) • ... , n:AN • ... in
883 [1]  hdlr_docu_ch[hi,di] ! mkEdit(hi,t=time_ch?,e,u,n) ;
884 [2]  let ti:TIME_INTERVAL • ... in
885 [2]  wait ti ; assert: time_ch? = t+ti
886 [3]  let mkEditComplete(ti′,...) = hdlr_docu_ch[hi,di] ? in assert ti′ ≅ ti
887      hdlr_hi(hattrs,augment(hhist,(di,mkEdit(hi,t,ti,e,u))))
877b     end end end end
```

The Handler Read Behaviour: Left Fig. D.4 on Page 299

888 The **handler behaviour**, on its own volition, decides to read a document, *di*, for which it has reading rights.
889 It then communicates the desire to read document *di* with at time *t*=time_ch? – with an annotation note (*n*).
890 Reading take some time, *ti*.
891 We can therefore assert that the time at which reading has completed is *t+ti*.
892 The handler behaviour accepts the read completion message from the document handler.
893 The handler behaviour can therefore resume with an updated document history.

```
877c  hdlr_edit_hi: HATTRS × HHIST → in,out {hdlr_docu_ch[hi,di]|di:DI•di∈dis} Unit
877c  hdlr_edit_hi(hattrs,hhist) ≡
888 [1]  let di:DI • di ∈ dom hattrs ∧ "read" ∈ hattrs(di), n:N • ... in
889 [1]  hdlr_docu_ch[hi,di] ! mkRead(hi,t=time_ch?,n) ;
890 [2]  let ti:TIME_INTERVAL • ... in
891 [2]  wait ti ; assert: time_ch? = t+ti
892 [3]  let mkReadComplete(ti,...) = hdlr_docu_ch[hi,di] ? in
893      hdlr_hi(hattrs,augment(hhist,(di,mkRead(di,t,ti))))
877c     end end end
```

The Handler Copy Behaviour: Right Fig. D.4 on Page 299

894 The **handler [copy] behaviour**, on its own volition, decides to copy a document, *di*, for which it has copying rights.
895 It communicates this copy request to the management behaviour.
896 After a while the handler [copy] behaviour receives acknowledgement of a completed copying from the management behaviour.
897 The handler [copy] behaviour records the request and acknowledgement in its, thus updated whereupon the handler [copy] behaviour resumes being the handler behaviour.

```
877d  hdlr_copy_hi: HATTRS × HHIST → in,out mgtm_hdlr_ch[hi] Unit
877d  hdlr_copy_hi(hattrs,hhist) ≡
894   [1] let di:DI • di ∈ dom hattrs ∧ "copy" ∈ hattrs(di) in
895   [1] mgtm_hdlr_ch[hi] ! mkCopy(di,hi,t=time_ch?) ;
896  [10] let mkCopyComplete(di′,di) = mgtm_hdlr_ch[hi] ? in
897  [10] hdlr_hi(hattrs,augment(hhist,time_ch?,(mkCopy(di,hi,,t),mkCopyComplete(di′))))
877d      end end
```

The Handler Grant Behaviour: Left Fig. D.5 on Page 300

898 The **handler [grant] behaviour** offers to accept grant permissions from the management behaviour.
899 In response it updates its handler attribute while resuming being a handler behaviour.

```
877e   hdlr_grant_hi: HATTRS × HHIST → in,out mgtm_hdlr_ch[hi] Unit
877e   hdlr_grant_hi(hattrs,hhist) ≡
898    [2] let mkGrant(di,acrs) = mgtm_hdlr_ch[hi] ? in
899    [2] hdlr_hi(hattrs†[di↦acrs],augment(hhist,time_ch?,mkGrant(di,acrs)))
877e       end
```

D.16.4 Document Behaviours

900 The document behaviour is involved in the following action traces:

- a **edit** — Fig. D.3 on Page 298 Right
- b **read** — Fig. D.4 on Page 299 Left
- c **shred** — Fig. D.5 on Page 300 Right

```
value
900    docu_di: DD × (DA × DC × DH) → in,out arch_docu_ch[di], {hdlr_docu_ch[hi,di]|hi:HI•hi∈his} Unit
900    docu_di(dattrs) ≡
900a        docu_edit_di(dd)(da,dc,dh)
900b      ⌈⌉ docu_read_di(dd)(da,dc,dh)
900c      ⌈⌉ docu_shred_di(dd)(da,dc,dh)
```

The Document Edit Behaviour: Right Fig. D.3 on Page 298

901 The **document [edit] behaviour** offers to accept edit requests from document handlers.
  - a The document contents is edited, over a time interval of ti, with respect to the handlers edit function (e),
  - b the document annotations are augmented with respect to the handlers note (n), and
  - c the document history is augmented with the fact that an edit took place, at a certain time, with a pair of *edit/undo* functions.

902 The *edit* (etc.) function(s) take some time, ti, to do.
903 The handler behaviour is notified, mkEditComplete(...) of the completion of the edit, and
904 the document behaviour is then resumed with updated programmable attributes.

```
value
900a   docu_edit_di: DD × (DA × DC × DH) → in,out {hdlr_docu_ch[hi,di]|hi:HI•hi∈his} Unit
900a   docu_edit_di(dd)(da,dc,dh) ≡
901    [2] let mkEdit(hi,t,e,u,n) = ⌈⌉⌊⌋{hdlr_docu_ch[hi,di]?|hi:HI•hi∈his} in
901a   [2] let dc′ = e(dc),
901b           da′ = augment(da,((hi,t),(″edit″,e,u),n)),
901c           dh′ = augment(dh,((hi,t),(″edit″,e,u))) in
902        let ti = time_ch? − t in
903        hdlr_docu_ch[hi,di] ! mkEditComplete(ti,...) ;
904        docu_di(dd)(da′,dc′,dh′)
900a       end end end
```

The Document Read Behaviour: Left Fig. D.4 on Page 299

905 The The **document [read] behaviour** offers to receive a read request from a handler behaviour.
906 The reading takes some time to do.
907 The handler behaviour is advised on completion.
908 And the document behaviour is resumed with appropriate programmable attributes being updated.

```
value
900b  docu_read_di: DD × (DA × DC × DH) → in,out {hdlr_docu_ch[hi,di]|hi:HI•hi∈his} Unit
900b  docu_read_di(dd)(da,dc,dh) ≡
905   [2] let mkRead(hi,t,n) = {hdlr_docu_ch[hi,di]?|hi:HI•hi∈his} in
906   [2] let ti:TIME_INTERVAL • ... in
906   [2] wait ti ;
907   [2] hdlr_docu_ch[hi,di] ! mkReadComplete(ti,...) ;
908   [2] docu_di(dd)(augment(da,n),dc,augment(dh,(hi,t,ti,"read")))
900b        end end
```

The Document Shred Behaviour: Right Fig. D.5 on Page 300

909 The **document [shred] behaviour** offers to accept a document shred request from the archive behaviour –
910 whereupon it **stops** !

```
value
900c  docu_shred_di: DD × (DA × DC × DH) → in,out arch_docu_ch[di] Unit
900c  docu_shred_di(dd)(da,dc,dh) ≡
909   [3] let mkShred(...) = arch_docu_ch[di] ? in
910        stop
900c  [3] end
```

D.16.5 Conclusion

This completes a first draft version of this document. The date time is: June 17, 2018: 10:12 am. Many things need to be done. First a careful checking of all types and functions: that all used names have been defined. The internal non-deterministic choices in formula Items 825 on Page 300, 861 on Page 303, 877 on Page 305 and 900 on the preceding page, need be checked. I suspect there should, instead, be som mix of both internal and external non-deterministic choices. Then a careful motivation for all the other non-deterministic choices.

D.17 Documents in Public Gornment

Public government, in the spirit of *Charles-Louis de Secondat, Baron de La Brède et de Montesquieu* (or just *Montesquieu*), has three branches:

- the **legislative**,
- the **executive**, and
- the **judicial**.

Our interpretation of these, with respect to documents, are as follows.

- The **legislative** branch produces laws, i.e., documents. To do so many preparatory documents are created, edited, read, copied, etc. Committees, subcommittees, individual lawmakers and ministry law office staff handles these documents. Parliament staff and legislators are granted limited or unlimited access rights to these documents. Finally laws are put into effect, are amended, changed or abolished. The legislative branch documents refer to legislative, executive and judicial branch documents.

- The **executive** branch produces guide lines, i.e., documents. Instructions on interpretation and implementation of laws; directives to ministry services on how to handle the laws; etcetera.
  These executive branch documents refer to legislative, executive and judicial branch documents.
- The **judicial** branch produces documents. Police cite citizens and enterprises for breach of law. Citizens and enterprise sue other citizens and/or enterprises. Attorneys on behalf of the governments, or citizens or enterprises prepare statements. Court proceedings are recorded. Justices pass verdicts.
  The judicial branch documents refer to legislative, executive and judicial branch documents.

D.18 Documents in Urban Planning

A separate research note [35, Urban Planning Processes] analyses & describes a domain of urban planning. There are the geographical documents:

- geodetic,
- geotechnic,
- meteorological,
- and other types of geographical documents.

In order to perform an informed urban planning further documents are needed:

- auxiliary documents which
- requirements documents which

Auxiliary documents presents such information that "fill in" details concerning current ownership of the land area, current laws affecting this ownership, the use of the land, etcetera. Requirements documents express expectations about the (base) urban plans that should result from the base urban planning. As a first result of base urban planning we see the emergence of the following kinds of documents:

- base urban plans
- and ancillary notes.

The base urban plans deal with

- cadestral,
- cartograhic and
- zoning

issues. The ancillary notes deal with such things as insufficiencies in the base planss, things that ought be improved in a next iteration base urban plannin, etc. The base plans and ancillerary notes, besides possible re-iteration of base urban planning, lead on to "derived urban planning" for

- light, medium and heavy industry zones,
- mixed shopping and residential zones,
- apartment building zones,
- villa zones,
- recreational zones,
- etcetera.

After these "first generation" derived urban plans are well underway, a "second generation" derived urban planning can start:

- transport infrastructure,
- water and waste resource management,
- electricity, natural gas, etc., infrastructure,
- etcetera.

And so forth. Literally "zillions upon zillions" of strongly and crucially interrelated documents accrue.

Urban planning evolves and revolves around documents.

Documents are the only "tangible" results or urban planning.[13]

[13] Once urban plans have been agreed upon by all relevant authorities and individuals, then urban development ("build") and, finally, "operation" of the developed, new urban "landscape". For development, the urban plans form one of the "tangible" inputs. Others are of financial and human and other resource nature.

E

Urban Planning

We examine concepts of urban planning. There is ***the urban space*** which we treat as a ***part*** and as a ***behaviour***. There are n distinct urban space ***analysers***, distinctly named (i.e., indexed) $\{a_1,a_2,...,a_n\}$, treated as [parts and] ***behaviours***. There is one ***master planner***, treated as a [part and as a] ***behaviour***. There are p distinctly named ***derived [urban] planners***, distinctly named (i.e., indexed) $\{d_1,d_2,...,d_p\}$ and treated as [parts and] ***behaviours***. .

To serve the one master and the p derived planners there are 1+p distinctly named ***input argument servers*** $\{m,d_1,d_2,...,d_p\}$, one ***output result server***, and one ***derived planner index generator***. All of these are also treated as ***parts*** and ***behaviours***. .

The behaviours (synchronise and) communicate via ***channels***. An array of channels communicate ***urban space attribute values*** to requesting analysers. The analysers provide ***analyses*** to all planners. The planners obtain ***input arguments*** from "their" servers. The planners provide ***result values*** to the common output result server. And the derived planner index generator provide possibly empty sets of ***derived planner indices*** to all planners. .

Emphasis, in this research note, is on the ***information*** (abstract "data") and ***functions*** and ***behaviours*** of urban space analysis and planning – and their ***interaction***. We separate ***urban space analysis*** from ***urban planning***. ***Urban space analysers*** analyse [existing] urban spaces and produce ***analyses***. ***Urban planners*** analyse the analysis results and, in case of the master planning, also the urban space [itself] – and produce plans and other information. The ***master [urban] planner*** produces a master plan [and other information]. The ***derived [urban] planners*** produce derivative [urban] plans [and other information]. That is, we thus distinguish between the two kinds of urban planning: the ***master***, 'ab initio', behaviour of determining "the general layout of the land (!)", and the ***derived***, 'follow-up', behaviours focused on social and technological infrastructures. Master urban planning applies to descriptions of "the land": ***geographic, that is, geodetic, cadastral, geotechnical, meteorological, socio-economic and rules & regulations.*** Examples of derived urban plannings are such which are focused on humans and on social and technological artifacts: ***industry zones, commercial (i.e., office and shopping) zones, residential zones, recreational areas***, etc., and ***health care, schools, transport, electricity, water, waste***, etc. This research note also discusses issues of urban planning project management, cf. Sect. E.16.4, and urban planning document management, cf. Sect. E.16.2. The overall aim of this paper is to suggest a formal foundation for urban planning. We must emphasize that all that is conceivable and describable in the domain can be described. We shall return to this remark, in this report, again-and-again.

E.1 Introduction

"Urban planning is a technical and political process concerned with the development and use of land, planning permission, protection and use of the environment, public welfare, and the design of the urban environment, including air, water, and the infrastructure passing into and out of urban areas, such as transportation, communications, and distribution networks."[1]

[1] https://en.wikipedia.org/wiki/Urban_planning

In this research note we shall try to understand two of the aspects of the domain underlying urban planning, (i) namely those of the "input" information to and "output" plans (etc.) from urban planning, and (ii) that of some possible urban planning (development) functions and processes. We are trying to understand and describe a domain, not requirements for IT for that domain and certainly not the IT (incl. its software). And: We are certainly not constructing any general or any specific urban plan !

The overall aim of this case study is to suggest a formal foundation for urban planning.

Another, secondary aim of this case study is to suggest that a number of requirements must be satisfied before a fully professional urban development project can be commenced.

E.1.1 On Urban Planning

We search for answers to the question: *"What is Urban Planning ?"*. First we identify "planning areas". Then we sketch element of a first domain model for Urban Planning.

Urban planning seems to be also be about ***infrastructure planning.*** So we examine these terms. First the latter, then the former.

Infrastructures

The term 'infrastructure' has gained currency in the last 80 years.[2]. It is more frequently used in socio--economic than in scientific, let alone computing science, contexts. According to the World Bank, 'infrastructure' is an umbrella term for many activities referred to as 'social overhead capital' by some development economists, and encompasses activities that share technical and economic features (such as economies of scale and spill-overs from users to non-users). We take a more technical view, and see infrastructures as concerned with supporting other systems or activities. Software for infrastructures is likely to be distributed and concerned in particular with supporting communication of data, people and/or materials. Hence issues of openness, timeliness, security, lack of corruption and resilience are often important.

Examples of infrastructures, or, more precisely, infrastructure components, are:

- transport systems (roads, railways, air traffic, canals/rivers/lake/ocean , etc.);
- water and sewage;
- telecommunications;
- postal service (physical letters, packages etc.);
- power: electricity, gas, oil, wind (generation, distribution); etc.
- the financial industry (banking, insurance, securities, clearing, etc.);
- documents (creation, editing, formatting, etc.);
- ministry of finance (taxation, budget, treasury, etc.);
- health care (private physicians, clinics, hospitals, pharmacies, etc.);
- education (kindergartens, pre-schools, primary schools, secondary schools, high schools, colleges, universities);
- manufacturing industry;
- et cetera.

Wikipedia: https://en.wikipedia.org/wiki/Urban_planning

"Urban planning is a technical and political process concerned with the development and use of land ***planning permission, protection and use of the environment,*** public well-fare***, and the design of the urban environment, including*** air, water, and the infrastructure ***passing into and out of urban areas, such as*** transportation, communications, and distribution networks ***[912]."***

"Urban planning is also referred to as urban and regional planning, regional planning, town planning, city planning, rural planning or some combination in various areas worldwide. It takes many forms and it can share perspectives and practices with urban design [911]."

"Urban planning guides orderly development in urban, suburban and rural areas. Although predominantly concerned with the planning of settlements and communities, urban planning is also

[2] Winston Churchill is quoted to have said, in the House of Commons, in 1936: *... the young Labourite speaker, that we just heard, obviously wishes to impress his constituency with the fact that he has attended Eton and Oxford when he uses such modern terms as 'infrastructure' ...*

responsible for the** planning and development of water use and resources, rural and agricultural land, parks and conserving areas of natural environmental **significance. Practitioners of urban planning are concerned with research and analysis, strategic thinking, architecture, urban design, public consultation, policy recommendations, implementation and management [913].”

“Urban planners work with the cognate fields of architecture, landscape architecture, civil engineering, and public administration to achieve strategic, policy and sustainability goals. Early urban planners were often members of these cognate fields. Today urban planning is a separate, independent professional discipline. The discipline is the broader category that includes different sub-fields such as land-use planning, zoning, economic development, environmental planning, and transportation planning [914].”

Theories of Urban Planning

“Planning theory is the body of scientific concepts, definitions, behavioral relationships, and assumptions that define the body of knowledge of urban planning. There are eight procedural theories of planning that remain the principal theories of planning procedure today: the rational-comprehensive approach, the incremental approach, the trans-active approach, the communicative approach, the advocacy approach, the equity approach, the radical approach, and the humanist or phenomenological approach [915].”

Technical aspects

Technical aspects of urban planning involve applying scientific, technical processes, considerations and features that are involved in planning for land use, urban design, natural resources, transportation, and infrastructure. Urban planning includes techniques such as: predicting population growth, zoning, geographic mapping and analysis, analyzing park space, surveying the water supply, identifying transportation patterns, recognizing food supply demands, allocating health-care and social services, and analyzing the impact of land use.

Urban planners

An urban planner is a professional who works in the field of urban planning for the purpose of optimizing the effectiveness of a community’s land use and infrastructure. They formulate plans for the development and management of urban and suburban areas, typically analyzing land use compatibility as well as economic, environmental and social trends. In developing the plan for a community (whether commercial, residential, agricultural, natural or recreational), urban planners must also consider a wide array of issues such as sustainability, air pollution, traffic congestion, crime, land values, legislation and zoning codes.

The importance of the urban planner is increasing throughout the 21st century, as modern society begins to face issues of increased population growth, climate change and unsustainable development. An urban planner could be considered a green collar professional.[clarification needed]

References

911. ”What is Urban Planning” (retrieved April 24, 2015)
https://mcgill.ca/urbanplanning/planning

“Modern urban planning emerged as a profession in the early decades of the 20th century, largely as a response to the appalling** sanitary, social, and economic **conditions of rapidly-growing industrial cities. Initially the disciplines of architecture and civil engineering provided the nucleus of concerned professionals. They were joined by** public health **specialists, economists, sociologists, lawyers, and geographers, as the complexities of managing cities came to be more fully understood. Contemporary urban and regional planning techniques for survey, analysis, design, and implementation developed from an interdisciplinary synthesis of these fields. Today, urban planning can be described as a technical and political process concerned with the welfare of people, control of the use of land, design of the urban environment including transportation and communication networks, and protection and enhancement of the natural environment.”

912. Van Assche, K., Beunen, R., Duineveld, M., & de Jong, H. (2013). Co-evolutions of planning and design: Risks and benefits of design perspectives in planning systems. Planning Theory, 12(2), 177-198.
913. Taylor, Nigel (2007). Urban Planning Theory since 1945, London, Sage.
914. https://www.planning.org/aboutplanning/whatisplanning.htm: "What Is Planning?". www.planning.org. Retrieved 2015-09-28.
915. https://www.planetizen.com/node/73570/how-planners-use-planning-theory: How Planners Use Planning Theory. Andrew Whitmore of the University of North Carolina Department of Urban and Regional Planning identifies planning theory in everyday practice.

E.1.2 On the Form of This Research Note

The present form of this research note, as of June 17, 2018: 10:12 am, is that of recording a development. The development is that of *trying to come to grips with what urban planning is*. We have made the decision, from an early start, that urban planning *"as a whole"* is a collection of one master and an evolving number of (initially zero) derived urban planning behaviours. Here we have made the choice to model the various behaviours of a complex of urban planning functions.

E.1.3 On the Structure of this Research Note

The page references in the items below refer to the first page of the part, section or subsections listed.

- It is always a good idea to study the contents listing. The author have made some effort in structuring the presentation. And the result of this effort is obviously reflected in the contents listing.
- Section E.3 [Page 317] can be skipped in any reading by those familiar with ***triptych*** approach to software development, ***formal methods***, my work on ***domain science & engineering***, etc. – topics that are otherwise covered in Sect. E.4. Sect. E.5 reviews the changes of my ***domain analysis & description calculus***, wrt. [2]. These changes take effect in our treatments of parts E.6 and E.11.

The next two parts are concerned with the [research &] development of a **model** of urban analysis and planning.

- Section E.6 [Page 320] treats the endurants of urban analysis and planning. It unfolds the model in four stages:
  - Sect. E.7 [Page 321] analyses & describes the universe of discourse, the structures and the (atomic) parts;
  - Sect. E.8 [Page 324] analyses & describes the unique identifiers of all atomic parts;
  - Sect. E.9 [Page 329] analyses & describes the mereologies of all atomic parts;
  - Sect. E.10 [Page 332] analyses & describes the attributes of all atomic parts.
- Section E.11 treats the perdurants of urban analysis and planning. It further unfolds the model in four stages:
  - Sect. E.12 [Page 341] calculates behaviours from parts;
  - Sect. E.13 [Page 343] analyses & describes channels by means of which the behaviours can synchronise & communicate;
  - Sect. E.14 [Page 346] calculate the basics of all atomic behaviours and define these behaviours; and
  - Sect. E.15 [Page 360] finally suggests an initial composition of the atomic behaviours.
- Section E.16 collects a number of "loose" ends:
  - Subsect. E.16.1 [Page 361] laments over the lack of assertions related to liveness and deadlock freeness of the defined behaviours and their initialisation;
  - Subsect. E.16.2 [Page 362] points out that documents, their distribution and sharing, play a central rôle in urban analysis and planning;
  - Subsect. E.16.3 [Page 362] muses over issues of validation and verification of the proposed model of urban analysis and planning; and
  - Subsect. E.16.4 [Page 362] points out that the model of urban analysis and planning implies a number of issues with respect to the organisation and management of urban analysis and planning projects.

E.2 An Urban Planning System

E.2.1 A First Iteration Overview

We think of urban planning to be "dividable" into master urban planning, master_planner, and derived urban plannings, $\mathsf{derived\_planner}_i$, where sub-index i indicate that there may be several, i.e., $i \in \{d_1, d_2, ..., d_n\}$, such derived urban plannings.

We think of master urban planning to "convert" physical (geographic, that is, geodetic, cadestral, geotechnical, meteorological, etc.) information about the land area to be developed into a master plan, that is, cartographic, cadestral and other such information (zoning, etc.). And we think of derived urban planning to "convert" master plans into societal and/or technological plans. Societal and technological urban planning concerns are typical such as ***industry zones, commercial (i.e., office and shopping) zones, residential zones, recreational areas***, etc., and ***health care, schools, transport, electricity, water, waste***, etc.

Each urban planning *behaviour*, whether 'master' or 'derived', is seen as a *sequence* of the applications of "the same" urban planning *function*, – but possibly to different goals so that each application (of "the same" urban planning *action*) resolves a sub-goal. Each urban planning action takes a number of information *arguments* and yield information *results*. The master urban planning behaviour may **start** one or more derived urban planning behaviours, $\mathsf{derived\_planner}_i$, at the end of "completion" of a master urban planning *action*. Let $\{d_1, d_2, ..., d_n\}$ index separate derived urban plannings, each concerned with a distinct, i.e., reasonably delineated technological and/or societal urban planning concern. During master urban planning actions may start any of these derived urban plannings once.

Thus we think of urban planning as a system of a single master urban planning process (i.e., behaviour), master_planner, which "spawns" zero, one or more (but a definite number of) derived urban planning processes (i.e., behaviours), $\mathsf{derived\_planner}_j$. Derived urban planning processes, $\mathsf{derived\_planner}_j$, may themselves start other derived urban planning processes, $\mathsf{derived\_planner}_i$, $\mathsf{derived\_planner}_k$, ..., $\mathsf{derived\_planner}_\ell$.

Figure E.1 is intended to illustrate the following: At time $t0$ a master urban planning is started. At time $t1$ the master urban planning initiates a number of derived urban development, $D1, ..., Di$. At time $t2$ the master urban planning initiates the Dj derived urban planning. At time $t3$ the derived urban planning Di initiates two derived urban plannings, Dk and $D\ell$. At time $t4$ the master urban planning ends. And at time $t5$ all urban plannings have ended. .

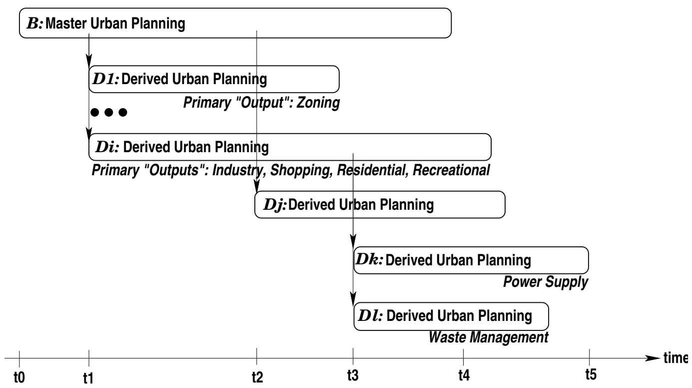


Fig. E.1. An Urban Planning Development

Urban planning actions are provided with "input" in the form of either geographic, geodetic, geotechnical, meteorological, etc., information, tusm:TUSm, or auxiliary information, m_aux:mAUX[3], or requirements information, m_req:mREQ. We shall detail issues of the urban space, auxiliary and requirements information later.

[3] The m_ value prefixes and the m type prefixes shall designate master urban planning entities.

E.2.2 A Visual Rendition of Urban Planning Development

We examine concepts of urban analysis and planning. We refer to Fig. E.2 [Page 316].There is ***the urban space:*** tus:TUS, which we treat as a ***part*** and as a ***behaviour***. There are a distinct urban space ***analysers***, distinctly named (i.e., indexed) $\{a_1,a_2,...,a_a\}$, treated as [parts and] ***behaviours***. There is one ***master planner***, treated as a [part and as a] ***behaviour***. There are p distinctly named ***derived [urban] planners***, distinctly named (i.e., indexed) $\{d_1,d_2,...,d_p\}$ and treated as [parts and] ***behaviours***.

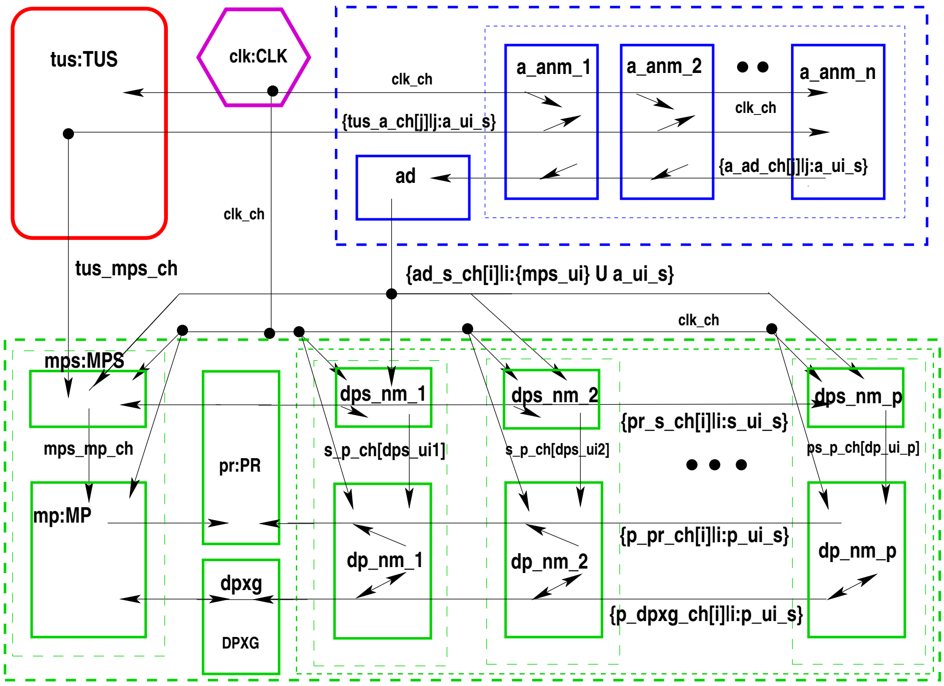


Fig. E.2. An Urban Analysis and Planning System

The behaviours (synchronise and) communicate via ***channels***. An array of channels communicate ***urban space attribute values*** to requesting analysers. The analysers provide ***analyses*** to all planners. The planners obtain ***input arguments*** from "their" servers. The planners provide ***result values*** to the common output result server. And the derived planner index generator provide possibly empty sets of ***derived planner indices*** to all planners.

Emphasis, in this research note, is on the ***information*** (abstract "data") and ***functions*** and ***behaviours*** of urban space analysis and planning – and their ***interaction***. We separate ***urban space analysis*** from ***urban planning***. ***Urban space analysers*** analyse [existing] urban spaces and produce ***analyses***. ***Urban planners*** analyse the analysis results and, in case of the master planning, also the urban space [itself] – and produce plans and other information. The ***master [urban] planner*** produces a master plan [and other information]. The ***derived [urban] planners*** produce derivative [urban] plans [and other information].

That is, we thus distinguish between the two kinds of urban planning: the ***master***, 'ab initio', behaviour of determining "the general layout of the land (!)", and the ***derived***, 'follow-up', behaviours focused on social and technological infrastructures. Master urban planning applies to descriptions of "the land": ***geographic, that is, geodetic, cadastral, geotechnical, meteorological, socio-economic and rules & regulations.*** Examples of derived urban plannings are such which are focused on humans and on social and technological artifacts: ***industry zones, commercial (i.e., office and shopping) zones, residential zones, recreational areas***, etc., and ***health care, schools, transport, electricity, water, waste***, etc.

E.3 Method

Several factors necessitated this part of this case study.

- In Sect. E.4, ["Prelude"] we briefly present basic issues of formal development.
- In Sect. E.5 ["Review & Refinement of the Method"] we then
  - review, in Sect. E.5.1 ["Review of Manifest Domains: Analysis & Description"] the specific approach basically taken when we describe manifest domains [2] and,
  - as a result of a number of recent (2016–2017) ***experimental research & engineering*** work, [34, 33, 35, 29, 241, 36],
  - we refine the approach described in [2], Sect. E.5.2 ["Refinement of the Method"].

E.4 Prelude

E.4.1 A Triptych of Software Development

Before hardware and software systems can be designed and coded we must have a reasonable grasp of "its" requirements; before requirements can be prescribed we must have a reasonable grasp of "the underlying" domain. To us, therefore, software engineering contains the three sub-disciplines:

- domain engineering,
- requirements engineering and
- software design.

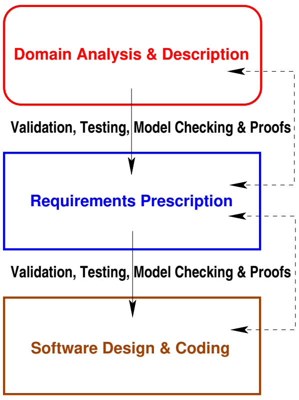


Fig. E.3. The Triptych of Software Development

By a domain description we understand a collection of pairs of narrative and of commensurate formal texts, where each pair describes either aspects of an endurant entity (i.e., information) or aspects of a perdurant entity (i.e., an action, event or behaviour).

E.4.2 On Formality

We consider software programs to be formal, i.e., mathematical, quantities — rather than of social/psychological interest. We wish to be able to reason about software, whether programs, or program specifications, or requirements prescriptions, or domain descriptions. Although we shall only try to understand some facets of the domain of urban planning we shall eventually let such an understanding, in the form of a precise, formal, mathematical, although non-deterministic, i.e., "multiple choice", description be the basis for subsequent requirements prescriptions for software support, and, again, eventually, "the real software itself", that is, tools, for urban planners. We do so, so that we can argue, eventually prove formally, that the software ***is correct*** with respect to the (i.e., its) formally prescribed requirements, and that the software ***meets customer***, i.e., domain users' ***expectations*** – as expressed in the formal domain description.

E.4.3 On Describing Domains

If we can describe some domain phenomenon in logical statements and if these can be transcribed into some form of mathematical logic and set theory then we may have to describe it: narratively and formally. That is, even though it may be humanly or even technologically very cumbersome or even impossible to implement what is described we may find it necessary to describe it. ***As to when we have to describe something – that is another matter!***[4] Let us give an example: The example is that of the domain of ***documents.*** Documents may be ***created, edited, read, copied, referred to,*** and ***shredded***. We may talk, meaningfully, that is, rationally, logically, about the previous ***version*** of a document, and hence we may be obliged to model document versions as from their first creation, ***who created, who edited, who read, who copied,*** and ***who shredded*** (sic !) a document, including, perhaps, the ***location*** and ***time*** of these operations, ***how they were edited***, etc., etc. Let us take another example. As for the meteorological properties of any specific geographic area, these properties, like temperature, humidity, wind, etc., vary, in reality, continuously over time, from location to location, including altitude. In modeling meteorological properties we may be well-served when modeling exactly their continuous, however "sporadic" nature. To a first approximation we do not have to bother as to whether we can actually "implement" the recording of such continuous, "sporadic" "behaviours". In that sense the domain analyser cum describer is expected to be like the physicists,[5] certainly not like programmers. That is: ***the domain analyser cum describer are not necessarily describing computable domains.***

E.4.4 Reiterating Domain Modeling

Any domain description is an approximation. One cannot ever hope to have described all facets of any domain. So, in setting out to analyse & describe a domain one is not trying to produce a definitive, final, model; one is merely studying and recording (some) results of that study. One is prepared to reiterate the study and produce alternative models. From such models one can develop requirements, [9], for software that in one way or another support activities of the domain. If you are to seriously develop software in this way, for example for the support of urban planners, then you must be prepared to "restart" the process, to develop, from scratch, a domain model. You have a basis from which to start, namely this report [35]. But do not try to simply modify it. Study [35] in depth, but rethink that basis. A description, any description, can be improved. Perhaps the emphasis should be refocused. For the example of software (incl. IT) support for the keeping, production, editing, etc., of the very many documents that are needed during urban planning, you may, in addition to refocusing the present report's focus on the documents of the very many document categories that are presumed, introduced and further elaborated upon in the present report, also study [29]. A principle guiding us in the reformulation of a domain model to be the basis for a specific software product is that we must strive to document all the assumptions about the context in which this software is to serve – otherwise we cannot hope to achieve a product that meets its customers expectations.

E.4.5 Partial, Precise, and Approximate Descriptions

By a ***partial description*** we mean a description which covers only a fraction of the domain as a group of people working in that domain, that is, professionals, would otherwise talk about. Descriptions are

[4] We may find occasions in this document to discuss this "other matter" !

[5] It is written above: that domain descriptions are based on mathematical logic and set theory. Yes, unfortunately ! To properly describe domains involving continuity we need "mix" logic with classical calculus: differential equations, integrals, etc. And here we have nothing to say: the ability, in an informed ways, to blend mathematical logic and set theoretic descriptions with differential equations, integrals, etc., is almost non-existent as of 2017/2018 !

here taken to describe ***behaviours:*** *first "do this", then "do that"!* By a ***precise description*** we mean a description which in whatever behaviour it describes, partially or fully, does so precisely, that is, it is precisely as described, no more, no less. By an ***approximate description*** we mean a description which in whatever behavior it describes, partially or fully, even when precisely so, allows for a set of interpretations.

We shall then avail ourselves of two forms of 'approximation': ***internal non-determinism*** and ***external non-determinism***. By ***internal non-deterministic behaviour*** we shall mean a behaviour whose "next step, next move" is "determined" by some "own flipping a coin". By ***external non-deterministic behaviour*** we shall mean a behaviour whose "next step, next move" is "determined" by some "outside demon"! In describing urban planning we shall allow for: partial descriptions: not all is described and what has been selected for description has been so, perhaps rather arbitrarily, by us, i.e., me, and both forms of 'approximation'. We shall endeavour to indicate where and why we present only partial descriptions, and deploy 'approximation'.

E.4.6 On Formal Notations

To be able to *prove formal correctness* and *meeting customer expectations* we avail ourselves of some formal notation. In this research note we use the `RAISE` [64] `Specification Language`, `RSL`, [22]. Other formal notations, such as `Alloy` [65], `Event B` [66], `VDM-SL` citevdm or `Z` [70] could be used. We choose `RSL` since it, to our taste, nicely embodies Hoare's concept of *Communicating Sequential Processes*, `CSP` [23]

E.5 Review & Refinement of the Method

The basis for the kind of domain analysis & description of this case study is [2]. It was submitted 19 Dec. 2014 and (paper) published in March 2017. Between those dates and in particular since March 2017 a number of ***experimental engineering cum research*** took place. We mention some of these. A ***credit card system*** modeling, [34, May 2016]. A ***weather forecast system*** modeling, [33, Nov. 2017]. The first phase, March 2017–July 2017, of this ***urban planning*** project [35]. A ***document system***, [29, July 2017]. A ***clarification*** of concepts so-called ***implicit/explicit semantics***, [241, Oct. 2017]. A ***swarms of drones*** modeling experiment, [36, Nov.–Dec.].

E.5.1 Review of Manifest Domains: Analysis & Description

We refer to [2, submitted 19 Dec. 2014, published March 2017] We present a terse, itemised summary of the method outlined in that paper:

- First we analyse & describe *endurants*:
  - the *form* of *parts, components* and *materials*.
  - then the *qualities* of *parts, components* and *materials*, that is:
    - the *unique identifiers* of *parts* and *components*, then
    - the *mereology* of *parts*, and finally
    - the *attributes* of *parts* and *materials*.
- Then we analyse & describe *perdurants*:
  - the notion of *domain states*,
  - the *actions*, then
  - the *events*, and finally
  - the *behaviours*.
- As part of the description of behaviours we analyse & describe
  - *channels*

We can summarise this in the *ontology diagram*, cf. Fig. E.4 [Page 320] of [2].

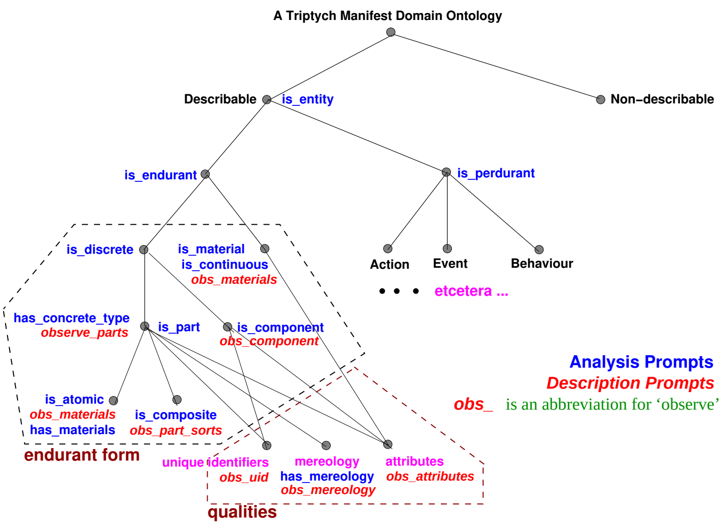


Fig. E.4. The Previous Upper Ontology

E.5.2 Refinement of the Method

- First we analyse & describe *endurants*:
  - the *form* of **structures**, *parts*, *components* and *materials*. The refinement of the *manifest domain analysis & description* approach is the addition of *endurant* <u>*structures*</u>. Structures are "abstract composite parts", though with no *qualities*,
  - then we analyse & describe *qualities* of *parts*, *components* and *materials*, that is:
    - the *unique identifiers* of *parts* and *components*, then
    - the *mereology* of *parts*, and finally
    - the *attributes* of *parts* and *materials*.
- Then we analyse & describe *perdurants*:
  - the notion of *domain states* and
  - the **channels**. We observe that this item has been "moved" to "before" analysis & description of subsequent analyses & descriptions.
  - The *behaviours*.
  - As part of the description of behaviours we analyse & describe
    - the *actions* and
    - the *events*

We can summarise this in a revised *ontology diagram*, cf. Fig. E.5 [Page 321].

E.6 ENDURANTS

By an *entity* we shall understand a phenomenon, i.e., something that can be observed, i.e., be seen or touched by humans, or that can be conceive d as an abstraction of an entity. We further demand that an entity can be objectively described

By an *endurant* we shall understand an entity that can be observed or conceived and described as a "complete thing" at no matter which given snapshot of time. Were we to "freeze" time we would still be able to observe the entire endurant.

By a *discrete endurant* we shall understand an endurant which is separate, individual or distinct in form or concept.

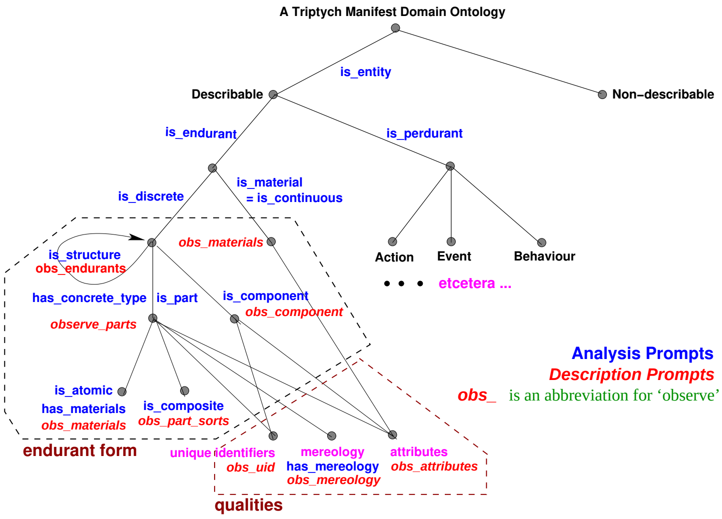


Fig. E.5. The Refined Upper Ontology

By a *part* we shall understand a discrete endurant which the domain engineer chooses to endow with *internal qualities*[6] such as *unique identification*, *mereology*, and one or more *attributes*. We shall define these three categories in Sects. E.8, E.9, respectively Sect. E.10. We refer in general to [2].

In this, a major section of this case study, we shall cover

- Sect. E.7: Parts,
- Sect. E.8: Unique Identifiers,
- Sect. E.9: Mereology, and
- Sect. E.10: Attributes.

E.7 Structures and Parts

From an epistemological[7] point of view a study of the parts of a universe of discourse is often the way to understand *"who the players"* of that domain are. From the point of view of [2] *knowledge about parts lead to knowledge about behaviours.* This is the reason, then, for our interest in parts.

E.7.1 The Urban Space, Clock, Analysis & Planning Complex

The domain-of-interest, i.e., the universe of discourse for this case study is that of *the urban space analysis & planning complex* – where the ampersand, '&', shall designate that we consider this complex as 'one' !

916. The *universe of discourse*, UoD, is here seen as a *structure* of four elements:
   a. a *clock*, CLK,
   b. *the urban space*, TUS,
   c. an *analyser aggregate*, AA,
   d. the *planner aggregate*, PA,

type
916 UoD, CLK, TUS, AAG, PA

[6] – where by *external qualities* of an endurant we mean whether it is discrete of *continuous*, whether it is a parts, or a *component* – such as these are defined in [2].
[7] Epistemology is the branch of philosophy concerned with the theory of knowledge.

value
916a obs_CLK: UoD → CLK
916b obs_TUS: UoD → TUS
916c obs_AAG: UoD → AAG
916d obs_PA: UoD → PA

The clock and the urban space are here considered *atomic*, the *analyser aggregate*, AA, and the the *planner aggregate*, PA, are here seen as *structures*.

E.7.2 The Analyser Structure and Named Analysers

917. The *analyser structure* consists of
    a. a *structure*, AC, which consists of two elements:
        i. a *structure* of an indexed set, hence named *analysers*,
        ii. A_{anm_1}, A_{anm_2}, ..., A_{anm_n},

    and
918. an *atomic analysis depository*, AD.

There is therefore defined a set, ANms, of

919. *analyser names*: $\{anm_1, anm_2, \ldots, anm_n\}$, where $n \geq 0$.

type
917 AA, AC, A, AD
917(a)i A = A_{anm_1} | A_{anm_2} | ... | A_{anm_n}
919 ANms = $\{|anm_1, anm_2, ..., anm_n|\}$
value
917a obs_AC: AA → AC
917(a)ii obs_AC$_{anm_i}$: AC → A_{anm_i}, i:[1..n]
918 obs_AD: AA → AD

Analysers and the *analysis depository* are here seen as atomic parts.

E.7.3 The Planner Structure

920. The composite *planner structure* part, consists of
    a. a *master planner structure*, MPA, which consists of
        i. an atomic *master planner server*, MPS, and
        ii. an atomic *master planner*, MP, and
    b. a *derived planner structure*, DPA, which consists of
        i. a *structure* in the form of an indexed set of (hence named) *derived planner structures*, DPC_{nm_j}, $j : [1..p]$, which each consists of
            1. a atomic *derived planner servers*, DPS_{nm_j}, $j : [1..p]$, and
            2. a atomic *derived planners*, DP_{nm_j}, $j : [1..p]$;
    c. an atomic *plan repository*, PR, and
    d. an atomic *derived planner index generator*, DPXG.

type
920 PA, MPA, MPS, MP, DPA, DPC_{nm_j}, DPS_{nm_j}, DP_{nm_j}, i:[1..p]

value
920a obs_MPA: PA → MPA
  920(a)i obs_MPS: MPA → MPS
  920(a)ii obs_MP: MPA → MP
920b obs_DPA: PA → DPA

920(b)i obs_DPC$_{nm_j}$: DPA → DPC$_{nm_j}$, i:[1..p]
920(b)i1 obs_DPS$_{nm_j}$: DPC$_{nm_j}$ → DPS$_{nm_j}$, i:[1..p]
920(b)i2 obs_DP$_{nm_j}$: DPC$_{nm_j}$ → DP$_{nm_j}$, i:[1..p]
920c obs_PR: PA → PR
920d obs_DPXG: → DPXG

We have chosen to model as *structures* what could have been modeled as *composite* parts. If we were to domain analyse & describe *management & organisation* facets of the urban space analysis & planning domain then we might have chosen to model some of these *structures* instead as *composite parts*.

E.7.4 Atomic Parts

The following are seen as atomic parts:

- *clock*,
- *urban space*,
- *analysis deposit*,
- each *analyser* in the indexed set of *analyser*$_{anm_i}$s,
- *master planner server*,
- *master planner*,
- each *server* in the indexed set of *derived planner server*$_{nm_j}$s,
- each *planner* in the indexed set of *derived planner*$_{nm_j}$s,
- *derived planner index generator*.
- *plan repository* and

We shall return to the these atomic part sorts when we explore their properties: *unique identifiers, mereologies* and *attributes*.

E.7.5 Preview of Structures and Parts

Let us take a preview of the parts, see Fig. E.6 [Page 323].

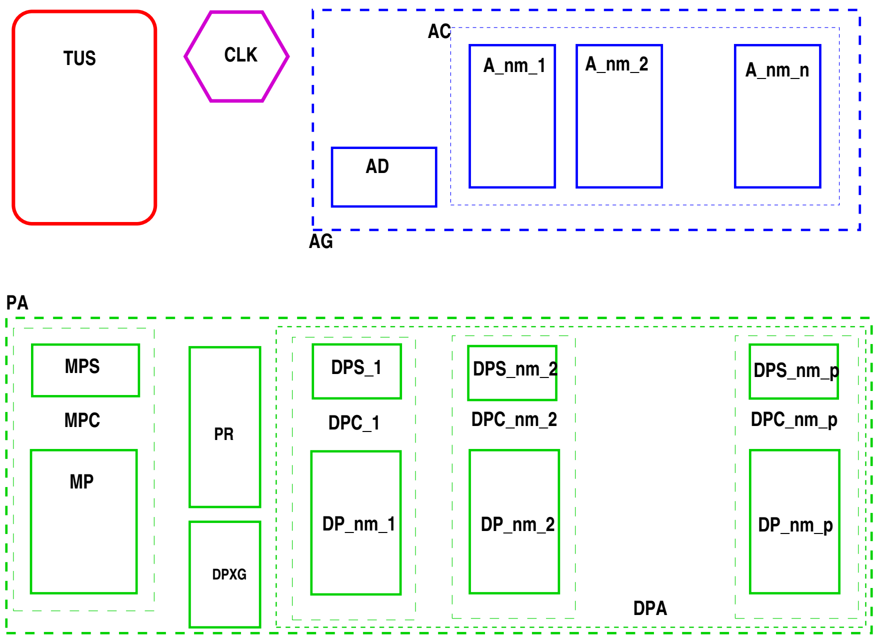


Fig. E.6. The Urban Analysis and Planning System: **Structures and Atomic Parts**

E.7.6 Planner Names

921. There is therefore defined identical sets of *derived planner aggregate names*, *derived planner server names*, and *derived planner names*: $\{dnm_1, dnm_2, \ldots, dnm_p\}$, where $g \geq 0$.

type
921 DNms = $\{|dnm_1, dnm_2, ..., dnm_p|\}$uod-084

E.7.7 Individual and Sets of Atomic Parts

In this closing section of Sect. E.7.7 we shall identify individual and sets of atomic parts.

922. We postulate an arbitrary *universe of discourse*, uod:UoD and let that be a constant value from which we the calculate a number of individual and sets of atomic parts.
923. There is the *clock*, clk:CLK,
924. *the urban space*, tus:TUS,
925. the set of *analysers*, a_{anm_i}:A_{anm_i}, i:[1..n],
926. the *analysis depository*, ad,
927. the *master planner server*, mps:MPS,
928. the *master planner*, mp:MP,
929. the set of *derived plannner servers*, $\{dps_{nm_i}$:$\mathsf{DPS}_{nm_i} \mid i$:[1..$p$]$\}$,
930. the set of *derived planners*, $\{dp_{nm_i}$:$\mathsf{DP}_{nm_i} \mid i$:[1..$p$]$\}$,
931. the *derived plan index generator*, $dpxg$,
932. the *plan repository*, pr, and
933. the set of pairs of *derived server* and *derived planner*s, sps.

value
922 uod : UoD
923 clk : CLK = obs_CLK(uod)
924 tus : TUS = obs_TUS(uod)
925 ans : A_{anm_i}**-set**, i:[1..n] =
925 { obs_A_{anm_i}(aa) | aa∈(obs_AA(uod)), i:[1..n]}
926 ad : AD = obs_AD(obs_AA(uod))
927 mps : MPS = obs_MPS(obs_MPA(uod))
928 mp : MP = obs_MP(obs_MPA(uod))
929 $dpss$: DPS_{nm_i}**-set**, i:[1..p] =
929 { obs_DPS_{nm_i}(dpc_{nm_i}) |
929 dpc_{nm_i}:DPC_{nm_i}•dpc_{nm_i}∈obs_DPCS_{nm_i}(obs_DPA(uod)), i:[1..p] }
930 dps : DP_{nm_i}**-set**, i:[1..p] =
930 { obs_DP_{nm_i}(dpc_{nm_i}) |
930 dpc_{nm_i}:DPC_{nm_i}•dpc_{nm_i}∈obs_DPCS_{nm_i}(obs_DPA(uod)), i:[1..p] }
931 $dpxg$: DPXG = obs_DPXG(uod)
932 pr : PR = obs_PR(uod)
933 $spsps$: (DPS_{nm_i}×DP_{nm_i})**-set**, i:[1..p] =
933 { (obs_DPS_{nm_i}(dpc_{nm_i}),obs_DP_{nm_i}(dpc_{nm_i})) |
933 dpc_{nm_i}:DPC_{nm_i}•dpc_{nm_i}∈ obs_DPCS_{nm_i}(obs_DPA(uod)), i:[1..p] }

E.8 Unique Identifiers

We introduce a notion of unique identification of parts. We assume (i) that all parts, p, of any domain P, have unique identifiers, (ii) that unique identifiers (of parts p:P) are abstract values (of the unique identifier, π, sort Π_UI of parts p:P), (iii) such that distinct part sorts, P_i and P_j, have distinctly named unique identifier sorts, say Π_UI_i and Π_UI_j, (iv) that all $\pi{:}\Pi\_UI_i$ and $\pi_j{:}\Pi\_UI_j$ are distinct, and (v) that the observer function uid_P applied to p yields the unique identifier, say $\pi{:}\Pi\_UI$, of p.

The analysis & description of unique identification is a prerequisite for talking about mereologies of universes of discourse, and the analysis & description of mereologies are a means for understanding how parts relate to one another.

Since we model as *structures* what elsewhere might have been modeled as *composite parts* we shall only deal with *unique identifiers* of *atomic parts*.

E.8.1 Urban Space Unique Identifier

934. The urban space has a unique identifier.

type
934 TUS_UI
value
934 uid_TSU: TSU → TUS_UI

E.8.2 Analyser Unique Identifiers

935. Each analyser has a unique identifier.
936. The analysis depository has a unique identifier.

type
935 A_UI = $\mathsf{A\_UI}_{anm_1}$ | $\mathsf{A\_UI}_{anm_2}$ | ... | $\mathsf{A\_UI}_{anm_n}$
936 AD_UI
value
935 uid_A: A_{nm_i} → $\mathsf{A\_UI}_{nm_i}, i:[1..n]$
936 uid_AD: AD → AD_UI
axiom
935 ∀ a_{nm_i}:A_{nm_i} •
935 **let** $\mathsf{a\_ui}_{nm_i}$ = uid_A(a_{nm_i}) **in** $\mathsf{a\_ui}_{nm_i} \simeq nm_i$ **end**

The mathematical symbol $\simeq$ (in this case study) denotes *isomorphy*.

E.8.3 Master Planner Server Unique Identifier

937. The *unique identifier* of the *master planner server*.

type
937 MPS_UI
value
937 uid_MPS: MPS → MPS_UI

E.8.4 Master Planner Unique Identifier

938. The *unique identifier* of the *master planner*.

type
938 MP_UI
value
938 uid_MP: MP → MP_UI

E.8.5 Derived Planner Server Unique Identifier

939. The *unique identifiers* of *derived planner servers*.

type
939 $\mathsf{DPS\_UI} = \mathsf{DPS\_UI}_{nm_1} \mid \mathsf{DPS\_UI}_{nm_2} \mid \ldots \mid \mathsf{DPS\_UI}_{nm_p}$
value
939 $\mathsf{uid\_DPS}\colon \mathsf{DPS}_{nm_i} \to \mathsf{DPS\_UI}_{nm_i},\ i:[1..p]$
axiom
939 $\forall\ \mathsf{dps}_{nm_i}{:}\mathsf{DPS}_{nm_i} \bullet$
939 **let** $\mathsf{dps\_ui}_{nm_i} = \mathsf{uid\_DPS}(\mathsf{dps}_{nm_i})$ **in** $\mathsf{dps\_ui}_{nm_i} \simeq nm_i$ **end**

E.8.6 Derived Planner Unique Identifier

940. The *unique identifiers* of *derived planners*.

type
940 $\mathsf{DP\_UI} = \mathsf{DP\_UI}_{nm_1} \mid \mathsf{DP\_UI}_{nm_2} \mid \ldots \mid \mathsf{DP\_UI}_{nm_p}$
value
940 $\mathsf{uid\_DP}\colon \mathsf{DP}_{nm_i} \to \mathsf{DP\_UI}_{nm_i},\ i:[1..p]$
axiom
940 $\forall\ \mathsf{dp}_{nm_i}{:}\mathsf{DP}_{nm_i} \bullet$
940 **let** $\mathsf{dp\_ui}_{nm_i} = \mathsf{uid\_DP}(\mathsf{dp}_{nm_i})$ **in** $\mathsf{dp\_ui}_{nm_i} \simeq nm_i$ **end**

E.8.7 Derived Plan Index Generator Identifier

941. The *unique identifier* of *derived plan index generator*:

type
941 DPXG_UI
value
941 $\mathsf{uid\_DPXG}\colon \mathsf{DPXG} \to \mathsf{DPXG\_UI}$

E.8.8 Plan Repository

942. The *unique identifier* of *plan repository*:

type
942 PR_UI
value
942 $\mathsf{uid\_PR}\colon \mathsf{PR} \to \mathsf{PR\_UI}$

E.8.9 Uniqueness of Identifiers

943. The identifiers of all analysers are distinct.
944. The identifiers of all derived planner servers are distinct.
945. The identifiers of all derived planners are distinct.
946. The identifiers of all other atomic parts are distinct.
947. And the identifiers of all atomic parts are distinct.

943 $\mathbf{card}\, ans = \mathbf{card}\, a_{ui}s$
944 $\mathbf{card}\, dpss = \mathbf{card}\, dps_{ui}s$
945 $\mathbf{card}\, dps = \mathbf{card}\, dp_{ui}s$
946 $\mathbf{card}\{clk_{ui}, tus_{ui}, ad_{ui}, mps_{ui}, mp_{ui}, dpxg_{ui}, plas_{ui}\} = 7$
947 $\cap(ans, dpss, dps, \{clk_{ui}, tus_{ui}, ad_{ui}, mps_{ui}, mp_{ui}, dpxg_{ui}, plas_{ui}\}) = \{\}$

E.8.10 Indices and Index Sets

It will turn out to be convenient, in the following, to introduce a number of index sets.

948. There is the *clock* identifier, clk_{ui}:CLK_UI.
949. There is *the urban space* identifier, tus_{ui}:TUS_UI.
950. There is the set, $a_{ui}s$:A_UI-**set**, of the identifiers of all *analysers*.
951. The *analysis depository* identifier, ad_{ui}.
952. There is the *master planner server* identifier, mps_{ui}:MPS_UI.
953. There is the *master planner* identifier, mp_{ui}:MP_UI.
954. There is the set, $dps_{ui}s$:DPS_UI-**set**, of the identifiers of all *derived planner servers*.
955. There is the set, $dp_{ui}s$:DP_UI-**set**, of the identifiers of all derived planners.
956. There is the *derived plan index generator* identifier, $dpxg_{ui}$:DPXG_UI.
957. And there is the *plan repository* identifier, pr_{ui}:PR_UI.

value
948 clk_{ui} : CLK_UI = uid_CLK(uod)
949 tus_{ui} : TUS_UI = uid_TUS(uod)
950 $a_{ui}s$: A_UI-**set** = {uid_A(a)|a:A•a ∈ ans}
951 ad_{ui} : AD_UI = uid_AD(ad)
952 mps_{ui} : MPS_UI = uid_MPS(mps)
953 mp_{ui} : MP_UI = uid_MP(mp)
954 $dps_{ui}s$: DPS_UI-**set** = {uid_DPS(dps)|dps:DPS•dps ∈ $dpss$}
955 $dp_{ui}s$: DP_UI-**set** = {uid_DP(dp)|dp:DP•dp ∈ dps}
956 $dpxg_{ui}$: DPXG_UI = uid_DPXG($dpxg$)
957 pr_{ui} : PR_UI = uid_PR(pr)

958. There is also the set of identifiers for all servers: $ps_{ui}s$:(MPS_UI|DPS_UI)-**set**,
959. there is then the set of identifiers for all planners: $ps_{ui}s$:(MP_UI|DP_UI)-**set**,
960. there is finally the set of pairs of paired *derived planner server* and *derived planner* identifiers.
961. there is a map from the unique derived server identifiers to their "paired" unique derived planner identifiers, and
962. there is finally the reverse map from planner to server identifiers.

value
958 $s_{ui}s$: (MPS_UI|DPS_UI)-**set** = {mps_{ui}} ∪ $dps_{ui}s$
959 $p_{ui}s$: (MP_UI|DP_UI)-**set** = {mp_{ui}} ∪ $dp_{ui}s$
960 $sips$: (DPS_UI×DP_UI)-**set** = {(uid_DPS(dps),uid_DP(dp))|(dps,dp):(DPS×DP)•(dps,dp)∈sps}
961 si_pi_m : DPS_UI $\overrightarrow{m}$ DP_UI = [uid_DPS(dps)↦uid_DP(dp)|(dps,dp):(DPS×DP)•(dps,dp)∈sps]
962 pi_si_m : DP_UI $\overrightarrow{m}$ DPS_UI = [uid_DP(dp)↦uid_DPS(dps)|(dps,dp):(DPS×DP)•(dps,dp)∈sps]

E.8.11 Retrieval of Parts from their Identifiers

963. Given the global set $dpss$, cf. 929 [Page 324], i.e., the set of all derived servers, and given a unique planner server identifier, we can calculate the derived server with that identifier.
964. Given the global set dps, cf. 930 [Page 324], the set of all derived planners, and given a unique derived planner identifier, we can calculate the derived planner with that identifier.

value
963 c_s: $dpss$ → DPS_UI → DPS
963 c_s($dpss$)(dps_ui) ≡ **let** dps:DPS•dps ∈$dpss$∧uid_DPS(dps)=dps_ui **in** dps **end**
964 c_p: dps → DP_UI → DP
964 c_p(dps)(dp_ui) ≡ **let** dp:DP•dp ∈dps∧uid_DPS(dp)=dp_ui **in** dp **end**

E.8.12 A Bijection: Derived Planner Names and Derived Planner Identifiers

We can postulate a unique relation between the names, dn:DNm-**set**, i.e., the names dn∈DNms, and the unique identifiers of the named planners:

965. We can claim that there is a function, extr_DNm, from the unique identifiers of derived planner servers to the names of these unique identifiers.
966. Similarly can claim that there is a function, extr_DNm, from the unique identifiers of derived planners to the names of these unique identifiers.

value
965 extr_Nm: DPS_UI → DNm
965 extr_Nm(dps_ui) ≡ ...
966 extr_Nm: DP_UI → DNm
966 extr_Nm(dp_ui) ≡ ...
axiom
965 ∀ dps_ui1,dps_ui2:DPS_ui • dps_ui1≠dps_ui2 ⇒ extr_Nm(dps_ui1) ≠ extr_Nm(dps_ui1)
966 ∀ dp_ui1,dp_ui2:DP_ui • dp_ui1≠dp_ui2 ⇒ extr_Nm(dp_ui1) ≠ extr_Nm(dp_ui1)

967. Let dps_ui_dnm:DPS_UI_DNm, dp_ui_dnm:DP_UI_DNm stand for maps from derived planner server, respectively derived planner unique identifiers to derived planner names.
968. Let nm_dp_ui:Nm_DP_UI, nm_dp_ui:Nm_DP_UI stand for the reverse maps.
969. These maps are bijections.

type
967 DPS_UI_DNm: DPS_UI $\overrightarrow{m}$ DP_Nm
967 DP_UI_DNm: DP_UI $\overrightarrow{m}$ DP_Nm
968 DNm_DPS_UI: DP_Nm $\overrightarrow{m}$ DP_UI
968 DNm_DP_UI: DP_Nm $\overrightarrow{m}$ DP_UI
axiom
969 ∀ dps_ui_dnm:DPS_UI_DNm • dps_ui_dnm^{-1}·dps_ui_dnnm = λx.x
969 ∀ dp_ui_dnm:DP_UI_DNm • dp_ui_dnm^{-1}·dp_ui_dnnm = λx.x
969 ∀ dnm_dps_ui:DNm_DPS_UI • dnm_dps_ui^{-1}·dnm_dps_ui = λx.x
969 ∀ dnm_dp_ui:DNm_DP_UI • dp_ui_dnm^{-1}·dnm_dps_ui = λx.x
that is:
969 ∀ dps_ui_dnm:DPS_UI_DNm, dp_ui_dnm:DP_UI_DNm, dps_ui:DPS_UI •
969 dps_ui ∈ **dom** dps_ui_dnm ⇒ dp_ui_dnm(dps_ui_dnm(dps_ui)) = dps_ui
et cetera !

970. The function mk_DNm_DUI takes the set of all derived planner servers, respectively derived planners and produces bijective maps, dnm_dps_ui, respectively dnm_dp_ui.
971. Let *dnm_dps_ui*:DNm_DPS_UI and
972. *dnm_dp_ui*:DNm_DP_UI

stand for such [global] maps.

value
970 mk_Nm_DPS_UI: DPS$_{nm_i}$-**set** → DNm_DPS_UI
970 mk_Nm_DPS_UI(*dpss*) ≡ [uid_DPS(dps)↦extr_Nm(uid_DPS(dps))|dps:DPS•dps ∈ *dpss*]
970 mk_Nm_DP_UI: DP$_{nm_i}$-**set** → DNm_DP_UI
970 mk_Nm_DP_UI(*dps*) ≡ [uid_DP(dp)↦extr_Nm(uid_DP(dp))|dp:DP•dps ∈ *dps*]

971 *nm_dps_ui*:Nm_DPS_UI = mk_Nm_DPS_UI(*dps*)
972 *nm_dp_ui*:Nm_DP_UI = mk_Nm_DP_UI(*dps*)

E.9 Mereologies

Mereology (from the Greek $\mu\varepsilon\rho o\varsigma$ 'part') is the *theory of part-hood relations:* of *the relations of part to whole* and *the relations of part to part within a whole*[8].

Part mereologies inform of how parts relate to other parts. As we shall see in the section on *perdurants*, mereologies are the basis for analysing & describing communicating between part behaviours.

Again: since we model as *structures* what is elsewhere modeled as *composite parts* we shall only consider *mereologies* of *atomic parts*.

E.9.1 Clock Mereology

973. The clock is related to all those parts that create information, i.e., documents of interest to other parts. Time is then used to *time-stamp* those documents. These other parts are: the *urban space*, the *analysers*, the *planner servers* and the *planners*.

type
973 CLK_Mer = TSU_UI×A_UI-**set**×MPS_UI×MP_UI×DPS_UI-**set**×DP_UI-**set**
value
973 mereo_CLK: CLK → Clk_Mer
axiom
973 mereo_CLK(uod) = $(tus_{ui}, a_{ui}s, mps_{ui}, mp_{ui}, dps_{ui}s, dp_{ui}s)$

E.9.2 Urban Space Mereology

The urban space stands in relation to those parts which consume urban space information: the *clock* (in order to time stamp urban space information), the *analysers* and the *master planner server*.

974. The mereology of the urban space is a triple of the clock identifier, the identifier of the master planner server and the set of all analyser identifiers. all of which are provided with urban space information.
975. The constraint here is expressed in the 'the': for the universe of discourse it must be the master planner aggregate unique identifier and the set of exactly all the analyser unique identifiers for that universe.

type
974 TUS_Mer = CLK_UI × A_UI-**set** × MPS_UI
value
974 mereo_TUS: TUS → TUS_Mer
axiom
975 mereo_TUS(tus) = $(clk_{ui}, a_{ui}s, mps_{ui})$

E.9.3 Analyser Mereology

976. The mereology of a[ny] *analyser* is that of a triple: the *clock identifier*, *the urban space identifier*, and the *analysis depository identifier.*

type
976 A_Mer = CLK_UI × TUS_UI × AD_UI
value
976 mereo_A: A → A_Mer

[8] Achille Varzi: Mereology, http://plato.stanford.edu/entries/mereology/ 2009 and [38].

E.9.4 Analysis Depository Mereology

977. The mereology of the *analysis depository* is a triple: the *clock identifier*, the *master planner server identifier*, and the set of *derived planner server identifiers*.

type
977 AD_Mer = CLK_UI × MPS_UI × DPS_UI-**set**
value
977 mereo_AD: AD → AD_Mer

E.9.5 Master Planner Server Mereology

978. The *master planner server* mereology is a quadruplet of the clock identifier (time is used to time stamp input arguments, prepared by the server, to the planner), the urban space identifier, the analysis depository and the master planner identifier.
979. And for all universes of discourse these must be exactly those of that universe.

type
978 MPS_Mer = CLK_UI × TUS_UI × AD_UI × MP_UI
value
978 mereo_MPS: MPS → MPS_Mer
axiom
979 mereo_MPS(mps) = $(clk_{ui},tus_{ui},ad_{ui},mp_{ui})$

E.9.6 Master Planner Mereology

980. The mereology of the *master planner* is a triple of: the clock identifier[9], master server identifier[10], derived planner index generator identifier[11], and the plan repository identifier[12].

type
980 MP_Mer = CLK_UI × MPS_UI × DPXG_UI × PR_UI
value
980 mereo_MP: MP → MP_Mer
axiom
980 mereo_MP(mp) = $(clk_{ui},mps_{ui},dpxg_{ui},pr_{ui})$

E.9.7 Derived Planner Server Mereology

981. The *derived planner server* mereology is a quadruplet of:

the clock identifier[13], the set of all analyser identifiers[14], the plan repository identifier,[15] and the derived planner identifier[16].

[9] From the clock the planners obtain the time with which they stamp all information assembled by the plannner.
[10] from which the master planner obtains essential input arguments
[11] in collaboration with which the master planner obtains a possibly empty set of derived planning indices
[12] with which it posits and from which it obtains summaries of all urban planning plans produced so far.
[13] From the clock the servers obtain the time with which they stamp all information assembled by the servers.
[14] From the analysers the servers obtain analyses.
[15] In collaboration with the plan repository the planners deposit plans etc. and obtains summaries of all urban planning plans produced so far
[16] The server provides its associated planner with appropriate input arguments.

type
981 DPS_Mer = CLK_UI × AD_UI × PLAS_UI × DP_UI
value
981 mereo_DPS: DPS → DPS_Mer
axiom
981 ∀ (dps,dp):(DPS×DP) • (dps,dp)∈sps ⇒
981 mereo_DPS(dps) = (clk_{ui},ad_{ui},$plas_{ui}$,uid_DP(dp))

E.9.8 Derived Planner Mereology

982. The *derived planner* mereology is a quadruplet of:

the clock identifier, the derived plan server identifier, the derived plan index generator identifier, and the plan repository identifier.

type
982 DP_Mer = CLK_UI × DPS_UI × DPXG_UI × PR_UI
value
982 mereo_DP: DP → DP_Mer
axiom
982 ∀ (dps,dp):(DPS×DP) • (dps,dp)∈sps ⇒
982 mereo_DP(dp) = (clk_{ui},uid_DPS(dps),$dpxg_{ui}$,pr_{ui})

E.9.9 Derived Planner Index Generator Mereology

983. The mereology of the derived planner index generator is the set of all planner identifiers: master and derived.

type
983 DPXG_Mer = (MP_UI|DP_UI)-**set**
value
983 mereo_DPXG: DPXG → DPXG_Mer
axiom
983 mereo_DPXG($dpxg$) = $ps_{ui}s$

E.9.10 Plan Repository Mereology

984. The plan repository mereology is the set of all planner identifiers: master and derived.

984 PR_Mer = (MP_UI|DP_UI)-**set**
value
984 mereo_PR: PR → PR_Mer
axiom
984 mereo_PR(pr) = $ps_{ui}s$

E.10 Attributes

Parts are typically recognised because of their spatial form and are otherwise characterised by their intangible, but measurable attributes. That is, whereas endurants, whether discrete (as are parts and components) or continuous (as are materials), are physical, tangible, in the sense of being spatial (or being abstractions, i.e., concepts, of spatial endurants), attributes are intangible: cannot normally be touched, or seen, but can be objectively measured. Thus, in our quest for describing domains where humans play an active rôle, we rule out subjective "attributes": feelings, sentiments, moods. Thus we shall abstain, in our domain science also from matters of aesthetics. A formal concept, that is, a type, consists of all the entities which all have the same qualities. Thus removing a quality from an entity makes no sense: the entity of that type either becomes an entity of another type or ceases to exist (i.e., becomes a non-entity)

E.10.1 Clock Attribute

Time and Time Intervals and their Arithmetic

985. Time is modeled as a continuous entity.
986. One can subtract two times and obtain a time interval.
987. There is an "infinitesimally" smallest time interval, δt:T.
988. Time intervals are likewise modeled as continuous entities.
989. One can add or subtract a time interval to, resp. from a time and obtain a time.
990. One can compare two times, or two time intervals.
991. One can add and subtract time intervals.
992. One can multiply time intervals with real numbers.

type
985 T
986 TI
value
986 sub: T $\times$ T $\to$ TI
987 δt:TI
989 add,sub: TI $\times$ T $\to$ T
990 $<,\leq,=,\geq,>$: ((T$\times$T)|(TI$\times$TI)) $\to$ **Bool**
991 add,sub: TI $\times$ TI $\to$ TI
992 mpy: TI $\times$ **Real** $\to$ TI

The Attribute

993. The only attribute of a clock is time. It is a programmable attribute.

type
993 T
value
993 attr_T: CLK $\to$ T
axiom
993 $\forall$ clk:CLK •
993 **let** (t,t′) = (attr_CLK(clk);attr_CLK(clk)) **in**
993 t$\leq$t′ **end**

The ';' in an expression (a;b) shall mean that first expression a is evaluated, then expression b.

E.10.2 Urban Space Attributes

The Urban Space

994. We shall assume a notion of *the urban space*, tus:TUS, from which we can observe the attribute:
995. an infinite, compact Euclidean set of points.
996. By a *point* we shall understand a further undefined atomic notion.
997. By an *area* we shall understand a concept, related to the urban space, that allows us to speak of *"a point being in an area"* and *"an area being equal to or properly within another area"*.
998. To an[y] *urban space* we can associate an area; we may think of an area being an *attribute* of the urban space.

type
994 TUS
995 PtS = Pt-**infset**sac-11
value
994 attr_PtS: TUS → Pt-**infset**
type
996 Ptsac00
997 Areasac10
value
998 attr_Area: TUS → Area
997 is_Pt_in_Area: Pt × (TUS|Area) → **Bool**
997 is_Area_within_Area: Area × (TUS|Area) → **Bool**

The Urban Space Attributes

By *urban space attributes* we shall here mean the facts by means of which we can characterize that which is subject to urban planning: the land, what is in and on it: its geodetics, its cadastra[17], its meteorology, its socio-economics, its rule of law, etc. As such we shall consider 'the urban space' to be a *part* in the sense of [2]. And we shall consider the *geodetic, cadastral, geotechnical, meteorological*, *"the law"* (i.e., *state, province, city* and *district ordinances*) and *socio-economic* properties as *attributes.*

Left: geodetic map, right: cadastral map.

Main Part and Attributes

One way of observing *the urban space* is presented: to the left, in the framed box, we **narrate** the story; to the right, in the framed box, we **formalise** it.

[17] Cadastra: A Cadastra is normally a parcel based, and up-to-date land information system containing a record of interests in land (e.g. rights, restrictions and responsibilities). It usually includes a geometric description of land parcels linked to other records describing the nature of the interests, the ownership or control of those interests, and often the value of the parcel and its improvements. See http://www.fig.net/

999. The Urban Space (TUS) has the following
   a. PointSpace attributes,
   b. Geodetic attributes,
   c. Cadastre attributes,
   d. Geotechnical attributes,
   e. Meteorological attributes,
   f. Law attributes,
   g. Socio-Economic attributes, etcetera.

tus000tus000tus000tus000tus000tus000

```
type
999   TUS, PtS, GeoD, Cada, GeoT, Met, Law, SocEco
value
999a  attr_Pts: TUS → PtS
999b  attr_GeoD: TUS → GeoD
999c  attr_Cada: TUS → Cada
999d  attr_GeoT: TUS → GeoT
999e  attr_Met: TUS → Met
999f  attr_Law: TUS → Law
999g  attr_SocEco: TUS → SocEco
```

The attr_A: P → A is the **signature** of a postulated *attribute (observer) function*. From parts of type P it **observes** attributes of type A. attr_A are postulated functions. They express that we can always observe attributes of type A of parts of type P.

Urban Space Attributes – Narratives and Formalisation

We describe attributes of the domain of urban spaces. As they are, in real life. Not as we may record them or represent them (on paper or within the computer). We can "freely" model that reality as we think it is. If we can talk about and describe it, then it is so ! For meteorological attributes it means that we describe precipitation, evaporation, humidity and atmospheric pressure as these physical phenomena "really" are: continuous over time ! Similar for all other attributes. Etcetera.

General Form of Attribute Models

1000. We choose to model the *General Form of Attributes*, such as geodetical, cadastral, geotechnical, meteorological, socio-economic, legal, etcetera, as [continuous] functions from time to maps from points or areas to the specific properties of the attributes.
1001. The points or areas of the properties maps must be in, respectively within, the area of the urban space whose attributes are being specified.

```
type
1000  GFA = T → ((Pt|Area) →m Properties)gfoam00
value
1001  wf_GFA: GFA × TUS → Bool
1001  wf_GFA(gfa,tus) ≡
1001     let area = attr_Area(tus) in
1001     ∀ t:T • t ∈ 𝒟 gfa ⇒
1001        ∀ pt:Pt • pt ∈ dom gfa(t) ⇒ is_Pt_in_Area(pt,area)
1001      ∧ ∀ ar:Area • ar ∈ dom gfa(t) ⇒ is_within_Area(ar,area)
1001     end
```

𝒟 is a hypothesized function which applies to continuous functions and yield their domain !

Geodetic Attribute[s]

1002. Geodetic attributes map points to
   a. land elevation and what kind of land it is; and (or) to
   b. normal and current water depths and what kind of water it is.
1003. Geodetic attributes also includes road nets and what kind of roads;
1004. etcetera,

```
type
1002   GeoD = T → (Pt →m ((Land|Water) × RoadNet × ...))
1002a  Land = Elevation × (Farmland|Urban|Forest|Wilderness|Meadow|Swamp|...)
1002b  Water = (NormDepth × CurrDepth) × (Spring|Creek|River|Lake|Dam|Sea|Ocean|...)
1003   RoadNet = ...
1004   ...
```

Cadastral Attribute[s]

A cadastre is a public register showing details of ownership of the real property in a district, including boundaries and tax assessments.

1005. Cadastral maps shows the boundaries and ownership of land parcels. Some cadastral maps show additional details, such as survey district names, unique identifying numbers for parcels, certificate of title numbers, positions of existing structures, section or lot numbers and their respective areas, adjoining and adjacent street names, selected boundary dimensions and references to prior maps.
1006. Etcetera.

type
1005 Cada = T → (Area $\overrightarrow{m}$ (Owner × Value × ...))
1006 ...

Geotechnical Attribute[s]

1007. Geotechnical attributes map points to
   a. top and lower layer soil etc. composition, by depth levels,
   b. ground water occurrence, by depth levels,
   c. gas, oil occurrence, by depth levels,
   d. etcetera.

type
1007 GeoT = (Pt $\overrightarrow{m}$ Composition)
1007a Composition = VerticalScaleUnit × Composite*
1007b Composite = (Soil|GroundWater|Sand|Gravel|Rock|...|Oil|Gas|...)
1007c Soil,Sand,Gravel,Rock,...,Oil,Gas,... = [chemical analysis]
1007d ...

Meteorological Attribute[s]

1008. Meteorological information records, for points (of an area) precipitation, evaporation, humidity, etc.;
   a. precipitation: the amount of rain, snow, hail, etc.; that has fallen at a given place and at the time-stamped moment[18], expressed, for example, in milimeters of water;
   b. evaporation: the amount of water evaporated (to the air);
   c. atmospheric pressure;
   d. air humidity;
   e. etcetera.

1008 Met = T → (Pt $\overrightarrow{m}$ (Precip × Evap × AtmPress × Humid × ...))
1008a Precip = MMs [milimeters]
1008b Evap = MMs [milimeters]
1008c AtmPress = MB [milibar]
1008d Humid = Percent
1008e ...

[18] – that is within a given time-unit

Socio-Economic Attribute[s]

1009. Socio-economic attributes include time-stamped area sub-attributes:
   a. income distribution;
   b. housing situation, by housing category: apt., etc.;
   c. migration (into, resp. out of the area);
   d. social welfare support, by citizen category;
   e. health status, by citizen category;
   f. etcetera.

type
1009 $\mathsf{SocEco} = \mathsf{T} \rightarrow (\mathsf{Area} \overrightarrow{m} (\mathsf{Inc}\times\mathsf{Hou}\times\mathsf{Mig}\times\mathsf{SoWe}\times\mathsf{Heal}\times...))$
1009a $\mathsf{Inc} = ...$
1009b $\mathsf{Hou} = ...$
1009c $\mathsf{Mig} = \{|\text{"in"},\text{"out"}|\} \overrightarrow{m} (\{|\text{"male"},\text{"female"}|\} \overrightarrow{m} (\mathsf{Agegroup} \times \mathsf{Skills} \times \mathsf{HealthSumm} \times ...))$
1009d $\mathsf{SoWe} = ...$
1009e $\mathsf{CommHeal} = ...$
1009f ...

Law Attribute[s]: State, Province, Region, City and District Ordinances

1010. By the law we mean any state, province, region, city, district or other ‘area’ ordinance[19].
1011. ...

type
1010 Law
value
1010 $\mathsf{attr\_Law}: \mathsf{TUS} \rightarrow \mathsf{Law}$
type
1010 $\mathsf{Law} = \mathsf{Area} \overrightarrow{m} \mathsf{Ordinances}$
1011 ...

Industry and Business Economics

TO BE WRITTEN

Etcetera

TO BE WRITTEN

The Urban Space Attributes – A Summary

Summarising we can model the aggregate of urban space attributes as follows.

1012. Each of these attributes can be given a name.
1013. And the aggregate can be modelled as a map (i.e., a function) from names to appropriately typed attribute values.

type
1012 $\mathsf{TUS\_Attr\_Nm} = \{|\text{"pts"},\text{"ged"},\text{"cad"},\text{"get"},\text{"law"},\text{"eco"},...|\}$
1013 $\mathsf{TUSm} = \mathsf{TUS\_Attr\_Nm} \overrightarrow{m} \mathsf{TUS\_Attr}$
axiom
1013 $\forall\ \mathsf{tusm:TUSm} \bullet \forall\ \mathsf{nm:TUS\_Attr\_Nm} \bullet \mathsf{nm} \in \mathbf{dom}\ \mathsf{tusm} \Rightarrow$

[19] Ordinance: a law set forth by a governmental authority; specifically a municipal regulation: for ex.: *A city ordinance forbids construction work to start before 8 a.m.*

1013 **case** (nm,mtusm(nm)) **of**
1013 (("pts",v) → is_PtS(v), "ged",v) → is_GeoD(v), ("cad",v) → is_CaDa(v),
1013 ("get",v) → is_GeoT(v), ("law",v) → is_Law(v), ("eco",v) → is_Eco(v), ...
1013 **end**

Discussion

TO BE WRITTEN

E.10.3 Scripts

The concept of *scripts* is relevant in the context of *analysers* and *planners*.

By a *script* we shall understand the structured, almost, if not outright, formally expressed, wording of a procedure on how to proceed, one that may have legally binding power, that is, which may be contested in a court of law.

Those who *contract* urban analyses and urban plannings may wish to establish that some procedural steps are taken. Examples are: the vetting of urban space information, the formulation of requirements to what the analysis must contain, the vetting of that and its "quality", the order of procedural steps, etc. We refer to [3, 286].

A[ny] *script*, as implied above, is *"like a program"*, albeit to be "computed" by humans.

Scripts may typically be expressed in some notation that may include: graphical renditions, like that of Fig. E.2 [Page 316], that illustrate that two or more independent groups of people, are expected to perform a number of named and more-or-less loosely described actions, expressed in, for example, the technical (i.e., domain) language of urban analysis, respectively urban planning.

The design of urban analysis and of urban planning scripts is an experimental research project with fascinating prospects for further understanding *what urban analysis* and *urban planning* is.

E.10.4 Urban Analysis Attributes

1014. Each *analyser* is characterised by a script, and
1015. the set of master and/or derived planner server identifiers – meaning that their "attached" planners might be interested in its analysis results.

type
1014 A_Script = A_Script$_{anm_1}$ | A_Script$_{anm_2}$ | ... | A_Script$_{anm_n}$uaa-000uaa-000
1015 A_Mer = (MPS_UI|DPS_UI)-**set**uaa-010
value
1014 attr_A_Script: A → A_Scripts
1015 attr_A_Mer: A → A_Mer
axiom
1015 ∀ a:A•a ∈ ans ⇒ attr_A_Mer(a) ⊆ $ps_{ui}s$

E.10.5 Analysis Depository Attributes

The purpose of the *analysis depository* is to *accept*, *store* and *distribute* collections of *analyses*; it *accepts* these analysis from the analysers. it *stores* these analyses "locally"; and it *distributes* aggregates of these analyses to *plan servers*.

1016. The *analysis depository* has just one attribute, AHist. It is modeled as a map from *analyser names* to *analysis histories*.
1017. An *analysis history* is a time-ordered sequence, of time stamped analyses, most recent analyses first.

type
1016 AHist = ANm $\overrightarrow{m}$ (s_T:T × s_Anal:Anal$_{anm_i}$)*ada-000
value
1016 attr_AHist: AD → AHist
axiom
1017 ∀ ah:AHist, anm:ANm • anm ∈ **dom** ah ⇒
1017 ∀ i:**Nat** • {i,i+1}⊆**inds** ah(anm) ⇒
1017 s_T((ah(nm))[i]) > s_T((ah(nm))[i+1])

E.10.6 Master Planner Server Attributes

The *planner servers*, whether for *master planners* or *derived planners*, assemble arguments for their associated (i.e., 'paired') planners. These arguments include information *auxiliary* to other arguments, such as urban space information for the master planner, and analysis information for all planners; in addition the server also provides *requirements* that are resulting planner plans are expected to satisfy. For every iteration of the planner behaviour the pair of *auxiliary* and *requirements* information is to be renewed and the renewed pairs must somehow *"fit"* the previously issued pairs.

1018. The *programmable* attributes of the master planner server are those of aux:AUXiliaries and req:REQuirements.
1019. We postulate a predicate function, fit_mAux_mReq, which takes a pair of pairs auxiliary and requirements arguments, and yields a truth value.

type
1018 mAUX, mREQmplaser-000mplaser-000
value
1018 attr_mAUX: MPS → mAUX
1018 attr_mREQ: MPS → mREQ
1019 fit_mAUX_mReq: (mAUX×mREQ)×(mAUX×mREQ) → **Bool**
1019 fit_mAUX_mReq(arg_prev,arg_new) ≡ ...

E.10.7 Master Planner Attributes

The *master planner* has the following attributes:

1020. a *master planner script* which is a *static attribute*;
1021. an aggregate of *script "counters"*, a *programmable attribute*; the aggregate designates *pointers* in the *master script* where resumption of *master planning* is to take place in a resumed planning;
1022. a set of *names* of the *analysers* whose analyses the master planner is, or may be interested in, a *static attribute*; and
1023. a set of *identifiers* of the *derived planners* which the master planner may initiate *static attribute*.

type
1020 MP_Script mpa-000
1021 MP_Script_Ptmpa-000
1021 MP_Script_Pts = MP_Script_pt-**set**mpa-005
1022 ANms = ANm-**set**mpa-010
1023 DPUIs = DP_UI-**set**mpa-020
value
1020 attr_MP_Script: MP → MP_Script
1021 attr_Script_Pts: MP → MP_Script_Pts
1022 attr_ANms: MP → ANms
1023 attr_DPUIs: MP → DPUIs
axiom
1022 attr_ANms(mp) ⊆ ANms
1023 attr_DPNms(mp) ⊆ DNms

E.10.8 Derived Planner Server Attributes

1024. The *programmable* attributes, of the derived planner servers are those of aux:AUXiliaries and req:REQuirements, one each of an indexed set.
1025. We postulate an indexed predicate function, fit_mAux_mReq, which takes a pair of pairs auxiliary and requirements arguments, and yields a truth value.

type
1018 dAUX = dAUX_{dnm_1} | dAUX_{dnm_2} | ... | dAUX_{dnm_p}mplaser-000mplaser-000
1018 dREQ = dREQ_{dnm_1} | dREQ_{dnm_2} | ... | dREQ_{dnm_p}mplaser-000mplaser-000
value
1024 attr_dAUX_{dnm_i}: MPS_{dnm_i} → dAUX_{dnm_i}
1024 attr_dREQ_{dnm_i}: MPS_{dnm_i} → dREQ_{dnm_i}
1025 fit_dAUX_dReq_{dnm_i}_dReq_{dnm_i}: (dAUX_{dnm_i}×dREQ_{dnm_i})×(dAUX_{dnm_i}×dREQdnm_i) → **Bool**
1025 fit_dAUX_dReq_i(arg_prev_{dnm_i},arg_new_{dnm_i}) ≡ ...

E.10.9 Derived Planner Attributes

1026. a *derived planner script* which is a *static attribute*;
1027. an aggregate of *script "counters"*, a *programmable attribute*; the aggregate designates *points* in the *derived planner script* where resumption of *derived planning* is to take place in a resumed planning;
1028. a set of *identifiers* of the *analysers* whose analyses the master planner is, or may be interested in, a *static attribute*; and
1029. a set of *identifiers* of the *derived planners* which any specific derived planner may initiate, a *static attribute*.

type
1026 DP_Scriptdpa-000
1027 DP_Script_ptdpa-005
1027 DP_Script_Pts = DP_Script_pt*dpa-005
1028 ANmsdpa-010
1029 DNmsdpa-020
value
1026 attr_MP_Script: MP → MP_Script
1027 attr_Script_Pts: MP → Script_Pts
1028 attr_ANms: MP → ANms
1029 attr_DNms: MP → DNms
axiom
1028 attr_AUIs(mp) ⊆ ANms
1029 attr_DPUIs(mp) ⊆ DNms

E.10.10 Derived Planner Index Generator Attributes

The *derived planner index generator* has two attributes:

1030. the set of all derived planner identifiers (a static attribute), and
1031. a set of already used planner identifiers (a programmable attribute).

type
1030 All_DPUIs = DP_UI-**set**dpiga-000
1031 Used_DPUIs = DP_UI-**set**dpiga-010
value
1030 attr_All_DPUIs: DPXG →
All_DPUIs
1031 attr_Used_DPUIs: DPXG →
Used_DPUIs
axiom
1030 attr_All_DPUIs($dpxg$) = $dp_{ui}s$
1031 attr_Used_DPUIs($dpxg$) ⊆ $dp_{ui}s$

E.10.11 Plan Repository Attributes

The rôle of the *plan repository* is to keep a record of all master and derived plans. There are two plan repository attributes.

1032. A bijective map between derived planner identifiers and names, and
1033. a pair of a list of time-stamped master plans and a map from derived planner names to lists of time-stamped plans, where the lists are sorted in time order, most recent time first.

```
type
1032  NmUIm = DNm →m DP_UIpra-000
1033  PLANS = ((MP_UI|DP_UI) →m(s_t:T×s_pla:PLA)*)pra-010
value
1032  attr_NmUIm: PR → NmUIm
axiom
1032  ∀ bm:NmUIm • bm⁻¹(bm) ≡ λx.x
value
1032  attr_PLANS: PR → PLANS
axiom
1033  let plans = attr_PLANS(pr) in
1033  dom plans ⊆ {mp_ui}∪dp_uis
1033  ∀ pui:(MP_UI|DP_UI)•pui ∈ {mp_ui}∪dp_uis ⇒ time_ordered(plans(pui))
1033  end
value
1033  time_ordered: (s_t:T×s_pla:PLA)* → Bool
1033  time_ordered(tsl) ≡ ∀ i:Nat•{i,i+1}⊆inds tsl ⇒ s_t(sl(i)) > s_t(tsl(i+1))
```

E.10.12 A System Property of Derived Planner Identifiers

Let there be given the set of derived planners dps.

1034. The function reachable identifiers is the one that calculates all derived planner identifiers reachable from a given such identifier, dp_ui:DP_UI, in dps.
    a. We calculate the derived planner, dp:DP, from dp_ui.
    b. We postulate a set of unique identifiers, uis, initialised with those that can are in the attr_DPUIs(dp) attribute.
    c. Then we recursively calculate the derived planner identifiers that can be reached from any identifier, ui, in uis.
    d. The recursion reaches a fix-point when there are no more identifiers "added" to uis in an iteration of the recursion.
1035. A derived planner must not "circularly" refer to itself.

```
value
1034   reachable_identifiers: DP-set × DP_UI → DP_UI-set
1034   (dps)(dp_ui) ≡
1034a      let dp = c_p(dps)(dp_ui) in
1034b      let uis = attr_DPUIs(dp) ∪
1034c              {ui|ui:DP_UI•ui ∈ uis ∧ ui ∈ reachable_identifiers(dps)(ui)}
1034d      in uis end end

1035   ∀ ui:DP_UI • ui ∈ dp_uis ⇒ ui ∉ names(dps)(ui)
```

The seeming "endless recursion" ends when an iteration of the dns construction and its next does not produce new names for dns — a least fix-point has been reached.

E.11 PERDURANTS

By a *perdurant* we shall an entity for which only a fragment exists if we look at or touch them at any given snapshot in time, that is, were we to freeze time we would only see or touch a fragment of the perdurant

This is the second major part of this case study. The first major part is Part E.6. In a number of subsections we shall cover

- Sect. E.12: the recursive *definition*
  - of the *compilation* of *structures*, and *composite parts*
  - into *translator* invocations;
- Sect. E.13: the *declaration* of *channels*; and
- Sect. E.14: the *definition* of the *translation* of *atomic parts* into
  - *behaviour signatures* and
  - *behaviour definition bodies*.

We observe that the term *train* can have the following "meanings": the *train*, as an *endurant*, parked at the railway station platform, i.e., as a *composite part;* the *train*, as a *perdurant*, as it "speeds" down the railway track, i.e., as a *behaviour;* the *train*, as an *attribute*, say in a time-table.

This observation motivates that we "magically", as it were, introduce a COMPILEr function, cf. [2, Sect. 4] We shall refer to this "magic" as a **transcendental interpretation**[20] of *parts* as *behaviours*.

E.12 The Structure COMPILERs

E.12.1 A UNIVERSE OF DISCOURSE COMPILER

In this section, i.e., all of Sect. E.12.1, we omit complete typing of behaviours.

1036. The universe of discourse, *uod*, COMPILEs and TRANSLATEs into the of its four elements:
   a. the translation of the atomic clock, see Item E.14.1 [Page 346],
   b. the translation of the atomic urban space, see Item E.14.2 [Page 347],
   c. the compilation of the analyser structure, see Item E.12.2 [Page 342],
   d. the compilation of planner structure. see Item E.12.3 [Page 342],

value
1036 COMPILE_UoD(*uod*) ≡
1036a TRANSLATE_CLK(*clk*),
1036b TRANSLATE_TUS(*tus*),
1036c COMPILE_AA(obs_AA(*uod*)),
1036d COMPILE_PA(obs_PA(*uod*))

The COMPILER apply to, as here, *structures*, or composite parts. The TRANSLATOR apply to atomic parts. In this section, i.e., Sect. E.12.1, we will explain the obvious meaning of these functions: we will not formalise their type, and we will make some obvious short-cuts.

[20] By *transcendental* we shall mean:

- (1) BEYOND THE CONTINGENT AND ACCIDENTAL IN HUMAN EXPERIENCE, BUT NOT BEYOND ALL HUMAN KNOWLEDGE,
- (2) PERTAINING TO, BASED UPON, OR CONCERNED WITH A PRIORI ELEMENTS IN EXPERIENCE, WHICH CONDITION HUMAN KNOWLEDGE [(2–3) http://www.dictionary.com/browse/transcendental],
- (3) OF OR RELATING TO EXPERIENCE AS DETERMINED BY THE MIND'S MAKEUP [(3) Merriam Webster, https://www.merriam-webster.com/dictionary/transcendental]

E.12.2 The Analyser Structure Compiler

1037. Compiling the analyser structure results in an RSL-**Text** which expresses the separate
   a. translation of each of its n analysers, see Item E.14.3 [Page 349], and
   b. the translation of the analysis depository, see Item E.14.4 [Page 350].

1037 COMPILE_AA(aa) ≡
1037a { TRANSLATE_A$_{anm_i}$(obs_A$_{anm_i}$(aa)) | i:[1..n] },
1037b TRANSLATE_AD(obs_AD(aa))

E.12.3 The Planner Structure Compiler

1038. The *planner structure*, pa:PA, compiles into four elements:
   a. the compilation of the *master planner structure*, see Item E.12.3 [Page 342],
   b. the translation of the *derived server index generator*, see Item E.14.5 [Page 351],
   c. the translation of the *plan repository*, see Item E.14.6 [Page 352], and
   d. the compilation of the *derived server structure*, see Item E.12.3 [Page 342].

1038 COMPILE_PA(pa) ≡
1038a COMPILE_MPA(obs_MPA(pa)),
1038b TRANSLATE_DPXG(obs_DPXG(pa)),
1038c TRANSLATE_PR(obs_PR(pa)),
1038d COMPILE_DPA(obs_DPA(pa))

The Master Planner Structure Compiler

1039. Compiling the *master planner structure* results in an RSL-**Text** which expresses the separate translations of the
   a. the atomic *master planner server*, see Item E.14.7 [Page 353] and
   b. the atomic *master planner*, see Item E.14.8 [Page 354].

1039 COMPILE_MPA(mpa) ≡
1039a TRANSLATE_MPS(obs_MPS(mpa)),
1039b TRANSLATE_MP(obs_MP(mpa))

The Derived Planner Structure Compiler

1040. The compilation of the *derived planner structure* results in some RSL-**Text** which expresses the set of separate compilations of each of the *derived planner pair structures*, see Item E.12.3 [Page 342].

1040 COMPILE_DPA(dpa) ≡ { COMPILE(obs_DPC$_{nm_j}$(pa)) | j:[1..p] }

The Derived Planner Pair Structure Compiler

1041. The compilation of the derived planner pair structure results in some RSL-**Text** which expresses
   a. the results of translating the *derived planner server*, see Item E.14.9 [Page 357] and
   b. the results of translating the *derived planner*, see Item E.14.10 [Page 358].

1041 COMPILE_DPC$_{nm_j}$(dpc$_{nm_j}$), i:[1..p] ≡
1041a TRANSLATE_DPS$_{nm_j}$(obs_DPS$_{nm_j}$(dpc$_{nm_j}$)),
1041b TRANSLATE_DP$_{nm_j}$(obs_DP$_{nm_j}$(dpc$_{nm_j}$))

E.13 Channel Analysis and Channel Declarations

The *transcendental interpretation* of *parts* as *behaviours* implies *existence* of *means of communication & synchronisation* of between and of these behaviours. We refer to Fig. E.7 [Page 343] for a summary of the channels of the urban space analysis and urban planning system.

MORE TO COME

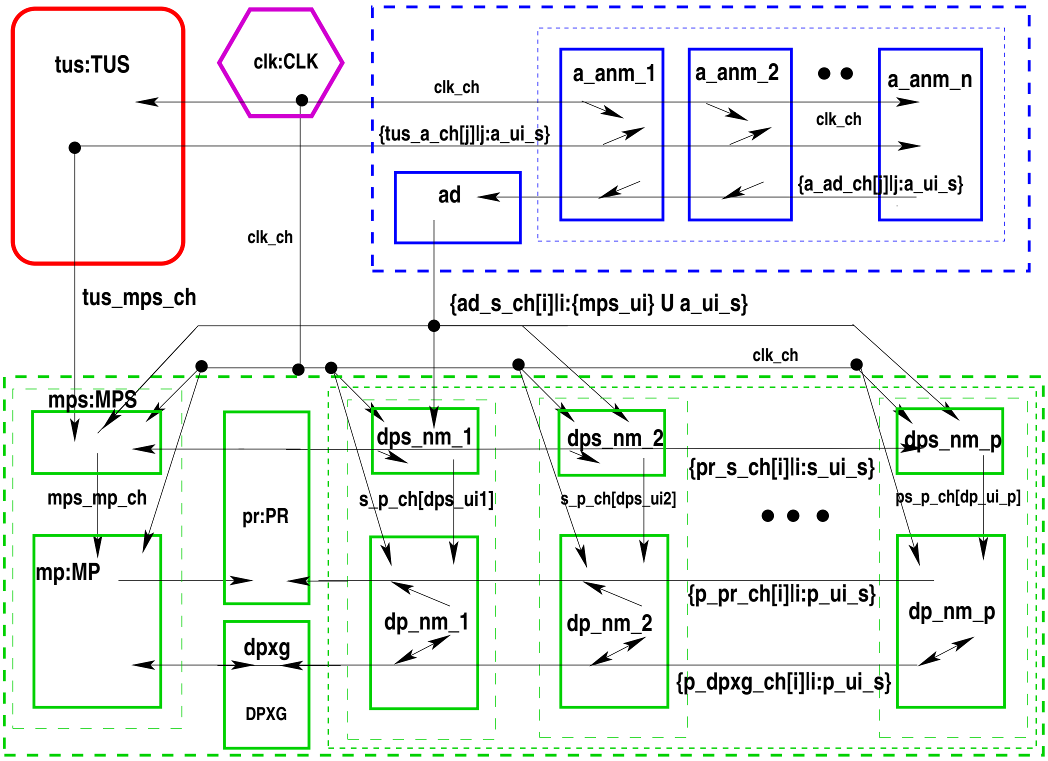


Fig. E.7. The Urban Space and Analysis Channels and Behaviours

E.13.1 The clk_ch Channel

The purpose of the clk_ch channel is, for the clock, to propagate Time to such entities who inquire. We refer to Sects. E.9.1 [Page 329], E.9.2 [Page 329], E.9.3 [Page 329], E.9.5 [Page 330], E.9.6 [Page 330], E.9.7 [Page 330] and E.9.8 [Page 331] for the mereologies that help determine the indices for the clk_ch channel.

1042. There is declared a (single) channel clk_ch
1043. whose messages are of type CLK_MSG (for Time).

The clk_ch is single. There is no need for enquirers to provide their identification. The clock "freely" dispenses of "its" time.

type
1042 CLK_MSG = T
channel
1043 clk_ch:CLK_MSG

E.13.2 The tus_a_ch Channel

The purpose of the tus_a_ch channel is, for the the urban space, to propagate urban space attributes to analysers. We refer to Sects. E.9.2 and E.9.3 for the mereologies that help determine the indices for the tus_a_-ch channel.

1044. There is declared an array channel tus_a_ch whose messages are of
1045. type TUS_MSG (for a *time stamped* aggregate of *urban space attributes*, TUSm, cf. Item 1013 [Page 336]).

type
1045 TUS_MSG = T × TUSm
channel
1044 {tus_a_ch[a_ui]:TUS_MSG|a_ui:A_UI•a_ui ∈ $a_{ui}s$}

The tus_a_ch channel is to offer urban space information to all analysers. Hence it is an array channel over indices ANms, cf. Item 919 [Page 322].

E.13.3 The tus_mps_ch Channel

The purpose of the tus_mps_ch channel is, for the the urban space, to propagate urban space attributes to the master planner server. We refer to Sects. E.9.2 and E.9.5 for the mereologies that help determine the indices for the tus_mps_ch channel.

1046. There is declared a channel tus_mps_ch whose messages are of
1045 type TUS_MSG (for a *time stamped* aggregate of *urban space attributes*, TUSm, cf. Item 1013 [Page 336]).

type
1045 TUS_MSG = T × TUSm
channel
1046 tus_mps_ch:TUS_MSG

The tus_s_ch channel is to offer urban space information to just the master server. Hence it is a single channel.

E.13.4 The a_ad_ch Channel

The purpose of the a_ad_ch channel is, for analysers to propagate analysis results to the analysis depository. We refer to Sects. E.9.3 and E.9.4 for the mereologies that help determine the indices for the a_ad_ch channel.

1047. There is declared a channel a_ad_ch whose *time stamped* messages are of
1048. type A_MSG (for *analysis message*).

type
1048 A_MSG_{anm_i} = (s_T:T × s_A:Analysis_{anm_i}), i:[1:n]
1048 A_MSG = A_MSG_{anm_1}|A_MSG_{anm_2}|...|A_MSG_{anm_n}
channel
1047 {a_ad_ch[a_ui]:A_MSG|a_ui:A_UI•a_ui ∈ $a_{ui}s$}

E.13.5 The ad_s_ch Channel

The purpose of the ad_s_ch channel is, for the analysis depository to propagate histories of analysis results to the server. We refer to Sects. E.9.4, E.9.5 and E.9.7 for the mereologies that help determine the indices for the ad_s_ch channel.

1049. There is declared a channel ad_s_ch whose messages are of
1050. type AD_MSG (defined as A_Hist for a *histories of analyses*), see Item 1016 [Page 337].

type
1050 AD_MSG = A_Hist
channel
1049 {ad_s_ch[s_ui]|s_ui:(MPS_UI|DPS_UI)•s_ui $\in \{mps_{ui}\} \cup dps_{ui}s\}$:AD_MSG

The ad_s_ch channel is to offer urban space information to the *master* and *derived servers*. Hence it is an array channel.

E.13.6 The mps_mp_ch Channel

The purpose of the mps_mp_ch channel is for the master server to propagate comprehensive master planner input to the master planner. We refer to Sects. E.9.5 and E.9.6 for the mereologies that help determine the indices for the mps_mp_ch channel.

1051. There is declared a channel mps_mp_ch whose messages are of
1052. type MPS_MSG which are quadruplets of time stamped urban space information, TUS_MSG, see Item 1045 [Page 344], analysis histories, A_Hist, see Item 1050 [Page 345], *master planner auxiliary* information, mAUX, and *master plan requirements*, mREQ.

type
1052 MPS_MSG = TUS_MSG×AD_MSG×mAUX×mREQ
channel
1051 mps_mp_ch:MPS_MSG

The mps_mp_ch channel is to offer MPS_MSG information to just the *master server*. Hence it is a single channel.

E.13.7 The p_pr_ch Channel

The purpose of the p_pr_ch channel is, for master and derived planners to deposit and retrieve master and derived plans to the plan repository. We refer to Sects. E.9.6 and E.9.10 for the mereologies that help determine the indices for the p_pr_ch channel.

1053. There is declared a channel p_pr_ch whose messages are of
1054. type PLAN_MSG – for *time stamped master plans*.

type
1054 PLAN_MSG = T × PLANS
channel
1053 {p_pr_ch[p_ui]:PLAN_MSG|p_ui:(MP_UI|DP_UI)•p_ui $\in p_{ui}s\}$

The p_pr_ch channel is to offer comprehensive records of all current plans to all the the *planners*. Hence it is an array channel.

E.13.8 The p_dpxg_ch Channel

The purpose of the p_dpxg_ch channel is, for planners to request and obtain derived planner index names of, respectively from the derived planner index generator. We refer to Sects. E.9.6 and E.9.9 for the mereologies that help determine the indices for the mp_dpxg_ch channel.

1055. There is declared a channel p_dpxg_ch whose messages are of
1056. type DPXG_MSG. DPXG_MSG messages are
   a. either *request* from the *planner* to the *index generator* to provide zero, one or more of an indicated set of *derived planner names*,
   b. or to accept such a (*response*) set from the *index generator*.

type
1056 DPXG_MSG = DPXG_Req | DPXG_Rsp
1056a DPXG_Req :: DNm-**set**
1056b DPXG_Rsp :: DNm-**set**
channel
1055 {p_dpxg_ch[ui]:DPXG_MSG|ui:(MP_UI|DP_UI)•ui ∈ $p_{ui}s$}

E.13.9 The pr_s_ch Channel

The purpose of the pr_s_ch channel is, for the plan repository to provide master and derived plans to the derived planner servers. We refer to Sects. E.9.10 and E.9.7 for the mereologies that help determine the indices for the pr_dps_ch channel.

1057. There is declared a channel pr_dps_ch whose messages are of
1058. type PR_MSGd, defined as PLAp, cf. Item 1033 [Page 340].

type
1058 PR_MSG = PLANS
channel
1057 {pr_s_ch[ui]:PR_MSGd|ui:(MPS_UI|DPS_UI)•ui ∈ $s_{ui}s$}

E.13.10 The dps_dp_ch Channel

The purpose of the dps_dp_ch channel is, for derived planner servers to provide input to the derived planners. We refer to Sects. E.9.7 and E.9.8 for the mereologies that help determine the indices for the dps_dp_ch channel.

1059. There is declared a channel dps_dp_ch[ui_nm_j], one for each *derived planner* pair.
1060. The channel messages are of type DPS_MSG$_{nm_j}$. These DPS_MSG$_{nm_i}$ messages are quadruplets of *analysis* aggregates, AD_MSG, *urban plan* aggregates, PLANS, *derived planner auxiliary information*, dAUX$_{nm_j}$, and *derived plan requirements*, dAUX$_{nm_j}$.

type
1060 DPS_MSG$_{nm_j}$ = AD_MSG×PLANS×dAUX$_{nm_j}$×dREQ$_{nm_j}$, j:[1..p]
channel
1059 {dps_dp_ch[ui]:DPS_MSG$_{nm_j}$|ui:DPS_UI•ui ∈ $dps_{ui}s$}

E.14 The Atomic Part TRANSLATORS

E.14.1 The CLOCK TRANSLATOR

We refer to Sect. E.10.1 for the attributes that play a rôle in determining the clock signature.

The TRANSLATE_CLK Function

1061. The TRANSLATE_CLK(clk) results in three text elements:
    a. the **value** keyword,
    b. the *signature* of the clock definition,
    c. and the *body* of that definition.

The clock signature contains the *unique identifier* of the clock; the *mereology* of the clock, cf. Item E.9.1 [Page 329]; and the *attributes* of the clock, in some form or another: the programmable time attribute and the channel over which the clock offers the time.

```
value
1061   TRANSLATE_CLK(clk) ≡
1061a      " value
1061b          clock: T → out clk_ch  Unit
1061c          clock(uid_CLK(clk),mereo_CLK(clk))(attr_T(clk)) ≡ ... "
```

The clock Behaviour

The purpose of the clock is to show the time. The “players” that need to know the time are: the urban space when informing requestors of aggregates of urban space attributes, the analysers when submitting analyses to the analysis depository, the planners when submitting plans to the plan repository.

1062. We see the clock as a behaviour.
1063. It takes a programmable input, the *current* time, t.
1064. It repeatedly emits the some *next* time on channel clk_ch.
1065. Each iteration of the clock it non-deterministically, internally increments the *current* time by either nothing or an infinitisimally small time interval δ ti, cf. Item 987 [Page 332].
1066. In each iteration of the clock it either offers this *next* time, or skips doing so;
1067. whereupon the clock resumes being the clock albeit with the new, i.e., *next* time.

```
value
1064   δti:TI = ... cf. Item 987 [Page 332]
1062   clock: T → out clk_ch  Unit
1063   clock(uid_clk,mereo_clk)(t) ≡
1065       let t′ = (t+δti) ⌈⌉ t in
1066       skip ⌈⌉ clk_ch!t′ ;
1067       clock(uid_clk,mereo_clk)(t′) end
1067   pre: uid_clk = clk_ui ∧
1067        mereo_clk = (tus_ui,a_ui s,mps_ui,mp_ui,dps_ui s,dp_ui s)
```

E.14.2 The URBAN SPACE TRANSLATOR

We refer to Sect. E.10.2 for the attributes that play a rôle in determining the urban space signature.

The TRANSLATE_TUS Function

1068. The TRANSLATE_TUS(tus) results in three text elements:
    a. the **value** keyword
    b. the *signature* of the urb_spa definition,
    c. and the *body* of that definition.

    The urban space signature contains the *unique identifier* of the urban space, the *mereology* of the urban space, cf. Item E.9.2 [Page 329], the static point space *attribute*.

value
1068 TRANSLATE_TUS(tus) ≡
1068a " **value**
1068b urb_spa: TUS_UI × TUS_Mer → Pts →
1068b **out** ... **Unit**
1068c urb_spa(uid_TUS(tus),mereo_TUS(tus))(attr_Pts(tus)) ≡ ... "

We shall detail the urb_spa signature and the urb_spa body next.

The urb_spa Behaviour

The urban space can be seen as a behaviour. It is "visualized" as the rounded edge box to the left in Fig. E.8 [Page 348]. It is a "prefix" of Fig. E.2 [Page 316]. In this section we shall refer to many other elements of our evolving specification. To grasp the seeming complexity of the urban space, its analyses and its urban planning functions, we refer to Fig. E.2 [Page 316].

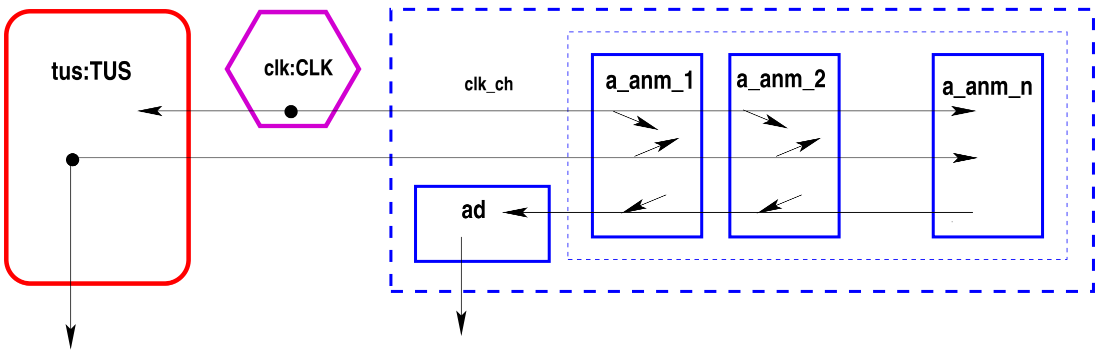


Fig. E.8. The Urban Space and Analysis Behaviours

1069. To every observable part, like tus:TUS, there corresponds a behaviour, in this case, the urb_spa.
1070. The urb_spa behaviour has, for this report, just one static attribute, the point space, Pts.
1071. The urb_spa behaviour has the following biddable and programmable attributes, the Cadastral, the Law and the SocioEconomic attributes. The biddable and programmable attributes "translate" into behaviour parameters.
1072. The urb_spa behaviour has the following dynamic, non-biddable, non-programmable attributes, the GeoDetic, GeoTechnic and the Meterological attributes The non-biddable, non-programmable dynamic attributes "translate", in the conversion from parts to behaviours, to input channels etc.

the_urb_spa behaviour offers its attributes, upon demand,

1073. to a urban space analysis behaviours, tus_ana_i and one master urban server.
1074. The urb_spa otherwise behaves as follows:
    a. it repeatedly "assembles" a tuple, tus, of all attributes;
    b. then it external non-deterministically either offers the tus tuple
    c. to either any of the urban space analysis behaviours,
    d. or to the master urban planning behaviour;
    e. in these cases it resumes being the urb_spa behaviour;
    f. or internal-non-deterministically chooses to
    g. update the law, the cadastral, and the socio-economic attributes;
    h. whereupon it resumes being the urb_spa behaviour.

```
channel
1072  attr_Pts_ch:Pts, attr_GeoD_ch:GeoD, attr_GeoT_ch:GeoT, attr_Met_ch:Met
1073  tus_mps_ch:TUSm
1073  {tus_a_ch[ai]|ai ∈ a_uis}:TUSm
value
1069  urb_spa: TUS_UI × TUS_Mer →
1070      Pts →
1071          (Cada×Law×Soc_Eco×...) →
1072              in  attr_Pts_ch, attr_GeoD_ch, attr_GeoT_ch, attr_Met_ch →
1073              out tus_mps_ch, {tus_ana_ch[ai]|ai ∈ [a_1...a_a]} → Unit
1074  urb_spa(pts)(pro) ≡
1074a     let geo = ["pts"↦attr_Pts_ch?"ged"↦attr_GeoD_ch?,"cad"↦cada,"get"↦attr_geoT_ch?,
1074a                "met"↦attr_Met_ch?,"law"↦law,"eco"↦eco,...] in
1074c     ((⨅ {tus_a_ch[ai]!geo|ai ∈ a_uis}
1074b       ⨅
1074d       tus_mps_ch!geo) ;
1074e     urb_spa(pts)(pro)) end
1074f     ⨅
1074g     let pro′:(Cada×Law×Soc_Eco×...)•fit_pro(pro,pro′) in
1074h     urb_spa(pts)(pro′) end

1074g  fit_pro: (Cada×Law×Soc_Eco×...) × (Cada×Law×Soc_Eco×...) → Bool
```

We leave the *fitness predicate* fit_pro further undefined. It is intended to ensure that the biddable and programmable attributes evolve in a commensurate manner.

E.14.3 The ANALYSER$_{anm_i}$, i:$[1:n]$ TRANSLATOR

We refer to Sect. E.10.4 for the attributes that play a rôle in determining the analyser signature.

The TRANSLATE_A$_{anm_j}$ Function

1075. The TRANSLATE_A$_{anm_j}$(a_{anm_j}) results in three text elements:
   a. the **value** keyword,
   b. the *signature* of the analysera_{anm_j} definition,
   c. and the *body* of that definition.

The analyser$_{anm_j}$ signature contains the *unique identifier* of the analyser, the *mereology* of the analyser, cf. Item E.9.3 [Page 329], and the *attributes*, here just the programmable attribute of the most recent analysisa_{anm_j} performed by the analyser$_{anm_j}$.

```
type
1075  Analysis = Analysis_nm1|Analysis_nm2|...|Analysis_nmn
value
1075  TRANSLATE_A_nmi(a_nmi):
1075    "  value
1075          analyser_nmi: (uid_A×mereo_A) →
1075              Analysis_nmi →
1075              in   tus_a_ch[uid_A(a_nmi)]
1075              out  a_ad_ch[uid_A(a_nmi)]
1075          analyser_uij(uid_A(a_nmi),mereo_A(a_nmi))(ana_nmi) ≡ ...  "
```

The analyser$_{ui_j}$ Behaviour

Analyses, or various kinds, of the urban space, is an important prerequisite for urban planning. We therefore introduce a number, n, of urban space analysis behaviours, analysis$_{anm_i}$ (for anm_i in the set $\{anm_1,...,anm_a\}$. The indexing designates that each analysis$_{anm_i}$ caters for a distinct kind of urban space analysis, each analysis with respect to, i.e., across existing urban areas: ..., (a_i) traffic statistics, (a_j) income distribution, ..., (a_k) health statistics, (a_ℓ) power consumption, ..., (a_a) ... . We shall model, by an indexed set of behaviours, ana$_i$, the urban [space] analyses that are an indispensable prerequisite for urban planning.

1076. Urban [space] analyser, tus_ana$_i$, for $a_i \in [a_1...a_a]$, performs analysis of an urban space whose attributes, except for its point set, it obtains from that urban space – via channel tus_ana_ch and
1077. offers analysis results to the mp_beh and the n derived behaviours.
1078. Urban analyser, ana$_{a_i}$, otherwise behaves as follows:
   a. The analyser obtains, from the urban space, its most recent set of attributes.
   b. The analyser then proceeds to perform the specific analysis as "determined" by its index a_i.
   c. The result, tus_ana$_{a_i}$, is communicated whichever urban, the master or the derived, planning behaviour inquires.
   d. Whereupon the analyser resumes being the analyser, improving and/or extending its analysis.

```
type
1075   Analysis = Analysis_anm1|Analysis_anm2|...|Analysis_anmn
value
1078   analyser_nmi(a_ui,a_mer)(analysis_nmi) ≡
1078a       let tusm = tus_a_ch[a_ui] ? in
1078b       let analysis′_nmi = perform_analysis_nmi(tusm)(analysis) in
1078c       ⌈⌉⌊⌋ a_ad_ch[a_ui] ! (clk_ck?,analysis′_nmi) ;
1078d       analyser_i(a_ui,a_mer)(analysis′_nmi) end end

1078b   perform_analysis_anm_i: TUSm → Analysis_anm_i → Analysis_anm_i
1078b   perform_analysis_anm_i(tusm)(analysis_anm_i) ≡ ...
```

E.14.4 The Analysis Depository Translator

We refer to Sect. E.10.5 for the attributes that play a rôle in determining the analysis depository signature.

The Translate_AD Function

1079. The Translate_AD(ad) results in three text elements:
   a. the **value** keyword
   b. the *signature* of the ana_dep definition,
   c. and the *body* of that definition.

The ana_dep signature essentially contains the *unique identifier* of the analyser, the *mereology* of the analyser, cf. Item E.9.4 [Page 330], and the *attributes*, in one form or another: the programmable attribute, a_hist, see Item 1016 [Page 337], the channels over which ana_dep either accepts time stamped *analyses*, Analysis$_{a_{u_i}}$, from *analyser*$_{anm_i}$, or offers a_hists to either the *master planner server* or the *derived planner servers*.

```
value
1079   Translate_AD(ad) ≡
1079a    " value
1079b         ana_dep: (A_UI × A_Mer) → AHist →
1079b            in {a_ad_ch[i]||i:A_UI•i ∈ a_ui s}
1079b            out {ad_s_ch[i]||i:A_UI•i ∈ s_ui s} Unit
1079c         ana_dep(ui_A(ad),mereo_A(ad))(attr_AHist(ad)) ≡ ...   "
```

The ana_dep Behaviour

The definition of the *analysis depository* is as follows.

1080. The behaviour of ana_dep is as follows: non-deterministically, externally (⌈⌉⌊⌋), ana_dep
1081. either (⌈⌉⌊⌋, line 1083) offers to accept a time stamped analysis *from some* analyser (⌈⌉⌊⌋{ ... | ... }),
   a. receiving such an analyses it "updates" its history,
   b. and resumes being the ana_dep behaviour with that updated history;
1082. or offers the analysis history *to the* master planner server
   and resumes being the ana_dep behaviour;
1083. or offers the analysis history
   a. *to whichever* (⌈⌉⌊⌋{ ... | ... }) planner server offers to accept a history
   b. and resumes being the ana_dep behaviour with that updated history.

```
value
1080   ana_dep(a_ui,a_mer)(ahist) ≡
1081     ⌈⌉⌊⌋ { (let ana = a_ad_ch[i] ? in
1081a         let ahist′ = ahist†[i↦⟨ana⟩⁀(ahist(i))] in
1081b         ana_dep(a_ui,a_mer)(ahist′) end end)
1081b       | i:A_UI•i∈ a_uis }
1082     ⌈⌉⌊⌋ (ad_mps_ch!ahist ; ana_dep(a_ui,a_mer)(ahist))
1083     ⌈⌉⌊⌋
1083a       ({ ad_s_ch[j]!ahist
1083a        | j:(MPS_UI|DPS_UI)•j∈s_uis};
1083b         ana_dep(a_ui,a_mer)(ahist))
```

E.14.5 The DERIVED PLANNER INDEX GENERATOR TRANSLATOR

We refer to Sect. E.10.10 for the attributes that play a rôle in determining the derived planner index generator signature.

The TRANSLATE_DPXG($dpxg$) Function

1084. The TRANSLATE_DPXG($dpxg$) results in three text elements:
   a. the **value** keyword
   b. the *signature* of the dpxg behaviour definition,
   c. and the *body* of that definition.

The signature of the dpxg behaviour definition has many elements: the *unique identifier* of the dpxg behaviour, the *mereology* of the dpxg behaviour, cf. Item E.9.9 [Page 331], and the *attributes* in some form or another:the *unique identifier*, the *mereology*, and the *attributes*, in some form or another: the programmable attribute All_DPUIs, cf. Item 1030 [Page 339], the programmable attribute Used_DPUIs, cf. Item 1031 [Page 339], the mp_dpxg_ch input/output channel, and the dp_dpxg_ch input/output array channel.

```
value
1084   TRANSLATE_DPXG(dpxg) ≡
1084a    " value
1084b         dpxg_beh: (DPXG_UI×DPXG_Mer) →
1084b             (All_DPUIs×UsedDPUIS) →
1084b             in,out {p_dpxg_ch[i]|i:(MP_UI|DP_UI)•i∈p_uis} Unit
1084c         dpxg_beh(uid_DPXG(dpxg),mereo_DPXG(dpxg))(all_dpuis,used_dpuis) ≡ ... "
```

The dpxg Behaviour

1085. The index generator otherwise behaves as follows:
   a. It non-deterministically, externally, offers to accept requests from any planner, whether master or server. The request suggests the names, req, of some derived planners.
   b. The index generator then selects a suitable subset, sel_dpuis, of these suggested derived planners from those that are yet to be started.
   c. It then offers these to the requesting planner.
   d. Finally the index generator resumes being an index generator, now with an updated used_dpuis programmable attribute.

```
value
1085   dpxg: (DPXG_UI×DPXG_Mer) → (All_DPUIs×Used_DPUIs) →
1085       in,out mp_dpxg_ch,
1085              {p_dpxg_ch[j]|j:(MP_UI|DP_UI)•j∈{p_ui s}} Unit
1085   dpxg(dpxg_ui,dpxg_mer)(all_dpuis,used_dpuis) ≡
1085a      ⌈⌉⌊⌋ { let req = p_dpxg_c[j] ? in
1085b            let sel_dpuis = all_dpuis \ used_dpuis • sel_dpuis ⊆ req_dpuis in
1085c            dp_dpxg_ch[j] ! sel_dpuis ;
1085d            dpxg(dpxg_ui,dpxg_mer)(all_dpuis,used_dpuis∪sel_dpuis) end end
1085          | j:(MP_UI|DP_UI)•j∈p_ui s }
```

E.14.6 The Plan Repository Translator

We refer to Sect. E.10.11 for the attributes that play a rôle in determining the plan repository signature.

The Translate_PR Function

1086. The Translate_PR(pr) results in three text elements:
   a. the **value** keyword,
   b. the *signature* of the plan repository definition,
   c. and the *body* of that definition.

The plan repository signature contains the *unique identifier* of the plan repository, the *mereology* of the plan repository, cf. Item E.9.10 [Page 331], and the *attributes*: the *programmable* plans, cf. 1033 [Page 340], and the *input/out channel* p_pr_ch.

```
value
1086   Translate_PR(pr) ≡
1086a       " value
1086b            plan_rep: PLANS →
1086b                 in   {p_pr_ch[i]|i:(MP_UI|DP_UI)•i∈p_ui s}
1086b                 out {s_pr_ch[i]|i:(MP_UI|DP_UI)•i∈s_ui s}  Unit
1086c            plan_rep(plans)(attr_AllDPUIs(pr),attr_UsedDPUIs(pr)) ≡ ...  "
```

The plan_rep Behaviour

1087. The plan repository behaviour is otherwise as follows:
   a. The plan repository non-deterministically, externally chooses between
      i. offering to accept time-stamped plans from a planner, p_{ui}, either the master planner or anyone of the derived planners,
      ii. from whichever planner so offers,
      iii. inserting these plans appropriately, i.e., at p_{ui}, as the new head of the list of "there",

iv. and then resuming being the plan repository behaviour appropriately updating its programmable attribute;

b. or
   i. offering to provide a full copy of its plan repository map
   ii. to whichever server requests so,
   iii. and then resuming being the plan repository behaviour.

```
value
1087      plan_rep(pr_ui,ps_uis)(plans) ≡
1087(a)i      ⨅ { let (t,plan) = p_pr_ch[i] ? in assert: i ∈ dom plans
1087(a)iii        let plans′ = plans † [i↦⟨(t,plan)⟩⁀plans(i)] in
1087(a)iv         plan_rep(pr_ui,ps_uis)(plans′) end end
1087(a)ii       | i:(MP_UI|DP_UI)•i∈p_uis }
1087b         ⨅
1087(b)i      ⨅ { s_pr_ch[i] ! plans ; assert: i ∈ dom plans
1087(b)iii        plan_rep(pr_ui,ps_uis)(plans)
1087(b)ii       | i:(MP_UI|DP_UI)•i∈p_uis }
```

E.14.7 The MASTER SERVER TRANSLATOR

We refer to Sect. E.10.6 for the attributes that play a rôle in determining the master server signature.

The TRANSLATE_MPS Function

1088. The TRANSLATE_MPS(*mps*) results in three text elements:
   a. the **value** keyword,
   b. the *signature* of the master_server definition,
   c. and the *body* of that definition.

The master_server signature contains the *unique identifier* of the master server, the *mereology* of the master server, cf. Item E.9.5 [Page 330], and the *dynamic attributes* of the master server: the most recently, previously produced *auxiliary* information, the most recently, previously produced plan *requirements* information, the clock channel, the urban space channel, the analysis depository channel, and the master planner channel.

```
value
1088    TRANSLATE_MPS(mps) ≡
1088a       " value
1088b           master_server: (mAUX×mREQ) →
1088b               in clk_ch, tus_m_ch, ad_s_ch[uid_MPS(mps)]
1088b               out mps_mp_ch Unit
1088c           master_server(uid_MPS(mps),mereo_MPS(mps))(attr_mAUX(mps),attr_mREQ(mps)) ≡ ...  "
```

The master_server Behaviour

1089. The master_server obtains time from the clock, see Item 1090c, information from the urban space, and the most recent analysis history, assembles these together with "locally produced"
   a. *auxiliary* planner information and
   b. plan *requirements*

   as input, MP_ARG, to the master planner.

1090. The master server otherwise behaves as follows:
   a. it obtains latest urban space information and latest analysis history, and

b. then produces auxiliary planning and plan requirements commensurate, i.e., fit, with the most recently, i.e., previously produced such information;
c. it then offers a time stamped compound of these kinds of information to the master planner,
d. whereupon the master server resumes being the master server, albeit with updated programmable attributes.

```
type
1089a  mAUX
1089b  mREQ
1089   mARG = (T × ((mAUX × mREQ) × (TUSm × AHist)))
value
1090   master_server(uid,mereo)(aux,req) ≡
1090a    let tusm = tus_m_ch ? , ahist = ad_s_ch[mps_ui] ? ,
1090b        maux:mAUX, mreq:mREQ • fit_AuxReq((aux,req),(maux,mreq)) in
1090c    s_p_ch[uid] ! (clk_ch?,((maux,mreq),(tusm,ahist))) ;
1090d    master_server(uid,mereo)(maux,mreq)
1090     end

1090b fitAuxReq: (mAUX×mREQ)×(mAUX×mREQ) → Bool
1090b fitAuxReq((aux,req),(maux,mreq)) ≡ ...
```

E.14.8 The MASTER PLANNER TRANSLATOR

We refer to Sect. E.10.7 for the attributes that play a rôle in determining the master planner signature.

The TRANSLATE_MP Function

1091. The TRANSLATE_MP(mp) results in three text elements:
   a. the **value** keyword,
   b. the *signature* of the master_planner definition,
   c. and the *body* of that definition.

The master_planner signature contains the *unique identifier* of the master planner, the *mereology* of the master planner, cf. Item E.9.6 [Page 330], and the *attributes* of the master planner: the script, cf. Sect. E.10.3 [Page 337] and Item 1014 [Page 337], a set of script pointers, cf. Item 1021 [Page 338], a set of analyser names, cf. Item 1022 [Page 338], a set of planner identifiers, cf. Item 1023 [Page 338], and the channels as implied by the master planner mereology.

```
value
1091  TRANSLATE_MP(mp) ≡
1091a    " value
1091b        master_planner: Mmp_ui:P_UI×MP_Mer×(Script×ANms×DPUIs) →
1091b            Script_Pts →
1091b               in      clk_ch, mps_mp_ch, ad_ps_ch[mp_ui]
1091b               out     p_pr_ch[mp_ui]
1091b               in,out  p_dpxg_ch[mp_ui]  Unit
1091c        master_planner(uid_MP(mp),mereo_MP(mp),
1091c            (attr_Script(mp),attr_ANms(mp),attr_DPUIs(mp)))(attr_Script_Ptrs(mp)) ≡ ...  "
```

The Master urban_planning Function

1092. The core of the master_planner behaviour is the master_urban_planning function.
1093. It takes as arguments: the script, a set of analyser names, a set of derived planner identifiers, a set of script pointers, and the time-stamped master planner argument, cf. Item 1089 [Page 353];
1094. and delivers, i.e., yields, a set of "remaining" derived planner identifiers, an updated set of script pointers, and a master result:M_RES, i.e., a master plan, mp:M_PLAN together with the time stamped master argument from which the plan was constructed.
1095. The master urban planning function is not defined by other than a predicate:
   a. the "remaining" derived planner identifiers is a subset of the arguments derived planner identifiers;
   b. the "resulting" master argument is the same as the input master argument, i.e., it is "carried forward";
   c. the arguments: the script, the analyser names, the derived planner identifiers, the set of script pointers, the time-stamped master planner argument, and the result plan otherwise satisfies a predicate $\mathscr{P}$(script,anms,dpuis,ptrs,marg)(mplan) expressing that the result mplan is an appropriate plan in view of the other arguments.

```
type
1094   M_PLAN
1094   M_RES = M_PLAN × DPUI-set × M_ARG
value
1093   master_urban_planning:
1093       Script × ANm-set × DP_UI-set × Script_Ptr-set × M_ARG
1094           → (DP_UI-set × Script_Ptr-set) × M_RES
1092   master_urban_planning(script,anms,dpuis,ptrs,marg)
1095a      as ((dpuis′,ptrs′),(mplan,marg′))
1095a         dpuis′ ⊆ dpuis
1095b      ∧ marg′ = marg
1095c      ∧ 𝒫(script,anms,dpuis,ptrs,marg)(mplan)
1092   𝒫: ((Script×ANM-set×DP_UI-set×Script_Ptr-set×M_ARG×MPLAN×Script_Ptr-set)
1092                  × (DP_UI-set×Script_Ptr-set×M_ARG×MPLAN)) → Bool
1092   𝒫((script,anms,dpuis,ptrs,marg,mplan,ptrs),(dpuis′,ptrs′,marg,mplan)) ≡ ...
```

The master_planner Behaviour

1096. The master_planner behaviours is otherwise as follows:
   a. The master_planner obtains, from the master server, its time stamped master argument, cf. Item 1089 [Page 353];
   b. it then invokes the master urban planning function;
   c. the time-stamped result is offered to the plan repository;
   d. if the result is OK as a final result,
   e. then the behaviour is stopped;
   f. otherwise
      i. the master planner inquires the derived planner index generator as for such derived planner identifiers which are not used;
      ii. the master planner behaviour is the resumed with the appropriately updated programmable script pointer attribute, in parallel with
      iii. the distributed parallel composition of the parallel behaviours of the derived servers
      iv. and the derived planners
      v. designated by the derived planner identifiers transcribed into (*nm_dps_ui*) derived server, respectively into (*nm_dp_ui*) derived planner names. For these transcription maps we refer to Sect. E.8.12 [Page 328], Item 971 [Page 328].

```
value
1096       master_planner(uid,mereo,(script,anms,puis))(ptrs) ≡
1096a        let (t,((maux,mreq),(tusm,ahist))) = mps_mp_ch ? in
1096b        let ((dpuis′,ptrs′),mres) = master_urban_planning(script,anms,dpuis,ptrs) in
1096c        p_pr_ch[uid] ! mres ;
1096d        if completed(mres) assert: ptrs′ = {}
1096e          then init_der_serv_planrs(uid,dpuis′)
1096f          else
1096(f)i           init_der_serv_plans(ui,dpuis)
1096(f)ii        ‖ master_planner(uid,mereo,(script,anms,puis))(ptrs′)
1096       end end end
```

The initiate derived servers and derived planners Behaviour

The init_der_serv_planrs behaviour plays a central rôle. The outcome of the urban planning functions, whether for master or derived planners, result in a possibly empty set of derived planner identifiers, dpuis. If empty then that shall mean that the planner, in the iteration, of the planner behaviour is suggesting that no derived server/derived planner pairs are initiated. If dpuis is not empty, say consists of the set $\{dp_{ui_i}, dp_{ui_j}, ..., dp_{ui_k}\}$ then the planner behaviour is suggesting that derived server/derived planner pairs whose planner element has one of these unique identifiers, be appropriately initiated.

1097. The init_der_serv_planrs behaviour takes the unique identifier, uid, of the “initiate issuing” planner and a suggested set of derived planner identifiers, dpuis.
1098. It then obtains, from the *derived planner index generator*, dpxg, a subset, dpuis′, that may be equal to dpuis.

It then proceeds with the parallel initiation of

1099. derived servers (whose names are extracted, extr_Nm, from their identifiers, cf. Item 965 [Page 328]),
1100. and planners (whose names are extracted, extr_Nm, from their identifiers, cf. Item 966 [Page 328])
1101. for every dp_ui in the set dpuis′.

However, we must first express the selection of appropriate arguments for these server and planner behaviours.

1102. The selection of the server and planner parts, making use of the identifier to part mapping *nms_dp_ui* and *nm_dp_ui*, cf. Items 971–972 [Page 328];
1103. the selection of respective identifiers,
1104. mereologies, and
1105. auxiliary and
1106. requirements attributes.

```
value
1097   init_der_serv_planrs: uid:(DP_UI|MP_UI) × DP_UI-set → in,out pr_dpxg[uid] Unit
1097   init_der_serv_planrs(uid,dpuis) ≡
1098     let dpuis′ = (pr_dpxg_ch[uid] ! dpuis ; pr_dpxg_ch[uid] ?) in
1102     ‖ { let p = c_p(dp_ui), s = c_s(nms_dp_ui(dp_ui)) in
1103         let ui_p = uid_DP(p), ui_s = uid_DPS(s),
1104             me_p = mereo_DP(p), me_s = mereo_DPS(s),
1105             aux_p = attr_sAUX(p), aux_s = attr_sAUX(s),
1106             req_p = attr_sREQ(p), req_s = attr_sREQ(s) in
1099         derived_server_extr_Nm(dp_ui)(ui_s,me_s,(aux_s,req_s)) ‖
1100         derived_planner_extr_Nm(dp_ui)(ui_p,me_p,(aux_p,req_p))
1101       | dp_ui:DP_UI•dpui ∈ dpuis′ end end }
1097     end
```

E.14.9 The DERIVED SERVER$_{nm_i}$, i:[1 : p] TRANSLATOR

We refer to Sect. E.10.8 for the attributes that play a rôle in determining the derived server signature.

The TRANSLATE_DPS$_{nm_j}$ Function

1107. The TRANSLATE_DPS(dps_{nm_j}) results in three text elements:
   a. the **value** keyword,
   b. the *signature* of the derived_server definition,
   c. and the *body* of that definition.

The derived_server$_{nm_j}$ signature of the derived server contains the *unique identifier*; the *mereology*, cf. Item E.9.7 [Page 330] – used in determining channels: the dynamic clock identifier, the analysis depository identifier, the derived planner identifier; and the *attributes* which are: the auxiliary, dAUX$_{nm_j}$ and the plan requirements, dREQ$_{nm_j}$.

value
1107 TRANSLATE_DPS(dps_{nm_j}) ≡
1107a " **value**
1107b derived_server$_{nm_j}$:
1107b DPS_UI$_{nm_j}$×DPS_Mer$_{nm_j}$ → (DAUX$_{nm_j}$×dREQ$_{nm_j}$) →
1107b **in** clk_ch, ad_s_ch[uid_DPS(dps_{nm_j})]
1107b **out** s_p_ch[uid_DPS(dps_{nm_j})] **Unit**
1107c derived_server$_{nm_j}$
1107c (uid_DPS(dps_{nm_j}),mereo_DPS(dps_{nm_j})),(attr_dAUX(dps_{nm_j}),attr_dREQ(dps_{nm_j})) ≡ ... "

The derived_server Behaviour

The derived_server is almost identical to the master server, cf. Sect. E.14.7, except that *plans* replace *urban space* information.

1108. The derived_server obtains time from the clock, see Item 1109c, , and the most recent analysis history, assembles these together with "locally produced"
   a. *auxiliary* planner information and
   b. plan *requirements*

   as input, MP_ARG, to the master planner.
1109. The master server otherwise behaves as follows:
   a. it obtains latest plans and latest analysis history, and
   b. then produces auxiliary planning and plan requirements commensurate, i.e., fit, with the most recently, i.e., previously produced such information;
   c. it then offers a time stamped compound of these kinds of information to the derived planner,
   d. whereupon the derived server resumes being the derived server, albeit with updated programmable attributes.

type
1108a dAUX$_{nm_j}$
1108b dREQ$_{nm_j}$
1108 dARG$_{nm_j}$ = (T × ((dAUX$_{nm_j}$ × dREQ$_{nm_j}$) × (PLANS × AHist)))
value
1109 derived_server$_{nm_j}$(uid,mereo)(aux,req) ≡
1109a **let** plans = ps_pr_ch[uid] ?, ahist = ad_s_ch[uid] ?,
1109b daux:dAUX, dreq:dREQ • fit_AuxReq$_{nm_j}$((aux,req),(daux,dreq)) **in**
1109c s_p_ch[uid] ! (clk_ch?,((maux,mreq),(plans,ahist))) ;
1109d derived_server$_{nm_j}$(uid,mereo)(daux,dreq)

1109 **end**

1109b $\mathsf{fitAuxReq}_{nm_j}$: $(\mathsf{dAUX}_{nm_j} \times \mathsf{dREQ}_{nm_j}) \times (\mathsf{dAUX}_{nm_j} \times \mathsf{dREQ}_{nm_j}) \rightarrow$ **Bool**
1109b $\mathsf{fitAuxReq}_{nm_j}$((aux,req),(daux,dreq)) $\equiv$ ...

You may wish to compare formula Items 1108–1109d above with those of formula Items 1089–1090d of Sect. E.14.7 [Page 353].

E.14.10 The Derived Planner$_{nm_i}$, i:[1 : p] Translator

We refer to Sect. E.10.9 for the attributes that play a rôle in determining the derived planner signature.

The Translate_DPdp_{nm_j} Function

This function is an "almost carbon copy" of the Translate_MPdp_{nm_j} function. Thus Items 1110–1110c [Page 358] are "almost the same" as Items 1091–1091c [Page 354].

1110. The Translate_DP($_{nm_j}$) results in three text elements:
   a. the **value** keyword,
   b. the *signature* of the derived_planner$_{nm_j}$ definition,
   c. and the *body* of that definition.

The derived_planner$_{nm_j}$ signature of the derived planner contains the *unique identifier*, the *mereology*, cf. Item E.9.8 [Page 331] and the *attributes:* the script, cf. Sect. E.10.3 [Page 337] and Item 1014 [Page 337], a set of script pointers, cf. Item 1027 [Page 339], a set of analyser names, cf. Item 1028 [Page 339], a set of planner identifiers, cf. Item 1029 [Page 339], and the channels as implied by the master planner mereology.

```
value
1110   TRANSLATE_DP(dp) ≡
1110a    " value
1110b       derived_planner: dp_ui:DP_UI×DP_Mer×(Script×ANms×DPUIs) → Script_Pts →
1110b             in     s_p_ch[dp_ui], clk_ch, ad_ps_ch[dp_ui]
1110b             out    p_pr_ch[dp_ui]
1110b             in,out p_dpxg_ch[dp_ui]  Unit
1110c       derived_planner(uid_DP(dp),mereo_DP(dp),
1110c          (attr_Script(dp),attr_ANms(dp),attr_DPUIs(dp)))(attr_Script_Ptrs(dp)) ≡ ...   "
```

The derived_urban_planning Function

This function is an "almost carbon copy" of the master_urban_planning function. Thus Items 1111–1114c [Page 359] are "almost the same" as Items 1092–1095c [Page 355].

1111. The core of the derived_planner behaviour is the derived_urban_planning function.
1112. It takes as arguments: the script, a set of analyser names, a set of derived planner identifiers, a set of script pointers, and the time-stamped derived planner argument, cf. Item 1089 [Page 353];
1113. and delivers, i.e., yields, a set of "remaining" derived planner identifiers, an updated set of script pointers, and a master result, M_RES, i.e., a master plan, mp:M_PLAN together with the time stamped master argument from which the plan was constructed.
1114. The master urban planning function is not defined by other that a predicate:
   a. the "remaining" derived planner identifiers is a subset of the arguments derived planner identifiers;
   b. the "resulting" master argument is the same as the input master argument, i.e., it is "carried forward";

c. the arguments: the script, the analyser names, the derived planner identifiers, the set of script pointers, the time-stamped master planner argument, and the result plan otherwise satisfies a predicate $\mathscr{P}_{dnm_i}$(script$_{dnm_i}$,anms,dpuis,ptrs,marg$_{dnm_i}$)(dplan$_{dnm_i}$) expressing that the result mplan is an appropriate plan in view of the other arguments.

type
1113 D_PLAN$_{dnm_i}$
1113 D_RES$_{dnm_i}$ = D_PLAN$_{dnm_i}$ × DP_UI-**set** × D_ARG$_{dnm_i}$
value
1112 derived_urban_planning$_{dnm_i}$:
1112 Script$_{dnm_i}$ × ANm-**set** × DP_UI-**set** × Script_Ptr-**set** × D_ARG$_{dnm_i}$
1113 → (DP_UI-**set** × Script_Ptr-**set**) × D_RES$_{dnm_i}$
1111 derived_urban_planning$_{dnm_i}$(script,anms,dpuis,ptrs,darg)
1114a **as** ((dpuis′,ptrs′),(dplan,ptrs′darg′))
1114a dpuis′ ⊆ dpuis
1114b ∧ darg′ = darg
1114c ∧ $\mathscr{P}_{dnm_i}$(script,anms,dpuis,ptrs,darg),((dpuis′,ptrs′),(dplan,ptrs′darg′))

1111 $\mathscr{P}_{dnm_i}$: ((Script$_{dnm_i}$×ANM-**set**×DP_UI-**set**×Script_Ptr-**set**×D_ARG$_{dnm_i}$)
1111 ×(DP_UI-**set**×Script$_{dnm_i}$_Ptr-**set**×D_RES$_{dnm_i}$)) → **Bool**
1111 $\mathscr{P}_{dnm_i}$((script$_{dnm_i}$,anms,dpuis,ptrs,darg$_{dnm_i}$),(dp_uis′,ptrs′,dres)) ≡ ...

The derived_planner$_{nm_j}$ Behaviour

This behaviour is an "almost carbon copy" of the derived_planner$_{nm_j}$ behaviour. Thus Items 1115–1115k [Page 359] are "almost the same" as Items 1096–1096(f)v [Page 355].

1115. The derived_planner behaviour is otherwise as follows:
   a. The derived_planner obtains, from the derived server, its time stamped master argument, cf. Item 1089 [Page 353];
   b. it then invokes the derived urban planning function;
   c. the time-stamped result is offered to the plan repository;
   d. if the result is OK as a final result,
   e. then the behaviour is stopped;
   f. otherwise
   g. the derived planner inquires the derived planner index generator as for such derived planner identifiers which are not used;
   h. the derived planner behaviour is the resumed with the appropriately updated programmable script pointer attribute, in parallel with
   i. the distributed parallel composition of the parallel behaviours of the derived servers
   j. and the derived planners
   k. designated by the derived planner identifiers transcribed into (*nm_dps_ui*) derived server, respectively into (*nm_dp_ui*) derived planner names. For these transcription maps we refer to Sect. E.8.12 [Page 328], Item 971 [Page 328].

value
1096 derived_planner$_{dnm_i}$(uid,mereo,(script$_{dnm_i}$,anms,puis))(ptrs) ≡
1096a **let** (t,((daux$_{dnm_i}$,dreq$_{dnm_i}$),(plans,ahist))) = s_p_ch[uid] ? **in**
1096b **let** ((dpuis′,ptrs′),dres$_{dnm_i}$) = derived_urban_planning$_{dnm_i}$(script$_{dnm_i}$,anms,dpuis,ptrs) **in**
1096c p_pr_ch[uid] ! dres$_{dnm_i}$;
1096d **if** completed(dres$_{dnm_i}$)
1096e **then** init_der_serv_planrs(uid,dpuis′) **assert:** ptrs′ = {}
1096f **else**
1096(f)i init_der_serv_plans(uid,dpuis′)

1096(f)ii ‖ derived_planner(uid,mereo,(script$_{dnm_i}$,anms,puis))(ptrs′)
1096 **end end end**

E.15 Initialisation of The Urban Space Analysis & Planning System

Section E.12 presents a *compiler* from *structures* and *parts* to *behaviours*. This section presents an initialisation of some of the behaviours. First we postulate a global *universe of discourse*, *uod*. Then we summarise the global values of *parts* and *part names*. This is followed by a summaries of *part qualities* – in four subsections: a summary of the global values of unique identifiers; a summary of channel declarations; the system as it is initialised; and the system of derived servers and planners as they evolve.

E.15.1 Summary of Parts and Part Names

value
922 [Page 324] *uod* : UoD
923 [Page 324] *clk* : CLK = obs_CLK(*uod*)
924 [Page 324] *tus* : TUS = obs_TUS(*uod*)
925 [Page 324] *ans* : A$_{anm_i}$**-set**, i:[1..n] = { obs_A$_{anm_i}$(aa) | aa∈(obs_AA(*uod*)), i:[1..n]}
926 [Page 324] *ad* : AD = obs_AD(obs_AA(*uod*))
927 [Page 324] *mps* : MPS = obs_MPS(obs_MPA(*uod*))
928 [Page 324] *mp* : MP = obs_MP(obs_MPA(*uod*))
929 [Page 324] *dpss* : DPS$_{nm_i}$**-set**, i:[1..p] =
929 [Page 324] { obs_DPS$_{nm_i}$(dpc$_{nm_i}$) |
929 [Page 324] dpc$_{nm_i}$:DPC$_{nm_i}$•dpc$_{nm_i}$∈obs_DPCS$_{nm_i}$(obs_DPA(*uod*)), i:[1..p] }
930 [Page 324] *dps* : DP$_{nm_i}$**-set**, i:[1..p] =
930 [Page 324] { obs_DP$_{nm_i}$(dpc$_{nm_i}$) |
930 [Page 324] dpc$_{nm_i}$:DPC$_{nm_i}$•dpc$_{nm_i}$∈obs_DPCS$_{nm_i}$(obs_DPA(*uod*)), i:[1..p] }
931 [Page 324] *dpxg* : DPXG = obs_DPXG(*uod*)
932 [Page 324] *pr* : PR = obs_PR(*uod*)
933 [Page 324] *spsps* : (DPS$_{nm_i}$×DP$_{nm_i}$)**-set**, i:[1..p] =
933 [Page 324] { (obs_DPS$_{nm_i}$(dpc$_{nm_i}$),obs_DP$_{nm_i}$(dpc$_{nm_i}$)) |
933 [Page 324] dpc$_{nm_i}$:DPC$_{nm_i}$•dpc$_{nm_i}$∈ obs_DPCS$_{nm_i}$(obs_DPA(*uod*)), i:[1..p] }

E.15.2 Summary of of Unique Identifiers

value
948 [Page 327] clk_{ui} : CLK_UI = uid_CLK(*uod*)
949 [Page 327] tus_{ui} : TUS_UI = uid_TUS(*uod*)
950 [Page 327] $a_{ui}s$: A_UI**-set** = {uid_A(a)|a:A•a ∈ *ans*}
951 [Page 327] ad_{ui} : AD_UI = uid_AD(*ad*)
952 [Page 327] mps_{ui} : MPS_UI = uid_MPS(*mps*)
953 [Page 327] mp_{ui} : MP_UI = uid_MP(*mp*)
954 [Page 327] $dps_{ui}s$: DPS_UI**-set** = {uid_DPS(dps)|dps:DPS•dps ∈ *dpss*}
955 [Page 327] $dp_{ui}s$: DP_UI**-set** = {uid_DP(dp)|dp:DP•dp ∈ *dps*}
956 [Page 327] $dpxg_{ui}$: DPXG_UI = uid_DPXG(*dpxg*)
957 [Page 327] pr_{ui} : PR_UI = uid_PR(*pr*)

957 [Page 327] $s_{ui}s$: (MPS_UI|DPS_UI)**-set** = $\{mps_{ui}\} \cup dps_{ui}s$
959 [Page 327] $p_{ui}s$: (MP_UI|DP_UI)**-set** = $\{mp_{ui}\} \cup dp_{ui}s$
960 [Page 327] *sips* : (DPS_UI×DP_UI)**-set** = {(uid_DPS(dps),uid_DP(dp))|(dps,dp):(DPS×DP)•(dps,dp)∈*sps*}
961 [Page 327] *si_pi_m* : DPS_UI $\overrightarrow{m}$ DP_UI = [uid_DPS(dps)↦uid_DP(dp)|(dps,dp):(DPS×DP)•(dps,dp)∈*sps*]
962 [Page 327] *pi_si_m* : DP_UI $\overrightarrow{m}$ DPS_UI = [uid_DP(dp)↦uid_DPS(dps)|(dps,dp):(DPS×DP)•(dps,dp)∈*sps*]

E.15.3 Summary of Channels

channel
1043 [Page 343] clk_ch:CLK_MSG
1044 [Page 344] {tus_a_ch[a_ui]:TUS_MSG|a_ui:A_UI•a_ui ∈ $a_{ui}s$}
1046 [Page 344] tus_mps_ch:TUS_MSG
1047 [Page 344] {a_ad_ch[a_ui]:A_MSG|a_ui:A_UI•a_ui ∈ $a_{ui}s$}
1049 [Page 345] {ad_s_ch[s_ui]|s_ui:(MPS_UI|DPS_UI)•s_ui ∈{mps_{ui}}∪$dps_{ui}s$}:AD_MSG
1051 [Page 345] mps_mp_ch:MPS_MSG
1053 [Page 345] {p_pr_ch[p_ui]:PLAN_MSG|p_ui:(MP_UI|DP_UI)•p_ui ∈ $p_{ui}s$}
1055 [Page 346] {p_dpxg_ch[ui]:DPXG_MSG|ui:(MP_UI|DP_UI)•ui ∈ $p_{ui}s$}
1057 [Page 346] {pr_s_ch[ui]:PR_MSGd|ui:(MPS_UI|DPS_UI)•ui ∈ $s_{ui}s$}
1059 [Page 346] {dps_dp_ch[ui]:DPS_MSG$_{nm_j}$|ui:DPS_UI•ui ∈ $dps_{ui}s$}

E.15.4 The Initial System

1068c [Page 347] urb_spa(uid_TUS(tus),mereo_TUS(tus))(attr_Pts(tus))
‖
1061c [Page 347] clock(uid_CLK(clk),mereo_CLK(clk))(attr_T(clk))
‖
1075 [Page 349] ‖ {analyser$_{ui_i}$(uid_A(a$_{ui_i}$),mereo_A(a$_{ui_i}$))(ana$_{anm_i}$) | ui_i:A_UID • ui_i ∈ $a_{ui}s$}
‖
1061c [Page 347] ana_dep(ui_A(ad),mereo_A(ad))(attr_AHist(ad))
‖
1086c [Page 352] plan_rep(plans)(attr_AllDPUIs(pr),attr_UsedDPUIs(pr))
‖
1084c [Page 351] dpxg_beh(uid_DPXG($dpxg$),mereo_DPXG($dpxg$))(all_dpuis,used_dpuis)
‖
1088c [Page 353] master_server(uid_MPS(mps),mereo_MPS(mps))(attr_mAUX(mps),attr_mREQ(mps))
‖
1091c [Page 354] master_planner(uid_MP(mp),mereo_MP(mp),
1091c [Page 354] (attr_Script(mp),attr_ANms(mp),attr_DPUIs(mp)))(attr_Script_Ptrs(mp))

E.15.5 The Derived Planner System

1107c [Page 357] { derived_server$_{dps_{nm_j}}$
1107c [Page 357] (uid_DPS(dps_{nm_j}),mereo_DPS(dps_{nm_j}))(attr_dAUX(dps_{nm_j}),attr_dREQ(dps_{nm_j}))
‖
1110c [Page 358] derived_planner(uid_DP($dpnm_j$),mereo_DP($dpnm_j$),
1110c [Page 358] (attr_Script($dpnm_j$),attr_ANms($dpnm_j$),attr_DPUIs($dpnm_j$)))
1110c [Page 358] | j:[1..p] }

E.16 Further Work

E.16.1 Reasoning About Deadlock, Starvation, Live-lock and Liveness

The current author is quite unhappy about the way in which he has defined the urban planning, oracle and repository behaviours. Such issues as which invariants are maintained across behaviours are not addressed. In fact, it seems to be good practice, following Dijkstra, Lamport and others, to formulate appropriate such invariants and only then "derive" behaviour definitions accordingly. In a rewrite of this research note, if ever, into a proper paper, the current author hopes to follow proper practices. He hopes to find younger talent to co-author this effort.

E.16.2 Document Handling

I may appear odd to the reader that I now turn to document handling. One central aspect of urban planning, strange, perhaps, to the reader, is that of handling the "zillions upon zillions" of documents that enter into and accrue from urban planning. If handling of these documents is not done properly a true nightmare will occur. So we shall briefly examine the urban planning document situation ! From that we conclude that we must first try understand:

- **What do we mean by a document?**

Urban Planning Documents

The urban planning functions and the urban planning behaviours, including both the base and the n derived variants, rely on documents. These documents are **created**, **edited**, **read**, **copied**, and, eventually, **shredded** by urban-planners. Editing documents result in new versions of "the same" document. While a document is being **edited** or **read** we think of it as not being **accessible** to other urban-planners. If urban-planners need to read a latest version of a document while that version is subject to editing by another urban planner, copies must first be made, before editing, one for each "needy" reader. Once, editing has and readings have finished, the "reader" copies need, or can, be shredded.

A Document Handling System

In Appendix D, [29], we sketch a document handling system domain. That is, not a document handling software system, not even requirements for a document handling software system, but just a description which, in essence, models documents and urban planners' actions on documents. (The urban planners are referred to as document handlers.) The description further models two 'aggregate' notions: one of 'handler management', and one of 'document archive'. Both seem necessary in order to "sort out" the granting of document access rights (that is, permissions to perform operations on documents), and the creation and shredding of documents, and in order to avoid dead-locks in access to and handling of documents.

E.16.3 Validation and Verification (V&V)

By **validation** of a document we shall mean: the primarily informal and social process of checking that the document description meets customer expectations.

Validation serves to get the right product.

By **verification** of a document we shall mean: the primarily formal, i.e., mathematical process of checking, testing and formal proof that the model, which the document description entails, satisfies a number of properties.

Verification serves to get the product right.

By **validation of the urban planning model** of this document we shall understand the social process of explaining the model to urban planning stakeholders, to obtain their reaction, and to possibly change the model according to stakeholder objections.

By **verification of the urban planning model** of this document we shall understand the formal process, based on formalisations of the argument and result types of the description, of testing, model checking and formally proving properties of the model.

MORE TO COME

E.16.4 Urban Planning Project Management

In this research note we have focused on the urban planning project behaviours, their interactions, and their information "passing". Usually publications about urban planning: research papers, technical papers, survey papers, etcetera, focus on specific "functions". In this research note we do not. We focus instead on what we can say about the domain of urban planning: the fact, or the possibility, that an initial, a core, here referred to as a base, urban planning effort (i.e., project, hence behaviour) can "spew off", generate, a number of (derived, i.e., in some sense subsidiary), more specialised, urban planning projects.

Urban Planning Projects

We think of a comprehensive urban planning project as carried out by urban planners. As is evident from the model the project consists of one base urban planning project and up to n derived urban planning projects. The urban planners involved in these projects are professionals in the areas of planning:

- master urban planning issues:
  - geodesy,
  - geotechniques,
  - meteorology,
- master urban plans:
  - cartography,
  - cadestral matters,
  - zoning;
- derived urban planning issues:
  - industries,
  - residential and shopping,
  - apartment buildings,
  - villas,
  - recreational,
  - etcetera;
- technological infrastructures:
  - transport,
  - electricity,
  - telecommunications,
  - gas,
  - water,
  - waste,
  - etcetera;
- societal infrastructures:
  - health care,
  - schools,
  - police,
  - fire brigades,
  - etcetera;
- etcetera, etcetera, etcetera !

To anyone with any experience in getting such diverse groups and individuals of highly skilled professionals to work together it is obvious that some form of management is required. The term ‘comprehensive’ was mentioned above. It is meant to express that the comprehensive urban planning project is the only one “dealing” with a given geographic area, and that no other urban planning projects “infringe” upon it, that is, “deal” with sub-areas of that given geographic area.

Strategic, Tactical and Operational Management

We can distinguish between

- strategic,
- tactical and
- operational

management.

Project Resources

But first we need take a look at the **resources** that management is charged with:

- the urban planners, i.e., humans,
- time,
- finances,
- office space,
- support technologies: computing etc.,
- etcetera.

Strategic Management

By **strategic management** we shall understand the analysis and decisions of, and concerning, scarce resources: people (skills), time, monies: their deployment and trade-offs.

Tactical Management

By **tactical management** we shall understand the analysis and decisions with respect to budget and time plans, and the monitoring and control of serially reusable resources: office space, computing.

Operational Management

By **operational management** we shall understand the monitoring and control of the enactment, progress and completion of individual deliverables, i.e., documents, the quality (adherence to “standards”, fulfillment of expectations, etc.) of these documents, and the day-to-day human relations.

Urban Planning Management

The above (*strategic, tactical & operational management*) translates, in the context of *urban planning*, into:

TO BE WRITTEN

E.17 Conclusion

TO BE WRITTEN

E.17.1 What Were Our Expectations ?

E.17.2 What Have We Achieved ?

TO BE WRITTEN

E.17.3 What Next ?

TO BE WRITTEN

E.17.4 Acknowledgement

TO BE WRITTEN

F

Swarms of Drones

We speculate on a domain of *swarms* and *drones monitored* and *controlled* by a *command center* in some *geography*. Awareness of swarms is registered only in an enterprise command center. We think of these swarms of drones as an enterprise of either package deliverers, crop-dusters, insect sprayers, search & rescuers, traffic monitors, or wildfire fighters – or several of these, united in a notion of *an enterprise* possibly consisting of of "disjoint" *businesses*. We analyse & describe the properties of these phenomena as enduratns and as perdurants: parts one can observe and behaviours that one can study. We do not yet examine the problem of drone air traffic management[1]. The analysis & description of this postulated domain follows the principles, techniques and tools laid down in [2].

F.1 An Informal Introduction

F.1.1 Describable Entities

The Endurants: Parts

In the universe of discourse we observe *endurants*, here in the form of parts, and *perdurants*, here in the form of behaviours.

The parts are *discrete endurants*, that is, can be seen or touched by humans, or that can be conceived as an abstraction of a discrete part.

We refer to Fig. F.1 [Page 366].

There is a *universe of discourse*, uod:UoD. The universe of discourse embodies: an *enterprise*, e:E. The enterprise consists of an *aggregate of enterprise drones*, aed:AED (which consists of a set, eds:EDs, of enterprise drones). and a *command center*, cc:CC; The universe of discourse also embodies a *geography*, g:G. The universe of discourse finally embodies an *aggregate of 'other' drones*, aod:AOD (which consists of a set, ods:ODs, of these 'other' drones). A *drone* is an *unmanned aerial vehicle*.[2] We distinguish between *enterprise drones*, ed:ED, and *'other' drones*, od:OD. The `pragmatics` of the enterprise swarms is that of providing enterprise drones for one or more of the following kinds of *businesses*:[3] delivering parcels (mail, packages, etc.)[4], crop dusting[5], aerial spraying[6], wildfire fighting[7], traffic control[8], search and

---

[1] https://www.nasa.gov/feature/ames/first-steps-toward-drone-traffic-management, http://www.sciencedirect.com/science/article/pii/S20460

[2] Drones are also referred to as UAVs.

[3] http://www.latimes.com/business/la-fi-drone-traffic-20170501-htmlstory.html

[4] https://www.amazon.com/Amazon-Prime-Air/b?node=8037720011 and https://www.digitaltrends.com/cool-tech/amazon-prime-air-delivery-drones-history-progress/

[5] http://www.uavcropdustersprayers.com/, http://sprayingdrone.com/

[6] https://abjdrones.com/commercial-drone-services/industry-specific-solutions/agriculture/

[7] https://www.smithsonianmag.com/videos/category/innovation/drones-are-now-being-used-to-battle-wildfires/

[8] https://business.esa.int/sites/default/files/Presentation%20on%20UAV%20Road%20Surface%20Monitoring %20and%20Traffic%20Information_0.pdf

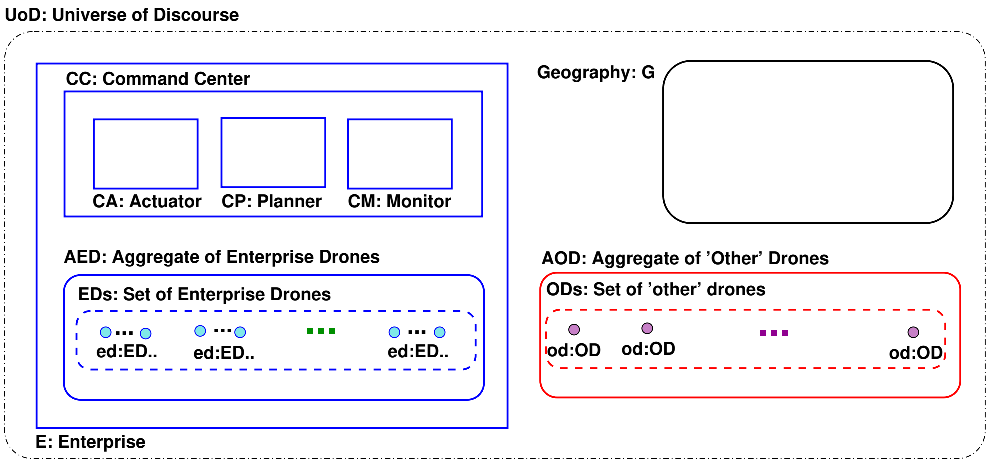


Fig. F.1. Universe of Discourse

rescue[9], etcetera. A notion of *swarm* is introduced. A swarm is a concept. As a concept a swarm is a set of drones. We associate swarms with businesses. A business has access to one or more swarms. The enterprise *command center*, cc:CC, can be seen as embodying three kinds of functions: a *monitoring* service, cm:CM, whose function it is to know the locations and dynamics of all drones, whether enterprise drones or 'other' drones; a *planning* service, cp:CP, whose function it is to plan the next moves of all that enterprise's drones; and an *actuator* service, ca:CA, whose functions it is to guide that enterprise's drones as to their next moves. The swarm concept "resides" in the command planner.

The Perdurants

The perdurants are entities for which only a fragment exists if we look at or touch them at any given snapshot in time, that is, were we to freeze time we would only see or touch a fragment of the perdurant.

The major ***

MORE TO COME

F.1.2 The Contribution of [2]

The major contributions of [2] are these: a methodology[10] for analysing & describing manifest domains[11], where the metodology builds on an *ontological principle* of viewing the domains as consisting of *endurants* and *perdurants*. Endurants possess properties such as *unique identifiers*, *mereologies*, and *attributes*. Perdurants are then analysed & described as either *actions*, *events*, or *behaviours*. The *techniques* to go with the ***

MORE TO COME

The *tools* are ***

MORE TO COME

[9] http://sardrones.org/

[10] By a *methodology* we shall understand a set of *principles* for `selecting` and `applying` a number of *techniques*, using *tools*, to – in this case – analyse & describe a domain.

[11] A manifest domain is a human- and artifact-assisted arrangement of endurant, that is spatially "stable", and perdurant, that is temporally "fleeting" entities. Endurant entities are either parts or components or materials. Perdurant entities are either actions or events or behaviours.

F.1.3 The Contribution of This Report

TO BE WRITTEN

We relate our work to that of [287].

•••

The main part of this report is contained in the next three sections: endurants; states, constants, and operations on states; and perdurants.

F.2 Entities, Endurants

By an *entity* we shall understand *a phenomenon, i.e., something that can be observe d, i.e., be seen or touched by humans, or that can be conceived as an abstraction of an entity. We further demand that an entity can be objectively described.*

By an *endurant* we shall understand *an entity that can be observed or conceived and described as a "complete thing" at no matter which given snapshot of time. Were we to "freeze" time we would still be able to observe the entire enduranr.*

F.2.1 Parts, Atomic and Composite, Sorts, Abstract and Concrete Types

By a *discrete endurant* we shall understand *an endurant which is separate, individual or distinct in form or concept.*

By a *part* we shall understand *a discrete endurant which the domain engineer chooses to endow with internal qualities such as unique identification, mereology, and one or more attributes.* We shall define the concepts of unique identifier, mereology and attribute later in this case study.

Atomic parts are *those which, in a given context, are deemed to not consist of meaningful, separately observable proper sub-parts.*

Sub-parts are parts.

Composite parts are *those which, in a given context, are deemed to indeed consist of meaningful, separately observable proper sub-parts.*

By a *sort* we shall understand *an abstract type.*

By a *type* we shall here understand *a set of values "of the same kind"* – where we do not further define what we mean by *the same kind*".

By an *abstract type* we shall understand *a type about whose values we make no assumption* [as to their atomicity or composition.

By a *concrete type* we shall understand *a type about whose values we are making certain assumptions as to their atomicity or composition, and, if composed then how and from which other types they are composed.*

Universe of Discourse

By a *universe of discourse* we shall understand *that which we can talk about, refer to and whose entities we can name.* Included in that universe is the *geography*. By *geography* we shall understand a section of the globe, an area of land, its geodecy, its meteorology, etc.

1116. In the **U**niverse **of** **D**iscourse we can observe the following parts:
   a. an atomic **G**eography,
   b. a composite **E**nterprise,
   c. and an aggregate of '**O**ther'[12] **D**rones.

[12] We apologize for our using the term 'other' drones. These 'other' drones are not necessarilt adversary or enemy drones. They are just there – coexisting with the enterprise drones.

type
1116 UoD, G, E, AOD
value
1116a obs_G: UoD → G
1116b obs_E: UoD → E
1116c obs_AOD: UoD → AOD

The Enterprise

1117. From an enterprise one can observe:
    a. a(n enterprise) command center. and
    b. an aggregate of enterprise drones.

type
1117a CC
1117a AED
value
1117a obs_CC: E → CC
1117b obs_AED: E → AED

From Abstract Sorts to Concrete Types

1118. From an *aggregate of enterprise drones*, AED, we can observe a possibly empty set of drones, EDs
1119. From an *aggregate of 'other' drones*, AOD, we can observe a possibly empty set, ODs, of *'other' drones*.

type
1118 ED
1118 EDs = ED-**set**
1119 OD
1119 ODs = OD-**set**
value
1118 obs_EDs: AED → EDs
1119 obs_ODs: AOD → ODs

Drones, whether 'other' or 'enterprise', are considered atomic.

The Auxiliary Function xtr_Ds:

We define an auxiliary function, xtr_Ds.

1120. From the universe of discourse we can extract all its drones;
1121. similarly from its enterprise;
1122. similarly from the aggregate of enterprise drones; and
1123. from an aggregate of 'other' drones.

1120 xtr_Ds: UoD → (ED|OD)-**set**
1120 xtr_Ds(uod) ≡
1120 ∪{xtr_Ds(obs_AED(obs_E(uod)))} ∪ xtr_Ds(obs_AOD(uod))
1121 xtr_Ds: E → ED-**set**
1121 xtr_Ds(e) ≡ xtr_Ds(obs_AED(e))
1122 xtr_Ds: AED → ED-**set**
1122 xtr_Ds(aed) ≡ obs_EDs(obs_EDs(aed))
1123 xtr_Ds: AOD → OD-**set**
1123 xtr_Ds(aod) ≡ obs_ODs(aod)

1124. In the universe of discourse a drone cannot be both among the enterprise drones and among the 'other' drones.

axiom
1124 ∀ uod:UoD,e:E,aed:ES,aod:AOD •
1124 e=obs_E(uod)∧aed=obs_AED(e)∧aod:obs_AOD(uod)
1124 ⇒ xtr_Ds(aed) ∩ xtr_Ds(aod) ={}

The functions are partial as the supplied swarm identifier may not be one of the universe of discourse, etc.

Command Center

Figure F.1 [Page 366] shows a graphic rendition of a space of interest. The command center, CC, a composite part, is shown to include three atomic parts: An atomic part, the monitor, CM. It monitors the location and dynamics of all drones. An atomic part, the planner, CP. It plans the next, "friendly", drone movements. The command center also has yet an atomic part, the actuator, CA. It informs "friendly" drones of their next movements. The planner is where "resides" the notion of a enterprise consisting of one or more businesses, where each business has access to zero, one or more swarms, where a swarm is a set of enterprise drone identifiers.

The purpose of the control center is to monitor the whereabouts and dynamics of all drones (done by CM); to plan possible next actions by enterprise drones (done by CP); and to instruct enterprise drones of possible next actions (done by CA).

Command Center Decomposition From the composite command center we can observe

1125. the center monitor, CM;
1126. the center planner, CP; and
1127. the center actuator, CA .

type
1125 CM
1126 CP
1127 CA

value
1125 obs_CM: CC → CM
1126 obs_CP: CC → CP
1127 obs_CA: CC → CA

F.2.2 Unique Identifiers

Parts are distinguishable through their unique identifiers. A *unique identifier* is a further undefined quantity which we associate with parts such that no two parts of a universe of discourse are identical.

The Enterprise, the Aggregates of Drones and the Geography

1128. Although we may not need it for subsequent descriptions we do, for completeness of description, introduce unique identifiers for parts and sub-parts of the universe of discourse:
  a. Geographies, g:G, have unique identification.
  b. Enterprises, e:E, have unique identification.
  c. Aggregates of enterprise drones, aed:AED, have unique identification.
  d. Aggregates of 'other' drones, aod:AOD, have unique identification.
  e. Command centers, cc:CC, have unique identification.

type
1128 GI, EI, AEDI, AODI, CCI
value
1128a uid_G: G → GI
1128b uid_E: E → EI
1128c uid_AED: AED → AEDI
1128d uid_OD: AOD → AODI
1128e uid_CC: CC → CCI

Unique Command Center Identifiers

1129. The monitor has a unique identifier.
1130. The planner has a unique identifier.
1131. The actuator has a unique identifier.

type
1129 CMI
1130 CPI
1131 CAI

value
1129 uid_CM: CM → CMI
1130 uid_CP: CP → CPI
1131 uid_CA: CA → CAI

Unique Drone Identifiers

1132. Drones have unique identifiers.
   a. whether enterprise or
   b. ‘other’ drones

type
1132 DI = EDI | ODI
value
1132a uid_ED: ED → EDI
1132b uid_OD: OD → ODI

Auxiliary Function: xtr_dis:

1133. From the aggregate of enterprise drones;
1134. From the aggregate of ‘other’ drones;
1135. and from the two parts of a universe of discourse: the enterprise and the ‘other’ drones.

value
1133 xtr_dis: AED → DI-**set**
1133 xtr_dis(aed) ≡ {uid_ED(ed)|ed:ED•ed ∈ obs_EDs(aed)}
1134 xtr_dis: AOD → DI-**set**
1134 xtr_dis(aod) ≡ {uid_D(od)|od:OD•od ∈ obs_ODs(aod)}
1135 xtr_dis: UoD → DI-**set**
1135 xtr_dis(uod) ≡ xtr_dis(obs_AED(uod)) ∪ xtr_dis(obs_AOD(uod))

Auxiliary Function: xtr_D:

1136. From the universe of discourse, given a drone identifier of that space, we can extract the identified drone;
1137. similarly from the enterprise;
1138. its aggregate of enterprise drones; and
1139. and from its aggregate of ‘other’ drones;

1136 xtr_D: UoD → DI $\stackrel{\sim}{\rightarrow}$ D
1136 xtr_D(uod)(di) ≡ **let** d:D • d ∈ xtr_Ds(uod)∧uid_D(d)=di **in** d **end**
1136 **pre**: di ∈ xtr_dis(soi)
1137 xtr_D: E → DI $\stackrel{\sim}{\rightarrow}$ D
1137 xtr_D(e)(di) ≡ **let** d:D • d ∈ xtr_Ds(obs_ES(e))∧uid_D(d)=di **in** d **end**
1137 **pre**: di ∈ xtr_dis(e)
1138 xtr_D: AED → DI $\stackrel{\sim}{\rightarrow}$ D
1138 xtr_D(aed)(di) ≡ ≡ **let** d:D • d ∈ xtr_Ds(aed)∧uid_D(d)=di **in** d **end**
1138 **pre**: di ∈ xtr_dis(es)

```
1139 xtr_D: AOD → DI ~→ D
1139 xtr_D(aod)(di) ≡ let d:D • d ∈ xtr_Ds(aod)∧uid_D(d)=di in d end
1139     pre: di ∈ xtr_dis(ds)
```

F.2.3 Mereologies

Definition

Mereology is the study and knowledge of parts and their relations (to other parts and to the "whole") [38].

Origin of the Concept of Mereology as Treated Here

We shall [thus] deploy the concept of mereology as advanced by the Polish mathematician, logician and philosopher Stanisław Léschniewski. Douglas T. ("Doug") Ross[13] also contributed along the lines of our approach [230] – hence [7] is dedicated to Doug.

Basic Mereology Principle

The basic principle in modelling the mereology of a any universe of discourse is as follows: Let p' be a part with unique identifier p'_{id}. Let p be a sub-part of p' with unique identifier p_{id}. Let the immediate sub-parts of p be $p_1, p_2, \ldots, p_n$ with unique identifiers $p_{1_{id}}, p_{2_{id}}, \ldots, p_{n_{id}}$. That p has mereology $(p'_{id}, \{p_{1_{id}}, p_{2_{id}}, \ldots, p_{n_{id}}\})$. The parts p_j, for $1 \leq j \leq n$ for $n{\geq}2$, if atomic, have mereologies $(p_{id}, \{p_{1_{id}}, p_{2_{id}}, \ldots, p_{j-1_{id}}, p_{j+1_{id}}, \ldots, p_{n_{id}}\})$ – where we refer to the second term in that pair by m; and if composite, have mereologies $(p_{id}, (m, m'))$, where the m' term is the set of unique identifiers of the sub-parts of p_j.

Engineering versus Methodical Mereology

We shall restrict ourselves to an engineering treatment of the mereology of our universe of discourse. That is in contrast to a strict, methodical treatment. In a methodical description of the mereologies of the various parts of the universe of discourse one assigns a mereology to every part: to the enterprise, the aggregate of 'other' drones and the geography; to the command center of the enterprise and its aggregate of drones; to the monitor, the planner and the actuator of the command center; to the drones of the aggregate of enterprise drones, and to the drones of the aggregate of 'other' drones. We shall "shortcut" most of these mereologies. The reason is this: The *pragmatics* of our attempt to model *drones*, is rooted in our interest in the interactions between the command center's monitor and actuator and the enterprise and 'other' drones. For "completeness" we also include interactions between the geography's meteorology and the above command center and drones. The mereologies of the enterprise, E, the enterprise aggregate of drones AED, and the set of (enterprise) drones, EDs, do not involve drone identifiers. The only "thing" that the monitor and actuator are interested in are the drone identifiers. So we shall thus model the mereologies of our universe of discourse by omitting mereologies for the enterprise, the aggregates of drones, the sets of these aggregates, and the geography, and only describe the mereologies of the monitor, planner and actuator, the enterprise drones and the 'other' drones.

[13] Doug Ross is the originator of the term CAD for *computer aided design*, of APT for *Automatically Programmed Tools*, a language to drive numerically controlled manufacturing, and also SADT for *Structure Analysis and Design Techniques*

Planner Mereology

1140. The planner mereology reflects the center planners awareness[14] of the monitor, the actuator,, and the geography of the universe of discourse.
1141. The plannner mereology further reflects that a *eureka*[15] is provided by, or from, an outside source reflected in the autonomous attribute CmdI. The value of this attribute changes at its own volition and ranges over commands that directs the planner to perform either of a number of operations.

Eureka examples are: calculate and effect a new flight plan for one or more designated swarms of a designated business; effect the transfer of an enterprise drone from a designated swarm of a business to another, distinctly designated swarm of the same business; etcetera.

```
type
1140   CPM = (CAI × CMI × GI) × Eureka
1141   Eureka == mkNewFP(BI×SI-set×Plan)
1141          | mkChgDB(fsi:SI×tsi:SI×di×DI)
1141          | ...
value
1140   mereo_CP: CP → CPM
1141   Plan = ...
```

We omit expressing a suitable axiom concerning center planner mereologies. Our behavioural analysis & description of monitoring & control of operations on the space of drones will show that command center mereologies may change.

Monitor Mereology

The monitor's mereology reflects its awareness of the drones whose position and dynamics it is expected to monitor.

1142. The mereology of the center monitor is a pair: the set of unique identifiers of the drones of the universe of discourse, and the unique identifier of the center planner.

```
type
1142   CMM = DI-set × CPI
value
1142   mereo_CM: CM → CMM
```

1143. For the universe of discourse it is the case that
   a. the drone identifiers of the mereology of a monitor must be exactly those of the drones of the universe of discourse, and
   b. the planner identifier of the mereology of a monitor must be exactly that of the planner of the universe of discourse.

```
axiom
1143    ∀ uod:UoD,e:E,cc:CC,cp:CP,cm:CM,g:G •
1143       e=obs_E(uod)∧cc=obs_CC(e)∧cp=obs_CP(cc)∧cm=obs_CM(cc) ⇒
1143          let (dis,cpi) = mereo_CM(cm) in
1143a         dis = xtr_dis(uod)
1143b      ∧ cpi = uid_CP(cp) end
```

[14] That "awareness" includes, amongst others, the planner obtaining information from the monitor of the whereabouts of all drones and providing the actuator with directives for the enterprise drones — all in the context of the *land* and "its" *meteorology*.

[15] "Eureka" comes from the Ancient Greek word εμρηκα heúrēka, meaning "I have found (it)", which is the first person singular perfect indicative active of the verb ευρηκω heuriskō "I find".[1] It is closely related to heuristic, which refers to experience-based techniques for problem solving, learning, and discovery.

Actuator Mereology

The center actuator's mereology reflects its awareness of the enterprise drones whose position and dynamics it is expected to control.

1144. The mereology of the center actuator is a pair: the set of unique identifiers of the business drones of the universe of discourse, and the unique identifier of the center planner.

```
type
1144   CAM = EDI-set × CPI
value
1144   mereo_CA: CA → CAM
```

1145. For all universes of discourse
    a. the drone identifiers of the mereology of a center actuator must be exactly those of the enterprise drones of the space of interest (of the monitor), and
    b. the center planner identifier of the mereology of a center actuator must be exactly that of the center planner of the command center of the space of interest (of the monitor)

```
axiom
1145   ∀ uod:UoD,e:E,cc:CC,cp:CP,ca:CA •
1145     e=obs_E(uod)∧cc=obs_CC(e)∧cp=obs_CP(cc)∧ca=obs_CA(cc) ⇒
1145        let (dis,cpi) = mereo_CA(ca) in
1145a       dis = tr_dis(e)
1145b     ∧ cpi = uid_CP(cp) end
```

Enterprise Drone Mereology

1146. The mereology of an enterprise drone is the triple of the command center monitor, the command center actuator[16], and the geography.

```
type
1146   EDM = CMI × CAI × GI
value
1146   mereo_ED: ED → EDM
```

1147. For all universes of discourse the enterprise drone mereology satisfies:
    a. the unique identifier of the first element of the drone mereology is that of the enterprise's command monitor,
    b. the unique identifier of the second element of the drone mereology is that of the enterprise's command actuator, and
    c. the unique identifier of the third element of the drone mereology is that of the universe of discourse's geography.

```
axiom
1147  ∀ uod:UoD,e:E,cm:CM,ca:CA,ed:ED,g:G •
1147     e=obs_E(uod)∧cm=obs_CM(obs_CC(e))∧ca=obs_CA(obs_CC(e))
1147   ∧ ed ∈ xtr_Ds(e)∧g=obs_G(uod) ⇒
1147        let (cmi,cai,gi) = mereo_D(ed) in
1147a       cmi = uid_CMM(ccm)
1147b     ∧ cai = uid_CAI(cai)
1147c     ∧ gi = uid_G(g) end
```

[16] The command center monitor and the command center actuator and their unique identifiers will be defined in Items 1125, 1127 [Page 369], 1129 and 1131 [Page 370].

‘Other’ Drone Mereology

1148. The mereology of an ‘other’ drone is a pair: the unique identifier of the monitor and the unique identifier of the geography.

type
1148 ODM = CMI × GI
value
1148 mereo_OD: OD → ODM

We leave it to the reader to formulate a suitable axiom, cf. axiom 1147 [Page 373].

Geography Mereology

1149. The geography mereology is a pair[17] of the unique of the unique identifiers of the planner and the set of all drones.

type
1149 GM = CPI × CMI × DI-**set**
value
1149 mereo_G: G → GM

We leave it to the reader to formulate a suitable axiom, cf. axiom 1147 [Page 373].

F.2.4 Attributes

We analyse & describe attributes for the following parts: *enterprise drones* and *‘other’ drones*, *monitor*, *planner* and *actuator*, and the *geography*. The attributes, that we shall arrive at, are usually concrete in the sense that they comprise values of, as we shall call them, *constituent* types. We shall therefore first analyse & describe these constituent types. Then we introduce the part attributes as expressed in terms of the constituent types. But first we introduce three notions core notions: time, Sect. F.2.4, positions, Sect. F.2.4, and flight plans, Sect. F.2.4.

The Time Sort

1150. Let the special sort identifier $\mathbb{T}$ denote times
1151. and the special sort identifier $\mathbb{TI}$ denote time intervals.
1152. Let identifier time designate a “magic” function whose invocations yield times.

type
1150 $\mathbb{T}$
1150 $\mathbb{TI}$
value
1150 time: **Unit** → $\mathbb{T}$

1153. Two times can not be added, multiplied or divided, but subtracting one time from another yields a time interval.
1154. Two times can be compared: smaller than, smaller than or equal, equal, not equal, etc.
1155. Two time intervals can be compared: smaller than, smaller than or equal, equal, not equal, etc.
1156. A time interval can be multiplied by a real number.

Etcetera.

[17] 30.11.2017: I think !

value
1153 $\ominus$: $\mathbb{T} \times \mathbb{T} \rightarrow \mathbb{TI}$
1154 $<,\leq,=,\neq,\geq,>$: $\mathbb{T} \times \mathbb{T} \rightarrow$ **Bool**
1155 $<,\leq,=,\neq,\geq,>$: $\mathbb{TI} \times \mathbb{TI} \rightarrow$ **Bool**
1156 $\otimes$: $\mathbb{TI} \times$ **Real** $\rightarrow \mathbb{TI}$

Positions

Positions (of drones) play a pivotal rôle.

1157. Each *position* being designated by
1158. *longitude, latitude* and *altitude*.

type
1158 LO, LA, AL
1157 P = LO $\times$ LA $\times$ AL

A Neighbourhood Concept

1159. Two positions are said to be *neighbours* if the *distance* between them is small enough for a drone to fly from one to the other in one to three minutes' time – for drones flying at a speed below `Mach 1`.

value
1159 neighbours: P $\times$ P $\rightarrow$ **Bool**

We leave the neighbourhood proposition further undefined.

Flight Plans

A crucial notion of our universe of discourse is that of flight plans.

1160. A *flight plan element* is a pair of a time and a position.
1161. A *flight plan* is a sequence of flight plan elements.

type
1160 FPE = $\mathbb{T} \times$ P
1161 FP = FLE*

1162. such that adjacent entries in flight plans
 a. record increasing times and
 b. neighbouring positions.

axiom
1162 $\forall$ fp:FP,i:**Nat** $\cdot$ {i,i+1}$\subseteq$**inds**fp $\Rightarrow$
1162 **let** (t,p)=fp[i], (t′,p′)=fp[i+1] **in**
1162a t $\leq$ t′
1162b $\wedge$ neighbours(p,p′)
1162 **end**

Enterprise Drone Attributes

Constituent Types

1163. Enterprise drones have *positions* expressed, for example, in terms of *longitude, latitude* and *altitude*.[18]
1164. Enterprise drones have *velocity* which is a vector of *speed* and three-dimensional, i.e., spatial, *direction*.
1165. Enterprise drones have *acceleration* which is a vector of *increase/decrease of speed per time unit* and *direction*.
1166. Enterprise drones have orientation which is expressed in terms of three quantities: *yaw, pitch* and *roll*.[19]

We leave *speed, direction* and *increase/decrease per time unit* unspecified.

type
1163 POS = P
1164 VEL = SPEED × DIRECTION
1165 ACC = IncrDecrSPEEDperTimeUnit × DIRECTION
1166 ORI = YAW × PITCH × ROLL
1164 SPEED = ...
1164 DIRECTION = ...
1165 IncrDecrSPEEDperTimeUnit = ...

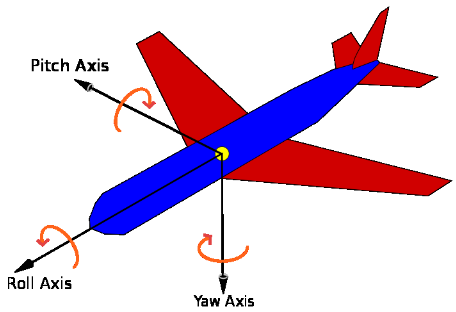


Fig. F.2. Aircraft Orientation

Attributes

1167. One of the enterprise properties is that of its *dynamics* which is seen as a quadruple of *velocity, acceleration, orientation* and *position*. It is recorded as a reactive attribute.

[18] *Longitude* is a geographic coordinate that specifies the east-west position of a point on the Earth's surface. It is an angular measurement, usually expressed in degrees and denoted by the Greek letter lambda. Meridians (lines running from the North Pole to the South Pole) connect points with the same longitude. *Latitude* is a geographic coordinate that specifies the northsouth position of a point on the Earth's surface. Latitude is an angle (defined below) which ranges from 0^o at the Equator to 90^o (North or South) at the poles. Lines of constant latitude, or parallels, run eastwest as circles parallel to the equator. *Altitude* or height (sometimes known as depth) is defined based on the context in which it is used (aviation, geometry, geographical survey, sport, and many more). As a general definition, altitude is a distance measurement, usually in the vertical or "up" direction, between a reference datum and a point or object. The reference datum also often varies according to the context.

[19] *Yaw, pitch* and *roll* are seen as symmetry axes of a drone: normal axis, lateral (or transverse) axis and longitudinal (or roll) axis. See Fig. F.2 [Page 376].

1168. Enterprise drones follow a flight course, as prescribed in and recorded as a programmable attribute, referred to a the *future flight plan*, FFP.
1169. Enterprise drones have followed a course recorded, also a programmable attribute, as a *past flight plan list*, PFPL.
1170. Finally enterprise drones "remember", in the form of a programmable attribute, the *geography* (i.e., the *area*, the *land* and the *weather*) it is flying over and in !

type
1170 ImG = A×L×W
1167 DYN = s_vel:VEL × s_acc:ACC × s_ori:ORI × s_pos:POS
1168 FPL = FP
1169 PFPL = FP*
value
1167 attr_DYN: ED → DYN
1168 attr_FPL: ED → FPL
1169 attr_PFPL: ED → PFPL
1170 attr_ImG: ED → ImG

Enterprise, as well as 'other' drone, positions must fall within the *Euclidian Point Space* of the geography of the universe of discourse. We leave that as an axiom to be defined – or we could decide that if a drone leaves that space then it is lost, and if drones suddenly "appear, out of the blue", then they are either "brand new", or "reappear".

Enterprise Drone Attribute Categories:

The position, velocity, acceleration, position and past position list attributes belong to the **reactive** category. The future position list attribute belong to the **programmable** category. Drones have a "zillion" more attributes – which may be introduced in due course.

'Other' Drones Attributes

Constituent Types

The constituent types of 'other' drones are similar to those of some of the enterprise drones.

Attributes

1171. 'Other' drones have *dynamics*, dyn:DYN.
1172. 'Other' drones "remember", in the form of a programmable attribute, the *immediate geography*, ImG (i.e., the *area*, the *land* and the *weather*) it is flying over and in !

type
1172 A, L, W
1172 ImG = A×L×W
value
1171 attr_DYN: OD → DYN
1172 attr_ImG: OD → ImG

Drone Dynamics

1173. By a timed drone dynamics, TiDYN, we understand a quadruplet of *time, position, dynamics* and *immediate geography*.
1174. By a *current drone dynamics* we shall understand a drone identifier-indexed set of timed drone dynamics.

1175. By a *record of [traces of] timed drone dynamics* we shall understand a drone identifier-indexed set of sequences of timed drone dynamics.

type
1173 $\mathsf{TiDYN} = \mathbb{T} \times \mathsf{POS} \times \mathsf{DYN} \times \mathsf{ImG}$
1174 $\mathsf{CuDD} = (\mathsf{EDI} \xrightarrow{m} \mathsf{TiDYN}) \cup (\mathsf{ODI} \xrightarrow{m} \mathsf{TiDYN})$
1175 $\mathsf{RoDD} = (\mathsf{EDI} \xrightarrow{m} \mathsf{TiDYN}^*) \cup (\mathsf{ODI} \xrightarrow{m} \mathsf{TiDYN}^*)$

We shall use the notion of *current drone dynamics* as the means whereby the *monitor* ascertains (obtains, by interacting with drones) the dynamics of drones, and the notion of a *record of [traces of] drone dynamics* in the *monitor*.

Drone Positions

1176. For all drones whether enterprise or 'other', their positions must lie within the geography of their universe of discourse.

axiom
1176 $\forall$ uod:UoD,e:E,g:G,d:(ED|OD) •
1176 e = obs_E(uod) $\wedge$ g = obs_G(uod) $\wedge$ d $\in$ xtr_Ds(uod) $\Rightarrow$
1176 **let** eps = attr_EPS(g), (_,_,p) = attr_DYN(d) **in** p $\in$ eps **end**

Monitor Attributes

The *monitor* "sits between" the *drones* whose dynamics it monitors and the *planner* which it provides with records of drone dymamics. Therefore we introduce the following.

1177. The monitor has just one, a programmable attribute: a trace of the most recent and all past time-stamped recordings of the dynamics of all drones, that is, an element rodd:RoDD, cf. Item 1175 [Page 378].

type
1177 MRoDD = RoDD
value
1177 attr_MRoDD: CM $\to$ MRoDD

The monitor "obtains" current drone dynamics, cudd:CuDD (cf. Item 1174 [Page 377]) from the *drones* and offers records of [traces of] drone dynamics,(cf. Item 1175 [Page 378]) rodd:RoDD, to the *planner*.

Planner Attributes

Swarms and Businesses:

The *planner* is where all decisions are made with respect to where enterprise drones should be flying; which enterprise drones fly together, which no longer – (with this notion of "flying together" leading us to the concept of *swarms*); which swarms of enterprise drones do which kinds of work – (with this notion of work specialisation leading us to the concept of businesses.)

1178. The is a notion of a *business identifier*, BI.

type
1178 BI

Planner Directories:

Planners have three directories. These are attributes, BDIR (businesses), SDIR (swarms) and DDIR (drones).

1179. BDIR records which swarms are resources of which businesses;
1180. SDIR records which drones "belong" to which swarms.
1181. DDIR "keeps track" of past and present enterprise drone positions, as per enterprise drone identifier.
1182. We shall refer to this triplet of directories by TDIR

type
1179 BDIR = BI $\overrightarrow{m}$ SI-**set**
1180 SDIR = SI $\overrightarrow{m}$ DI-**set**
1181 DDIR = DI $\overrightarrow{m}$ RoDD
1182 TDIR = BDIR × SDIR × DDIR
value
1179 attr_BDIR: CP → BDIR
1180 attr_SDIR: CP → SDIR
1181 attr_DDIR: CP → DPL

All three directories are *programmable attributes*.

The business swarm concept can be visualized by grouping together drones of the same swarm in the visualizarion of the aggregate set of enterprise drones. Figure F.3 [Page 379] attempts this visualization.

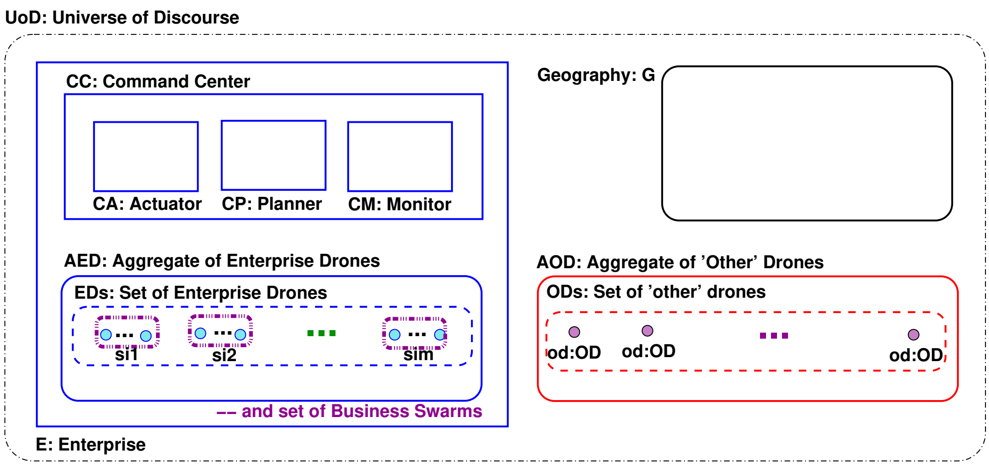


Fig. F.3. Conceptual Swarms of the Universe of Discourse

1183. For the planners of all universes of discourse the following must be the case.
   a. The swarm directory must
      i. have entries for exactly the swarms of the business directory,
      ii. define disjoint sets of enterprise drone identifiers, and
      iii. these sets must together cover all enterprise drones.
   b. The drone directory must record the present position, the past positions, a list, dpl:DPL, and, besides satisfying axioms 1176, satisfy some further constraints:
      i. they must list exactly the drone identifiers of the aggregate of enterprise drones, and the sum total of its enterprise drone identifiers must be exactly those of the enterprise drones aggregate of enterprise swarms, and

ii. the head of a drone's present and past position list must similarly be within reasonable distance of that drone's current position.

axiom
```
1183        ∀ uod:UpD,e:E,cp:CP,g:G •
1183           e=obs_E(uod)∧cp=obs_CP(obs_CC(e)) ⇒
1183a          let (bdir,sdir,ddir) = (attr_BDIR,attr_SDIR,attr_DDIR)(cp) in
1183(a)i          ∪ rng bdir = dom sdir
1183(a)ii      ∧ ∀ si,si′SI•{si,si′}⊆dom sdir∧si≠si′ ⇒
1183(a)ii              sdir(s) ∩ sdir(s′) = {}
1183(a)iii          ∧ ∪ rng sdir = xtr_dis(e)
1183(b)i      ∧ dom ddir = xtr_dis(e)
1183(b)ii      ∧ ∀ di:DI•di ∈ dom ddir
1183(b)ii              let (d,dpl) = (attr_DDIR(cp))(di) in
1183(b)ii               dpl ≠ ⟨⟩
1183(b)ii                  ⇒ neighbours(f,hd(dpl))
1183(b)ii                  ∧ neighbours(hd(dpl),
1183(b)ii                                     attr_EDPOS(xtr_D(obs_Ss(e))(di)))
1183           end end
```

Actuator Attributes

The actuator receives, from the planner, flight directives as to which enterprise drones should be redirected. The actuator maintains a record of most recent and all past such flight directives. Finally, the actuator, effects the directives by informing designated enterprise drones as to their next flight plans.

1184. Actuators have one programmable attribute: a flight directive directory. It lists, for each enterprise drone, by identifier, a pair: its current flight plan and a list of past flight plans.

type
```
1184  FDDIR = EDI →m (FP × FP*)
```
value
```
1184  attr_FDDIR: CA → FDDIR
```

Geography Attributes

Constituent Types:

The constituent types of *longitude, latitude* and *altitude* and *positions*, of a *geography*, were introduced in Items 1118.

1185. A further concept of geography is that of *area*.
1186. An area, a:A, is a subset of positions within the geography.

type
```
1185   A = P-infset
```
axiom
```
1186   ∀ uod:UoD,g:G,a:A• g=obs_G(uod) ⇒ a ⊆ attr_EPS(g)
```

Attributes

1187. Geographies have, as one of their attributes, a *Euclidian Point Space*, in this case, a *compact*[20] infinite set of three-dimensional positions.

type
1187 EPS = P-**infset**
value
1187 attr_EPS: G → EPS

Further geography attributes reflect the "lay of the land and the weather right now !".

1188. The "lay of the land", L is a "conglomerate" further undefined geodetics and cadestra[21]
1189. The "weather"W is another "conglomerate" of temperature, humidity, precipitation, air pressure, etc.

type
1188 L
1189 W
value
1188 attr_L: G → L
1189 attr_W: G → W

F.3 Operations on Universe of Discourse States

Before we analyse & describe perdurants let us take a careful look at the actions that drone and swarm behaviours may take. We refer to this preparatory analysis & description as one of analysing & describing the state operations. From this analysis & description we move on to the analysis & description of behaviours, events and actions. The idea is to be able to prove some relations between the two analyses & descriptions: the state operation and the behaviour analyses & descriptions. We refer to [241, Sects. 2.3 and 2.5].

F.3.1 The Notion of a State

A state is any subset of parts each of which contains one or more dynamic attributes. Following are examples of states of the present case study: a space of interest, an aggregate of 'business' swarms, an aggregate of 'other' swarms, a pair of the aggregates just mentioned, a swarm, or a drone.

F.3.2 Constants

Some quantities of a given universe of discourse are constants. Examples are the unique identifiers of the:

1190. enterprise, e_i;
1191. aggregate of 'other' drones, oi;
1192. geography, g_i;
1193. command center, cc_i;
1194. monitor, cm_i;
1195. planner, cp_i;
1196. actuator, ca_i;
1197. set of 'other' drones, od_{is};
1198. set of enterprise drones, ed_{is};
1199. and the set of all drones, ad_{is}.

[20] In mathematics, and more specifically in general topology, compactness is a property that generalizes the notion of a subset of Euclidean space being closed (that is, containing all its limit points) and bounded (that is, having all its points lie within some fixed distance of each other). Examples include a closed interval, a rectangle, or a finite set of points.

[21] land surface altitude, streets, buildings (tall or not so tall), power lines, etc.

```
value
1190 aed_i:EI = uid_AED(obs_AED(uod))
1191 aod_i:OI = uid_AOD(obs_AOD(uod))
1192 g_i:GI = uid_G(obs_G(uod))
1193 cc_i:CCI = uid_CC(obs_CC(obs_AED(uod)))
1194 cm_i:CMI = uid_CM(obs_CM(obs_CC(obs_AED(uod))))
1195 cp_i:CPI = uid_CP(obs_CP(obs_CC(obs_AED(uod))))
1196 ca_i:CAI = uid_CA(obs_CA(obs_CC(obs_AED(uod))))
1197 od_is:ODIs = xtr_dis(obs_AOD(uod))
1198 ed_is:EDIs = xtr_dis(obs_AED(uod))
1199 ad_is:DI-set = od_is ∪ ed_is
```

F.3.3 Operations

An operation is a function from states to states. Following are examples of operations of the present case study: a drone *transfer:* leaving a swarm to join another swarm, a drone *changing course:* an enterprise drone changing course, a swarm *split:* a swarm splitting into two swarms, and swarm *join:* two swarms joining to form one swarm.

A Drone Transfer

1200. The *transfer* operator specifies two distinct and unique identifiers, si, si′, of two enterprise swarms, and the unique identifier, di, of an enterprise drone – all of the same univserse of discourse. The *transfer* operation further takes a universe of discourse and yields a universe of discourse as follows:
1201. The input argument ‘from’ and ‘to’ swarm identifiers are diffent.
1202. The initial and the final state aggregates of enterprise drones, ‘other’ drones and geographies are unchanged.
1203. The initial and final state monitors and actuators are unchanged.
1204. The business and the drone directors of the initial and final planner are unchanged.
1205. The ’from’ and ‘to’ input argument swarm identifiers are in the swarm directory and the input argument drone identifiers is in the initial swarm directory entry for the ‘from’ swarm identifier.
1206. The input argument drone identifier is in final the swarm directory entry for the ‘to’ swarm identifier.
1207. And the final swarm directory is updated ...

```
value
1200 transfer: DI × SI × SI → UoD ∼→ UoD
1200 transfer(di,fsi,tsi)(uod) as uod′
1201     fsi ≠ tsi ∧
1200     let aed = obs_AED(uod), aed′ = obs_AED(uod′), g = obs_G(uod), g′ = obs_G(uod′) in
1200     let cc = obs_CC(aed), cc′ = obs_CC(aed′), aod = obs_AOD(uod), aod′ = obs_AOD(uod′) in
1200     let cm = obs_CM(cc), cm′ = obs_CM(cc′), cp = obs_CP(cc), cp′ = obs_CP(cc′) in
1200     let ca = obs_CA(cc), ca′ = obs_CA(cc′) in
1200     let bdir = attr_BDIR(cc), bdir′ = attr_BDIR(cc′),
1200         sdir = attr_SDIR(cc), sdir′ = attr_SDIR(cc′),
1200         ddir = attr_DDIR(cc), ddir′ = attr_DDIR(cc′) in
1202     post: aed = aed′ ∧ aod = aod′ ∧ g = g′ ∧
1203             cm = cm′ ∧ ca = ca′ ∧
1204             bdir = bdir′ ∧ ddir = ddir′
1205     pre {fsi,tsi} ⊆ dom sdir ∧ di ∈ sdir(fsi)
1206     post di ∉ sdir(fsi′) ∧ di ∈ sdir(tsi′) ∧
1207           sdir′ = sdir † [fsi↦sdir(fsi)∪ di] † [tsi↦sdir(tsi)\di]
1200     end end end end end
```

An Enterprise Drone Changing Course

TO BE WRITTEN

A Swarm Splitting into Two Swarms

TO BE WRITTEN

Two Swarms Joining to form One Swarm

TO BE WRITTEN

Etcetera

TO BE WRITTEN

F.4 Perdurants

We observe that the term *train* can have the following "meanings": the *train*, as an *endurant*, parked at the railway station platform, i.e., as a *composite part*; the *train*, as a *perdurant*, as it "speeds" down the railway track, i.e., as a *behaviour*; the *train*, as an *attribute*. This observation motivates that we "magically", as it were, introduce a COMPILEr function, cf. [2, Sect. 4]

F.4.1 System Compilation

The COMPILEr function "worms" its way, so-to-speak, "down" the "hierarchy" of parts, from the universe of discourse, via its immediate sup-parts, and from these to their sub-parts, and so on, until the COMPILEr reaches atomic parts. We shall henceforth do likewise.

The Compile Functions

1208. Compilation of a *universe of discourse* results in
   a. the RSL-**Text** of the *core* of the universe of discourse behaviour (which we set to **skip** – allowing us to ignore *core* arguments),
   b. followed by the RSL-**Text** of the parallel composition of the compilation of the enterprise,
   c. followed by the RSL-**Text** of the parallel composition of the compilation of the geography,
   d. followed by the RSL-**Text** of the parallel composition of the compilation of the aggregate of 'other' drones.

1208 COMPILE_{UoD}(uod) ≡
1208a $\mathscr{M}_{\text{uid\_UoD(uod)}}$(mereo_UoD(uod),sta(uod))(pro(uod))
1208b ‖ COMPILE_{AED}(obs_AED(uod))
1208c ‖ COMPILE_{G}(obs_G(uod))
1208d ‖ COMPILE_{AOD}(obs_AOD(uod))

1209. Compilation of an *enterprise* results in
   a. the RSL-**Text** of the *core* of the enterprise behaviour (which we set to **skip** – allowing us to ignore *core* arguments),
   b. followed by the RSL-**Text** of the parallel composition of the compilation of the enterprise aggregate of enterprise drones,
   c. followed by the RSL-**Text** of the parallel composition of the compilation of the enterprise command center.

1209 $\textsc{Compile}_{AED}$(e) ≡
1209a $\mathscr{M}_{\text{uid\_AED(e)}}$(mereo_E(e),sta(e))(pro(e))
1209b ∥ $\textsc{Compile}_{EDs}$(obs_EDs(e))
1209c ∥ $\textsc{Compile}_{CC}$(obs_CC(e))

1210. Compilation of an *enterprise aggregate of enterprise drones* results in
   a. the RSL-**Text** of the *core* of the aggregate behaviour (which we set to **skip** – allowing us to ignore *core* arguments),
   b. followed by the RSL-**Text** of the parallel composition of the distributed compilation of the enterprise aggregate's set of enterprise drones.

1210 $\textsc{Compile}_{EDs}$(es) ≡
1210a $\mathscr{M}_{\text{uid\_EDs(es)}}$(mereo_EDS(es),sta(es))(pro(es))
1210b ∥ {$\textsc{Compile}_{ED}$(ed)|ed:ED•ed ∈ obs_EDs(s)}

1211. Compilation of an *enterprise drone* results in
   a. the RSL-**Text** of the *core* of the enterprise drone behaviour – which is what we really wish to express – and since enterprise drones are here considered atomic, that is where the compilation of enterprise ends.

1211 $\textsc{Compile}_{ED}$(ed) ≡
1211a $\mathscr{M}_{\text{uid\_ED(ed)}}$(mereo_ED(ed),sta(ed))(pro(ed))

1212. Compilation of an *aggregate of 'other' drones* results in
   a. the RSL-**Text** of the *core* of the aggregate 'other' drones behaviour (which we set to **skip** – allowing us to ignore *core* arguments) –
   b. followed by the RSL-**Text** of the parallel composition of the distributed compilation of the 'other' drones in the 'other' drones' aggregate set of 'other' drones.

1212 $\textsc{Compile}_{AOD}$(aod) ≡
1212a $\mathscr{M}_{\text{uid\_OD(od)}}$(mereo_S(ods),sta(ods))(pro(ods))
1212b ∥ {$\textsc{Compile}_{OD}$(od)|od:OD•od ∈ obs_ODs(ods)}

1213. Compilation of a(n) *'other' drone* results in
   a. the RSL-**Text** of the *core* of the 'other' drone behaviour – which is what we really wish to express – and since 'other' drones are here considered atomic, that is where the compilation of the 'other' drones aggregate

1213a $\textsc{Compile}$_{OD}(ed) ≡
1213a $\mathscr{M}_{\text{uid\_OD(od)}}$(mereo_OD(od),sta(od))(pro(od))

1214. Compilation of an atomic *geography* results in
   a. the RSL-**Text** of the *core* of the geography behaviour.

1214 $\textsc{Compile}_{G}$(g) ≡
1214a $\mathscr{M}_{\text{uid\_G(g)}}$(mereo_G(g),sta(g))(pro(g))

1215. Compilation of a composite *command center* results in
   a. the RSL-**Text** of the *core* of the command center behaviour (which we set to **skip** – allowing us to ignore *core* arguments)
   b. followed by the RSL-**Text** of the parallel composition of the compilation of the command monitor,

c. followed by the RSL-**Text** of the parallel composition of the compilation of the command planner,
d. followed by the RSL-**Text** of the parallel composition of the compilation of the command actuator.

1215 $\text{COMPILE}_{M}(\text{cc}) \equiv$
1215a $\mathscr{M}_{\text{uid\_CC(cc)}}(\text{mereo\_CC(cc),sta(cc)})(\text{pro(cc)})$
1215b $\parallel \text{COMPILE}_{CC}(\text{obs\_CM(cc)})$
1215c $\parallel \text{COMPILE}_{CP}(\text{obs\_CP(cc)})$
1215d $\parallel \text{COMPILE}_{CA}(\text{obs\_CA(cc)})$

1216. Compilation of an atomic *command monitor* results in
a. the RSL-**Text** of the *core* of the monitor behaviour.

1216 $\text{COMPILE}_{CM}(\text{cm}) \equiv$
1216a $\mathscr{M}_{\text{uid\_CM(cm)}}(\text{mereo\_CM(cm),sta(cm)})(\text{pro(cm)})$

1217. Compilation of an atomic *command planner* results in
a. the RSL-**Text** of the *core* of the planner behaviour.

1217 $\text{COMPILE}_{CP}(\text{cp}) \equiv$
1217a $\mathscr{M}_{\text{uid\_CP(cp)}}(\text{mereo\_CP(cp),sta(cp)})(\text{pro(cp)})$

1218. Compilation of an atomic *command actuator* results in
a. the RSL-**Text** of the *core* of the actuator behaviour.

1218 $\text{COMPILE}_{CA}(\text{ca}) \equiv$
1218a $\mathscr{M}_{\text{uid\_CA(ca)}}(\text{mereo\_CA(ca),sta(ca)})(\text{pro(ca)})$

Some CSP Expression Simplifications

We can justify the following CSP simplifications [288, 289, 290, 291]:

1219. **skip** in parallel with any CSP expression *csp* is *csp*.
1220. The distributed parallel composition of the distributed parallel composition of CSP expressions, csp(i,j), *i* indexed over *I*, i.e., *i*:*I*, and *j*:*J* respectively, is the distributed parallel composition over CSP expressions, csp(i,j), i.e., indexed over $(i, j){:}I \times J$ – where the index sets iset and jset are assumed.

axiom
1220 **skip** $\parallel$ csp $\equiv$ csp
1220 $\parallel\{\parallel\{\text{csp(i,j)}|\text{i:I}\bullet\text{i}\in\text{iset}\}|\text{j:J}\bullet\text{j}\in\text{jset}\} \equiv \parallel\{\text{csp(i.j)}|\text{i:I,j:J}\bullet\text{i}\in\textbf{I-set}\wedge\text{j}\in\textbf{J-set}\}$

The Simplified Compilation

1221. The simplified compilation results in:

1221 $\text{COMPILE}(\text{uod}) \equiv$
1211a $\{\ \mathscr{M}_{\text{uid\_ED(ed)}}(\text{mereo\_ED(ed),sta(ed)})(\text{pro(ed)})$
1211a $|\ \text{ed:ED} \bullet \text{ed} \in \text{xtr\_Ds(obs\_AED(uod))}\ \}$
1213a $\parallel \{\ \mathscr{M}_{\text{uid\_OD(od)}}(\text{mereo\_OD(od),sta(od)})(\text{pro(od)})$
1213a $|\ \text{od:OD} \bullet \text{od} \in \text{xtr\_ODs(obs\_AOD(uod))}\ \}$
1214a $\parallel \mathscr{M}_{\text{uid\_G(g)}}(\text{mereo\_G(g),sta(g)})(\text{pro(g)})$
1214a **where** g $\equiv$ obs_G(uod)

1216a $\parallel \mathscr{M}_{\text{uid\_CM(cm)}}$(mereo_CM(cm),sta(cm))(pro(cm))
1216a **where** cm ≡ obs_CM(obs_CC(obs_E(uod)))
1217a $\parallel \mathscr{M}_{\text{uid\_CP(cp)}}$(mereo_CP(cp),sta(cp))(pro(cp))
1217a **where** cp ≡ obs_CP(obs_CC(obs_E(uod)))
1218a $\parallel \mathscr{M}_{\text{uid\_CA(ca)}}$(mereo_CA(ca),sta(ca))(pro(ca))
1218a **where** ca ≡ obs_CA(obs_CC(obs_E(uod)))

1222. In Item 1221's Items 1211a, 1213a, 1214a, 1216a, 1217a, and 1218a we replace the "anonymous" behaviour names $\mathscr{M}$ by more meaningful names.

1222 COMPILE(uod) ≡
1211a { enterprise_drone$_{\text{uid_ED(ed)}}$(mereo_ED(ed),sta(ed))(pro(ed))
1211a | ed:ED • ed ∈ xtr_Ds(obs_AED(uod)) }
1213a ∥ { other_drone$_{\text{uid_OD(od)}}$(mereo_OD(od),sta(od))(pro(od))
1213a | od:OD • od ∈ xtr_ODs(obs_AOD(uod)) }
1214a ∥ geography$_{\text{uid_G(g)}}$(mereo_G(g),sta(g))(pro(g))
1214a **where** g ≡ obs_G(uod)
1216a ∥ monitor$_{\text{uid_CM(cm)}}$(mereo_CM(cm),sta(cm))(pro(cm))
1216a **where** cm ≡ obs_CM(obs_CC(obs_E(uod)))
1217a ∥ planner$_{\text{uid_CP(cp)}}$(mereo_CP(cp),sta(cp))(pro(cp))
1217a **where** cp ≡ obs_CP(obs_CC(obs_E(uod)))
1218a ∥ actuator$_{\text{uid_CA(ca)}}$(mereo_CA(ca),sta(ca))(pro(ca))
1218a **where** ca ≡ obs_CA(obs_CC(obs_E(uod)))

F.4.2 An Early Narrative on Behaviours

Either Endurants or Perdurants, Not Both !

First the reader should observe that the manifest parts, in some sense, do no longer "exist" ! They have all been replaced by their corresponding behaviours. These behaviours embody all the qualities of their "origin": the unique identifiers, the mereology, and all the attributes – the latter in one form or another: the static attributes as constants (referred to in the bodies of the behaviour definitions); the programmable attributes as arguments (''carried over'' from one invocation to the next); and the remaining dynamic attributes as "inputs" (whose varying values are ''accessed'' through [dynamic attribute] channels).

Focus on Some Behaviours, Not All !

Secondly we focus, in this case study, only on the behaviour of the *planner*. The other behaviours, the *'other' drones*, *enterprise drones*, *monitor*, *actuator*, and the *geography*, are, in this case study of less interest to us. That is, other case studies could focus on the behaviours of *drones*, or *geographies*, or *monitor*, or *actuator*.

The Behaviours – a First Narrative

*Drone*s "continuously" offer their identified dynamics (location, velocity, and possibly more) **to** the *monitor*. *Enterprise drones* "continuously", and in addition, offers to accept flight guidance **from** the *actuator*. The *monitor* "continuously sweeps" the air space and collects the identities of all recognizable drones and their dynamics, and offers this **to** the *planner*. The *planner* does all the interesting work ! It effects the *allocation/reallocation* of drones to/from business swarms; it *calculates enterprise drone flights* and instructs the *actuator* to offer such flight plans to relevant drones; etcetera ! Finally the *actuator*, as instructed by the *planner*, offers flight guidance, as per instructions **from** the *planner*, **to** all or some *enterprise drones*.

F.4.3 Channels

Channels is a concept of CSP [288, 284, 289].

CSP channels are a means for synchronising behaviours and for communicating values between synchronised behaviours, as well as, as a technicality, conveying values of most kinds of dynamic attributes of parts (i.e., endurants) to "their" behavioural counterparts.

There are thus two starting point for the analysis & description of channels: the mereologies and the dynamic attributes of parts. Here we shall single out the following parts and behaviours: the command *monitor, planner* and *actuator*, the *enterprise drones* and the *'other' drones*, and the *geography*. We refer to Fig. F.4 [Page 387], a slight "refinement" of Fig. F.1 [Page 366].

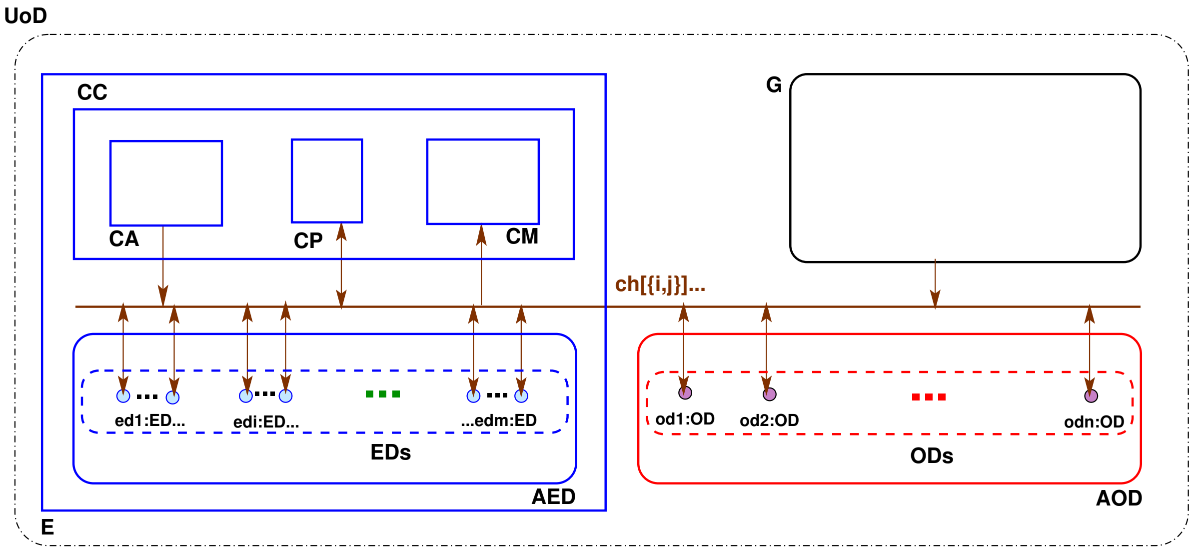


Fig. F.4. Universe of Discourse with General Channel: **ch[{i,j}] ...**

The Part Channels

General Remarks:

Let there be given a *universe of discourse*. Let us analyse the *unique identifiers* and the *mereologies* of the *planner* cp: (cpi,cpm), *monitor* cm: (cmi,cmm) and *geography* g: (mi,mm), where cpm = (cai,cmi,gi), cmm = $(\{di_1, di_2, \ldots, di_n\}$,cpi) and gm = (cpi,$\{di_1, di_2, \ldots, di_n\}$).

We now interpret these facts. When the *planner mereology* specifies the unique identifiers of the *actuator*, the *monitor*, and the *geography*, then that shall mean there there is a way of communicating messages between the actuator, and the geography, amd one side, and the plannner on the other side.

1223. We shall therefore, in a first step of specification development, think of a "grand" array channel over which all communication between behaviours take place. See Fig. F.4 [Page 387].
1224. Example indexes into this array channel are shown in the formulas just below.

type
1223 MSG
channel
1223 {ch[fui,tui]|fui,tui:PI • ...}:MSG
value

1224 ch[cpi,cai]!msg *output from* planner *to* actuator.
1224 ch[cpi,cai]? *input from* planner *to* actuator.
1224 etc.

We presently leave the type of messages, MSG, that can be communicated over this “grand” channel further unspecified. We also leave unspecified the pair of distinct unique identifiers that index the channel array. We emphasize that the uniqueness of all part identifiers allow us to use pairs of such as indices. Expression ch[fui,tui]!,sg thus expresses *output from* behaviour indexed by fuit *to* behaviour indexed by tui, whereas expression ch[tui,fui]? thus expresses *input from* behaviour indexed by tui *to* behaviour indexed by fui. Not all combinations of unique identifiers are needed. The channel array is “sparse” ! That property allows us to refine the “grand” channel into the channels illustrated on Fig. F.5 [Page 388]. Some channels are

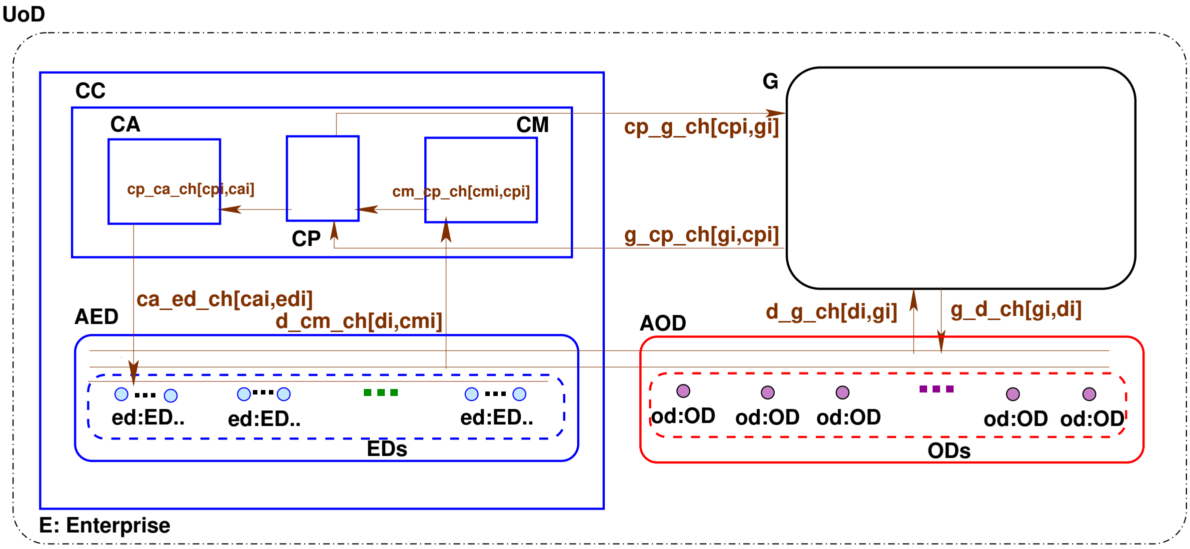


Fig. F.5. Universe of Discourse with Specific Channels

array channels: The channels to the drones whether all drones, or just the enterprise drones. Other channels are “single” channels: these are the channels which are anchored in parts with a priori known, i.e., constant unique identifiers.

Part Channel Specifics

1225. There is an array channel, d_cm_ch[di,cm_i]:D_CM_MSG, from any *drone* ([di]) behaviour to the *monitor* behaviour (whose unique identifier is cm_i). The channel, as an array, forwards the current drone dynamics D_CM_MSG = CuDD.

type
1225 D_CM_MSG = CuDD
channel
1225 {d_cm_ch[di,cm_i]|di:(EDI|ODI)•di $\in$ dis}:D_CM_MSG

1226. There is a channel, cm_cp_ch[cm_i,cp_i, from the *monitor* behaviour (cm_i) to the *planner* behaviour (cp_i). It forwards the monitor’s records of drone dynamics CM_CP_MSG = MRoDD.

type
1226 CM_CP_MSG = MRoDD

1226 **channel** m_cp_ch[cm_i,cp_i]:CM_CP_MSG

1227. There is a channel, cp_ca_ch[cp_i,ca_i]:CP_CA_MSG, from the *plannner* behaviour (cp_i) to the *actuator* behaviour (ca_i). It forwards flight plans CP_CA_MSG = FP.

type
1227 CP_CA_MSG = EID $\overrightarrow{m}$ FP
channel
1227 cp_ca_ch[cp_i,ca_i]:CM_CP_MSG

1228. There is an array channel, ca_ed_ch[cai,edi], from the *actuator behaviour* (ca_i) to the *enterprise drone* behaviours (edi for suitable edis). It forwards flight plans, CA_ED_MSG = FP, to enterprise drones in a designated set.

type
1228 CA_ED_MSG = EID × FP
channel
1228 {ca_ed_ch[cai,edi]|edi:EDI•edi ∈ $edis$}:CA_ED_MSG

1229. There is an array channel, g_d_ch[di,g_i]:D_G_MSG, from all the *drone* behaviours (di) to the *geography* behaviour The channels convey, requests for an *immediate geography* for and around a *point*: D_G_MSG = P.

type
1229 D_G_MSG = P
channel
1229 {d_g_ch[di,g_i]|di:(EDI|ODI)•di ∈ dis}:D_H_MSG

1230. There is an array channel, g_d_ch[g_i,di]:G_D_MSG, from the *geography* behaviour to all the *drone* behaviours. The channels convey, for a requested *point*, the immediate geography for that area: G_D_MSG = ImG.

type
1230 G_D_MSG = ImG
channel
1230 {g_d_ch[g_i,di]|di:(EDI|ODI)•di ∈ dis}:G_D_MSG

Attribute Channels, General Princiles

Some of the drone attributes are *reactive*. Being reactive means that their values change surreptitiously. In the physical world of parts that means that these vales must be measured, or somehow ascertained, whenever needed, i.e., "on the fly". Now "our world" is that of a domain description. When dealing with endurants, the value of an attribute, a:A, of part p:P, is expressed as attr_A(p). When dealing with perdurants, that same value is to be expressed as attr_A_ch[uid_P(p)] ?.

1231. This means that we must declare a channel for each part with one or more *dynamic*, however not including *programmable*, attributes A1, A2, ..., An.

channel
1231 attr_A1_ch[p$_i$]:A1, attr_A2_ch[p$_i$]:A2, ..., attr_An_ch[p$_i$]:An

1232. If there are several parts, p1,p2,. . . ,pm:P then an array channel over indices p1$_i$,p2$_i$,. . . ,pm$_i$ is declared for each applicable attribute.

channel
1232 {attr_A1_ch[pj$_i$]|pj$_i$:PI•pj$_i$∈{p1$_i$,p2$_i$,...,pm$_i$}}:A1,
1232 {attr_A2_ch[pj$_i$]|pj$_i$:PI•pj$_i$∈{p1$_i$,p2$_i$,...,pm$_i$}}:A2,
1232 ...
1232 {attr_An_ch[pj$_i$]|pj$_i$:PI•pj$_i$∈{p1$_i$,p2$_i$,...,pm$_i$}}:An

The Case Study Attribute Channels

‘Other’ Drones:

‘Other’ drones have the following not biddable or programmable dynamic channels:

1233. dynamics, including velocity, acceleration, orientation and position, {attr_DYN_ch[odi]:DYN|odi:ODI•odi∈*odis*}.

channel
1233 {attr_DYN_ch[odi]:DYN|odi:ODI•odi ∈ *odis*}

Enterprise Drones:

Enterprise drones have the following not biddable or programmable dynamic channels:

1234. dynamics, including velocity, acceleration, orientation and position, {attr_DYN_ch[edi]:DYN|edi:EDI•edi∈*odis*}.

channel
1234 {attr_DYN_ch[odi]:DYN|odi:ODI•odi ∈ *odis*}

Geography:

The geography has the following not biddable or programmable dynamic channels:

1235. land, attr_L_ch[g_i]:L, and
1236. weather, attr_W_ch[g_i]:W.

channel
1235 attr_L_ch[g_i]:L
1236 attr_W_ch[g_i]:W

We do not show any graphics for the attribute channels.

F.4.4 The Atomic Behaviours

TO BE WRITTEN

Monitor Behaviour

1237. The signature of the monitor behaviour
   a. lists the monitor’s unique identifier, carries the monitor’s mereology, has no static arguments (... maybe ...), has the programmable time-stamped recordings, dtp, of all drone positions (present and past) and
   b. further designates the **in**put channel d_cm_ch[*.*] from all drones and the channel **out**put cm_cp_ch[cmi,cpi] to the planner.
1238. The monitor [otherwise] behaves as follows:
   a. All drones provide as input, d_cm_ch[di,cmi]?, their time-stamped positions, rec.
   b. The programmable mrodd attribute is updated, mrodd′, to reflect the latest time stamped dynamics per drone identifier.
   c. The updated attribute is is provided to the planner.
   d. Then the monitor resumes being the monitor, forwarding, as the progammable attribute, the time-stamped drone position recording.

```
value
1237a  monitor: cmi:CMI×cmm:(dis:DI-set×cpi:CPI) → MRoDD →
1237b     in {d_cm_ch[di,cmi]|di:DI•di∈dis} out cm_cp_ch  Unit
1238     monitor(mi,(dis,cpi))(mrodd) ≡
1238a     let rec = {[di ↦ d_cm_ch[di,cmi]?|di:DI•di∈dis]} in
1238b     let mrodd′ = mrodd † [di↦⟨rec(di)⟩⌢mrodd(di)|di:DI•di∈dis] in
1238c     cm_cp_ch[cmi,cpi] ! mrodd′;
1238d     monitor(cmi,(dis,cpi))(mrodd′)
1238       end end
1238   axiom cmi=cm_i∧cpi=cp_i
```

We have decided to let the monitor maintain the present and past time-stamped drone positions. It is the monitor which records these positions. Not the planner. But the monitor provides these traces, again-and-again, to the planner.

Planner Behaviour

1239. The signature of the planner behaviour
   a. lists the planner's unique identifier, carries the planner's mereology, has, perhaps ..., some static arguments, has the programmable planner directories, and
   b. further designates the single **in**put channel cm_cp_ch and the single **out**put channel cp_ca_ch.
1240. The planner [otherwise] behaves as follows:
   a. the planner [internal] non-deterministically ("coin-flipping") decides whether to transfer a drone between business swarms, or to calculate flight plans, or ... other.
   b. Depending on the [outcome of the "coin-flipping"] the planner
   c. either effects a transfer,
      i. by delegating to an auxiliary function, transfer, the necessary modifications of the swarm directory –
      ii. whereupon the planner behaviour resumes;
   d. or effects a [re-]calculation on drone flights,
      i. by, again, delegating to an auxiliary function, flight_planning, the necessary calculations –
      ii. which are communicated to the *actuator*,
      iii. whereupon the planner behaviour resumes;
   e. or ... other!

```
value
1240   planner: cpi:CPI × (cai>CAI×cmi:CMI×gi:GI) × TDIR →
1240      in cm_cp_ch[cmi,cpi], g_cp_ch[gi,cpi] out cp_ca_ch[cpi,cai]  Unit
1239   planner(cpi,(cai,cmi,gi),...)(bdir,sdir,ddir) ≡
1240a     let cmd = "transfer" ⌈⌉ "flight_plan" ⌈⌉ ... in
1240b     cases cmd of
1240c        "transfer" →
1240(c)i           let sdir′ = transfer(tdir) in
1240(c)ii           planner(cpi,(cai,cmi,gi),...)(bdir,sdir′,ddir) end
1240d        "flight_plan" →
1240(d)i           let ddir′ = flight_planning(tdir) in
1240(d)ii           planner(cpi,(cai,cmi,gi),...)(bdir,sdir,ddir′) end
1240e        ...
1239     end
1239   axiom cpi=cp_i∧cai=ca_i∧cmi=cm_i∧gi=g_i
```

The Auxiliary transfer Function

1241. The *transfer* function has a simpler signature than the planner behaviour in that it need not communicate with other behaviours.
    a. The transfer function *internal non-deterministically chooses* a business designator, bi;
    b. from among that business' swarm designators it *internal non-deterministically chooses* two distinct swarm designators, fsi,tsi;
    c. and from the fsi entry in sdir (which is set of enterprise drone identifiers), it *internal non-deterministically chooses* an enterprise drone identifier, di.
    d. Given the swarm and drone identifiers *the resulting swarm directory* can now be made to reflect the transfer: reference to di is *removed* from the fsi entry in sdir and that reference instead *inserted* into the tsi entry.

```
value
1241    transfer: TDIR → SDIR
1241    transfer(bdir,sdir,ddir) ≡
1241a       let bi:BI•bi ∈ dom bdir in
1241b       let fsi,tsi:SI•{fsi,tsi}⊆bdir(bi)∧fsi≠tsi in
1241c       let di:DI•di ∈ sdir(fsi) in
1241d       sdir † [fsi↦sdir(fsi)\{di}] † [tsi↦sdir(tsi)∪{di}]
1241      end end end
```

The Auxiliary flight_planning Function

1242. The signature of the flight_planning behaviour needs two elements: the triplet of business, swarm and drone directories, and the planner-to-actuator channel.
    a. The flight_planning behaviour offers to accept the time-stamped recordings of the most recent drone positions and dynamics as well as all the past such recordings.
    b. The flight_planning behaviour selects, *internal, non-deterministically* a business, designated by bi,
    c. one of whose swarms, designated by si, it has thus decided to perform a flight [re-]calculation for.
    d. An objective for the new flight plan is chosen.
    e. The flight_plan is calculated.
    f. That flight plan is communicated to the *actuator*.
    g. And the flight plan, appended to the drone directory's (past) flight plans.

```
value
1242    flight_planning: TDIR → in cm_cp_ch[cm_i,cp_i], out cp_ca_ch[cp_i,ca_i]  DTP
1242    flight_planning(bdir,sdir,ddir) ≡
1242a       let dtp = cm_cp_ch[cp_i,ca_i] ? ,
1242b           bi:BI • bi ∈ dom bdir
1242c       let si:SI • si ∈ bdir(bi) in
1242d       let fp_obj:fp_objective(bi,si) in
1242e       let flight_plan = calculate_flight_plan(dtp,sdir(si),fp_obj,tdir) in
1242f       cp_ca_ch[cp_i,ca_i] ! flight_plan ;
1242g       ⟨flight_pla⟩⌢ddir
1242      end end end end
type
1242d     FP_OBJ
value
1242d     fp_objective: BI × SI → FP_OBJ
1242d     fp_objective(bi,si) ≡ ...
```

1243. The calculate_flight_plan function is the absolute focal point of the *planner*.

```
1243   calculate_flight_plan: DTP × DI-set × FP−OBJ × TDIR → FP
1243   calculate_flight_plan(dtp,sdir(si),fp_obj,tdir) ≡ ...
```

There are many ways of calculating flight plans.

[287, Mehmood et al., Stony Brook, 2018: *Declarative vs Rule-based Control for Flocking Dynamics*] is one such:

TO BE WRITTEN

In [292, 293, 294, Craig Reynolds: OpenSteer, *Steering Behaviours for Autonomous Characters*]

TO BE WRITTEN

In [295, Reza Olfati-Saber: Flocking for Multi-agent Dynamic Systems: Algorithms and Theory, 2006]

TO BE WRITTEN

The calculate_flight_plan function, Item 1243 [Page 392], is deliberately provided with all such information that can be gathered and hence can be the only 'external'[22] data that can be provided to such calculation functions,[23] and is therefore left further unspecified; future work[24] will show whether this assumption holds. If it does, then, OK, and we can proceed. If it does not, we shall revise the present model.

Actuator Behaviour

1244. The actuator accepts a current flight plan, cfp:CFP, i.e., a number of enterprise drone identifier-indexed flight plans, from the planner.
1245. The signature of the actuator behaviour lists the actuator's unique identifier, carries the actuator's mereology, has, perhaps ..., some static arguments, has the programmable flight directory, and further designates the **in**put channel cp_ca_ch[cpi,cai] and the **out**put channel ca_ed_ch[cai,*].
1246. The actuator further behaves as follows:
   a. It offers to accept a current flight plan from the planner.
   b. It then proceeds to offer those enterprise drones which are designated in the flight plan their flight plan.
   c. Whereupon the actuator resumes being the actuator, now with its programmable flight plan directory updated with the latest such !

```
type
1244   CFP = EDI →m FP
value
1245   actuator: cai:CAI × (cpi:CPI×edis:EDI-set) → FDDIR →
1245       in cp_ca_ch[cpi,cai] out {ca_ed_ch[cai,edi]|edi:EDI•edi ∈ edis}  Unit
1246   actuator(cai,(cpi,edis),...)(pfp,pfpl) ≡
1246a      let cfp = ca_cp_ch[cai,cpi] ? in comment: fp:EDI →m FP
1246b      || {ca_ed_ch[cai,edi]!cfp(edi)|edi:EDI•edi ∈ dom cfp} ;
1246c      actuator(cai,(cpi,edis),...)(cfp,⟨pfp⟩⌢pfpl)
1244       end

1245   axiom cai=ca_i∧cpi=cp_i
```

[22] Flight plan *objectives* are here referred to as 'internal'.

[23] Well – better check this!

[24] – for you ShaoFa !

‘Other’ Drone Behaviour

1247. The signature of the *‘other’ drone* behaviour
   a. lists the ‘other’ drone’s unique identifier, the ‘other’ drone’s mereology, has, perhaps ..., some static arguments; then the programmable attribute of the geography (i.e., the area, the land and the weather) it is moving over and in;
   b. then, as **in**put channels, the *inert, active, autonomous* and *biddable* attributes: velocity, acceleration, orientation and position, and, finally
   c. further designates the array **in**put channel g_d_ch[*] from the *geography* and the array **out**put channel d_cm_ch[*] to the *monitor*.
1248. The ‘other’ drone otherwise behaves as follows:
1249. internal, non-deterministically the ‘other’ drone chooses to either ..., or "pro"viding to the monitors request for drone "dyn"amics, or ... .
1250. If the choice is ... ,
1251. If the choice is "provide dynamics" the behaviour drone_monitor is invoked, with arguments similar to that of other_drone, but “marked” with an additional, “frontal” argument: "other", and with “tail”, programmable arguments (⟨⟩,⟨⟩).
1252. If the choice is ... .

```
value
1247   other_drone: odi:ODI × (cmi:CMI×gi:GI) × ... → (DYN×ImG) →
1247b      in attr_DYN_ch[odi],g_d_ch[gi,odi] out d_cm_ch[odi,cmi]  Unit
1248   other_drone(odi,(cmi,gi),...)(dyn:(v,a,o,p),img) ≡
1249      let mode = "..." ⌈⌉ "pro_dyn" ⌈⌉ "..." in
1249      case mode of
1250         "..." → ... ,
1251         "pro_dyn" → drone_moni(odi,(cmi,gi),...)(dyn:(v,a,o,p),img)
1252         "..." → ...
1249      end
1247      end
```

1253. If the choice is "provide dynamics"
   a. then the drone-monitor behaviour ascertains its dynamics (velocity, acceleration, orientation and position),
   b. informs the monitor ‘thereof’, and
   c. resumes being the ‘other’ drone with that updated, programmable dynamics.

```
value
1253    drone_moni: odi:ODI × (cmi:CMI×gi:GI) × ... → (DYN×ImG) →
1253       in attr_DYN_ch[odi],g_d_ch[gi,odi] out d_cm_ch[odi,cmi]  Unit
1252    drone_moni(odi,(cmi,gi),...)(dyn:(v,a,o,p),img) ≡
1253a      let (ti,dyn′,img′) =
1253a            (time(),
1253a               (let (v′,a′,o′,p′) = attr_DYN[odi]? in
1253a               (v′,a′,o′,p′),
1253a                  d_g_ch[odi,gi]!p′ ; g_d_ch[gi,odi]? end)) in
1253b       d_cm_ch[odi,cmi] ! (ti,dyn′) ;
1253c       other_drone(cai,(cpi,edis),...)(dyn′,img′)
1253a       end
```

Enterprise Drone Behaviour

1254. The enterprise donor lists its enterprise drone's unique identifier, carries it's mereology, has, perhaps ..., some static arguments, the programmable enterprise drone attributes: a pair of the present flight plan, and the past flight plans, and a pair of the most recently observed dynamics and immediate geography, and further designates the single **in**put channel and the **out**put channel array .

Enterprise drones otherwise behave as follows:

1255. internal, non-deterministically an enterprise drone chooses to either "rec"ording the "geo"graphy, i.e., the area, land and weather it is situated in, or "pro"viding to the monitors request for drone "dyn"amics, or "acc"epting the actuators offer of a new "f"light "p"lan, or "move" "on" (i.e., continue to fly), either "follow"ing the "flight plan" most recently received from the actuator, or, "ignor"ing this directive, "just plondering on" !
1256. If the choice is "rec_geo" then the enterprise_geo behaviour is invoked,
1257. If the choice is "pro_dyn" (provide dynamics to the *monitor*) then the enterprise_moni behaviour is invoked,
1258. If the choice is "acc_fp" then the enterprise_accept_flight_plan behaviour is invoked,
1259. If the choice is "move_on" then the enterprise drone decides either to "ignore" the flight plan, or to "follow" it.
    a. If it "ignore"s the flight plan then the enterprise_ignore behaviour is invoked,
    b. If the choice is "follow" then the enterprise_follow behaviour is invoked.

```
1254   enterprise_drone: edi:EDI×(cmi:CMI×cai:CAI×gi:GI) →
1254      ((FPL×PFPL)×(DDYN×ImG)) →
1254      in attr_DYN_ch[edi],g_d_ch[gi,edi],ca_ed_ch[cai,edi]
1254      out d_cm_ch[edi,cmi],d_g_ch[edi,gi]  Unit
1254   enterprise_drone(edi,(cmi,cai,gi),...)(fpl,pfpl,(ddyn,img)) ≡
1255      let mode = "rec_geo" ⌈⌉ "pro_dyn" ⌈⌉ "acc_fp" ⌈⌉ "move_on" in
1255      case mode of
1256         "rec_geo" → enterprise_geo(edi,(cmi,cai,gi),...)(fpl,pfpl,(ddyn,img))
1257         "pro_dyn" → enterprise_moni(edi,(cmi,cai,gi),...)(fpl,pfpl,(ddyn,img))
1258         "acc_fp" → enterprise_acc_fl_pl(edi,(cmi,cai,gi),...)(fpl,pfpl,(ddyn,img))
1259         "move_on" →
1259              let m_o_mode = "ignore" ⌈⌉ "follow" in
1259              case m_o_mode of
1259a                  "ignore" → enterprise_ignore(edi,(cmi,cai,gi),...)(fpl,pfpl,(ddyn,img))
1259b                  "follow" → enterprise_follow(edi,(cmi,cai,gi),...)(fpl,pfpl,(ddyn,img))
1265              end
1265              end
1255      end
1255      end
1254   axiom  cmi=cm_i∧cai=ca_i∧gi=g_i
```

1260. If the choice is "rec_geo"
    a. then dynamics is ascertained so as to obtain a positions;
    b. that position is used in order to obtain a "fresh" immediate geography;
    c. with which to resume the enterprise drone behaviour.

```
1254   enterprise_geography: edi:EDI×(cmi:CMI×cai:CAI×gi:GI) →
1254      ((FPL×PFPL)×(DDYN×ImG)) →
1254      in attr_DYN_ch[edi],g_d_ch[gi,edi],ca_ed_ch[cai,edi]
1254      out d_cm_ch[edi,cmi],d_g_ch[edi,gi]  Unit
1254   enterprise_geography(edi,(cmi,cai,gi),...)((fpl,pfpl),(ddyn,img)) ≡
1260a    let (v,a,o,p) = attr_DYN_ch[edi]? in
```

```
1260b     let img′ = d_g_ch[edi,gi]!p;g_d_ch[gi,edi]? in
1260c     enterprise_drone(edi,(cmi,cai,gi),...)((fpl,pfpl),((v,a,o,p),img′))
1260a     end end
```

1261. If the choice is "pro_dyn" (provide dynamics to the *monitor*)
   a. then a triplet is obtained as follows:
   b. the current time,
   c. the dynamics (v,a,o,p), and
   d. the immediate geography of position p,
   e. such that the *monitor* can be given the current dynamics,
   f. and the enterprise drone behaviour is resumed with updated dynamics and immediate geography.

```
1254   enterprise_monitor: edi:EDI×(cmi:CMI×cai:CAI×gi:GI) →
1254       ((FPL×PFPL)×(DDYN×ImG)) →
1254       in attr_DYN_ch[edi],g_d_ch[gi,edi],
1254       out d_cm_ch[edi,cmi],d_g_ch[edi,gi]  Unit
1254   enterprise_monitor(edi,(cmi,cai,gi),...)((fpl,pfpl),(ddyn,img)) ≡
1261a      let (ti,ddyn′,img′) =
1261b              (time(),
1261c                  (let (v,a,o,p) = attr_DYN[edi]? in
1261c                  (v,a,o,p),
1261d                       d_g_ch[edi,gi]!p;g_d_ch[gi,edi]? end)) in
1261e       d_cm_ch[edi,cmi] ! (ti,ddyn′) ;
1261f       enterprise_drone(edi,(cmi,cai,gi),...)((fpl,pfpl),(ddyn′,img′))
1261a       end
```

1262. If the choice is "acc_fp"
   a. the enterprise drone offers to accept a new flight plan from the *actuator*
   b. and the enterprise drone behaviour is resumed with that flight plan now becoming the next current flight plan and whatever is left of the hitherto current flight plan appended to the past flight plan list.

```
1254   enterprise_acc_fl_pl: edi:EDI×(cmi:CMI×cai:CAI×gi:GI) →
1254       ((FPL×PFPL)×(DDYN×ImG)) → in ca_ed_ch[cai,edi]  Unit
1254   enterprise_axx_fl_pl(edi,(cmi,cai,gi),...)((fpl,pfpl),(ddyn,img)) ≡
1262a      let fpl′ = ca_ed_ch[cmi,edi] ? in
1262b      enterprise_drone(edi,(cmi,cai,gi),...)(fp′,⟨fpl⟩⁀pfpl,(ddyn,img))
1262a      end
```

1263. If the choice is "move_on" and the enterprise drone decides to "ignore" the flight plan,
   a. then it ascertains where it might be moving with the current dynamics
   b. and then it just keeps moving on till it reaches that dynamics
   c. from about where it resumes the enterprise drone behaviour.

```
1254   enterprise_ignore: edi:EDI×(cmi:CMI×cai:CAI×gi:GI) →
1254       ((FPL×PFPL)×(DDYN×ImG)) →
1254       in attr_DYN_ch[edi] out d_cm_ch[edi,cmi],d_g_ch[edi,gi]  Unit
1254   enterprise_ignore(edi,(cmi,cai,gi),...)((fpl,pfpl),(ddyn,img)) ≡
1263a      let (v′,a′,o′,p′) = increment(dyn,img) in
1263b      while let (v″,a″,o″,p″) = attr_DYN_ch[odi]? in
1263b            ∼close(p′,p″) end do manoeuvre(dyn,img) ; wait δ t end ;
1263c      enterprise_drone(cai,(cpi,edis),...)(fpl,pfpl,(attr_DYN_ch[odi]?,img))
1263a      end
```

1264. The manoeuvre behaviour is further unspecified. For a fixed wing aircraft it controls the *yaw*, the *roll* and the *pitch* of the aircraft, hence its flight path, by operating the *elevator, aileron, ruddr* and the *thrust* of the aircraft based on its current dynamics, weight (including aircraft fuel), meteorological conditions (winds etc.).

```
value
1264    manoeuvre: DYN × ImG → Unit
1264    manoeuvre(dyn,img) ≡ ...
```

The **wait** δ t is some drone constant.

1265. If the choice is "move_on" and the enterprise drone decides to "follow" the flight plan,
   a. then, if the current flight plan has been exhausted, i.e., "used-up" it aborts (**chaos**[25])
   b. otherwise it ascertains where it might be moving, i.e., a next dynamics from with the current dynamics.
   c. So it then "moves along" until it has reached that dynamics –
   d. from about where it resumes the enterprise drone behaviour.

```
value
1254    enterprise_follow: edi:EDI×(cmi:CMI×cai:CAI×gi:GI) →
1254       ((FPL×PFPL)×(DDYN×ImG)) →
1254       in attr_DYN_ch[edi] out d_cm_ch[edi,cmi],d_g_ch[edi,gi]  Unit
1254    enterprise_follow(edi,(cmi,cai,gi),...)((fpl,pfpl),(ddyn,img)) ≡
1265a     if fpl = ⟨⟩ then chaos else
1265b     let (v′,a′,o′,p′) = increment(dyn,img,hd fpl) in
1265c     while let (v″,a″,o″,p″) = attr_DYN_ch[odi]? in
1265c             ∼close(p′,p″) end do manoeuvre(hd fpl,dyn,img) ; wait δ t end ;
1265d     enterprise_drone(edi,(cmi,cai,gi),...)((tlfpl,pfpl),(attr_DYN_ch[odi]?,img))
1265a     end end
```

1266. The (overloaded) manoeuvre behaviour is further unspecified. For a fixed wing aircraft it controls the *yaw*, the *roll* and the *pitch* of the aircraft, hence its flight path, by operating the *elevator, aileron, ruddr* and the *thrust* of the aircraft based on its current dynamics, weight (including aircraft fuel), meteorological conditions (winds etc.).

```
value
1266    manoeuvre: FPE × DYN × ImG → Unit
1266    manoeuvre(fpe,dyn,img) ≡ ...
```

The **wait** δ t is some drone constant.

Geography Behaviour

1267. The *geography* behaviour definition
   a. lists the geography behaviour's unique identifier, carries the its mereology, has the static argument of its Euclidean point space, and
   b. further designates the single **in**put channels cp_g_ch[cpi,gi] from the *planner* and d_g_ch[*,gi] from the drones and the **out**put channels g_cp_ch[gi,cpi] to the *planner* and g_d_ch[gi,*] to the *drones*.

1268. The *geography* otherwise behaves as follows:
   a. Internal, non-deterministically the geography chooses to either "resp"ond to a request from the "plan"ner.
   b. If the choice is
   c. "resp_plan"

[25] **chaos** means that we simply decide not to describe what then happens !

i. then the *geography* offers to accept a request from the *planner* for the *immediate geography* of an *area* “around” a *point* and
ii. then the *geography* offers that information to the *planner*,
iii. whereupon the geography resumes being that;

else if the choice is

d. `"resp_dron"`
   i. then then the *geography* offers to accept a request from the *planner* for the *immediate geography* of an *area* “around” a *point* and
   ii. then the *geography* offers that information to the *planner*,
   iii. whereupon the geography resumes being that.

1269. The area function takes a pair of a point and a pair of *land* and *weather* and yields an *immediate geography*.

```
value
1267        geography: gi:GI × gm:(cpi:CPI×cmi:CMI×dis:DI-set) × EPS →
1267a           in cp_g_ch[cpi,gi], d_g_ch[∗,gi]
1267b           out g_cp_ch[gi,cpi], g_d_ch[gi,∗]  Unit
1267        geography(gi,(cpi,cmi,dis),eps) ≡
1268a          let mode = "resp_plan" ⌈⌉ "resp_dron" ⌈⌉ ... in
1268b          case mode of
1268c             "resp_plan" →
1268(c)i                let p = cp_g_ch[cpi,gi] ? in
1268(c)ii               g_cp_ch[gi,cpi] ! area(p,(attr_L_ch[gi]?,attr_W_ch[gi]?)) end
1268(c)iii              geography(gi,(cpi,cmi,dis),eps)
1268d             "resp_dron" →
1268(d)i                let (p,di) = ⌈⌉⌊⌋{(d_g_ch[di,gi]?,di)|di:DI•di ∈ dis} in
1268(d)ii               g_cp_ch[di,cpi] ! area(p,(attr_L_ch[gi]?,attr_W_ch[gi]?)) end
1268(d)iii              geography(gi,(cpi,cmi,dis),eps)
1267           end end
axiom
1267        gi=g_i∧cpi=cp_i∧smi=cm_i dis=dis
value
1269        area: P × (L × W) → ImG
1269        area(p,(l,w)) ≡ ...
```

F.5 Conclusion

TO BE WRITTEN

Part **VII**

RSL

G

An RSL Primer

This is an ultra-short introduction to the RAISE Specification Language, RSL.

G.1 Types

The reader is kindly asked to study first the decomposition of this section into its sub-parts and sub-sub-parts.

G.1.1 Type Expressions

Type expressions are expressions whose value are type, that is, possibly infinite sets of values (of "that" type).

Atomic Types

Atomic types have (atomic) values. That is, values which we consider to have no proper constituent (sub-)values, i.e., cannot, to us, be meaningfully "taken apart".

RSL has a number of *built-in* atomic types. There are the Booleans, integers, natural numbers, reals, characters, and texts.

type
[1] **Bool**
[2] **Int**
[3] **Nat**
[4] **Real**
[5] **Char**
[6] **Text**

Composite Types

Composite types have composite values. That is, values which we consider to have proper constituent (sub-)values, i.e., can, to us, be meaningfully "taken apart".

From these one can form type expressions: finite sets, infinite sets, Cartesian products, lists, maps, etc.

Let A, B and C be any type names or type expressions, then:

[7] A-**set**
[8] A-**infset**
[9] $A \times B \times \ldots \times C$
[10] A^*

[11] A^ω
[12] A $\overrightarrow{m}$ B
[13] A → B
[14] A $\xrightarrow{\sim}$ B
[15] (A)
[16] A | B | ... | C
[17] mk_id(sel_a:A,...,sel_b:B)
[18] sel_a:A ... sel_b:B

The following are generic type expressions:

1. The Boolean type of truth values **false** and **true**.
2. The integer type on integers ..., –2, –1, 0, 1, 2, ...
3. The natural number type of positive integer values 0, 1, 2, ...
4. The real number type of real values, i.e., values whose numerals can be written as an integer, followed by a period ("."), followed by a natural number (the fraction).
5. The character type of character values ″a″, ″bb″, ...
6. The text type of character string values ″aa″, ″aaa″, ..., ″abc″, ...
7. The set type of finite cardinality set values.
8. The set type of infinite and finite cardinality set values.
9. The Cartesian type of Cartesian values.
10. The list type of finite length list values.
11. The list type of infinite and finite length list values.
12. The map type of finite definition set map values.
13. The function type of total function values.
14. The function type of partial function values.
15. In (A) A is constrained to be:
    - either a Cartesian B × C × ... × D, in which case it is identical to type expression kind 9,
    - or not to be the name of a built-in type (cf., 1–6) or of a type, in which case the parentheses serve as simple delimiters, e.g., (A $\overrightarrow{m}$ B), or (A^*)**-set**, or (A**-set**)list, or (A|B) $\overrightarrow{m}$ (C|D|(E $\overrightarrow{m}$F)), etc.
16. The postulated disjoint union of types A, B, ..., and C.
17. The record type of mk_id-named record values mk_id(av,...,bv), where av, ..., bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.
18. The record type of unnamed record values (av,...,bv), where av, ..., bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.

G.1.2 Type Definitions

Concrete Types

Types can be concrete in which case the structure of the type is specified by type expressions:

type
  A = Type_expr

Some schematic type definitions are:

[1] Type_name = Type_expr /∗ without |s or subtypes ∗/
[2] Type_name = Type_expr_1 | Type_expr_2 | ... | Type_expr_n
[3] Type_name ==
        mk_id_1(s_a1:Type_name_a1,...,s_ai:Type_name_ai) |
        ... |
        mk_id_n(s_z1:Type_name_z1,...,s_zk:Type_name_zk)
[4] Type_name :: sel_a:Type_name_a ... sel_z:Type_name_z
[5] Type_name = {| v:Type_name′ • $\mathscr{P}$(v) |}

where a form of [2–3] is provided by combining the types:

```
Type_name = A | B | ... | Z
A == mk_id_1(s_a1:A_1,...,s_ai:A_i)
B == mk_id_2(s_b1:B_1,...,s_bj:B_j)
...
Z == mk_id_n(s_z1:Z_1,...,s_zk:Z_k)
```

Types A, B, ..., Z are disjoint, i.e., shares no values, provided all mk_id_k are distinct and due to the use of the disjoint record type constructor ==.

```
axiom
  ∀ a1:A_1, a2:A_2, ..., ai:Ai •
    s_a1(mk_id_1(a1,a2,...,ai))=a1 ∧ s_a2(mk_id_1(a1,a2,...,ai))=a2 ∧
    ... ∧ s_ai(mk_id_1(a1,a2,...,ai))=ai ∧
  ∀ a:A • let mk_id_1(a1′,a2′,...,ai′) = a in
    a1′ = s_a1(a) ∧ a2′ = s_a2(a) ∧ ... ∧ ai′ = s_ai(a) end
```

Subtypes

In RSL, each type represents a set of values. Such a set can be delimited by means of predicates. The set of values b which have type B and which satisfy the predicate $\mathscr{P}$, constitute the subtype A:

```
type
  A = {| b:B • 𝒫(b) |}
```

Sorts — Abstract Types

Types can be (abstract) sorts in which case their structure is not specified:

```
type
  A, B, ..., C
```

G.2 The RSL Predicate Calculus

G.2.1 Propositional Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values (**true** or **false** [or **chaos**]). Then:

```
false, true
a, b, ..., c ∼a, a∧b, a∨b, a⇒b, a=b, a≠b
```

are propositional expressions having Boolean values. ∼, ∧, ∨, ⇒, = and ≠ are Boolean connectives (i.e., operators). They can be read as: *not*, *and*, *or*, *if then* (or *implies*), *equal* and *not equal*.

G.2.2 Simple Predicate Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values, let x, y, ..., z (or term expressions) designate non-Boolean values and let i, j, ..., k designate number values, then:

false, **true**
a, b, ..., c
∼a, a∧b, a∨b, a⇒b, a=b, a≠b
x=y, x≠y,
i<j, i≤j, i≥j, i≠j, i≥j, i>j

are simple predicate expressions.

G.2.3 Quantified Expressions

Let X, Y, ..., C be type names or type expressions, and let $\mathscr{P}(x)$, $\mathscr{Q}(y)$ and $\mathscr{R}(z)$ designate predicate expressions in which x, y and z are free. Then:

∀ x:X • $\mathscr{P}(x)$
∃ y:Y • $\mathscr{Q}(y)$
∃ ! z:Z • $\mathscr{R}(z)$

are quantified expressions — also being predicate expressions.

They are "read" as: For all x (values in type X) the predicate $\mathscr{P}(x)$ holds; there exists (at least) one y (value in type Y) such that the predicate $\mathscr{Q}(y)$ holds; and there exists a unique z (value in type Z) such that the predicate $\mathscr{R}(z)$ holds.

G.3 Concrete RSL Types: Values and Operations

G.3.1 Arithmetic

type
  Nat, **Int**, **Real**
value
  +,−,∗: **Nat**×**Nat**→**Nat** | **Int**×**Int**→**Int** | **Real**×**Real**→**Real**
  /: **Nat**×**Nat**$\tilde{\to}$**Nat** | **Int**×**Int**$\tilde{\to}$**Int** | **Real**×**Real**$\tilde{\to}$**Real**
  <,≤,=,≠,≥,> (**Nat**|**Int**|**Real**) → (**Nat**|**Int**|**Real**)

G.3.2 Set Expressions

Set Enumerations

Let the below a's denote values of type A, then the below designate simple set enumerations:

{{}, {a}, {e_1,e_2,...,e_n}, ...} ∈ A-**set**
{{}, {a}, {e_1,e_2,...,e_n}, ..., {e_1,e_2,...}} ∈ A-**infset**

Set Comprehension

The expression, last line below, to the right of the ≡, expresses set comprehension. The expression "builds" the set of values satisfying the given predicate. It is abstract in the sense that it does not do so by following a concrete algorithm.

type
  A, B
  P = A → **Bool**
  Q = A $\stackrel{\sim}{\to}$ B
value
  comprehend: A-**infset** × P × Q → B-**infset**
  comprehend(s,P,Q) ≡ { Q(a) | a:A • a ∈ s ∧ P(a)}

G.3.3 Cartesian Expressions

Cartesian Enumerations

Let e range over values of Cartesian types involving A, B, ..., C, then the below expressions are simple Cartesian enumerations:

type
  A, B, ..., C
  A × B × ... × C
value
  (e1,e2,...,en)

G.3.4 List Expressions

List Enumerations

Let a range over values of type A, then the below expressions are simple list enumerations:

  {⟨⟩, ⟨e⟩, ..., ⟨e1,e2,...,en⟩, ...} ∈ A^*
  {⟨⟩, ⟨e⟩, ..., ⟨e1,e2,...,en⟩, ..., ⟨e1,e2,...,en,... ⟩, ...} ∈ A^ω

  ⟨ a_i .. a_j ⟩

The last line above assumes a_i and a_j to be integer-valued expressions. It then expresses the set of integers from the value of e_i to and including the value of e_j. If the latter is smaller than the former, then the list is empty.

List Comprehension

The last line below expresses list comprehension.

type
  A, B, P = A → **Bool**, Q = A $\stackrel{\sim}{\to}$ B
value
  comprehend: A^ω × P × Q $\stackrel{\sim}{\to}$ B^ω
  comprehend(l,P,Q) ≡
    ⟨ Q(l(i)) | i **in** ⟨1..**len** l⟩ • P(l(i))⟩

G.3.5 Map Expressions

Map Enumerations

Let (possibly indexed) *u* and *v* range over values of type *T*1 and *T*2, respectively, then the below expressions are simple map enumerations:

```
type
   T1, T2
   M = T1 →m T2
value
   u,u1,u2,...,un:T1, v,v1,v2,...,vn:T2
   [], [u↦v], ..., [u1↦v1,u2↦v2,...,un↦vn] ∀ ∈ M
```

Map Comprehension

The last line below expresses map comprehension:

```
type
   U, V, X, Y
   M = U →m V
   F = U ~→ X
   G = V ~→ Y
   P = U → Bool
value
   comprehend: M×F×G×P → (X →m Y)
   comprehend(m,F,G,P) ≡
      [ F(u) ↦ G(m(u)) | u:U • u ∈ dom m ∧ P(u)]
```

G.3.6 Set Operations

Set Operator Signatures

```
value
   19 ∈: A × A-infset → Bool
   20 ∉: A × A-infset → Bool
   21 ∪: A-infset × A-infset → A-infset
   22 ∪: (A-infset)-infset → A-infset
   23 ∩: A-infset × A-infset → A-infset
   24 ∩: (A-infset)-infset → A-infset
   25 \: A-infset × A-infset → A-infset
   26 ⊂: A-infset × A-infset → Bool
   27 ⊆: A-infset × A-infset → Bool
   28 =: A-infset × A-infset → Bool
   29 ≠: A-infset × A-infset → Bool
   30 card: A-infset ~→ Nat
```

Set Examples

examples

$a \in \{a,b,c\}$
$a \notin \{\}, a \notin \{b,c\}$
$\{a,b,c\} \cup \{a,b,d,e\} = \{a,b,c,d,e\}$
$\cup\{\{a\},\{a,bb\},\{a,d\}\} = \{a,b,d\}$
$\{a,b,c\} \cap \{c,d,e\} = \{c\}$
$\cap\{\{a\},\{a,bb\},\{a,d\}\} = \{a\}$
$\{a,b,c\} \setminus \{c,d\} = \{a,bb\}$
$\{a,bb\} \subset \{a,b,c\}$
$\{a,b,c\} \subseteq \{a,b,c\}$
$\{a,b,c\} = \{a,b,c\}$
$\{a,b,c\} \neq \{a,bb\}$
card $\{\} = 0$, **card** $\{a,b,c\} = 3$

Informal Explication

19. $\in$: The membership operator expresses that an element is a member of a set.
20. $\notin$: The nonmembership operator expresses that an element is not a member of a set.
21. $\cup$: The infix union operator. When applied to two sets, the operator gives the set whose members are in either or both of the two operand sets.
22. $\cup$: The distributed prefix union operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.
23. $\cap$: The infix intersection operator. When applied to two sets, the operator gives the set whose members are in both of the two operand sets.
24. $\cap$: The prefix distributed intersection operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.
25. $\setminus$: The set complement (or set subtraction) operator. When applied to two sets, the operator gives the set whose members are those of the left operand set which are not in the right operand set.
26. $\subseteq$: The proper subset operator expresses that all members of the left operand set are also in the right operand set.
27. $\subset$: The proper subset operator expresses that all members of the left operand set are also in the right operand set, and that the two sets are not identical.
28. $=$: The equal operator expresses that the two operand sets are identical.
29. $\neq$: The nonequal operator expresses that the two operand sets are *not* identical.
30. **card**: The cardinality operator gives the number of elements in a finite set.

Set Operator Definitions

The operations can be defined as follows ($\equiv$ is the definition symbol):

value

$s' \cup s'' \equiv \{ a \mid a:A \bullet a \in s' \vee a \in s'' \}$
$s' \cap s'' \equiv \{ a \mid a:A \bullet a \in s' \wedge a \in s'' \}$
$s' \setminus s'' \equiv \{ a \mid a:A \bullet a \in s' \wedge a \notin s'' \}$
$s' \subseteq s'' \equiv \forall a:A \bullet a \in s' \Rightarrow a \in s''$
$s' \subset s'' \equiv s' \subseteq s'' \wedge \exists a:A \bullet a \in s'' \wedge a \notin s'$
$s' = s'' \equiv \forall a:A \bullet a \in s' \equiv a \in s'' \equiv s \subseteq s' \wedge s' \subseteq s$
$s' \neq s'' \equiv s' \cap s'' \neq \{\}$
card $s \equiv$
  if $s = \{\}$ **then** 0 **else**
  let $a:A \bullet a \in s$ **in** $1 +$ **card** $(s \setminus \{a\})$ **end end**

```
    pre s /* is a finite set */
  card s ≡ chaos /* tests for infinity of s */
```

G.3.7 Cartesian Operations

```
type
  A, B, C
  g0: G0 = A × B × C
  g1: G1 = ( A × B × C )
  g2: G2 = ( A × B ) × C
  g3: G3 = A × ( B × C )

value
  va:A, vb:B, vc:C, vd:D
  (va,vb,vc):G0,
  (va,vb,vc):G1
  ((va,vb),vc):G2
  (va3,(vb3,vc3)):G3

decomposition expressions
  let (a1,b1,c1) = g0,
      (a1′,b1′,c1′) = g1 in .. end
  let ((a2,b2),c2) = g2 in .. end
  let (a3,(b3,c3)) = g3 in .. end
```

G.3.8 List Operations

List Operator Signatures

```
value
  hd: A^ω ~→ A
  tl: A^ω ~→ A^ω
  len: A^ω ~→ Nat
  inds: A^ω → Nat-infset
  elems: A^ω → A-infset
  .(.): A^ω × Nat ~→ A
  ^: A* × A^ω → A^ω
  =: A^ω × A^ω → Bool
  ≠: A^ω × A^ω → Bool
```

List Operation Examples

```
examples
  hd⟨a1,a2,...,am⟩=a1
  tl⟨a1,a2,...,am⟩=⟨a2,...,am⟩
  len⟨a1,a2,...,am⟩=m
  inds⟨a1,a2,...,am⟩={1,2,...,m}
  elems⟨a1,a2,...,am⟩={a1,a2,...,am}
  ⟨a1,a2,...,am⟩(i)=ai
  ⟨a,b,c⟩^⟨a,b,d⟩ = ⟨a,b,c,a,b,d⟩
  ⟨a,b,c⟩=⟨a,b,c⟩
  ⟨a,b,c⟩ ≠ ⟨a,b,d⟩
```

Informal Explication

- **hd**: Head gives the first element in a nonempty list.
- **tl**: Tail gives the remaining list of a nonempty list when Head is removed.
- **len**: Length gives the number of elements in a finite list.
- **inds**: Indices give the set of indices from 1 to the length of a nonempty list. For empty lists, this set is the empty set as well.
- **elems**: Elements gives the possibly infinite set of all distinct elements in a list.
- $\ell(i)$: Indexing with a natural number, i larger than 0, into a list ℓ having a number of elements larger than or equal to i, gives the ith element of the list.
- $\frown$: Concatenates two operand lists into one. The elements of the left operand list are followed by the elements of the right. The order with respect to each list is maintained.
- $=$: The equal operator expresses that the two operand lists are identical.
- $\neq$: The nonequal operator expresses that the two operand lists are *not* identical.

The operations can also be defined as follows:

List Operator Definitions

```
value
  is_finite_list: Aω → Bool

  len q ≡
    case is_finite_list(q) of
      true → if q = ⟨⟩ then 0 else 1 + len tl q end,
      false → chaos end

  inds q ≡
    case is_finite_list(q) of
      true → { i | i:Nat • 1 ≤ i ≤ len q },
      false → { i | i:Nat • i≠0 } end

  elems q ≡ { q(i) | i:Nat • i ∈ inds q }

  q(i) ≡
    if i=1
      then
        if q≠⟨⟩
          then let a:A,q′:Q • q=⟨a⟩⁀q′ in a end
          else chaos end
      else q(i−1) end

  fq ⁀ iq ≡
      ⟨ if 1 ≤ i ≤ len fq then fq(i) else iq(i − len fq) end
       | i:Nat • if len iq≠chaos then i ≤ len fq+len end ⟩
    pre is_finite_list(fq)

  iq′ = iq″ ≡
    inds iq′ = inds iq″ ∧ ∀ i:Nat • i ∈ inds iq′ ⇒ iq′(i) = iq″(i)

  iq′ ≠ iq″ ≡ ∼(iq′ = iq″)
```

G.3.9 Map Operations

Map Operator Signatures and Map Operation Examples

Map Operations

value
m(a): M → A $\xrightarrow{\sim}$ B, m(a) = b

dom: M → A-**infset** [domain of map]
dom [a1↦b1,a2↦b2,...,an↦bn] = {a1,a2,...,an}

rng: M → B-**infset** [range of map]
rng [a1↦b1,a2↦b2,...,an↦bn] = {b1,b2,...,bn}

†: M × M → M [override extension]
[a↦b,a′↦bb′,a″↦bb″] † [a′↦bb″,a″↦bb′] = [a↦b,a′↦bb″,a″↦bb′]

∪: M × M → M [merge ∪]
[a↦b,a′↦bb′,a″↦bb″] ∪ [a‴↦bb‴] = [a↦b,a′↦bb′,a″↦bb″,a‴↦bb‴]

\: M × A-**infset** → M [restriction by]
[a↦b,a′↦bb′,a″↦bb″]\{a} = [a′↦bb′,a″↦bb″]

/: M × A-**infset** → M [restriction to]
[a↦b,a′↦bb′,a″↦bb″]/{a′,a″} = [a′↦bb′,a″↦bb″]

=,≠: M × M → **Bool**

°: (A $\overrightarrow{m}$ B) × (B $\overrightarrow{m}$ C) → (A $\overrightarrow{m}$ C) [composition]
[a↦b,a′↦bb′] ° [bb↦c,bb′↦c′,bb″↦c″] = [a↦c,a′↦c′]

Map Operation Explication

- $m(a)$: Application gives the element that a maps to in the map m.
- **dom**: Domain/Definition Set gives the set of values which *maps to* in a map.
- **rng**: Range/Image Set gives the set of values which *are mapped to* in a map.
- †: Override/Extend. When applied to two operand maps, it gives the map which is like an override of the left operand map by all or some “pairings” of the right operand map.
- ∪: Merge. When applied to two operand maps, it gives a merge of these maps.
- \: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements that are not in the right operand set.
- /: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements of the right operand set.
- =: The equal operator expresses that the two operand maps are identical.
- ≠: The nonequal operator expresses that the two operand maps are *not* identical.
- °: Composition. When applied to two operand maps, it gives the map from definition set elements of the left operand map, m_1, to the range elements of the right operand map, m_2, such that if a is in the definition set of m_1 and maps into b, and if b is in the definition set of m_2 and maps into c, then a, in the composition, maps into c.

Map Operation Redefinitions

The map operations can also be defined as follows:

value
  rng m ≡ { m(a) | a:A • a ∈ **dom** m }

  m1 † m2 ≡
    [a↦b | a:A,b:B •
      a ∈ **dom** m1 \ **dom** m2 ∧ bb=m1(a) ∨ a ∈ **dom** m2 ∧ bb=m2(a)]

  m1 ∪ m2 ≡ [a↦b | a:A,b:B •
        a ∈ **dom** m1 ∧ bb=m1(a) ∨ a ∈ **dom** m2 ∧ bb=m2(a)]

  m \ s ≡ [a↦m(a) | a:A • a ∈ **dom** m \ s]
  m / s ≡ [a↦m(a) | a:A • a ∈ **dom** m ∩ s]

  m1 = m2 ≡
    dom m1 = **dom** m2 ∧ ∀ a:A • a ∈ **dom** m1 ⇒ m1(a) = m2(a)
  m1 ≠ m2 ≡ ∼(m1 = m2)

  m°n ≡
    [a↦c | a:A,c:C • a ∈ **dom** m ∧ c = n(m(a))]
    pre rng m ⊆ **dom** n

G.4 λ-Calculus + Functions

G.4.1 The λ-Calculus Syntax

type /∗ A BNF Syntax: ∗/
  ⟨L⟩ ::= ⟨V⟩ | ⟨F⟩ | ⟨A⟩ | (⟨A⟩)
  ⟨V⟩ ::= /∗ variables, i.e. identifiers ∗/
  ⟨F⟩ ::= λ⟨V⟩ • ⟨L⟩
  ⟨A⟩ ::= (⟨L⟩⟨L⟩)
value /∗ Examples ∗/
  ⟨L⟩: e, f, a, ...
  ⟨V⟩: x, ...
  ⟨F⟩: λ x • e, ...
  ⟨A⟩: f a, (f a), f(a), (f)(a), ...

G.4.2 Free and Bound Variables

Let x, y be variable names and e, f be λ-expressions.

- ⟨V⟩: Variable x is free in x.
- ⟨F⟩: x is free in $\lambda y \bullet e$ if $x \neq y$ and x is free in e.
- ⟨A⟩: x is free in $f(e)$ if it is free in either f or e (i.e., also in both).

G.4.3 Substitution

In RSL, the following rules for substitution apply:

- **subst**([N/x]x) ≡ N;
- **subst**([N/x]a) ≡ a,
  for all variables a≠ x;
- **subst**([N/x](P Q)) ≡ (**subst**([N/x]P) **subst**([N/x]Q));
- **subst**([N/x](λx•P)) ≡ λ y•P;
- **subst**([N/x](λ y•P)) ≡ λy• **subst**([N/x]P),
  if x≠y and y is not free in N or x is not free in P;
- **subst**([N/x](λy•P)) ≡ λz•**subst**([N/z]**subst**([z/y]P)),
  if y≠x and y is free in N and x is free in P
  (where z is not free in (N P)).

G.4.4 α-Renaming and β-Reduction

- α-renaming: λx•M
  If x, y are distinct variables then replacing x by y in λx•M results in λy•**subst**([y/x]M). We can rename the formal parameter of a λ-function expression provided that no free variables of its body M thereby become bound.
- β-reduction: (λx•M)(N)
  All free occurrences of x in M are replaced by the expression N provided that no free variables of N thereby become bound in the result. (λx•M)(N) ≡ **subst**([N/x]M)

G.4.5 Function Signatures

For sorts we may want to postulate some functions:

```
type
   A, B, C
value
   obs_B: A → B,
   obs_C: A → C,
   gen_A: BB×C → A
```

G.4.6 Function Definitions

Functions can be defined explicitly:

```
value
   f: Arguments → Result
   f(args) ≡ DValueExpr

   g: Arguments ~→ Result
   g(args) ≡ ValueAndStateChangeClause
   pre P(args)
```

Or functions can be defined implicitly:

value
f: Arguments $\rightarrow$ Result
f(args) **as** result
post P1(args,result)

g: Arguments $\stackrel{\sim}{\rightarrow}$ Result
g(args) **as** result
pre P2(args)
post P3(args,result)

The symbol $\stackrel{\sim}{\rightarrow}$ indicates that the function is partial and thus not defined for all arguments. Partial functions should be assisted by preconditions stating the criteria for arguments to be meaningful to the function.

G.5 Other Applicative Expressions

G.5.1 Simple let Expressions

Simple (i.e., nonrecursive) **let** expressions:

let a = $\mathscr{E}_d$ **in** $\mathscr{E}_b$(a) **end**

is an "expanded" form of:

$(\lambda$a.$\mathscr{E}_b$(a))($\mathscr{E}_d$)

G.5.2 Recursive let Expressions

Recursive **let** expressions are written as:

let f = λa:A • E(f) **in** B(f,a) **end**

is "the same" as:

let f = **Y**F **in** B(f,a) **end**

where:

F $\equiv$ λg•λa•(E(g)) and YF = F(YF)

G.5.3 Predicative let Expressions

Predicative **let** expressions:

let a:A • $\mathscr{P}$(a) **in** $\mathscr{B}$(a) **end**

express the selection of a value a of type A which satisfies a predicate $\mathscr{P}$(a) for evaluation in the body $\mathscr{B}$(a).

G.5.4 Pattern and "Wild Card" let Expressions

Patterns and *wild cards* can be used:

let {a} ∪ s = set **in** ... **end**
let {a,_} ∪ s = set **in** ... **end**

let (a,b,...,c) = cart **in** ... **end**
let (a,_,...,c) = cart **in** ... **end**

let ⟨a⟩⁀ℓ = list **in** ... **end**
let ⟨a,_,bb⟩⁀ℓ = list **in** ... **end**

let [a↦bb] ∪ m = map **in** ... **end**
let [a↦b,_] ∪ m = map **in** ... **end**

G.5.5 Conditionals

Various kinds of conditional expressions are offered by RSL:

if b_expr **then** c_expr **else** a_expr
end

if b_expr **then** c_expr **end** ≡ /∗ same as: ∗/
  if b_expr **then** c_expr **else skip end**

if b_expr_1 **then** c_expr_1
elsif b_expr_2 **then** c_expr_2
elsif b_expr_3 **then** c_expr_3
...
elsif b_expr_n **then** c_expr_n **end**

case expr **of**
  choice_pattern_1 → expr_1,
  choice_pattern_2 → expr_2,
  ...
  choice_pattern_n_or_wild_card → expr_n
end

G.5.6 Operator/Operand Expressions

⟨Expr⟩ ::=
  ⟨Prefix_Op⟩ ⟨Expr⟩
  | ⟨Expr⟩ ⟨Infix_Op⟩ ⟨Expr⟩
  | ⟨Expr⟩ ⟨Suffix_Op⟩
  | ...
⟨Prefix_Op⟩ ::=
  − | ∼ | ∪ | ∩ | **card** | **len** | **inds** | **elems** | **hd** | **tl** | **dom** | **rng**
⟨Infix_Op⟩ ::=
  = | ≠ | ≡ | + | − | ∗ | ↑ | / | < | ≤ | ≥ | > | ∧ | ∨ | ⇒
  | ∈ | ∉ | ∪ | ∩ | \ | ⊂ | ⊆ | ⊇ | ⊃ | ⁀ | † | °
⟨Suffix_Op⟩ ::= !

G.6 Imperative Constructs

G.6.1 Statements and State Changes

Often, following the RAISE method, software development starts with highly abstract-applicative constructs which, through stages of refinements, are turned into concrete and imperative constructs. Imperative constructs are thus inevitable in RSL.

```
    Unit
value
    stmt: Unit → Unit
    stmt()
```

- Statements accept no arguments.
- Statement execution changes the state (of declared variables).
- **Unit** → **Unit** designates a function from states to states.
- Statements, stmt, denote state-to-state changing functions.
- Writing () as "only" arguments to a function "means" that () is an argument of type **Unit**.

G.6.2 Variables and Assignment

```
0. variable v:Type := expression
1. v := expr
```

G.6.3 Statement Sequences and skip

Sequencing is expressed using the ';' operator. **skip** is the empty statement having no value or side-effect.

```
2. skip
3. stm_1;stm_2;...;stm_n
```

G.6.4 Imperative Conditionals

```
4. if expr then stm_c else stm_a end
5. case e of: p_1→S_1(p_1),...,p_n→S_n(p_n) end
```

G.6.5 Iterative Conditionals

```
6. while expr do stm end
7. do stmt until expr end
```

G.6.6 Iterative Sequencing

```
8. for e in list_expr • P(b) do S(b) end
```


G.7 Process Constructs

G.7.1 Process Channels

Let A and B stand for two types of (channel) messages and i:KIdx for channel array indexes, then:

channel c:A
channel { k[i]:B • i:Idx }
channel { k[i,j,...,k]:B • i:Idx,j:Jdx,...,k:Kdx }

declare a channel, c, and a set (an array) of channels, k[i], capable of communicating values of the designated types (A and B).

G.7.2 Process Composition

Let P and Q stand for names of process functions, i.e., of functions which express willingness to engage in input and/or output events, thereby communicating over declared channels. Let P() and Q stand for process expressions, then:

P ‖ Q Parallel composition
P ⌈⌉⌊⌋ Q Nondeterministic external choice (either/or)
P ⌈⌉ Q Nondeterministic internal choice (either/or)
P ⫲ Q Interlock parallel composition

express the parallel (‖) of two processes, or the nondeterministic choice between two processes: either external (⌈⌉⌊⌋) or internal (⌈⌉). The interlock (⫲) composition expresses that the two processes are forced to communicate only with one another, until one of them terminates.

G.7.3 Input/Output Events

Let c, k[i] and e designate channels of type A and B, then:

c ?, k[i] ? Input
c ! e, k[i] ! e Output

expresses the willingness of a process to engage in an event that "reads" an input, respectively "writes" an output.

G.7.4 Process Definitions

The below signatures are just examples. They emphasise that process functions must somehow express, in their signature, via which channels they wish to engage in input and output events.

value
P: **Unit** → **in** c **out** k[i]
Unit
Q: i:KIdx → **out** c **in** k[i] **Unit**

P() ≡ ... c ? ... k[i] ! e ...
Q(i) ≡ ... k[i] ? ... c ! e ...

The process function definitions (i.e., their bodies) express possible events.

G.8 Simple RSL Specifications

Often, we do not want to encapsulate small specifications in schemes, classes, and objects, as is often done in RSL. An RSL specification is simply a sequence of one or more types, values (including functions), variables, channels and axioms:

type
 ...
variable
 ...
channel
 ...
value
 ...
axiom
 ...

Part **VIII**

Indexes

H

Indexes

H.1 General Monograph Indexes

H.1.1 Definitions

H.1.2 Concepts

H.1.3 Examples

H.1.4 Analysis Prompts

H.1.5 Description Prompts

H.1.6 Attribute Categories

H.1.7 Philosophy Index

H.2 Formal Domain Model Indexes

This index covers the RSL formulas of the Example, Sect. 1.8 of Chapter 1, and Case Studies E (Urban Planning, Pages 311–364) and F (Swarms of Drones, Pages 365–398).

H.2.1 Sorts

H.2.2 Sort Observers

TO BE WRITTEN

H.2.3 Types

H.2.4 Functions

H.2.5 Channels

H.2.6 Behaviours

H.2.7 Global Values

H.3 RSL Symbolds