Domain Science & Engineering. A Foundation for Software Development

Bjørner, Dines

Publication date:
2020

Document Version
Publisher's PDF, also known as Version of record

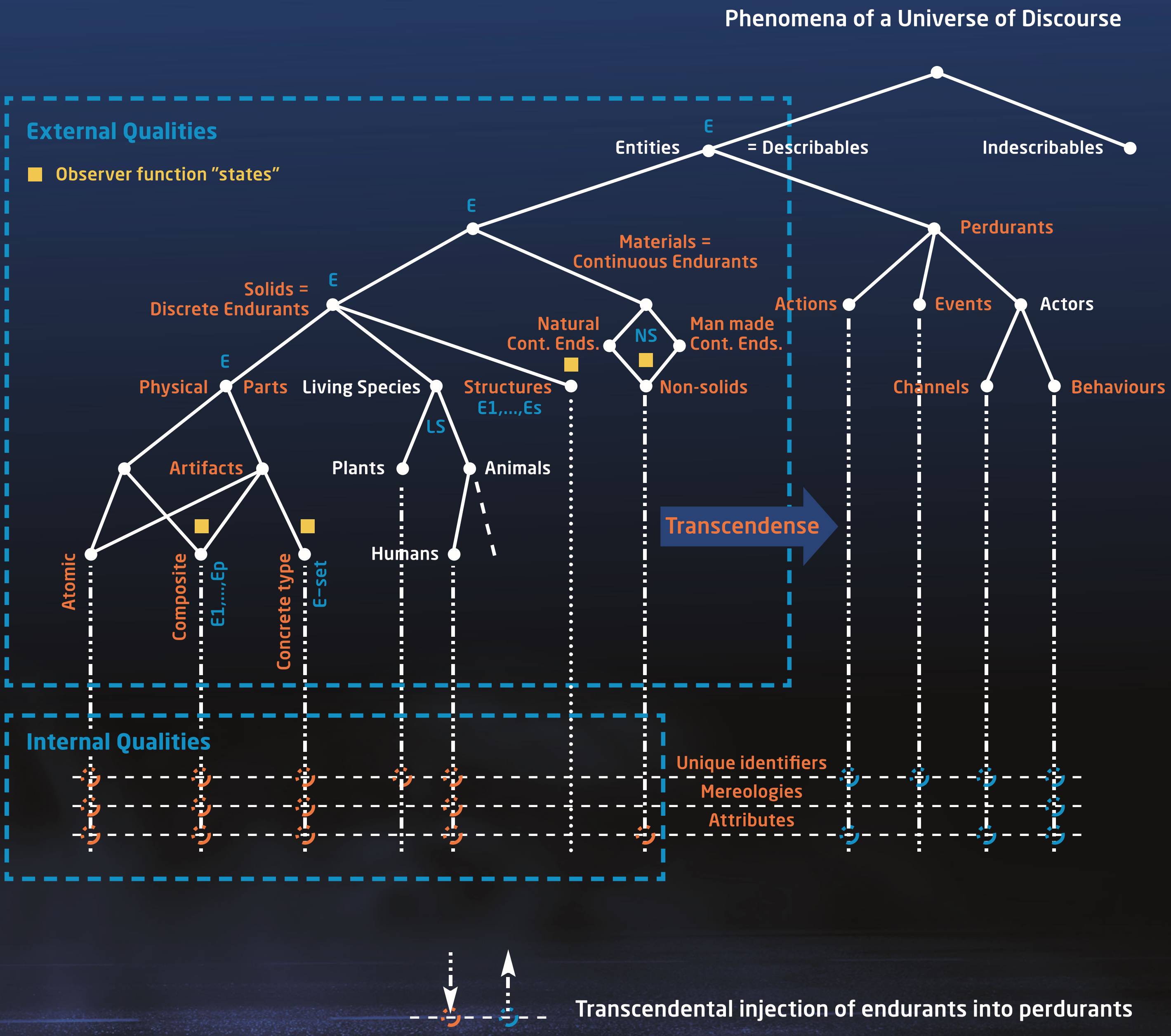
Citation (APA):
Bjørner, D. (2020). *Domain Science & Engineering. A Foundation for Software Development*. Technical University of Denmark.

DTU Compute
Department of
Applied Mathematics
and Computer Science

Domain Science & Engineering
A Foundation for Software Development

Department of Applied Mathematics and Computer Science

Dines Bjørner
Dr. Techn. Thesis 2020



Domain Science & Engineering
A Foundation for Software Development
By Dines Bjørner

A Dr. Techn. Thesis Submission for The Technical University of Denmark
September 6, 2019: 10:51am
Printed 2020

E-mail: bjorner@gmail.com
URL: www.imm.dtu.dk/~dibj

Published by DTU, Department of Applied Mathematics and Computer Science,
Richard Petersens Plads, Building 324, DK-2800 Kgs. Lyngby, Denmark
www.dtu.dk

ISBN 978-87-643-2002-2 (Printed version)
ISBN 978-87-643-2003-9 (Electronic version)

DTU

Denne afhandling er af Danmarks Tekniske Universitet antaget til forsvar for den tekniske doktorgrad. Antagelsen er sket efter bedømmelse af den foreliggende afhandling.

Kgs. Lyngby, den 28. september 2020

Anders O. Bjarklev

Rektor

Rasmus Larsen

Prorektor

This thesis has been accepted by the Technical University of Denmark for public defence in fulfilment of the requirements for the degree of Doctor Technices. The acceptance is based on an evaluation of the present dissertation.

Kgs. Lyngby, 28. september 2020

Anders O. Bjarklev
President

Rasmus Larsen
Provost

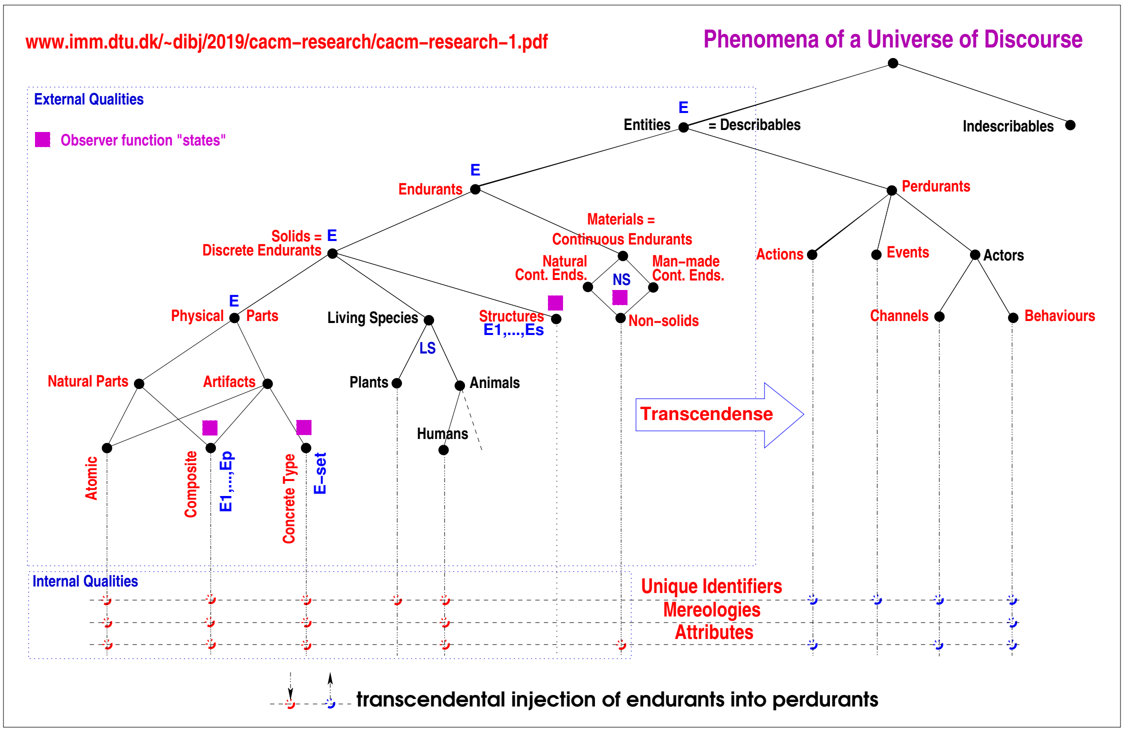
Dines Bjørner

Domain Science & Engineering

A Foundation for Software Development

A Dr. Techn. Thesis Submission for The Technical University of Denmark

September 6, 2019: 16:27



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

Dr.techn. Tesen

Domæne Videnskab og Ingeniørkunst

En Basis for Programmeludvikling

Afhandlingens Tese

Denne afhandlings tese er to-foldig:

- (i) *domæne videnskab og ingeniørkunst* tilbyder en mulig, første fase af programmel-udvikling;
- (ii) *domæne videnskab og ingeniørkunst* tilbyder et værdigt forsknings-felt.

Vi motiverer denne dobbelte påstand som følger:

- (a) begreberne *domæne videnskab* og *domæne ingeniørkunst* er nye;
- (b) begreberne *domæne videnskab* og *domæne ingeniørkunst* gives en veldefineret betydning;
- (c) *domæne videnskab* og *domæne ingeniørkunst* tilskrives et fundament in denne tese;
- (d) og deres rolle i programmel-udvikling etableres.

Domæne videnskab og domæne ingeniørkunst får programmel-udvikling til at fremstaa i et helt nyt lys.

Dansk Resumé

Afhandlingen[1] består af revisioner af seks tidligere publicerede artikler. Afhandingen fremstilles og indsendes som en monografi.

Kapitel 1, *Domain Analysis & Description* [*Domæne Analyse og Beskrivelse*], er afhandlingens kerne-punkt. Her studeres kalkyler til brug ved analyse og beskrivelse af observérbare statiske entiteter (kaldet ‘enduranter’) i manifestérbare, diskret dynamiske domæner. Til ‘enduranter’ knyttes der tre klasser “interne” kvaliteter: eentyding identification, mereologi og attributter. Ved en transcendental deduktion knyttes tids-bestemte processer (kaldet ‘perduranter’) til diskrete enduranter — og det vises hvorledes disse processer’s signatur følger fra ‘interne’ endurant kvaliteter.

Kapitel 2, *Domain Facets: Analysis & Description* [*Domæne Facetter*], studerer yderligere klasser af egenskaber ved manifestérbare, diskret dynamiske domæner: teknologi-understøttelse, regler og regulativer, manuskripter, licens sprog, ledelse & organisation, og menneskelig opførsel.

Kapitel 3, *Towards Formal Models of [Analysis & Description] Processes and Prompts* [Formelle Modeller af Analyse & Beskrivelse], præsenterer en rimelig matematisk model af domæne-analyse & -beskrivelses processen.

Kapitel 4, *To Every Manifest Domain Mereology [there corresponds] a CSP Expression* [*Til enhver Manifestérbar Domæne Mereologi korrensponderes et CSP Udtryk*], viser hvorledes Stanisław Leśniewski’s axiom system for mereologi har en model i den måde vi (f.eks., i Kapitel 1) beskriver sådanne domæner.

Kapitel 5, *From Domain Descriptions to Requirements Prescriptions* [*Fra Domæne-Beskrivelser til Krav-Specifikationer*], viser hvorledes man rigorøst kan aflede krav-specifikationer fra domænebeskrivelser. Der indføres, som noget nyt, det at *projicere, instantiere, determinere, udvide* og *“tilpasse”* domæne/krav-specifikationer. Såkaldte “bruger-” og “system”-krav “erstattes” af *grænseflade-*, *afledte-* og *maskin-krav*.

Kapitel 6, *Demos, Simulators, Monitors and Controllers* [“Demo”er, Simulatorer, Overvågning og Styring], udgør en “let” sag, en fortolkning af de muligheder de indbyrdes afhængigheder, som trekløveret *domæne-*, *krav-*, hhv. *program*-specifikationer, muliggør ved fortolkningen af begreberne simulering, overvågning og styring.

Kapitel 7, *Summing Up* [Opsummering], opsummerer og konkluderer.

[1] Siderne i–vi + 1–214 + bibliografi

The Dr.techn. Thesis

Domain Science and Engineering

A Foundation for Software Development

The Thesis

The thesis of this submission is twofold:

- (i) *domain science & engineering* is a possible, initial phase of software development;
- (ii) *domain science & engineering* is a worthwhile topic of research.

We support this claim as follows:

- (a) the concepts of *domain science* and *domain engineering* are new;
- (b) the terms *domain science* and *domain engineering* are well-defined;
- (c) *domain science* and *domain engineering* are given a foundation in this thesis;
- (d) and their rôle in software development is established.

Domain Science & Engineering casts a completely new light on Software Development.

English Summary

The thesis[2] consists of revisions of six previously published papers — presented and submitted as a monograph.

Chapter 1, *Domain Analysis & Description*, is the core chapter of the thesis. It studies calculi for the analysis and description of observable static entities (called ‘endurants’) of manifest, discrete dynamics domains. With endurants we then associate “internal” qualities: unique identification, mereology and attributes. By a transcendental deduction we then associate time-evolving behaviours with discrete endurants — and it is shown how the signature of such behaviours follows from ‘internal’ qualities of the discrete endurants.

Chapter 2, *Domain Facets: Analysis & Description*, studies further classes of properties of manifest, discrete dynamics domains: *technology support*, *rules & regulations*, *scripts*, *license languages*, *management & organisation*, and *human behaviour*.

Chapter 3, *Formal Models of [Analysis & Description] Processes and Prompts*, presents a suitable mathematical model of the domain analysis & description process.

Chapter 4, *To Every Manifest Domain Mereology [there corresponds] a CSP Expression*, shows how Stanisław Leśniewski’s axiom system for mereology has a model in the way we (f.ex., in Chapter 1) describe such domains.

Chapter 5, *From Domain Descriptions to Requirements Prescriptions*, shows how one can rigorously derive requirements prescriptions from domain descriptions. We introduce, as new concepts, *projection*, *instantiation*, *determination*, *extension*, and the *“fitting”* of domain/requirements specifications. So-called “user” and “system” requirements are replaced by *interface*, *derived* and *machine requirements.*

Chapter 6, *Demos*, *Simulators*, *Monitors and Controllers*, is a somewhat “lightweight” chapter: an interpretation of the of the mutual dependencies, which the triptych — of *domain*, *requirements* and *program* specifications — makes possible in the interpretation of the concepts of simulation, monitoring and control.

Chapter 7, *Summing Up*, summarises and concludes.

BJØRNER

Dines Bjørner, September 6, 2019: 16:27
Fredsvej 11, DK–2840 Holte, Denmark

[2] Pages i–vi, 1–214 + bibliography.

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 monograph is twofold: (i) that domain engineering is a viable, yes, we would claim, necessary initial phase of software development; and (ii) that domain science & engineering is a worthwhile topic of research. 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 thesis 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 this thesis: 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 chapters, 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

Six (twelve) revised publications (in early, published, and in a recent, only in two cases, republished, form) are collected in this thesis:

• Chapter 1: [75, 67, Domains Analysis & Description]	Pages 3–74
• Chapter 2: [71, 43, Domain Facets: Analysis & Description]	Pages 75–103
• Chapter 3: [61, 57, Formal Models of Processes and Prompts]	Pages 105–127
• Chapter 4: [73, 39, To Every Manifest Domain Mereology a CSP Expression]	Pages 129–149
• Chapter 5: [63, 33, From Domain Descriptions to Requirements Prescriptions]	Pages 153–198
• Chapter 6: [62, 50, Domains: Their Simulation, Monitoring and Control]	Pages 201–210

We urge the reader to study the **Contents** listing and from there to learn that there is a

- **Bibliography** common to all six chapters,
- **an RSL primer**, and
- a set of **Indexes** into definitions, concepts, analysis and description prompts, and RSL symbols.

Chapter Dependency

Chapters 1–6 relate as diagrammed:

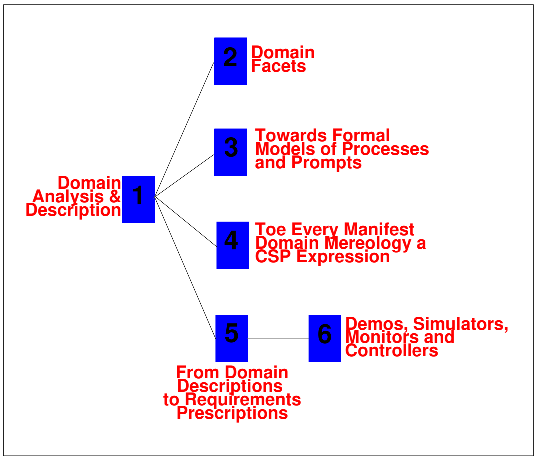

1
Domain Analysis & Description
2
Domain Facets
3
Towards Formal Models of Processes and Prompts
4
Toe Every Manifest Domain Mereology a CSP Expression
5
From Domain Descriptions to Requirements Prescriptions
6
Demos, Simulators, Monitors and Controllers

Fig. 0.1. Chapter Dependency

- Chapters 2–5 all require Chapter 1 as a prerequsite, and can be read in any order.
- Chapter 6 require Chapter 5 as a prerequsite.

Common Frame

By a **method** we shall mean a set of **principles** of **analysis** and for **selecting** and **applying** a number of **techniques** and **tools** in the construction of some artefact, say a domain description. We shall present a method for constructing domain descriptions. Among the tools we shall only be concerned with are the analysis and synthesis languages.

By a **formal method** we shall understand a method whose **techniques** and **tools** can be explained in **mathematics.** If, for example, the method includes a **specification language**, then that language has a **formal syntax**, a **formal semantics**, and a **formal proof system**. The **techniques** of a formal method help **construct** a specification, and/or **analyse** a specification, and/or **transform** (**refine**) one (or more) specification(s) into a program.

By **computer science** we shall understand the study of and knowledge about the mathematical structures that "exists inside" computers.

By **computing science** we shall understand the study of and knowledge about how to construct those structures. The term **programming methodology** is here used synonymously with computing science. Software engineering is the actual pursuit of software development based primarily on computing science insight.

By **engineering** we shall understand the design of technology based on scientific insight and the analysis of technology in order to assess its properties (including scientific content) and practical applications.

Software engineering, to us, ideally entails the **engineering** of **domain descriptions** ($\mathscr{D}$), the **engineering** of **requirements prescriptions** ($\mathscr{R}$), the **engineering** of **software designs** [and code] ($\mathscr{S}$), and the engineering of informal and formal relations between domain descriptions and requirements prescriptions: **$\mathscr{R}$ is a model of $\mathscr{D}$**, and domain descriptions, requirements prescriptions and software designs: **$\mathscr{S}$ can be proved correct with respect to $\mathscr{R}$ in the context of $\mathscr{D}$**.

Contents

Last page: Page 267

Part I

The Domain Analysis & Description Method

1

Domain Analysis & Description

We[1] present a *method* for **analysing and describing manifest (discrete dynamics) domains.**

1.1 Introduction

By a **domain** we shall understand a **rationally describable** segment of a **discrete dynamics** segment of a **human assisted** reality, i.e., of the world, its **physical parts: natural** ["God-given"] and **artifactual** ["man-made"], and **living species**: **plants** and **animals** including, notably, **humans**. These are **endurants** ("still"), as well as **perdurants** ("alive"). Emphasis is placed on **"human-assistedness"**, that is, that there is *at least one (man-made)* **artifact** and, therefore, that **humans** are a primary cause for change of endurant **states** as well as perdurant **behaviours**.

Please note the 'delimiter': *discrete dynamics*. Control theory, the study of the control of continuously operating dynamical systems in engineered processes and machines, is one thing; domain engineering is "a different thing". Where control theory builds upon classical physics, and uses classical mathematics, partial differential equations, etc., to model phenomena of physics and therefrom engineered 'machines'; domain science & engineering, in some contrast, builds upon mathematical logic, and, to some extent, modern algebra, to model phenomena of mostly artefactual systems.

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 the *natural physical world*, 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 tools: the physics models are based on *classical mathematics*, *differential equations* and *integrals*, etc.; our models are based on *mathematical logic*, *set theory*, and *algebra*.

Where physicists thus classically use a variety of *differential* and *integral calculi* to model the physical world, we shall be using the *analysis & description calculi* presented in this chapter to model primarily artefactual domains.

1.1.1 Foreword

Dear reader! You are about to embark on a journey. The chapter in front of you is long ! But it is not the number of pages, 74, or duration of your studying the chapter 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

[1] Chapter 1 is primarily based on [75]. That paper was based on [67]. Section 1.5.2's Part Relations is changed wrt. [75, Sect. 4.2.1]. Section 9.7 of [75] has here been replaced by Sect. 1.10.7 which is taken from [67]. Remaining editing changes are of syntactics art.

& 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, we immodestly claim, with 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, some "almost" formal, 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.

1.1.2 An Engineering and a Science Viewpoint

A Triptych of Software Development

It seems reasonable to expect that before **software** can be designed we must have a reasonable grasp of its **requirements**; before **requirements** can be expressed we must have a reasonable grasp of the underlying **domain**. It therefore seems reasonable to structure software development into: **domain engineering**, in which "the underlying" domain is *analysed and described*[2]; **requirements engineering**, in which requirements are *analysed and prescribed* – such as we suggest it [33, 63] – based on a domain description[3]; and **software design**, in which the software is *rigorously "derived"* from a requirements prescription[4]. Our interest, in this chapter, lies sôlely in domain analysis & description.

Domain Science & Engineering:

The present chapter outlines a *methodology* for an aspect of software development. Domain analysis & description can be pursued in isolation, for example, without any consideration of any other aspect of software development. As such domain analysis & description represents an aspect of **domain science & engineering**. Other aspects are covered in: Chap. 2 [71, *Domain Facets*], Chap. 3 [61, *An Analysis & Description Process Model*], Chap. 4 [73, *From Mereologies to Lambda-Expressions*], Chap. 5 [63, *Requirements Engineering*], and in Chap. 6 [62, 50, Domains: Their Simulation, Monitoring and Control]. This work is over-viewed in [72, Domain Science & Engineering – A Review of 10 Years Work]. They are all facets of an emerging **domain science & engineering**. *We consider the present chapter to outline the basis for this science and engineering.*

1.1.3 Some Issues: Metaphysics, Epistemology, Mereology and Ontology

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

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

Traditional metaphysics seeks to answer, in a "suitably abstract and fully general manner", the questions: *What is there ?* and *And what is it like ?*[6]. Topics of metaphysical investigation include existence, objects and their properties, space and time, cause and effect, and possibility.

Epistemology is the branch of philosophy concerned with the theory of knowledge[7] ■

[2] including the statement and possible proofs of properties of that which is denoted by the domain description

[3] including the statement and possible proofs of properties of that which is denoted by the requirements prescription with respect also to the domain description

[4] including the statement and possible proofs of properties of that which is specified by the software design with respect to both the requirements prescription and the domain description

[5] ■ is used to signal the end of a characterisation, a definition, or an example.

[6] https://en.wikipedia.org/wiki/Metaphysics

[7] https://en.wikipedia.org/wiki/Epistemology

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*.[8]

Ontology: 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.[9]. 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[10] ■

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 [96][11] ■

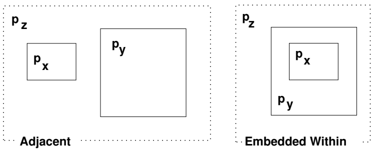


Fig. 1.1. Immediately 'Adjacent' and 'Embedded Within' Parts

Accordingly two parts, p_x and p_y, (of a same "whole") are either "adjacent", or are "embedded within", one within the other, as loosely indicated in Fig. 1.1. 'Adjacent' parts are direct parts of a same third part, p_z, i.e., p_x and p_y are "embedded within" 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" p_z; et cetera; as loosely indicated in Fig. 1.2, or one is "embedded within" the other — etc. as loosely indicated in Fig. 1.2. Parts, whether 'adjacent' or 'embedded within', can share properties. For adjacent 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. 1.3 on the next page. Instead of depicting parts sharing properties as in Fig. 1.3 on the following page[L]eft, where shaded, dashed rounded-edge rectangles stands for 'sharing', we shall (eventually) show parts sharing properties as in Fig. 1.3 on the next page[R]ight where •—• connections connect those parts.

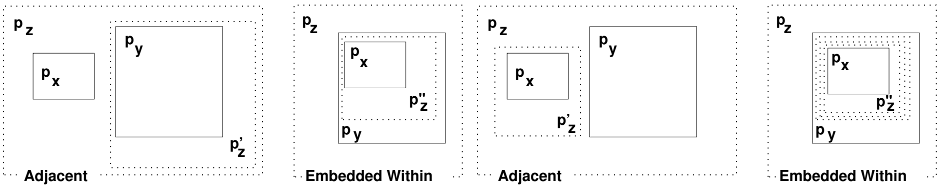


Fig. 1.2. Transitively 'Adjacent' and 'Embedded Within' Parts

We refer to [73, *From Mereologies to Lambda-Expressions*].

Mereology is basically the contribution [159, 224] of the Polish philosopher, logician and mathematician Stanisław Leśniewski (1886–1939).

[8] https://en.wikipedia.org/wiki/Metaphysics

[9] https://en.wikipeda.org/wiki/On-tology_(information_science)

[10] https://en.wikipedia.org/wiki/Upper_ontology

[11] https://plato.stanford.edu/entries/mereology

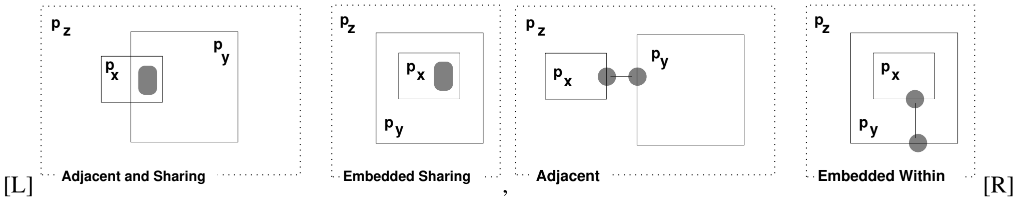


Fig. 1.3. Two models, [L,R], of parts sharing properties

Kai Sørlander's Philosophy:

We shall base some of our modelling decisions of Kai Sørlander's Philosophy [219, 220, 221, 222]. A main contribution of Kai 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**.

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 Kai 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 for the analysis & description of domains and that a number of relations between domain entities can be understood transcendentally and as "variants" of laws of physics, biology, etc. !

1.1.4 The Precursor

The present chapter is based on a revision of the published [67] – as published in [75, April 2019]. The revision considerably simplifies and considerably extends the domain analysis & description calculi of [67]. The major revision that prompts this complete rewrite is due to a serious study of Kai Sørlander's Philosophy. As a result we extend [67]'s ontology of endurants: describable phenomena 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 paper and [67] are the result of extensive "non-toy" example case studies, see the example: *Universes of Discourse* – on Page 9. The last half of these were carried out in the years since [67] was first submitted (i.e., 2014). The present paper omits the extensive introduction[12] and closing of [67, Sects. 1 and 5]. Most notably, however, is a clarified view on the transition from **parts** to **behaviours**, a **transcendental deduction** from *domain space* to *domain time*.

1.1.5 What is this Chapter About ?

We present a *method* for *analysing &*[13] *describing domains*.

Definition 1 **Domain:** By a **domain** we shall understand a **rationally describable** segment of a **discrete dynamics** segment of a **human assisted** reality, i.e., of the world, its **physical parts**, **natural** ["God-given"] and **artefactual** ["man-made"], and **living species**: **plants** and **animals** including, predominantly,

[12] **Note added in proof:** Omitted from the extensive, five page, literature survey of [67] was [102, Section 5.3]. It is an interesting study of the domain of geography.

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

humans. These are **endurants** ("still") as well as **perdurants** ("alive"). Emphasis is placed on **"human-assistedness"**, that is, that there is *at least one (man-made)* **artefact** and that **humans** are a primary cause for change of endurant **states** as well as perdurant **behaviours** ■

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*[14] can be studied separately, for 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.*

1.1.6 Structure of this Chapter

Sections 1.2–1.8 form the core of this chapter. 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*, *materials*, *living species* and *artefacts*. In doing so it covers the *external quality analysis prompts*. Section 1.4 covers the *external quality 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 quality analysis prompts* and the *internal quality description prompts*. Sections 1.3–1.5 cover what this chapter has to say about *endurants*. Section 1.6 "bridges" Sects. 1.3–1.5 and Sect. 1.8 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. 1.6–1.8 is to *"translate"* endurant parts into perdurant behaviours. Section 1.8 – 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 [137]. Hence we introduce the CSP notions of *channels* and channel *input/output*. Section 1.8 then "derives" the types of the behaviour arguments from the internal endurant qualities. Section 1.10 summarises the achievements and discusses open issues. Section 1.10.2 on Page 64 summarises the four languages used in this chapter.

Framed texts either delineate major figures, so-called *observer* and *behaviour* schemes.

One major example, that of the domain analysis & description of a road transport system, intersperses the methodology presentation of 38 examples. Section 1.9 completes that road transport system example.

1.2 Entities: Endurants and Perdurants

1.2.1 A Generic Domain Ontology – A Synopsis

Figure 1.4 on the next page shows an *upper ontology* for domains such a defined in Defn. 1 on the facing page.

Kai Sørlander's Philosophy justifies our organising the *entities* of any describable domain, for example[15], as follows: We shall review Fig. 1.4 on the next page by means of a top-down, left-traversal of the

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

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

tree (whose root is at the top). 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 *continuous*

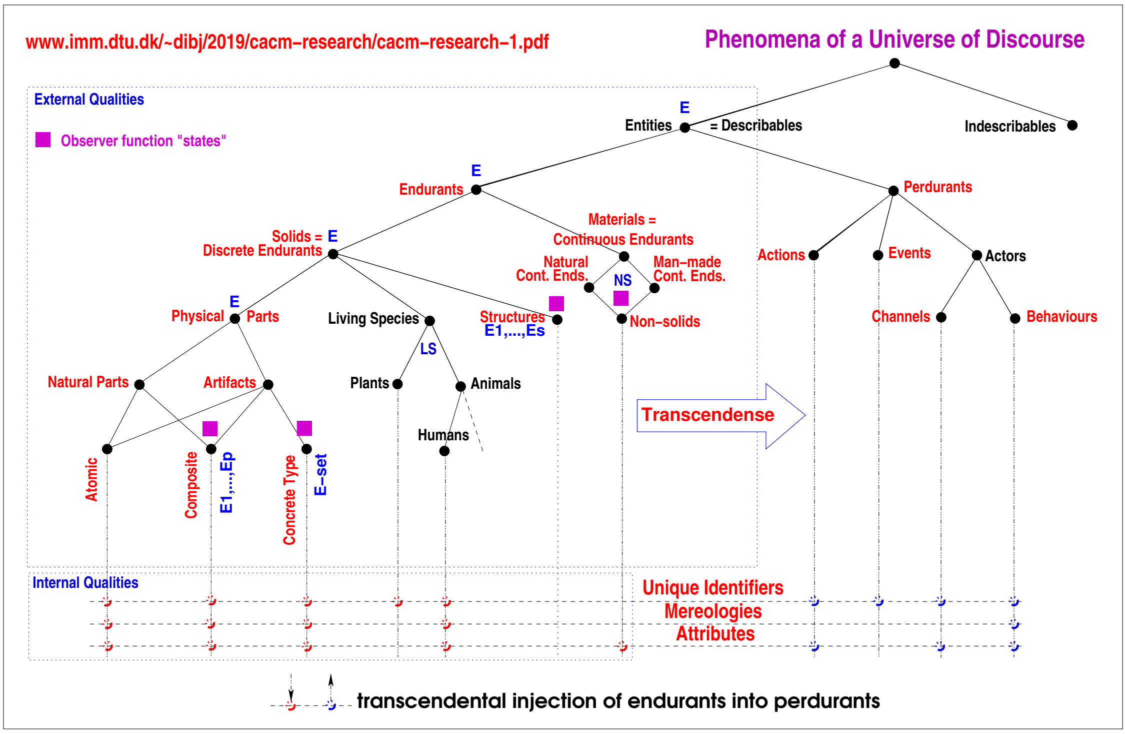


Fig. 1.4. An Upper Ontology for Domains

– in which latter case we call them *materials*[16]. *Discrete* endurants are *physical parts*, *living species*, or are *structures*. *Physical parts* are either *naturals*, *artefacts*, i.e. man-made. Natural and man-made parts are either *atomic* or *composite*. We additionally analyse artefacts into *sets* of identically *typed parts*. That additional analysis could also be expressed for natural parts but as we presently find no use for that we omit such further analysis. *Living Species* are either *plants* or *animals*. Among animals we have the *humans*. *Structures* consist of one or more endurants. Structures really are parts, but for pragmatic reasons we choose to not model them as [full fledged] parts. The categorisation into structures, natural parts, artefactual parts, plants, and animalsis thus partly based in Sørlander's Philosophy, partly pragmatic. The distinction between endurants and perdurants, are necessitated by Sørlander as being in space, respectively in space **and** time; discrete and continuous are motivated by arguments of natural sciences; structures are purely pragmatic; plants and animals, including humans, are necessitated by Kai Sørlander's Philosophy. The distinction between natural, physical parts, and artefacts is not necessary in Sørlander's Philosophy, but, we claim, necessary, philosophically, in order to perform the *intentional "pull"*, a transcendental deduction.

On Pragmatics: We have used the term 'pragmatic' a few times. On one hand there is philosophy's need for absolute clarity. On the other hand, when applying the natural part, artefactual part, and living species, concepts in practice, there can be a need for "loosening" up. As for example: a structure really is a collection of parts and relations between them. As we shall later see, parts are transcendentally to be understood

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

as behaviours. We know that modelling is imperative when we model a domain, but we may not wish to model a discrete endurant as a behaviour so we decide, pragmatically, to model it as a structure.

Our reference, here, to Kai Sørlander's Philosophy, is very terse. We refer to a detailed research report: *A Philosophy of Domain Science & Engineering*[17] 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 Philosophy.

1.2.2 Universes of Discourse

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

Example 1: Universes of Discourse

We refer to a number of Internet accessible experimental reports[18] *of descriptions of the following domains:*

- **railways** *[23, 81, 26],*
- **container shipping** *[31],*
- **stock exchange** *[46],*
- **oil pipelines** *[53],*
- **"The Market"** *[24],*
- **Web systems** *[45],*
- **weather information** *[64],*
- **credit card systems** *[60],*
- **document systems** *[68],*
- **urban planning** *[88],*
- **swarms of drones** *[66],*
- **container terminals** *[70]*

It may be a **"large" domain**, that is, consist of many, as we shall see, *endurants* and *perdurants*, of many *parts* and *materials*, of many *humans* and *artefacts*, 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 modelling*, 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.

Example 2: Universe of Discourse

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, (nodes, i.e., street intersections) and links, L, (edges, i.e., 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.

[17] http://www.imm.dtu.dk/˜dibj/2018/philosophy/filo.pdf

1.2.3 Entities

Characterisation 1 **Entity:** 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 *conceived* 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* [156, Vol. I, pg. 665] ■

Analysis Prompt 1 *is_entity: The domain analyser analyses "things" (θ) into entities or non-entities. The method provides the* **domain analysis prompt***:*

- *is_entity – where* `is_entity(`θ`)` *holds if θ is an entity*[19] ■

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

To sum up: *An entity is what we can analyse and describe using the analysis & description prompts outlined in this chapter.*

The *entities* that we are concerned with are those with which Kai Sørlander's Philosophy is likewise concerned. They are the ones that are *unavoidable* in any description of any possible world. And then, which are those entities ? In both [219] and [222] Kai Sørlander 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 Sects. 1.3.4 on Page 15 and 1.5.3 on Page 32.

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 performs operations on, i.e. processes data, we shall, however, introduce these two notions: *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"* [156, Vol. I, pg. 656]. Were we to "freeze" time we would still be able to observe the entire endurant ■

Example 3: Endurants

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.* **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 2 *is_endurant: The domain analyser analyses an entity, ϕ, into an endurant as prompted by the* **domain analysis prompt***:*

- *is_endurant – ϕ is an endurant if* `is_endurant(`ϕ`)` *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. 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 [156, Vol. II, pg. 1552] ■

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

Example 4: Perdurants

Geography: *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.* **Railway Systems:** *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 3 `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` ■

Occurrent is a synonym for 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 ■

To simplify matters we shall allow separate elements of a discrete endurant to be continuous !

Example 5: Discrete Endurants

The individual endurants of the above example of railway system endurants 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 4 `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. 1.3.5 on Page 16.

Example 6: 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 5 `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

We analyse discrete endurants into *physical parts*, *living species* and *structures*. Physical parts and living species can be identified as separate entities – following Kai Sørlander's Philosophy. To model discrete endurants as structures represent a pragmatic choice which relieves the domain describer from transcendentally considering structures as behaviours.

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*[20] ■

Analysis Prompt 6 `is_physical_part`: *The domain analyser analyses "things" (η) into physical part. The method provides 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.

Living Species

Definition 3 Living Species, I: By a *living species* we shall understand a discrete endurant, subject to laws of physics, and additionally subject to *causality of purpose*.[21] [Defn. 9 on Page 15 elaborates further on this point] ■

Analysis Prompt 7 `is_living_species`: *The domain analyser analyses "things" (e) into living species. The method provides 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 [69] for details. Section 1.3.4 continues our treatment of living species.

Structures

Definition 4 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

[20] This characterisation is the result of our study of relations between philosophy and computing science, notably influenced by Kai Sørlander's Philosophy. We refer to our research report [69, www.imm.dtu.dk/˜dibj/2018/philosophy/-filo.pdf].

[21] See Footnote 20.

interested only in the (external and internal) qualities of the elements of the compound, but not in the qualities of the structure itself.

Example 7: Structures

A transport system is modelled as 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.

Example 8: Structures – Contd.

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. 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*. When we choose that an endurant sort should be modelled as a part sort with unique identification, mereology and proper attributes, then it is because we eventually shall consider the part sort as being the basis for transcendentally deduced behaviours.

Analysis Prompt 8 *`is_structure`: The domain analyser analyse endurants, e, into structure entities as prompted by the* **domain analysis prompt***:*

- *`is_structure` e is a structure if `is_structure(e)` holds* ■

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 *materials* (Sect. 1.3.5) and *artefacts* (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.

1.3.3 Physical Parts

Physical parts are either *natural parts*, or *sets of parts* of the same type, or are *artefacts* i.e. man-made parts. The categorisation of physical parts into these four is pragmatic. *Physical parts* follow from Kai Sørlander's Philosophy. *Natural parts* are what Sørlander's Philosophy is initially about. *Artefacts* follow from *humans* acting according to their *purpose* in making "physical 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 5 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 ■

Artefacts

Characterisation 8 Man-made Parts: Artefacts: Artefacts 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. Artefacts are subject to the *laws of physics* ■

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

Definition 6 Artefact: By an artefact 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 *artefacts* all come with an *attribute* of kind *intent*, that is, a set of purposes for which the artefact was constructed, and for which it is intended to serve. We continue our treatment of artefacts in Sect. 1.3.6 below.

Parts

We revert to our treatment of parts.

Example 9: Parts

The geography examples (of Page 10) of are all natural parts. The railway system examples (of Page 10) are all artefacts ■

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

Analysis Prompt 9 `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 artefactual 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 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 structure. If the domain engineer wishes to investigate the details of a discrete endurant then the domain engineer chooses to model[22] the discrete endurant as a part.

Atomic Parts

Definition 7 Atomic Part: Atomic parts 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 10 `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* ■

[22] We use the term *to model* interchangeably with the composite term *to analyse & describe*; similarly *a model* is used interchangeably with *an analysis & description*.

Example 10: Atomic Road Net Parts

From one point of view all of the following can be considered atomic parts: hubs, links[23]*, and automobiles.*

Composite Parts

Definition 8 Composite Part: Composite parts are those which, in a given context, are deemed to *indeed* consist of meaningful, separately observable proper *sub-parts* ■

Analysis Prompt 11 *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 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 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 12) of the concept of *living species*[24]: a discrete endurant existing in time, subject to laws of physics, and additionally subject to *causality of purpose*[25]

Definition 9 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** ■

Analysis Prompt 12 *is_living_species: The domain analyser analyse discrete endurants, ℓ, into living species entities as prompted by the* **domain analysis prompt***:*

- *is_living_species – where is_living_speciesℓ holds if ℓ is a living species* ■

Plants

We start with some examples.

Example 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 13 *is_plant: The domain analyser analyses "things" (ℓ) into a plant. The method provides 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).

[23] Hub ≡ street intersection; link ≡ street segments with no intervening hubs.
[24] See analysis prompt 7 on Page 12.
[25] See Footnote 20 on Page 12.

Animals

Definition 10 **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 14 *is_animal: The domain analyser analyses "things" (ℓ) into an animal. The method provides 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 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 chapter, whether to model animals singly or as sets[26] of such.

Humans

Definition 11 **Human:** A *human* (a *person*) is an *animal*, cf. Definition 10, with the additional properties of having *language*, being *conscious* of *having knowledge* (of its own situation), and *responsibility* ■

Analysis Prompt 15 *is_human: The domain analyser analyses "things" (ℓ) into a human. The method provides 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 [69, Sects. 10.4–10.5] for a specific treatment of living species, animals and humans, and to [69] 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 artefacts[27], that is, that these artefacts 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 !

1.3.5 Continuous Endurants ≡ Materials

Definition 12 **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. 1.4.4 on Page 22). So an endurant can (itself) "be" a set of materials. But physical parts may contain (has_materials) materials: natural parts may contain natural materials, artefacts may contain

[26] school of dolphins, flock of geese, herd of cattle, pack of dogs, pride of lions, swarm of flies,

[27] Cf. Sect. 1.3.6 below.

natural and artefactual materials. We leave it to the reader to provide analysis predicates for natural and artefactual "materials".

Example 14: Natural and Man-made Materials

A **natural part***, say a land area, may contain lakes, rivers, irrigation dams and border seas.*
An **artefact***, say an automobile, usually contains gasoline, lubrication oil, engine cooler liquid and window screen washer water.*

Analysis Prompt 16 `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. 1.4.4 on Page 22 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.

1.3.6 Artefacts

Definition 13 Artefacts: By artefacts we shall understand a man-made physical part or a man-made material ■

Example 15: More Artefacts

From the shipping industry: ship, container vessels, container, container stack, container terminal port, harbour.

Analysis Prompt 17 `is_artefact`: *The domain analyser analyses "things" (p) into artefacts. The method provides the* **domain analysis prompt***:*

- `is_artefact` *– where* `is_artefact(p)` *holds if p is an artefact* ■

1.3.7 States

Definition 14 State: By a *state* we shall understand any number of physical parts and/or materials each possessing as we shall later introduce them at least one dynamic attribute. There is no need to introduce time at this point ■

Example 16: Artefactual 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. 1.8. We shall there exemplify states further: example *Constants and States [Indexed States]* Page 41.

1.4 Endurants: The Description Calculus

1.4.1 Parts: Natural or Man-made

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

On Discovering Endurant Sorts

This section is not really relevant for the "discovery" of artefacts. Artefacts are man-made. Usually the designers – the engineers, the craftsmen – who make these parts start out by ascribing specific names to them. And these names become our sort names. So the α,β,γ points below are more relevant for the analysis of natural discrete endurants.

Our aim now is to present the basic principles that let the domain analyser decide on *part sorts*. 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*[28]*, 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. If p is a part of sort P, then we express that as: p:P.

Analysis Prompt 18 *observe_endurant_sorts: The* **domain analysis prompt***:*

- *observe_endurant_sorts*

directs the domain analyser to observe the sub-endurants of an endurant e and to suggest their sorts. Let *observe_endurant_sorts(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 artefacts (man-made parts) or structures, 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 and materials, and parts "down" to their atomicity. Materials are considered "atomic", i.e., to not contain further analysable endurants. For the structures, parts (whether natural or man-made) 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 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 ($1 \leq i \leq \mathsf{m}$) denotes disjoint sets of entities we must prove it.

[28] We use the term 'sort' for abstract types, i.e., for the type of values whose concrete form we are not describing. The term 'sort' is commonly used in algebraic semantics [212].

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` Observer schema

Narration:

[s] ... narrative text on sorts ...
[o] ... narrative text on sort observers ...
[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**_E_i: E → E_i i:[1..m]
proof obligation [Disjointness of endurant sorts]
[p] $\mathscr{PO}: \forall\, e{:}(E_1|E_2|...|E_m) \bullet \bigwedge \{\mathbf{is\_}E_i(e) \equiv \bigwedge \{\sim\mathbf{is\_}E_j(e)|j{:}[1..m] \setminus \{i\}\}|i{:}[1..m]\}$

The **is**_E_j(e) is defined by E_i i:[1..m]. `is_composite` is a **prerequisite prompt** of `observe_endurant_sorts`. That is, the composite may satisfy `is_natural` or `is_artefact` ■

Note: The above schema as well as the following schemes introduce, i.e., define in terms of a function signature, a number of functions whose names begin with bold-faced **obs_..., uid_..., mereo_..., attr_...** et cetera. These observer functions are one of the bases of domain descriptions.

In any specific domain analysis & description the analyser cum describer chooses which subset of composite sorts to analyse & describe. That is: any one domain model emphasises certain aspects and leaves out many "other" aspects. This means that there may be many different domain descriptions covering "more-or-less" similar ground !

We do not here state techniques for discharging proof obligations.[29]

Example 17: Composite Endurant Sorts

1 *There is the universe of discourse, UoD.*

It is structured into

2 *a road net, RN, and*
3 *a fleet of vehicles, FV.*

Both are structures.

type
1 UoD **axiom** ∀ uod:UoD • is_structure(uod).
2 RN **axiom** ∀ rn:RN • is_structure(rn).
3 FV **axiom** ∀ fv:FV • is_structure(fv).
value
2 obs_RN: UoD → RN
3 obs_FV: UoD → FV ■

[29] – such techniques are given in standard texts on formal specification languages.

Note: A proper description has two texts, a *narrative* and a *formalisation* each is itemised and items are pairwise numbered.

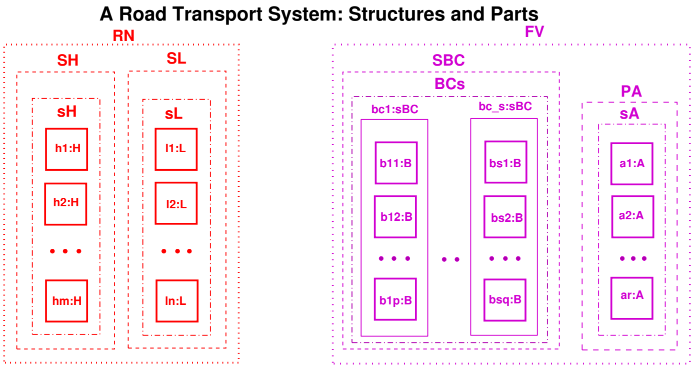


Fig. 1.5. A Road Transport System ■

Example 18: Structures

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

```
type
4a  SH   axiom ∀ sh:SH • is_structure(sh)
4b  SL   axiom ∀ sl:SL • is_structure(sl)
5a  SBC  axiom ∀ sbc:SBC • is_structure(bc)
5b  PA   axiom ∀ pa:PA • is_structure(pa)
value
4a  obs_SH: RN → SH
4b  obs_SL: RN → SL
5a  obs_BC: FV → BC
5b  obs_PA: FV → PA
```

1.4.2 Concrete Part Types

Sometimes it is expedient to ascribe concrete types to sorts.

Analysis Prompt 19 `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)*[30]

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 **Observer** schema

Narration:

[t₁] ... narrative text on sorts and types S_i ...
[t₂] ... narrative text on types T ...
[o] ... narrative text on type observers ...

Formalisation:

type
[t₁] S_1, S_2, ..., S_m, ..., S_n,
[t₂] $T = \mathscr{E}(S_1,S_2,...,S_n)$
value
[o] **obs_**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 (i.e., sort) expressions.[31] 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.

Example 19: Concrete Part Types

6 *The structure of hubs is a set, sH, of atomic hubs, H.*
7 *The structure of links is a set, sL, of atomic links, L.*
8 *The structure of buses is a set, sBC, of composite bus companies, BC.*
9 *The composite bus companies, BC, are sets of buses, sB.*
10 *The structure of private automobiles is a set, sA, of atomic automobiles, A.*

6 H, sH = **H-set** **axiom** ∀ h:H • is_atomic(h)
7 L, sL = **L-set** **axiom** ∀ l:L • is_atomic(l)
8 BC, BCs = **BC-set** **axiom** ∀ bc:BC • is_composite(bc)
9 B, Bs = **B-set** **axiom** ∀ b:B • is_atomic(b)
10 A, sA = **A-set** **axiom** ∀ a:A • is_atomic(a)
value
6 obs_sH: SH → sH
7 obs_sL: SL → sL
8 obs_sBC: SBC → BCs
9 obs_Bs: BCs → Bs
10 obs_sA: SA → sA ■

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

[31] T=**A-set** or $T=A^*$ or $T=ID \xrightarrow[m]{} A$ 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.

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 ![32,33]

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. That is, it is as if there is a reservoir of an indefinite-size set of such names from which these names are selected, and once obtained are never again selected. 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 Materials

We refer to Sect. 1.3.5 on Page 16 for our initial treatment of 'materials'. Continuous endurants (i.e., **material**s) are entities, m, which satisfy:

- `is_material`(e) $\equiv$ `is_continuous`(e)

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

Domain Description Prompt 3 *`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:

3. `observe_material_sorts` Observer schema

Narration:

[32] 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 ?

[33] At an IFIP WG2.2 meeting in Kyoto, August 1978, John McCarthy [164, 165], "waking up" from deep thoughts, asked, in connection with my presentation of abstract models of various database models [157], *"is there any recursion in all this ?"*, to which I replied, *"No !* – whereupon he resumed his interrupted thoughts/

[s] ... narrative text on material sorts ...
[o] ... narrative text on material sort observers ...
[p] ... narrative text on material sort proof obligations ...

Formalisation:

type
[s] M1, M2, ..., Mn
[s] M = M1 | M2 | ... | Mn
[s] MS = M-**set**
value
[o] **obs**_M$_i$: P → M, [i:1..n]
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]}

The **is**_M$_j$(e) is defined by Mi, 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* ■

Example 20: Materials

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

11 Links may contain material.
12 That material is water, W.

type
12 W
value
11 obs_material: L → W
11 **pre**: obs_material(l) ≡ has_material(h) ■

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

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 identifiers*, (ii) that *unique identifiers* (of parts and components p:P) are *abstract values* (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, 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 4 *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:*

4. observe_unique_identifier Observer schema

Narration:

```
    [s]  ... narrative text on unique identifier sort PI ...
    [u]  ... narrative text on unique identifier observer uid_P ...
    [a]  ... axiom on uniqueness of unique identifiers ...
```

Formalisation:

```
    type
    [s]  PI
    value
    [u]  uid_P: P → PI
    axiom [Disjointness of Domain Identifier Types]
    [a]  𝒜: 𝒰(PI,PI_i,PI_j,...,PI_k)
```

Example 21: Unique Identifiers

13 *We assign unique identifiers to all parts.*
14 *By a road identifier we shall mean a link or a hub identifier.*
15 *By a vehicle identifier we shall mean a bus or an automobile identifier.*
16 *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 buses of all bus companies have distinct identifiers.*
  e *All automobiles have distinct identifiers.*
  f *All parts have distinct identifiers.*

```
type
13   H_UI, L_UI, BC_UI, B_UI, A_UI
14   R_UI = H_UI | L_UI
15   V_UI = B_UI | A_UI
value
16a   uid_H: H → H_UI
16b   uid_L: H → L_UI
16c   uid_BC: H → BC_UI
16d   uid_B: H → B_UI
16e   uid_A: H → A_UI
```

Section 1.9.1 on Page 56 presents some auxiliary functions related to unique identifiers ■

We ascribe, in principle, unique identifiers to all parts whether natural or artefactual, and to all components. We find, from our many experiments, cf. the *Universes of Discourse* example, Page 9, that we really focus on those domain entities which are artefactual endurants and their behavioural "counterparts".

1.5.2 Mereology

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 [96, 55].

Part Relations:

Which are the relations that can be relevant for part-hood ? There are basically two relations: (i) a physical one, and (ii) a conceptual one.

(i) Physically two or more parts may be topologically either adjacent to one another, like rails of a line, or within a part, like links and hubs of a road net.

(ii) Conceptually some parts, like automobiles, "belong" to an embedding part, like to an automobile club, or are registered in the local department of vehicles, or are 'intended' to drive on roads

Part Mereology: Types and Functions

Analysis Prompt 20 `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 types* and *mereology observers* (i.e., part relations).

17 We may, to illustration, define a **mereology type** of a part $p{:}P$ as a triplet type expression over set of unique [part] identifiers.
18 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 $p{:}P$.
19 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.
20 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 $P_{o_1}, P_{o_2}, ..., P_{o_o}$.
21 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"`.

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

Domain Description Prompt 5 *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:

5. observe_mereology Observer 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[34]
    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* ■

Example 22: Mereology

22 *The mereology of hubs is a pair: (i) the set of all bus and automobile identifiers*[35]*, and (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).*[36]*.*

23 *The mereology of links is a pair: (i) the set of all bus and automobile identifiers, and (ii) the set of the two distinct hubs they are connected to.*

24 *The mereology of of a bus company is a set the unique identifiers of the buses operated by that company.*

25 *The mereology of a bus is a pair: (i) the set of the one single unique identifier of the bus company it is operating for, and (ii) the unique identifiers of all links and hubs*[37]*.*

26 *The mereology of an automobile is the set of the unique identifiers of all links and hubs*[38]*.*

```
type
22   H_Mer = V_UI-set×L_UI-set
22     axiom ∀ (vuis,luis):H_Mer • luis⊆$l_{ui}s$ ∧ vuis=$v_{ui}s$
23   L_Mer = V_UI-set×H_UI-set
23     axiom ∀ (vuis,huis):L_Mer •
23               vuis=$v_{ui}s$ ∧ huis⊆$h_{ui}s$ ∧ cardhuis=2
24   BC_Mer = B_UI-set
24     axiom ∀ buis:H_Mer • buis = $b_{ui}s$
25   B_Mer = BC_UI×R_UI-set
25     axiom ∀ (bc_ui,ruis):H_Mer•bc_ui∈$bc_{ui}s$∧ruis=$r_{ui}s$
26   A_Mer = R_UI-set
26     axiom ∀ ruis:A_Mer • ruis=$r_{ui}s$
value
22   mereo_H: H → H_Mer
23   mereo_L: L → L_Mer
24   mereo_BC: BC → BC_Mer
25   mereo_B: B → B_Mer
```

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

```
26  mereo_A: A → A_Mer
```

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

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

axiom

```
27  ∀ h:H,l:L • h ∈ hs ∧ l ∈ ls ⇒
        let (_,luis)=mereo_H(h), (_,huis)=mereo_L(l)
28      in 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 ■

Formulation of Mereologies:

The observe_mereology domain descriptor, Page 26, 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 [58] we present a model of one form of evaluation of the TripTych analysis and description prompts, see also Sect. 1.10.2 on Page 66.

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 on Page 9, we have found no really interesting such cases. All our experimental case studies appears to focus on the mereology of artefacts. And, finally, in modelling humans, we find that their mereology encompass all other humans and all artefacts ! Humans cannot be tamed to refrain from interacting with everyone and everything.

Some domain models may emphasize *physical mereologies* based on spatial relations, others may emphasize *conceptual mereologies* based on logical "connections".

1.5.3 Attributes

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[39], or

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

seen[40], but can be objectively measured[41]. 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 21 *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 attribute_types(p) = $\{A_1,A_2,...,A_m\}$.

Whether by attribute_types(p) we mean the names of the types $\{A_1, A_2, ..., A_m\}$ for example $\{\eta A_1, \eta A_2, ..., \eta A_m\}$ where η is some meta-function which applies to a type and yields its name, or or we mean the [full] types themselves, i.e., some possibly infinite, suitably structured set of values (of that type), we shall here leave open !

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 6 *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:*

6. observe_attributes Observer 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_i $[1 \leq i \leq m]$
value
[o] **attr**_A_i: P→A_i i:[1..m]
proof obligation [Disjointness of Attribute Types]
[p] $\mathscr{PO}$: **let** P be any part sort **in** [the domain description]

[40] 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.

[41] 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.

[p] **let** a:(A_1|A_2|...|A_m) **in** **is**_A_i(a) $\neq$ **is**_A_j(a) **end end** [i≠i, i,j:[1..m]]

The **is**_A_j(e) is defined by Ai, i:[1..n].

Let A_1, A_2, ..., A_n be the set of all conceivable attributes of parts *p*:*P*. (Usually *n* is a rather large natural number, say in the order of a hundred conceivable such.) In any one domain model the domain analyser cum describer selects a modest subset, A_1, A_2, ..., A_m, i.e., $m < n$. Across many domain models for *"more-or-less the same"* domain *m* varies and the attributes, A_1, A_2, ..., A_m, selected for one model may differ from those, A'_1, A'_2, ..., $A'_{m'}$, chosen for another model.

The **type** definitions: A_1, A_2, ..., A_m, inform us that the domain analyser has decided to focus on the distinctly named A_1, A_2, ..., A_m attributes.[42] 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 a part sort denote disjoint sets of values. Therefore we must prove it.

Attribute Categories: Michael A. Jackson [144] 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 [144, 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.

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 a dynamic attribute whose values, 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 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 1.6 on the following page captures an attribute value ontology.

[42] 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.

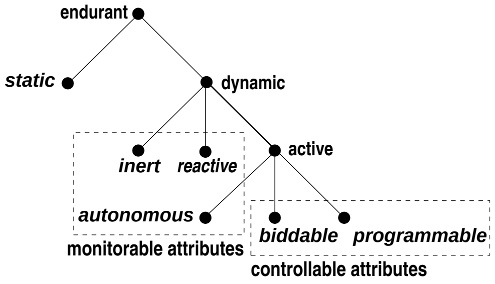


Fig. 1.6. Attribute Value Ontology

Example 23: Attributes

We treat part attributes, sort by sort. **Hubs:** We show just a few attributes:

29 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.

30 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.

31 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.

32 The link identifiers of hub states must be in the set, $l_{ui}s$, of the road net's link identifiers.

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

Attributes for remaining sorts are shown in Sect. 1.9.2 on Page 57.

Calculating Attributes:

33 Given a part p we can *meta-linguistically*[43] calculate names for its static attributes.
34 Given a part p we can *meta-linguistically* calculate name for its monitorable attributes attributes.
35 Given a part p we can *meta-linguistically* calculate name for its monitorable and controllable attributes.
36 Given a part p we can *meta-linguistically* calculate names for its controllable attributes.
37 These three sets make up all the attributes of part p.

The type names nSA, nMA nMCA, nCA designate sets of names.

```
value
33   stat_attr_typs: P → nSA-set
34   mon_attr_typs: P → nMA-set
35   mon_ctrl_attr_typs: P → nMCA-set
36   ctrl_attr_typs:  P → nCA-set

axiom
37   ∀ p:P •
33   let stat_nms = stat_attr_typs(p),
34       mon_nms = mon_attr_typs(p),
35       mon_ctrl_nms = mon_ctrl_attr_typs(p),
36       ctrl_nms = mon_ctrl_typs(p) in
37   card stat_nms + card mon_nms + card mon_ctrl_nms + card ctrl_nms
37   = card(stat_nms ∪ mon_nms ∪ mon_ctrl_nms ∪ ctrl_nms) end
```

The above formulas are indicative, like mathematical formulas, they are not computable.

38 Given a part p we can *meta-linguistically* calculate its static attribute values.
39 Given a part p we can *meta-linguistically* calculate its controllable, i.e., programmable attribute values.

Et cetera for monitorable and monitorable & controllable attribute values.
The type names sa1, ..., cac refer to the types denoted by the corresponding types name nsa1, ..., ncac.

```
value
38   stat_attr_vals: P → SA1×SA2×...×SAs
38   stat_attr_vals(p) ≡ let {nsa1,nsa2,...,nsas}
38   = stat_attr_typs(p) in (attr_sa1(p),attr_sa2(p),...,attr_sas(p)) end

39   ctrl_attr_vals: P → CA1×CA2×...×CAc
39   ctrl_attr_vals(p) ≡ let {nca1,nca2,...,ncac}
39   = ctrl_attr_typs(p) in (attr_ca1(p),attr_ca2(p),...,attr_cac(p)) end
```

The "ordering" of type values, (attr_sa1(p),...,attr_sas(p)), respectively (attr_ca1(p),...,attr_cac(p)), is arbitrary.

[43] By using the term *meta-linguistically* here we shall indicate that we go outside what is computable – and thus appeal to the reader's forbearance.

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 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 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 Philosophy [222, 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 ? They 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*.

We shall not develop the theme of *living species: plants and animals*, thus excluding, most notably *humans*, much further in this chapter. We claim that the present chapter, due to its foundation in Kai Sørlander's Philosophy, provides a firm foundation withing which we, or others, can further develop this theme: *analysis & description of living species.*

Intentionality: *Intentionality is a philosophical concept and is defined by the* `Stanford Encyclopedia of Philosophy`[44] *as "the power of minds to be about, to represent, or to stand for, things, properties and states of affairs."*

Definition 15 Intentional Pull: Two or more artefactual 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 24: Intentional Pull

We illustrate the concept of intentional "pull":

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

```
40  attr_Intent: A → ('transport'|...)-set
41  attr_Intent: H → ('transport'|...)-set
41  attr_Intent: L → ('transport'|...)-set
```

Manifestations of `'transport'` is reflected in *automobiles* having the automobile position attribute, APos, Item 123 Pg. 59, *hubs* having the *hub traffic* attribute, H_Traffic, Item 31 Pg. 30, and in *links* having the *link traffic* attribute, L_Traffic, Item 115 Pg. 58.

42 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;
43 seen from the point of view of a hub there is its own traffic history, H_Traffic Item 31 Pg. 30, which is a (time ordered) sequence of timed maps from automobile identities into automobile positions; and
44 seen from the point of view of a link there is its own traffic history, L_Traffic Item 115 Pg. 58, which is a (time ordered) sequence of timed maps from automobile identities into automobile positions.

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

45 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
42    A_Hi  = (𝕋 × APos)*
31     H_Trf = A_UI →m  (𝕋 × APos)*
115     L_Trf = A_UI →m  (𝕋×APos)*
45   AllATH=𝕋 →m (AUI →m APos)
45   AllHTH=𝕋 →m (AUI →m APos)
45   AllLTH =𝕋 →m (AUI →m APos)
axiom
45   let allA=mrg_AllATH({(a,attr_A_Hi(a))|a:A•a ∈ as}),
45       allH=mrg_AllHTH({attr_H_Trf(h)|h:H•h ∈ hs}),
45       allL =mrg_AllLTH({attr_L_Trf(l)|l:L•h ∈ ls}) in
45    allA = mrg_HLT(allH,allL) end
```

[44] 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 leave the definition of the four merge functions to the reader !

Discussion: 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[45]. 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 !.

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

Assignment of Attributes: So what can we deduce from the above, a little more than two pages ?

The attributes of **natural parts** and **natural materials** are generally of such concrete types – expressible as some **real** with a dimension[46] 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 artefacts humans create these with an intent.

Example 25: Intentional Pull

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.

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

Example 26: Intents

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.

Artefact 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 Some Axioms and Proof Obligations

To remind you, an **axiom** – in the *context* of domain analysis & description – means 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:* 4 on Page 24 and of *mereologies:* 5 on Page 26.

Another reminder: a **proof obligation** – in the *context* of domain analysis & description – means 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 19, about *materials sorts:* 3 on Page 23, and about *attribute types:* 6 on Page 28.

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

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

- **We use axioms** when our formula cannot otherwise express it simply, but when physical or other *properties of the domain*[47] dictates property consistency.
- **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. 1.8, 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 48.

1.5.5 Discussion of Endurants

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

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*, especially *ontology*. In this section on a sub-field of epistemology, namely that of a number of issues of *transcendental* nature, we refer to [139, Oxford Companion to Philosophy, pp 878–880] [6, The Cambridge Dictionary of Philosophy, pp 807–810] [92, The Blackwell Dictionary of Philosophy, pp 54–55 (1998)].

Definition 16 **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 17 **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** ■

[47] – 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.

Example 27: Some Transcendental Deductions

We give some intuitive examples of transcendental deductions. They are from the "domain" of programming languages. There is the syntax of a programming language, and there are the programs that supposedly adhere to this syntax. Given that, the following are now transcendental deductions. The software tool, a syntax checker, that takes a program and checks whether it satisfies the syntax, including the statically decidable context conditions, i.e., the statics semantics – that tool is one of several forms of transcendental deductions; The software tools, an automatic theorem prover[48] *and a model checker, for example* `SPIN` *[138], that takes a program and some* `theorem`*, respectively a* `Promela` *statement, and proves, respectively checks, the program correct with respect the theorem, or the statement. A compiler and an interpreter for any programming language. Yes, indeed, any abstract interpretation [106, 89] reflects a transcendental deduction: First these examples show that there are many transcendental deductions. Secondly they show that there is no single-most preferred transcendental deduction.*

A transcendental deduction, crudely speaking, is just any abstraction that can be "linked" to another, not by logical necessity, but by logical (and philosophical) possibility !

Definition 18 Transcendentality: By **transcendentality** we shall here mean the philosophical notion: the state or condition of being transcendental ■

Example 28: 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**[49] ■

The above example, 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.[50]

1.6.2 Classical Transcendental Deductions

We present a few of the transcendental deductions of [222, Kai Sørlander: *Introduction to The Philosophy*, 2016]

[50] – the attribute statement was "thrown" in "for good measure", i.e., to highlight the issue !

Space:

[222, 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"

Time:

[222, pp 159] *"Two 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"*

1.6.3 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 descriptions of perdurants, 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 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 Space and Time

This section is a necessary prelude to our treatment of perdurants.

Following Kai Sørlander's Philosophy we must accept that space and time are rationally potentially mandated in any domain description. It is, however not always necessary to model space and time. We can talk about space and time; **and** when we do, we must model them.

1.7.1 Space

General:

Mathematicians and physicists model space in, for example, the form of Hausdorf (or topological) space[51]; or a metric space which 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; or Euclidean space, due to *Euclid of Alexandria*.

[51] Armstrong, M. A. (1983) [1979]. Basic Topology. Undergraduate Texts in Mathematics. Springer. ISBN 0-387-90839-0.

Space Motivated Philosophically

Characterisation 9 **Indefinite Space:** We motivate the concept of indefinite space as follows: [222, 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*.

Characterisation 10 **Definite Space:** By a **definite space** we shall understand a space with a definite metric ■

There is but just one space. It is all around us, from the inner earth to the farthest galaxy. It is not manifest. We can not observe it as we observe a road or a human.

Space Types

The Spatial Value:

46 There is an abstract notion of (definite) $\mathbb{SPACE}$(s) of further unanalysable points; and
47 there is a notion of $\mathbb{POINT}$ in $\mathbb{SPACE}$.

```
type
46  𝕊ℙ𝔸ℂ𝔼
47  ℙ𝕆𝕀ℕ𝕋
```

Space is not an attribute of endurants. Space is just there. So we do not define an observer, `observe_space`. For us, bound to model mostly artifactual worlds on this earth there is but one space. Although $\mathbb{SPACE}$, as a type, could be thought of as defining more than one space we shall consider these isomorphic !

Spatial Observers

48 A point observer, observe_$\mathbb{POINT}$, is a function which applies to physical endurants, e, and yield a point, $\ell : \mathbb{POINT}$.

```
value
48  observe_ℙ𝕆𝕀ℕ𝕋: E → ℙ𝕆𝕀ℕ𝕋
```

1.7.2 Time

Concepts of time[52] continue to fascinate thinkers [229, 116, 166, 187, 192, 193, 194, 195, 196, 197, 205] and [120, Mandrioli et al.].

[52] **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.
[6, (i) Plato, (ii) Aristotle, (iii) Plotinus, (iv) Augustine].

Time Motivated Philosophically

Characterisation 11 **Indefinite Time:** *We motivate the abstract notion of time as follows.* [222, pp 159] *"Two 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"* ■

Characterisation 12 **Definite Time:** By a **definite time** we shall understand an abstract representation of time such as for example year, month, day, hour, minute, second, et cetera ■

Example 29: Temporal Notions of Endurants

By temporal notions of endurants we mean time properties of endurants, usually modelled as attributes. Examples are: (i) the time stamped link traffic, cf. Item 115 on Page 58 and (ii) the time stamped hub traffic, cf. Item 31 on Page 30.

Time Values

We shall not be concerned with any representation of time. That is, we leave it to the domain analyser cum describer to choose an own representation [120]. Similarly we shall not be concerned with any representation of time intervals.[53]

49 So there is an abstract type $\mathbb{T}$*ime*,

50 and an abstract type $\mathbb{TI}$: $\mathbb{T}$*ime*$\mathbb{I}$*nterval*.

51 There is no $\mathbb{T}$*ime* origin, but there is a "zero" $\mathbb{TI}$me interval.

52 One can add (subtract) a time interval to (from) a time and obtain a time.

53 One can add and subtract two time intervals and obtain a time interval – with subtraction respecting that the subtrahend is smaller than or equal to the minuend.

54 One can subtract a time from another time obtaining a time interval respecting that the subtrahend is smaller than or equal to the minuend.

55 One can multiply a time interval with a real and obtain a time interval.

56 One can compare two times and two time intervals.

type
49 $\mathbb{T}$
50 $\mathbb{TI}$
value
51 $\mathbf{0}{:}\mathbb{TI}$
52 $+,-\colon \mathbb{T} \times \mathbb{TI} \to \mathbb{T}$
53 $+,-\colon \mathbb{TI} \times \mathbb{TI} \overset{\sim}{\to} \mathbb{TI}$
54 $-\colon \mathbb{T} \times \mathbb{T} \to \mathbb{TI}$
55 $*\colon \mathbb{TI} \times$ **Real** $\to \mathbb{TI}$
56 $<,\leq,=,\neq,\geq,>\colon \mathbb{T} \times \mathbb{T} \to$ **Bool**
56 $<,\leq,=,\neq,\geq,>\colon \mathbb{TI} \times \mathbb{TI} \to$ **Bool**
axiom
52 $\forall\ t{:}\mathbb{T} \cdot t+\mathbf{0} = t$

Temporal Observers

57 We define the signature of the meta-physical time observer.

type
57 $\mathbb{T}$
value
57 **record**_$\mathbb{TIME}$(): **Unit** $\to \mathbb{T}$

[53] – but point out, that although a definite time interval may be referred to by number of years, number of days (less than 365), number of hours (less than 24), number of minutes (less than 60)number of seconds (less than 60), et cetera, this is not a time, but a time interval.

The time recorder applies to nothing and yields a time. **record**_$\mathbb{TIME}$() can only occur in action, event and behavioural descriptions.

Models of Time:

Modern models of time, by mathematicians and physicists evolve around spacetime[54] We shall not be concerned with this notion of time. Models of time related to computing differs from those of mathematicians and physicists in focusing on divergence and convergence, zero (Zenon) time and interleaving time [238] are relevant in studies of real-time, typically distributed computing systems. We shall also not be concerned with this notion of time.

Spatial and Temporal Modelling:

It is not always that we are compelled to endow our domain descriptions with those of spatial and/or temporal properties. In our experimental domain descriptions, for example, [64, 88, 68, 66, 31, 46, 60, 24], we have either found no need to model space and/or time, or we model them explicitly, using slightly different types and observers than presented above.

1.7.3 Whither Attributes ?

Are space and time attributes of endurants ? Of course not ! Space and time surround us. Every endurant is in the one-and-only space we know of. Every endurant is “somewhere” in that space. We represent that ‘somewhere’ by a point in space. Every endurant point can be recorded. And every such recording can be time-stamped.

1.7.4 Whither Entities ?

Are space and time entities ? Of course not ! They are simply abstract concepts that apply to any entity.

1.8 Perdurants

The main transcendental deduction of this chapter is that of associating with each part a behaviour. This section shows the details of this association. A main conjecture of this chapter is this:

Perdurants are understood in terms of a notion of *time* and a notion of *state*. We covered the notion of state in Sect. 1.3.7 on Page 17 and time in Sect. 1.7.2 on Page 38.

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

Example 30: Constants and States

Constants: ....

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

From that state we can calculate other states.

59 The set of all hubs, hs.
60 The set of all links, ls.

[54] The concept of **Spacetime** was first “announced” by Hermann Minkowski, 1907–08 – based on work by Henri Poincaré, 1905–06, https://en.wikisource.org/wiki/Translation:_The_Fundamental_Equations_for_Electromagnetic_Processes_in_Moving_Bodies

61 The set of all hubs and links, hls.
62 The set of all bus companies, bcs.
63 The set of all buses, bs.
64 The map from the unique bus company identifiers to the set of all the identifies bus company's buses, $bc_{ui}bs$.
65 The set of all private automobiles, as.
66 The set of all parts, ps.

value
58 rts:UoD [58]
59 hs:H-**set** ≡:H-**set** ≡ obs_sH(obs_SH(obs_RN(rts)))
60 ls:L-**set** ≡:L-**set** ≡ obs_sL(obs_SL(obs_RN(rts)))
61 hls:(H|L)-**set** ≡ hs∪ls
62 bcs:BC-**set** ≡ obs_BCs(obs_SBC(obs_FV(obs_RN(rts))))
63 bs:B-**set** ≡ ∪{obs_Bs(bc)|bc:BC•bc ∈ bcs}
64 as:A-**set** ≡ obs_BCs(obs_SBC(obs_FV(obs_RN(rts))))

Indexed States: ....

We shall

67 index bus companies,
68 index buses, and
69 index automobiles

using the unique identifiers of these parts.

type
67 BC$_{ui}$
68 B$_{ui}$
69 A$_{ui}$
value
67 $ibcs$:BC$_{ui}$-**set** ≡
67 { bc$_{ui}$ | bc:BC,bc:BC$_{ui}$:BC$_{ui}$ • bc∈bcs∧ui=uid_BC(bc) }
68 ibs:B$_{ui}$-**set** ≡
68 { b$_{ui}$ | b:B,b:B$_{ui}$:B$_{ui}$ • b∈bs∧ui=uid_B(b) }
69 ias:A$_{ui}$-**set** ≡
69 { a$_{ui}$ | a:A,a:A$_{ui}$:A$_{ui}$ • a∈as∧ui=uid_A(a) }

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 [153, Leslie Lamport] and Duration Calculus: A Formal Approach to Real-Time Systems [240, Zhou ChaoChen and Michael Reichhardt Hansen] (and [30, Chapter 15]). For a seminal book on "time in computing" we refer to the eclectic [120, Mandrioli et al., 2012]. And for seminal book on time at the epistemology level we refer to [229, J. van Benthem, 1991].

Actors

Definition 19 **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[55]. 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 20 **Discrete Action:** By a **discrete action** [234, 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 21 **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 [112, 201, 168, 109, 130, 12, 95, 185, 94].

Discrete Behaviours

Definition 22 **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* [137] For those behaviours we must therefore establish their means of *communication* via *channels*; their *signatures*; and their *definitions* – as *translated* from endurant parts.

Example 31: Behaviours

In the figure of the Channels example of Page 44 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. 1.5 on Page 20, 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 ■

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

1.8.2 Channels and Communication

We choose to exploit the CSP [137]subset of RSL since CSP is a suitable vehicle for expressing suitably abstract synchronisation and communication between behaviours.

The mereology of domain parts induces channel declarations.

CSP channels are loss-free. That is: two CSP processes, of which one offers and the other offers to accept a message do so synchronously and without forgetting that message. If you model actual, so-called "real-life" communication via queues or allowing "channels" to forget, then you must model that explicitly in CSP. We refer to [137, 206, 214].

The CSP Story:

CSP processes (models of domain behaviours), $P_i, P_j, ..., P_k$ can proceed in parallel:

```
P_i ∥ P_j ∥ ... ∥ P_k
```

Behaviours sometimes synchronise and usually communicate. Synchronisation and 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 identifier types.

The expression

```
P_i ⊓ P_j ⊓ ... ⊓ P_k
```

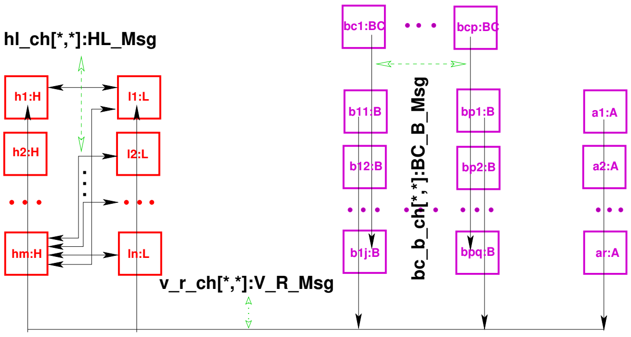
can be understood as a choice: either P_i, or P_j, or ... or P_k is *non-deterministically* **internally** chosen with no stipulation as to why !

The expression

```
P_i ▯ P_j ▯ ... ▯ P_k
```

can be understood as a choice: either P_i, or P_j, or ... or P_k is *deterministically* **externally** chosen on the basis that the one chosen offers to participate in either an input, ch ?, or an output, ch ! msg, event. If more than one P_i offers a communication then one is arbitrarily chosen. If no P_i offers a communication the behaviour halts till some P_j offers a communication.

Example 32: Channels



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 above.

Channel Message Types: ....

We ascribe types to the messages offered on channels.

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

type
71 H_L_Msg, L_H_Msg
70 HL_Msg = H_L_Msg | L_F_Msg
72 BC_B_Msg = T × BusTimTbl
73 V_R_Msg = T × (BPos|APos)

Channel Declarations: ....

74 *This justifies the channel declaration which is calculated to be:*

channel
74 { hl_ch[h_ui,l_ui]:H_L_Msg
74 | h_ui:H_UI,l_ui:L_UI•i ∈ $h_{ui}s$∧j ∈ $lh_{ui}m$(h_ui) }
74 ∪
74 { hl_ch[h_ui,l_ui]:L_H_Msg
74 | 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.

75 *This justifies the channel declaration which is calculated to be:*

channel
75 { bc_b_ch[bc_ui,b_ui] | bc_ui:BC_UI, b_ui:B_UI
75 • bc_ui ∈ $bc_{ui}s$ ∧ b_ui ∈ $b_{ui}s$ }: 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.

76 *This justifies the channel declaration which is calculated to be:*

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

The channel calculations are described on Pages 48–50 ■

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 1.7 shows two dotted rectangle box diagrams. The left fragment of the figure 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.[56] Figure 1.8 shows an arrangement similar to that of Fig. 1.7, but for an `m:n Constallation`.

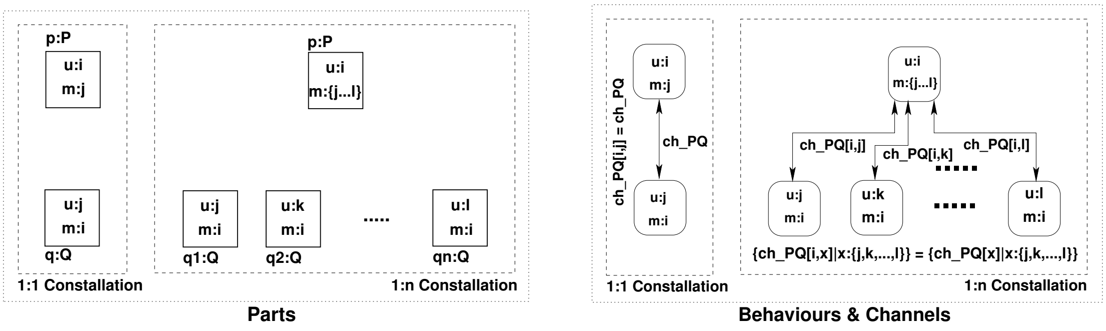


Fig. 1.7. Two Part and Channel Constallations. *u:p* unique id. *p*; *m:p* mereology *p*

The channel declarations corresponding to Figs. 1.7 and 1.8 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]:

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

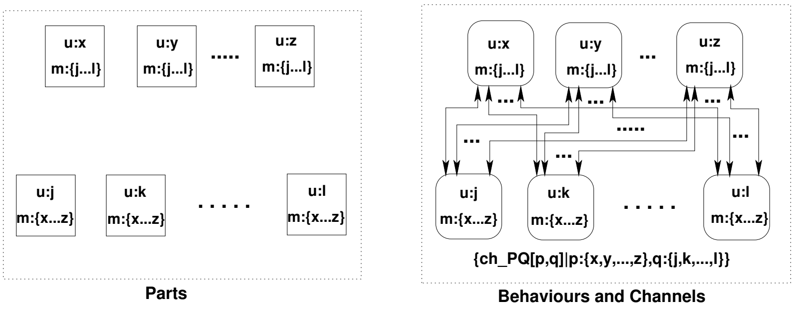


Fig. 1.8. Multiple Part and Channel Arrangements: *u:p* unique id. *p*; *m:p* mereology *p*

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

77 The following description identities holds:

```
77  { ch_PQ[x]:MPQ | x:{j,k,...,l} } ≡ ch_PQ[j],ch_PQ[k],...,ch_PQ[l],

77  { ch_PQ[p,q]:MPQ | p:{x,y,...,z}, q:{j,k,...,l} } ≡
77     ch_PQ[x,j],ch_PQ[x,k],...,ch_PQ[x,l],
77     ch_PQ[y,j],ch_PQ[y,k],...,ch_PQ[y,l],
77     ...,
77     ch_PQ[z,j],ch_PQ[z,k],...,ch_PQ[z,l]
```

We can sketch a diagram similar to Figs. 1.7 on the preceding page and 1.8 for the case of composite parts.

Continuous Behaviours:

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

1.8.3 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 23 Function Signature: By a **function signature** we shall understand a *function name* and a *function type expression* ■

Definition 24 Function Type Expression: By a **function type expression** we shall understand a pair of *type expressions*. separated by a *function type constructor* either $\to$ (for **total function**) or $\overset{\sim}{\to}$ (for **partial function**) ■

The *type expressions* 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**, $\times$, *, $\xrightarrow[m]{}$ 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:

$$\begin{array}{l} \mathsf{action}\text{: } \mathsf{VAL} \to \Sigma \overset{\sim}{\to} \Sigma \\ \mathsf{action(v)}(\sigma) \ \textbf{as} \ \sigma' \\ \qquad \textbf{pre}\text{: } \ \mathscr{P}(\mathsf{v},\sigma) \\ \qquad \textbf{post}\text{: } \mathscr{Q}(\mathsf{v},\sigma,\sigma') \end{array}$$

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 $\overset{\sim}{\to}$ 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.

Perdurant (action) analysis thus proceeds as follows: identifying relevant actions, assigning names to these, delineating the "smallest" relevant state[57], 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:

$$\begin{array}{l} \textbf{value} \\ \quad \mathsf{event}\text{: } \Sigma \times \Sigma \overset{\sim}{\to} \textbf{Bool} \\ \quad \mathsf{event}(\sigma,\sigma') \ \textbf{as} \ \mathsf{tf} \\ \qquad \textbf{pre}\text{: } \ P(\sigma) \\ \qquad \textbf{post}\text{: } \mathsf{tf} = Q(\sigma,\sigma') \end{array}$$

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 $\overset{\sim}{\to}$ 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

Signatures: We shall only cover behaviour signatures when expressed in `RSL/CSP` [123]. 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**:

$$\mathsf{behaviour}\text{: } \ldots \to \ldots \ \textbf{Unit}$$

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

[57] 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.

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.

Attribute Access, An Interpretation:

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.

- **Static attributes** designate constants. As such they can be “compiled” into behaviour definitions. We choose, instead to list them as arguments.
- **Inert attributes** designate values provided by external stimuli, that is, must be obtained by channel input: attr_Inert_A_ch ?, i.e., are considered monitorable.
- **Reactive attributes** are functions of other attribute values.
- **Autonomous attributes** must be input, like inert attributes: attr_Autonomous_A_ch ?, i.e., are considered monitorable.
- **Programmable attribute** values are calculated by their behaviours. 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 now considered programmable attributes, but when provided, in possibly tail recursive invocations of their behaviour, the calculated biddable attribute value is *modified*, usually by some *perturbation* of the calculated value – to reflect that although they *are prescribed* they *may fail to be observed as such.*

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.

78 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[58]) right pointing arrow, →, is inserted.[59]

[58] https://en.wikipedia.org/wiki/Currying

[59] We refer to the three parts of the mereology value as the input, the input/output and the output mereology (values).

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
78    calc_i_o_chn_refs: P → RSL⁺Text
78    calc_i_o_chn_refs(p) ≡ ;
78a       let ui = uid_P(p), (ics,iocs,ocs) = obs_mereo_(p), atrvs = obs_attrib_values_P(p) in
78b       if ics ∪ iocs ∪ ocs ∪ atrvs ≠ {} then ≪ → ≫ end ;
78c       if ics ∪ atrvs ≠{} then ≪in≫ calc_attr_chn_refs(ui,atrvs), calc_chn_refs(ui,ichs) end ;
78d       if iocs≠{} then ≪in,out≫  calc_chn_refs(ui,iochs) end ;
78e       if ocs≠{} then ≪out≫  calc_chn_refs(ui,ochs) end end
```

79 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.

```
79a   calc_attr_chn_refs: UI × A-set →  RSL⁺Text
79b   calc_attr_chn_refs(ui,mas) ≡ { ≪ attr_ηa_ch[ui]  ≫ | a:A•a ∈ mas }
```

80 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.

```
80a   calc_chn_refs: P_UI × Q_UI-set →  RSL⁺Text
80b   calc_chn_refs(pui,quis) ≡ { ≪ η(pui,qui)_ch[pui,qui]  ≫ | qui:Q_UI•qui ∈ quis }
```

81 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.

```
81a   calc_all_chn_dcls: P_UI × Q_UI-set → RSL⁺Text
81a   calc_all_chn_dcls(pui,quis) ≡ { ≪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.

82 The overloaded η operator[60] is here applied to a pair of unique identifiers.

```
82    η: (UI → RSL⁺Text)|((X_UI×Y_UI) → RSL⁺Text)
82    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:

[60] The η operator applies to a type and yields the ηame of the type.

$$\{\mathsf{PQ\_ch}[\mathsf{p1}_{ui},\mathsf{qui}]|\mathsf{p1}_{ui}:\mathsf{P\_UI},\mathsf{qui}:\mathsf{Q\_UI}\bullet\mathsf{qui} \in \mathsf{quis}\}$$
$$\{\mathsf{PQ\_ch}[\mathsf{p2}_{ui},\mathsf{qui}]|\mathsf{p2}_{ui}:\mathsf{P\_UI},\mathsf{qui}:\mathsf{Q\_UI}\bullet\mathsf{qui} \in \mathsf{quis}\}$$
$$\ldots$$
$$\{\mathsf{PQ\_ch}[\mathsf{p}n_{ui},\mathsf{qui}]|\mathsf{p}n_{ui}:\mathsf{P\_UI},\mathsf{qui}:\mathsf{Q\_UI}\bullet\mathsf{qui} \in \mathsf{quis}\}$$

These distinct single channel references can be assembled into one:

$$\{\ \mathsf{PQ\_ch}[\mathsf{pui},\mathsf{qui}]\ |\ \mathsf{pui}:\mathsf{P\_UI},\mathsf{qui}:\mathsf{Q\_UI}\ :\ -\mathsf{pui} \in \mathsf{puis},\mathsf{qui} \in \mathsf{quis}\ \}$$
$$\textbf{where}\ \mathsf{puis} = \{\ \mathsf{p1}_{ui},\mathsf{p2}_{ui},\ldots,\mathsf{p}n_{ui}\}$$

As an example we have already calculated the array channels for Fig. 1.8 Pg. 46– cf. the left, the **Parts**, of that figure – cf. Items [1–3] Pages 45–46.The identities Item 77 Pg. 46 apply.

1.8.4 Discrete Behaviour Definitions

We associate with each part, *p*:*P*, a behaviour name $\mathscr{M}_P$. 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.[61] A behaviour signature is therefore:

$$\mathscr{M}_P\text{: ui:UI}\times\text{me:MT}\times\text{stat\_attr\_typs(p)} \rightarrow \text{ctrl\_attr\_typs(p)} \rightarrow \text{calc\_i\_o\_chn\_refs(p)}\ \textbf{Unit}$$

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[62] sub-sorts E_1, E_2, ..., E_n. The behaviour description *translated* from p:P, is composed from a behaviour description, $\mathscr{M}_P$, 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.

Abstract is_composite(p) Behaviour schema

```
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
```

[61] We leave out consideration of possible components and materials of the part.

[62] – structures or composite

Expression $\mathscr{B}_P$(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 19. We informally explain the **Translate**$_{\mathsf{P}_i}$ function. It takes endurants and produces RSL$^+$Text. Resulting texts are bracketed: ≪rsl_text≫

Example 33: Signatures

We first decide on names of behaviours. In Sect. 1.8.4, Pages 50–53, 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.

83 *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
83    hub$_{h_{ui}}$:
83a     h_ui:H_UI×(vuis,luis,__):H_Mer×HΩ
83b       → (HΣ×H_Traffic)
83c       → in,out { h_l_ch[h_ui,l_ui] | l_ui:L_UI•l_ui ∈ luis }
83d            { ba_r_ch[h_ui,v_ui] | v_ui:V_UI•v_ui∈vuis } Unit
83a        pre: vuis = $v_{ui}s$ ∧ luis = $l_{ui}s$
```

84 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
84    link$_{l_{ui}}$:
84a     l_ui:L_UI×(vuis,huis,__):L_Mer×LΩ
84b       → (LΣ×L_Traffic)
84c       → in,out { h_l_ch[h_ui,l_ui] | h_ui:H_UI:h_ui ∈ huis }
84d            { ba_r_ch[l_ui,v_ui] | v_ui:(B_UI|A_UI)•v_ui∈vuis } Unit
84a       pre: vuis = $v_{ui}s$ ∧ huis = $h_{ui}s$
```

85 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 allowing communication between the bus company and buses.

```
value
85    bus_company$_{bc_{ui}}$:
85a     bc_ui:BC_UI×(__,__,buis):BC_Mer
85b       → BusTimTbl
85c          in,out {bc_b_ch[bc_ui,b_ui]|b_ui:B_UI•b_ui∈buis} Unit
```

```
85a     pre: buis = $b_{ui}s$ ∧ huis = $h_{ui}s$
```

....

86 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/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
86    bus$_{b_{ui}}$:
86a   b_ui:B_UI×(bc_ui,_,ruis):B_Mer
86b     → (LN × BTT × BPOS)
86c     → out bc_b_ch[bc_ui,b_ui],
86d           {ba_r_ch[r_ui,b_ui]|r_ui:(H_UI|L_UI)•ui∈$v_{ui}s$} Unit
86a     pre: ruis = $r_{ui}s$ ∧ bc_ui ∈ $bc_{ui}s$
```

....

87 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 allowing communication between the automobile and the hub and link behaviours.

```
value
87    automobile$_{a_{ui}}$:
87a   a_ui:A_UI×(_,_,ruis):A_Mer×rn:RegNo
87b     → apos:APos
87c       in,out {ba_r_ch[a_ui,r_ui]|r_ui:(H_UI|L_UI)•r_ui∈ruis} Unit
87a     pre: ruis = $r_{ui}s$ ∧ a_ui ∈ $a_{ui}s$ ■
```

For the case that an endurant is a structure there is only its elements to compile; otherwise Schema 2 is as Schema 1.

Abstract is_structure(e) **Behaviour** schema

```
value
  Translate$_E$(e) ≡
        Translate$_{E_1}$(obs_endurant_sorts_E$_1$(e))
    ⟪⟫ Translate$_{E_2}$(obs_endurant_sorts_E$_2$(e))
    ⟪⟫ ...
    ⟪⟫ Translate$_{E_n}$(obs_endurant_sorts_E$_n$(e))
```

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.

Concrete is_composite(p) **Behaviour** schema

```
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
            $\mathscr{M}_P$: P_UI×MT×ST  CT  IOR  Unit
            $\mathscr{M}_P$(ui,me,sa)ca ≡ $\mathscr{B}_P$(ui,me,sa)ca  ≫
            { ≪,≫ Translate$_Q$(q)|q:Q•q ∈ qs }
      end
```

Atomic `is_atomic(p)` **Behaviour** schema

```
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
            $\mathscr{M}_P$: P_UI×MT×ST  PT  IOR  Unit
            $\mathscr{M}_P$(ui,me,sa)ca ≡ $\mathscr{B}_P$(ui,me,sa)ca  ≫
      end
```

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

Core Behaviour schema

```
$\mathscr{B}_P$: 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 [67, Process Schema V: Core Process (II), Page 40] for possible forms of $\mathscr{F}_P$.

Example 34: Automobile Behaviour (at a hub)

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.*

88 We abstract automobile behaviour at a Hub *(hui).*
89 The vehicle remains at that hub, "idling",
90 informing the hub behaviour,
91 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,*

92 *or, again internally non-deterministically,*
93 *the vehicle “disappears — off the radar” !*

```
88  automobile_{a_ui}(a_ui,({},(ruis,vuis),{}),rn)
88          (apos:atH(fl_ui,h_ui,tl_ui)) ≡
89     (ba_r_ch[a_ui,h_ui] ! (record_TIME(),atH(fl_ui,h_ui,tl_ui));
90      automobile_{a_ui}(a_ui,({},(ruis,vuis),{}),rn)(apos))
91     ⌈⌉
91a    (let ({fh_ui,th_ui},ruis′)=mereo_L(℘(tl_ui)) in
91a           assert: fh_ui=h_ui ∧ ruis=ruis′
88      let onl = (tl_ui,h_ui,0,th_ui) in
91b      (ba_r_ch[a_ui,h_ui] ! (record_TIME(),onL(onl)) ||
91b       ba_r_ch[a_ui,tl_ui] ! (record_TIME(),onL(onl))) ;
91c       automobile_{a_ui}(a_ui,({},(ruis,vuis),{}),rn)
91c              (onL(onl)) end end)
92     ⌈⌉
93       stop
```

Section 1.9.3 presents the definition of the remaining automobile, hub, link, bus company and bus behaviours.

1.8.5 Running Systems

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.

Example 34: Initial System

Initial States: We recall the *hub, link, bus company, bus* and the *automobile states* first mentioned in Sect. 1.3.7 Page 17.

```
value
59  hs:H-set ≡ ≡ obs_sH(obs_SH(obs_RN(rts)))
60  ls:L-set ≡ ≡ obs_sL(obs_SL(obs_RN(rts)))
62  bcs:BC-set ≡ obs_BCs(obs_SBC(obs_FV(obs_RN(rts))))
63  bs:B-set ≡ ∪{obs_Bs(bc)|bc:BC•bc ∈ bcs}
65  as:A-set ≡ obs_BCs(obs_SBC(obs_FV(obs_RN(rts))))
```

Starting Initial Behaviours: We are reaching the end of this domain modelling example. Behind us there are narratives and formalisations1 Pg. 19 – 137 Pg. 62.Based on these we now express the signature and the body of the definition of a *“system build and execute”* function.

94 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
94   initial_system: Unit → Unit
94   initial_system() ≡
94c      ‖ { hub$_{h_{ui}}$(h_ui,me,hω)(htrf,hσ)
94c         | h:H•h ∈ hs,
94c           h_ui:H_UI•h_ui=uid_H(h),
94c           me:HMetL•me=mereo_H(h),
94c           hω:HΩ•hω=attr_HΩ(h),
94c           htrf:H_Traffic•htrf=attr_H_Traffic_H(h),
94c           hσ:HΣ•hσ=attr_HΣ(h)∧hσ ∈ hω
94c         }

94a      ‖
94d      ‖ { link$_{l_{ui}}$(l_ui,me,lω)(ltrf,lσ)
94d           l:L•l ∈ ls,
94d           l_ui:L_UI•l_ui=uid_L(l),
94d           me:LMet•me=mereo_L(l),
94d           lω:LΩ•lω=attr_LΩ(l),
94d           ltrf:L_Traffic•ltrf=attr_L_Traffic_H(l),
94d           lσ:LΣ•lσ=attr_LΣ(l)∧lσ ∈ lω
94d         }

94a      ‖
94e      ‖ { bus_company$_{bc_{ui}}$(bcui,me)(btt)
94e           bc:BC•bc ∈ bcs,
94e           bc_ui:BC_UI•bc_ui=uid_BC(bc),
94e           me:BCMet•me=mereo_BC(bc),
94e           btt:BusTimTbl•btt=attr_BusTimTbl(bc)
94e         }

94a      ‖
94f      ‖ { bus$_{b_{ui}}$(b_ui,me)(ln,btt,bpos)
94f          b:B•b ∈ bs,
94f          b_ui:B_UI•b_ui=uid_B(b),
94f          me:BMet•me=mereo_B(b),
94f          ln:LN:pln=attr_LN(b),
94f          btt:BusTimTbl•btt=attr_BusTimTbl(b),
94f          bpos:BPos•bpos=attr_BPos(b)
94f        }

94a      ‖
94g       ‖ { automobile$_{a_{ui}}$(a_ui,me,rn)(apos)
94g            a:A•a ∈ as,
94g            a_ui:A_UI•a_ui=uid_A(a),
94g            me:AMet•me=mereo_A(a),
94g            rn:RegNo•rno=attr_RegNo(a),
94g            apos:APos•apos=attr_APos(a)
94g          }  ■
```

1.8.6 Concurrency: Communication and Synchronisation

Process Schemas I, II, III and V (Pages 50, 52, 52 and 53), reveal that two or more parts, which temporally coexist (i.e., at the same time), imply a notion of *concurrency*. Process Schema IV, Page 53, 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*.

1.8.7 Summary and Discussion of Perdurants

The most significant contribution of Sect. 1.8 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 modelling 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 modelling 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.

1.9 The Example Concluded

Example 34:

1.9.1 Unique Identifier Concepts

We define a few concepts related to unique identification.

Extract Parts from Their Unique Identifiers: . . . .

95 From the unique identifier of a part we can retrieve, $\wp$, the part having that identifier.

type
95 P = H | L | BC | B | A
value
95 $\wp$: H_UI→H | L_UI→L | BC_UI→BC | B_UI→B | A_UI→A
95 $\wp$(ui) ≡ **let** p:(H|L|BC|B|A)•p∈ps∧uid_P(p)=ui **in** p **end**

Unique Identifier Constants . . . .
We can calculate:

96 the set, $h_{ui}s$, of unique hub identifiers;
97 the set, $l_{ui}s$, of unique link identifiers;
98 the map, $hl_{ui}m$, from unique hub identifiers to the set of unique link iidentifiers of the links connected to the zero, one or more identified hubs,
99 the map, $lh_{ui}m$, from unique link identifiers to the set of unique hub iidentifiers of the two hubs connected to the identified link;
100 the set, $r_{ui}s$, of all unique hub and link, i.e., road identifiers;

101 *the set, $bc_{ui}s$, of unique bus company identifiers;*
102 *the set, $b_{ui}s$, of unique bus identifiers;*
103 *the set, $a_{ui}s$, of unique private automobile identifiers;*
104 *the set, $v_{ui}s$, of unique bus and automobile, i.e., vehicle identifiers;*
105 *the map, $bcb_{ui}m$, from unique bus company identifiers to the set of its unique bus identifiers; and*
106 *the (bijective) map, $bbc_{ui}bm$, from unique bus identifiers to their unique bus company identifiers.*

```
96   $h_{ui}s$:H_UI-set ≡ {uid_H(h)|h:H•h ∈ $hs$}
97   $l_{ui}s$:L_UI-set ≡ {uid_L(l)|l:L•l ∈ $ls$}
100  $r_{ui}s$:R_UI-set ≡ $h_{ui}s$∪$l_{ui}s$
98   $hl_{ui}m$:(H_UI $\underset{m}{\rightarrow}$ L_UI-set) ≡
98      [h_ui↦luis|h_ui:H_UI,luis:L_UI-set•h_ui∈$h_{ui}s$
98                 ∧(_,luis,_)=mereo_H($\eta$(h_ui))]  [cf. Item 22]
99   $lh_{ui}m$:(L+UI $\underset{m}{\rightarrow}$ H_UI-set) ≡
99      [l_ui↦huis                         [cf. Item 23]
99      | h_ui:L_UI,huis:H_UI-set • l_ui∈$l_{ui}s$
99       ∧ (_,huis,_)=mereo_L($\eta$(l_ui))]
101  $bc_{ui}s$:BC_UI-set ≡ {uid_BC(bc)|bc:BC•bc ∈ $bcs$}
102  $b_{ui}s$:B_UI-set ≡ ∪{uid_B(b)|b:B•b ∈ $bs$}
103  $a_{ui}s$:A_UI-set ≡ {uid_A(a)|a:A•a ∈ $as$}
104  $v_{ui}s$:V_UI-set ≡ $b_{ui}s$ ∪ $a_{ui}s$
105  $bcb_{ui}m$:(BC_UI $\underset{m}{\rightarrow}$ B_UI-set) ≡
105     [ bc_ui ↦ buis
105     | bc_ui:BC_UI, bc:BC •
105       bc∈$bcs$ ∧ bc_ui=uid_BC(bc)
105      ∧ (_,_,buis)=mereo_BC(bc) ]
106  $bbc_{ui}bm$:(B_UI $\underset{m}{\rightarrow}$ BC_UI) ≡
106     [ b_ui ↦ bc_ui
106     | b_ui:B_UI,bc_ui:BC_ui •
106       bc_ui=dom$bcb_{ui}m$∧b_ui∈$bcb_{ui}m$(bc_ui) ]
```

Uniqueness of Part Identifiers: . . . .
We refer to Sect. 1.5.4 Pg. 34. We must express the following axioms:

107 All hub identifiers are distinct.
108 All link identifiers are distinct.
109 All bus company identifiers are distinct.
110 All bus identifiers are distinct.
111 All private automobile identifiers are distinct.
112 All part identifiers are distinct.

```
107  card $hs$ = card $h_{ui}s$
108  card $ls$ = card $l_{ui}s$
109  card $bcs$ = card $bc_{ui}s$
110  card $bs$ = card $b_{ui}s$
111  card $as$ = card $a_{ui}s$
112  card {$h_{ui}s$∪$l_{ui}s$∪$bc_{ui}s$∪$b_{ui}s$∪$a_{ui}s$}
112     = card $h_{ui}s$+card $l_{ui}s$+card $bc_{ui}s$+card $b_{ui}s$+card $a_{ui}s$
```

1.9.2 Further Transport System Attributes

Links: *We show just a few attributes.* . . . .

113 *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 $from$ hub h_f to hub h_t. Link states can have either 0, 1 or 2 elements.*

114 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".

115 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.

116 The hub identifiers of link states must be in the set, $h_{ui}s$, of the road net's hub identifiers.

```
type
113  LΣ = H_UI-set                     [programmable, Df.8 Pg.29]
axiom
113  ∀ lσ:LΣ•card lσ=2
113  ∀ l:L • obs_LΣ(l) ∈ obs_LΩ(l)
type
114  LΩ = LΣ-set                               [static, Df.1 Pg.29]
115  L_Traffic                         [programmable, Df.8 Pg.29]
115  L_Traffic = (A_UI|B_UI) →m (𝕋×(H_UI×Frac×H_UI))*
115  Frac = Real, axiom frac:Fract • 0<frac<1
value
113  attr_LΣ: L → LΣ
114  attr_LΩ: L → LΩ
115  attr_L_Traffic: : → L_Traffic
axiom
115  ∀ lt:L_Traffic,ui:(A_UI|B_UI)•ui ∈ dom ht
115        ⇒ time_ordered(ht(ui))

116  ∀ l:L • l ∈ ls ⇒
116      let lσ = attr_LΣ(l) in
116      ∀ (h_ui i,h_ui i′):(H_UI×K_UI) •
116          (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 buses.

117 Bus companies create, maintain, revise and distribute [to the public (not modeled here), and to buses] bus time tables, not further defined.

```
type
117   BusTimTbl                        [programmable, Df.8 Pg.29]
value
117   attr_BusTimTbl: BC → BusTimTbl
```

There are two notions of time at play here: the indefinite "real" or "actual" time; and the definite calendar, hour, minute and second time designation occurring in some textual form in, e.g., time tables.

Buses: *We show just a few attributes:* . . . .

118 Buses run routes, according to their line number, ln:LN, in the

119 bus time table, btt:BusTimTbl obtained from their bus company, and and keep, as inert attributes, their segment of that time table.

120 Buses 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 hubs, fh_ui, in the direction of the other identified hub, th_ui.

121 Et cetera.

```
type
118   LN                            [programmable, Df.8 Pg.29]
119   BusTimTbl                     [inert, Df.3 Pg.29]
120   BPos   == atHub | onLink      [programmable, Df.8 Pg.29]
120a  atHub    :: h_ui:H_UI
120b  onLink   :: fh_ui:H_UI×l_ui:L_UI×frac:Fract×th_ui:H_UI
120b  Fract    = Real, axiom frac:Fract • 0<frac<1
121   ...
value
119   attr_BusTimTbl: B → BusTimTbl
120   attr_BPos: B → BPos
```

Private Automobiles: *We show just a few attributes:* . . . .

We illustrate but a few attributes:

122 Automobiles have static number plate registration numbers.

123 Automobiles have dynamic positions on the road net:

[120a] either at a hub identified by some h_ui,

[120b] or on a link, some fraction, frac:Fract down an identified link, l_ui, from one of its identified connecting hubs, fh_ui, in the direction of the other identified hub, th_ui.

```
type
122   RegNo                          [static, Df.1 Pg.29]
123   APos   == atHub | onLink       [programmable, Df.8 Pg.29]
120a  atHub    :: h_ui:H_UI
120b  onLink   :: fh_ui:H_UI × l_ui:L_UI × frac:Fract × th_ui:H_UI
120b  Fract    = Real, axiom frac:Fract • 0<frac<1
value
122   attr_RegNo: A → RegNo
123   attr_APos: A → 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 117 on the facing page, Page 58. Observe then that buses each have their own distinct *bus time table*, and that these are model-led as *inert*, Item 119 on the preceding page, Page 58. In Items 133–134b Pg. 61 we shall see how the buses communicate with their respective bus companies in order for the buses to obtain the *programmed* bus time tables "in lieu" of their *inert* one! In Items 31 Pg. 30 and 115 Pg. 58, 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.[63]

1.9.3 Behaviours

Automobile Behaviour (on a link) . . . .

124 We abstract automobile behaviour on a Link.

[63] 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.

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” !

```
124        automobile$_{a_{ui}}$(a_ui,({},ruis,{}),rno)
124                            (vp:onL(fh_ui,l_ui,f,th_ui)) ≡
124(a)ii    (ba_r_ch[thui,aui]!atH(lui,thui,nxt_lui) ;
124(a)i      automobile$_{a_{ui}}$(a_ui,({},ruis,{}),rno)(vp))
124b         ⊓
124(b)i     (if not_yet_at_hub(f)
124(b)i        then
124(b)i1         (let incr = increment(f) in
88                let onl = (tl_ui,h_ui,incr,th_ui) in
124(b)i2           ba−r_ch[l_ui,a_ui] ! onL(onl) ;
124(b)i3           automobile$_{a_{ui}}$(a_ui,({},ruis,{}),rno)
124(b)i3                                  (onL(onl))
124(b)i            end end)
124(b)ii        else
124(b)ii1         (let nxt_lui:L_UI•nxt_lui ∈ mereo_H(℘(th_ui)) in
124(b)ii2          ba_r_ch[thui,aui]!atH(l_ui,th_ui,nxt_lui) ;
124(b)ii3          automobile$_{a_{ui}}$(a_ui,({},ruis,{}),rno)
124(b)ii3                                 (atH(l_ui,th_ui,nxt_lui)) end)
124(b)i      end)
124c         ⊓
124d         stop
124(b)i1    increment: Fract → Fract
```

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

125 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 31 Pg. 30.

```
value
125   hub$_{h_{ui}}$(h_ui,(,(luis,vuis)),h$\omega$)(h$\sigma$,ht) ≡
125a       ▯
125b          { let m = ba_r_ch[h_ui,v_ui] ? in
125c                assert: m=(__,atHub(__,h_ui,__))
125d             let ht′ = ht † [h_ui ↦ ⟨m⟩⌢ht(h_ui)] in
125e            hub$_{h_{ui}}$(h_ui,(,(luis,vuis)),(hω))(hσ,ht′)
125f         | v_ui:V_UI•v_ui∈vuis end end }
125g    post: ∀ v_ui:V_UI•v_ui ∈ dom ht′⇒time_ordered(ht′(v_ui))
```

Link Behaviour

126 The link behaviour non-deterministically, externally offers
127 to accept timed vehicle positions —
128 which will be on the link, from some vehicle, v_ui.
129 The timed vehicle link position is appended to the front of that vehicle's entry in the link's traffic table;
130 whereupon the link proceeds as a link behaviour with the updated link traffic table.
131 The link behaviour offers to accept from any vehicle.
132 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 115 Pg. 58.

```
126   link$_{l_{ui}}$(l_ui,(_,(huis,vuis),_),lω)(lσ,lt) ≡
126      □
127         { let m = ba_r_ch[l_ui,v_ui] ? in
128               assert: m=(_,onLink(_,l_ui,_,_))
129           let lt′ = lt † [l_ui ↦ ⟨m⟩⌢lt(l_ui)] in
130           link$_{l_{ui}}$(l_ui,(huis,vuis),hω)(hσ,lt′)
131         | v_ui:V_UI•v_ui∈vuis end end }
132      post: ∀ v_ui:V_UI•v_ui ∈ dom lt′⇒time_ordered(lt′(v_ui))
```

Bus Company Behaviour

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.

133 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;
134 "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.

```
85    bus_company$_{bc_{ui}}$(bcui,(_,buis,_))(btt) ≡
133a   (let btt′ = update(btt,...) in
133b    bus_company$_{bc_{ui}}$(bcui,(_,buis,_))(btt′) end )
134    ⌷
134a   ( □ {bc_b_ch[bc_ui,b_ui] ! btt | b_ui:B_UI•b_ui∈buis
134b        bus_company$_{bc_{ui}}$(bcui,(_,buis,_))(record_TIME(),btt) } )
```

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).

135 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.
136 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.

```
135     bus$_{b_{ui}}$(b_ui,(_,(bc_ui,ruis),_))(ln,btt,bpos) ≡
135a        (let btt′ = b_bc_ch[b_ui,bc_ui] ? in
135b         bus$_{b_{ui}}$(b_ui,({},(bc_ui,ruis),{}))(ln,btt′,bpos) end)
136       ⊓
136a        (case bpos of
136(a)i         atH(fl_ui,h_ui,tl_ui) →
136(a)i              atH_bus$_{b_{ui}}$(b_ui,(_,(bc_ui,ruis),_))(ln,btt,bpos),
136(a)ii        aonL(fh_ui,l_ui,f,th_ui) →
136(a)ii             onL_bus$_{b_{ui}}$(b_ui,(_,(bc_ui,ruis),_))(ln,btt,bpos)
136a         end)
```

Bus Behaviour at a Hub 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 89–124d below presents "nothing new" !

```
136(a)i    atH_bus$_{b_{ui}}$(b_ui,(_,(bc_ui,ruis),_))
136(a)i                   (ln,btt,atH(fl_ui,h_ui,tl_ui)) ≡
89           (ba_r_ch[b_ui,h_ui] ! (record_TIME(),atH(fl_ui,h_ui,tl_ui));
90            bus$_{b_{ui}}$(b_ui,({},(bc_ui,ruis),{}))(ln,btt,bpos))
135a          ⊓
91a          (let ({fh_ui,th_ui},ruis′)=mereo_L(℘(tl_ui)) in
91a                  assert: fh_ui=h_ui ∧ ruis=ruis′
88            let onl = (tl_ui,h_ui,0,th_ui) in
91b            (ba_r_ch[b_ui,h_ui] ! (record_TIME(),onL(onl)) ∥
91b             ba_r_ch[b_ui,tl_ui] ! (record_TIME(),onL(onl))) ;
91c            bus$_{b_{ui}}$(b_ui,({},(bc_ui,ruis),{}))
91c                       (ln,btt,onL(onl)) end end )
124c          ⊓
124d           stop
```

Bus Behaviour on a Link . . . .

The onL_bus$_{b_{ui}}$ behaviour definition is a similar simple transcription of the automobile$_{a_{ui}}$ (onL) behaviour definition. So formula lines 89–124d below presents "nothing new" !

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

```
136(a)ii   onL_bus$_{b_{ui}}$(b_ui,(_,(bc_ui,ruis),_))
136(a)ii                  (ln,btt,bpos:onL(fh_ui,l_ui,f,th_ui)) ≡
89           (ba_r_ch[b_ui,h_ui] ! (record_TIME(),bpos);
90            bus$_{b_{ui}}$(b_ui,({},(bc_ui,ruis),{}))(ln,btt,bpos))
135a            ⊓
124(b)i          (if not_yet_at_hub(f)
124(b)i              then
124(b)i1                 (let incr = increment(f) in
88                        let onl = (tl_ui,h_ui,incr,th_ui) in
124(b)i2                  ba−r_ch[l_ui,b_ui] ! onL(onl) ;
124(b)i3                  bus$_{b_{ui}}$(b_ui,({},(bc_ui,ruis),{}))
124(b)i3                          (ln,btt,onL(onl))
124(b)i                   end end)
124(b)ii             else
124(b)ii1                 (let nl_ui:L_UI•nxt_lui∈mereo_H(℘(th_ui)) in
124(b)ii2                  ba_r_ch[thui,b_ui]!atH(l_ui,th_ui,nxt_lui) ;
124(b)ii3                  bus$_{b_{ui}}$(b_ui,({},(bc_ui,ruis),{}))
124(b)ii3                          (ln,btt,atH(l_ui,h_ui,nxt_lui))
124(b)ii1                  end)end)
```

```
124c       ⌈⌉
124d         stop
```

1.10 Closing

Domain models abstract some reality. They do not pretend to capture all of it.

1.10.1 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 *artefacts*, *unique identifiers*, *mereology* and *attributes*.

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.

The Transcendental Deduction: A concept of *transcendental deduction* has been introduced. It is used to justify the interpretation of *endurant parts* as *perdurant behaviours* – à la CSP. 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*. We claim that when, as an example, programmers, in thinking about or in explaining their code, anthropomorphically[64], say that *"the program does so and so"* they 'perform' and transcendental deduction. We refer to the forthcoming [69, Philosophical Issues in Domain Modeling].

- This concept should be studied further: *Transcendental Deduction in Computing Science*.

Living Species: The concept of *living species* has been introduced, but it has not been "sufficiently" studied, that is, we have, in Sect. 1.5.3 on Page 32, hinted at a number of 'living species' notions: *causality of purpose* et cetera, but no hints has been given as to the kind of attributes that *living species*, especially *humans* give rise to.

- This concept should be studied further: *Attributes of Living Species in Computing Science*.

Intentional "Pull": 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.

- This concept should be studied further: *Intentional "Pull" in Computing Science*.

What Can Be Described ? 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 [156, The Oxford Shorter English Dictionary] and from the Internet based [239, 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.

[64] Anthropomorphism is the attribution of human traits, emotions, or intentions to non-human entities.

Some other issues are to be further studied. (i) When to use *physical mereologies* and when to apply *conceptual mereologies*, cf. final paragraph of Sect. 1.5.2 on Page 27. (ii) How do we know that the categorisation into unique identification, mereology and attributes embodies all internal qualities; could there be a fourth, etc. ? (iii) Is *intent* an attribute, or does it "belong" to a fourth internal quality category, or a fifth ? (iv) It seems that most of what we first thought off as natural parts really are materials: geographic land masses, etc. – subject, still, to the laws of physics: geo-physics.

• We refer to the forthcoming study [69, Philosophical Issues in Domain Modeling] based on [219, 220, 221, 222].

A Conjecture: It could be interesting to study *under what circumstances, including for which kind of behaviours, we can postulate the following:*

Conjecture: Parts ≅ Behaviours

To every part there is a behaviour, and to every suitably expressed behaviour there is a part.

We shall leave this study to the reader !

The Contribution: In summary we have shown that the domain analysis & description calculi form a sound, consistent and complete approach to domain modelling, and that this approach takes its "resting point" in Kai Sørlander's Philosophy.

1.10.2 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* 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 [123, 27]. We shall refer to this language as $\mathtt{RSL}^{+}$. Main features of $\mathtt{RSL}^{+}$ evolves in this chapter, mainly in Sect. 1.8.3.

Here we shall make it clear that we need three languages: (i) an **analysis language**, (ii) a **description language**, i.e., $\mathtt{RSL}^{+}$, and (iii) the language of explaining domain analysis & description, (iv) in modelling "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:

The Analysis Prompts

a. is_ entity, 10
b. is_ endurant, 10
c. is_ perdurant, 11
d. is_ discrete, 11
e. is_ continuous, 11
f. is_ physical_ part, 12
g. is_ living_ species, 12
h. is_ structure, 13
i. is_ part, 14
j. is_ atomic, 14
k. is_ composite, 15
l. is_ living_ species, 15
m. is_ plant, 15
n. is_ animal, 16
o. is_ human, 16
p. has_ materials, 17
q. is_ artefact, 17
r. observe_ endurant_ sorts, 18
s. has_ concrete_ type, 20
t. has_ mereology, 25
u. attribute_ types, 28

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 prompts:**

The Description Prompts

[1] `observe_ endurant_ sorts`, 18	[4] `observe_ unique_ identifier`, 24
[2] `observe_ part_ type`, 20	[5] `observe_ mereology`, 25
[3] `observe_ material_ sorts`, 22	[6] `observe_ attributes`, 28

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 [123, 27], that is: no assignable variables. Also we omit RSL's elaborate *scheme, class, object* notions.

The Description Language Primitives

- *Structures, Parts, Components and Materials:*
  - **obs_**E, — dfn. 1, [o] pg. 19
  - **obs_**T: P, — dfn. 2, [t$_2$] pg. 21
- *Part and Component Unique Identifiers:*
  - **uid_**P, — dfn. 4, [u] pg. 24
- *Part Mereologies:*
  - **obs_mereo_**P, — dfn. 5, [m] pg. 26
- *Part and Material Attributes:*
  - **attr_**A$_i$, — dfn. 6, [a] pg. 28

We refer, generally, to all these functions as observer functions. They are defined by the analyser cum describer when "applying" description prompts. That is, they should be considered user-defined. In our examples we use the non-bold-faced observer function names.

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*.

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 *endurants* 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.

An Analysis & Description Process:

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

Program Schema: A Domain Analysis & Description Process

```
type
    V = Part_VAL | Komp_VAL | Mat_VAL
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_part(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 ) ,
             has_components(v) → let {k1:K1,k2:K2,...,kn:Kn} = observe_components(v) in
                        new := new ∪ {k1,k2,...,kn} ; txt := txt ∪ observe_component_sorts(v) end ,
             has_materials(v) → txt := txt ∪ observe_material_sorts(v) ,
             is_structure(v) → ... [EXERCISE FOR THE READER !]
             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 txt – the output text.

1.10.3 Relation to Other Formal Specification Languages

In this contribution we have based the analysis and description calculi and the specification texts emanating as domain descriptions on RSL [123]. There are other formal specification languages:

- **Alloy** [143],
- **B** (etc.) [1],
- **CafeObj** [121],
- **CASL** [105],
- **VDM** [83, 84, 118],
- **Z** [236],

to mention a few. Two conditions appear to apply for any of these other formal specification languages to become a basis for analysis and description calculi similar to the ones put forward in the current chapter: (i) it must be possible, as in RSL, to define and express sorts, i.e., *further undefined types*, and (ii) it must be possible, as with RSL’s “built-in” **CSP** [137] in some form or another, to define and express concurrency. Insofar as these and other formal languages can satisfy these two conditions, they can certainly also be the basis for domain analysis & description.

We do not consider **Coq** [110, 140, 182][65], **CSP** [137] **The Duration Calculus** [240] nor **TLA+** [153] as candidates for expressing full-fledged domain descriptions. Some of these formal specification languages, like **Coq**, are very specifically oriented towards proofs (of properties of specifications). Some, like **The Duration Calculus** and **CSP**, go very well in hand with other formal specification languages like **VDM, RAISE**[66] and **Z**. It seems, common to these languages, that, taken in isolation, they can be successfully used for the development and proofs of properties of algorithms and code for, for example safety-critical and embedded systems. But our choice (of not considering) is not a “hard nailed” one ! Also less formal, usually computable, languages, like **Scala** `[https://www.scala-lang.org/]` or **Python** `[https:/www.python.org/]`, can, if they satisfy criteria (i-ii), serve similarly. We refer, for a more general discussion – of issues related to the choice of other formal language being the basis for domain analysis & description – to [82, 40 Years of Formal Methods — 10 Obstacles and 3 Possibilities] for a general discussion that touches upon the issue of formal, or near-formal, specification languages.

1.10.4 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 MODELLING *?* My answer assumes a “state-of-affairs” of domain science & engineering in which domain modelling is an established subject, i.e., where the domain analysis & description topic, i.e., its methodology, is taught, where there are “text-book”

[65] http://doi.org/10.5281/zenodo.1028037

[66] A variant of **CSP** is thus “embedded” in **RSL**

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**.

1.10.5 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 chapter. Another thing is its practice. I find myself, when experimentally pursuing the modelling of domains, as, for example, reported in [23, 81, 26, 200, 226, 54, 53, 31, 21, 66, 64, 88, 68, 70], **that I am often 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 modelling get's *unstuck* ! I remarked this situation to a dear friend and colleague. 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.*

The lesson is: *waver between adhering to the method and being innovative, curious – a dreamer* !

1.10.6 Domain Science & Engineering

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 *Universes of Discourse* Page 9, the author seriously think that reasonably convincing arguments are given for this *domain science & engineering*. We comment on some previous publications: [43, 71] explores additional views on analysing & describing domains, in terms of *domain facets: intrinsics*, *support technologies*, *rules & regulations*, *scripts*, *management & organisation*, and *human behaviour*. [39, 73] explores relations between Stanisław Leśhnieiski's mereology and ours. [33, 63] shows how to rigorously transform domain descriptions into software system requirements prescriptions. [59] explores relations between the present domain analysis & description approach and issues of *safety critical software design*. [62] discusses various interpretations of domain models: as bases for demos, simulators, real system monitors and real system monitor & controllers. [77] 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.

1.10.7 Comparison to Related Work[67]

[67] This section was not in [75]. It is a slightly edited version of [67, Sect. 5.3].

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. Figure 1.4 on Page 8 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 [14] 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*. 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 [232]. Description Logic [8] has been proposed as a language for the Semantic Web [9].

The interplay between endurants and perdurants is studied in [18]. 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. [18] 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 [148]. In our study 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 [148] 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 [148] 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. [148] 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 [148] 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 [156, 6, 170, 225] 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 [114]. The aim of *knowledge engineering* was formulated, in 1983, by an originator of the concept, Edward A. Feigenbaum [117] *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 [85]. 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 [10]; RDM: the relational data model [104]; and ER: entity set relationshp model [98] "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 [176], 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 [189, 190, 191]. Firstly, our understanding of *domain analysis* basically coincides with Prieto-Dĩaz’s. Secondly, in, for example, [189], 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 DSLs. 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[68] defines a *Domain-specific language* (DSL) *as a computer programming language of limited expressiveness focused on a particular domain* [119]. Other references are [169, 223]. Common to [223, 169, 119] 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 [135] 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 specific language, DSL, manifested by track layout drawings and signal interlocking tables, are translated into `SystemC` [126]. [135] 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. [135] is a great paper, but it is not solving our problem, that of systematically describing any manifest domain. [135] does, however, point a way to search for — say graphical — DSLs and the possible translation of their programs into executable ones. [135] rely on the DSL of that paper. But it does not give an analysis and a description, i.e., a semantics, of the raliway system domain. Such a description, in fact any domain analysis & description, such as we advocate, can then be a basis for one or more specific railway domain DSLs.

DSL Dogma: Domain Specific Languages

Our dogma with respect to DSL’s is: The basis for the design of any DSL, $\mathbb{DSL}$, must be a domain analysis & description of that domain, for example, as per the method of the present chapter. Based on such a domain description, $\mathscr{D}$, we can give semantics to $\mathbb{DSL}$ and somehow show that that semantics relates to $\mathscr{D}$.

[68] http://martinfowler.com/dsl.html

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 [150] 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 [107]. 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 [51] 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 [145] 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 [146] 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[69] 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 [228]. The [228] 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 [171, 216, 167]. 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

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

"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, **Chapter 4, From Domain Descriptions to Requirements Prescriptions, [63]**, that *requirements* can be "derived" systematically from, and formally related back to *domain descriptions*s 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)[70] *"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."*[71] We have studied some of the DDD literature, mostly only accessible on the Internet, but see also [136], 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 [90, 210, 147]. 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 [90, 210, 147] 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 [10, Bachman, 1969] and [98, 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 [231]. I, myself, find [154] full of very useful, non-trivial insight. [111] is seminal in that it brings a number or early contributions and views on *requirements engineering*. Conventional text books, notably [184, 188, 218] all have their "mandatory", yet conventional coverage of *requirements engineering*. None of them "derive"

[70] Eric Evans: http://www.domaindrivendesign.org/

[71] http://en.wikipedia.org/wiki/Domain-driven_design

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 [231]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 [154] can beneficially be followed also in `TripTych` *requirements development*. Neither [231]or [154] 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. The ontology work, as commented upon in Item **[1]** (Pages 69–70), is partly relevant to our work: There are at least two issues: Different classes of domains may need distinct upper ontologies. Our approach admits that there may be different upper ontologies for non-manifest domains such as *financial systems*, etcetera. This seems to warrant at least a comparative study. We have assumed, cf. Sect. 1.5.3, that attributes cannot be separated from parts. [148, Johansson 2005] develops the notion that *persisting quality instances are enduring particulars.* The issue need further clarification.

Of all the other “comparison” items (**[2]**–**[12]**) basically only Jackson’s *problem frames* (Item **[8]**) and [135] (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, [127, 146, 135], 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.

1.10.8 Tony Hoare’s Summary on ‘Domain Modelling’

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[72]:

“There are many unique contributions that can be made by domain modelling.

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.
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.”

All of these issues were covered in [30, Part IV].

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

2

Domain Facets: Analysis & Description

We[1] investigate some principles and techniques for analysing & describing domain facets.

2.1 Introduction

In Chapter 1 we outlined a *method* for analysing &[2] and describing domains. 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, not covered in Chapter 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

[1] Chapter 2 is primarily based on [71]. which itself was based on publication [43]. Introductory sections are different, but of no real consequence to this thesis. The present chapter represents the with respect to [43]: Unnumbered initial paragrapphs of [43] are not present in chapter 2. Sections 1–3 of [43] are basically omitted here. Their contents already, in another form, present in Chapter 1 of this thesis. Section 4.1 of [43] corresponds, roughly, to Sects. 2.2. Example 7, Traffic Signals, of Section 2.3 of the present chapter is new. Section 2.6 of this chapter is new wrt. [43]. Present Sect. 2.7 moved wrt. Sects. 4.4–4.5 of [43]. Example 20 of Sect. 2.7 is new.

[2] 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.

[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.

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 is a rather complete rewrite of [43]. The reason for the rewriting is the expected publication of [76]. [43] was finalised already in 2006, 10 years ago, before the analysis & description calculus of [76] had emerged. It was time to revise [43] rather substantially.

2.1.3 Structure of Chapter

The structure of this 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 [63]. 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 Chapter 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* $({}^{c_|}\!\underset{c}{Y}{}^{c_/})$ *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} :

$$\begin{array}{l}\{\{\},\\ \{(c,c_|)\},\{(c_|,c)\},\{(c,c_|),(c_|,c)\},\\ \{(c,c_/)\},\{(c_/,c)\},\{(c,c_/),(c_/,c)\},\{(c_/,c),(c_|,c)\},\\ \{(c,c_|),(c_|,c),(c_/,c)\},\{(c,c_/),(c_/,c),(c_|,c)\},\{(c_/,c),(c,c_|)\},\{(c,c_/),(c_|,c)\}\}\end{array}$$

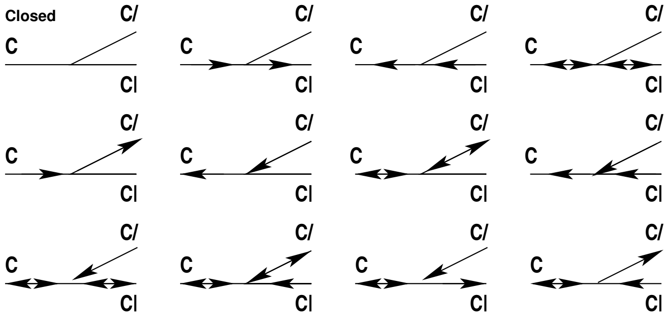


Fig. 2.1. Possible states of a rail switch

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 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. Chapter 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*

[4] to other editors, readers, etc.

[5] i.e., prevention of future operations

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. We refer to experimental technical research report [68]. ■

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

Chapter 5 illustrates requirements "derived" from the intrinsics of a road transport system – as outlined in Chapter 1. So the presentchapter has little to add to the subject of requirements "derived" from intrinsics.

2.2.3 On Modeling Intrinsics

Chapter 1 outlines 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 [121]), model-oriented specification languages (like Alloy [143], B [1], VDM-SL [83, 84, 118], RSL [123], or Z [236]), event-based languages (like Petri nets or [202] or CSP [137], respectively in process-based specification languages (like MSCs [142], LSCs [133], Statecharts [132], or CSP [137]. 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 [179].

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 [161].

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.* ■

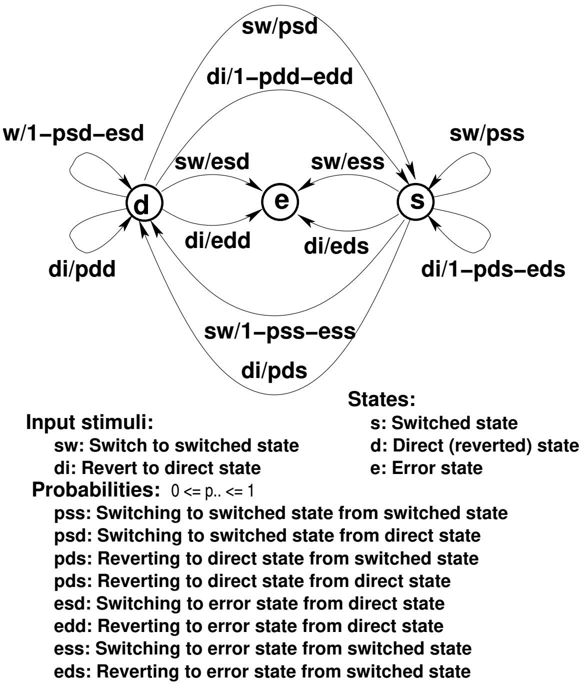

sw/psd
di/1−pdd−edd
w/1−psd−esd
sw/pss
sw/esd
sw/ess
d
e
s
di/edd
di/eds
di/pdd
di/1−pds−eds
sw/1−pss−ess
di/pds
States:
s: Switched state
d: Direct (reverted) state
e: Error state
Input stimuli:
sw: Switch to switched state
di: Revert to direct state
Probabilities: 0 <= p.. <= 1
pss: Switching to switched state from switched state
psd: Switching to switched state from direct state
pds: Reverting to direct state from switched state
pds: Reverting to direct state from direct state
esd: Switching to error state from direct state
edd: Reverting to error state from direct state
ess: Switching to error state from switched state
eds: Reverting to error state from switched state

Fig. 2.2. Probabilistic state switching

Example 6 **Traffic Signals:** *A traffic signal represents a technology in support of visualising hub states (transport net road intersection signaling states) and in effecting state changes.*

138 *A traffic signal, ts:TS, is here*[7] *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.*

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

[7] In Chapter 1 a traffic signal was an attribute of a hub.

139 *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.*

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

141 *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").*

142 *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.*

143 *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
138  TS ≡ H, HΣ, HΩ
value
139  attr_HΣ: H,TS → HΣ
139  attr_HΩ: H,TS → HΩ
type
140  HΣ = Htriple-set
140  HΩ = HΣ-set
141  Htriple = LI×Colour×LI
axiom
139  ∀ ts:TS • attr_HΣ(ts) ∈ attr_HΩ(ts)
type
141  Colour == red | yellow | green
142  Signaling = (HΣ×TI)*
142  TI
143  Sempahore = (HΣ×HΣ) →m Signalling
value
143  attr_Semaphore:TS → Sempahore
```

144 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.

145 We can think of the change of hub states as taking place based the result of some internal, non-deterministic choice.

```
value
144.  hub: HI × LI-set × (HΩ×Semaphore) → HΣ in ... out ... Unit
144.  hub(hi,lis,(hω,sema))(hσ) ≡
144.      ...
145.      ⌈⌉ let hσ′:HI • ... in hub(hi,lis,(hω,sema))(signaling(hσ,hσ′)) end
144.      ...
144.      pre: {hσ,hσ′} ⊆ hω
```

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

146 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.
147 Now *signaling* is a sequence of one or more successful hub state changes.

```
value
146   signaling: (HΣ×HΣ) × Semaphore → HΣ → HΣ
147   signaling(hσ_init,hσ_end,sema)(hσ) ≡ let sg = sema(hσ_init,hσ_end) in signal_sequence(sg)(hσ) end
147   pre hσ_init = hσ ∧ (hσ_init,hσ_end) ∈ 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.

```
147   signal_sequence(⟨⟩)(hσ) ≡ hσ
147   signal_sequence(⟨(hσ‘,ti)⟩⌢sg)(hσ) ≡ 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 [135]. ■

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 →m P)
   iTF = T → NetTraffic
   sTF = T →m NetTraffic
   oG = iTF ~→ sTF
value
   close: NetTraffic × TN × NetTraffic ~→ 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 [63] 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 [76]. 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[8], and the hydraulic "technologies" of Moorish Spain[9] to the centrifugal and gear pumps of the early industrial age, etcetera, The techniques – to mention those that have influenced this author – of [240, 149, 178, 135] 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 [202] or CSP) [137], respectively process-based specification languages (like MSCs, [142], LSCs [133], Statecharts [132], or CSP) [137], as well as temporal languages (like the Duration Calculus and [240] and Temporal Logic of Actions, TLA+) [153]).

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.

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:*

[8] https://en.wikipedia.org/wiki/Su_Song

[9] http://www.islamicspain.tv/Arts-and-Science/The-Culture-of-Al-Andalus/Hydraulic-Technology.htm

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: $\Theta \to \Theta$, 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, $(\Theta \times \Theta) \to$ **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 = Θ → Θ
    RUL = (Θ × Θ) → Bool
value
    meaning: Stimulus → STI
    meaning: Rule → RUL
    valid: Stimulus × Rule → Θ → Bool
    valid(sy_sti,sy_rul)(θ) ≡ 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 $= (\Theta \times \Theta) \to$ **Bool**, and a domain configuration-changing function, act_reg:Act_REG, where Act_REG $= \Theta \to \Theta$, 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 $\theta' =$ (meaning(sy_sti))(θ). Let θ' violate the rule,

∼valid(sy_sti,sy_rul)(θ), then if predicate part, pre_reg, of the meaning of the regulation, sy_reg, holds in that violating next configuration, pre_reg(θ,(meaning(sy_sti))(θ)), then the action part, act_reg, of the meaning of the regulation, sy_reg, must be applied, act_reg(θ), 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 θ:Θ 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 [143], B or event-B [1], RSL [123], VDM-SL [83, 84, 118], and Z [236]). In some cases it may be relevant to model using some constraint satisfaction notation [3] or some Fuzzy Logic notations [230].

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))($\rho$,$\alpha$,$\mu$,$\ell$) ≡
        let (b,d′) = $\ell$(m) in
        if $\alpha$(a)≥p
            then
                let i = interest(mi,b,period(d,d′)),
                    $\ell'$ = $\ell$ † [m↦$\ell$(m)−(p−i)]
                    $\alpha'$ = $\alpha$ † [a↦$\alpha$(a)−p,$a_i$↦$\alpha$($a_i$)+i] in
                (($\rho$,$\alpha'$,$\mu$,$\ell'$),ok) end
            else
                (($\rho$,$\alpha'$,$\mu$,$\ell$),nok)
        end end
        pre c ∈ dom $\mu$ ∧ a ∈ dom $\alpha$ ∧ m ∈ $\mu$(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 [103]. [103] 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 [13, 129, 204, 213, 227, 235]. [29, 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 [170]*

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, [15, 5, 99, 100, 101, 141, 97, 128, 131, 158, 174, 172, 160, 152, 211, 199, 198, 2, 175], 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 chapter we shall not touch upon the technical issues of "downloading"(whether "streaming" or copying, or other). That and other issues should be analysed in [237].*

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[10] 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.

[10] 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)*[11]*, 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.[12] *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.*

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

[12] 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[13] *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.)*

[13] 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 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[14]*; 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.*

[14] In the UK: the NHS, 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.

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:**

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.

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

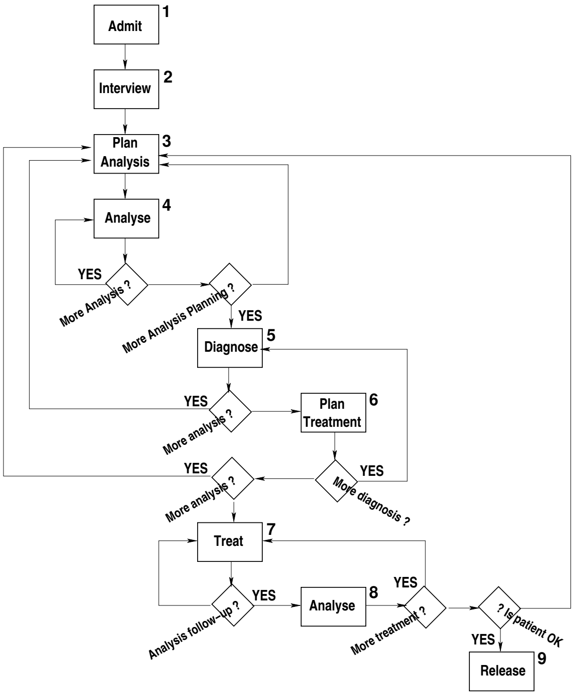


Fig. 2.3. An example single-illness non-fatal hospitalisation plan. States: {1,2,3,4,5,6,7,8,9}

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** $\{c_{c_1}, c_{c_2}, \ldots, c_{c_c}\}$ **to perform actions** $\{$CREATE, EDIT, READ, COPY, FREEZE, SHRED$\}$ **on documents** $\{d_{d_1}, d_{d_2}, \ldots, d_{d_d}\}$. *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 δ_d : τ-t or advanced δ_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** $\{\mathbf{e}_{e_1}, \mathbf{e}_{e_2}, \ldots, \mathbf{e}_{e_e}\}$. *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 [137]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)*.

•••

Chapter 1 outlined the general principle, techniques and tools for analysing & decribing discrete, composite endurants. Organisations and the management of these form such composite endurants. We shall therefore, really, not have much really new to add in this section !

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.*[15] *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.

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 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

[15] 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.

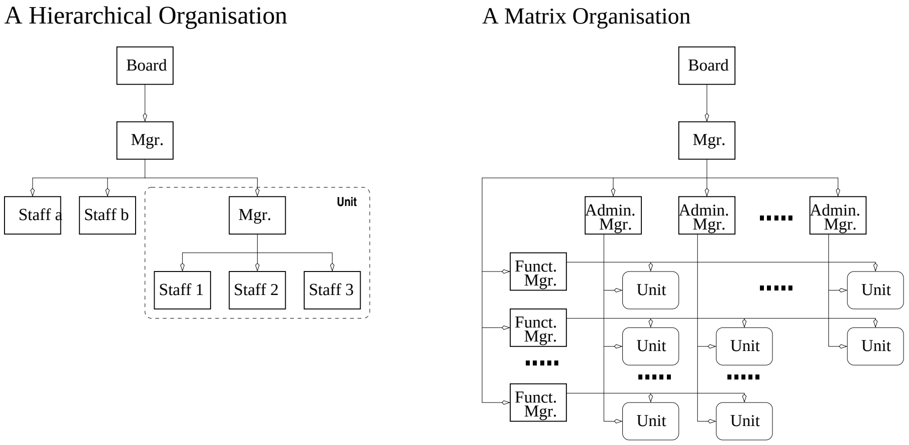


Fig. 2.4. Organisational structures

```
    Msg, Ψ, Σ, Sx
    SΣ = Sx →m Σ
channel
    { ms[i]:Msg | i:Sx }
value
    sσ:SΣ, ψ:Ψ

    sys: Unit → Unit
    sys() ≡ ‖ { 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 (⌈⌉).

```
    mg: Ψ → in,out {ms[i]|i:Sx} Unit
    mgr(ψ) ≡
(1)     let (ψ′,m)=m_out(ψ) in ‖ {ms[i]!m|i:Sx};mgr(ψ′) end
        ⌈⌉
(2)     let ψ′ = ▯ {let m=ms[i]? in m_in(i,m)(ψ) end|i:Sx} in mgr(ψ′) end

    m_out: Ψ → Ψ × MSG,
    m_in: Sx × MSG → Ψ → Ψ
```

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 (⌈⌉).

```
    stf: i:Sx → Σ → in,out ms[i] Unit
    stf(i)(σ) ≡
(1)   let m = ms[i]? in stf(i)(stf_in(m)(σ)) end
```

```
      $\sqcap$
(2)  let $(\sigma'$,m) = st_out($\sigma$) in ms[i]!m; stf(i)($\sigma'$) 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.  $\Sigma$, $\Sigma_s,\Sigma_t,\Sigma_o,\Sigma_u,\Sigma_e,\Sigma_w$
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 $\sigma'_s$ = stra(str($\sigma_s$))($\sigma'_t,\sigma'_o,\sigma'_u,\sigma'_e,\sigma'_w$),
7.        $\sigma'_t$ = tact(tac($\sigma_t$))($\sigma'_s,\sigma'_o,\sigma'_u,\sigma'_e,\sigma'_w$),
8.        $\sigma'_o$ = oper(opr($\sigma_o$))($\sigma'_s,\sigma'_t,\sigma'_u,\sigma'_e,\sigma'_w$),
9.        $\sigma'_u$ = supr(sup($\sigma_u$))($\sigma'_s,\sigma'_t,\sigma'_o,\sigma'_e,\sigma'_w$),
10.       $\sigma'_e$ = team(tea($\sigma_e$))($\sigma'_s,\sigma'_t,\sigma'_o,\sigma'_u,\sigma'_w$),
11.       $\sigma'_w$ = work(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* objective*s 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* ■

Although we otherwise do not go into any depth with respect to the analysis & description of humans, we shall momentarily depart from this “abstinence”.

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))$(\rho,\alpha,\mu,\ell) \equiv$
    let (b,d$'$) $= \ell$(m) in
    if q($\alpha$(a),p) [$\alpha$(a)$\leq$p$\vee\alpha$(a)$=$p$\vee\alpha$(a)$\leq$p$\vee$...]
        then
            let i $=$ f$_1$(interest(mi,b,period(d,d$'$))),
                $\ell' = \ell \dagger$ [m$\mapsto$f$_2$($\ell$(m)$-$(p$-$i))],
                $\alpha' = \alpha \dagger$ [a$\mapsto$f$_3$($\alpha$(a)$-$p),$a_i\mapsto$f$_4$($\alpha(a_i)$+i),$a_{\text{“staff”}}\mapsto$f$_{\text{“staff”}}$($\alpha(a_{\text{“staff”}})$+i)] in
            (($\rho,\alpha',\mu,\ell'$),ok) end
        else
            (($\rho,\alpha',\mu,\ell$),nok)
    end end
    pre c $\in$ dom $\mu \wedge$ m $\in \mu$(c)

q: P $\times$ P $\tilde{\rightarrow}$ Bool
f$_1$,f$_2$,f$_3$,f$_4$,f$_{\text{“staff”}}$: P $\tilde{\rightarrow}$ P  [typically: f$_{\text{“staff”}} = \lambda$p.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 = $\Theta \overset{\sim}{\rightarrow} \Theta$-infset
value
    hum_int: Rule → $\Theta$ → RUL-infset
    action: Stimulus → $\Theta$ → $\Theta$
    hum_beha: Stimulus × Rules → Action → $\Theta \overset{\sim}{\rightarrow} \Theta$-infset
    hum_beha(sy_sti,sy_rul)($\alpha$)($\theta$) as $\theta$set
        post
            $\theta$set = $\alpha(\theta)$ ∧ action(sy_sti)($\theta$) ∈ $\theta$set
             ∧ ∀ $\theta'$:$\Theta$•$\theta'$ ∈ $\theta$set ⇒
               ∃ se_rul:RUL•se_rul ∈ hum_int(sy_rul)($\theta$)⇒se_rul($\theta$,$\theta'$)
```

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` [121] and `RSL` [123]) 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 [4] 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 [76, 63] and in the present chapter and reports [71, 65]. 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 [76, 63].

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:

[63] 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. [51, *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[17]

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.

2.10 Bibliographical Notes

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 [27, 29, 30].

[17] In the early-to-mid 2000s there were a rush of research foundations and scientists enumerating *"Grand Challenges of Informatics"*

3

Towards Formal Models of Processes and Prompts

We[1] sketch an approach to a formal semantics of the domain analysis & description process of Chapter 1.

3.1 Introduction

Chapter 1 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.

The presentation of a calculus for analysing and describing manifest domains, introduced in Chapter 1 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[2] 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[3]), the meaning of the evolving description, and thereby the relation between the two.

[1] Chapter 3 is primarily based on [57]. which evolved into [61]. The present chapter presents an analysis & description process and prompt semantics model based on the domain ontology and the analysis & description process as presented in [57]. It may not be exactly the domain ontology of this thesis; but the differences are not substantial enough, we think, to warrant a rewrite of the formulas of the present chapter.

[2] in contrast to a formal mathematical meaning

[3] 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, [67], 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'.

3.1.1 Related Work

To this author's knowledge there are not many papers, other than the author's own, [75, 67, 71, 63, 62] 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: [180, Leon Osterweil]. As the title of that paper reveals: *"Software Processes Are Software Too"* the emphasis is on 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.[4] 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 [11, 173]. The calculus here is a set of refinement rules, a *Refinement Calculus*[5], 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 [67] "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 [67, Sects. 2.–3., Pages 7–29.]. 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

We refer to Chapter 1

Both [57] and [61] brought at this point extensive sections on the *analysis & description method* of this thesis, i.e., Chapter 1. Here we just refer to that chapter.

[4] The **RAISE** **S**pecification **L**anguage [124] does have a proof system.

[5] 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 calculus all.html&menu=3.

3.3 Syntax and Semantics

3.3.1 Form and Content

Sections 1.4, 1.5 and 1.8 [Chapter 1] appears to be expressed in the syntax of the **R**aise [124] **S**pecification **L**anguage, **RSL** [123]. 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:

a. `is_ entity`, 10
b. `is_ endurant`, 10
c. `is_ perdurant`, 11
d. `is_ discrete`, 11
e. `is_ continuous`, 11
f. `is_ physical_ part`, 12
g. `is_ living_ species`, 12
h. `is_ structure`, 13
i. `is_ part`, 14
j. `is_ atomic`, 14
k. `is_ composite`, 15
l. `is_ living_ species`, 15
m. `is_ plant`, 15
n. `is_ animal`, 16
o. `is_ human`, 16
p. `has_ materials`, 17
q. `is_ artefact`, 17
r. `observe_ endurant_ sorts`, 18
s. `has_ concrete_ type`, 20
t. `has_ mereology`, 25
u. `attribute_ types`, 28

and

[1] `observe_ endurant_ sorts`, 18
[2] `observe_ part_ type`, 20
[3] `observe_ material_ sorts`, 22
[4] `observe_ unique_ identifier`, 24
[5] `observe_ mereology`, 25
[6] `observe_ attributes`, 28

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 22 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 [215, 7]. 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 in 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 Chapter 1. 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 $\imath$p:P[6] 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. 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. The part and material 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_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 separate entities, and these, if not atomic, into their sub-entities. 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”[7], rambling, perhaps in an un-structured manner, across its domain, Δ, and its

[6] $\imath$ is Whitehead and Russell’s description operator [233, Principia Mathematica]: an inverted ι; plato.stanford.edu/entries/pm-notation

[7] 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”.

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.

3.4.2 A Model of the Analysis & Description Process

A Process State

148 Let Nm denote either a part or a material sort name.
149 A global variable αps will accumulate all the sort names being discovered.
150 A global variable νps will hold names of sorts that have been "discovered", but have yet to be analysed & described.

type
148. Nm = PNm | MNm
variable
149. αps := [Δnm] **type** Nm-**set**
150. 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 112. Each iteration of the "root" function, analyse_and_describe_endurant_sort(Nm,$\imath$: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.

151 The selection occurs from the global state νps (hence: ()) and changes that state (hence **Unit**).

value
151. sel_and_rem_Nm: **Unit** → Nm
151. sel_and_rem_Nm() ≡ **let** nm:Nm • nm ∈ νps **in** νps := νps \ {nm} ; nm **end**; **pre**: νps ≠ {}

A Technicality

152 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
152. NmVAL = (PNm×PVAL) | (MNm×MVAL)

Analysis & Description of Endurants

153 To analyse and describe endurants means to first
  a examine those endurants which have yet to be so analysed and described
  b by selecting (and removing from νps) a yet un-examined sort nm;
  c then analyse and describe an endurant entity ($\imath$:nm) of that sort — this analysis, when applied to composite parts, leads to the insertion of zero[8] or more sort names[9].

[8] If the sub-parts of $\imath$nm are all either atomic and have no materials or have already been analysed, then no new sort names are added to the repository νps).
[9] These new sort names are then "picked-up" for sort analysis &c. in a next iteration of the while loop.

As is indicated in Sect. 1.5.2 [Chapter 1], 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,

154 then, if it has a mereology,
  a to analyse and describe the mereology of each part sort,
155 and finally to analyse and describe the attributes of each sort.

```
value
153.    analyse_and_describe_endurants: Unit → Unit
153.    analyse_and_describe_endurants() ≡
153a.       while ∼is_empty(νps) do
153b.           let nm = sel_and_rem_Nm() in
153c.           analyse_and_describe_endurant_sort(nm,ι:nm) end end ;
154.        for all nm:PNm • nm ∈ αps do if has_mereology(nm,ι:nm)[10]
154a.             then observe_mereology(nm,ι:nm)[11] end end
155.        for all nm:Nm • nm ∈ αps do observe_attributes(nm,ι:nm)[12] end
```

The ι:nm of Items 153c, 154, 154a and 155 are crucial. The domain analyser is focused on (part or material) sort nm and is "directed" (by those items) to choose (select) an endurant (a part or a material) ι:nm of that sort.

156 To analyse and describe an endurant
  a is to find out whether it is a part. If so then it is to analyse and describe it.
  b If it instead is a material, then to analyse and describe it as a material.

```
value
156.    analyse_and_describe_endurant_sort: NmVAL → Unit
156.    analyse_and_describe_endurant_sort(nm,val) ≡
156a.       is_part(nm,val)[13] →[14] analyse_and_describe_part_sorts(nm,val),
156b.       is_material(nm,val)[15] → observe_material_part_sort(nm,val)[16]
```

157 To analyse and describe the internal qualities of a part
  a first describe its unique identifier.
  b If the part is atomic it is analysed and described as such;
  c If composite it is analysed and described as such.
  d Part *p* must be discrete.

```
value
157.    analyse_and_describe_part_sorts: NmVAL → Unit
157.    analyse_and_describe_part_sorts(nm,val) ≡
157a.           observe_unique_identifier(nm,val)[17];
157b.           is_atomic(nm,val)[18]→ analyse_and_describe_atomic_part(nm,val),
157c.           is_composite(nm,val)[19]→ analyse_and_describe_composite_parts(nm,val)
157d.           pre: is_discrete(nm,val)[20]
```

[12] We formalise `has_mereology` in Sect. 3.8.2 on Page 123.

[12] We formalise `observe_mereology` in Sect. 3.8.3 on Page 125.

[12] We formalise `observe_attributes` in Sect. 3.8.3 on Page 125.

[16] We formalise `is_part` in Sect. 3.8.2 on Page 122.

[16] The conditional clause: $\mathsf{cond}_1 \rightarrow \mathsf{clau}_1, \mathsf{cond}_2 \rightarrow \mathsf{clau}_2, \ldots, \mathsf{cond}_n \rightarrow \mathsf{clau}_n$
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** .

[16] We formalise `is_material` in Sect. 3.8.2 on Page 122.

[16] We formalise `observe_material_part_sort` in Sect. 3.8.3 on Page 126.

158 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
158.   analyse_and_describe_atomic_part: NmVAL → Unit
158.   analyse_and_describe_atomic_part(nm,val) ≡
158a.      if has_material(nm,val)[21] then observe_material_part_sort(nm,val)[22] end
```

159 To analyse and describe a composite endurant of sort nm (and value val)
- a is to analyse if the sort has a concrete type
- b then we analyse and describe that concrete sort type
- c else we analyse and describe the abstract sort.

```
value
159.   analyse_and_describe_composite_endurant: NmVAL → Unit
159.   analyse_and_describe_composite_endurant(nm,val) ≡
159a.      if has_concrete_type(nm,val)[23]
159b.         then observe_concrete_type(nm,val)[24]
159c.         else observe_abstract_sorts(nm,val)[25]
159a.      end
159.       pre is_composite(nm,val)[26]
```

We do not associate materials 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.

Termination

The sort name reservoir *v***ps** is "reduced" by one name in each iteration of the **while** loop of the analyse_and_describe_endurants, cf. Item 153b on Page 109, 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. [67] exemplifies axioms and sketches some proof obligations.

[20] We formalise `observe_unique_identifier` in Sect. 3.8.3 on Page 124.
[20] We formalise `is_atomic` in Sect. 3.8.2 on Page 122.
[20] We formalise `is_composite` in Sect. 3.8.2 on Page 123.
[20] We formalise `is_discrete` in Sect. 3.8.2 on Page 122.
[22] We formalise `has_material` in Sect. 3.8.2 on Page 123.
[22] We formalise `observe_part_material_sort` in Sect. 3.8.3 on Page 125.
[23] We formalise `has_concrete_type` in Sect. 3.8.2 on Page 123.
[23] We formalise `observe_concrete_type` in Sect. 3.8.3 on Page 124.
[23] We formalise `observe_part_sorts` in Sect. 3.8.3 on Page 124.
[23] We formalise `is_composite` in Sect. 3.8.2 on Page 123.

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 109 applies to the set interpretation. A list interpretation is:

value
153b. sel_and_rem_Nm: **Unit** $\rightarrow$ Nm
153b. 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, $\mathscr{D}_a$ and $\mathscr{D}_b$ are two domain description prompts where $\mathscr{D}_a$ and $\mathscr{D}_b$ 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 $\mathscr{D}_a$ is related to the description of P_a and $\mathscr{D}_b$ to P_b. Here we would expect $\mathscr{D}_a$ and $\mathscr{D}_b$ to commute, that is $\mathscr{D}_a;\mathscr{D}_b$ yields same result as does $\mathscr{D}_b;\mathscr{D}_a$. In [49] 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 [67]. 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 3 (Page 105) as well as to the discussion, *"Towards a methodology of manifest domain analysis & description"* of [67, 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[24], from "scratch", that is, having "thrown away", previous drafts[25]. 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.

•••

[24] – and if that otherwise planned, final analysis & description is not satisfactory, then yet one more iteration is taken.

[25] 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.

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. 1.4. 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 and Materials

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 and material sorts. Finally, in Sects. 3.6.1–3.6.1, we analyse this syntax with respect to a number of well-formedness criteria.

Syntax of Part and Material Sort Names

160 There is a further undefined sort, N, of tokens (which we shall consider atomic and the basis for forming names).

161 From these we form three disjoint sets of sort names:

a part sort names and

b material sort names.

$$\begin{array}{ll}
160 & \quad \mathsf{N}\\
161a & \mathsf{PNm} \ :: \mathsf{mkPNm(N)}\\
161b & \mathsf{MNm} :: \mathsf{mkMNm(N)}
\end{array}$$

An Abstract Syntax of Domain Endurants

162 We think of the types of parts and materials to be a map from their type names to respective type expressions.
163 Thus part types map part sort names into part types; and
164 material types map material sort names into material types.
165 Thus we can speak of endurant types to be either part types or material types.
166 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.
167 An atomic part type expression consists of a type expression for the qualities of the atomic part and, optionally, a material type name.
168 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.
169 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.
170 A material part type expression consists of of a type expression for the qualities of the material and an optional part type name.

Endurants: Syntactic Types

$$\begin{array}{lrcl}
162 & \mathsf{TypDef} & = & \mathsf{PTypes} \cup \mathsf{MTypes}\\
163 & \mathsf{PTypes} & = & \mathsf{PNm} \ \overrightarrow{_{m}}\ \mathsf{PaTyp}\\
164 & \mathsf{MTypes} & = & \mathsf{MNm} \ \overrightarrow{_{m}}\ \mathsf{MaTyp}\\
165 & \mathsf{ENDType} & = & \mathsf{PaTyp} \mid \mathsf{MaTyp}\\
166 & \mathsf{PaTyp} & == & \mathsf{AtPaTyp} \mid \mathsf{AbsCoPaTyp} \mid \mathsf{ConCoPaTyp}\\
167 & \mathsf{AtPaTyp} & :: & \mathsf{mkAtPaTyp(s\_qs:PQ,s\_omkn:(\{|''nil''|\}|MNn))}\\
168 & \mathsf{AbsCoPaTyp} & :: & \mathsf{mkAbsCoPaTyp(s\_qs:PQ,s\_pns:PNm}\textbf{-set}\mathsf{)}\\
168 & & & \textbf{axiom}\ \forall\ \mathsf{mkAbsCoPaTyp(pq,pns):AbsCoPaTyp} \bullet \mathsf{pns} \neq \{\}\\
169 & \mathsf{ConCoPaTyp} & :: & \mathsf{mkConCoPaTyp(s\_qs:PQ,s\_p:PNm)}\\
170 & \mathsf{MaTyp} & :: & \mathsf{mkMaTyp(s\_qs:MQ,s\_opn:(\{|''nil''|\}|PNm))}
\end{array}$$

Quality Types

171 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.
172 The type unique part identifiers is a not further defined atomic quantity.
173 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.
174 The type of attributes pairs distinct attribute names with attribute types —
175 both of which we presently leave further undefined.
176 Material attributes is the only aspect to material qualities.

Qualities: Syntactic Types

```
171          PQ  =  s_ui:UI×s_me:ME×s_atrs:ATRS}
172           UI
173          ME ==  “nil”|mkUI(s_ui:UI)|mkUIset(s_uil:UI)|...
174        ATRS  =  ANm $\xrightarrow[m]{}$ ATyp
175 ANm, ATyp
176          MQ  =  s_atrs:ATRS
```

It is without loss of generality that we do not distinguish between part and material attribute names and types. Material attributes do not refer to any part or any other material attributes.

Well-formed Syntactic Types

Well-formed Definitions

177 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 an abstract composite part type then the sort names of its parts.
  b If a concrete composite part type then the sort name is that of the sort of its set of parts.
  c If a material type then sort name is that of the sort of its optional parts.

```
value
177.   names: TypDef → (PNm|MNm) → (PNm|MNm)-set
177.   names(td)(n) ≡
177.       ∪ { ns | ns:(PNm|MNm)-set •
177.                   case td(n) of
177a.                      mkAbsCoPaTyp(_,ns′) → ns=ns′,
177b.                      mkConCoPaTyp(_,pn) → ns={pn},
177c.                      mkMaTyp(_,n′) → ns={n′}
177.                   end }
```

178 Endurant sort names being referenced in part types, PaTyp and in material types, MaTyp, of the typdef:Typdef definition, *must be defined in* the defining set, **dom** typdef, of the typdef:Typdef definition.

```
value
178.  wf_TypDef_1: TypDef → Bool
178.  wf_TypDef_1(td) ≡ ∀ n:(PNm|MNm)•n ∈ dom td ⇒ names(td)(n)⊆dom td
```

Perhaps Item 178. should be sharpened:

179 from “*must be defined in*” [178.] to “*must be equal to*”:

```
179.    ∧ ∀ n:(PNm|MNm) • n ∈ dom td ⇒ names(td)(n)=dom td
```

No Recursive Definitions

180 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 and material types. This is captured in the

- mncs (Item 167),
- pns (Item 168),
- p (Item 169) and
- pns (Item 170),

selectable elements of respective type definitions. These elements identify type names of material 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
180.  wf_TypDef_2: TypDef → Bool
180.  wf_TypDef_2(typdef) ≡ ∀ n:(PNm|MNm)• n ∈ dom typdef ⇒ n ∉ type_names(typdef)(n)

180a.  type_names: TypDef → (PNm|MNm) → (PNm|MNm)-set
180a.  type_names(typdef)(nm) ≡
180b.      let ns = names(typdef)(nm) ∪ { names(typdef)(n) | n:(PNm|MNm) • n ∈ ns } in
180c.      nm ∉ ns end
```

ns is the least fix-point solution to the recursive definition of ns.

3.6.2 Semantic Types: Parts and Materials

Part and Material Values

We define the values corresponding to the type definitions of Items 160.–176, structured as per type definition Item 165 on Page 114.

181 An endurant value is either a part value, a material values or a component value.
182 A part value is either the value of an atomic part, or of an abstract composite part, or of a concrete composite part.
183 A atomic part value has a part quality value and, optionally, either a material or a possibly empty set of component values.
184 An abstract composite part value has a part quality value and of at least (hence the **axiom**) of
185 one or more (distinct part type) part values.
186 A concrete composite part value has a part quality value and a set of part values.
187 A material value has a material quality value (of material attributes) and a (usually empty) finite set of part values.

Endurant Values: Semantic Types

```
181      ENDVAL  =  PVAL | MVAL
182        PVAL ==  AtPaVAL|AbsCoPVAL|ConCoPVAL
183     AtPaVAL  ::  mkAtPaVAL(s_qval:PQVAL,s_omkvals:({|"nil"|}|MVAL))
184 AbsCoPVAL  ::  mkAbsCoPaVAL(s_qval:PQVAL,s_pvals:(PNm →m PVAL))
185                 axiom ∀ mkAbsCoPaVAL(pqs,ppm):AbsCoPVAL • ppm ≠ []
186 ConCoPVAL  ::  mkConCoPaVAL(s_qval:PQVAL,s_pvals:PVAL-set)
187        MVAL  ::  mkMaVAL(s_qval:MQVAL,s_pvals:PVAL-set)
```

Quality Values

188 A part quality value consists of three qualities:
189 a unique identifier type name, resp. value, which are both further undefined (atomic value) tokens;
190 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

191 an aggregate of attribute values, modeled here as a map from attribute type names to attribute values.
192 In this chapter we leave attribute type names and attribute values further undefined.
193 A material quality value consists just of an aggregate of attribute values, modeled here as a map from attribute type names to attribute values.

Qualities: Semantic Types

188 PQVAL $=$ UIVAL$\times$MEVAL$\times$ATTRVALS
189 UIVAL
190 MEVAL $==$ mkUIVAL(s_ui:UIVAL)|mkUIVALset(s_uis:UIVAL-**set**)|...
191 ATTRVALS $=$ ANm $\overrightarrow{m}$ AVAL
192 ANm, AVAL
193 MQVAL $=$ 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)$\to$(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$\to$UI)|(MEVAL$\to$ME)|(AVAL$\to$ATyp)|((ANm $\overrightarrow{m}$ AVAL)$\to$(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 160–176 (Pages 113–114), and that of the *semantic domain*s, formula Items 181–187 (Pages 116–116), 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:

194 Nm_to_ENDVALS are maps from endurant sort names to respective sets of all corresponding endurant values of, and
195 ENDVAL_to_Nm are maps from endurant values to respective sort names.

type
194. $\mathsf{Nm\_to\_ENDVALS} = (\mathsf{PNm} \xrightarrow[m]{} \mathsf{PVAL}\text{-}\mathbf{set}) \cup (\mathsf{MNm} \xrightarrow[m]{} \mathsf{MVAL}\text{-}\mathbf{set})$
195. $\mathsf{ENDVAL\_to\_Nm} = (\mathsf{PVAL} \xrightarrow[m]{} \mathsf{PNm}\ ) \cup (\mathsf{MVAL} \xrightarrow[m]{} \mathsf{MNm})$

We can derive values of these map types from type definitions:

196 a function, typval, from type definitions, typdef:TypDef to Nm_to_ENDVALS, and
197 a function valtyp, from Nm_to_ENDVALS, to ENDVAL_to_Nm.

value
196. $\mathsf{typval}\colon \mathsf{TypDef} \overset{\sim}{\to} \mathsf{Nm\_to\_ENDVALS}$
197. $\mathsf{valtyp}\colon \mathsf{Nm\_to\_ENDVALS} \overset{\sim}{\to} \mathsf{ENDVAL\_to\_Nm}$

198 The typval function is defined in terms of a meaning function M (let ρ:ENV abbreviate Nm_to_ENDVALS:

198. $\mathsf{M}\colon (\mathsf{PaTyp} \to \mathsf{ENV} \overset{\sim}{\to} \mathsf{PVAL}\text{-}\mathbf{set}) | (\mathsf{MaTyp} \to \mathsf{ENV} \overset{\sim}{\to} \mathsf{MVAL}\text{-}\mathbf{set})$

196. $\mathsf{typval(td)} \equiv \mathbf{let}\ \rho = [\,\mathsf{n} \mapsto \mathsf{M(td(n))}(\rho) | \mathsf{n{:}(PNm|MNm)}\bullet\mathsf{n} \in \mathbf{dom}\ \mathsf{td}\,]\ \mathbf{in}\ \rho\ \mathbf{end}$
197. $\mathsf{valtyp}(\rho) \equiv [\,\mathsf{v} \mapsto \mathsf{n} | \mathsf{n{:}(PNm|MNm),v{:}(PVAL|MVAL)}\bullet\mathsf{n} \in \mathbf{dom}\ \rho \wedge \mathsf{v} \in \rho(\mathsf{n})\,]$

The environment, ρ, of typval, Item 196, is the least fix point of the recursive equation

- 196. $\mathbf{let}\ \rho = [\,\mathsf{n} \mapsto \mathsf{M(td(n))}(\rho) | \mathsf{n{:}(PNm|MNm)}\bullet\mathsf{n} \in \mathbf{dom}\ \mathsf{td}\,]\ \mathbf{in}\ \ldots$

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 196 and 198. The definition of M is decomposed into five sub-definitions, one for each kind of endurant type:

- Atomic parts: $\mathsf{mkAtPaTyp(s\_qs{:}(UI \times ME \times ATRS),s\_omkn{:}(\{|\text{"nil"}|\}|MNn))}$, Items 199 on the facing page;
- Abstract composite parts: $\mathsf{mkAbsCoPaTyp(s\_qs{:}PQ,s\_pns{:}PNm}\text{-}\mathbf{set})$, 200 on the next page;
- Concrete composite parts: $\mathsf{mkConCoPaTyp(s\_qs{:}PQ,s\_p{:}PNm)}$, Items 201 on Page 120; and
- Materials: $\mathsf{mkMaTyp(s\_qs{:}MQ,s\_opn{:}(\{|\text{"nil"}|\}|PNm))}$, Items 202 on Page 120.

We abbreviate, by ENV, the M function argument, ρ, of type: Nm_to_ENDVALS.

Atomic Parts

199 The meaning of an atomic part type expression,
Item 167. mkAtPaTyp((ui,me,attrs),omkn)
in mkAtPaTyp(s_qs:PQ,s_omkn:({|"nil"|}|MNn)),
is the set of all atomic part values,
Items 183., 188., 191. mkAtPaVAL((uiv,mev,attrvals),omkval)
in mkAtPaVAL(s_qval:(UIVAL×MEVAL×(ANm $\underset{m}{\rightarrow}$ 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 $\underset{m}{\rightarrow}$ AVAL) of type (ANm $\underset{m}{\rightarrow}$ ATyp), and
d omkvals is a value in ({|"nil"|}|MVAL):
  i either ''nil'',
  ii or one material value of type MNm.

```
199.      M: mkAtPaTyp((UI×ME×(ANm $\underset{m}{\rightarrow}$ ATyp))×({|"nil"|}|MVAL))→ENV$\overset{\sim}{\rightarrow}$PVAL-set
199.      M(mkAtPaTyp((ui,me,attrs),omkn))($\rho$) ≡
199.          { mkATPaVAL((uiv,mev,attrval),omkvals) |
199a.             uiv:UIVAL•type_of(uiv)=ui,
199b.             mev:MEVAL•type_of(mev)=me,
199c.             attrval:(ANm $\underset{m}{\rightarrow}$ AVAL)•type_of(attrval)=attrs,
199d.             omkvals: case omkn of
199(d)i.              "nil" → "nil",
199(d)ii.             mkMNn(_) → mval:MVAL•type_of(mval)=omkn
199d.             end }
```

Formula terms 199a–199(d)ii 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

200 The meaning of an abstract composite part type expression,
Item 168. mkAbsCoPaTyp((ui,me,attrs),pns)
in mkAbsCoPaTyp(s_qs:PQ,s_pns:PNm-**set**),
is the set of all abstract, composite part values,
Items 184., 188., 191., mkAbsCoPaVAL((uiv,mev,attrvals),pvals)
in mkAbsCoPaVAL(s_qval:(UIVAL×MEVAL×(ANm $\underset{m}{\rightarrow}$ AVAL)),s_pvals:(PNm $\underset{m}{\rightarrow}$ 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 $\underset{m}{\rightarrow}$ AVAL) of type (ANm $\underset{m}{\rightarrow}$ ATyp), and
d pvals is a map of part values in (PNm $\underset{m}{\rightarrow}$ PVAL), one for each name, pn:PNm, in pns such that these part values are of the type defined for pn.

```
200.   M: mkAbsCoPaTyp((UI×ME×(ANm $\underset{m}{\rightarrow}$ ATyp)),PNm-set) → ENV $\overset{\sim}{\rightarrow}$ PVAL-set
200.   M(mkAbsCoPaTyp((ui,me,attrs),pns))($\rho$) ≡
200.       { mkAbsCoPaVAL((uiv,mev,attrvals),pvals) |
200a.          uiv:UIVAL•type_of(uiv)=ui
200b.          mev:MEVAL•type_of(mev)=me,
200c.          attrvals:(ANm $\underset{m}{\rightarrow}$ ATyp)•type_of(attrsval)=attrs,
200d.          pvals:(PNm $\underset{m}{\rightarrow}$ PVAL)•pvals∈{[pn↦pval|pn:PNm,pval:PVAL•pn∈ pns∧pval∈$\rho$(pn)]} }
```

Concrete Composite Parts

201 The meaning of a concrete composite part type expression, Item 169.
mkConCoPaTyp((ui,me,attrs),pn)
in mkConCoPaTyp(s_qs:(UI$\times$ME$\times$(ANm $\underset{m}{\rightarrow}$ ATyp)),s_pn:PNm),
is the set of all concrete, composite *set* part values,
Item 186. mkConCoPaVAL((uiv,mev,attrvals),pvals)
in mkConCoPaVAL(s_qval:(UIVAL$\times$MEVAL$\times$(ANm $\underset{m}{\rightarrow}$ 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 $\underset{m}{\rightarrow}$ 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.

201. M: mkConCoPaTyp((UI$\times$ME$\times$(ANm $\underset{m}{\rightarrow}$ ATyp))$\times$PNm) $\rightarrow$ ENV $\overset{\sim}{\rightarrow}$ PVAL-**set**
201. M(mkConCoPaTyp((ui,me,attrs),pn))(ρ) $\equiv$
201. { mkConCoPaVAL((uiv,mev,attrvals),pvals) |
201a. uiv:UIVAL•type_of(uiv)=ui,
201b. mev:MEVAL•type_of(mev)=me,
201c. attrsval:(ANm $\underset{m}{\rightarrow}$ AVAL)•type_of(attrsval)=attrs,
201d. pvals:PVAL-**set**•pvals$\subseteq\rho$(pn) }

Materials

202 The meaning of a material type, 170.,
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.

202. M: mkMaTyp(s_mq:(ANm $\underset{m}{\rightarrow}$ ATyp),s_pn:PNm) $\rightarrow$ ENV $\overset{\sim}{\rightarrow}$ MVAL-**set**
202. M(mq,pn)(ρ) $\equiv$
202. { mkMVAL(mqval,ps) |
202a. mqval:MVAL•type_of(mqval)=mq,
202b. ps:PVAL-**set**•ps$\subseteq\rho$(pn) }

3.7.4 The ι Description Function

We can now define the meaning of the syntactic clause:

- ιNm:Nm

203 ιNm:Nm “chooses” an arbitrary value from amongst the values of sort Nm:

value
203. ι nm:Nm $\equiv$ iota(nm)
203. iota: Nm $\rightarrow$ TypDef $\rightarrow$ VAL
203. iota(nm)(td) $\equiv$ **let** val:(PVAL|MVAL|KVAL)•val $\in$ (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.

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:

a. is_ entity, 10
b. is_ endurant, 10
c. is_ perdurant, 11
d. is_ discrete, 11
e. is_ continuous, 11
f. is_ physical_ part, 12
g. is_ living_ species, 12
h. is_ structure, 13
i. is_ part, 14
j. is_ atomic, 14
k. is_ composite, 15
l. is_ living_ species, 15
m. is_ plant, 15
n. is_ animal, 16
o. is_ human, 16
p. has_ materials, 17
q. is_ artefact, 17
r. observe_ endurant_ sorts, 18
s. has_ concrete_ type, 20
t. has_ mereology, 25
u. attribute_ types, 28

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 2 on Page 10 and Formula Item 156a on Page 110.

value

is_endurant: Nm×VAL → TypDef $\overset{\sim}{\rightarrow}$ **Bool**
is_endurant(_,val)(td) ≡ val ∈ **dom** valtyp(typval(td)); **pre**: VAL is any value type

is_discrete

See analysis prompt definition 4 on Page 11 and Formula Item 157d on Page 110.

value

is_discrete: NmVAL → TypDef $\overset{\sim}{\rightarrow}$ **Bool**
is_discrete(_,val)(td) ≡ (is_PaTyp|is_CoTyp)(td((valtyp(typval(td)))(val)))

is_part

See analysis prompt definition 9 on Page 14 and Formula Item 156a on Page 110.

value

is_part: NmVAL → TypDef $\overset{\sim}{\rightarrow}$ **Bool**
is_part(_,val)(td) ≡ is_PaTyp(td((valtyp(typval(td)))(val)))

is_material [≡ is_continuous]

See analysis prompt definition 5 on Page 11 and Formula Item 156b on Page 110.
We remind the reader that is_continuous≡is_material.

value

is_material: NmVAL → TypDef $\overset{\sim}{\rightarrow}$ **Bool**
is_material(_,val)(td) ≡ is_MaTyp(td((valtyp(typval(td)))(val)))

is_atomic

See analysis prompt definition 10 on Page 14 and Formula Item 157b on Page 110.

value

is_atomic: NmVAL → TypDef $\overset{\sim}{\rightarrow}$ **Bool**
is_atomic(_val)(td) ≡ is_AtPaTyp(td((valtyp(typval(td)))()))

`is_composite`

See analysis prompt definition 11 on Page 15 and Formula Item 157c on Page 110.

value

$\mathsf{is\_composite: NmVAL} \rightarrow \mathsf{TypDef} \xrightarrow{\sim} \mathbf{Bool}$

$\mathsf{is\_composite(\_,val)(td)} \equiv \mathsf{(is\_AbsCoPaTyp|is\_ConCoPaTyp)(td((valtyp(typval(td)))(val)))}$

`has_concrete_type`

See analysis prompt definition 19 on Page 20 and Formula Item 159a on Page 111.

value

$\mathsf{has\_concrete\_type: NmVAL} \rightarrow \mathsf{TypDef} \xrightarrow{\sim} \mathbf{Bool}$

$\mathsf{has\_concrete\_type(\_,val)(td)} \equiv \mathsf{is\_ConCoPaTyp(td((valtyp(typval(td)))(val)))}$

`has_mereology`

See analysis prompt definition 20 on Page 25 and Formula Item 154 on Page 110.

value

$\mathsf{has\_mereology: NmVAL} \rightarrow \mathsf{TypDef} \xrightarrow{\sim} \mathbf{Bool}$

$\mathsf{has\_mereology(\_,val)(td)} \equiv \mathsf{s\_me(td((valtyp(typval(td)))(val)))} \neq {}''\mathtt{nil}''$

`has_materials`

See analysis prompt definition 16 on Page 17 and Formula Item 158a on Page 111.

value

$\mathsf{has\_material: NmVAL} \rightarrow \mathsf{TypDef} \xrightarrow{\sim} \mathbf{Bool}$

$\mathsf{has\_material(\_,val)(td)} \equiv \mathsf{is\_MNm(s\_omkn(td((valtyp(typval(td)))(val))))}$

  pre: $\mathsf{is\_AtPaTyp(td((valtyp(typval(td)))(val)))}$

3.8.3 The Description Prompts

These are the domain description prompts to be defined:

[1] `observe_ endurant_ sorts`, 18
[2] `observe_ part_ type`, 20
[3] `observe_ material_ sorts`, 22
[4] `observe_ unique_ identifier`, 24
[5] `observe_ mereology`, 25
[6] `observe_ attributes`, 28

A Description State

In addition to the analysis state components αps and νps there is now an additional, the description text state component.

204 Thus a global variable τ will hold the (so far) generated (in this case only) formal domain description text.

variable

204. $\tau := [\,]$ **Text-set**

We shall explain the use of [...]s and the operations of $\setminus$ and $\oplus$ on the above variables in Sect. 3.4.3 on Page 112.

observe_part_sorts

See description prompt definition 1 on Page 19 and Formula Item 159c on Page 111.

```
value
    observe_part_sorts: NmVAL → TypDef → Unit
    observe_part_sorts(nm,val)(td) ≡
        let mkAbsCoPaTyp(_,{$P_1$,$P_2$,...,$P_n$}) = td((valtyp(typval(td)))(val)) in
                        $\tau$ := $\tau$ ⊕ [" type $P_1$,$P_2$,...,$P_n$;
                                        value
                                            obs_part_$P_1$nm→$P_1$
                                            obs_part_$P_2$:nm→$P_2$
                                            ...,
                                            obs_part_$P_n$:nm→$P_n$;
                                        proof obligation
                                            $\mathscr{D}$; " ]
                      ‖ $\nu$ps := $\nu$ps ⊕ ([$P_1$,$P_2$,...,$P_n$] \ $\alpha$ps)
                      ‖ $\alpha$ps := $\alpha$ps ⊕ [$P_1$,$P_2$,...,$P_n$]
        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 2 on Page 21 and Formula Item 159b on Page 111.

```
value
    observe_concrete_type: NmVAL → TypDef → Unit
    observe_concrete_type(nm,val)(td) ≡
        let  mkConCoPaTyp(_,P) = td((valtyp(typval(td)))(val)) in
                $\tau$ := $\tau$ ⊕ ["type  T = P-set ; value obs_part_T: nm→T; " ]
              ‖ $\nu$ps := $\nu$ps ⊕ ([P] \ $\alpha$ps)
              ‖ $\alpha$ps := $\alpha$ps ⊕ [P]
        end
        pre: is_ConCoPaTyp(td((valtyp(typval(td)))(val)))
```

observe_unique_identifier

See description prompt definition 4 on Page 24 and Formula Item 157a on Page 110.

```
value
    observe_unique_identifier: P → TypDef → Unit
    observe_unique_identifier(nm,val)(td) ≡
        $\tau$ := $\tau$ ⊕ [" type PI ; value uid_PI: nm → PI ; axiom $\mathscr{U}$; " ]
```

$\mathscr{U}$ is a predicate expression over unique identifiers.

observe_mereology

See description prompt definition 5 on Page 26 and Formula Item 154a on Page 110.

```
value
    observe_mereology: NmVAL → TypDef → Unit
    observe_mereology(nm,val)(td) ≡
        $\tau$ := $\tau$ ⊕ ["type MT = $\mathscr{M}$(PI1,PI2,...,PIn) ;
                    value obs_mereo_P: nm → MT ;
                    axiom $\mathscr{ME}$; " ]
        pre: has_mereology(nm,val)(td) [26]
```

$\mathscr{M}$(PI1,PI2,...,PIn) is a type expression over unique part identifiers. $\mathscr{ME}$ is a predicate expression over unique part identifiers.

observe_part_attributes

See description prompt definition 6 on Page 28 and Formula Item 155 on Page 110.

```
value
    observe_part_attributes: NmVAL → TypDef → Unit
    observe_part_attributes(nm,val)(td) ≡
      let {$A_1$,$A_2$,...,$A_a$} = dom s_attrs(s_qs(val)) in
      $\tau$ := $\tau$ ⊕ ["  type  $A_1$, $A_2$, ..., $A_a$
                    value attr_$A_1$: nm→$A_i$
                          attr_$A_2$: nm→$A_1$
                          ...
                          attr_$A_a$: nm→$A_i$
                    proof obligation [Disjointness of Attribute Types]
                          $\mathscr{A}$ ; " ]
      end
```

$\mathscr{A}$ is a predicate over attribute types A_1, A_2, ..., A_a.

observe_part_material_sort

See description prompt definition 3 on Page 22 and Formula Item 158a on Page 111.

```
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
        $\tau$ := $\tau$ ⊕ [" type M ; value obs_mat_sort_M:nm→M " ]
        ‖ $\nu$ps := $\nu$ps ⊕ ([M]\$\alpha$ps)
        ‖ $\alpha$ps := $\alpha$ps ⊕ [M]
        end
        pre: is_AtPaVAL(val) ∧ is_MNm(s_pns(td((valtyp(typval(td)))(val))))
```

[26] See analysis prompt definition 20 on Page 25

observe_material_part_sort

See description prompt definition 3 on Page 22 and Formula Item 158a on Page 111.

```
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 " ]
        ‖ νps := νps ⊕ ([P] \ αps)
        ‖ αps := αps ⊕ [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 proposes a set of domain analysis & description prompts. 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

We[1] give an abstract model of parts and part-hood relations, of Stansław Leśniewski's *mereology* [96].

4.1 Introduction

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` [137], 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, [96].

"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 [67]. 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 on the following page. 'Adjacent' parts are direct parts of a same third part, p_z, i.e., p_x and p_y are "embedded within" 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" p_z; etcetera; as loosely indicated in Fig. 4.2 on the next page, or one is "embedded within" the other — etc. as loosely indicated in Fig. 1.2 on Page 5. Parts, whether 'adjacent' or 'embedded within', can share properties. For adjacent 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 on the next page. Instead of depicting parts sharing properties as in Fig. 4.3 on the following page[L]eft, where shaded, dashed rounded-edge rectangles stands for 'sharing', we shall (eventually) show parts sharing properties as in Fig. 4.3 on the next page[R]ight where •—• connections connect those parts.

[1] This paper is a complete rewrite of [56].

[2] Achille Varzi: Mereology, http://plato.stanford.edu/entries/mereology/ 2009 and [96].

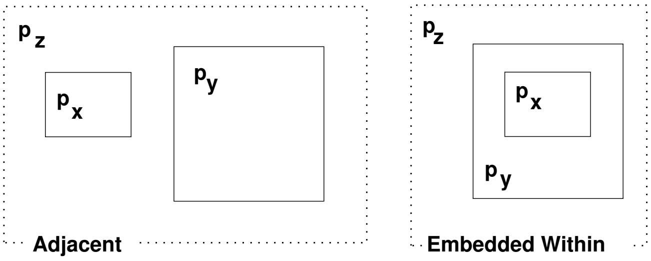


Fig. 4.1. Immediately ‘Adjacent’ and ‘Embedded Within’ Parts

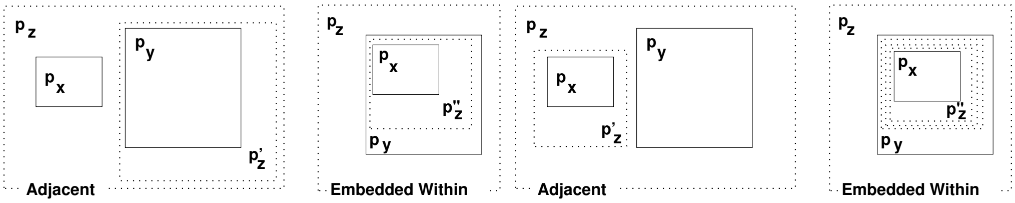


Fig. 4.2. Transitively ‘Adjacent’ and ‘Embedded Within’ Parts

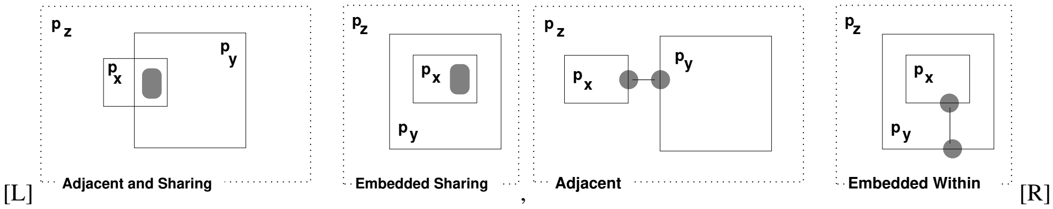


Fig. 4.3. Two models, [L,R], of parts sharing properties

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, cf. Chapter 5. Thus to properly **design** 𝕊oftware we need to **understand** its or their ℝequirements; and to properly **prescribe** ℝequirements one must **understand** its 𝔻omain. To **argue** correctness of 𝕊oftware with respect to ℝequirements one must usually **make assumptions** about the 𝔻omain: $\mathbb{D},\mathbb{S} \models \mathbb{R}$. Thus **description** of 𝔻omains become an indispensable part of 𝕊oftware 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 Chapter

A general contribution of this chapter 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` [137] program; and (iii) suggesting complementing **syntactic** and **semantic** theories of mereology.

4.1.5 Structure of Chapter

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. The abstract model as well as the axiom system of Sect. 4.5 focuses on the **syntax of mereologies.** Following that, in Sect. 4.6, **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 136, as adjacent within the pipeline system; two or more banks of a banking system, cf. Fig. 4.6 on Page 134, 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

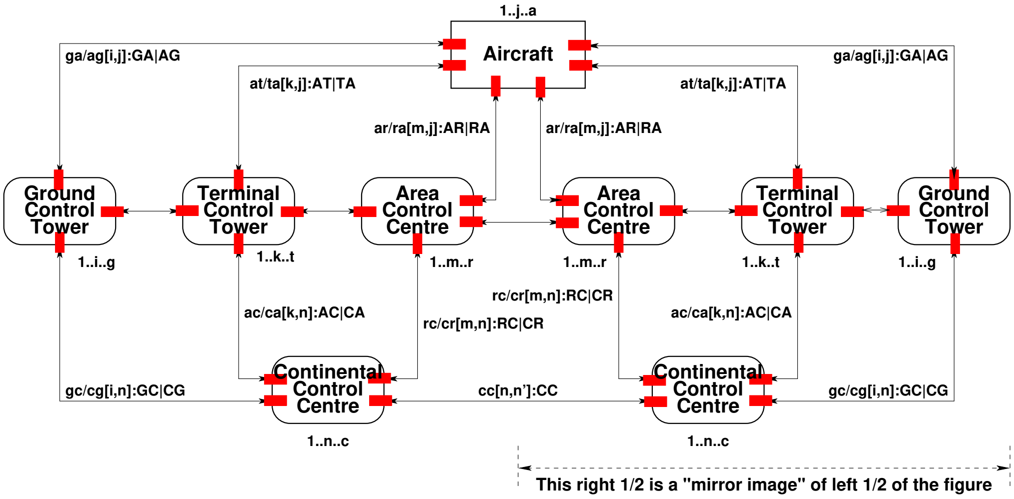


Fig. 4.4. A schematic air traffic system

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.

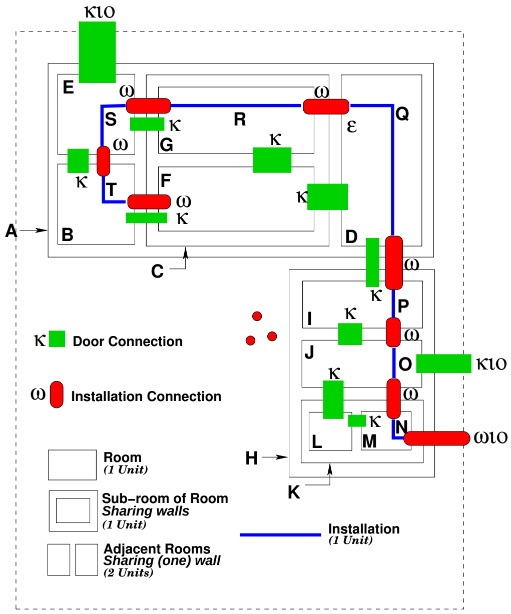


Fig. 4.5. A building plan with installation

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 "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

[3] There are 36 such rectangles in Fig. 4.4 on the facing page.

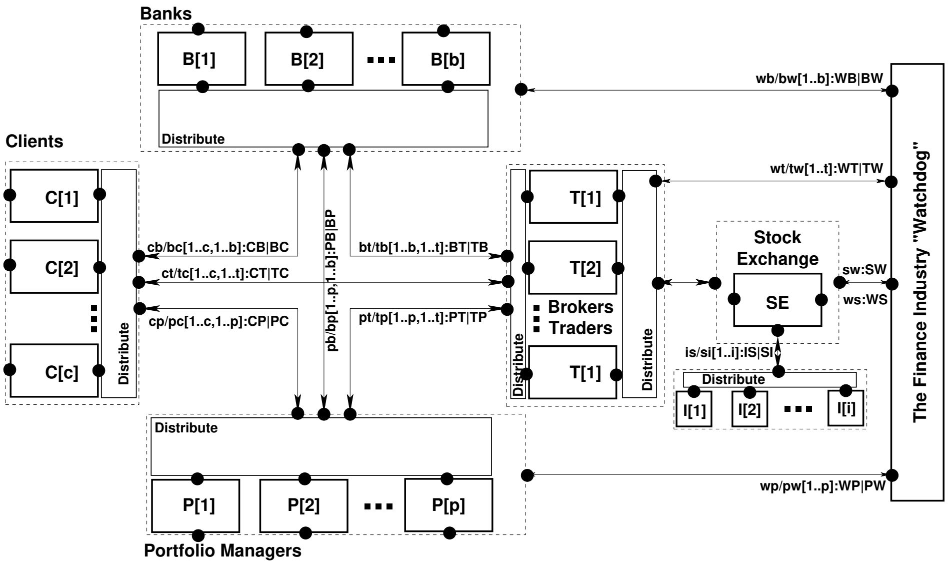


Fig. 4.6. A Financial Service Industry

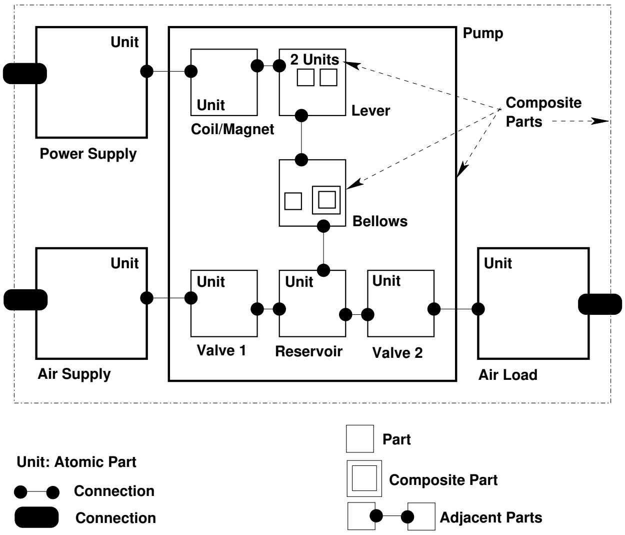


Fig. 4.7. An air pump, i.e., a physical mechanical system

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.

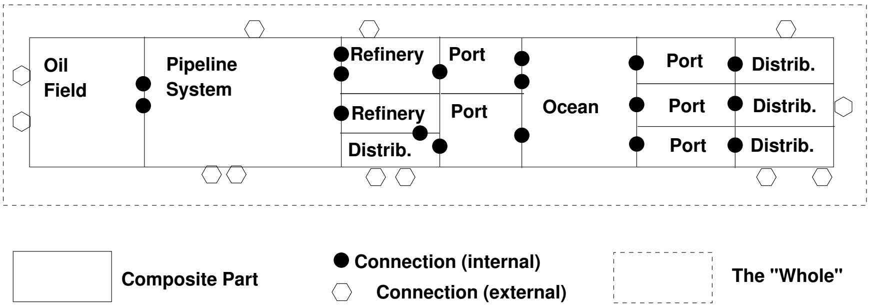


Fig. 4.8. A Schematic of an Oil Industry

Oil Industry

“The” Overall Assembly

Figure 4.8 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 on the following page 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 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 [53] 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 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 on the following page 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.

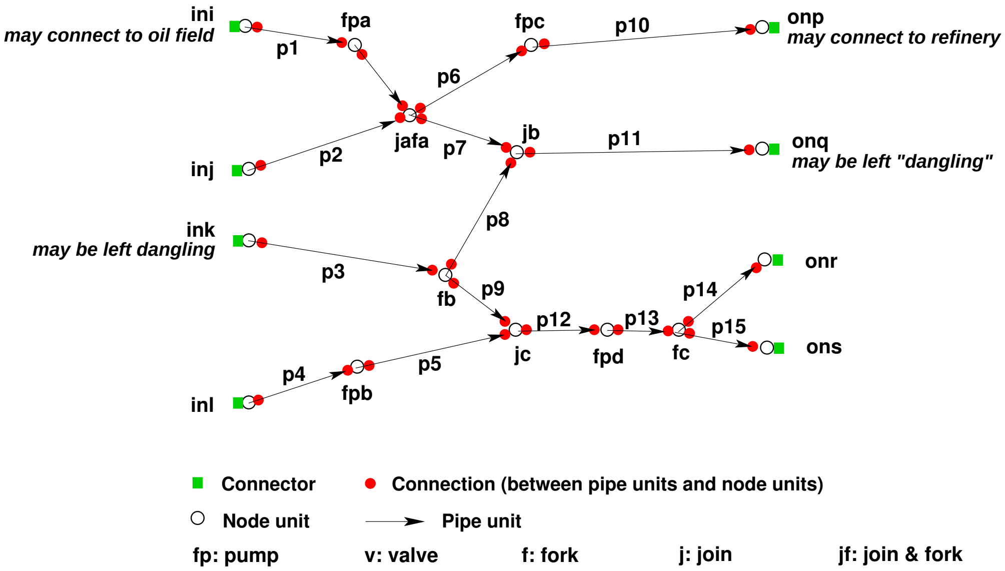


Fig. 4.9. A Pipeline System

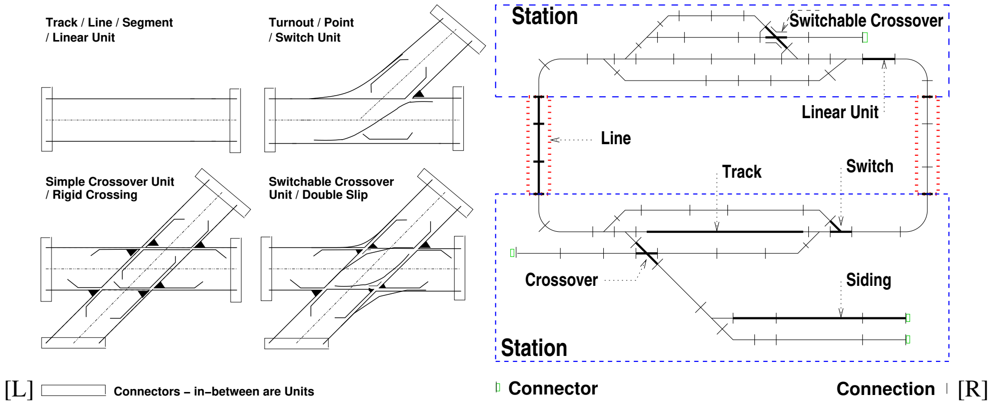


Fig. 4.10. To the left: Four rail units.To the right: A “model” railway net:
An Assembly of four Assemblies: two stations and two lines.
Lines here consist of linear rail units.
Stations of all the kinds of units shown to the left.
There are 66 connections and four “dangling” connectors

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

205 We distinguish between **atomic** and **composite parts.**
206 Atomic parts do not contain separately distinguishable parts.
207 Composite parts contain at least one separately distinguishable part.

type
205. P == AP | CP[4]
206. AP :: mkAP(...)[5]
207. 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 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".

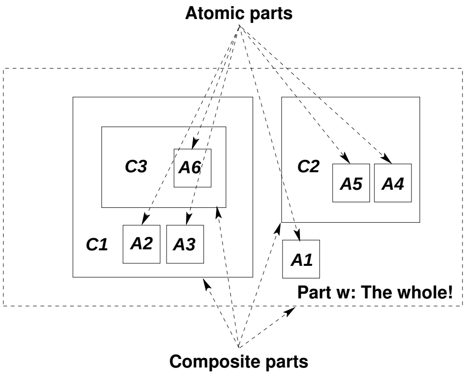


Fig. 4.11. Atomic and Composite Parts

4.3.2 No "Infinitely" Embedded Parts

The above syntax, Items 205–207, does not prevent composite parts, p, to contain composite parts, p', "ad-infinitum" ! But we do not wish such "recursively" contained parts !

208 To express the property that parts are finite we introduce a notion of *part derivation*.
209 The part derivation of an atomic part is the empty set.
210 The part derivation of a composite part, p, mkC(...,ps) where ... is left undefined, is the set ps of subparts of p.

[4] In the RAISE [124] Specification Languge, RSL [123], writing type definitions X == Y|Z means that Y and Z are to be disjoint types. In Items 206.–207. 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. [164, 165].

[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. [164, 165].

```
value
208.  pt_der: P → P-set
209.  pt_der(mkAP(...)) ≡ {}
210.  pt_der(mkCP(...,ps)) ≡ ps
```

211 We can also express the part derivation, pt_der(ps) of a set, ps, of parts.
212 If the set is empty then pt_der({}) is the empty set, {}.
213 Let mkA(pq) be an element of ps, then pt_der({mkA(pq)}∪ps′) is ps′.
214 Let mkC(pq,ps′) be an element of ps, then pt_der(ps′∪ps) is ps′.

```
211.   pt_der: P-set → P-set
212.   pt_der({}) ≡ {}
213.   pt_der({mkA(..)}∪ps) ≡ ps
214.   pt_der({mkC(..,ps')}∪ps) ≡ ps'∪ps
```

215 Therefore, to express that a part is finite we postulate
216 a natural number, *n*, such that a notion of iterated part set derivations lead to an empty set.
217 An iterated part set derivation takes a set of parts and part set derive that set repeatedly, *n* times.
218 If the result is an empty set, then part *p* was finite.

```
value
215.  no_infinite_parts: P → Bool
216.  no_infinite_parts(p) ≡
216.      ∃ n:Nat • it_pt_der({p})(n)={}
217.  it_pt_der: P-set → Nat → P-set
218.  it_pt_der(ps)(n) ≡
218.      let ps′ = pt_der(ps) in
218.      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.

219 In order to refer to specific parts we endow all parts, whether atomic or composite, with **u**nique **id**entifications.
220 We postulate functions which observe these **u**nique **id**entifications, whether as parts in general or as atomic or composite parts in particular.
221 such that any to parts which are distinct have **u**nique **id**entifications.

```
type
219.  UI
value
220.  uid_UI: P → UI
axiom
221.  ∀ 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
206.  AP :: mkA(s_pq:(s_uid:UI,...))
207.  CP :: mkC(s_pq:(s_uid:UI,...),s_sps:P-set)
value
220.  uid_UI(mkA((ui,...))) ≡ ui
220.  uid_UI(mkC((ui,...)),...) ≡ ui
```

Figure 4.12 illustrates the unique identifications of composite and atomic parts.

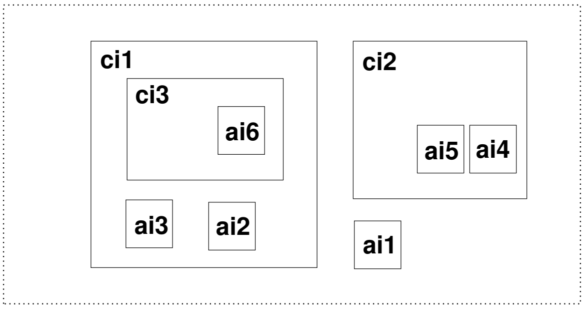


Fig. 4.12. ai_j: atomic part identifiers, ci_k: composite part identifiers

No two parts have the same unique identifier.

222 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*.
223 no_prts_uis is defined in terms of yet an auxiliary function, sum_no_pts_uis.

```
value
222.  no_prts_uis: P → (Nat × UI-set) → (Nat × UI-set)
222.  no_pts_uis(mkA(ui,...))(n,uis) ≡ (n+1,uis∪{ui})
222.  no_pts_uis(mkC((ui,...),ps))(n,uis) ≡
222.      let (n′,uis′) = sum_no_pts_uis(ps) in
222.      (n+n′,uis∪uis′) end
222.  pre: no_infinite_parts(p)
223.  sum_no_pts_uis: P-set → (Nat × UI-set) → (Nat × UI-set)
223.  sum_no_pts_uis(ps)(n,uis) ≡
223.    case ps of
223.      {}→(n,uis),
223.      {mkA(ui,...)}∪ps′→sum_no_pts_uis(ps′)(n+1,uis∪{ui}),
223.      {mkC((ui,...),ps′)}∪ps″ →
223.         let (n″,uis″)=sum_no_pts_uis(ps′)(1,{ui}) in
223.         sum_no_pts_uis(ps″)(n+n″,uis∪uis″) end
223.    end
223.    pre: ∀ p:P•p ∈ ps ⇒ no_infinite_parts(p)
```

224 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
224.  ∀ p:P • let (n,uis)=no_prts_uis(0,{}) in n=card uis end
```

4.3.4 Attributes

Attribute Names and Values

225 Parts have sets of named attribute values, attrs:ATTRS.
226 One can observe attributes from parts.
227 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
225.  ANm, AVAL, ATTRS = ANm $\xrightarrow[m]{}$ AVAL
value
226.  attr_ATTRS: P → ATTRS
227.  share: P×P → Bool
227.  share(p,p′) ≡
227.      p≠p′ ∧ ∼trans_adj(p,p′) ∧
227a.     ∃ anm:ANm • anm ∈ dom attr_ATTRS(p) ∩ dom attr_ATTRS(p′) ⇒
227b.         □ (attr_ATTRS(p))(anm) = (attr_ATTRS(p′))(anm)
```

The function trans_adj is defined in Sect. 4.4.4 on Page 143.

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:

228 $\mathscr{S}_{\mathscr{A}}$ applies to attrs:ATTRS and yields a grouping (sa$_1$,sa$_2$,...,sa$_{n_s}$)[7], of **static** attribute values.
229 $\mathscr{C}_{\mathscr{A}}$ applies to attrs:ATTRS and yields a grouping (ca$_1$,ca$_2$,...,ca$_{n_c}$)[8] of **controllable** attribute values.
230 $\mathscr{E}_{\mathscr{A}}$ applies to attrs:ATTRS and yields a set, {eA$_1$,eA$_2$,...,eA$_{n_e}$}[9] of **external** attribute names.

[7] – where {sa$_1$,sa$_2$,...,sa$_{n_s}$}⊆**rng** attrs
[8] – where {ca$_1$,ca$_2$,...,ca$_{n_s}$}⊆**rng** attrs
[9] – where {eA$_1$,eA$_2$,...,eA$_{n_e}$}⊆**dom** attrs

type
  SA,CA = AVAL*
  EA = ANm−st
value
  228. $\mathscr{S}_{\mathscr{A}}$: ATTRS → SA
  229. $\mathscr{C}_{\mathscr{A}}$: ATTRS → CA
  230. $\mathscr{E}_{\mathscr{A}}$: 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.

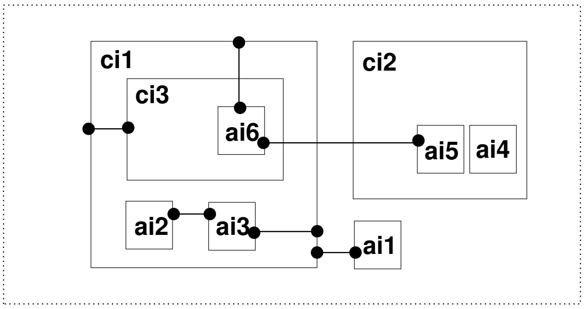


Fig. 4.13. Mereology: Relations between Parts

231 The mereology of a part is a set of unique identifiers of other parts.

type
231. 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:

- $\{ci_1, ci_3\}$,
- $\{ai_2, ai_3\}$,
- $\{ai_6, ci_1\}$,
- $\{ai_3, ci_1\}$,
- $\{ai_6, ai_5\}$ and
- $\{ai_1, ci_1\}$.

4.3.6 The Model

232 The “whole” is a part.
233 A part value has a part sort name and is either the value of an atomic part or of an abstract composite part.
234 An atomic part value has a part quality value.
235 An abstract composite part value has a part quality value and a set of at least of one or more part values.
236 A part quality value consists of a unique identifier, a mereology, and a set of one or more attribute named attribute values.

```
232   W = P
233    P = AP | CP
234   AP :: mkA(s_pq:PQ)
235   CP :: mkC(s_pq:PQ,s_ps:P-set)
236   PQ = UI×ME×(ANm $\xrightarrow[m]{}$ 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’

237 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
237.  imm_within: P × P → Bool
237.  imm_within(p,p′) ≡
237.      case p′ of
237.          (_,mkA(_,ps)) → p ∈ ps,
237.          (_,mkC(_,ps)) → p ∈ ps,
237.          _ → false
237.      end
```

4.4.2 ‘Transitive Within’

We can generalise the ‘immediate within’ property.

238 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
238.   trans_wihin: P × P → Bool
238.   trans_within(p,p′) ≡
238a.      imm_within(p,p′)
238b.      ∨
238c.      case p′ of
238c.          (_,mkC(_,ps)) → p ∈ ps ∧
238c.              ∃ p″:P• p″ ∈ ps ∧ trans_within(p″,p),
238c.          _ → false
238.      end
```

4.4.3 ‘Adjacency’

239 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
239.   imm_adj: P × P → P → Bool
239.   imm_adj(p,p′)(mkA(__,ps)) ≡ p≠p′ ∧ {p,p′}⊆ps
239.   imm_adj(p,p′)(mkC(__,ps)) ≡ p≠p′ ∧ {p,p′}⊆ps
239.   imm_adj(p,p′)(mkA(__)) ≡ false
```

4.4.4 Transitive ‘Adjacency’

We can generalise the immediate ‘adjacent’ property.

240 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 $\mathscr{P}$arts. 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: $\mathscr{P}$arts, and $\mathscr{A}$ttributes.[10]

- **type** $\mathscr{P}$,$\mathscr{A}$

$\mathscr{A}$ttributes are associated with $\mathscr{P}$arts. 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 $\mathscr{P}$arts and $\mathscr{A}$ttributes.

- $\in$: $\mathscr{A} \times \mathscr{P} \to$ **Bool**.

4.5.2 The Axioms

The axiom system to be developed in this section is a variant of that in [96]. We introduce the following relations between parts:

part_of:	$\mathbb{P} : \mathscr{P} \times \mathscr{P} \to$ **Bool**	Page 144
proper_part_of:	$\mathbb{PP} : \mathscr{P} \times \mathscr{P} \to$ **Bool**	Page 144
overlap:	$\mathbb{O} : \mathscr{P} \times \mathscr{P} \to$ **Bool**	Page 144
underlap:	$\mathbb{U} : \mathscr{P} \times \mathscr{P} \to$ **Bool**	Page 144
over_crossing:	$\mathbb{OX} : \mathscr{P} \times \mathscr{P} \to$ **Bool**	Page 144
under_crossing:	$\mathbb{UX} : \mathscr{P} \times \mathscr{P} \to$ **Bool**	Page 144
proper_overlap:	$\mathbb{PO} : \mathscr{P} \times \mathscr{P} \to$ **Bool**	Page 144
proper_underlap:	$\mathbb{PU} : \mathscr{P} \times \mathscr{P} \to$ **Bool**	Page 144

[10] Identifiers P and A stand for model-oriented types (parts and atomic parts), whereas identifiers $\mathscr{P}$ and $\mathscr{A}$ 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

241 P as the meaning of $\mathscr{P}$
242 ATR as the meaning of $\mathscr{A}$,
243 imm_within as the meaning of $\mathbb{P}$,
244 trans_within as the meaning of $\mathbb{PP}$,
245 ∈: ATTR×ATTRS-**set**→**Bool** as the meaning of ∈: $\mathscr{A} \times \mathscr{P} \rightarrow$**Bool** and
246 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×ATTRS-**set**→**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 [137].

4.7.1 Parts ≃ 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.

247 Let w be the "whole" (i.e., a part).
248 Let uis be the set of all unique identifiers of the "whole".
249 Let M be the type of messages sent over channels.
250 Channels provide means for processes to synchronise and communicate.

```
value
247.  w:P
248.  uis = let (_,uis′)=no_prts_uis(w) in uis′ end
type
249.  M
channel
250.  {ch[{ui,ui′}]:M|ui,ui′:UI•ui≠ui′ ∧ {ui,ui′}⊆uis}
```

251 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
251. {xch[ui,an]:AVAL|ui:UI• ui ∈ 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 140) that $\mathscr{C}_\mathscr{A}$(attrs) is a grouping, (ca_1,ca_2,...,ca_{n_c}), of controlled attribute values.

252 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.

```
252.  in_chs: ui:UI × attrs:ATTRS → RSL−Text
252.  in_chs(ui,attrs) ≡ “in { xch[ui,$xa_i$] | $xa_i$:ANm • $xa_i$∈$\mathscr{E}_\mathscr{A}$(attrs) }”
```

253 The in_out_chs function applies to a pair, a unique identifier and a mereology, and yields some **RSL**-Text, in the form of **in**put/**out**put channel declarations, one for each unique identifier in the mereology.

```
253.  in_out_chs: ui:UI × me:ME → RSL−Text
253.  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
    𝓕(ui,me,attrs)(ca) ≡
[1]       [] {let val=ch[{ui,ui′}]? in
[2]            (ch[{ui,ui′}]!in_reply(val,(ui,me,attrs))(ca)) ;
[3]             in_update(val,(ui,me,attrs))(ca) end
[4]          | ui′:UI • ui′ ∈ me}
[5]    ⌈⌉ [] {let val=await_reply(ui′,me,attrs)(ca) in
[6]            ch[{ui,ui′}]!val ;
[7]            out_update(val,(ui,me,attrs))(ca) end
[8]          | ui′:UI • ui′ ∈ me}
[9]    ⌈⌉     (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

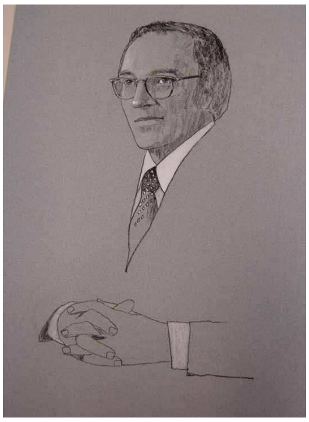
The notion of parts is far more general than that of Chapter 1. We have been able to treat Stansław Leśniewski’s notion of mereology sôlely based on parts, that is, their semantic values, without introducing the notion of the syntax of parts. Our compilation functions are (thus) far more general than defined in Chapter 1.

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 [209] Douglas T. Ross and John E. Ward report on the 1958–1967 MIT project for *computer-aided* *design (CAD) for numerically controlled production.*[12] Pages 13–17 of [209] reflects on issues bordering to and behind the concerns of mereology. Ross' thinking is clearly seen in the following text:



Douglas T. Ross 1927–2007.
Courtesy MIT Museum

> *"... our consideration of fundamentals begins not with design or problem-solving or programming or even mathematics, but with philosophy (in the old-fashioned meaning of the word) – we begin by establishing a "world-view". We have repeatedly emphasized that there is no way to bound or delimit the potential areas of application of our system, and that we must be prepared to cope with any conceivable problem. Whether the system will assist in any way in the solution of a given problem is quite another matter, ..., but in order to have a firm and uniform foundation, we must have a uniform philosophical basis upon which to approach any given problem. This "world-view" must provide a working framework and methodology in terms of which any aspect of our awareness of the world may be viewed. It must be capable of expressing the utmost in reality, giving expression to unending layers of ever-finer and more concrete detail, but at the same time abstract chimerical*[13] *visions bordering on unreality must fall within the same scheme. "Above all, the world-view itself must be concrete and workable, for it will form the basis for all involvement of the computer in the problem-solving process, as well as establishing a viewpoint for approaching the unknown human component of the problem-solving team."*

Yes, indeed, the philosophical disciplines of ontology, epistemology and mereology, amongst others, ought be standard curricula items in the computer science and software engineering studies, or better: domain engineers cum software system designers ought be imbued by the wisdom of those disciplines as was Doug.

> *"... in the summer of 1960 we coined the word plex to serve as a generic term for these philosophical ruminations. "Plex" derives from the word plexus, "An interwoven combination of parts in a structure", (Webster). ... The purpose of a* **'modeling plex'** *is to represent completely and in its entirety a "thing", whether it is concrete or abstract, physical or conceptual. A 'modeling plex' is a trinity with three primary aspects, all of which must be present. If any one is missing a complete representation or modeling is impossible. The three aspects of plex are* **data, structure,** *and* **algorithm.** *... "* which *"... is concerned with the behavioral characteristics of the plex model – the interpretive rules for making meaningful the data and structural aspects of the plex, for assembling specific instances of the plex, and for interrelating the plex with other plexes and operators on plexes. Specification of the algorithmic aspect removes the ambiguity of meaning and interpretation of the data structure and provides a complete representation of the thing being modeled."*

In the terminology of the current chapter a plex is a part (whether composite or atomic), the data are the properties (of that part), the structure is the mereology (of that part) and the algorithm is the process (for that part). Thus Ross was, perhaps, a first instigator (around 1960) of object-orientedness. A first, "top of the iceberg" account of the mereology-ideas that Doug had then can be found in the much later (1976) three page note [208]. Doug not only 'invented' CAD but was also the father of AED (Algol Extended for Design), the Automatically Programmed Tool (APT) language, SADT (Structured Analysis and Design

[12] Doug is said to have coined the term and the abbreviation CAD [207].

[13] Chimerical: existing only as the product of unchecked imagination: fantastically visionary or improbable

Technique) and helped develop SADT into the IDEF0[14] method for the Air Force's Integrated Computer-Aided Manufacturing (ICAM) program's IDEF suite of analysis and design methods. Douglas T. Ross went on for many years thereafter, to deepen and expand his ideas of relations between mereology and the programming language concept of type at the IFIP WG2.3 working group meetings. He did so in the, to some, enigmatic, but always fascinating style you find on Page 63 of [208].

In [155] **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).

[96] **R. Casati** and **A. Varzi:** *Parts and Places: the structures of spatial representation* has been the major source for this chapter's understanding of mereology. Our motivation was not the spatial or topological mereology, [217]. The present chapter does not utilize any of these concepts' axiomatisation in [96, 217]. Still it is best to say that this chapter 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* [122].

> *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`).

[14] IDEF0: Icam DEFinition for Function Modeling: https://en.wikipedia.org/wiki/IDEF0

Part **II**

A Requirements Engineering Method

5

From Domain Descriptions to Requirements Prescriptions

Chapter 1 introduces 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 Introduction

We survey preliminary issues.

5.1.1 The Triptych Dogma of Software Development

We see software development progressing as follows: *Before one can design software one must have a firm grasp of the requirements. Before one can prescribe requirements one must have a reasonably firm grasp of the domain.* Software engineering, to us, therefore include these three phases: *domain engineering*, *requirements engineering* and *software design*.

5.1.2 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".

5.1.3 The Contribution of Chapter

We claim that the present chapter 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 [67], 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.

5.1.4 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.

5.1.5 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.[2] 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 [67] and formalised in [58]. That

[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 new contributions — so are left out.

[2] The example of this section is that of the "running example" of Chapter 1.

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 (254a.) with a road net, (254b.) with a fleet of vehicles and (254c.) of whose individual position on the road net we can speak, that is, monitor.[3]

254 We analyse the traffic system into
  a a composite road net,
  b a composite fleet (of vehicles), and
  c an atomic monitor.

type
254 Δ
254a N
254b F
254c M
value
254a **obs_part**_N: $\Delta \rightarrow$ N
254b **obs_part**_F: $\Delta \rightarrow$ F
254c **obs_part**_M: $\Delta \rightarrow$ M

Applying `observe_endurant_sorts`, the domain description prompt 1 on Page 19, to a net, n:N, yields the following.

255 The road net consists of two composite parts,
  a an aggregation of hubs and
  b an aggregation of links.

type
255a HA
255b LA
value
255a **obs_part**_HA: N $\rightarrow$ HA
255b **obs_part**_LA: N $\rightarrow$ LA

Hubs and Links

Applying `observe_part_type`, the domain description prompt 2 on Page 21, to hub and link aggregates yields the following.

256 Hub aggregates are sets of hubs.
257 Link aggregates are sets of links.
258 Fleets are set of vehicles.

[3] The monitor can be thought of, i.e., conceptualised. It is not necessarily a physically manifest phenomenon.

```
type
256  H, HS = H-set
257  L, LS = L-set
258  V, VS = V-set
value
256  obs_part_HS: HA → HS
257  obs_part_LS: LA → LS
258  obs_part_VS: F → VS
```

259 We introduce some auxiliary functions.
   a links extracts the links of a network.
   b hubs extracts the hubs of a network.

```
value
259a  links: Δ → L-set
259a  links(δ) ≡ obs_part_LS(obs_part_LA(obs_part_N(δ)))
259b  hubs: Δ → H-set
259b  hubs(δ) ≡ obs_part_HS(obs_part_HA(obs_part_N(δ)))
```

Unique Identifiers

Applying `observe_unique_identifier`, the domain description prompt 4 on Page 24, to the observed parts yields the following.

260 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
260a  NI, HAI, LAI, HI, LI, FI, VI, MI
value
260c  uid_NI:  N → NI
260c  uid_HAI: HA → HAI
260c  uid_LAI: LA → LAI
260c  uid_HI:  H → HI
260c  uid_LI:  L → LI
260c  uid_FI: F → FI
260c  uid_VI: V → VI
260c  uid_MI: M → MI
axiom
260b  NI∩HAI=∅, NI∩LAI=∅, NI∩HI=∅, etc.
```

where axiom 260b. is expressed semi-formally, in mathematics. We introduce some auxiliary functions:

261 xtr_lis extracts all link identifiers of a traffic system.
262 xtr_his extracts all hub identifiers of a traffic system.
263 Given an appropriate link identifier and a net get_link ‘retrieves’ the designated link.
264 Given an appropriate hub identifier and a net get_hub ‘retrieves’ the designated hub.

```
value
261  xtr_lis: Δ → LI-set
261  xtr_lis(δ) ≡
261     let ls = links(δ) in {uid_LI(l)|l:L•l ∈ ls} end
262  xtr_his: Δ → HI-set
262  xtr_his(δ) ≡
262     let hs = hubs(δ) in {uid_HI(h)|h:H•k ∈ hs} end
263  get_link: LI → Δ →̃ L
263  get_link(li)(δ) ≡
263      let ls = links(δ) in
263      let l:L • l ∈ ls ∧ li=uid_LI(l) in l end end
263      pre: li ∈ xtr_lis(δ)
264  get_hub: HI → Δ →̃ H
264  get_hub(hi)(δ) ≡
264      let hs = hubs(δ) in
264      let h:H • h ∈ hs ∧ hi=uid_HI(h) in h end end
264      pre: hi ∈ 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 `observe_mereology`, the domain description prompt 5 on Page 26, to hubs, links, vehicles and the monitor yields the following.

265 Hub mereologies reflect that they are connected to zero, one or more links.
266 Link mereologies reflect that they are connected to exactly two distinct hubs.
267 Vehicle mereologies reflect that they are connected to the monitor.
268 The monitor mereology reflects that it is connected to all vehicles.
269 For all hubs of any net it must be the case that their mereology designates links of that net.
270 For all links of any net it must be the case that their mereologies designates hubs of that net.
271 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
265  obs_mereo_H: H → LI-set
266  obs_mereo_L: L → HI-set
axiom
266  ∀ l:L•card obs_mereo_L(l)=2
value
267  obs_mereo_V: V → MI
268  obs_mereo_M: M → VI-set
axiom
269  ∀ δ:Δ, hs:HS•hs=hubs(δ), ls:LS•ls=links(δ) •
269      ∀ h:H•h ∈ hs•obs_mereo_H(h)⊆xtr_lis(δ) ∧
270      ∀ l:L•l ∈ ls•obs_mereo_L(l)⊆xtr_his(δ) ∧
271a     let f:F•f=obs_part_F(δ) ⇒
271a         let m:M•m=obs_part_M(δ),
271a             vs:VS•vs=obs_part_VS(f) in
271a         ∀ v:V•v ∈ vs⇒uid_V(v) ∈ obs_mereo_M(m)
271b       ∧ obs_mereo_M(m) = {uid_V(v)|v:V•v ∈ vs}
```

```
271b    end end
```

Attributes, I

We may not have shown all of the attributes mentioned below — so consider them informally introduced !

- **Hubs:** *location*s[4] 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 `observe_attributes`, the domain description prompt 6 on Page 28, to hubs, links, vehicles and the monitor yields the following.

First hubs.

272 Hubs
  a have geodetic locations, GeoH,
  b have *hub states* which are sets of pairs of identifiers of links connected to the hub[5],
  c and have *hub state spaces* which are sets of hub states[6].
273 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.
274 We introduce an auxiliary function: xtr_lis extracts all link identifiers of a hub state.

```
type
272a    GeoH
272b    H$\Sigma$ = (LI×LI)-set
272c    H$\Omega$ = H$\Sigma$-set
value
272a    attr_GeoH: H → GeoH
272b    attr_H$\Sigma$: H → H$\Sigma$
272c    attr_H$\Omega$: H → H$\Omega$
axiom
273      $\forall$ $\delta$:$\Delta$ • let hs = hubs($\delta$) in
273        $\forall$ h:H • h $\in$ hs •
273a             xtr_lis(h)$\subseteq$xtr_lis($\delta$)
273b           $\wedge$ attr_$\Sigma$(h) $\in$ attr_$\Omega$(h)
273        end
value
274   xtr_lis: H → LI-set
274   xtr_lis(h) $\equiv$ {li | li:LI,(li′,li″):LI×LI • (li′,li″) $\in$ attr_H$\Sigma$(h) $\wedge$ li $\in$ {li′,li″}}
```

Then links.

275 Links have lengths.

[4] By location we mean a geodetic position.
[5] A hub state "signals" which input-to-output link connections are open for traffic.
[6] A hub state space indicates which hub states a hub may attain over time.

276 Links have geodetic location.
277 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
275    LEN
276    GeoL
277a   LΣ = (HI×HI)-set
277b   LΩ = LΣ-set
value
275    attr_LEN: L → LEN
276    attr_GeoL: L → GeoL
277a   attr_LΣ: L → LΣ
277b   attr_LΩ: L → LΩ
axiom
277   ∀ n:N • let ls = xtr−links(n), hs = xtr_hubs(n) in
277     ∀ l:L•l ∈ ls ⇒
277a        let lσ = attr_LΣ(l) in
277a        0≤card lσ≤4
277a      ∧ ∀ (hi′,hi″):(HI×HI)•(hi′,hi″) ∈ lσ ⇒ {hi′,hi″}=obs_mereo_L(l)
277b      ∧ attr_LΣ(l) ∈ attr_LΩ(l)
277      end end
```

Then vehicles.

278 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
278     VPos = onL | atH
278a    onL :: LI HI HI R
278b    R = Real        axiom ∀ r:R • 0≤r≤1
278c    atH :: HI LI LI
value
278     attr_VPos: V → VPos
axiom
278a    ∀ n:N, onL(li,fhi,thi,r):VPos •
278a        ∃ l:L•l ∈obs_part_LS(obs_part_N(n)) ⇒ li=uid_L(l)∧{fhi,thi}=obs_mereo_L(l),
278c    ∀ n:N, atH(hi,fli,tli):VPos •
278c        ∃ h:H•h ∈obs_part_HS(obs_part_N(n)) ⇒ hi=uid_H(h)∧(fli,tli) ∈ attr_LΣ(h)
```

279 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 !

280 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
279b   MAP = VI →m VPos
axiom
279b   ∀ map:MAP • card dom map = card rng map
value
279    distribute: VS → N → MAP
279    distribute(vs)(n) ≡
279a      let (hs,ls) = (xtr_hubs(n),xtr_links(n)) in
279a      let vps = {onL(uid_(l),fhi,thi,r) | l:L•l ∈ls∧{fhi,thi} ⊆obs_mereo_L(l)∧0≤r≤1}
279a                ∪ {atH(uid_H(h),fli,tli) | h:H•h ∈hs∧{fli,tli} ⊆obs_mereo_H(h)} in
279b      [uid_V(v)↦vp|v:V,vp:VPos•v ∈vs∧vp∈vps] end
279       end


280a   xtr_links: N → L-set
280a   xtr_links(n) ≡ obs_part_LS(obs_part_LA(n))
280b   xtr_hubs: N → H-set
280a   xtr_hubs(n) ≡ obs_part_H(obs_part_HA_Δ(n))
```

And finally monitors. We consider only one monitor attribute.

281 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
281   Traffic = VI →m (T × VPos)*
value
281   attr_Traffic: M → Traffic
axiom
281b  ∀ δ:Δ •
281b     let m = obs_part_M(δ) in
281b     let tf = attr_Traffic(m) in
281b     dom tf ⊆ xtr_vis(δ) ∧
281b     ∀ vi:VI • vi ∈ dom tf •
281b        let tr = tf(vi) in
281b        ∀ i,i+1:Nat • {i,i+1}⊆dom tr •
281b           let (t,vp)=tr(i),(t′,vp′)=tr(i+1) in
281b           t<t′
```

```
281(b)i              ∧ case (vp,vp′) of
281(b)i                  (onL(li,fhi,thi,r),onL(li′,fhi′,thi′,r′))
281(b)i                    → li=li′∧fhi=fhi′∧thi=thi′∧r≤r′ ∧ li ∈ xtr_lis(δ) ∧ {fhi,thi} = obs_mereo_L(get_link(li)(δ)),
281(b)ii                 (atH(hi,fli,tli),atH(hi′,fli′,tli′))
281(b)ii                   → hi=hi′∧fli=fli′∧tli=tli′ ∧ hi ∈ xtr_his(δ) ∧ (fli,tli) ∈ obs_mereo_H(get_hub(hi)(δ)),
281(b)iii                (onL(li,fhi,thi,1),atH(hi,fli,tli))
281(b)iii                  → li=fli∧thi=hi ∧ {li,tli} ⊆ xtr_lis(δ) ∧ {fhi,thi}=obs_mereo_L(get_link(li)(δ))
281(b)iii                   ∧ hi ∈ xtr_his(δ) ∧ (fli,tli) ∈ obs_mereo_H(get_hub(hi)(δ)),
281(b)iv                 (atH(hi,fli,tli),onL(li′,fhi′,thi′,0))
281(b)iv                   →  etcetera,
281b                 _ → false
281b         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 [67]): 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

282 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.
283 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
282   insert_hub: H → N → N
283   insert_hub(h)(n) as n′
282a      pre: uid_H(h) ∉ xtr_his(n)
282b        ∧  obs_mereo_H= {}
282c        ∧  ...
283a      post: obs_part_Ls(n) = obs_part_Ls(n′)
283b            ∧ obs_part_Hs(n) ∪ {h} = obs_part_Hs(n′)
```

Link Disappearance Event

We formalise aspects of the link disappearance event:

284 The result net, n′:N′, is not well-formed.
285 For a link to disappear there must be at least one link in the net;
286 and such a link may disappear such that
287 it together with the resulting net makes up for the “original” net.

```
value
284  link_diss_event: N × N′ × Bool
284  link_diss_event(n,n′) as tf
285     pre:  obs_part_Ls(obs_part_LS(n))≠{}
286     post: ∃ l:L•l ∈ obs_part_Ls(obs_part_LS(n)) ⇒
287              l ∉ obs_part_Ls(obs_part_LS(n′))
287              ∧ n′ ∪ {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.

288 besides the domain, δ:Δ,
289 a net, n:N,
290 a set of vehicles, vs:V-**set**,
291 a monitor, m:M, and
292 a clock, clock, behaviour.
293 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
288  δ:Δ
289  n:N = obs_part_N(δ),
289  ls:L-set=links(δ),hs:H-set=hubs(δ),
289  lis:LI-set=xtr_lis(δ),his:HI-set=xtr_his(δ)
290  va:VS=obs_part_VS(obs_part_F(δ)),
290  vs:Vs-set=obs_part_Vs(va),
290  vis:VI-set = {uid_VI(v)|v:V•v ∈ vs},
291  m:obs_part_M(δ),
291  mi=uid_MI(m),
291  ma:attributes(m)
292  clock: 𝕋 → out {clk_ch[vi|vi:VI•vi ∈ vis]}  Unit
293  vm:MAP•vpos_map = distribute(vs)(n);
```

Channels:

294 We additionally declare a set of vehicle-to-monitor-channels indexed
  a by the unique identifiers of vehicles
  b and the (single) monitor identifier.[7]
and communicating vehicle positions.

[7] Technically speaking: we could omit the monitor identifier.

channel
294 {v_m_ch[vi,mi]|vi:VI•vi ∈ vis}:VPos

Behaviour Signatures:

295 The road traffic system behaviour, rts, takes no arguments (hence the first **Unit**)[8]; and "behaves", that is, continues forever (hence the last **Unit**).
296 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".
297 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
295 rts: **Unit** → **Unit**
296 $\mathsf{veh}_{vi:VI}$: mi:MI → vp:VPos → **out** vm_ch[vi,mi] **Unit**
297 $\mathsf{mon}_{mi:MI}$: vis:VI-**set** → RTF → **in** {v_m_ch[vi,mi]|vi:VI•vi ∈ vis},clk_ch **Unit**

The Road Traffic System Behaviour:

298 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
298 rts() =
298a ‖ {$\mathsf{veh}_{\mathbf{uid}\_VI(v)}$(mi)(vm(**uid**_VI(v)))|v:V•v ∈ vs}
298b ‖ 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.

299 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,

[8] The **Unit** designator is an RSL technicality.

  - 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" !

```
299      veh$_{vi}$(mi)(vp:atH(hi,fli,tli)) ≡
299a           v_m_ch[vi,mi]!vp ; veh$_{vi}$(mi)(vp)
299b         ⌈⌉
299(b)i        let {hi′,thi}=obs_mereo_L(get_link(tli)(n)) in
299(b)i                                 assert: hi′=hi
299(b)ii       v_m_ch[vi,mi]!onL(tli,hi,thi,0) ;
299(b)ii       veh$_{vi}$(mi)(onL(tli,hi,thi,0)) end
299c         ⌈⌉ stop
```

300 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" !

```
300        veh$_{vi}$(mi)(vp:onL(li,fhi,thi,r)) ≡
300a         v_m_ch[vi,mi]!vp ; veh$_{vi}$(mi,va)(vp)
300b       ⌈⌉ if r + $\ell_\varepsilon$≤1
300(b)i        then
300(b)i          v_m_ch[vi,mi]!onL(li,fhi,thi,r+$\ell_\varepsilon$) ;
300(b)i          veh$_{vi}$(mi)(onL(li,fhi,thi,r+$\ell_\varepsilon$))
300(b)ii       else
300(b)ii1          let li′:LI•li′ ∈ obs_mereo_H(get_hub(thi)(n)) in
300(b)ii2           v_m_ch[vi,mi]!atH(li,thi,li′);
300(b)ii2           veh$_{vi}$(mi)(atH(li,thi,li′)) end end
300c       ⌈⌉ stop
```

The Monitor Behaviour

301 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".

302 Either the monitor “does own work”
303 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.

```
301   mon_mi(vis)(trf) ≡
302        mon_mi(vis)(trf)
303     ⊓
303a      □{let tvp = (clk_ch?,v_m_ch[vi,mi]?) in
303b         let trf′ = trf † [vi ↦ trf(vi)⌢<tvp>] in
303c         mon_mi(vis)(trf′)
303d         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* [43, 2008] of (i) *intrinsic*. It does not reflect the other *domain facets*: (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 intrinsic*. 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 25 **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 [127, 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 382–385), (iii) of that part of the toll-gate software which interfaces the toll-gate and the road pricing calculator (cf. Items 390–392) and (i) of the road pricing calculator (cf. Items 421–434) ■

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 381–385, (ii) that the GNSS regularly offers vehicles correct information as to their global position (cf. Item 382), (iii) that *toll-gates* behave as prescribed in Items 387–392, 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 386–387 ■

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 381–385, and (ii) that the road pricing calculator behave as per Items 421–434 ■

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 [108, 145, 240, 154, 231] 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 26 **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 27 **Domain Projection:** By a **domain projection** is meant *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[9]: 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[10].

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

254 $\Delta_{\mathscr{P}}$

254a $\mathsf{N}_{\mathscr{P}}$

254b $\mathsf{F}_{\mathscr{P}}$

value

254a **obs_part**_$\mathsf{N}_{\mathscr{P}}$: $\Delta_{\mathscr{P}} \rightarrow \mathsf{N}_{\mathscr{P}}$

254b **obs_part**_$\mathsf{F}_{\mathscr{P}}$: $\Delta_{\mathscr{P}} \rightarrow \mathsf{F}_{\mathscr{P}}$

type

255a $\mathsf{HA}_{\mathscr{P}}$

255b $\mathsf{LA}_{\mathscr{P}}$

value

255a **obs_part**_HA: $\mathsf{N}_{\mathscr{P}} \rightarrow \mathsf{HA}$

255b **obs_part**_LA: $\mathsf{N}_{\mathscr{P}} \rightarrow \mathsf{LA}$

Concrete Types

type

256 $\mathsf{H}_{\mathscr{P}}$, $\mathsf{HS}_{\mathscr{P}} = \mathsf{H}_{\mathscr{P}}$**-set**

257 $\mathsf{L}_{\mathscr{P}}$, $\mathsf{LS}_{\mathscr{P}} = \mathsf{L}_{\mathscr{P}}$**-set**

258 $\mathsf{V}_{\mathscr{P}}$, $\mathsf{VS}_{\mathscr{P}} = \mathsf{V}_{\mathscr{P}}$**-set**

value

256 **obs_part**_$\mathsf{HS}_{\mathscr{P}}$: $\mathsf{HA}_{\mathscr{P}} \rightarrow \mathsf{HS}_{\mathscr{P}}$

257 **obs_part**_$\mathsf{LS}_{\mathscr{P}}$: $\mathsf{LA}_{\mathscr{P}} \rightarrow \mathsf{LS}_{\mathscr{P}}$

258 **obs_part**_$\mathsf{VS}_{\mathscr{P}}$: $\mathsf{F}_{\mathscr{P}} \rightarrow \mathsf{VS}_{\mathscr{P}}$

259a links: $\Delta_{\mathscr{P}} \rightarrow$ L**-set**

259a links($\delta_{\mathscr{P}}$) $\equiv$ **obs_part**_$\mathsf{LS}_{\mathscr{R}}$(**obs_part**_$\mathsf{LA}_{\mathscr{R}}$($\delta_{\mathscr{R}}$))

259b hubs: $\Delta_{\mathscr{P}} \rightarrow$ H**-set**

259b hubs($\delta_{\mathscr{P}}$) $\equiv$ **obs_part**_$\mathsf{HS}_{\mathscr{P}}$(**obs_part**_$\mathsf{HA}_{\mathscr{P}}$($\delta_{\mathscr{P}}$))

[9] By ‘relate to … these’ we mean that the required system does not rely on domain phenomena that have been “projected away”.

[10] Restrictions of the net to the toll road nets, hinted at earlier, will follow in the next domain requirements steps.

Unique Identifiers

```
type
260a  HI, LI, VI, MI
value
260c  uid_HI: $H_{\mathscr{P}} \to$ HI
260c  uid_LI: $L_{\mathscr{P}} \to$ LI
260c  uid_VI: $V_{\mathscr{P}} \to$ VI
260c  uid_MI: $M_{\mathscr{P}} \to$ MI
axiom
260b  $HI \bigcap LI = \emptyset$, $HI \bigcap VI = \emptyset$, $HI \bigcap MI = \emptyset$,
260b  $LI \bigcap VI = \emptyset$, $LI \bigcap MI = \emptyset$, $VI \bigcap MI = \emptyset$
```

Mereology

```
value
265  obs_mereo_$H_{\mathscr{P}}$: $H_{\mathscr{P}} \to$ LI-set
266  obs_mereo_$L_{\mathscr{P}}$: $L_{\mathscr{P}} \to$ HI-set
266       axiom $\forall$ l:$L_{\mathscr{P}}$ • card obs_mereo_$L_{\mathscr{P}}$(l)=2
267  obs_mereo_$V_{\mathscr{P}}$: $V_{\mathscr{P}} \to$ MI
268  obs_mereo_$M_{\mathscr{P}}$: $M_{\mathscr{P}} \to$ VI-set
axiom
269  $\forall$ $\delta_{\mathscr{P}}$:$\Delta_{\mathscr{P}}$, hs:HS•hs=hubs($\delta$), ls:LS•ls=links($\delta_{\mathscr{P}}$) $\Rightarrow$
269     $\forall$ h:$H_{\mathscr{P}}$•h $\in$ hs $\Rightarrow$ obs_mereo_$H_{\mathscr{P}}$(h)$\subseteq$xtr_his($\delta_{\mathscr{P}}$) $\wedge$
270     $\forall$ l:$L_{\mathscr{P}}$•l $\in$ ls • obs_mereo_$L_{\mathscr{P}}$(l)$\subseteq$xtr_lis($\delta_{\mathscr{P}}$) $\wedge$
271a    let f:$F_{\mathscr{P}}$•f=obs_part_$F_{\mathscr{P}}$($\delta_{\mathscr{P}}$) $\Rightarrow$ vs:$VS_{\mathscr{P}}$•vs=obs_part_$VS_{\mathscr{P}}$(f) in
271a       $\forall$ v:$V_{\mathscr{P}}$•v $\in$ vs $\Rightarrow$ uid_$V_{\mathscr{P}}$(v) $\in$ obs_mereo_$M_{\mathscr{P}}$(m)
271b       $\wedge$ obs_mereo_$M_{\mathscr{P}}$(m) = {uid_$V_{\mathscr{P}}$(v)|v:V•v $\in$ vs}
271b    end
```

Attributes: We project attributes of hubs, links and vehicles. First **hubs:**

```
type
272a  GeoH
272b  $H\Sigma_{\mathscr{P}}$ = (LI×LI)-sett
272c  $H\Omega_{\mathscr{P}}$ = $H\Sigma_{\mathscr{P}}$-set
value
272b  attr_$H\Sigma_{\mathscr{P}}$: $H_{\mathscr{P}} \to H\Sigma_{\mathscr{P}}$
272c  attr_$H\Omega_{\mathscr{P}}$: $H_{\mathscr{P}} \to H\Omega_{\mathscr{P}}$
axiom
273   $\forall$ $\delta_{\mathscr{P}}$:$\Delta_{\mathscr{P}}$,
273     let hs = hubs($\delta_{\mathscr{P}}$) in
273     $\forall$ h:$H_{\mathscr{P}}$ • h $\in$ hs •
273a        xtr_lis(h)$\subseteq$xtr_lis($\delta_{\mathscr{P}}$)
273b       $\wedge$ attr_$\Sigma_{\mathscr{P}}$(h) $\in$ attr_$\Omega_{\mathscr{P}}$(h)
273     end
```

Then **links:**

```
type
276   GeoL
277a  $L\Sigma_{\mathscr{P}}$ = (HI×HI)-set
277b  $L\Omega_{\mathscr{P}}$ = $L\Sigma_{\mathscr{P}}$-set
value
```

```
276   attr_GeoL: L → GeoL
277a  attr_$L\Sigma_{\mathscr{P}}$: $L_{\mathscr{P}}$ → $L\Sigma_{\mathscr{P}}$
277b  attr_$L\Omega_{\mathscr{P}}$: $L_{\mathscr{P}}$ → $L\Omega_{\mathscr{P}}$
axiom
      277a– 277b on Page 159.
```

Finally **vehicles:** For 'road pricing' we need vehicle positions. But, for "technical reasons", we must abstain from the detailed description given in Items 278–278c[11] We therefore *simplify* vehicle positions.

304 A simplified vehicle position designates
  a either a link
  b or a hub,

```
type
304   SVPos = SonL | SatH
304a  SonL :: LI
304b  SatH :: HI
axiom
278a'   ∀ n:N, SonL(li):SVPos • ∃ l:L•l ∈obs_part_LS(obs_part_N(n)) ⇒ li=uid_L(l)
278c'   ∀ n:N, SatH(hi):SVPos • ∃ h:H•h ∈obs_part_HS(obs_part_N(n)) ⇒ hi=uid_H(h)
```

Global Values

```
value
288   $\delta_{\mathscr{P}}$:$\Delta_{\mathscr{P}}$,
289   n:$N_{\mathscr{P}}$ = obs_part_$N_{\mathscr{P}}$($\delta_{\mathscr{P}}$),
289   ls:$L_{\mathscr{P}}$-set = links($\delta_{\mathscr{P}}$),
289   hs:$H_{\mathscr{P}}$-set = hubs($\delta_{\mathscr{P}}$),
289   lis:LI-set = xtr_lis($\delta_{\mathscr{P}}$),
289   his:HI-set = xtr_his($\delta_{\mathscr{P}}$)
```

Behaviour Signatures: We omit the monitor behaviour.

305 We leave the vehicle behaviours' attribute argument undefined.

```
type
305   ATTR
value
295   trs$_{\mathscr{P}}$: Unit → Unit
296   veh$_{\mathscr{P}}$: VI×MI×ATTR → ... Unit
```

The System Behaviour: We omit the monitor behaviour.

```
value
298a  trs$_{\mathscr{P}}$()=‖{veh$_{\mathscr{P}}$(uid_VI(v),obs_mereo_V(v),_) | v:V$_{\mathscr{P}}$•v ∈ vs}
```

The Vehicle Behaviour: Given the simplification of vehicle positions we *simplify* the vehicle behaviour given in Items 299–300

```
299′      veh$_{vi}$(mi)(vp:SatH(hi)) ≡
299a′              v_m_ch[vi,mi]!SatH(hi) ; veh$_{vi}$(mi)(SatH(hi))
299(b)i′         ⌈⌉ let li:LI•li ∈ obs_mereo_H(get_hub(hi)(n)) in
299(b)ii′              v_m_ch[vi,mi]!SonL(li) ; veh$_{vi}$(mi)(SonL(li)) end
299c′          ⌈⌉ stop
```

[11] 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*).

```
300′       veh$_{vi}$(mi)(vp:SonL(li)) ≡
300a′              v_m_ch[vi,mi]!SonL(li) ; veh$_{vi}$(mi)(SonL(li))
300(b)ii1′    ⌈⌉ let hi:HI•hi ∈ obs_mereo_L(get_link(li)(n)) in
300(b)ii2′           v_m_ch[vi,mi]!SatH(hi) ; veh$_{vi}$(mi)(atH(hi)) end
300c′         ⌈⌉ stop
```

We can simplify Items 299′–300c′ further.

```
306   veh$_{vi}$(mi)(vp) ≡
307         v_m_ch[vi,mi]!vp ; veh$_{vi}$(mi)(vp)
308      ⌈⌉ case vp of
308            SatH(hi) →
309               let li:LI•li ∈ obs_mereo_H(get_hub(hi)(n)) in
310               v_m_ch[vi,mi]!SonL(li) ; veh$_{vi}$(mi)(SonL(li)) end,
308            SonL(li) →
311               let hi:HI•hi ∈ obs_mereo_L(get_link(li)(n)) in
312               v_m_ch[vi,mi]!SatH(hi) ; veh$_{vi}$(mi)(atH(hi)) end end
313       ⌈⌉ stop
```

306 This line coalesces Items 299′ and 300′.
307 Coalescing Items 299a′ and 300′.
308 Captures the distinct parameters of Items 299′ and 300′.
309 Item 299(b)i′.
310 Item 299(b)ii′.
311 Item 300(b)ii1′.
312 Item 300(b)ii2′.
313 Coalescing Items 299c′ and 300c′.

The above vehicle behaviour definition will be transformed (i.e., further "refined") in Sect. 5.5.1's Example 5.15; cf. Items 381– 385 on Page 184 ■

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 28 **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$'_{\mathscr{P}}$.

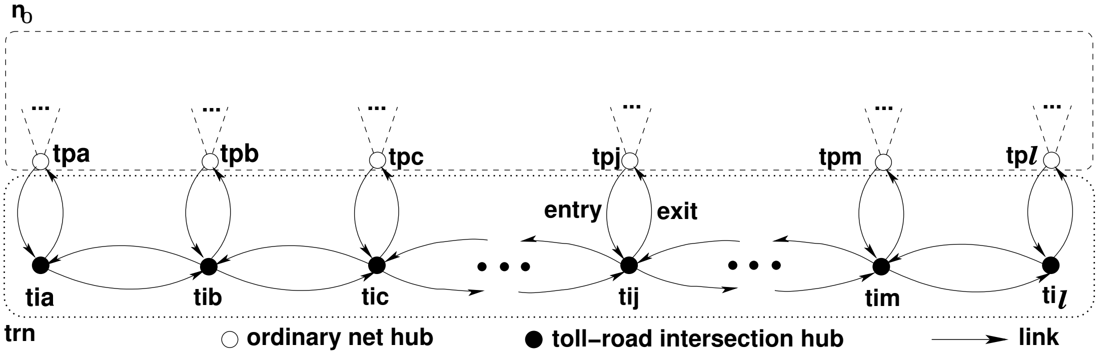


Fig. 5.1. A simple, linear toll-road net trn. tp_j: *t*oll *p*laza j, ti_j: *t*oll road *i*ntersection j.
Upper dashed sub-figure hint at an ordinary road net n$_o$.
Lower dotted sub-figure hint at a toll-road net trn.
Dash-dotted (- - -) "V"-images above tp_js hint at links to remaining "parts" of n$_o$.

314 The instantiated domain, $\delta_{\mathscr{I}}$:$\Delta_{\mathscr{I}}$ has just the net, n$_{\mathscr{I}}$:N$_{\mathscr{I}}$ being instantiated.
315 The road net consists of two "sub-nets"
  a an "ordinary" road net, n$_o$:N$_{\mathscr{P}'}$ and
  b a toll-road net proper, trn: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:TRN$_{\mathscr{I}}$[12], 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$_o$: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, $\{l_{i_1}, l_{i_2}, \cdots, l_{i_k}\}$.
 315b The toll-road net, trn: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 ("*u*p" and "*d*own") 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 ("*u*p" and "*d*own") links

[12] We (sometimes) omit the subscript $_{\mathscr{I}}$ when it should be clear from the context what we mean.

```
type
314    $\Delta_{\mathscr{T}}$
315    $\mathsf{N}_{\mathscr{T}} = \mathsf{N}_{\mathscr{P}'} \times \mathsf{HIL} \times \mathsf{TRN}$
315a   $\mathsf{N}_{\mathscr{P}'}$
315b   $\mathsf{TRN}_{\mathscr{T}} = (\mathsf{L}{\times}\mathsf{L})^{*}{\times}\mathsf{H}^{*}{\times}(\mathsf{L}{\times}\mathsf{L})^{*}$
315c   $\mathsf{HIL} = \mathsf{HI}^{*}$

axiom
315d   $\forall\ \mathsf{n}_{\mathscr{T}}{:}\mathsf{N}_{\mathscr{T}} \bullet$
315d       let $(\mathsf{n}_{\Delta},\mathsf{hil},(\mathsf{exll},\mathsf{hl},\mathsf{lll})) = \mathsf{n}_{\mathscr{T}}$ in
315d       len hil = len exll = len hl = len lll + 1
315d       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 $\mathsf{N}_{\mathscr{T}}$ requires some additional well-formedness conditions to be satisfied.

316 The toll-road intersection hubs all[13] have distinct identifiers.

```
316  wf_dist_toll_road_isect_hub_ids: $\mathsf{H}^{*}\to$Bool
316  wf_dist_toll_road_isect_hub_ids(hl) $\equiv$ len hl = card xtr_his(hl)
```

317 The toll-road links all have distinct identifiers.

```
317  wf_dist_toll_road_u_d_link_ids: $(\mathsf{L}{\times}\mathsf{L})^{*}\to$Bool
317  wf_dist_toll_road_u_d_link_ids(lll) $\equiv 2 \times$ len lll = card xtr_lis(lll)
```

318 The toll-road entry/exit links all have distinct identifiers.

```
318  wf_dist_e_x_link_ids: $(\mathsf{L}{\times}\mathsf{L})^{*}\to$Bool
318  wf_dist_e_x_link_ids(exll) $\equiv 2 \times$ len exll = card xtr_lis(exll)
```

319 Proper net links must not designate toll-road intersection hubs.

```
319  wf_isoltd_toll_road_isect_hubs: $\mathsf{HI}^{*}{\times}\mathsf{H}^{*}\to\mathsf{N}_{\mathscr{T}}\to$Bool
319  wf_isoltd_toll_road_isect_hubs(hil,hl)($\mathsf{n}_{\mathscr{T}}$) $\equiv$
319      let ls=xtr_links($\mathsf{n}_{\mathscr{T}}$) in
319      let his = $\cup$ {obs_mereo_L(l)|l:L•l $\in$ ls} in
319      his $\cap$ xtr_his(hl) = {} end end
```

320 The plaza hub identifiers must designate hubs of the 'ordinary' net.

```
320  wf_p_hubs_pt_of_ord_net: $\mathsf{HI}^{*}\to\mathsf{N}'_{\Delta}\to$Bool
320  wf_p_hubs_pt_of_ord_net(hil)($\mathsf{n}'_{\Delta}$) $\equiv$ elems hil $\subseteq$ xtr_his($\mathsf{n}'_{\Delta}$)
```

321 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.

[13] A 'must' can be inserted in front of all 'all's,

```
321  wf_p_hub_interf: N'_Δ→Bool
321  wf_p_hub_interf(n_o,hil,(exll,_,_)) ≡
321      ∀ i:Nat • i ∈ inds exll ⇒
321          let h = get_H(hil(i))(n'_Δ) in
321          let lis = obs_mereo_H(h) in
321          let lis' = lis \ xtr_lis(n') in
321          lis' = xtr_lis(exll(i)) end end end
```

322 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.

```
322   wf_toll_road_isect_hub_iface: N_𝒥→Bool
322   wf_toll_road_isect_hub_iface(_,_,(exll,hl,lll)) ≡
322       ∀ i:Nat • i ∈ inds hl ⇒
322           obs_mereo_H(hl(i)) =
322a           xtr_lis(exll(i)) ∪
322           case i of
322b                 1 → xtr_lis(lll(1)),
322b                 len hl → xtr_lis(lll(len hl−1))
322b                 _ → xtr_lis(lll(i)) ∪ xtr_lis(lll(i−1))
322           end
```

323 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.

```
323  wf_exll: (L×L)*×HI*×H*→Bool
323  wf_exll(exll,hil,hl) ≡
323      ∀ i:Nat • i ∈ len exll
323          let (hi,(el,xl),h) = (hil(i),exll(i),hl(i)) in
323          obs_mereo_L(el) = obs_mereo_L(xl)
323          = {hi} ∪ {uid_H(h)} end
323      pre: len eell = len hil = len hl
```

324 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.

```
324  wf_u_d_links: (L×L)*×H*→Bool
324  wf_u_d_links(lll,hl) ≡
324      ∀ i:Nat • i ∈ inds lll ⇒
324          let (ul,dl)  = lll(i) in
324          obs_mereo_L(ul) = obs_mereo_L(dl) =
324a        uid_H(hl(i)) ∪ uid_H(hl(i+1)) end
324      pre: len lll = len hl+1
```

We have used some additional auxiliary functions:

```
    xtr_his: $H^* \to$ HI-set
    xtr_his(hl) $\equiv$ {uid_HI(h)|h:H•h $\in$ elems hl}
    xtr_lis: (L×L)$\to$LI-set
    xtr_lis(l',l'') $\equiv$ {uid_LI(l')}$\cup${uid_LI(l'')}
    xtr_lis: $(L \times L)^* -$ LI-set
    xtr_lis(lll) $\equiv$
     $\cup${xtr_lis(l',l'')|(l',l''):(L×L)•(l',l'')$\in$ elems lll}
```

325 The well-formedness of instantiated nets is now the conjunction of the individual well-formedness predicates above.

```
325  wf_instantiated_net: $N_{\mathscr{I}} \to$ Bool
325  wf_instantiated_net($n'_\Delta$,hil,(exll,hl,lll))
316     wf_dist_toll_road_isect_hub_ids(hl)
317    $\wedge$ wf_dist_toll_road_u_d_link_ids(lll)
318    $\wedge$ wf_dist_e_e_link_ids(exll)
319    $\wedge$ wf_isolated_toll_road_isect_hubs(hil,hl)(n')
320    $\wedge$ wf_p_hubs_pt_of_ord_net(hil)(n')
321    $\wedge$ wf_p_hub_interf($n'_\Delta$,hil,(exll,_,_))
322    $\wedge$ wf_toll_road_isect_hub_iface(_,_,(exll,hl,lll))
323    $\wedge$ wf_exll(exll,hil,hl)
324    $\wedge$ 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
326    abstraction: $N_{\mathscr{I}} \to N_{\mathscr{P}}$
327    abstraction($n'_\Delta$,hil,(exll,hl,lll)) $\equiv$
328       let $n_{\mathscr{P}}$:$N_{\mathscr{P}}$ •
328           let hs = obs_part_HS$_{\mathscr{P}}$(obs_part_HA$_{\mathscr{P}}$($n'_{\mathscr{P}}$)),
328               ls = obs_part_LS$_{\mathscr{P}}$(obs_part_LA$_{\mathscr{P}}$($n'_{\mathscr{P}}$)),
328               ths = elems hl,
328               eells = xtr_links(eell), llls =  xtr_links(lll) in
329               hs$\cup$ths=obs_part_HS$_{\mathscr{P}}$(obs_part_HA$_{\mathscr{P}}$($n_{\mathscr{P}}$))
330          $\wedge$  ls$\cup$eells$\cup$llls=obs_part_LS$_{\mathscr{P}}$(obs_part_LA$_{\mathscr{P}}$($n_{\mathscr{P}}$))
331       $n_{\mathscr{P}}$ end end
```

326 The abstraction function takes a concrete net, $n_{\mathscr{I}}$:$N_{\mathscr{I}}$, and yields an abstract net, $n_{\mathscr{P}}$:$N_{\mathscr{P}}$.
327 The abstraction function doubly decomposes its argument into constituent lists and sub-lists.
328 There is postulated an abstract net, $n_{\mathscr{P}}$:$N_{\mathscr{P}}$, such that
329 the hubs of the concrete net and toll-road equals those of the abstract net, and
330 the links of the concrete net and toll-road equals those of the abstract net.
331 And that abstract net, $n_{\mathscr{P}}$:$N_{\mathscr{P}}$, is postulated to be an abstraction of the concrete net.

Discussion

Domain descriptions, such as illustrated in [67, *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 28 on Page 171, 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)[14].

5.4.3 Domain Determination

Definition 29 **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

332 *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 172,
  c such that each pair allows traffic in opposite directions.

```
332   opposite_traffics: (L×L)* × (L×L)* → Bool
332   opposite_traffics(exll,lll) ≡
332       ∀ (lt,lf):(L×L) • (lt,lf) ∈ elems exll⌢lll ⇒
332a           let (ltσ,lfσ) = (attr_LΣ(lt),attr_LΣ(lf)) in
332a′.          attr_LΩ(lt)={ltσ}∧attr_LΩ(ft)={ftσ}
332a″.      ∧ card ltσ = 1 = card lfσ
332         ∧ let ({(hi,hi′)},{(hi″,hi‴)}) = (ltσ,lfσ) in
332c           hi=hi‴ ∧ hi′=hi″
332          end end
```

Predicates 332a′. and 332a″. express the same property.

All Toll-road Hubs are Free-flow

333 *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 *ex*it *l*ink*s*,

[14] Here we disregard the fact that this toll-road does not start/end in neither *Bolzano Nord* nor *Trento Sud*.

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*.

```
333   free_flow_toll_road_hubs: (L×L)*×(L×L)*→Bool
333   free_flow_toll_road_hubs(exl,ll) ≡
333       ∀ i:Nat•i ∈ inds hl ⇒
333            attr_HΣ(hl(i)) =
333a                hσ_ex_ls(exl(i))
333b              ∪ hσ_et_ls(exl(i),(i,ll))
333c              ∪ hσ_tx_ls(exl(i),(i,ll))
333d              ∪ hσ_tt_ls(i,ll)
```

333a: from *e*ntry links back to the paired e*x*it *l*ink*s*:

```
333a  hσ_ex_ls: (L×L)→LΣ
333a  hσ_ex_ls(e,x) ≡ {(uid_LI(e),uid_LI(x))}
```

333b: from *e*ntry links to emanating *t*oll-road *l*ink*s*:

```
333b  hσ_et_ls: (L×L)×(Nat×(em:L×in:L)*)→LΣ
333b  hσ_et_ls((e,_),(i,ll)) ≡
333b      case i of
333b          2          → {(uid_LI(e),uid_LI(em(ll(1))))},
333b          len ll+1 → {(uid_LI(e),uid_LI(em(ll(len ll))))},
333b          _          → {(uid_LI(e),uid_LI(em(ll(i−1)))),
333b                           (uid_LI(e),uid_LI(em(ll(i))))}
333b      end
```

The *em* and *in* in the toll-road link list (em:L×in:L)* designate selectors for *em*anating, respectively *in*cident links. 333c: from incident *t*oll-road links to e*x*it *l*ink*s*:

```
333c  hσ_tx_ls: (L×L)×(Nat×(em:L×in:L)*)→LΣ
333c  hσ_tx_ls((_,x),(i,ll)) ≡
333c      case i of
333c          2          → {(uid_LI(in(ll(1))),uid_LI(x))},
333c          len ll+1 → {(uid_LI(in(ll(len ll))),uid_LI(x))},
333c          _          → {(uid_LI(in(ll(i−1))),uid_LI(x)),
333c                           (uid_LI(in(ll(i))),uid_LI(x))}
333c      end
```

333d: from incident *t*oll-road link to emanating *t*oll-road *l*ink*s*:

```
333d  hσ_tt_ls: Nat×(em:L×in:L)*→LΣ
333d  hσ_tt_ls(i,ll) ≡
333d      case i of
333d          2          → {(uid_LI(in(ll(1))),uid_LI(em(ll(1))))},
333d          len ll+1 → {(uid_LI(in(ll(len ll))),uid_LI(em(ll(len ll))))},
333d          _          → {(uid_LI(in(ll(i−1))),uid_LI(em(ll(i−1)))),
333d                           (uid_LI(in(ll(i))),uid_LI(em(ll(i))))}
333d      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 30 **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 31 **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:

334 There is a domain, $\delta_{\mathscr{E}}$:$\Delta_{\mathscr{E}}$, which contains
335 a fleet, f$_{\mathscr{E}}$:F$_{\mathscr{E}}$, that is,
336 a set, vs$_{\mathscr{E}}$:VS$_{\mathscr{E}}$, of
337 extended vehicles, v$_{\mathscr{E}}$:V$_{\mathscr{E}}$ — their extension amounting to
338 a dynamic reactive attribute, whose value, ti-gpos:TiGpos, at any time, reflects that vehicle's *time-stamped global position.*[15]
339 The vehicle's GNSS receiver calculates, *loc_pos*, its local position, lpos:LPos, based on these signals.
340 Vehicles access these *external attribute*s via the *external attribute* channel, attr_TiGPos_ch.

[15] 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 [113].

```
type
334     Δ_ℰ
335     F_ℰ
336     VS_ℰ = V_ℰ-set
337     V_ℰ
338     TiGPos = 𝕋 × GPos
339     GPos, LPos
value
334     δ_ℰ:Δ_ℰ
335     obs_part_F_ℰ: Δ_ℰ → F_ℰ
335     f = obs_part_F_ℰ(δ_ℰ)
336     obs_part_VS_ℰ: F_ℰ → VS_ℰ
336     vs = obs_part_VS_ℰ(f)
336     vis = xtr_vis(vs)
338     attr_TiGPos_ch[vi]?
339     loc_pos: GPos → LPos
channel
339     {attr_TiGPos_ch[vi]|vi:VI•vi ∈ vis}:TiGPos
```

We define two auxiliary functions,

341 xtr_vs, which given a domain, or a fleet, extracts its set of vehicles, and
342 xtr_vis which given a set of vehicles generates their unique identifiers.

```
value
341   xtr_vs: (Δ_ℰ|F_ℰ|VS_ℰ) → V_ℰ-set
341   xtr_vs(arg) ≡
341       is_Δ_ℰ(arg) → obs_part_VS_ℰ(obs_part_F_ℰ(arg)),
341       is_F_ℰ(arg) → obs_part_VS_ℰ(arg),
341       is_VS_ℰ(arg) → arg
342   xtr_vis: (Δ_ℰ|F_ℰ|VS_ℰ) → VI-set
342   xtr_vis(arg) ≡ {uid_VI(v)|v ∈ xtr_vs(arg)}
```

[b] Road Pricing Calculator: Basic Sort and Unique Identifier:

343 The domain $\delta_{\mathscr{E}}$:$\Delta_{\mathscr{E}}$, also contains a pricing calculator, c:$C_{\delta_{\mathscr{E}}}$, with unique identifier ci:CI.

```
type
343     C, CI
value
343     obs_part_C: Δ_ℰ → C
343     uid_CI: C → CI
343     c = obs_part_C(δ_ℰ)
343     ci = uid_CI(c)
```

[c] Vehicle to Road Pricing Calculator Channel:

344 Vehicles can, on their own volition, offer the timed local position, viti-lpos:VITiLPos
345 to the pricing calculator, c:$C_{\mathscr{E}}$ along a vehicles-to-calculator channel, v_c_ch.

```
type
344    VITiLPos = VI × (𝕋 × LPos)
channel
345    {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.

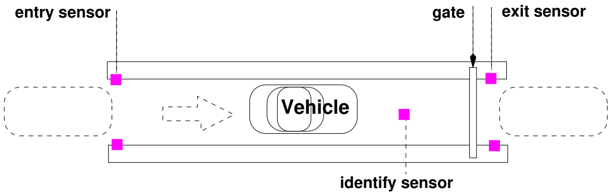


Fig. 5.2. A toll plaza gate

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 362- 363 on the next page).

346 The domain contains the extended net, $\mathsf{n}{:}\mathsf{N}_{\mathscr{E}}$,
347 with the net extension amounting to the toll-road net, $\mathsf{TRN}_{\mathscr{E}}$, that is, the instantiated toll-road net, $\mathsf{trn}{:}\mathsf{TRN}_{\mathscr{I}}$, is extended, into $\mathsf{trn}{:}\mathsf{TRN}_{\mathscr{E}}$, with *entry*, eg:EG, and *exit*, xg:XG, toll-gates.

From entry- and exit-gates we can observe

348 their unique identifier and
349 their mereology: pairs of entry-, respectively exit link and calculator unique identifiers; further
350 a pair of gate entry and exit sensors modeled as *external attribute* channels, (ges:ES,gls:XS), and
351 a time-stamped vehicle identity sensor modeled as *external attribute* channels.

type
346 $\mathsf{N}_{\mathscr{E}}$
347 $\mathsf{TRN}_{\mathscr{E}} = (\mathsf{EG}\times\mathsf{XG})^* \times \mathsf{TRN}_{\mathscr{I}}$
348 GI
value
346 **obs_part**_$\mathsf{N}_{\mathscr{E}}$: $\Delta_{\mathscr{E}} \rightarrow \mathsf{N}_{\mathscr{E}}$
347 **obs_part**_$\mathsf{TRN}_{\mathscr{E}}$: $\mathsf{N}_{\mathscr{E}} \rightarrow \mathsf{TRN}_{\mathscr{E}}$
348 **uid**_G: (EG|XG) → GI
349 **obs_mereo**_G: (EG|XG) → (LI×CI)
347 $\mathsf{trn}{:}\mathsf{TRN}_{\mathscr{E}} =$ **obs_part**_$\mathsf{TRN}_{\mathscr{E}}(\delta_{\mathscr{E}})$
channel
350 {attr_entry_ch[gi]|gi:GI•xtr_eGIds(trn)} ″`enter`″
350 {attr_exit_ch[gi]|gi:GI•xtr_xGIds(trn)} ″`exit`″
351 {attr_identity_ch[gi]|gi:GI•xtr_GIds(trn)} TIVI
type
351 $\mathsf{TIVI} = \mathbb{T} \times \mathsf{VI}$

We define some **auxiliary functions** over toll-road nets, $\mathsf{trn}{:}\mathsf{TRN}_{\mathscr{E}}$:

352 xtr_eGℓ extracts the ℓist of entry gates,
353 xtr_xGℓ extracts the ℓist of exit gates,
354 xtr_eGIds extracts the $s$et of entry gate identifiers,
355 xtr_xGIds extracts the $s$et of exit gate identifiers,
356 xtr_Gs extracts the $s$et of all gates, and
357 xtr_GIds extracts the $s$et of all gate identifiers.

```
value
352  xtr_eGℓ: $TRN_{\mathscr{E}} \to EG^*$
352  xtr_eGℓ(pgl,_) ≡ {eg|(eg,xg):(EG,XG)•(eg,xg)∈ elems pgl}
353  xtr_xGℓ: $TRN_{\mathscr{E}} \to XG^*$
353  xtr_xGℓ(pgl,_) ≡ {xg|(eg,xg):(EG,XG)•(eg,xg)∈ elems pgl}
354  xtr_eGIds: $TRN_{\mathscr{E}} \to$ GI-set
354  xtr_eGIds(pgl,_) ≡ {uid_GI(g)|g:EG•g ∈ xtr_eGs(pgl,_)}
355  xtr_xGIds: $TRN_{\mathscr{E}} \to$ GI-set
355  xtr_xGIds(pgl,_) ≡ {uid_GI(g)|g:EG•g ∈ xtr_xGs(pgl,_)}
356  xtr_Gs: $TRN_{\mathscr{E}} \to$ G-set
356  xtr_Gs(pgl,_) ≡ xtr_eGs(pgl,_) ∪ xtr_xGs(pgl,_)
357  xtr_GIds: $TRN_{\mathscr{E}} \to$ GI-set
357  xtr_GIds(pgl,_) ≡ xtr_eGIds(pgl,_) ∪ xtr_xGIds(pgl,_)
```

358 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

359 the unique identifier, ci:CI, of the road pricing calculator, c:C, and
360 the unique identifiers, vis:VI-**set**, of the fleet vehicles.

```
axiom
358   ∀ n:$N_{\mathscr{R}_3}$, trn:$TRN_{\mathscr{R}_3}$ •
358      let (exgl,(exl,hl,lll)) = obs_part_$TRN_{\mathscr{R}_3}$(n) in
358a      len exgl = len exl = len hl = len lll + 1
358b     ∧ card xtr_GIds(exgl) = 2 ∗ len exgl
358c     ∧ ∀ i:Nat•i ∈ inds exgl•
358c          let ((eg,xg),(el,xl)) = (exgl(i),exl(i)) in
358c          obs_mereo_G(eg) = (uid_U(el),ci,vis)
358c        ∧ obs_mereo_G(xg) = (uid_U(xl),ci,vis)
358      end end
```

[e] Toll-gate to Calculator Channels:

361 We distinguish between *entry* and *exit* gates.
362 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
363 to the road pricing calculator via a (gate to calculator) channel.

```
type
361  EE = "entry"|"exit"
362  EEVITiLI = EE×(VI×($\mathbb{T}$×SonL))
channel
363  {g_c_ch[gi,ci]|gi:GI•gi ∈ gis}:EETiVILI
```

[f] Road Pricing Calculator Attributes:

364 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

365 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,

366 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
364      TRM = VI $\xrightarrow[m]{}$ ($\mathbb{T}$ × SVPos)*
365      VPLF = GRM → LPos → (SVPos | "off_N")
366      GRM
value
364      attr_TRM: $C_{\mathscr{E}}$ → TRM
365      attr_VPLF: $C_{\mathscr{E}}$ → VPLF
```

The geodetic road map maps geodetic locations into hub and link identifiers.

276 Geodetic link locations represent the set of point locations of a link.

272a Geodetic hub locations represent the set of point locations of a hub.

367 A geodetic road map maps geodetic link locations into link identifiers and geodetic hub locations into hub identifiers.

368 We sketch the construction, *geo_GRM*, of geodetic road maps.

```
type
367  GRM = (GeoL $\xrightarrow[m]{}$ LI) ∪ (GeoH $\xrightarrow[m]{}$ HI)
value
368  geo_GRM: N → GRM
368  geo_GRM(n) ≡
368     let ls = xtr_links(n), hs = xtr_hubs(n) in
368     [attr_GeoL(l)↦uid_LI(l)|l:L•l ∈ ls]
368     ∪
368     [attr_GeoH(h)↦uid_HI(h)|h:H•h ∈ hs] end
```

369 The vplf:VPLF function obtains a simple vehicle position, svpos, from a geodetic road map, grm:GRM, and a local position , lpos:

```
value
369  obtain_SVPos: GRM → LPos → SVPos
369  obtain_SVPos(grm)(lpos) as svpos
369  post: case svpos of
369            SatH(hi)    → within(lpos,grm(hi)),
369            SonL(li)    → within(lpos,grm(li)),
369            "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:

370 There is a given domain, $\delta_{\mathscr{E}}:\Delta_{\mathscr{E}}$;
371 there is the net, $\mathsf{n}_{\mathscr{E}}:\mathsf{N}_{\mathscr{E}}$, of that domain;
372 there is toll-road net, $\mathsf{trn}_{\mathscr{E}}:\mathsf{TRN}_{\mathscr{E}}$, of that net;
373 there is a set, $\mathsf{egs}_{\mathscr{E}}:\mathsf{EG}_{\mathscr{E}}$**-set**, of entry gates;
374 there is a set, $\mathsf{xgs}_{\mathscr{E}}:\mathsf{XG}_{\mathscr{E}}$**-set**, of exit gates;
375 there is a set, $\mathsf{gis}_{\mathscr{E}}:\mathsf{GI}_{\mathscr{E}}$**-set**, ofgate identifiers;
376 there is a set, $\mathsf{vs}_{\mathscr{E}}:\mathsf{V}_{\mathscr{E}}$**-set**, of vehicles;
377 there is a set, $\mathsf{vis}_{\mathscr{E}}:\mathsf{VI}_{\mathscr{E}}$**-set**, of vehicle identifiers;
378 there is the road-pricing calculator, $\mathsf{c}_{\mathscr{E}}:\mathsf{C}_{\mathscr{E}}$ and
379 there is its unique identifier, $\mathsf{ci}_{\mathscr{E}}:\mathsf{CI}$.

value
370 $\delta_{\mathscr{E}}:\Delta_{\mathscr{E}}$
371 $\mathsf{n}_{\mathscr{E}}:\mathsf{N}_{\mathscr{E}} = \textbf{obs\_part\_}\mathsf{N}_{\mathscr{E}}(\delta_{\mathscr{E}})$
372 $\mathsf{trn}_{\mathscr{E}}:\mathsf{TRN}_{\mathscr{E}} = \textbf{obs\_part\_}\mathsf{TRN}_{\mathscr{E}}(\mathsf{n}_{\mathscr{E}})$
373 $\mathsf{egs}_{\mathscr{E}}:\mathsf{EG}\textbf{-set} = \mathsf{xtr\_egs}(\mathsf{trn}_{\mathscr{E}})$
374 $\mathsf{xgs}_{\mathscr{E}}:\mathsf{XG}\textbf{-set} = \mathsf{xtr\_xgs}(\mathsf{trn}_{\mathscr{E}})$
375 $\mathsf{gis}_{\mathscr{E}}:\mathsf{XG}\textbf{-set} = \mathsf{xtr\_gis}(\mathsf{trn}_{\mathscr{E}})$
376 $\mathsf{vs}_{\mathscr{E}}:\mathsf{V}_{\mathscr{E}}\textbf{-set} = \textbf{obs\_part\_}\mathsf{VS}(\textbf{obs\_part\_}\mathsf{F}_{\mathscr{E}}(\delta_{\mathscr{E}}))$
377 $\mathsf{vis}_{\mathscr{E}}:\mathsf{VI}\textbf{-set} = \{\textbf{uid\_}\mathsf{VI}(\mathsf{v}_{\mathscr{E}})|\mathsf{v}_{\mathscr{E}}:\mathsf{V}_{\mathscr{E}}\bullet\mathsf{v}_{\mathscr{E}} \in \mathsf{vs}_{\mathscr{E}}\}$
378 $\mathsf{c}_{\mathscr{E}}:\mathsf{C}_{\mathscr{E}} = \textbf{obs\_part\_}\mathsf{C}_{\mathscr{E}}(\delta_{\mathscr{E}})$
379 $\mathsf{ci}_{\mathscr{E}}:\mathsf{CI}_{\mathscr{E}} = \textbf{uid\_}\mathsf{CI}(\mathsf{c}_{\mathscr{E}})$

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.

380 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
380 $\mathsf{sys}\text{: }\textbf{Unit} \rightarrow \textbf{Unit}$
380 $\mathsf{sys}() \equiv$
380a $\quad \| \{\mathsf{veh}_{\textbf{uid\_}V(v)}(\textbf{obs\_mereo\_}\mathsf{V}(\mathsf{v}))|\mathsf{v}:\mathsf{V}\bullet\mathsf{v} \in \mathsf{vs}\}$

```
380b    ∥ ∥ {gate$_{uid\_EG(eg)}$(obs_mereo_G(eg),"entry")|eg:EG•eg ∈ egs}
380c    ∥ ∥ {gate$_{uid\_XG(xg)}$(obs_mereo_G(xg),"exit")|xg:XG•xg ∈ xgs}
380d      ∥  calc$_{uid\_C(c)}$(vis,gis)(rlf)(trm)

381  veh$_{vi}$: (ci:CI×gis:GI-set) → in attr_TiGPos[vi] out v_c_ch[vi,ci] Unit
387  gate$_{gi}$: (ci:CI×VI-set×LI)×ee:EE →
387      in attr_entry_ch[gi,ci],attr_id_ch[gi,ci],attr_exit_ch[gi,ci]
387      out attr_barrier_ch[gi],g_c_ch[gi,ci]  Unit
421  calc$_{ci}$: (vis:VI-set×gis:GI-set)×VPLF→TRM→
421      in {v_c_ch[vi,ci]|vi:VI•vi ∈ vis},{g_c_ch[gi,ci]|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 [67, 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 299 and Items 300, Page 164 and, projected, Page 171.

381 Instead of moving around by explicitly expressed internal non-determinism[16] vehicles move around by unstated internal non-determinism and instead receive their current position from the global positioning subsystem.
382 At each moment the vehicle receives its time-stamped global position, (τ,gpos):TiGPos,
383 from which it calculates the local position, lpos:VPos
384 which it then communicates, with its vehicle identification, (vi,(τ,lpos)), to the road pricing subsystem —
385 whereupon it resumes its vehicle behaviour.

```
value
381  veh$_{vi}$: (ci:CI×gis:GI-set) →
381      in attr_TiGPos_ch[vi] out v_c_ch[vi,ci] Unit
381  veh$_{vi}$(ci,gis) ≡
382      let ($\tau$,gpos) = attr_TiGPos_ch[vi]? in
383      let lpos = loc_pos(gpos) in
384      v_c_ch[vi,ci] ! (vi,($\tau$,lpos)) ;
385      veh$_{vi}$(ci,gis) end end
381      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* [67, 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 [43].

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:

[16] We refer to Items 299b, 299c on Page 164 and 300b, 300(b)ii, 300c on Page 164

386 We parameterise the toll-gate behaviour as either an entry or an exit gate.
387 Toll-gates operate autonomously and cyclically.
388 The attr_enter_ch event "triggers" the behaviour specified in formula line Item 389–391 starting with a "Raise" barrier action.
389 The time-of-passing and the identity of the passing vehicle is sensed by attr_passing_ch channel events.
390 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.
391 And finally, after that vehicle has left the entry or exit gate the barrier is again "Lower"ered and
392 that toll-gate's behaviour is resumed.

```
type
386   EE = "enter" | "exit"
value
387   gate_gi: (ci:CI×VI-set×LI)×ee:EE →
387         in attr_enter_ch[gi],attr_passing_ch[gi],attr_leave_ch[gi]
387         out attr_barrier_ch[gi],g_c_ch[gi,ci] Unit
387   gate_gi((ci,vis,li),ee) ≡
388         attr_enter_ch[gi] ? ; attr_barrier_ch[gi] ! "Lower"
389         let (τ,vi) = attr_passing_ch[gi] ? in assert vi ∈ vis
390         (attr_barrier_ch[gi] ! "Raise"
390          ∥ g_c_ch[gi,ci] ! (ee,(vi,(τ,SonL(li))))) ;
391         attr_leave_ch[gi] ? ; attr_barrier_ch[gi] ! "Lower"
392         gate_gi((ci,vis,li),ee)
387         end
387         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* [67, 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 (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 [43] ■

We shall define the *cal*culator behaviour in Sect. 5.5.1 on Page 191. 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 [240, 149, 178]. 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 [240, 149, 178]. [149] 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 32 **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 requirements*', $p_{d_{r_1}}, p_{d_{r_2}}, \ldots, p_{d_{r_n}}$, and m *shared domain requirements*, $s_{d_{r_1}}, s_{d_{r_2}}, \ldots, s_{d_{r_m}}$, 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 33 **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 requirements** we mean a domain requirements which is short of (that is, is missing) some prescription parts: text and formula ■ By a **shared domain requirements** 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 [43].

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 381- 385 on Page 184. The toll-gate monitoring and control [sub-]system would focus on implementing the calculator behaviour, see Items 387- 392 on the preceding page. The fitting amounts to (a) making precise the narrative and formal texts 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 ![17] 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 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).

[17] So how do we cope with the statement: *"the system must be user friendly"* ? We refer to Sect. 5.5.3 on Page 195 for a discussion of this issue.

Example 5.12. . **Interface Requirements — Projected Extensions**: We refer to Fig. 5.2 on Page 180.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
340 {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 387–392 Instead we represent their dynamic gate attributes:

(350) the vehicle entry sensors (leftmost ■s),
(350) the vehicle identity sensor (center ■), and
(351) the vehicle exit sensors (rightmost ■s)

by channels — we refer to Example 5.10 (Sect. 5.5.1, Page 180):

channel
350 {attr_entry_ch[gi]|gi:GI•xtr_eGIds(trn)} ″enter″
350 {attr_exit_ch[gi]|gi:GI•xtr_xGIds(trn)} ″exit″
351 {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.

393 There are r_l and r_h “recorders” recording link, respectively hub properties – with each recorder having a unique identity.
394 Each recorder is charged with the recording of a set of links or a set of hubs according to some partitioning of all such.
395 The recorders inform a central data base, net_db, of their recordings (ri,hol,(u$_j$,m$_j$,attrs$_j$)) where

396 ri is the identity of the recorder,
397 hol is either a `hub` or a `link` literal,
398 u_j = **uid_**L(l) or **uid_**H(h) for some link or hub,
399 m_j = **obs_mereo_**L(l) or **obs_mereo_**H(h) for that link or hub and
400 $attrs_j$ are *attributes* for that link or hub — where *attributes* is a function which "records" all respective static and dynamic attributes (left undefined).

```
type
393   RI
value
393   rl,rh:NAT axiom rl>0 ∧ rh>0
type
395   M = RI×"link"×LNK | RI×"hub"×HUB
395   LNK = LI × HI-set × LATTRS
395   HUB = HI × LI-set × HATTRS
```

```
value
394   partitioning: L-set→Nat→(L-set)*
394                   | H-set→Nat→(H-set)*
394   partitioning(s)(r) as sl
394   post: len sl = r ∧ ∪ elems sl = s
394            ∧ ∀ si,sj:(L-set|H-set) •
394               si≠{}∧sj≠{}∧{si,sj}⊆elems ss⇒si ∩ sj={}
```

401 The $r_l + r_h$ recorder behaviours interact with the one net_db behaviour

```
channel
401   r_db: RI×(LNK|HUB)
value
401   link_rec: RI → L-set → out r_db  Unit
401   hub_rec: RI → H-set → out r_db  Unit
401   net_db: Unit → in r_db  Unit
```

402 The data base behaviour, net_db, offers to receive messages from the link and hub recorders.
403 The data base behaviour, net_db, deposits these messages in respective variables.
404 Initially there is a net, $n : N$,
405 from which is observed its links and hubs.
406 These sets are partitioned into r_l, respectively r_h length lists of non-empty links and hubs.
407 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.
408 We construct, for technical reasons, as the reader will soon see, disjoint lists of link, respectively hub recorder identities.

```
value
402   net_db:
variable
403   lnk_db: (RI×LNK)-set
403   hub_db: (RI×HUB)-set
value
```

```
404   n:N
405   ls:L-set = obs_Ls(obs_LS(n))
405   hs:H-set = obs_Hs(obs_HS(n))
406   lsl:(L-set)* = partitioning(ls)(rl)
406   lhl:(H-set)* = partitioning(hs)(rh)
408   rill:RI* axiom len rill = rl = card elems rill
408   rihl:RI* axiom len rihl = rh = card elems rihl
407   ab_initio_data: Unit → Unit
407   ab_initio_data() ≡
407      ‖ {lnk_rec(rill[i])(lsl[i])|i:Nat•1≤i≤rl} ‖
407      ‖ {hub_rec(rihl[i])(lhl[i])|i:Nat•1≤i≤rh}
407      ‖ net_db()
```

409 The link and the hub recorders are near-identical behaviours.
410 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
411 the unique identifier,
412 the mereology, and
413 the attributes.
414 These "data" are sent, as a message, prefixed the senders identity, to the data base behaviour.
415 We presently leave the . . . unexplained.

```
value
401   link_rec: RI → L-set → Unit
409   link_rec(ri,ls) ≡
410      for ∀ l:L•l ∈ ls do uid_L(l)
411         let lnk = (uid_L(l),
412                    obs_mereo_L(l),
413                    attributes(l)) in
414         rdb ! (ri,"link",lnk);
415         ... end
410     end


401   hub_rec: RI × H-set → Unit
409   hub_rec(ri,hs) ≡
410      for ∀ h:H•h ∈ hs do uid_H(h)
411         let hub = (uid_L(h),
412                    obs_mereo_H(h),
413                    attributes(h)) in
414         rdb ! (ri,"hub",hub);
415         ... end
410     end
```

416 The net_db data base behaviour revolves around a seemingly "never-ending" cyclic process.
417 Each cycle "starts" with acceptance of some,
418 either link or hub data.
419 If link data then it is deposited in the link data base,
420 if hub data then it is deposited in the hub data base.

```
value
416   net_db() ≡
417      let (ri,hol,data) = r_db ? in
418      case hol of
419         "link" → ... ; lnk_db := lnk_db ∪ (ri,data),
420         "hub"  → ... ; hub_db := hub_db ∪ (ri,data)
418      end end ;
416′      ... ;
416      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 415); (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 416′) ■

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 178–185.

Example 5.15. . **Interface Requirements — Shared Behaviours**: **Road Pricing Calculator Behaviour:**

421 The road-pricing calculator alternates between offering to accept communication from
422 either any vehicle
423 or any toll-gate.

```
421   calc: ci:CI×(vis:VI-set×gis:GI-set)→RLF→TRM→
422        in {v_c_ch[ci,vi]|vi:VI•vi ∈ vis},
423           {g_c_ch[ci,gi]|gi:GI•gi ∈ gis}  Unit
421   calc(ci,(vis,gis))(rlf)(trm) ≡
422      react_to_vehicles(ci,(vis,gis))(rlf)(trm)
421    □
423      react_to_gates(ci,(vis,gis))(rlf)(trm)
421      pre ci = ci_ℰ ∧ vis = vis_ℰ ∧ gis = gis_ℰ
```

The *cal*culator 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 [67, Sect. 4.1.1 and 4.5.2].

424 If the communication is from a vehicle inside the toll-road net
425 then its toll-road net position, vp, is found from the road location function, rlf,
426 and the calculator resumes its work with the traffic map, trm, suitably updated,
427 otherwise the calculator resumes its work with no changes.

```
422   react_to_vehicles(ci,(vis,gis),vplf)(trm) ≡
422      let (vi,(τ,lpos)) = [] {v_c_ch[ci,vi]?|vi:VI•vi∈ vis} in
424         if vi ∈ dom trm
425            then let vp = vplf(lpos) in
426                   calc(ci,(vis,gis),vplf)(trm†[vi↦trm⌢⟨(τ,vp)⟩]) end
427            else   calc(ci,(vis,gis),vplf)(trm) end end
```

428 If the communication is from a gate,
429 then that gate is either an entry gate or an exit gate;
430 if it is an entry gate
431 then the calculator resumes its work with the vehicle (that passed the entry gate) now recorded, afresh, in the traffic map, trm.
432 Else it is an exit gate and
433 the calculator concludes that the vehicle has ended its to-be-paid-for journey inside the toll-road net, and hence to be billed;
434 then the calculator resumes its work with the vehicle now removed from the traffic map, trm.

```
423   react_to_gates(ci,(vis,gis),vplf)(trm) ≡
423      let (ee,(τ,(vi,li))) = [] {g_c_ch[ci,gi]?|gi:GI•gi∈ gis} in
429      case ee of
430         "Enter" →
431            calc(ci,(vis,gis),vplf)(trm∪[vi↦⟨(τ,SonL(li))⟩]),
432         "Exit" →
433            billing(vi,trm(vi)⌢⟨(τ,SonL(li))⟩);
434            calc(ci,(vis,gis),vplf)(trm\{vi}) end end
```

The above behaviour is the one for which we are to design software ■

5.5.2 Derived Requirements

Definition 34 **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 35 **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 Page 191. 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 421–434, Pages 191–192.

435 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*[18], that maps vehicle identifiers into time-stamped sequences of simple vehicle positions, i.e., as a subset of the *trm:TRM* programmable attribute.
436 The actor-to-calculator *begin* or *end* trace command, *cmd:Cmd*, is modeled as an autonomous dynamic attribute of the *cal*culator.
437 The *cal*culator signature is furthermore augmented with the three attributes mentioned above.
438 The occurrence and handling of an actor trace command is modeled as a non-deterministic external choice and a *react_to_trace_cmd* behaviour.
439 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
435a   TRA = VI-set
435b   VDU = TRM
436      Cmd = BTr | ETr
436      BTr :: VI
436      ETr :: VI
```

```
value
437   calc: ci:CI×(vis:VI-set×gis:GI-set) → RLF → TRM → TRA
422,423     in {v_c_ch[ci,vi]|vi:VI•vi ∈ vis},
422,423        {g_c_ch[ci,gi]|gi:GI•gi ∈ gis},
438,439        attr_cmd_ch,attr_vdu_ch  Unit
421   calc(ci,(vis,gis))(rlf)(trm)(tra) ≡
422         react_to_vehicles(ci,(vis,gis),)(rlf)(trm)(tra)
423     ⌷ react_to_gates(ci,(vis,gis))(rlf)(trm)(tra)
438     ⌷ react_to_trace_cmd(ci,(vis,gis))(rlf)(trm)(tra)
421       pre ci = ci_ℰ ∧ vis = vis_ℰ ∧ gis = gis_ℰ
439       axiom □ attr_vdu_ch[ci]? = trm|tra
```

The *438,439 attr_cmd_ch,attr_vdu_ch* of the *cal*culator signature models the *cal*culator's external *command* and *visual display unit* attributes.

440 The *react_to_trace_cmd* alternative behaviour is either a "Begin" or an "End" request which identifies the affected vehicle.

[18] VDU: visual display unit

441 If it is a ”Begin” request
442 and the identified vehicle is already being traced then we do not prescribe what to do !
443 Else we resume the calculator behaviour, now recording that vehicle as being traced.
444 If it is an ”End” request
445 and the identified vehicle is already being traced then we do not prescribe what to do !
446 Else we resume the calculator behaviour, now recording that vehicle as no longer being traced.

```
440          react_to_trace_cmd(ci,(vis,gis))(vplf)(trm)(tra) ≡
440              case attr_cmd_ch[ci]? of
441,442,443          mkBTr(vi) → if vi ∈ tra then chaos else calc(ci,(vis,gis))(vplf)(trm)(tra ∪ {vi}) end
444,445,446          mkETr(vi) → if vi ∉ tra then chaos else calc(ci,(vis,gis))(vplf)(trm)(tra\{vi}) end
440              end
```

The above behaviour, Items 421–446, is the one for which we are to design software ■

Example 5.16 exemplifies an action requirement as per definition 35: (a) the action is conceptual, it has no physical counterpart in the domain; (b) it calculates (439) 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 36 **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 ■

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 191. 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 421–446, Pages 191–194.

447 We augment the *cal*culator signature with
448 a time-stamped natural number valued dynamic programmable attribute, *(t:𝕋,max:Max)*.
449 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.
450 No changes are to be made to the react_to_gates behaviour (Items 423–434 Page 192) when a vehicle exits the toll-road net.

```
type
448        MAX = 𝕋 × NAT
value
437,447    calc: ci:CI×(vis:VI-set×gis:GI-set) → RLF → TRM → TRA → MAX
422,423        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
423        react_to_gates(ci,(vis,gis))(vplf)(trm)(tra)(t,m) ≡
423            let (ee,(τ,(vi,li))) = ⌈⌉{g_c_ch[ci,gi]|gi:GI•gi∈ gis} in
429            case ee of
449                "Enter" →
449                    calc(ci,(vis,gis))(vplf)(trm∪[vi↦⟨(τ,SonL(li))⟩])(tra)(τ,if card dom trm=m then m+1 else m end),
450                "Exit" →
450                    billing(vi,trm(vi)⌢⟨(τ,SonL(li))⟩); calc(ci,(vis,gis))(vplf)(trm\{vi})(tra)(t,m) end
429            end
```

The above behaviour, Items 421 on Page 191 through 449c on the facing page, is the one for which we are to design software ■

Example 5.17 exemplifies a derived event requirement as per Definition 36: (a) the event is conceptual, it has no physical counterpart in the domain; (b) it calculates (449b) 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 [108, 154, 231].

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. [30, 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 [30, 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 [145, 231, 146] 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[19] and steps[20] of this "derivation"[21] 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 19., of domain requirements into five steps, cf. Footnote 20 (Page 196), 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 descriptions*. Only by including *domain facet descriptions* can we, in "deriving" *requirements prescriptions*, 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 [231, Axel van Lamsweerde]. [231] 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.

[19] (a) domain, (b) interface and (c) machine requirements

[20] For domain requirements: (i) projection, (ii) instantiation, (iii) determination, (iv) extension and (v) fitting; etc.

[21] We use double quotation marks: "... " to indicate that the derivation is not automatable.

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 [149] and [108], referring in any technical sense to ‘domains’ such as we understand them.

[149, 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 [149, Jones, Hayes & Jackson]. A purpose of [149] (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 [149] (earlier in the same paragraph) as in contrast to our form of domain modeling. [149] reveals assumptions about the domain when expressing *rely guarantees* in tight conjunction with expressing the *guarantee* (requirements). That is, analysing and expressing requirements, in [149], goes hand-in-hand with analysing and expressing fragments of the domain. The current chapter takes the view that since, as demonstrated in [67], 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 [149]. We do appreciate the “tight coupling” of rely–guarantees of [149]. 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 express. Our, i.e., the present, chapter — with its background in [67, Sect. 4.7] — develops — with a background in [144, M.A. Jackson] — a set of principles and techniques for the access of attributes. The “discovery” of the `CM` and `SG` channels of [149] and of the type of their messages, seems, compared to our approach, less systematic. Also, it is not clear how the [149] case study “scales” up to a larger domain. The *sluice gate* of [149] 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 [149], 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.

[108, 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 [108] as follows: Agents, actions, entities and relations are, in [108], seen as specialisations of a concept of *objects*. The nearest analogy to relations, in [67], 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 ! [108] otherwise include describing parts, relations between parts, actions and events much like [67] and this chapter does. [108] then introduces a notion of *goal*. *A* ***goal****, in [108], 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* ■[22]″ [108] then goes on to exemplify goals. In this, the

[22] 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

current chapter, we are not considering *goal*s, also a major theme of [231].[23] Typically the expression of goals of [108, 231], are "within" computer & computing science and involve the use of temporal logic.[24] *″Constraints are operational objectives 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″* [108]. [108] then goes on to express goals and constraints operationalising these. [108] is a fascinating paper[25] as it shows how to build goals and constraints on domain description fragments.

•••

These papers, [149] and [108], as well as the current chapter, together with such seminal monographs as [240, 178, 231], clearly shows that there are many diverse ways in which to achieve precise requirements prescriptions. The [240, 178] 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[26]. 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 [240, 178, 231] 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.

5.8 Bibliographical Notes

I have thought about domain engineering for more than 20 years. But serious, focused writing only started to appear since [30, Part IV] — with [25, 22] being exceptions: [32] suggests a number of domain science and engineering research topics; [43] covers the concept of domain facets; [80] explores compositionality and Galois connections. [33, 79] show how to systematically, but, of course, not automatically, "derive" requirements prescriptions from domain descriptions; [48] takes the triptych software development as a basis for outlining principles for believable software management; [39, 55] presents a model for Stanisław Leśniewski's [96] concept of mereology; [44, 49] present an extensive example and is otherwise a precursor for the present chapter; [51] 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; [52] analyses the `TripTych`, especially its domain engineering approach, with respect to [162, 163, Maslow]'s and [183, Peterson's and Seligman's]'s notions of humanity: how can computing relate to notions of humanity; the first part of [58] is a precursor for [67] with the second part of [58] 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 [59] focus on domain safety criticality.

and actions 'available' to some agent of the system. For example, wrt. the ongoing library examples of [108], *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.

[23] 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.

[24] 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 [240, `The Duration Calculus`].

[25] — that might, however, warrant a complete rewrite.

[26] The Duration Calculus [`DC`], respectively `DC, Timed Automata` and `Z`

Part III

Some Implications for Software

6

Demos, Simulators, Monitors and Controllers
A Divertimento of Ideas and Suggestions.

We[1] muse over the concepts of 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[2] 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 [67, 71, 63].

"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

[1] This chapter is a slightly edited rendition of [50].

[2] – take that of Chapter 1

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.

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 [32, 80, 37, 49], and (β) a software development theory of domain engineering versus requirements engineering [48, 33, 38, 44].

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. Hence the "divertimento" suffix to the chapter title.

Structure of Chapter:

The structure of the chapter is as follows. 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[3], cf. [63].

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* [127].

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 to to be.***

We further refer to the literature for examples: [23, *railways* (2000)], [24, *the 'market'* (2000)], [38, *public government, IT security, hospitals* (2006) chapters 8–10], [33, *transport nets* (2008)] and [44, *pipelines* (2010)]. On the net you may find technical reports covering "larger" domain descriptions. "Older" publications on the concept of domain descriptions are [44, 49, 39, 80, 33, 32, 43] all summarised in [67, 71, 63].

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.

[3] 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[4]: 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 Chapter 1. The other is summarised below. It is from Chapter 5 on "deriving requirements prescriptions from domain descriptions". The summary follows.

The domain description of Sect. 5.2 outlines an abstract concept of transport nets (of hubs [street intersections, train stations, harbours, airports] and links [road segments, rail tracks, shipping lanes, air-lanes]), 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.

[4] The concepts of 'endurants' and 'perdurants' were defined in [67].

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.

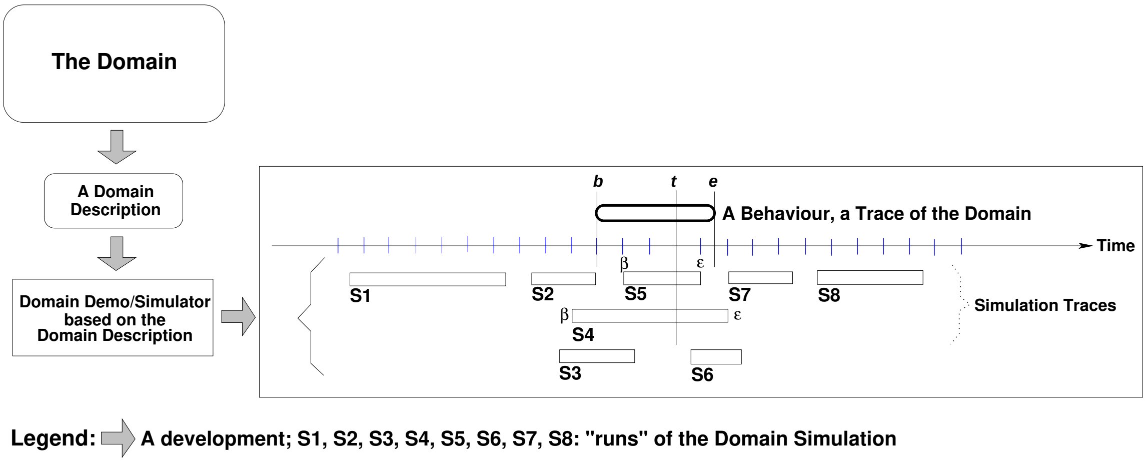

The Domain
A Domain Description
Domain Demo/Simulator based on the Domain Description
b
t
e
A Behaviour, a Trace of the Domain
Time
β
ε
S1
S2
S5
S7
S8
β
ε
S4
S3
S6
Simulation Traces
Legend:
A development; S1, S2, S3, S4, S5, S6, S7, S8: "runs" of the Domain Simulation

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 206 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 $\frac{t-\mathsf{b}}{\mathsf{e\text{-}t}} = \frac{t-\beta}{\varepsilon-t}$ — 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"[5] 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.

[5] We deliberately leave the notion of chunk vague so as to allow as wide an spectrum of simulations.

6.3.3 Monitoring & Control

Figure 6.2 shows three different kinds of uses of software systems (where (2) [Monitoring] and (3) [Monitoring & Control] represent further) developments from the demo or simulation software system mentioned in Sect. 6.3.1 and Sect. 6.3.2 on the previous page. We have added some (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 204). 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.

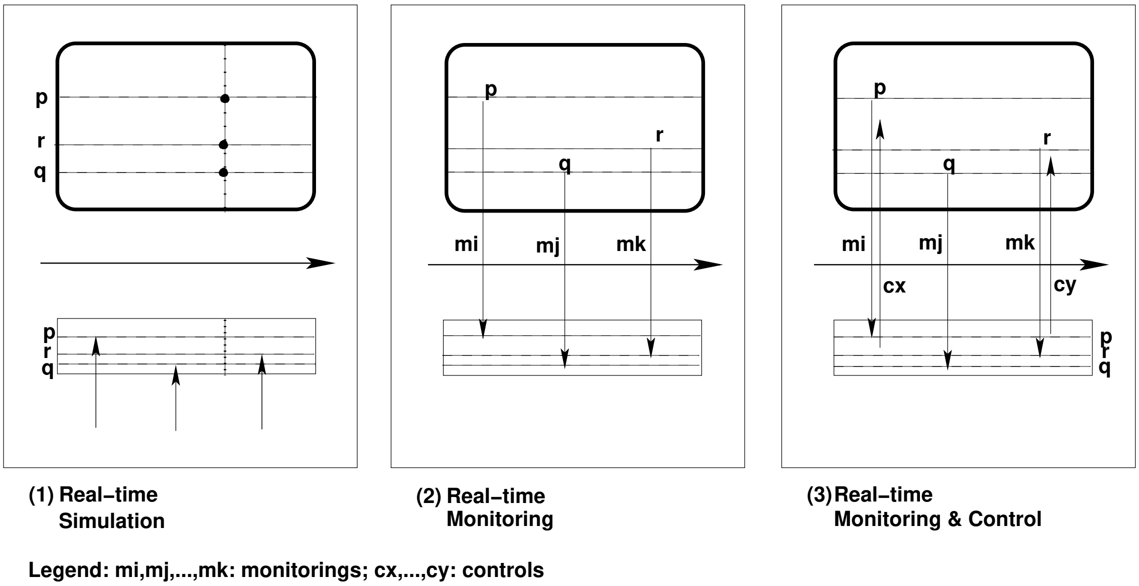

p
r
q
p
r
q
p
r
q
mi
mj
mk
p
r
q
mi
mj
mk
cx
cy
(1) Real–time Simulation
(2) Real–time Monitoring
(3) Real–time Monitoring & Control
Legend: mi,mj,...,mk: monitorings; cx,...,cy: controls

Fig. 6.2. Simulation, Monitoring and Monitoring & Control

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*[6]. We bring examples that are taken from Sect. 5.2, cf. Sect. 6.3.1 on Page 203 of the present chapter. (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

[6] We omit consideration of *fitting*.

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.)

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, $\mathbb{R}_{\text{DE}}$, for, and from $\mathbb{R}_{\text{DE}}$ one can develop a domain demo machine specification $\mathbb{M}_{\text{DE}}$ such that $\mathbb{D},\mathbb{M}_{\text{DE}} \models \mathbb{R}_{\text{DE}}$. 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, $\mathbb{S}_{\text{DE}}$, one can develop requirements, $\mathbb{R}_{\text{SI}}$, for, and from such a $\mathbb{R}_{\text{SI}}$ one can develop a domain simulator machine specification $\mathbb{M}_{\text{SI}}$ such that $(\mathbb{D};\mathbb{M}_{\text{DE}}),\mathbb{M}_{\text{SI}} \models \mathbb{R}_{\text{SI}}$. We have "lumped" $(\mathbb{D};\mathbb{M}_{\text{DE}})$ 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.

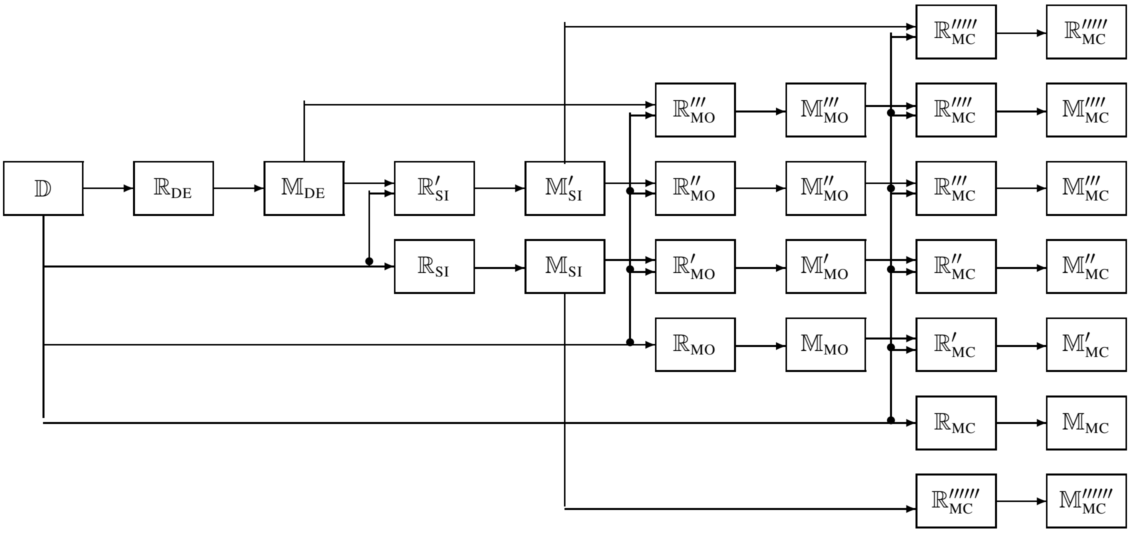
(D) From a domain description, $\mathbb{D}$, and a domain simulator machine specification, $\mathbb{M}_{\text{SI}}$, one can develop requirements, $\mathbb{R}_{\text{MO}}$, for, and from such a $\mathbb{R}_{\text{MO}}$ one can develop a domain monitor machine specification $\mathbb{M}_{\text{MO}}$ such that $(\mathbb{D};\mathbb{M}_{\text{SI}}),\mathbb{M}_{\text{MO}} \models \mathbb{R}_{\text{MO}}$.

(E) From a domain description, $\mathbb{D}$, and a domain monitor machine specification, $\mathbb{M}_{\text{MO}}$, one can develop requirements, $\mathbb{R}_{\text{MC}}$, for, and from such a $\mathbb{R}_{\text{MC}}$ one can develop a domain monitor & controller machine specification $\mathbb{M}_{\text{MC}}$ such that $(\mathbb{D};\mathbb{M}_{\text{MO}}),\mathbb{M}_{\text{MC}} \models \mathbb{R}_{\text{MC}}$.

Chains of Verifiable Developments

The above illustrated just one chain (A–E) of developments. There are others. All are shown in Fig. 6.3 on the facing page.

Figure 6.3 on the next page can also be interpreted as prescribing a widest possible range of machine cum software products [91, 186] 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 $\mathbb{M}_{\text{T}}$ 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 $\mathbb{M}_{\text{T}}$ machines). What does it mean that a machine is a primed version? Well, here it means, for example, that $\mathbb{M}'_{\text{SI}}$ embodies facets of the demo machine $\mathbb{M}_{\text{DE}}$, 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



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

monitor $\mathbb{M}''_{MO}$. Whether such requirements are desirable is left to product customers and their software providers [91, 186] 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 [17, 125] and the EU ESPRIT ProCoS [19, 20] projects; [127] 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 [32, 43] and one [134, 115, 93] emerging out of the Software Engineering Institute at CMU where it is also called *product line engineering*[7]. 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 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 [67, 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.

[7] http://en.wikipedia.org/wiki/Domain_engineering.

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 $(\mathbb{D};\mathbb{M}_{\mathrm{XYZ}}),\mathbb{M} \models \mathbb{R}$ clauses. That is, we should like to prove conjectured type theoretic inclusion relations like:

$$\wp([\![\mathscr{M}_{\mathrm{X}_{\underline{\mathsf{mod}}\ \mathit{ext.}}}]\!]) \sqsupseteq \wp([\![\mathscr{M}^{\prime\cdots\prime}_{\mathrm{X}_{\underline{\mathsf{mod}}\ \mathit{ext.}}}]\!]), \quad \wp([\![\mathscr{M}^{\prime\cdots\prime}_{\mathrm{X}_{\underline{\mathsf{mod}}\ \mathit{ext.}}}]\!]) \sqsupseteq \wp([\![\mathscr{M}^{\prime\prime\cdots\cdot\prime}_{\mathrm{X}_{\underline{\mathsf{mod}}\ \mathit{ext.}}}]\!])$$

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 $\wp([\![\mathscr{M}_{x_{\underline{\mathsf{mod}}\ \mathit{ext.}}}]\!])$ 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.

What Should We Do Next

This chapter has the subtitle: *A Divertimento of Ideas and Suggestions.* It is not a proper theoretical chapter. 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, [137] 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**

Conclusion

7

Summing Up

Each of Chapters 1–6 have their own closings. Here we summarise their conclusions.

7.1 What Have We Achieved ?

7.1.1 Chapter–by–Chapter Achievement Enumeration

Chapter 1, Pages 3–74: Domain Analysis & Description

The main contribution is that of introducing the domain analysis & description method: principles, technques and tools. It was hinted at that the ontology for domain entities can be justified on philosophical grounds. A paper, [74], will expand considerably on this topic. Conventional software engineering previously began with requirements engineering. Now a predecessor phase has been "put" before that.

I consider this the major contribution of this thesis.

Chapter 2, Pages 75–103: Domain Facets

The main contribution is that of introducing the concept of domain facets and its manifestation: domain intrinsics, domain support technology, domain rules & regulations, domain scripts, domain license languages, domain management & organisation, and domain humain behaviour.

Chapter 3, Pages 105–127: Towards Formal Models of Processes and Prompts

The contribution of this chapter is rather somewhat contrary to traditionalist thinking. Instead of formulating semantic domains for syntactic quantities we turn matters "upside-down": from semantic entities we "derive" syntactic ones !

Chapter 4, Pages 129–149: To Every Manifest Mereology a CSP Expression

The contribution of this chapter is both traditional and novel. Traditional, in that we show how Casati and Varzi's axiom system [96] for Leśniewski's mereology, can be given a model in terms of the domain ontoloy sorts of Chapter 1. Novel, in that we show, for the first time, in 2009, how manifest merologies, by transcendental deduction, can be "modelled" as `CSP` [137] processes.

Chapter 5, Pages 153–198: From Domains ... to Requirements ...

The contribution of this chapter is methdological. It is not a theory, but it is a set of principles and techniques for systematically “deriving” requirements prescriptions from domain descriptions. The principles include the separation of requirements concerns into *domain*, *interface* and *machine requirements*, and, within *domain requirements*, the novel concepts of domain *projection*, *instantiation*, *determination*, *extension* and *fitting*.

Chapter 6, Pages 201–210: Demos, Simulators, Monitors and Controllers

The contributions of this chapter are not of scientific nature. They are rather of a “pedagogical” engineering nature – in that they throw a different light on such notions as *demos*, *simulators*, *monitors* and *controllers* by relating these to relations between *domain descriptions*, *requirements prescriptions* and *software designs*.

7.1.2 Fulfillment of Thesis

We refer back to the statement of the thesis of this submission, cf. Page iv.

THE THESIS OF THIS MONOGRAPH

The thesis of this monograph is twofold:

- (i) *domain science & engineering* is a possible, initial phase of software development;
- (ii) *domain science & engineering* is a worthwhile topic of research.

We support this claim as follows:

- (a) the concepts of *domain science* and *domain engineering* are new;
- (b) the terms *domain science* and *domain engineering* are well-defined;
- (c) *domain science* and *domain engineering* are given a foundation in this thesis;
- (d) and their rôle in software development is established.

Domain Science & Engineering casts a completely new light on Software Development.

We can now conclude:

THESIS FULFILLED

We claim that the thesis has been fulfilled.

Dines BJØRNER

Dines Bjørner, September 6, 2019: 16:27
Fredsvej 11, DK–2840 Holte, Denmark

7.2 Acknowledgements

The author has worked on the topic of domain science and engineering since the early 1990s. Upon his retirement in 2007 he was able to devote full time to its study. The chapters of this thesis are all based on publications after 2007. Here I want to acknowledge the encouragement I have received from colleagues in Europe and Asia – for example by their giving me much needed interaction with PhD students. The list below includes references to either ■ books or domain case studies 🟥 [78] that were worked out during my stays (usually PhD lectures) at these colleagues:

- **Jin Song Dong**, July 2004 – June 2005, NUS, Singapore ■ [28]
- **Kokichi Futatsugi**, Feb. 2006 – Jan. 2007, JAIST, Kanazawa, Japan ■ [77, 34, 35, 36, 40, 41, 42]
- **Dominique Méry**, Oct. – Nov. 2007, Nancy, France
- **Wolfgang J. Paul**, March 2008, Saarbrücken, Germany
- **Bernhard K. Aichernig**, Oct. – Nov. 2008, Graz, Austria 🟥 [53]
- **Alan Bundy**, Sept. – Nov. 2009, Edinburgh, Scotland
- **Tetsuo Tamai**, Nov. – Dec. 2009, Tokyo, Japan 🟥 [47]
- **Jens Knoop**, April 2010, Vienna, Austria 🟥 [45]
- **Niklaj Nikitchenko**, May 2010, Kiev, Lviv, Odessa, Ukraine
- **Istenes Zoltán**, Oct. 2010, Budapest, Hungary
- **Andreas Hamfeldt**, Nov. 2010, Uppsala, Sweden
- **Sun Meng**, Nov. 2012, Beijing, China
- **Zhu HuBiao**, Dec. 2012, ECNU, Shanghai, China
- **Zhan NaiJun**, Dec. 2012, Beijing, China
- **Viktor Ivannikov**, April 2014, Moscow, Russia
- **Jin Song Dong**, May 2014, NUS, Singapore
- **José N. Oliveira**, May – June 2015, Braga, Portugal
- **Jens Knoop**, Oct. 2015, Vienna, Austria
- **Andreas Hamfeldt**, May 2016, Uppsala, Sweden 🟥 [60]
- **Magne Haveraaen**, Nov. 2016, Bergen, Norway 🟥 [64]
- **Otthein Herzog**, Sept., 2017, Tong Ji Univ., Shanghai, China 🟥 [88, 68]
- **Chin Wei Ngan**, NUS, Singapore
- **Zhu HuBiao**, 2018, ECNU, Shanghai, China 🟥 [70]
- **Mauro Pezze**, May 2019, Lugano, Switzerland
- **Dino Mandrioli**, May 2019, Milano, Italy

Part V

A Common Bibliography

8

Bibliography

1. 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.
2. Open Mobile Alliance. OMA DRM V2.0 Candidate Enabler. http://www.openmobilealliance.org/-release_program/drm_v2_0.html, Sep 2005.
3. Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, August 2003. ISBN 0521825830.
4. K. Araki et al., editors. *IFM 1999–2018: Integrated Formal Methods*,
    - 1st IFM, 1999: e-ISBN-13:978-1-1-4471-851-1,

    and

    - 2nd IFM, 2000: LNCS 1945,
    - 3rd IFM, 2002: LNCS 2335,
    - 4th IFM, 2004: LNCS 2999,
    - 5th IFM, 2005: LNCS 3771,
    - 6th IFM, 2007: LNCS 4591,
    - 7th IFM, 2009: LNCS 5423,
    - 8th IFM, 2010: LNCS 6396,
    - 9th IFM, 2012: LNCS 7321,
    - 10th IFM, 2013: LNCS 7940,
    - 11th IFM, 2014: LNCS 8739,
    - 12th IFM, 2016: LNCS 9681,
    - 13th IFM, 2017: LNCS 10510,
    - 14th IFM, 2018: LNCS 11023,
    - 15th IFM, 2018: LNCS 11918.

    Springer
    Lecture Notes in Computer Science

    .
5. 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.
6. Rober Audi. *The Cambridge Dictionary of Philosophy*. Cambridge University Press, The Pitt Building, Trumpington Street, Cambridge CB2 1RP, England, 1995.
7. John Longshaw Austin. *How To Do Things With Words*. Oxford University Press, second edition, 1976.
8. 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.
9. 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.
10. C. Bachman. Data structure diagrams. *Data Base, Journal of ACM SIGBDP*, 1(2), 1969.
11. Ralph-Johan Back, Abo Akademi, J. von Wright, and F. B. Schneider. *Refinement Calculus: A Systematic Introduction*. Springer–Verlag New York, Inc., 1998.
12. Alain Badiou. *Being and Event*. Continuum, 2005. (Lêtre et l'événements, Edition du Seuil, 1988).
13. Jaco W. de Bakker. *Control Flow Semantics*. The MIT Press, Cambridge, Mass., USA, 1995.
14. 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.

15. Yochai Benkler. Coase's Penguin, or Linux and the Nature of the Firm. *The Yale Law Journal*, 112, 2002.
16. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. EATCS Series: Texts in Theoretical Computer Science. Springer, 2004.
17. 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.
18. 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 [203].
19. 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.
20. 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.
21. 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.
22. 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.
23. 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.
24. 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. *URL:* http://www2.imm.dtu.dk/~dibj/themarket.pdf.
25. 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. *URL:* http://www2.imm.dtu.dk/~dibj/zohar.pdf.
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. *URL:* http://www2.imm.dtu.dk/~dibj/ifac-dynamics.pdf.
27. Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
28. Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. See [34, 40].
29. 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.
30. Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
31. Dines Bjørner. A Container Line Industry Domain. Techn. report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, June 2007. *URL:* imm.dtu.dk/~db/container-paper.pdf.
32. 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.
33. 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. *URL:* imm.dtu.dk/~dibj/montanari.pdf.
34. Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Qinghua University Press, 2008.
35. Dines Bjørner. *Software Engineering, Vol. 2: Specification of Systems and Languages*. Qinghua University Press, 2008.

36. Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Qinghua University Press, 2008.
37. 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.
38. 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:
    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.
39. 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. *URL:* imm.dtu.dk/~dibj/bjorner-hoare75-p.pdf.
40. Dines Bjørner. **Chinese:** *Software Engineering, Vol. 1: Abstraction and Modelling*. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.
41. Dines Bjørner. **Chinese:** *Software Engineering, Vol. 2: Specification of Systems and Languages*. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.
42. Dines Bjørner. **Chinese:** *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.
43. 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.
44. Dines Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics, Part I of II: The Engineering Part*. *Kibernetika i sistemny analiz*, 2(4):100–116, May 2010.
45. Dines Bjørner. On Development of Web-based Software: A Divertimento of Ideas and Suggestions. Technical, Technical University of Vienna, August–October 2010. *URL:* imm.dtu.dk/~dibj/wfdftp.pdf.
46. Dines Bjørner. The Tokyo Stock Exchange Trading Rules. R&D Experiment, Techn. Univ. of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, 2010. *URL:* imm.dtu.dk/~db/todai/tse-1.pdf, imm.dtu.dk/~db/todai/tse-2.pdf.
47. Dines Bjørner. The Tokyo Stock Exchange Trading Rules *URL:* himm.dtu.dk/~db/todai/tse-1.pdf, imm.dtu.dk/~db/todai/tse-2.pdf. R&D Experiment, Techn. Univ. of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, January, February 2010.
48. Dines Bjørner. Believable Software Management. *Encyclopedia of Software Engineering*, 1(1):1–32, 2011.
49. 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(3):100–120, June 2011.
50. 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. *URL:* imm.dtu.dk/~dibj/maurer-bjorner.pdf.
51. 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. *URL:* imm.dtu.dk/~dibj/maurer-bjorner.pdf.

52. 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).
53. Dines Bjørner. Pipelines – a Domain. Experimental Research Report 2013-2, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013. *URL:* imm.dtu.dk/~dibj/pipe-p.pdf.
54. Dines Bjørner. Road Transportation – a Domain Description. Experimental Research Report 2013-4, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013. *URL:* imm.dtu.dk/~dibj/road-p.pdf.
55. 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.
56. 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.
57. Dines Bjørner. Domain Analysis: Endurants – An Analysis & Description Process Model. In Shusaku Iida and José Meseguer and Kazuhiro Ogata, editor, *Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi*. Springer, May 2014. *URL:* imm.dtu.dk/~dibj/2014/kanazawa/kanazawa-p.pdf.
58. Dines Bjørner. Domain Analysis: Endurants – An Analysis & Description Process Model. In Shusaku Iida and José Meseguer and Kazuhiro Ogata, editor, *Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi*. Springer, May 2014. *URL:* imm.dtu.dk/~dibj/2014/kanazawa/kanazawa-p.pdf.
59. 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. *URL:* imm.dtu.dk/~dibj/2014/assc-april-bw.pdf.
60. Dines Bjørner. A Credit Card System: Uppsala Draft. Technical Report: Experimental Research, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, November 2016. *URL:* imm.dtu.dk/~dibj/2016/credit/accs.pdf.
61. 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 [58]. *URL:* imm.dtu.dk/~dibj/2016/process/process-p.pdf.
62. 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 [50]. *URL:* imm.dtu.dk/~dibj/2016/demos/faoc-demo.pdf.
63. 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 [33] *URL:* compute.dtu.dk/~dibj/2015/faoc-req/faoc-req.pdf.
64. 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. *URL:* imm.dtu.dk/~dibj/2016/wis/wis-p.pdf.
65. Dines Bjørner. Manifest Domains: Analysis & Description – A Philosophical Basis. , 2016–2017. *URL:* imm.dtu.dk/~dibj/2016/apb/daad-apb.pdf.
66. Dines Bjørner. A Space of Swarms of Drones. Research Note, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, December 2017. *URL:* imm.dtu.dk/~dibj/2017/swarms/swarm-paper.pdf.
67. Dines Bjørner. Manifest Domains: Analysis & Description. *Formal Aspects of Computing*, 29(2):175–225, March 2017. Online: 26 July 2016.
68. Dines Bjørner. What are Documents? Research Note, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, July 2017. *URL:* imm.dtu.dk/~dibj/2017/docs/docs.pdf.
69. Dines Bjørner. A Philosophy of Domain Science & Engineering – An Interpretation of Kai Sørlander's Philosophy. Research Note, 95 pages, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, Spring 2018. *URL:* imm.dtu.dk/~dibj/2018/philosophy/filo.pdf.
70. Dines Bjørner. Container Terminals. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, September 2018. An incomplete draft report; currently 60+ pages. *URL:* imm.dtu.dk/~dibj/2018/yangshan/maersk-pa.pdf.
71. Dines Bjørner. Domain Facets: Analysis & Description. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, May 2018. Extensive revision of [43]. *URL:* imm.dtu.dk/~dibj/2016/facets/faoc-facets.pdf.

72. Dines Bjørner. Domain Science & Engineering – A Review of 10 Years Work and a Laudatio. In NaiJun Zhan and Cliff B. Jones, editors, *Symposium on Real-Time and Hybrid Systems – A Festschrift Symposium in Honour of Zhou ChaoChen*, LNCS 11180, pp. 6184. Springer Nature Switzerland AG *URL:* imm.dtu.dk/~dibj/2017/zcc/ZhouBjorner2017.pdf, June 2018.
73. Dines Bjørner. To Every Manifest Domain a CSP Expression — A Rôle for Mereology in Computer Science. *Journal of Logical and Algebraic Methods in Programming*, 1(94):91–108, January 2018. *URL:* compute.dtu.dk/~dibj/2016/mereo/mereo.pdf.
74. Dines Bjørner. Domain Analysis & Description – A Philosophy Basis. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, November 2019. Submitted for review. *URL:* imm.dtu.dk/~dibj/2019/filo/main2.pdf.
75. Dines Bjørner. Domain Analysis & Description – Principles, Techniques and Modelling Languages. *ACM Trans. on Software Engineering and Methodology*, 28(2):66 pages, March 2019. *URL:* imm.dtu.dk/~dibj/2018/tosem/Bjorner-TOSEM.pdf.
76. Dines Bjørner. Domain Analysis & Description – Principles, Techniques and Modelling Languages. *ACM Trans. on Software Engineering and Methodology*, 28(2), April 2019. 68 pages. *URL:* imm.dtu.dk/~dibj/2018/tosem/Bjorner-TOSEM.pdf.
77. Dines Bjørner. *Domain Engineering: Technology Management, Research and Engineering*. A JAIST Press Research Monograph # 4, 536 pages, March 2009.
78. Dines Bjørner. Domain Case Studies:
    - 2019: *Container Line*, ECNU, Shanghai, China *URL:* imm.dtu.dk/~db/container-paper.pdf
    - 2018: *Documents*, TongJi Univ., Shanghai, China *URL:* imm.dtu.dk/~dibj/2017/docs/docs.pdf
    - 2017: *Urban Planning*, TongJi Univ., Shanghai, China *URL:* imm.dtu.dk/~dibj/2018/BjornerUrbanPlanning24Jan2018.pdf
    - 2017: *Swarms of Drones*, Inst. of Softw., Chinese Acad. of Sci., Peking, China *URL:* imm.dtu.dk/~dibj/2017/swarms/swarm-paper.pdf
    - 2013: *Road Transport*, Techn. Univ. of Denmark *URL:* imm.dtu.dk/~dibj/road-p.pdf
    - 2012: *Credit Cards*, Uppsala, Sweden *URL:* imm.dtu.dk/~dibj/2016/credit/accs.pdf
    - 2912: *Weather Information*, Bergen, Norway *URL:* imm.dtu.dk/~dibj/2016/wis/wis-p.pdf
    - 2010: *Web-based Transaction Processing*, Techn. Univ. of Vienna, Austria *URL:* imm.dtu.dk/~dibj/wfdftp.pdf
    - 2010: *The Tokyo Stock Exchange*, Tokyo Univ., Japan *URL:* imm.dtu.dk/~db/todai/tse-1.pdf, *URL:* imm.dtu.dk/~db/todai/tse-2.pdf
    - 2009: *Pipelines*, Techn. Univ. of Graz, Austria *URL:* imm.dtu.dk/~dibj/pipe-p.pdf
    - 2007: *A Container Line Industry Domain*, Techn. Univ. of Denmark *URL:* imm.dtu.dk/~dibj/container-paper.pdf
    - 2002: *The Market*, Techn. Univ. of Denmark *URL:* imm.dtu.dk/~dibj/themarket.pdf
    - 1995–2004: *Railways*, Techn. Univ. of Denmark - a compendium *URL:* imm.dtu.dk/~dibj/train-book.pdf

    Experimental research reports, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark.
79. 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.
80. 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.
81. 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. *URL:* http://www2.imm.dtu.dk/~dibj/pasadena-25.pdf.
82. Dines Bjørner and Klaus Havelund. 40 Years of Formal Methods — 10 Obstacles and 3 Possibilities. In *FM 2014, Singapore, May 14-16, 2014*. Springer, 2014. Distinguished Lecture. *URL:* imm.dtu.dk/~dibj/2014/fm14-paper.pdf.

83. Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.
84. Dines Bjørner and Cliff B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.
85. 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.
86. Nikolaj Bjørner, Anca Browne, Michael Colon, Bernd Finkbeiner, Zohar Manna, Henny Sipma, and Tomas Uribe. Verifying Temporal Properties of Reactive Systems: A STeP Tutorial. *Formal Methods in System Design*, 16:227–270, 2000.
87. Nikolaj Bjørner, Ken McMillan, and Andrey Rybalchenko. Higher-order Program Verification as Satisfiability Modulo Theories with Algebraic Data-types. In *Higher-Order Program Analysis*, June 2013. http://hopa.cs.rhul.ac.uk/files/proceedings.html.
88. Dines Bjørner. Urban Planning Processes. Research Note, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, July 2017. *URL:* imm.dtu.dk/~dibj/2017/up/urban-planning.pdf.
89. Bruno Blanchet, Patrick Cousot, Radhia Cousot, Laurent Mauborgne Jerome Feret, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Programming Language Design and Implementation*, pages 196–207, 2003 .
90. Grady Booch, Jim Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
91. Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. ACM Press/Addison-Wesley, New York, NY, 2000.
92. 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.
93. F. Buschmann, K. Henney, and D.C. Schmidt. *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*. John Wiley & Sons Ltd., England, 2007.
94. Roberto Casati and Achille Varzi. Events. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, spring 2010 edition, 2010.
95. 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.
96. Roberto Casati and Achille C. Varzi. *Parts and Places: the structures of spatial representation*. MIT Press, 1999.
97. C.E.C. Digital Rights: Background, Systems, Assessment. Commission of The European Communities, Staff Working Paper, 2002. Brussels, 14.02.2002, SEC(2002) 197.
98. Peter P. Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Trans. Database Syst*, 1(1):9–36, 1976.
99. 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.
100. 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.
101. 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.
102. Jesper Vinther Christensen. *Specifying Geographic Information – Ontology, Knowledge Representation, and Formal Constraints*. Phd thesis, Technical University of Denmark, Computer Science and Engineering, DK 2800 Kgs. Lyngby, August 2007.
103. 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, *URL:* 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, 2015. 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.

104. E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.
105. CoFI (The Common Framework Initiative). CASL *Reference Manual*, volume 2960 of *Lecture Notes in Computer Science (IFIP Series)*. Springer–Verlag, 2004.
106. Patrick Cousot and Rhadia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4th POPL: Principles of Programming and Languages*, pages 238–252. ACM Press, 1977.
107. Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, 2000.
108. Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Sci. Comput. Program.*, 20(1-2):3–50, April 1993.
109. Donald Davidson. *Essays on Actions and Events*. Oxford University Press, 1980.
110. The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.
111. Merlin Dorfman and Richard H. Thayer, editors. *Software Requirements Engineering*. IEEE Computer Society Press, 1997.
112. F. Dretske. Can Events Move? *Mind*, 76(479-492), 1967. Reprinted in [95, 1996], pp. 415-428.
113. ESA. Global Navigation Satellite Systems. Web, http://en.wikipedia.org/wiki/Satellite_navigation, European Space Agency. There are several global navigation satellite systems (http://en.wikipedia.org/wiki/Satellite_navigation) either in operation or being developed: (1.) the US developed and operated `GPS` (`NAVSTAR`) system, http://en.wikipedia.org/wiki/Global_Positioning_System; (2.) the EU developed and (to be) operated `Galileo` system, http://en.wikipedia.org/wiki/Galileo_positioning_system; (3.) the Russian developed and (to be) operated `GLONASS`, http://en.wikipedia.org/wiki/GLONASS; and (4.) the Chinese `Compass Navigation System`, http://en.wikipedia.org/wiki/Compass_navigation_system.
114. 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.
115. 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.
116. 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.
117. Edward A. Feigenbaum and Pamela McCorduck. *The fifth generation*. Addison-Wesley, Reading, MA, USA, 1st ed. edition, 1983.
118. 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.
119. Martin Fowler. *Domain Specific Languages*. Signature Series. Addison Wesley, October 20120.
120. Carlo A. Furia, Dino Mandrioli, Angelo Morzenti, and Matteo Rossi. *Modeling Time in Computing*. Monographs in Theoretical Computer Science. Springer, 2012.
121. 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.
122. Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis — Mathematical Foundations*. Springer-Verlag, January 1999.
123. 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.
124. 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.
125. 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.
126. T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer, Dordrecht, 2002.
127. 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.

128. 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.
129. C.A. Gunther. *Semantics of Programming Languages*. The MIT Press, Cambridge, Mass., USA, 1992.
130. P.M.S. Hacker. Events and Objects in Space and Time. *Mind*, 91:1–19, 1982. reprinted in [95], pp. 429-447.
131. Joseph Y. Halpern and Vicky Weissman. A Formal Foundation for XrML. In *Proc. of the 17th IEEE Computer Security Foundations Workshop (CSFW'04)*, 2004.
132. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
133. David Harel and Rami Marelly. *Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
134. M. Harsu. A Survey on Domain Engineering. Review, Institute of Software Systems, Tampere University of Technology, Finland, December 2002.
135. Anne Elisabeth Haxthausen, Jan Peleska, and Sebastian Kinder. A formal approach for the construction and and verification of railway control systems. *Formal Aspects of Computing*, 23:191–219, 2011.
136. Dan Haywood. *Domain-Driven Design Using Naked Objects*. The Pragmatic Bookshelf (an imprint of 'The Pragmatic Programmers, LLC.'), http://pragprog.com/, 2009.
137. Charles Anthony Richard Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985. Published electronically: *URL:* `usingcsp.com/cspbook.pdf` (2004).
138. Gerard J. Holzmann. *The SPIN Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.
139. Ted Honderich. *The Oxford Companion to Philosophy*. Oxford University Press, Walton St., Oxford OX2 6DP, England, 1995.
140. G. Huet, G. Kahn, and Ch. Paulin-Mohring. *The* Coq *Proof Assistant - A tutorial - Version 7.1*, October 2001. http://coq.inria.fr.
141. ContentGuard Inc. XrML: Extensible rights Markup Language. http://www.xrml.org, 2000.
142. ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992, 1996, 1999.
143. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-10114-9.
144. Michael A. Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. ACM Press. Addison-Wesley, Reading, England, 1995.
145. Michael A. Jackson. *Problem Frames — Analyzing and Structuring Software Development Problems*. ACM Press, Pearson Education. Addison-Wesley, England, 2001.
146. 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.
147. Ivar Jacobson, Grady Booch, and Jim Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
148. 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.
149. 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.
150. 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.
151. Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, June 2000.
152. 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.
153. Leslie Lamport. *Specifying Systems*. Addison–Wesley, Boston, Mass., USA, 2002.

154. Søren Lauesen. *Software Requirements - Styles and Techniques*. Addison-Wesley, UK, 2002.
155. Henry S. Leonard and Nelson Goodman. The Calculus of Individuals and its Uses. *Journal of Symbolic Logic*, 5:45–44, 1940.
156. 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.
157. Hans Henrik Løvengreen and Dines Bjørner. On a formal model of the tasking concepts in Ada. In *ACM SIGPLAN Ada Symp.*, Boston, 1980.
158. IPR Systems Pty Ltd. Open Digital Rights Language (ODRL). http://odrl.net, 2001.
159. E.C. Luschei. *The Logical Systems of Leśniewksi*. North Holland, Amsterdam, The Netherlands, 1962.
160. 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.
161. 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.
162. Abraham Maslow. A Theory of Human Motivation. *Psychological Review*, 50(4):370–96, 1943. http://-psychclassics.yorku.ca/Maslow/motivation.htm.
163. Abraham Maslow. *Motivation and Personality*. Harper and Row Publishers, 3rd ed., 1954.
164. John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machines, Part I. *Communications of the ACM*, 3(4):184–195, 1960.
165. John McCarthy. Towards a Mathematical Science of Computation. In C.M. Popplewell, editor, *IFIP World Congress Proceedings*, pages 21–28, 1962.
166. J. M. E. McTaggart. The Unreality of Time. *Mind*, 18(68):457–84, October 1908. New Series. See also: [187].
167. 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.
168. D.H. Mellor. Things and Causes in Spacetime. *British Journal for the Philosophy of Science*, 31:282–288, 1980.
169. 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.
170. 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.
171. 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.
172. 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.
173. C. Carroll Morgan. *Programming from Specifications*. International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK, 1990.
174. 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.
175. 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.
176. James M. Neighbors. The Draco Approach to Constructing Software from Reusable Components. *IEEE Transactions of Software Engineering*, SE-10(5), September 1984.
177. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL, A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
178. Ernst-Rüdiger Olderog and Henning Dierks. *Real-Time Systems: Formal Specification and Automatic Verification*. Cambridge University Press, UK, 2008.
179. 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.

180. 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.
181. Sam Owre, Natarajan Shankar, John M. Rushby, and David W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.
182. Christine Paulin-Mohring. Modelisation of timed automata in Coq. In N. Kobayashi and B. Pierce, editors, *Theoretical Aspects of Computer Software (TACS'2001)*, volume 2215 of *Lecture Notes in Computer Science*, pages 298–315. Springer-Verlag, 2001.
183. Christopher Peterson and Martin E.P. Seligman. *Character strengths and virtues: A handbook and classification*. Oxford University Press, 2004.
184. Shari Lawrence Pfleeger. *Software Engineering, Theory and Practice*. Prentice–Hall, 2nd edition, 2001.
185. Chia-Yi Tony Pi. *Mereology in Event Semantics*. Phd, McGill University, Montreal, Canada, August 1999.
186. K. Pohl, G. Bockle, and F. van der Linden. *Software Product Line Engineering*. Springer, Berlin, Heidelberg, New York, 2005.
187. Robin Le Poidevin and Murray MacBeath, editors. *The Philosophy of Time*. Oxford University Press, 1993.
188. Roger S. Pressman. *Software Engineering, A Practitioner's Approach*. International Edition, Computer Science Series. McGraw–Hill, 5th edition, 1981–2001.
189. Rubén Prieto-Díaz. Domain Analysis for Reusability. In *COMPSAC 87*. ACM Press, 1987.
190. Rubén Prieto-Díaz. Domain analysis: an introduction. *Software Engineering Notes*, 15(2):47–54, 1990.
191. Rubén Prieto-Díaz and Guillermo Arrango. *Domain Analysis and Software Systems Modelling*. IEEE Computer Society Press, 1991.
192. Arthur Prior. *Changes in Events and Changes in Things*, chapter in [187]. Oxford University Press, 1993.
193. Arthur N. Prior. *Logic and the Basis of Ethics*. Clarendon Press, Oxford, UK, 1949.
194. Arthur N. Prior. *Formal Logic*. Clarendon Press, Oxford, UK, 1955.
195. Arthur N. Prior. *Time and Modality*. Oxford University Press, Oxford, UK, 1957.
196. Arthur N. Prior. *Past, Present and Future*. Clarendon Press, Oxford, UK, 1967.
197. Arthur N. Prior. *Papers on Time and Tense*. Clarendon Press, Oxford, UK, 1968.
198. 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.
199. 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.
200. 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. *URL:* http://www2.imm.dtu.dk/~dibj/martin.pdf.
201. A. Quinton. Objects and Events. *Mind*, 88:197–214, 1979.
202. Wolfgang Reisig. *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*. Leitfäden der Informatik. Vieweg+Teubner, 1st edition, 15 June 2010. 248 pages; ISBN 978-3-8348-1290-2.
203. 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.
204. John C. Reynolds. *The Semantics of Programming Languages*. Cambridge University Press, 1999.
205. 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.
206. A. W. Roscoe. *Theory and Practice of Concurrency*. C.A.R. Hoare Series in Computer Science. Prentice-Hall, 1997. *URL:* http://www.comlab.ox.ac.uk/people/bill.roscoe/publications/68b.pdf.
207. Douglas T. Ross. Computer-aided design. *Commun. ACM*, 4(5):41–63, 1961.
208. 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.
209. Douglas T. Ross and J. E. Ward. Investigations in computer-aided design for numerically controlled production. Final Technical Report ESL-FR-351, , May 1968. 1 December 1959 – 3 May 1967. Electronic Systems Laboratory Electrical Engineering Department, MIT, Cambridge, Massachusetts 02139.
210. Jim Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

211. Pamela Samuelson. Digital rights management {and, or, vs.} the law. *Communications of ACM*, 46(4):41–45, Apr 2003.
212. Donald Sannella and Andrzej Tarlecki. *Foundations of Algebraic Semantcs and Formal Software Development*. Monographs in Theoretical Computer Science. Springer, Heidelberg, 2012.
213. David A. Schmidt. *Denotational Semantics: a Methodology for Language Development*. Allyn & Bacon, 1986.
214. 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.
215. John R. Searle. *Speech Act*. CUP, 1969.
216. Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
217. Barry Smith. Mereotopology: A Theory of Parts and Boundaries. *Data and Knowledge Engineering*, 20:287–303, 1996.
218. Ian Sommerville. *Software Engineering*. Pearson, 8th edition, 2006.
219. Kai Sørlander. *Det Uomgængelige – Filosofiske Deduktioner [The Inevitable – Philosophical Deductions, with a foreword by Georg Henrik von Wright]*. Munksgaard · Rosinante, 1994. 168 pages.
220. Kai Sørlander. *Under Evighedens Synsvinkel [Under the viewpoint of eternity]*. Munksgaard · Rosinante, 1997. 200 pages.
221. Kai Sørlander. *Den Endegyldige Sandhed [The Final Truth]*. Rosinante, 2002. 187 pages.
222. Kai Sørlander. *Indføring i Filosofien [Introduction to The Philosophy]*. Informations Forlag, 2016. 233 pages.
223. Diomidis Spinellis. Notable design patterns for domain specific languages. *Journal of Systems and Software*, 56(1):91–99, February 2001.
224. J.T.J. Srzednicki and Z. Stachniak, editors. *Leśniewksi's Lecture Notes in Logic*. Dordrecht, 1988.
225. Staff of Encyclopœdia Brittanica. Encyclopœdia Brittanica. Merriam Webster/Brittanica: Access over the Web: http://www.eb.com:180/, 1999.
226. 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. *URL:* `http://www2.imm.dtu.dk/~dibj/albena.pdf`.
227. Robert Tennent. *The Semantics of Programming Languages*. Prentice–Hall Intl., 1997.
228. Will Tracz. Domain-specific software architecture (DSSA) frequently asked questions (FAQ). *Software Engineering Notes*, 19(2):52–56, 1994.
229. 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.
230. 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.
231. Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.
232. 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.
233. 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.
234. George Wilson and Samuel Shpall. Action. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, summer 2012 edition, 2012.
235. G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, Cambridge, Mass., USA, 1993.
236. James Charles Paul Woodcock and James Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.
237. 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.
238. Wang Yi. *A Calculus of Real Time Systems*. PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, 1991.

239. Edward N. Zalta. The Stanford Encyclopedia of Philosophy. 2016. Principal Editor: https://plato.stanford.edu/.

240. 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.

Part VI

RSL

A

An RSL Primer

This is an ultra-short introduction to the RAISE Specification Language, RSL.

A.1 Types

The reader is kindly asked to study first the decomposition of this section into its sub-parts and sub-sub-parts.

A.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.

Basic Types:

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:

Composite Type Expressions:

[7] A-**set**
[8] A-**infset**
[9] A × B × ... × C
[10] A^{*}
[11] A^{ω}
[12] A $\xrightarrow[m]{}$ 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 $\xrightarrow[m]{}$ B), or (A^{*})-**set**, or (A-**set**)list, or (A|B) $\xrightarrow[m]{}$ (C|D|(E $\xrightarrow[m]{}$ 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.

A.1.2 Type Definitions

Concrete Types

Types can be concrete in which case the structure of the type is specified by type expressions:

Type Definition:

type
  A = Type_expr

Some schematic type definitions are:

Variety of Type Definitions:

```
[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′ • 𝒫(v) |}
```

where a form of [2–3] is provided by combining the types:

Record 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
```

Note: Values of type A, where that type is defined by A::B×C×D, can be expressed A(b,c,d) for b:B, c:D, d:D.

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 𝒫, constitute the subtype A:

Subtypes:

```
type
  A = {| b:B • 𝒫(b) |}
```

Sorts — Abstract Types

Types can be (abstract) sorts in which case their structure is not specified:

Sorts:

```
type
  A, B, ..., C
```

A.2 The RSL Predicate Calculus

A.2.1 Propositional Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values (**true** or **false** [or **chaos**]). Then:

Propositional Expressions:

false, **true**
$\mathsf{a}, \mathsf{b}, ..., \mathsf{c}\ \sim\mathsf{a}, \mathsf{a}\wedge\mathsf{b}, \mathsf{a}\vee\mathsf{b}, \mathsf{a}\Rightarrow\mathsf{b}, \mathsf{a}=\mathsf{b}, \mathsf{a}\neq\mathsf{b}$

are propositional expressions having Boolean values. $\sim$, $\wedge$, $\vee$, $\Rightarrow$, $=$ and $\neq$ are Boolean connectives (i.e., operators). They can be read as: *not*, *and*, *or*, *if then* (or *implies*), *equal* and *not equal*.

A.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:

Simple Predicate Expressions:

false, **true**
$\mathsf{a}, \mathsf{b}, ..., \mathsf{c}$
$\sim\mathsf{a}, \mathsf{a}\wedge\mathsf{b}, \mathsf{a}\vee\mathsf{b}, \mathsf{a}\Rightarrow\mathsf{b}, \mathsf{a}=\mathsf{b}, \mathsf{a}\neq\mathsf{b}$
$\mathsf{x}=\mathsf{y}, \mathsf{x}\neq\mathsf{y},$
$\mathsf{i}<\mathsf{j}, \mathsf{i}\leq\mathsf{j}, \mathsf{i}\geq\mathsf{j}, \mathsf{i}\neq\mathsf{j}, \mathsf{i}\geq\mathsf{j}, \mathsf{i}>\mathsf{j}$

are simple predicate expressions.

A.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:

Quantified Expressions:

$\forall\ \mathsf{x}{:}\mathsf{X} \bullet \mathscr{P}(x)$
$\exists\ \mathsf{y}{:}\mathsf{Y} \bullet \mathscr{Q}(y)$
$\exists\ !\ \mathsf{z}{:}\mathsf{Z} \bullet \mathscr{R}(z)$

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.

A.3 Concrete RSL Types: Values and Operations

A.3.1 Arithmetic

Arithmetic:

type
  Nat, **Int**, **Real**
value
  $+,-,*$: $\mathbf{Nat}\times\mathbf{Nat}\rightarrow\mathbf{Nat}$ | $\mathbf{Int}\times\mathbf{Int}\rightarrow\mathbf{Int}$ | $\mathbf{Real}\times\mathbf{Real}\rightarrow\mathbf{Real}$
  $/$: $\mathbf{Nat}\times\mathbf{Nat}\xrightarrow{\sim}\mathbf{Nat}$ | $\mathbf{Int}\times\mathbf{Int}\xrightarrow{\sim}\mathbf{Int}$ | $\mathbf{Real}\times\mathbf{Real}\xrightarrow{\sim}\mathbf{Real}$
  $<,\leq,=,\neq,\geq,>$ $(\mathbf{Nat}|\mathbf{Int}|\mathbf{Real}) \rightarrow (\mathbf{Nat}|\mathbf{Int}|\mathbf{Real})$

A.3.2 Set Expressions

Set Enumerations

Let the below a's denote values of type A, then the below designate simple set enumerations:

Set Enumerations:

$\{\{\},\ \{\mathsf{a}\},\ \{\mathsf{e}_1,\mathsf{e}_2,...,\mathsf{e}_n\},\ ...\} \in \mathsf{A}\textbf{-set}$
$\{\{\},\ \{\mathsf{a}\},\ \{\mathsf{e}_1,\mathsf{e}_2,...,\mathsf{e}_n\},\ ...,\ \{\mathsf{e}_1,\mathsf{e}_2,...\}\} \in \mathsf{A}\textbf{-infset}$

Set Comprehension

The expression, last line below, to the right of the $\equiv$, 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.

Set Comprehension:

type
  $\mathsf{A},\ \mathsf{B}$
  $\mathsf{P} = \mathsf{A} \to \textbf{Bool}$
  $\mathsf{Q} = \mathsf{A} \overset{\sim}{\to} \mathsf{B}$
value
  $\mathsf{comprehend}\colon \mathsf{A}\textbf{-infset} \times \mathsf{P} \times \mathsf{Q} \to \mathsf{B}\textbf{-infset}$
  $\mathsf{comprehend(s,P,Q)} \equiv \{\ \mathsf{Q(a)} \mid \mathsf{a{:}A} \bullet \mathsf{a} \in \mathsf{s} \wedge \mathsf{P(a)}\}$

A.3.3 Cartesian Expressions

Cartesian Enumerations

Let e range over values of Cartesian types involving A, B, $\ldots$, C, then the below expressions are simple Cartesian enumerations:

Cartesian Enumerations:

type
  $\mathsf{A},\ \mathsf{B},\ ...,\ \mathsf{C}$
  $\mathsf{A} \times \mathsf{B} \times ... \times \mathsf{C}$
value
  $\mathsf{(e1,e2,...,en)}$

A.3.4 List Expressions

List Enumerations

Let a range over values of type A, then the below expressions are simple list enumerations:

List Enumerations:

$\{\langle\rangle, \langle e\rangle, \ldots, \langle e1,e2,\ldots,en\rangle, \ldots\} \in A^*$
$\{\langle\rangle, \langle e\rangle, \ldots, \langle e1,e2,\ldots,en\rangle, \ldots, \langle e1,e2,\ldots,en,\ldots\rangle, \ldots\} \in A^\omega$

$\langle\ a\_i\ ..\ a\_j\ \rangle$

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.

List Comprehension:

type
  $A, B, P = A \to \mathbf{Bool}, Q = A \overset{\sim}{\to} B$
value
  $\text{comprehend}: A^\omega \times P \times Q \overset{\sim}{\to} B^\omega$
  $\text{comprehend}(l,P,Q) \equiv$
    $\langle\ Q(l(i))\ |\ i\ \mathbf{in}\ \langle 1..\mathbf{len}\ l\rangle \bullet P(l(i))\rangle$

A.3.5 Map Expressions

Map Enumerations

Let (possibly indexed) u and v range over values of type $T1$ and $T2$, respectively, then the below expressions are simple map enumerations:

Map Enumerations:

type
  $T1, T2$
  $M = T1 \underset{m}{\to} T2$
value
  $u,u1,u2,\ldots,un{:}T1,\ v,v1,v2,\ldots,vn{:}T2$
  $[\,],\ [u\mapsto v],\ \ldots,\ [u1\mapsto v1,u2\mapsto v2,\ldots,un\mapsto vn]\ \forall \in M$

Map Comprehension

The last line below expresses map comprehension:

Map Comprehension:

type
  U, V, X, Y
  $M = U \underset{m}{\to} V$
  $F = U \overset{\sim}{\to} X$
  $G = V \overset{\sim}{\to} Y$
  $P = U \to \mathbf{Bool}$
value
  $\text{comprehend}: M{\times}F{\times}G{\times}P \to (X \underset{m}{\to} Y)$
  $\text{comprehend}(m,F,G,P) \equiv$
    $[\ F(u) \mapsto G(m(u))\ |\ u{:}U \bullet u \in \mathbf{dom}\ m \wedge P(u)]$

A.3.6 Set Operations

Set Operator Signatures

Set Operations:

```
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

Set Examples:

```
examples
   a ∈ {a,b,c}
   a ∉ {}, a ∉ {b,c}
   {a,b,c} ∪ {a,b,d,e} = {a,b,c,d,e}
   ∪{{a},{a,bb},{a,d}} = {a,b,d}
   {a,b,c} ∩ {c,d,e} = {c}
   ∩{{a},{a,bb},{a,d}} = {a}
   {a,b,c} \ {c,d} = {a,bb}
   {a,bb} ⊂ {a,b,c}
   {a,b,c} ⊆ {a,b,c}
   {a,b,c} = {a,b,c}
   {a,b,c} ≠ {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):

Set Operation Definitions:

```
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 ≡
        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 ∗/
```

A.3.7 Cartesian Operations

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
```

A.3.8 List Operations

List Operator Signatures

List Operations:

value

$\textbf{hd}\text{: } A^\omega \overset{\sim}{\to} A$
$\textbf{tl}\text{: } A^\omega \overset{\sim}{\to} A^\omega$
$\textbf{len}\text{: } A^\omega \overset{\sim}{\to} \textbf{Nat}$
$\textbf{inds}\text{: } A^\omega \to \textbf{Nat-infset}$
$\textbf{elems}\text{: } A^\omega \to A\textbf{-infset}$
$.(.)\text{: } A^\omega \times \textbf{Nat} \overset{\sim}{\to} A$
$^\frown\text{: } A^* \times A^\omega \to A^\omega$
$=\text{: } A^\omega \times A^\omega \to \textbf{Bool}$
$\neq\text{: } A^\omega \times A^\omega \to \textbf{Bool}$

List Operation Examples

List Examples:

examples

$\textbf{hd}\langle a1,a2,...,am\rangle = a1$
$\textbf{tl}\langle a1,a2,...,am\rangle = \langle a2,...,am\rangle$
$\textbf{len}\langle a1,a2,...,am\rangle = m$
$\textbf{inds}\langle a1,a2,...,am\rangle = \{1,2,...,m\}$
$\textbf{elems}\langle a1,a2,...,am\rangle = \{a1,a2,...,am\}$
$\langle a1,a2,...,am\rangle(i) = ai$
$\langle a,b,c\rangle^\frown\langle a,b,d\rangle = \langle a,b,c,a,b,d\rangle$
$\langle a,b,c\rangle = \langle a,b,c\rangle$
$\langle a,b,c\rangle \neq \langle a,b,d\rangle$

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

List Operator Definitions:

value

$\text{is\_finite\_list: } A^\omega \to \textbf{Bool}$

```
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″)
```

A.3.9 Map Operations

Map Operator Signatures and Map Operation Examples

Map Operations

```
value
    m(a): M → A →̃ 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 ⇸m B) × (B ⇸m C) → (A ⇸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:

Map Operation Redefinitions:

```
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
```

A.4 λ-Calculus + Functions

A.4.1 The λ-Calculus Syntax

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), ...
```

A.4.2 Free and Bound Variables

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).

A.4.3 Substitution

In RSL, the following rules for substitution apply:

Substitution:

- **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](λ*x*•*P*)) ≡ λ y•P;
- **subst**([N/x](λ y•P)) ≡ λ*y*• **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)).

A.4.4 α-Renaming and β-Reduction

α and β Conversions:

- 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) $\equiv$ **subst**([N/x]M)

A.4.5 Function Signatures

For sorts we may want to postulate some functions:

Sorts and Function Signatures:

type
  A, B, C
value
  obs_B: A $\rightarrow$ B,
  obs_C: A $\rightarrow$ C,
  gen_A: BB$\times$C $\rightarrow$ A

A.4.6 Function Definitions

Functions can be defined explicitly:

Explicit Function Definitions:

value
  f: Arguments $\rightarrow$ Result
  f(args) $\equiv$ DValueExpr

  g: Arguments $\overset{\sim}{\rightarrow}$ Result
  g(args) $\equiv$ ValueAndStateChangeClause
  pre P(args)

Or functions can be defined implicitly:

Implicit Function Definitions:

value
  f: Arguments $\rightarrow$ Result
  f(args) **as** result
  post P1(args,result)

  g: Arguments $\overset{\sim}{\rightarrow}$ Result
  g(args) **as** result
  pre P2(args)
  post P3(args,result)

The symbol $\overset{\sim}{\rightarrow}$ 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.

A.5 Other Applicative Expressions

A.5.1 Simple let Expressions

Simple (i.e., nonrecursive) **let** expressions:

Let Expressions:

$$\textbf{let}\ \mathsf{a} = \mathscr{E}_d\ \textbf{in}\ \mathscr{E}_b(\mathsf{a})\ \textbf{end}$$

is an "expanded" form of:

$$(\lambda \mathsf{a}.\mathscr{E}_b(\mathsf{a}))(\mathscr{E}_d)$$

A.5.2 Recursive let Expressions

Recursive **let** expressions are written as:

Recursive let Expressions:

$$\textbf{let}\ \mathsf{f} = \lambda \mathsf{a{:}A} \bullet \mathsf{E(f)}\ \textbf{in}\ \mathsf{B(f,a)}\ \textbf{end}$$

is "the same" as:

$$\textbf{let}\ \mathsf{f} = \mathbf{Y}\mathsf{F}\ \textbf{in}\ \mathsf{B(f,a)}\ \textbf{end}$$

where:

$$\mathsf{F} \equiv \lambda \mathsf{g}{\bullet}\lambda \mathsf{a}{\bullet}(\mathsf{E(g)})\ \text{and}\ \mathsf{YF} = \mathsf{F(YF)}$$

A.5.3 Predicative let Expressions

Predicative **let** expressions:

Predicative let Expressions:

$$\textbf{let}\ \mathsf{a{:}A} \bullet \mathscr{P}(\mathsf{a})\ \textbf{in}\ \mathscr{B}(\mathsf{a})\ \textbf{end}$$

express the selection of a value a of type A which satisfies a predicate $\mathscr{P}(\mathsf{a})$ for evaluation in the body $\mathscr{B}(\mathsf{a})$.

A.5.4 Pattern and "Wild Card" let Expressions

Patterns and *wild cards* can be used:

Patterns:

```
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
```

A.5.5 Conditionals

Various kinds of conditional expressions are offered by RSL:

Conditionals:

```
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
```

A.5.6 Operator/Operand Expressions

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⟩ ::= !
```

A.6 Imperative Constructs

A.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.

Statements and State Change:

```
   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**.

A.6.2 Variables and Assignment

Variables and Assignment:

```
0. variable v:Type := expression
1. v := expr
```

A.6.3 Statement Sequences and skip

Sequencing is expressed using the ‘;’ operator. **skip** is the empty statement having no value or side-effect.

Statement Sequences and skip:

```
2. skip
3. stm_1;stm_2;...;stm_n
```

A.6.4 Imperative Conditionals

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
```

A.6.5 Iterative Conditionals

Iterative Conditionals:

6. **while** expr **do** stm **end**
7. **do** stmt **until** expr **end**

A.6.6 Iterative Sequencing

Iterative Sequencing:

8. **for** e **in** list_expr • P(b) **do** S(b) **end**

A.7 Process Constructs

A.7.1 Process Channels

Let A and B stand for two types of (channel) messages and i:KIdx for channel array indexes, then:

Process Channels:

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).

A.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:

Process Composition:

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.

A.7.3 Input/Output Events

Let c, k[i] and e designate channels of type A and B, then:

Input/Output Events:

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.

A.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.

Process Definitions:

```
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.

A.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:

Simple *RSL* Specifications:

```
   type
      ...
   variable
      ...
   channel
      ...
   value
      ...
   axiom
      ...
```

Part **VII**

Indexes

B

Indexes

B.1 Definitions

B.2 Concepts

B.3 Examples

B.4 Analysis Prompts

B.5 Description Prompts

B.6 Attribute Categories

B.7 RSL Symbolds

DTU Compute is a unique and internationally recognized academic environment spanning the science disciplines mathematics, statistics, computer science, and engineering.

We conduct research, teaching and innovation of high international standard - producing new knowledge and technology-based solutions to societal challenges.

We have a long-term involvement in applied and interdisciplinary research, big data and data science, artificial intelligence (AI), internet of things (IoT), smart and secure societies, smart manufacturing, and life science.

Technical
University
of Denmark

DTU Compute
Richard Petersens Plads
Building 324
DK-2800 Kgs. Lyngby
Tel +45 45 25 30 31
compute@compute.dtu.dk

www.compute.dtu.dk