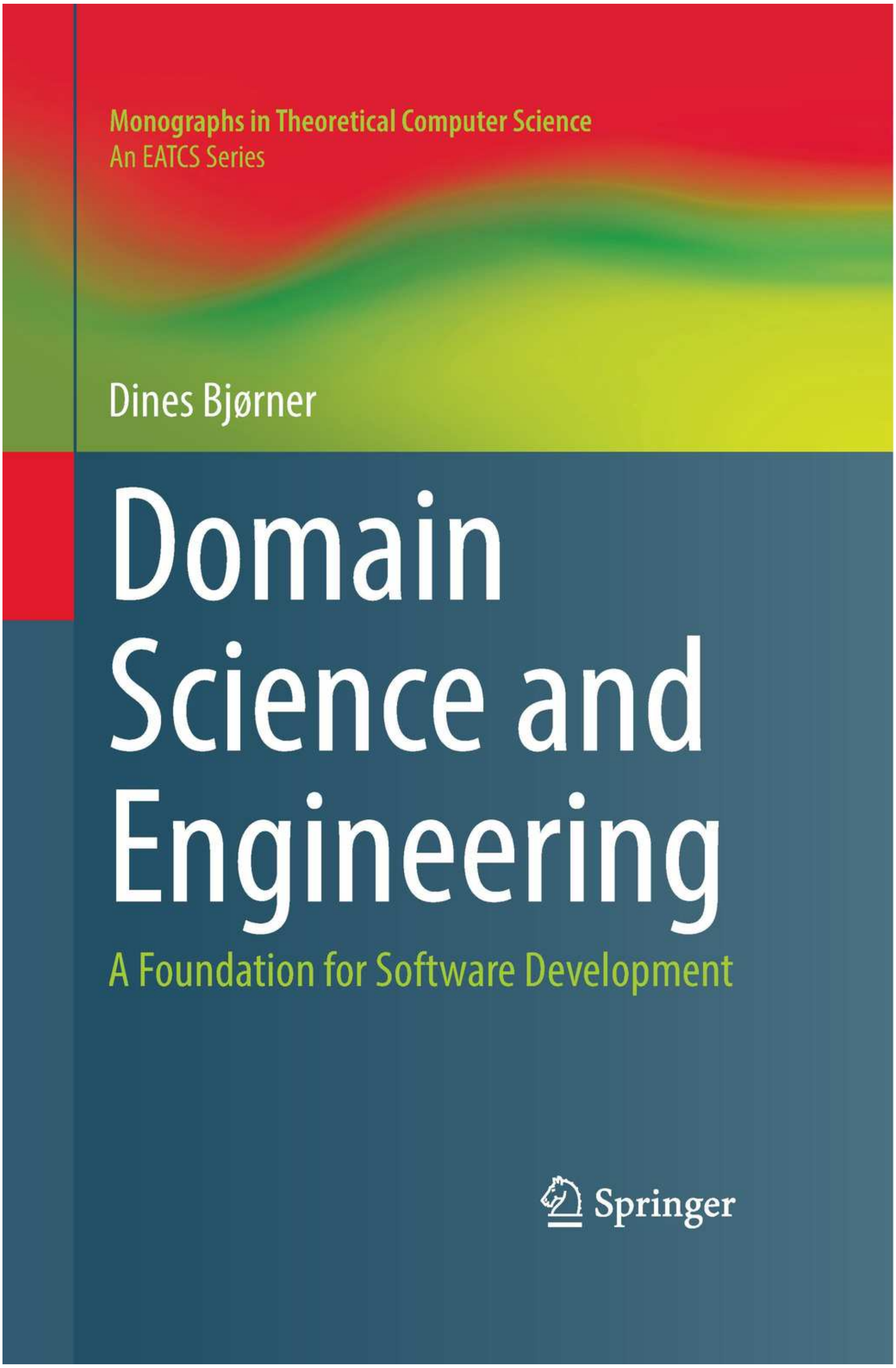


Monographs in Theoretical Computer Science
An EATCS Series

Dines Bjørner

Domain Science and Engineering

A Foundation for Software Development

Springer

A Tutorial edition of this ↑ November 2021 publication October 19, 2022: 10:18 am

TU Wien Lectures

- **Day # 1** von Neumann **Monday 24 Oct. 2022** • **Seminar & Example, I** • **10:15–11:00, 11:15–12:00**
  - **Domain Overview** 25–34
  - **Example: Road Transport** 131–149
- **Day # 2** von Neumann **Tuesday 25 Oct. 2022** • **Endurants, I** • **8:15–9:00, 9:15–10:00**
  - **External Qualities**, **Analysis** 35–50
  - **External Qualities**, **Synthesis** 51–57
- **Day # 3** von Neumann **Thursday 27 Oct. 2022** • **Endurants, II** • **8:15–9:00, 9:15–10:00**
  - **Internal Qualities**, **Unique Identifiers** 59–66
  - **Internal Qualities**, **Mereology** 67–71
- **Day # 4** von Neumann **Friday 28 Oct. 2022** • **Endurants, III** • **8:15–9:00, 9:15–10:00**
  - **Internal Qualities**, **Attributes** 72–89
- **Day # 5** von Neumann **Monday 31 Oct. 2022** • **Example, II** • **8:15–9:00, 9:15–10:00**
  - **Example: Pipelines** 151–168
- **Day # 6** von Neumann **Thursday 3 Nov. 2022** • **Perdurants, I** • **8:15–9:00, 9:15–10:00**
  - **The “Discrete Statics”** 99–107
- **Day # 7** Gödel **Friday 4 Nov. 2022** • **Perdurants, II** • **8:15–9:00, 9:15–10:00**
  - **The “Discrete Dynamics”** 107–116
  - **Summary Discussion** 117–122

  von Neumann and Gödel Lecture Rooms and 3rd floor Laboratory:
  https://www.tuwien.at/fileadmin/Assets/dienstleister/gebaeude_und_technik/FS/Plaene_2/Favoritenstrasse_9-11_1040_HA-HI_IP_09012020.pdf

Dines Bjørner

Domain Science & Engineering

A Tutorial[1]

---

[1] – first inteded for an M.Sc./Ph.D. course at the Technical University of Vienna, 24.10–4.11, 2022

Dines Bjørner

DTU Compute
Technical University of Denmark
DK-2800 Kgs.Lyngby, Denmark

Fredsvej 11, DK-2840 Holte, Denmark

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** domains.

Domains – What Are They ?

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 *solid or fluid entities*: *natural* ["God-given"] and *artefactual* ["man-made"], and its *living species entities*: *plants* and *animals* including, notably, *humans*. Examples of domains are: *rail, road, sea and air transport; water, oil and gas pipelines; industrial manufacturing; consumer, retail and wholesale markets; health care;* et cetera.

Aim and Objectives

- The **aim** of this monograph is to contribute to a methodology for analysing and describing domains.
- The **objectives** – in the sense of 'how is the aim achieved' – is reflected in the structure and contents and the didactic approach of this monograph.
- The main elements of my approach – along one concept-axis – can be itemized:
  - There is the founding of our analysis & description approach in providing a base **philosophy**, cf. Chapter 2.
  - There is the application of ideas of **taxonomy** to understand the possibly hierarchical structuring of domain phenomena respectively the understanding of properties of phenomena and relations between them.
  - There are the notions **endurants** and **perdurants** – with *endurants* being the phenomena that can be observed, or conceived and described, as a "complete thing" at no matter which given snapshot of time [116, Vol. I, pg. 656], and *perdurants* being the phenomena for which only a fragment exists if we look at or touch them at any given snapshot in time [116, Vol. II, pg. 1552].
  - There is the introduction of base elements of **calculi** for analysing and describing domains.
  - There is the application of ideas of **ontology** to understand the possibly hierarchical structuring of these calculi.
  - And finally there is the notion of **transcendental deduction**, cf. Sect. **2.1.2**, for "morphing" certain kinds of endurants into certain kinds of perdurants, Chapter 6.
- Along another conceptual-axis the below are further elements of our approach:
  - We consider domain descriptions, requirements prescriptions and software design specifications to be **mathematical** quantities.
  - And we consider them basically in the sense of **recursive function theory** [138, Hartley Rogers, 1952] and **type theory** [125, Benjamin Pierce, 1997].

Methodology

By a **method** we shall understand a set of **principles**[2] and **procedures**[3] for selecting and applying a set of **techniques**[4] and **tools**[5] to a problem in order to achieve an orderly construction of a **solution**, i.e., an **artefact**.

By **methodology** we shall understand the *study & application* of one or more methods.

By a **formal method** we shall understand a method whose decisive *principles* include that of considering its artefacts as *mathematical* quantities; whose decisive *procedures* include those of whose decisive *techniques* include those of whose decisive *tools* include those of one or more **formal languages**

By a **language** we shall here understand a set of strings of characters, i.e., sentences, sentences which are structured according to some **syntax**, i.e., **grammar**, are given meaning by some **semantics**, and are used according to some **pragmatics**.

By a **formal language** we shall here understand a languages whose *syntax* and *semantics* can both be expressed **mathematically** and for whose sentences one can **rationally reason** (*argue, prove*) **properties**.

We refer to Chapter 1 of [49] for an 8 page, approximately 50 entries set of concept definitions such as the above.

We refer to the 'Method' index, Sect. **D.3** on page 204.

•••

In this **tutorial** we shall use the formal specification language, RSL, the RAISE[6] Specification Language, RSL [84] – and we shall notably rely on RSL's adaptation of **CSP**, Tony Hoare's *Communicating Sequential Processes* [100]; and we shall propagate a definitive method for the study and description of domains.

An Emphasis

When we say *domain analysis & description* we mean that the result of such a domain analysis & description is to be a model that describes a usually infinite set of domain instances. Domains exhibit endurants and perdurants. A domain model is therefore something that defines the *nouns* (roughly speaking the endurants) and *verbs* (roughly speaking the) – and their combination – of a *language* spoken in and used in writing by the practitioners of the domain. Not an instantiation of nouns, verbs and their combination, but all possible and sensible instantiations.

---

[2] By a principle we mean: *a principle is a proposition or value that is a guide for behavior or evaluation* `[Wikipedia]`*, i.e., code of conduct*

[3] By a procedure we mean: *instructions or recipes, a set of commands that show how to achieve some result, such as to prepare or make something* `[Wikipedia]`*, i.e., an established way of doing something*

[4] By a technique we mean: *a technique, or skill, is the learned ability to perform an action with determined results with good execution often within a given amount of time, energy, or both* `[Wikipedia]`*, i.e., a way of carrying out a particular task*

[5] By a tool we mean: *a tool is an object that can extend an individual's ability to modify features of the surrounding environment* `[Wikipedia]`

[6] **RAISE: R**igorous **A**pproach[es] **i**n **S**oftware **E**ngineering, [85]

A Caveat

Experienced RSL [84] readers might observe our, perhaps cavalier (offhand), use of RSL. Perhaps, in some places, the syntax of RSL clauses is not quite right. Our non-use of RSL's module (`Scheme, Class` and `Object`) constructs force me to declare channels in the same way types, values and variables are introduced.

Dines Bjørner

October 19, 2022: 10:18 am

Contents

Chapter 1
Introduction

Contents

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** domains.

This **tutorial** is both a significantly reduced version of the scientific monograph [49] and a revision of some of its findings.

1.1 Why This Primer ?

This **tutorial** is intended as a **textbook**. The courses that I have in mind are, in the **lectures, to focus** on Chapters 3–6, i.e., Pages 25–116. The **serious students**, whether just readers or actual, physical course lecture attendants, are expected to study Chapters 1–2 as well as Chapter 7 and the Bibliography (Chapter 8) and the appendices on their own !

The **tutorial** is about how to *analyse & describe* man-made domains (including their possible interaction with nature). We emphasize the ampersand: '&'.[7] We justify competency in *Domain Science & Engineering* for two reasons. (i) For reasons of proper *engineering* software development – as indicated by the above **Triptych Dogma**. In possible proofs of software properties references are made, not only to the software code itself and the requirements, but also to the domain, the latter in the form of *assumptions about the domain*. In our mind no software development project ought be undertaken unless it more-or-less starts with a proper domain engineering phase. And (ii) for reasons of *scientifically* understanding our own everyday practical world: financial institutions, the transport industry (road, rail and air traffic, shipping), feeder systems (such as oil, gas, water and other such pipeline systems), etc.

1.2 Structure

The **tutorial**, beyond the present chapter, has, syntactically speaking, three elements:

1. **Chapter 2** covers the *philosophy* of `Kai Sørlander` [145, 146, 147, 148, 149].
   Yes, a major contribution of [49] and this **tutorial** is to justify important domain concepts by their sheer inevitability in any world description.
2. **Chapters 3–6** presents *the methodology of domain engineering*. It is split into four chapters for practical and pragmatic reasons. Chapter 3 gives a "capsule introduction" into Chapters 4–6.
3. **Chapters 7–8** and **Appendices A–D** cover such things as 'closing remarks' (**7**), a 'bibliography' (**8**), a 'Road Transport' example (**A**), a 'Pipeline System' example (**B**), an '`RSL` formal specification language' primer (**C**), and 'Indexes' to definitions, concepts, etc. (**D**).

1.3 Prerequisite Skills

The reader is expected to possess the following skills:

- To be reasonably versed in **discrete mathematics:** mathematical logic and set theory.
- To have had, even if only a fleeting, acquaintance with abstract specifications in the style of `VDM` [56, 57, 79], `Z` [158], `CafeObj` [81], `Maude` [118, 67], or the like – and thus to enjoy abstractions[8].
- To have reasonable experience with **functional programming** a la `Standard ML` or F [122, 94, 90] respectively [91] – or similar such language.
- To have reasonable experience with **`CSP`** [99, 101, 100, 139, 143].

The reader is further expected to possess the following mindset:

- To basically consider software as **mathematical objects**. That is: as quantities about which one can (and must) reason logically.
- To **think and "act" abstractly**. An essence of abstraction is expressed in the next section.

---

[7] By not writing 'and', but '&', we shall emphasize that in *A&B* we are dealing with **one** concept which consists of both *A* and *B* "tightly interacting".

[8] Some say: *"Mathematics is the Science of Abstractions"* ! Others say that both *"Mathematics and Physics are Abstractions of Reality"*.

- To **act responsibly**[9], that is to make sure that You have indeed understood Your domain, that You have indeed reasoned about adequacy of your requirements, and You have indeed model-checked, proved and formally tested your specifications.

1.4 Abstraction

> `Conception, my boy, fundamental brain-work,`
> `is what makes the difference in all art`
> *D.G. Rossetti*[10]*: letter to H. Caine*[11]

Abstraction is a tool, used by the human mind, and to be applied in the process of describing (understanding) complex phenomena.

Abstraction is the most powerful such tool available to the human intellect.

Science proceeds by simplifying reality. The first step in simplification is abstraction. Abstraction (in the context of science) means leaving out of account all those empirical data which do not fit the particular, conceptual framework within which science at the moment happens to be working.

Abstraction (in the process of specification) arises from a conscious decision to advocate certain desired objects, situations and processes as being fundamental; by exposing, in a first, or higher, level of description, their similarities and — at that level — ignoring possible differences.

[From the opening paragraphs of [98, C.A.R. Hoare *Notes on Data Structuring*]]

1.5 Software Engineering

1.5.1 Domain Science & Engineering

This **tutorial** covers only the *Domain* of software development. There are two things to say about that. One is that facets of *requirements*, essential ones, is covered in [49, Chapter 8], general ones in [21, Software Engineering, III, Part V]; the other is that the pursuit of developing domain models is not just for the sake of software development, but also for the sake of just understanding the man-made world around us. Domain science and engineering can thus be pursued in-and-by itself. Such as [the study of] most basic and theoretical physics.

1.5.2 Software Engineering

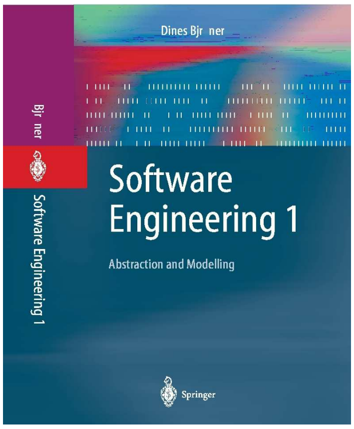
In 2006 I published these books: [19, 20, 21]:

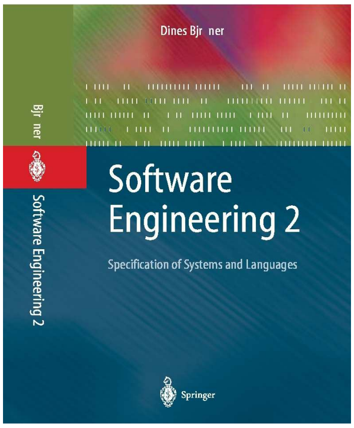
---

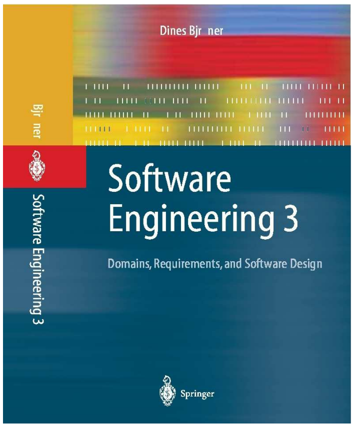
[9] In is, today, 31 July 2022, very fashionable to propagate messages of ‘ethics’ to programmers – without even touching upon issues such as *“have You understood your application domain thoroughly ?”*, or *“have You reasoned about adequacy of your requirements ?”*, or *“have You model-checked, proved and formally tested your specifications (descriptions and prescriptions) and Your code ?”*, etc.

[10] Dante Gabrielli Rosetti, 1828–1882, English poet, illustrator, painter and translator.

[11] T. Hall Caine, 1853–1931, British novelist, dramatist, short story writer, poet and critic.







1.5.2.1 Domain Engineering: 2016–2022

The first inklings of the domain science and engineering of [49] appeared in [31, 35, 2010]. More-or-less "final" ideas were published, first in [42, 2017], then in [45, March 2019]. The book [49] with updates in this **tutorial**, then constitutes the most recent status of our work in domain science & engineering.

[21, Software Engineering, III, Part V] does not cover the *Domain Engineering* material covered in [49, Chapter 8]. that latter was researched and developed between the appearance of [21] and, obviously, [49].

Part V of [21], except for Chapters 17–18 is still relevant. Chapters 17–18 of [21] are now to be replaced in any study by Chapters 4–7 of [49] **or** this **tutorial** !

1.5.2.2 Requirements Engineering

This **tutorial** does not show You how to proceed into software development according to the **Triptych Dogma**. This is strongly hinted at in [49, Chapter 9]. (That chapter is an adaptation of [23, May 2008].) Our approach to *requirements engineering* is rather different from that of both [114, A. van Laamswerde] and [108, M. A. Jackson] – to cite two relevant works. It is, I strongly think, commensurate with these works. I wish that someone could take up this line of research: making more precise, perhaps more formal, the ideas of *projection, intialisation, determination, extension* and *fitting*; and comparing, perhaps unifying our approach with that of Lamsweerde and Jackson.

1.5.2.3 Software Design

For the software design phase, after requirements engineering, we, of course, recommend [19, 20, *Software Engineering* vols. 1–2]

1.6 The Structuring of The Text

The reader will find that this text consists of "diverse" kinds of usually small paragraphs of texts: **definition**s – properly numbered and labeled; **example**s – properly numbered and labeled; **analysis predicate, function,** and **description prompt** "formalisations"; **method** principle, procedure, technique and tool paragraphs; – all of these delineated by closing ■s; – with short, usually one or two small paragraphs of introductory or otherwise explaining texts. All of this is "brought to You in living colours" ![12] So be prepared: Study such paragraphs: paragraph-by-paragraph. Each form a separate "whole".

1.7 Self-Study

This **tutorial** is primarily intended to support actual, physical lectures. For self-study by B.Sc. and M.Sc. students and practicing novice software engineers we recommend to use this **tutorial** in connection with its "origin" [49]. For self-study by Ph.D. students and graduated computer scientist we recommend going directly to the source: [49].

1.8 Two Examples

There are around 80 examples, scattered all over the first 120 pages. In addition we bring two larger examples:

- Road Transport, Appendix **A**, pages 131–149,
- Pipelines, Appendix **B**, pages 151–168.

1.9 Relation to [49]

This **tutorial** is based on [49, Nov. 2021]. Chapter 2 is a complete rewrite of [49, Chapter 2]. Chapters 4–6 is a "condensation" of [49, Chapters 4–7]: [49, Chapter 6] has been shortened and appears in this **tutorial** as Sect. **2.1.2**. From [49, Chapter 4] we have, in Chapter 4, omitted all material on – what is there referred to as *Conjoins*. And we have further sharpened the notion of *type names*. We have sharpened the focus on methods: principle, procedures, techniques and tools. You will find, in the *Indexes* section, Sect. **D.3** on page 204, a summary of references to these. Work is still in progress on highlighting more of the method steps. Section **6.5** is new.

1.10 The RAISE Specification Language, RSL, and RSL$^+$

The formal notation (to go with the informal text) of this **tutorial** is that of RSL [84], the **R**AISE **S**pecification **L**anguage, where RAISE stand for **R**igorous **A**pproach to **I**ndustrial **S**oftware **E**ngineering [85]. Other formal notations could be used instead. Replacement examples could be VDM [56, 57, 79], Z [158], or Alloy [106]. We are more using the RAISE specification language, RSL than using the method. And we are using it in two ways:

- Informally, to present and explain the domain analysis & description methods of this **tutorial**, and
- formally, to present domain descriptions.

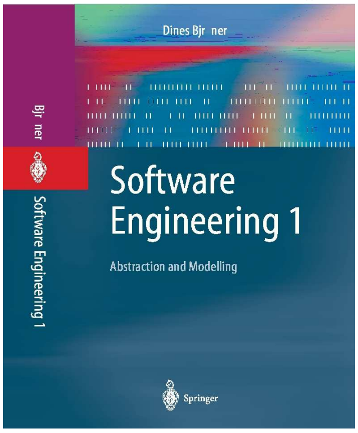
---

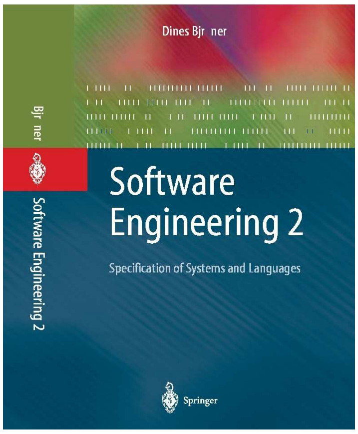
[12] – as the NBC Television Network programmmes would "proudly" announce in he 1960s !

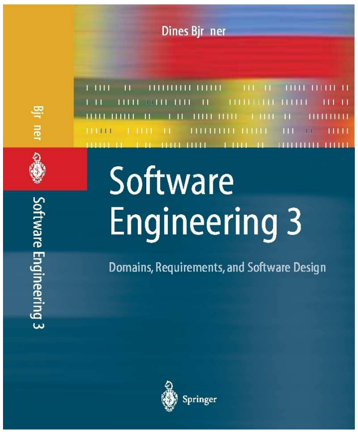
The informal RSL is an extended version, RSL$^+$.[13] The two ways are otherwise not related. One could use another specification language for either the informal or for the formal aspects.

1.11 Closing

The purpose of this introduction is to place the present **tutorial** in the context of my other books on software development and possible lectures and self study.







---

[13] See Appendix Sect. **C.12** on page 196.

Chapter 2
Kai Sørlander's Philosophy

Definition 1 . **Philosophy** [14] is the study of general and fundamental questions, such as those about existence, reason, knowledge, values, mind, and language[15] ▪

2.1 Introduction

In philosophising questions are asked. One does not necessarily get answers to these questions. Questions are examined. Light is thrown on the questions and their derivative questions.

Philosophy is man's endeavour, our quest, for uncovering the necessary characteristics of our world and our situation as humans is that world.

We shall focus on the issues of existence, i.e., metaphysics.

The treatment in this chapter is based very much on the works of the Danish philosopher **Kai Sørlander** (1944) [145, 146, 147, 148, 149, 1994–2022] both in contrast to and inspired by the German philosopher **Immanuel Kant** (1724–1804) [87].

The reason why I, as a computer scientist, is interested in philosophy, is that philosophers over more than 2500 years[16] have thought about existence: why is the world as it is – and computer scientists, like other scientists (notably physicists and economists), repeatedly model fragments of the world; and the reason why I focus on Kai Sørlander, is that his philosophy addresses issues that are crucial to our understanding how we must proceed when modelling domains – and, I think, in a way that helps us model domains with a high assurance that our models are reasonable, can withstand close scrutiny. Kai Sørlander thinks and writes logically, rationally. The area of his philosophy that I am focusing on here is metaphysics.

---

[14] From Greek: $\phi\iota\lambda o\sigma\phi\iota\alpha$, philosophia, 'love of wisdom'

[15] Many of the 'definitions' in this **tutorial** are in the style used in philosophy. They are not in the 'precise' style commonly used in mathematics and computer science. You may wish to call them **characterisation**s. In mathematics and computer science the definer usually has a formal base on which to build. In domain science & engineering we do not have a formal base, we have the "material" world of natural and man-made phenomena.

[16] – starting, one could claim, with:

2.1.1 Metaphysics

The branch of philosophy that we are focusing on is referred to as metaphysics. To explain that concept I quote from `[Wikipedia]`:

"Metaphysics is the branch of philosophy that studies the fundamental nature of reality, the first principles of being, identity and change, space and time, causality, necessity, and possibility.[17] *It includes questions about the nature of consciousness and the relationship between mind and matter, between substance and attribute, and between potentiality and actuality.*[18] *The word "metaphysics" comes from two Greek words that, together, literally mean "after or behind or among [the study of] the natural". It has been suggested that the term might have been coined by a first century CE editor who assembled various small selections of Aristotle's works into the treatise we now know by the name Metaphysics ($\mu\epsilon\tau\alpha$ $\tau\alpha$ $\phi\upsilon\sigma\iota\kappa\alpha$, meta ta physika, lit. 'after the Physics', another of Aristotle's works) [69].*

Metaphysics studies questions related to what it is for something to exist and what types of existence there are. Metaphysics seeks to answer, in an abstract and fully general manner, the questions:[19]

- *What is there ?*
- *What is it like? ?*

Topics of metaphysical investigation include existence, objects and their properties, space and time, cause and effect, and possibility. Metaphysics is considered one of the four main branches of philosophy, along with epistemology, logic, and ethics" `en.m.wikipedia.org/wiki/Metaphysics`.

- *Thales of Milet* 624–545 [everything originates from `water`] [123];
- *Anaximander* 610–546 [`‘apeiron’` (the ‘un-differentiated’, ‘the unlimited’) is the origin] [70];
- *Anaximenes* 586–526 [`air` is the basis for everything] [120];
- *Heraklit of Efesos* 540–480 [`fire is` the basis and everything in nature `is in never-ending ‘‘battle’’`] [5];
- *Empedokles* 490–430 [there are four base elements: `fire, water, air` and `soil`] [159];
- *Parminedes* 515–470 [everything that exists is `eternal and immutable` [97]];
- *Demokrit* 460–370 [all is built from `atoms`] [1];
- the Sophists: Protagoras, Gorgias (fifth and fourth centuries BC),
- *Socrates* (470–399) [2],
- *Plato* (424–347) [77],
- *Aristotle* (384–322) [6],
- etcetera.

After more than 1800 years came

- *René Descartes* (1596–1650) [74],
- *Baruch Spinoza* (1632–1677) [150],
- *John Locke* (1632–1704) [117],
- *George Berkeley* (1685–1753) [9],
- *David Hume* (1711–1776) [104],
- *Immanuel Kant* (1724–1804) [111],
- *Johan Gottlieb Fichte* (1762–1814) [109],
- *Georg Wilhelm Friedrich Hegel* (1770–1831) [95],
- *Friedrich Wilhelm Schelling* (1775–1864) [8],
- *Edmund Husserl* (1859–1938) [105],
- *Bertrand Russel* (1872–1970) [141, 154, 140, 142],
- *Ludwig Wittgenstein* (1889–1951) [156, 157],
- *Martin Heidegger* (1889–1976) [96],
- *Rudolf Karnap* (1891–1970) [128],
- *Karl Popper* (1902–1994) [128, 129],
- etcetera.

(This list is "pilfered" from [148, Pages 33–127].) [148] presents an analysis of the metaphysics of these philosophers. Except for those of Russel, Wittgenstein, Karnap and Popper, these references are just that.

[17] www.encyclopedia.com/philosophy-and-religion/philosophy/philosophy-terms-and-concepts/metaphysics

[18] Metaphysics. American Heritage Dictionary of the English Language (5th ed.). 2011.

[19] What is it (that is, whatever it is that there is) like? Hall, Ned (2012). "David Lewis's Metaphysics". In Edward N. Zalta (ed.). The Stanford Encyclopedia of Philosophy (Fall 2012 ed.). Center for the Study of Language and Information, Stanford University.

2.1.2 Transcendental Deductions

A crucial element in Kant's and Sørlander's philosophies is that of *transcendental deduction*.

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 *ontology*. For this chapter we reflect on a sub-field of epistemology, we reflect on issues of *transcendental* nature. Should you wish to follow-up on the concept of transcendentality, we refer to [87, Immanuel Kant], [103, Oxford Companion to Philosophy, pp 878–880], [4, The Cambridge Dictionary of Philosophy, pp 807–810], [64, The Blackwell Dictionary of Philosophy, pp 54–55 (1998)], and [148, Sørlander].

2.1.2.1 Some Definitions

Definition 2 . **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 3 . **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** ▪

2.1.2.2 Some Informal Examples

Example 1 . **Transcendental Deductions – Informal Examples:** 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* – such a tool is one of several forms of transcendental deductions.

The software tools, ***an automatic theorem prover*** and ***a model checker***, for example `SPIN` [102], 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*** [72, 62] reflects a transcendental deduction: firstly, 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 4 . **Transcendentality:** By **transcendentality** we shall here mean the philosophical notion: "the state or condition of being transcendental" ▪

Example 2 . **Transcendentality:** We[20] 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**[21] ▪

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.[22]

2.1.2.3 Bibliographical Note

The philosophical concept of *transcendental deduction* is is a subtle one. Arguments of transcendental nature, across the literature of philosophy, does not follow set principles and techniques. We refer to [4, *The Cambridge Dictionary of Philosophy*, pages 807–810] and [64, *The Blackwell Companion to Philosophy*, Chapter 22: Kant (David Bell), pages 589–606, Bunnin and Tsui-James, eds.] for more on 'transcendence'.

2.2 The Philosophical Question

Sørlander focuses on the philosophical question of **"what is thus necessary that it could not, under any circumstances, be otherwise ?"**.

To study and try answer that question Sørlander thinks rationally, that is, *reasons*, rather than express emotions. The German philosopher Immanuel Kant (1724–1804) suggests that our philosophising as to the philosophical question above must build on *"something which no person can consistently can deny, and thus, something that every person can rationally justify, as a consequence of be able to think at all"*. Kant then goes on to build his philosophy [111] on *the possibility of self-awareness* – something of which we all are aware. Sørlander then, in for example [148], shows that this leads to solipsism[23], i.e., to nothing.

---

[20] I first came across this example when it was presented to me by Paul Lindgreen, an early Danish computer scientist (1936–2021) – and then as a problem of data modelling [115, 1983].

[21] – in this case rather: as a fragment of a bus time table *attribute*.

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

[23] Solipsism: the view or theory that the self is all that can be known to exist.

2.3 Three Principles

2.3.1 The Possibility of Truth

Instead Sørlander suggests that **the possibility of truth** be the basis for the thinking of an answer to the highlighted question above. *The possibility of truth* is shared by all of us.

2.3.2 The Principle of Contradiction

Once we accept that *the possibility of truth* cannot be denied, we have also accepted **the principle of contradiction**, that is, that an assertion and its negation cannot both be true.

2.3.3 The Implicit Meaning Theory

We must thus also accept *the implicit meaning theory*.

Definition 5 . The Implicit Meaning Theory implies that there is a *mutual relationship* between the (α) *meaning of designations* and (β) *consistency relations between assertions* ■

As an example of what "goes into" the *implicit meaning theory*, we bring, albeit from the world of computer science, that of the description of the **stack** data type (its endurant data types and perdurant operations).

Example 3 . The Implicit Meaning Theory.: Narrative:

α The Designations:

1 Stacks, s:S, have elements, e:E;
2 the empty_S operation takes no arguments and yields a result stack;
3 the is_empty_S operation takes an argument stack and yields a Boolean value result.
4 the stack operation takes two arguments: an element and a stack and yields a result stack.
5 the unstack operation takes an non-empty argument stack and yields a stack result.
6 the top operation takes an non-empty argument stack and yields an element result.

β The Consistency Relations:

7 an empty_S stack is_empty, and a stack with at least one element is not;
8 unstacking an argument stack, stack(e,s), results in the stack s; and
9 inquiring the top of a non-empty argument stack, stack(e,s), yields e.

Formalisation.

The designations:

type
1. E, S
value
2. empty_S: **Unit** $\rightarrow$ S
3. is_empty_S: S $\rightarrow$ **Bool**
4. stack: E $\times$ S $\rightarrow$ S
5. unstack: S $\overset{\sim}{\rightarrow}$ S
6. top: S $\overset{\sim}{\rightarrow}$ E

The consistency relations:

axiom
7. is_empty(empty_S()) = **true**
7. is_empty(stack(e,s)) = **false**
8. unstack(stack(e,s)) = s
9. top(stack(e,s)) = e ■

2.3.4 A Domain Analysis & Description Core

The three concepts: (i) *the possibility of truth*, (ii) *the principle of contradiction* and (iii) *the implicit meaning theory* thus form the core – and imply that (a) *the indispensably necessary characteristics of any possible world, i.e., domain*, are *equivalent* with (b) *the similarly indispensably necessary conditions for any possible domain description*.

2.4 The Deductions

2.4.1 Assertions

Definition 6 . **Assertion:** An assertion is a declaration, an utterance, that something is the case ▪

Assertions may typically be either propositions or predicates.

2.4.2 The Logical Connectives

Any domain description must necessarily contain assertions. Assertions are expressed in terms of negation, $\sim$, conjunction, $\wedge$, disjunction, $\vee$, and implication, $\Rightarrow$.

2.4.2.1 $\sim$: Negation

Negation is defined by the principle of contradiction. If an assertion, a, holds, then its negation, $\sim a$, does not hold.

2.4.2.2 Simple Assertions

Simple assertions, i.e., propositions, are formed from assertions, f.x. a,b, by means of the logical connectives.

2.4.2.3 $\wedge$: Conjunction

The simple assertion $a \wedge b$ holds if both a and b holds.

2.4.2.4 $\vee$: Disjunction

The simple assertion $a \vee b$ holds if either or both a and b holds.

2.4.2.5 $\Rightarrow$: Implication

The simple assertion $a \Rightarrow b$ holds if a is *inconsistent* with the negation of b.

2.4.2.6 Model Theory Explication of The Logical Connectives

A model theory explication of the binary logical connectives is given on Page 178.

2.4.3 Modalities

2.4.3.1 Necessity

Definition 7 . Necessity: An assertion is *necessarily true* if its truth ("true") follows from the definition of the designations by means of which it is expressed. Such an assertion holds under all circumstances ▪

Example 4 . Necessity: *"It may rain someday"* is necessarily true.

2.4.3.2 Possibility

Definition 8 . Possibility: An assertion is *possibly true* if its negation is not *necessarily true* ▪

Example 5 . Possibility: *"it will rain tomorrow"* is possibly true.

2.4.4 Empirical Assertions

Definition 9 . Empirical Knowledge: In philosophy, knowledge gained from experience rather than from innate ideas or deductive reasoning is empirical knowledge. In the sciences, knowledge gained from experiment and observation rather than from theory is empirical knowledge ▪

Example 6 . Expressing Empirical Knowledge: There are innumerable ways of expressing empirical knowledge.

a. There are two automobiles in that garage.[24]
b. The two automobiles in that garage are distinct.[25]
c. The two automobiles in that garage are parked next to one another.[26]
d. That automobile, the one to the left, in that garage is [painted] red.[27]
e. The automobile to the right in that garage has just returned from a drive.[28]

---

[24] *The automobiles are solid* endurants*, and so is the garage, that is, they are both* parts.

[25] *Their distinctness gives rise to their respective, distinct, i.e.,* unique identifiers.

[26] *The topological ordering of the two automobiles is an example of their* mereology.

[27] *The red colour of the automobile is an* attribute *of that automobile.*

[28] *The fact that that automobile, to the right in the garage, has just returned from a drive, is a* possibly time-stamped attribute *of that automobile.*

f. The automobile, with Danish registration number `AB 12345`, is currently driving on the `Copenhagen area` city `Holte` road `Fredsvej` at position `'top of the hill'`.[29]
g. The automobile on the roof of that garage is pink.

The pronoun 'that' shall be taken to mean that someone gestures at, points out, the garage in question. If there is no such garage then the assertion denotes the **chaos** value ! Statements (a.–g.) are assertions. The assertions contain *references* to quantities "outside the assertions" — 'outside' in the sense that they are not defined in the assertions. Assertion (g.) does not make sense, i.e., yields **chaos**. The term 'roof' has not been defined ▪

> $\mathcal{I}$: ***The Object Language.*** *The language used in the above assertions is quite 'free-wheeling'. The language to be used in "our" domain descriptions is, i.e., will be, more rigid* ▪

Definition 10 . **Empirical Assertion:** The domain description language of assertions, contain **references**, i.e., *designators*, and **operators**. All of these shall be properly defined in terms of names of *endurants* and their *unique identifiers*, *mereologies* and *attributes*; and in terms of their *perdurant "counterparts"* ▪

•••

From Possible Predicates to Conceptual Logic Description Framework. The ability to deduce which type of predicates that a phenomena of any domain can be ascribed is thus equivalent to deducing the conceptual logical conditions for every possibly possible domain description.

•••

By a so-called *transcendental deduction* we have shown that simple empirical assertions consist of a **subject** which **refers** to an independently existing entity and a **predicate** which ascribes a **property** to the referred entity [148, π 146 ℓ 1–5].

The world, or as we shall put it, the domains, that we shall be concerned with, are *what can be described in simple assertions*, then any possible such world, i.e., domain must *primarily consist of such entities* [148, π 146 ℓ 5–7].

We shall therefore, in the following, explicate a system of **concepts** by means of which the entities, that may be referred to in simple assertions, can be described [148, π 146 ℓ 8–11].

> $\mathcal{I}$: *These* ***concepts*** *are those of* entities, endurants, perdurants, unique identity, mereology *and* attributes. ▪

2.4.5 Identity and Difference

We can now assume that the world consists of an indefinite number of entities: Different empirical assertions may refer to distinct entities. Most immediately we can define two interconnected concepts: **identity** and **diversity**.

---

[29] *The automobile in question is now a* perdurant *having a so-called* time-stamped progammable event attribute *of the Copenhagen area city of Holte, "top of the hill".*

2.4.5.1 Identity

Definition 11 . **Identical:** "An entity referred to by the name A is *identical* to an entity referred to by the name B if A cannot be ascribed a property which is incommensurable with a property ascribed to B" [148, π 146 ℓ 14-23] ▪

2.4.5.2 Difference

Definition 12 . **Different:** "A and B are *distinct*, differs from one another, if the can be ascribed incommensurable properties." [148, π 146 ℓ 23-26] ▪

•••

"These formal definitions, by transcendental deduction, introduces the concepts of of **identity** and **difference**. "They can thus be assumed in any transcendental deduction of a domain description which, in principle, must be expressed in any possible language". [148, π 147 ℓ 1-5]

Definition 13 . **Unique Identification:** By a *transcendental deduction* we introduce the concept of manifest, physical entities each being uniquely identified ▪

We make no assumptions about any representation of unique identifiers.

2.4.6 Relations

2.4.6.1 Identity and Difference

Definition 14 . **Relation:** "Implicitly", from the two concepts of *identity* and *difference*, follows the concept of **relation**s. "A identical to B is a relational assertion. So is A different from B" [148, π 147 ℓ 6-10] ▪

2.4.6.2 Symmetry

Definition 15 . **Symmetry:** If A is identical to B then B must be identical to A. This expresses that the *identical to* relation is *symmetric*. And, If A is different from B then B must be different from A. This expresses that the *different from* relation is also *symmetric* ▪

2.4.6.3 Asymmetry

Definition 16 . **Asymmetry:** A relation which holds between A and B but does not hold between B and A is *asymmetric* [148, π 147 ℓ 25–27] ▪

2.4.6.4 Transitivity

Definition 17 . **Transitivity:** "If A is identical to B and if B is identical to C then A must be identical to C. So the relation *identical to* is *transitive*" [148, π 147-148 ℓ 28-30,1-4] ▪

The relation *different from* is not transitive.

2.4.6.5 Intransitivity

Definition 18 . In-transitivity: If, on the other hand, we can logically deduce that a relation, $\mathcal{R}$ holds' from A to B and the same relation, $\mathcal{R}$, holds from B to C but $\mathcal{R}$ does not hold from A to C then relation $\mathcal{R}$ is *intransitive* [148, π 148 ℓ 9–12] ▪

2.4.7 Sets, Quantifiers and Numbers

2.4.7.1 Sets

The possibility now exists that two or more entities may be prescribed the same property.

Definition 19 . Sets: The "same properties" could, for example, be that two or more uniquely distinguished entities, $x,y,...,z$, have [at least] one attribute kind (type) and value, (t,v), in common. This means that (t,v) distinguishes a set $s_{(s,v)}$ – by a *transcendental deduction*. A fact, just t likewise distinguishes a possibly other, most likely "larger", set s_t ▪

From the transcendentally deduced notion of set follows the relations: equality, $=$, inequality, $\neq$, proper subset, $\subset$, subset, $\subseteq$, set membership, $\in$, set intersection, $\cap$, set union, $\cup$, set subtraction, $\setminus$, set cardinality, **card**, etc. !

2.4.7.2 Quantifiers

By a further *transcendental deduction* we can place the *quantifiers* among the concepts that are necessary in order to describe domains.

Definition 20 . The Universal Quantifier: The universal quantifier expresses that all members, x, of a set, s, possess a certain $\mathcal{P}$roperty: $\forall x : S \bullet \mathcal{P}(x)$ ▪

Definition 21 . The Existential Quantifier: The existential quantifier expresses that at least one member, x, of a set, s, possess a certain $\mathcal{P}$roperty: $\exists x : S \bullet \mathcal{P}(x)$ ▪

2.4.7.3 Numbers

Numbers can, again by *transcendental deduction*, be introduced, not as observable phenomena, but as a rational, logic consequence of sets.

Definition 22 . Numbers: Numbers can be motivated, for example, as follows:

- Start with an empty set, say $\{\}$. It can be said to represent the number zero.[30]
- Then add the empty set $\{\}$ to $\{\}$ and You get $\{\{\}\}$ said to represent 1.
- Continue with adding $\{\}$ to $\{\{\}\}$ and You get $\{\{\},\{\{\}\}\}$, said to represent 2.
- And so forth – ad infinitum ▪

In this way one[31] can define the natural numbers. We could also do it by just postulating distinct entities which are then added, one by one to a an initially empty set [148, π 150 ℓ 8-13].

We can then, still in the realm of philosophy, proceed with the introduction of the arithmetic operations designated by addition, **+**, subtraction, , multiplication, $*$, division, $\div$, equality, **=**,

---

[30] Which, in the decimal notation is written as 0.

[31] https://en.wikipedia.org/wiki/Set-theoretic_definition_of_natural_numbers

inequality, ≠, larger than, >, larger than or equal, ≥, smaller than, <, smaller than or equal, ≤, etcetera !

From explaining numbers on a purely philosophical basis one can now proceed mathematically into the realm of *number theory* [92].

2.4.8 Primary Entities

We now examine the concept of *primary objects*.

The next two definitions, in a sense, "fall outside" the line of the present philosophical inquiry. They will be "corrected" to then "fall inside" our inquiry.

Definition 23 . **Object:** By an *object* we, in our context, mean something material that may be perceived by the senses[32] ▪

Definition 24 . **Primary Object:** By a *primary object* we[33] mean an object that exists as its own *entity* independent[34] of other objects ▪

In the last definition we have used the term *entity*. That term, 'entity', will be used henceforth instead of the term 'object'.

We have deduced the relations *identity, difference, symmetry, asymmetry, transitivity* and *intransitivity* in Sects. **2.4.5–2.4.6**. You may ask: *for what purpose ?* And our answer is: *to justify the next set of deductions.* First we reason that there is the possibility of there being many entities. We argue that that is possible due to there being the relation of asymmetry. If it holds between two entities then they must necessarily be ascribed different predicates, hence be distinct.

Similarly we can argue that two entities, B and C which both are asymmetric wrt. to an entity A may stand in a symmetric relation to one another. This opens for the *possibility* that every pair of distinct entities may stand in a pair of mutual relations. First the asymmetry relation that expresses their distinctness. Secondly, the possibility of a symmetry relation which expresses the two entities individually with respect to one-another. *The above forms a transcendental basis for how two or more [primary] entities must necessarily be characterised by predicates.*

2.4.9 Space and Time

The asymmetry and symmetry relations between entities cannot be *necessary* characteristics of every possibly reality if they cannot also posses an *unavoidable rôle* in our own concrete reality. Next we examine two such *unavoidable rôles*.

2.4.9.1 Space

One pair of such rôles are *distance* and *direction*. *Distance* is a relation that holds between any pair of distinct entities. It is a symmetric relation. *Direction* is an asymmetric relation that also holds between pair of distinct entities. Hence we conclude that **space** is an unavoidable

---

[32] www.merriam-webster.com/dictionary/object

[33] help.hcltechsw.com/commerce/8.0.0/tutorials/tutorial/ttf_cmcdefineprimaryobject.html

[34] Yes, we know: we have not defined what is meant by 'as its own' and 'independent' !

characteristics of every possibly reality. Hence we conclude that entities exist in space. They must "fill" some space, have *extension*, they must *fill* some space, have *surface* and *form*. From this we can define the notions of spatial point, spatial straight line, spatial surface, etcetera. Thus we can philosophically motivate geometry.

2.4.9.2 Time

Primary empirical entities may be accrue predicates that it is not logically necessary that they accrue. That is, it is logically possible that primary entities accrue predicates that they actually accrue. How is it possible that one and the same primary entity may accrue incommensurable predicates ?

That is only possible if one and the same primary entity can exist in *different states*. It may exist in one state in which it accrue a certain predicate. And it may exist in another state in which it accrue a therefrom incommensurable predicate.

What can we say about these states ? First that these states accrue different, incommensurable predicates. How can we assure that ! Only if the states stand in a asymmetric relation to one another. From this we can conclude that primary entities necessarily may exist in a number of states each of which stand in an asymmetric relation to their predecessor state. So these states also stand in a *transitive* relation.

This is a necessary characteristics of any possible world. So it is also a characteristics of our world. That relation is **time**. It possesses the *before*, *after*, *in-between*, and other [temporal] relations. We have thus deduced that every possible world must "occur in time" and that primary entities may exist in before or after states.

From the above we can derive a whole algebra of temporal types and operations, for example:

- TIME and TIME INTERVAL types;
- addition of TIME and TIME INTERVAL to obtain TIME;
- addition of TIME INTERVALs to obtain TIME INTERVALs;
- subtraction of two TIMEs to become TIME INTERVALs; and
- subtraction of of TIME INTERVALs to obtain TIME INTERVAL.

2.4.10 The Causality Principle

But what is it that *cause* primary entities to undergo *state changes* ? Assertions about how a primary entity is at different times, such assertions must necessarily be logically independent. That follows from primary entities necessarily must accrue incommensurable predicates at different times. It is therefore logically impossible to conclude from how a primary entity is at one time to how it is at another time. How, therefore, can assertions about a primary entity at different times be about the same entity ?

We can therefore transcendentally deduce that there must be a *special implication-relationship* between assertions about how a primary entity is at different times. Such a *special implication-relationship* must depend on the *empirical circumstances* under which the primary entity exists. That is, we must deduce the conditions under which it is, at all, possible to consistently make statements about primary entities going from one state in which it accrues a specific predicate to another state in which it accrues a therefrom incommensurable predicate. There must be something in the empirical circumstances which implicates the state transition. If the the empirical circumstances are *stable* then Thebes is nothing in these circumstances that imply entity changes. If the primary entity changes, then that assumes that there must have been a prior change in the circumstances – with those changes having that consequence. . . . [35] We name such a change of the circumstances *a cause*. And we conclude that every change of a primary entity must have a cause. We also conclude that *equivalent cause* imply *equivalent effects*.

This form of implication is called the *causality principle*. It assumes logical implication. But it cannot be reduced to logical implication. It is logically necessary that every primary entity – and therefore every possible world – is subject to the *causality principle*. In this way Kai Sørlander transcendentally deduce the principle of causality. Every change has a cause. The same cause under the same circumstances lead to same effects.

2.4.11 Newton's Laws

Sørlander then shows how Newton's laws can be deduced. These laws, in summary, are:

- **Newton's First Law:** An entity is at rest or moving at a constant speed in a straight line, it will remain at rest or keep moving in a straight line at constant speed unless it is acted upon by a force.
- **Newton's Second Law:** When an entity is acted upon by a force, the time rate of change of its momentum equals the force.
- **Newton's Third Law:** To every action there is always opposed an equal reaction; or, the mutual actions of two bodies upon each other are always equal, and directed to contrary entities.

2.4.11.1 Kinematics

Above we have deduced that primary entities are in both space and time. They have *extent* in both space and time. That means that they must change with respect to their spatial properties: place and form. The change in place is the fundamental. A primary entity which changes place is said to be in *movement*. A primary entity in movement must follow a certain geometric route. It must move a certain length of route in a certain interval of time, i.e., have a *velocity:* speed and direction. A primary entity which changes velocity has an *acceleration*. That is, we have deduced he basics of *kinematics*.

2.4.11.2 Dynamics

When we to the above add that primary entities are in time, then they are subject to causality. That means that we are entering the doctrine of the influence of *force*s on primary entities. That is, *dynamics*. Kinematics imply that an entity changes if it goes from being at rest to move, or if it goes from moving to being at rest. An entity also changes if it goes from moving at one velocity to moving at a different velocity. We introduce the notion of *momentum*. An entity has same momentum if at two times it has the same velocity and acceleration.

2.4.11.3 Newton's First Law

When we combine kinematics with causality then we can deduce that if an entity changes momentum then there must be a cause in the circumstances which causally implies the

---

[35] We skip some of Sørlander's reasoning, [148, Page 162, lines 1–12]

change. We call that cause a *force*. The force must be proportional to the change in momentum. This implies that an entity which is not subject to an external force remains in the same momentum. This is **The Law of Inertia, Newtons First Law.**

2.4.11.4 Newton's Second Law

That a certain force is necessary in order to change an entity's momentum must imply that such an entity must provide a certain *resistance* against change of momentum. It must have a *mass*. From this it follows that the change of an entity's momentum not only must be proportional to the applied force but also inversely proportional to that entity's mass. This is **Newtons Second Law.**

2.4.11.5 Newton's Third Law

Where do the forces that influence the momentum of entities come from ? It must, it can only, be from primary entities. Primary entities must be the source of the forces that influence other entities. Here we shall argue one such reason. The next section, on universal gravitation, presents a second reason.

Primary entities may be in one an other's way. Hence they may eventually collide. If a primary entity has a certain velocity it may collide with another primary entity crossing its way. In the mutual collision the two entities influence one another such that they change momentum. They influence each other with forces. Since none of the two entities have any special position, i.e., rank, the forces by means of which they affect one another must be equal and oppositely directed. This is **Newtons Third Law.**

2.4.12 Universal Gravitation

But[36], really, how can primary entities be the source of forces that affects one another ? We must dig deeper ! How can primary entities have mass such that it requires force to change their momentum ? Our answer is that the reason they have mass must be due to mutual influence between the primary objects themselves. It must be an influence which is oppositely directed to that which they expose on one another when they collide. Because this, in principle, applies to all primary entities, these must be characterised by a mutual universal attraction. And that is what we call *universal gravitation*. That concept has profound implications.

•••

We shall not go into details here but just, casually, as it were, mention that such concepts as speed limit, elementary particles and Einstein's theories are "more-or-less" transcendentally deduced !

---

[36] This section is from [148, Pages 168–173]

2.4.13 Purpose, Life and Evolution

We shall briefly summarise Sørlander's analysis and deductions with respect to the concepts of *living species: plants* and *animals*, the latter including *humans*.

Up till now Sørlander's analyses and deductions have focused on the physical world, "culminating" in Newton's Laws and Einstein's theories.

If[37] there is to be language and meaning then, as a first condition, there must be the possibility that there are primary entities which are not locked-in "only" in that physical world deduced till now. This is only possible if such primary entities are additionally subject to a *purpose-causality*, one that is so constructed as to *strive* to *maintain* its own *existence*. We shall refer to this kind of primary entities as *living species*.

2.4.13.1 Living Species

As living species they must be subject to all the physical conditions for existence and mutual influence. Additionally they must have a form which they are *causally determined to reach and maintain*. This development and maintenance must take place in a *substance exchange* with its surroundings. Living species need these substances in order to develop and maintain their form.

It must furthermore be possible to distinguish between two forms of living species: (i) one form which is characterised only by *development, form and substance exchange*; and (ii) another form which, additional to (i), is characterised by *being able to move*. The first form we call *plants*. The second form we call *animals*.

2.4.13.2 Animals

For animals to move they must (i) possess *sense organs*, (ii) *organs of movement* and (iii) *instincts, incentives,* or *feelings*. All this still subject to the physical laws and to satisfy motion.

This is only possible if animals are **not** built (like the elementary particles of physics) but by special physical units. These cells must satisfy the *purpose-causality* of animals. And we know, now, from the *biological sciences* that something like that is indeed the case. Indeed animals are built from cells all of which possess *genomes* for the whole animal and, for each such cell, a proper fraction of its genome controls whether it is part of a sensory organ, or a nerve, or a motion organ, or a more specific function. Thus it has transcendentally been deduced that such must be the case and biology has confirmed this.

2.4.13.2.1 Humans

We briefly summarise[38], in six steps, (i–vi), Sørlander's reasoning that leads from animals, in general, see above, to humans, in particular.

(i) First the concept of ***level of consciousness*** is introduced. On the basis of animals being able to *learn* from *experience* the concept of *consciousness level* is introduced. It is argued that *neurons* provide part of the basis for *learning* and the *consciousness level*.

(ii) Secondly the concept of ***social instincts*** is introduced. For animals to interact social instincts are required.

---

[37] We now treat the material of [148, *Chapter 10, Pages 174–179*].

[38] [148, *Chapter 11, Pages 180–183*]

(iii) Thirdly the concept of ***sign language*** is introduced. In order for animals to interact some such animals, notably the humans, develop a sign language.

(iv) Fourthly the concept of ***language*** is introduced. The animals that we call *humans* finally develop their sign language into a language that can be spoken, heard and understood. Such a language, regardless of where it is developed, that is, regardless of which language it is, must assume, i.e., build on the same set of basic concepts as had been uncovered so far in our deductions of what must necessarily be in any description of any world.

We continue summarise[39] Sørlander's reasoning that leads from generalities about humans to humans with knowledge and responsibility.

(v) Fifthly the concept of ***knowledge*** is introduced. An animal which is *conscious* must *sense* and must react to what it senses. To do so it must have *incentives* as causal conditions for its specific such actions. If the animal has, possess, language, then it must be able to express that and what it senses and that it acts accordingly, and why it does so. It must be able to express that it can express this. That is, that what it expresses, is true. To express such assertions, with sufficient reasons for why they are true, is equivalent to *knowing* that they are true. Such animals, as possess the above "skills", become persons, humans.

(vi) Sixthly the concept of ***responsibility*** is introduced. Humans conscious of their concrete situation, must also know that these situations change. They are conscious of earlier situations. Hence they have *memory*. So that can formulate *experience* with respect to the *consequences* of their actions. Thus humans are (also) characterised by being able to understand the consequences of future actions. A person who considers how he ought act, can also be ascribed *responsibility* – and can be judged *morally*.

•••

This ends our eposé of Sørlander's metaphysics wrt. living species. That is, we shall not cover neither non-human animals, nor plants.

2.5 Philosophy, Science and the Arts

We quote extensively from [146, Kai Sørlander, 1997].

[146, pp 178] "Philosophy, science and the arts are products of the human mind."
[146, pp 179] "Philosophy, science and the arts each have their own goals."

- **Philosophers** seek to find the inescapable characteristics of any world.
- **Scientists** seek to determine how the world actually and our situation in that world.
- **Artists** seek to create objects for experience.

We shall elaborate. [146, pp 180] "Simplifying, but not without an element of truth, we can relate the three concepts by the **modalities:**"

- ***philosophy*** is the **necessary**,
- ***science*** is the **real**, and
- ***art*** is the **possible**.

... Here we have, then, a distinction between philosophy and science. ... From [145] we can conclude the following about the results of philosophy and science. These results must be consistent [with one another]. This is a necessary condition for their being *correct*. ... ... The **real** must be a *concrete realisation* of the **necessary**.

---

[39] [148, *Chapter 12, Pages 184–187*]

2.6 A Word of Caution

The present chapter represents an attempt to give an English interpretation of Kai Sørlander's Philosophy. I will "mull" over this interpretation for a while. Then I will present it to Kai Sørlander for his comments. We shall see.

Chapter 3
Domains

Contents

This chapter is informal. Here we introduce You to important main concepts of domains. Subsequent chapters will be more technical. They will define most of the domain concepts of this chapter properly.

3.1 Domain Definition

Definition 25 . **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 *more-or-less related solid or fluid entities*: *natural* ["God-given"] and *artefactual* ["man-made"], and its *living species entities*: *plants* and *animals* – including, notably, *humans* ▪

Example 7 . **Domains:** A few, more-or-less self-explanatory examples:

- **Rivers –** with their natural sources, deltas, tributaries, waterfalls, etc., and their man-made dams, harbours, locks, etc. [50]
- **Road nets –** with street segments and intersections, traffic lights, and automobiles.
- **Pipelines –** with their wells, pipes, valves, pumps, forks, joins and wells [36].
- **Container terminals –** with their container vessels, containers, cranes, trucks, etc. [44] ▪

The definition relies on the understanding of the terms *'rationally describable'*, *'discrete dynamics'*, *'human assisted'*, *'solid'* and *'fluid'*. The last two will be explained later. By ***rationally describable*** we mean that what is described can be understood, including reasoned about, in a rational, that is, logical manner. By ***discrete dynamics*** we imply that we shall basically rule out such domain phenomena which have properties which are continuous with respect to their time-wise, i.e., dynamic, behaviour. By ***human-assisted*** we mean that the domains – that we are interested in modelling – have, as an important property, that they possess man-made entities.

This primer presents a *method*, its *principles*, *procedures*, *techniques* and *tools*, for *analysing* &[40] *describing* domains.

3.2 Phenomena and Entities

Definition 26 . **Phenomena** By a *phenomenon* we shall understand a fact that is observed to exist or happen ▪

Some phenomena are rationally describable – to a large or full degree – others are not.

Definition 27 . **Entities** By an *entity* we shall understand a more-or-less rationally describable phenomenon ▪

Example 8 . **Phenomena and Entities:** Some, but not necessarily all aspects of a river can be rationally described, hence can be still be considered entities. Similarly, many aspects of a road net can be rationally described, hence will be considered entities ▪

3.3 Endurants and Perdurants

3.3.1 Endurants

Definition 28 . **Endurants** those quantities of domains that we can observe (see and touch), in *space*, as "complete" entities at no matter which point in *time* – "material" entities that persists, endures ▪

---

[40] We use here the ampersand, '&', as in $A\&B$, to emphasize that we are treating A and B as one concept.

Example 9 . **Endurants:** a street segment [link], a street intersection [hub], an automobile ▪

Domain endurants, when eventually modelled in software, typically become data. Hence the careful analysis of domain endurants is a prerequisite for subsequent careful conception and analyses of data structures for software, including data bases.

3.3.2 Perdurants

Definition 29 . **Perdurants** those quantities of domains for which only a fragment exists, in *space*, if we look at or touch them at any given snapshot in *time* ▪

Example 10 . **Perdurant:** a moving automobile ▪

Domain perdurants, when eventually modelled in software, typically become processes. Hence the careful analysis of domain perdurants is a prerequisite for subsequent careful conception and analyses of functions (procedures).

3.4 External and Internal Endurant Qualities

3.4.1 External Qualities

Definition 30 . **External qualities:** of endurants of a manifest domain are, in a simplifying sense, those we can sea, touch and have spatial extent. They, so to speak, take form.

Example 11 . **External Qualities:** The Cartesian[41] of sets of solid atomic street intersections, and of sets of solid atomic street segments, and of sets of solid automobiles of a road transport system where the Cartesian, sets, atomic, and solid reflect external qualities ▪

3.4.1.1 Discrete or Solid Endurants

Definition 31 . **Discrete or Solid Endurants:** By a *solid* [or *discrete*] endurant we shall understand an endurant which is separate, individual or distinct in form or concept, or, rephrasing: have 'body' [or magnitude] of three-dimensions: length, breadth and depth [116, Vol. II, pg. 2046] ▪

Example 12 . **Solid Endurants:** The wells, pipes, valves, pumps, forks, joins and sinks of pipelines are solids. [These units may, however, and usually will, contain fluids, e.g., oil, gas or water] ▪

We shall mostly be analysing and describing solid endurants.

As we shall see, in the next chapter, we analyse and describe solid endurants as either parts or living species: animals and humans. We shall mostly be concerned with parts. That is, we shall just, as: "in passing", for sake of completeness, mention living species !

---

[41] Cartesian after the French philosopher, mathematician, scientist René de Descartes (1596–1650)

3.4.1.2 Fluids

Definition 32 . **Fluid Endurants:** By a *fluid endurant* we shall understand an endurant which is prolonged, without interruption, in an unbroken series or pattern; or, rephrasing: a substance (liquid, gas or plasma) having the property of flowing, consisting of particles that move among themselves [116, Vol. I, pg. 774] ∎

Example 13 . **Fluid Endurants:** water, oil, gas, compressed air, smoke ∎

Fluids are otherwise liquid, or gaseous, or plasmatic, or granular[42], or plant products, i.e., chopped sugar cane, threshed, or otherwise[43], et cetera. Fluid endurants will be analysed and described in relation to solid endurants, viz. their "containers".

3.4.1.3 Parts

Definition 33 . **Parts:** The non-living species solids are what we shall call parts ∎

Parts are the "work-horses" of man-made domains. That is, we shall mostly be concerned with the analysis and description of endurants into parts.

Example 14 . **Parts:** The previous example of solids was also an example of parts ∎

We distinguish between atomic and compound parts.

3.4.1.3.1 Atomic Parts

Definition 34 . **Atomic Part, I** By an *atomic part* we shall understand a part which the domain analyser considers to be indivisible in the sense of not meaningfully, for the purposes of the domain under consideration, that is, to not meaningfully consist of sub-parts ∎

3.4.1.3.2 Compound Parts

We, pragmatically, distinguish between Cartesian-product-, and set- oriented parts. If Cartesian-oriented, to consist of two or more distinctly sort-named endurants (solids or fluids), If set-oriented, to consist of an indefinite number of zero, one or more parts.

Definition 35 . **Compound Part, I** ***Compound part***s are those which are either Cartesian-product- or are set- oriented parts ∎

Example 15 . **Compound Parts:** A road net consisting of a set of hubs, i.e., street intersections or "end-of-streets", and a set of links, i.e., street segments (with no contained hubs), is a Cartesian compound; and the sets of hubs and the sets of links are part set compounds ∎

---

[42] This is a purely pragmatic decision. "Of course" sand, gravel, soil, etc., are not fluids, but for our modelling purposes it is convenient to "compartmentalise" them as fluids !

[43] See footnote 42.

3.4.2 An Aside: An Upper Ontology

We have been reasonably careful to just introduce and state informal definitions of phenomena and some classes thereof. In the next chapter we shall, in a sense, “repeat” coverage of these phenomena. But now in a more analytic manner. Figure 3.1 is intended to indicate this.

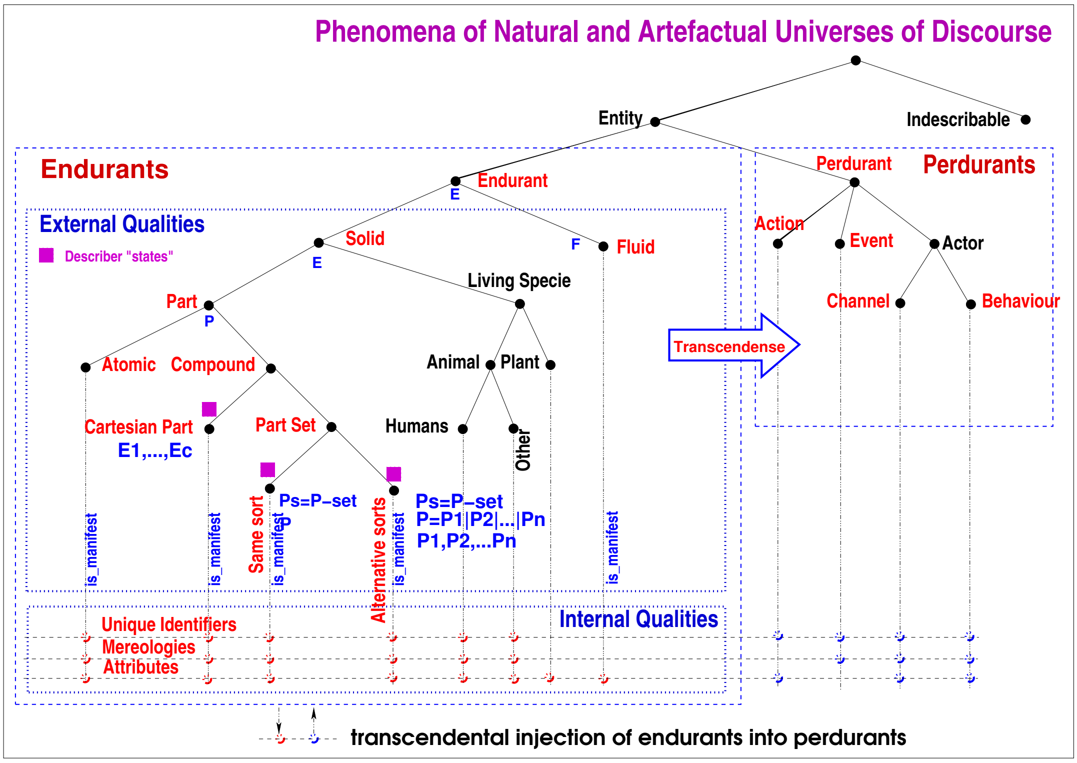


Fig. 3.1 Upper Ontology

So far we have only touched upon the ‘External Qualities’ labeled, dotted-dashed box of the ‘Endurants’ label-led dashed box of Fig. 3.1. In Chapter 4 we shall treat external qualities in more depth — more systematically: analytically and descriptionally.

3.4.3 Internal quality

Definition 36 . **Internal qualities:** those properties [of endurants] that do not occupy *space* but can be measured or spoken about ▪

Example 16 . **Internal qualities:** the unique identity of a part, the relation of part to other parts, and the endurant attributes such as temperature, length, colour ▪

3.4.3.1 Unique identity

Definition 37 . Unique identity: an immaterial property that distinguishes two *spatially* distinct solids ▪

Example 17 . Unique identities: Each hub in a road net is unique identified, so is each link and automobile ▪

3.4.3.2 Mereology

Definition 38 . Mereology: a theory of [endurant] part-hood relations: of the relations of an [endurant] parts to a whole and the relations of [endurant] parts to [endurant] parts within that whole ▪

Example 18 . Mereology: that a link is topologically *connected* to exactly two specific hubs, that hubs are *connected* to zero, one or more specific links, and that links and hubs are *open* to specific subsets of automobiles ▪

3.4.3.3 Attribute

Definition 39 . Attributes: Properties of endurants that are not *spatially* observable, but can be either physically (electronically, chemically, or otherwise) measured or can be objectively spoken about ▪

Example 19 . Attribute: Links have lengths, and, at any one time, zero, one or more automobiles are occupying the links ▪

3.5 Prompts

3.5.1 Analysis Prompts

Definition 40 . Analysis prompt: a predicate or a function that may be posed by humans to facets of a domain. Observing the domain the analyser may then act upon the combination of the particular prompt (whether a predicate or a function,and then what particular one of these it is) thus "applying" it to a domain phenomena, and yielding, in the minds of the humans, either a truth value or some other form of value ▪

3.5.1.1 Analysis Predicate

Definition 41 . Analysis predicates: an analysis prompt which yields a truth value ▪

Example 20 . Analysis Predicates: General examples are can an observable phenomena be rationally described, i.e., an entity, is an entity a solid or a fluid. is a solid endurant a part or a living species ▪

3.5.1.2 Analysis Function

Definition 42 . Analysis function: an analysis prompt which yields some RSL-Text ▪

Example 21 . **Analysis Functions:** Two examples: one yields the endurants of a Cartesian part and their respective sort names, another yields the set of a parts of a part set and their common type ▪

3.5.2 Description Prompt

Definition 43 . **Description prompt:** a function that may be posed by humans who may then act upon it: "applying" it to a domain phenomena, and "yielding" narrative and formal RSL-Texts describing what is being observed ▪

Example 22 . **Description Prompts:** result in RSL-Texts describing for example a (i) Cartesian endurant, or (ii) its unique identifier, (iii) or its mereology, or (iv) its attributes, (iv) or other ▪

3.6 Perdurant Concepts

3.6.1 "Morphing" Parts into Behaviours

As already indicated we shall transcendentally deduce (perdurant) behaviours from those (endurant) parts which we, as domain analysers cum describers, have endowed with all three kinds of internal qualities: unique identifiers, mereologies and attributes. Chapter 6, will show how.

3.6.2 State

Definition 44 . **State:** A state is any set of the parts of a domain ▪

Example 23 . **A Road System State:** The domain analyser cum describer may, In brief, decide that a road system state consists of the road net aggregate (of hubs and links)[44], all the hubs, and all the links, and the automobile aggregate (of all the automobiles)[45], and all the individual automobiles ▪

3.6.3 Actors

Definition 45 . **Actors:** An actor is anything that can initiate an action, an event or a behaviour ▪

---

[44] The road net aggregate, in its perdurant form, may "model" the *Department of Roads* of some country, province, or town.

[45] The automobile aggregate aggregate, in its perdurant form, may "model" the *Department of Vehicles* of some country, province, or town.

3.6.3.1 Action

Definition 46 . **Actions:** An action is a function that can purposefully change a state ▪

Example 24 . **Road Net Actions:** These are some road net actions: The insertion of a new or removal of an existing hub; or the insertion of a new, or removal of an existing link;

3.6.3.2 Event

Definition 47 . **Events:** An event is a function that surreptitiously changes a state ▪

Example 25 . **Road Net Events:** These are some road net events: The blocking of a link due to a mud slide; the failing of a hub traffic signal due to power outage; the blocking of a link due to an automobile accident.

3.6.3.3 Behaviour

Definition 48 . **Behaviours** a behaviour is a set of sequences of actions, events and behaviours ▪

Example 26 . **Road Net Traffic:** Road net traffic can be seen as a behaviour of all the behaviours of automobiles, where each automobile behaviour is seen as sequence of start, stop, turn right, turn left, etc., actions; of all the behaviours of links where each link behaviour is seen as a set of sequences (i.e., behaviours) of "following" the link entering, link leaving, and movement of automobiles on the link; of all the behaviours of hubs (etc.); of the behaviour of the aggregate of roads, viz. *The Department of Roads*, and of the behaviour of the aggregate of automobiles, viz, *The Department of Vehicles*.

3.6.4 Channel

Definition 49 . **Channel:** A channel is anything that allows synchronisation and communication of values between two behaviours ▪

We shall use Tony Hoare's CSP concept [100] to express synchronisation and communication of values between behaviours ***i*** and ***j***. Hence the behaviour ***i*** statement ***ch[j] ! value*** to state that behaviour ***i*** offers, "outputs": **!**, ***value*** to behaviours indicated by ***j***. And behaviour **i** expresses ***ch[j] ?*** that it is willing to accept "input from & synchronise with" behaviour ***i***, **?**, any ***value***.

3.7 Domain Analysis & Description

3.7.1 Domain Analysis

Definition 50 . **Domain Analysis** is the act of studying a domain as well as the result of that study in the form of **informal** statements ▪

3.7.2 Domain Description

Definition 51 . **Domain Description** is the act of describing a domain as well as the result of that act in the form of **narratives** and **formal `RSL`-Text** ▪

3.8 Closing

This chapter has introduced the main concepts of domains such as we shall treat (analyse and describe) domains.[46] The next three chapters shall now systematically treat the analysis and description of domains. That treatment takes concept by concept and provides proper definitions and introduces appropriate analysis and description prompts; one-by-one, in an almost pedantic, hence perhaps "slow" progression! The reader may be excused if they, now-and-then, loose sight of "their way". Hence the present chapter. To show "the way": that, for example, when we treat external endurant qualities, there is still the internal endurant qualities, and that the whole thing leads of to perdurants: actors, actions, events and behaviours.

---

[46] We have omitted treatment of *living species: plants* and *animals* – the latter including *humans*. They will be treated in the next chapter !

Chapter 4

Endurants: External Domain Qualities

Contents

This, the present chapter, as well as Chapter 3, is based on Chapter 4 of [49]. You may wish to study that chapter for more detail.

4.1 Universe of Discourse

Definition 52 . **Universe of Discourse, UoD** By a ***universe of discourse*** we shall understand the same as the ***domain of interest***, that is, the *domain* to be *analysed & described* ▪

4.1.1 Identification

The **first task** of a domain analyser cum describer is to settle upon the domain to be analysed and described. That domain has first to be given a *name*.

4.1.2 Naming

A **first decision** is to give a name to the overall domain sort, that is, the type of the domain seen as an endurant, with that sort, or type, name being freely chosen by the analyser cum describer – with no such sort names having been chosen so far !

4.1.3 Examples

Examples of UoDs[47] We refer to a number of Internet accessible experimental reports[48] of descriptions of the following domains:

- ***railways*** [15, 54, 17],
- ***"The Market"*** [16],
- ***container shipping*** [22],
- ***Web systems*** [32],
- ***stock exchange*** [33],
- ***oil pipelines*** [36],

- ***credit card systems*** [39],
- ***weather information*** [40],
- ***swarms of drones*** [41],
- ***document systems*** [43],
- ***container terminals*** [44],
- ***retail systems*** [47],
- ***assembly plants*** [48],
- ***waterway systems*** [50],
- ***shipping*** [51],
- ***urban planning*** [61].

4.1.4 Sketching

The **second task** of a domain analyser cum describer is to develop a *rough sketch narrative* of the domain. The rough-sketching of [what] a domain [is,] is not a trivial matter. It is not done by a committee! It usually requires repeated "trial sketches". To carry it out, i.e., the sketching, normally requires a combination of physical visits to domain examples, if possible; talking with domain professionals, at all levels; and reading relevant literature. It also includes searching the `Internet` for information. We shall show an example next.

Example 27 . Sketch of a Road Transport System UoD: The road transport system that we have in mind consists of a road net and a set of automobiles (private, trucks, buses, etc.) such that the road net serves to convey automobiles. We consider the road net to consist of hubs, i.e., street intersections, including street segment connection points, and links, i.e., street segments between adjacent hubs[49] ■

4.1.5 Universe of Discourse Description

The general universe of discourse, i.e., domain, description prompt can be expressed as follows:

Domain Description Prompt 1 `calc_Universe_of_Discourse:`
0. `calc_Universe_of_Discourse()` **describer**

❝ **Naming:**
  type UoD
Rough Sketch:
  Text ❞

The above ❝ RSL-**Text** ❞ expresses that the `calc_Universe_of_Discourse()` domain describer generates RSL-**Text**. Here is another example rough sketch:

Example 28 . A Rough Sketch Domain Description: The example is that of the production of rum, say of a **Rum Production** domain. From

10 the sowing, watering, and tending to of sugar cane plants;
11 via the "burning" of these prior to harvest;

---

[49] This "rough" narrative fails to narrate what hubs, links, vehicles, automobiles are. In presenting it here we rely on your a priori understanding of these terms. But that is dangerous! The danger, if we do not painstakingly narrate and formalise what we mean by all these terms, then readers (software designers, etc.) may make erroneous assumptions.

12. the harvest;
13. the collection of harvest from sugar cane fields to
14. the chopping, crushing, (and sometimes repeated) boiling, cooling and centrifuging of sugar cane when making sugar and molasses (into A, B, and low grade batches);
15. the fermentation, with water and yeast, producing a ‘wash’;
16. the (pot still or column still) distilling of the wash into rum;
17. the aging of rum in oak barrels;
18. the charcoal filtration of rum;
19. the blending of rum;
20. the bottling of rum;
21. the preparation of cases of rum for sales/export; and
22. the transportation away from the rum distiller of the rum ▪

Some comments on Example 28: Each of the enumerated items above is phrased in terms of perdurants. Behind each such perdurant lies some endurant. That is, in English, *“every noun can be verbed”, and vice-versa.* So we anticipate the transcendence, from endurants to perdurants.

•••

Method Principle 1 . ***From the “Overall” to The Details:*** Our first principle, as the first task in any new domain modelling project, is to “focus” on the “overall”, that is, on the “entire”, generic domain ▪

4.2 Entities

A core concept of domain modelling is that of an *entity*.

Definition 53 . **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* [116, Vol. I, pg. 665]. If a phenomenon cannot be so **observed and described** then it is not en entity ▪

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

- `is_entity` – where `is_entity(`θ`)` holds if θ is an entity ▪ [50]

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

On Analysis Prompts

The **`is_entity`** predicate function represents the first of a number of analysis prompts. They are “applied” by the domain analyser to phenomena of domains. They yield truth values, true or false, “left” in the mind of the domain analyser ▪

•••

---

[50] ▪ marks the end of an analysis prompt definition.

We have just shown how the `is_entity` predicate prompt can be applied to a universe of discourse. From now on we shall see prompts being applicable to successively more analysed entities. Figure 4.1 [Page 39][51] diagrams a **domain description ontology** of entities. That ontology indicates the sub-classes of endurants for which we shall motivate and for which we shall introduce prompts, predicates and functions.

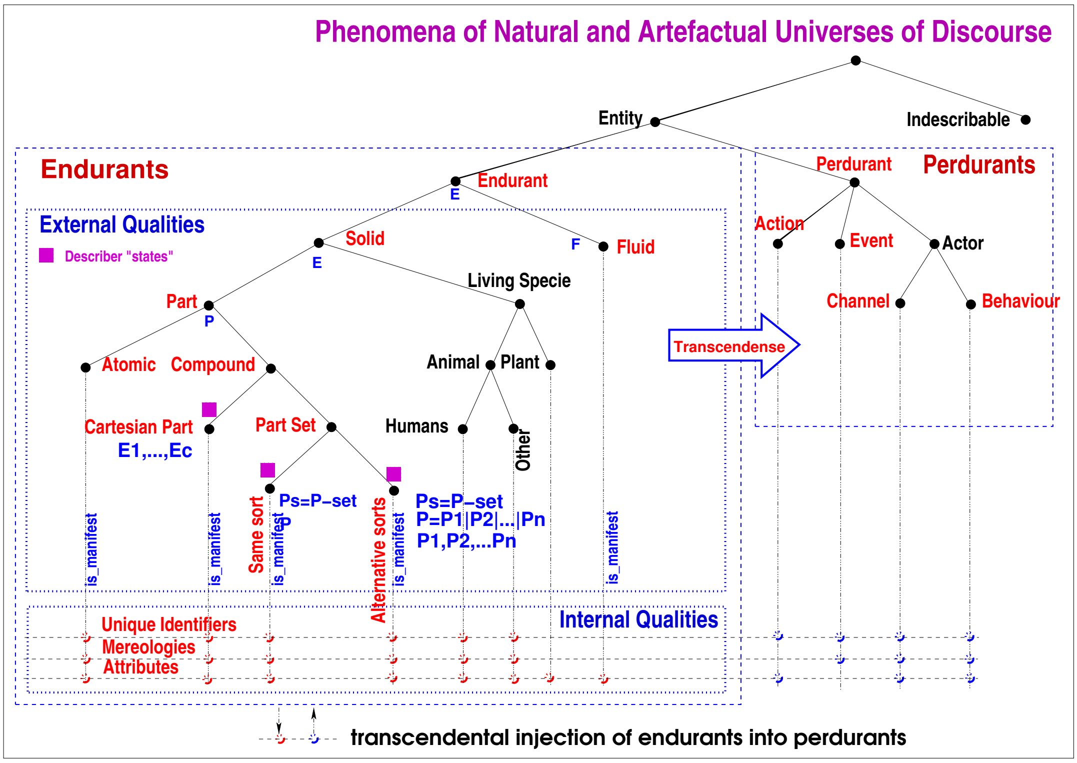


Fig. 4.1 The Upper Ontology

The present chapter shall focus only on the external qualities, that is, on the "contents" of the leftmost dotted box.

• • •

Method Principle 2 . ***Justifying Analysis along Philosophical Lines:*** The concept of *entities* as a main focal point is justified in *Kai Sørlander*'s philosophy[145, 146, 147, 148, 149, 1994–2022]. Entities are there referred to as *primary objects*. They are the ones about which we express predicates ▪

4.3 Endurants and Perdurants

Method Principle 3 . ***Separation of Endurants and Perdurants:*** As we shall see in this **tutorial**, the domain analysis & description method calls for the separation of first considering the careful analysis & description of endurants, then considering perdurants. This principle is based on the transcendental deduction of the latter from the former ▪

---

[51] This ontology was first shown, as Fig. **??** [Page **??**]

4.3.1 Endurants

Definition 54 . **Endurant** By an ***endurant***, to repeat, 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"* [116, Vol. I, pg. 656]. Were we to "freeze" time we would still be able to observe the entire endurant ▪

Example 29 . **Natural and Artefactual Endurants:**
Geography Endurants: fields, meadows, lakes, rivers, forests, hills, mountains, et cetera.
Railway Track Endurants: a railway track, its net, its individual tracks, switch points, trains, their individual locomotives, signals, et cetera.
Road Transport System Endurants: the transport system, its road net aggregate and the aggregate of automobiles, the set of links (road segments) and hubs (road intersections) of the road net aggregate, these links and hubs, and the automobiles.

Analysis Predicate 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`. `is_endurant` is a method tool.

4.3.2 Perdurants

Definition 55 . **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 [116, Vol. II, pg. 1552] ▪

Example 30 . **Perdurants:**
Geography Perdurants: the continuous changing of the weather (meteorology); the erosion of coastlines; the rising of some land area and the "sinking" of other land area; volcanic eruptions; earthquakes; et cetera.
Railway System Perdurants: 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 Predicate 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` ▪

`is_perdurant` is a method tool.

• • •

We repeat method principle 3 on page 39:

Method Principle 4 . ***Separation of Endurants and Perdurants:*** First domain analyse & describe endurants; then domain analyse & describe perdurants ▪

4.4 Solids and Fluids

For *pragmatic* reasons we distinguish between solids and fluids.

Method Principle 5 . ***Abstraction, I:*** The principle of abstraction is now brought into "full play": In analysing & describing entities the domain analyser cum describer is "free" to not consider all facets of entities, that is, to abstract. We refer to our characterisation of abstraction in Sect. **1.4** on page 3.

4.4.1 Solids

Definition 56 . **Solid Endurant:** By a ***solid endurant*** we shall understand an endurant which is separate, individual or distinct in form or concept, or, rephrasing: a body or magnitude of three-dimensions, having length, breadth and thickness [116, Vol. II, pg. 2046] ▪

Analysis Predicate Prompt 4 `is_solid:` The domain analyser analyses endurants, e, into solid entities as prompted by the ***domain analysis prompt***:

- `is_solid` – e is solid if `is_solid`(e) holds ▪

To simplify matters we shall allow separate elements of a solid endurant to be fluid ! That is, a solid endurant, i.e., a part, may be conjoined with a fluid endurant, a fluid. `is_solid` is a method tool.

Example 31 . **Artefactual Solid Endurants:** The individual endurants of the above example of **railway system** endurants, Example 29 on the preceding page, were all solid. Here are examples of solid endurants of **pipeline systems**. A pipeline and its individual units: wells, pipes, valves, pumps, forks, joins, regulator, and sinks.

4.4.2 Fluids

Definition 57 . **Fluid Endurant** By a ***fluid endurant*** we shall understand an endurant which is prolonged, without interruption, in an unbroken series or pattern; or, rephrasing: a substance (liquid, gas or plasma) having the property of flowing, consisting of particles that move among themselves [116, Vol. I, pg. 774] ▪

Analysis Predicate Prompt 5 `is_fluid:` The domain analyser analyses endurants *e* into fluid entities as prompted by the ***domain analysis prompt***:

- `is_fluid` – *e* is fluid if `is_fluid(`*e*`)` holds ▪

`is_fluid` is a method tool. Fluids are otherwise liquid, or gaseous, or plasmatic, or granular[52], or plant products[53], et cetera.

Example 32 . **Fluids:** Specific examples of fluids are: water, oil, gas, compressed air, etc. A container, which we consider a solid endurant, may be *conjoined* with another, a fluid, like a gas pipeline unit may "contain" gas ▪

4.5 Parts and Living Species

We analyse endurants into either of two kinds: *parts* and *living species*. The distinction between *parts* and *living species* is motivated in Kai Sørlander's Philosphy [145, 146, 147, 148, 149].

4.5.1 Parts

Definition 58 . **Parts** By a *part* we shall understand a solid endurant existing in time and subject to laws of physics, including the *causality principle* and *gravitational pull*[54] ▪

Analysis Predicate Prompt 6 `is_part:` The domain analyser analyses "things" (e) into part. The method can thus be said to provide the ***domain analysis prompt***:

- `is_part` – where `is_part(e)` holds if e is a part ▪

`is_part` is a method tool.
Parts are either *natural* parts, or are *artefactual* parts, i.e. man-made. Natural and man-made parts are either *atomic* or *compound*.

4.5.1.1 Atomic Parts

The term 'atomic' is, perhaps, misleading. It is not used in order to refer to nuclear physics. It is, however, chosen in relation to the notion of *atomism: a doctrine that the physical or physical and mental universe is composed of simple indivisible minute particles* `[Merriam Webster]`.

---

[52] This is a purely pragmatic decision. "Of course" sand, gravel, soil, etc., are not fluids, but for our modelling purposes it is convenient to "compartmentalise" them as fluids !

[53] i.e., chopped sugar cane, threshed, or otherwise. See footnote 52.

[54] This characterisation is the result of our study of relations between philosophy and computing science, notably influenced by Kai Sørlander's Philosphy [145, 146, 147, 148, 149]

Definition 59 . **Atomic Part, II** By an *atomic part* we shall understand a part which the domain analyser considers to be indivisible in the sense of not meaningfully, for the purposes of the domain under consideration, that is, to not meaningfully consist of sub-parts ▪

Example 33 . **Atomic Parts:** We refer to Example 31 on page 41: pipeline systems. The wells, pumps, valves, pipes, forks, joins and sinks can be considered atomic ▪

Analysis Predicate Prompt 7 `is_atomic:` The domain analyser analyses "things" (e) into atomic part. The method can thus be said to provide the ***domain analysis prompt***:

- `is_atomic` – where `is_atomic(e)` holds if e is an atomic part ▪

`is_atomic` is a method tool.

4.5.1.2 Compound Parts, II

We, pragmatically, distinguish between Cartesian-product-, and set- oriented parts. That is, if Cartesian-product-oriented, to consist of two or more distinctly sort-named endurants (solids or fluids), or, if set-oriented, to consist of an indefinite number of zero, one or more identically sort-named parts.

Definition 60 . **Compound Part** ***Compound part***s are those which are either Cartesian-product- or are set- oriented parts ▪

Analysis Predicate Prompt 8 `is_compound:` The domain analyser analyses "things" (e) into compound part. The method can thus be said to provide the ***domain analysis prompt***:

- `is_compound` – where `is_compound(e)` holds if e is a compound part ▪

`is_compound` is a method tool.

4.5.1.2.1 Cartesian Parts

Definition 61 . **Cartesian Part** ***Cartesian part***s are those (compound parts) which consists of an "indefinite number" of two or more parts of distinctly named sorts ▪

Some clarification may be needed. (i) In mathematics, as in RSL [84], a value is a Cartesian value if it can be expressed, for example as $(a,b,...,c)$, where $a,b,...,c$ are mathematical (or, which is the same, RSL) values. Let the sort names of these be $A,B,...,C$ – with these being required to be distinct. We wrote "indefinite number": the meaning being that the number is fixed, finite, but not specific. (ii) The requirement: 'distinctly named' is pragmatic. If the domain analyser cum describer thinks that two or more of the components of a Cartesian part are [really] of the same sort, then that person is most likely confused and must come up with suitably distinct sort names for these "same sort" parts! (iii) Why did we not write "definite number"? Well, at the time of first analysing a Cartesian part, the domain analyser cum describer may not have thought of all the consequences, i.e., analysed, the compound part. Additional sub-parts, of the Cartesian compound, may be "discovered", subsequently and can then, with the approach we are taking wrt. the modelling of these, be "freely" added subsequently !

Example 34 . **Cartesian Automobiles:** We refer to Example 29 on page 40, the **transport system** sub-example. We there viewed (hubs, links and) automobiles as atomic parts. From another point of view we shall here understand automobiles as Cartesian parts: the engine train, the chassis, the car body, four doors (left front, left rear, right front, right rear), and the wheels. These may again be considered Cartesian parts.

Analysis Predicate Prompt 9 `is_Cartesian:` The domain analyser analyses "things" (e) into Cartesian part. The method can thus be said to provide the ***domain analysis prompt***:

- `is_Cartesian` – where `is_Cartesian(e)` holds if e is a Cartesian part ▪

`is_Cartesian` is a method tool.

4.5.1.2.2 Calculating Cartesian Part Sorts

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

• • •

On Determination Functions

Determination functions apply to compound parts and yield their sub-parts and the sorts of these. *That is, we observe the domain and our observation results in a focus on a subset of that domain and sort information about that subset.*

An RSL Extension

The determine_··· functions below are expressed as follows:

value determine_···(e) **as** (parts,sorts)

where we focus here on the sorts clause. Typically that clause is of the form ηA,ηB,...,ηC.[55] That is, a "pattern" of sort names: A,B,...,C. These sort names are provided by the domain analyser cum describer. They are chosen as "full names", or as mnemonics, to capture an essence of the (to be) described sort. Repeated invocations, by the domain analyser cum describer, of these (...,sorts) analysis functions normally lead to new sort names distinct from previously chosen such names.

---

[55] ηA,ηB,...,ηC are **names** of types. $\eta\theta$ is the type of all type names !

4.5.1.2.2.1 Cartesian Part Determination

Observer Function Prompt 1 `determine_Cartesian_parts:`
The domain analyser analyses a part into a Cartesian part. The method provides the ***domain observer prompt***:

- `determine_Cartesian_parts` — it directs the domain analyser to determine the definite number of values and corresponding distinct sorts of the part.

  value
  determine_Cartesian_parts: E → (E1×E2×...×En) × (ηE1×ηE2×...×ηEn)[56]
  determine_Cartesian_parts(e) **as** ((e1,...,en),(ηE1,...,ηEn))

  where by E, Ei we mean endurants, i.e., part values, and by ηEi we mean the names of the corresponding types.

`determine_Cartesian_parts` is a method tool.

On Calculate Prompts

Calculation prompts apply to compound parts: Cartesians and sets, and yield an RSL-Text description.

Domain Description Prompt 2 `calc_Cartesian_parts:` If `is_Cartesian`(e) holds, then the analyser "applies" the ***domain description prompt***

- `calc_Cartesian_parts`(e)

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

1. `calc_Cartesian_parts(e)` describer

let (_[57],(ηE$_1$,...,ηE$_m$)) = determine_Cartesian_parts_sorts(e)[58] **in**
❝**Narration:**
[s] ... narrative text on sorts ...
[o] ... narrative text on sort observers ...
[p] ... narrative text on proof obligations ...
Formalisation:
type
[s] E$_1$, ❞...❝ , E$_m$
value
[o] obs_E$_1$: E → E$_1$, ❞...❝ , obs_E$_m$: E → E$_m$
proof obligation
[p] [Disjointness of endurant sorts]❞
end

`calc_Cartesian_parts` is a method tool.

---

[56] The ordering, ((e1,...,en),(ηE1,...,ηEn)), is pairwise arbitrary.

[57] The use of the underscore, _ , shall inform the reader that there is no need, here, for naming a value.

[58] For `determine_composite_parts` see Sect. **4.5.1.2.2.1**

Elaboration 1 **Type, Values and Type Names:** Note the use of quotes above. Please observe that when we write obs_E then obs_E is the name of a function. The E, when juxtaposed to obs_ is now a name ▪

Observer Function Prompt 2 **type_name, type_of:**
The definition of **type_name, type_of** implies the informal definition of

$$\begin{array}{l} \mathsf{obs\_E}_i(e)=e_i \equiv \mathsf{type\_name}(e_i)=\text{“}E_i\text{”} \wedge \\ \mathsf{type\_of}(e_i) \equiv E_i \wedge \\ \mathsf{is\_E}_i(e_i) \end{array}$$

Example 35 . **A Road Transport System Domain: Cartesians:** [59]

23 There is the *universe of discourse*, RTS.
It is composed from
24 a *road net*, RN, and
25 an *aggregate of automobiles*, AA.

type
23 RTS
24 RN
25 AA

value
24 obs_RN: RTS → RN
25 obs_AA: RTS → AA ▪

We continue the analysis & description of "our" road transport system:

26 The road net consists of
  a an aggregate, AH, of hubs and
  b an aggregate, AL, of links.

type
26a AH
26b AL

value
26a obs_AH: RN → AH
26b obs_AL: RN → AL

4.5.1.2.3 Part Sets

Definition 62 . **Part Sets** ***Part set***s are those which, in a given context, are deemed to *meaningfully* consist of separately observable a ["root"] part and an indefinite number of proper ["sibling"] *sub-parts* ▪

For pragmatic reasons we distinguish between parts sets all of whose parts are of the same, single, further un-analysed sort, and of two or more distinct atomic sorts.

---

[59] Example 35′ **Narration** is not representative of what it should be. Here is a more reasonable narration:

- A road net is a set of hubs (road intersections) and links such that links are connected to adjacent hubs, and such that connected links and hubs form *roads* and where `a road is a thoroughfare, route, or way on land between two places that has been paved or otherwise improved to allow travel by foot or some form of conveyance, including a motor vehicle, cart, bicycle, or horse [Wikipedia]`

We bring this clarification here, once, and allow ourselves, with the reader's permission, to narrate only very steno-graphically.

Definition 63 . **Single Sort Part Sets** ***Single sort part set***s are those which, in a given context, are deemed to *meaningfully* consist of separately observable a ["root"] part and an indefinite number of proper ["sibling"] *sub-part*s of the same, i.e., single sort ▪

Analysis Predicate Prompt 10 `is_single_sort_set:` The domain analyser analyses a solid endurant, i.e., a part p into a set endurant:

- `is_single_sort_set`: p is a composite endurant if `is_single_sort_set(`p`)` holds ▪

`is_single_sort_set` is a method tool.

The `is_single_sort_set` predicate is informal. So are all the domain analysis predicates (and functions). That is, Their values are "calculated" by a human, the domain analyser. That person observes parts in the "real world". The determination of the predicate values, hence, are subjective.

Definition 64 . **Alternative Atomic Part Sets** ***Alternative sorts part set***s are those which, in a given context, are deemed to *meaningfully* consist of separately observable a ["root"] part and an indefinite number of proper ["sibling"] *sub-part*s of two or more atomic parts of distinct sorts ▪

Analysis Predicate Prompt 11 `is_alternative_sorts_set:` The domain analyser analyses a solid endurant, i.e., a part p into a set endurant:

- `is_alternative_sorts_set`: p is a composite endurant if `is_alternative_sorts_set(`p`)` holds ▪

`is_alternative_sorts_set` is a method tool.

4.5.1.2.3.1 Determine Same Sort Part Sets

Observer Function Prompt 3 `determine_same_sort_parts_set:`
The domain analyser observes parts into same sorts part sets. The method provides the ***domain observer prompt***:

- `determine_alternative_sorts_part_set` directs the domain analyser to determine the values and corresponding sorts of the part.

  value
  determine_same_sort_part_set: E → (P-**set**×θP)
  determine_same_sort_part_set(e) **as** (ps,ηPn)

`determine_same_sort_part_set` is a method tool.

4.5.1.2.3.2 Determine Alternative Sorts Part Sets

Observer Function Prompt 4 `determine_alternative_sorts_part_set:`
The domain analyser observes parts into alternative sorts part sets. The method provides the ***domain observer prompt***:

- `determine_alternative_sorts_part_set` directs the domain analyser to determine the values and corresponding sorts of the part.

```
value
    determine_alternative_sorts_part_set: E → ((P1×θP1)×...×(Pn,θPn))
    determine_alternative_sorts_part_set(e) as ((p1,ηp1),...,(pn,ηPn))
```

The set of parts, of different sorts, may have more than one element, $p, p', ..., p''$ being of the same sort *Ei*.

`determine_alternative_sorts_part_set` is a method tool.

4.5.1.2.3.3 Calculating Single Sort Part Sets

Domain Description Prompt 3 `calc_single_sort_parts_sort:` If `is_single_set_sort_parts`(e) holds, then the analyser "applies" the ***domain description prompt***

- `calc_single_sort_parts_sort`(e)

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

2. `calculate_single_sort_parts_sort(e)` Describer

```
    let (_,ηP) = determine_single_sort_part(e)[60] in
“Narration:
    [s]  ... narrative text on sort ...
    [o]  ... narrative text on sort observer ...
    [p]  ... narrative text on proof obligation ...
Formalisation:
    type
    [s]  P
    [s]  Ps = P-set
    value
    [o]  obs_Ps: E → Ps ”
    proof obligation
    [p]  [ Single ‶sortness″ of Ps ]”
    end
```

`calculate_single_sort_parts_sort` is a method tool.

Elaboration 2 **Type, Values and Type Names:** Note the use of quotes above. Please observe that when we write obs_Ps then obs_Ps is the name of a function. The Ps, when juxtaposed to obs_ is now a name ■

Example 36 . **Road Transport System: Sets of Hubs, Links and Automobiles:** We refer to Example 35 on page 46.

27 The road net aggregate of road net hubs consists of a set of [atomic] hubs,
28 The road net aggregate of road net links consists of a set of [atomic] links,

---

[60] For `determine_single_sort_part` see Defn. 63 on the previous page.

29 The road net aggregate of automobiles consists of a set of [atomic] automobiles.

```
type
27.  Hs = H-set, H
27.  Ls – L-set, L
27.  As = A-set, A
value
27.  obs_Hs: AH → Hs
27.  obs_Ls: AL → Ls
27.  obs_As: AA → As
```
▪

4.5.1.2.3.4 Calculating Alternative Sort Part Sets

We leave it to the reader to decipher the `calculate_alternative_sort_part_sorts` prompt.

Domain Description Prompt 4 `calculate_alternative_sort_part_sorts:` If `is_alternative_sort_parts_sorts`(*e*) holds, then the analyser “applies” the ***domain description prompt***

- `calculate_alternative_sort_part_sorts`(*e*)

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

`3.calculate_alternative_sort_part_sorts(e)` **Describer**

```
    let ((p1,ηE_1),...,(pn,ηE_n)) = determine_alternative_sorts_part_set_sorts(e)[61] in
❝ Narration:
    [s]  ... narrative text on alternative sorts ...
    [o]  ... narrative text on sort observers ...
    [p]  ... narrative text on proof obligations ...
Formalisation:
    type
    [s]  Ea = E_1 | ... | E_n
    [s]  E_1 :: End_1, ..., E_n :: End_n
    value
    [o]  obs_Ea: E → Ea
    proof obligation
    [p]  [ disjointness of alternative sorts ] E_1, ..., E_n ❞
    end
```

The set of parts, of different sorts, may have more than one element, say $p,p',...,p''$ being of the same sort E_i. Since parts are not mentioned in the sort description above, cf., _, only the distinct alternative sort observers appear in that description.

`calculate_alternative_sort_part_sorts` is a method tool.

Example 37 . **Alternative Rail Units:**

30 The example is that of a railway system.
31 We focus on railway nets. They can be observed from the railway system.

---

[61] For `determine_alternative_sort_part_sorts` see Defn. 64 on page 47.

32 The railway net embodies a set of [railway] net units.
33 A net unit is either astraight or curved **linear** unit, or a simple switch, i.e., a **turnout**, unit[62] or a simple cross-over, i.e., a **rigid** crossing unit, or a single switched cross-over, i.e., a **single** slip unit, or a double switched cross-over, i.e., a **double** slip unit, or a **terminal** unit.
34 As a formal specification language technicality disjointness of the respective rail unit types is afforded by `RSL`'s :: type definition construct.

We refer to Figure 4.2.

type
30. RS
31. RN
value
31. obs_RN: RS → RN
type
32. NUs = NU-**set**
33. NU = LU|PU|RU|SU|DU|TU
34. LU :: LinU
34. PU :: PntU
34. SU :: SwiU
34. DU :: DblU
34. TU :: TerU
value
32. obs_NUs: RN → NUs

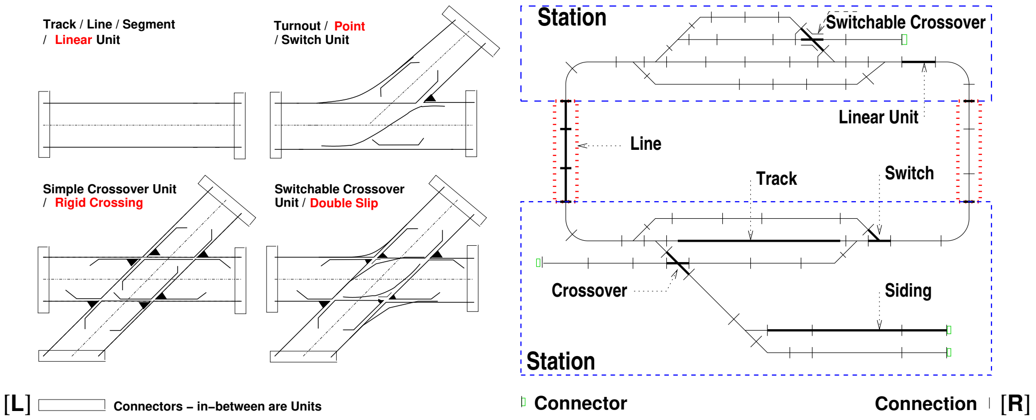


Fig. 4.2 Left: Four net units (LU, PU, SU, DU); Right: A railway net

•••

Method Principle 6 . ***Pedantic Steps of Development:*** This section, i.e., Sect. **4.5.1**, has illustrated a principle of "small, pedantic" analysis & description steps. You could also call it a principle of separation of concerns ▪

4.5.1.3 Ontology and Taxonomy

We can speak of two kinds of ontologies: the general ontologies of domain analysis & description, cf. Fig. 4.1 on page 39, and a specific domain's possible endurant ontologies. We shall here focus on a ["restricted"] concept of taxonomies[63]

---

[62] https://en.wikipedia.org/wiki/Railroad_switch

[63] By taxonomy (or taxonomical classification) we shall here understand a scheme of classification, especially a hierarchical classification, in which things are organized into groups `[Wikipedia]`.

Definition 65 . **Domain Taxonomy** By a domain taxonomy we shall understand a hierarchical structure, usually depicted as a(n "upside-down") tree, whose "root" designates a compound part and whose "siblings" (proper sub-trees)designate parts or fluids ■

The 'restriction' amounts to considering only endurants. That is, not considering perdurants. `Taxonomy` is a method technique.

Example 38 . **The Road Transport System Taxonomy:** Figure 4.3 shows a schematised, i.e., the . . . , taxonomy for the *Road Transport System* domain of Example 4.1 on page 39.

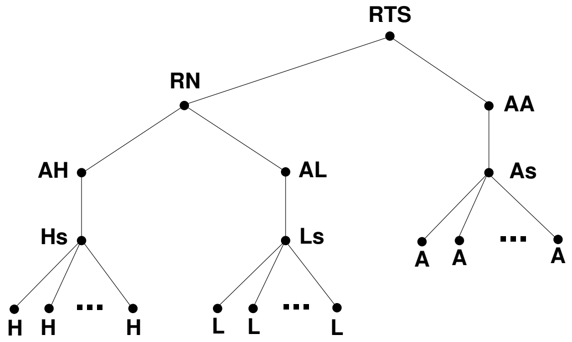


Fig. 4.3 A Road Transport System Taxonomy ■

4.5.1.4 "Root" and "Sibling" Parts

For compound parts, cf. Definition 60 on page 43, we introduce the specific domain taxonomy concepts of "root" and "sibling" parts. (We also refer to Fig. 4.3.)

When observing, as a human, a compound part one may ask the question *"a tree consisting of a specific domain taxonomy node labelled, e.g., X and the sub-trees labelled, e.g., $Y_1, Y_2, \ldots, Y_n$ does that tree designate one "indivisible" part or does it designate $n+1$ parts ?"* We shall, in general, consider the answer to be the latter: $n+1$!

We shall, in general, consider compound parts to consist of a "root" parts and n "sibling parts and fluids". What the analyser cum describer observes appears as one part, "the whole", with n "embedded" sub-parts. What the analyser cum describer is asked to model is 1, the root part, and n, the sibling, parts and fluids. The fact that the root part is separately modelled from the sibling parts, may seem to disappear in this separate modelling — but, as You shall see, in the next chapter, their relation: the siblings to "the whole", i.e., the root, will be modelled, specifically through their mereologies, as will be covered in Sect. **5.3**, but also through their respective attributes, Sect. **5.4**. We shall see this non-embbedness of root and sibling parts further accentuated in the modelling of their transcendentally deduced respective (perdurant) behaviours as distinct concurrent behaviours in Chapter 6.

4.5.2 Living Species

Living Species are either ***plants*** or ***animals***. Among animals we have the ***humans***.

Definition 66 . **Living Species** By a *living species* we shall understand a solid endurant, subject to laws of physics, and additionally subject to *causality of purpose*.

Living species must have some *form they can be developed to reach*; a form they must be *causally determined to maintain*. This *development and maintenance* must further engage in *exchanges of matter with an environment*. It must be possible that living species occur in two forms: **plants**, respectively **animals**, forms which are characterised by *development, form and exchange*, which, additionally, can be characterised by the *ability of purposeful movement* ■

Analysis Predicate Prompt 12 `is_living_species:` The domain analyser analyses "things" (e) into living species. The method can thus be said to provide the ***domain analysis prompt***:

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

`is_living_species` is a method tool.

It is appropriate here to mention **Carl Linnaeus** (1707–1778). He was a Swedish botanist, zoologist, and physician who formalised, in the form of a binomial nomenclature, the modern system of naming organisms. He is known as the "father of modern taxonomy". We refer to his 'Species Plantarum' **gutenberg.org/files/20771/20771-h/20771-h.htm**.

4.5.2.1 Plants

Example 39 . **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 Predicate Prompt 13 `is_plant:` The domain analyser analyses "things" (ℓ) into a plant. The method can thus be said to provide the ***domain analysis prompt***:

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

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

4.5.2.2 Animals

Definition 67 . **Animal** We refer to the initial definition of *living species* above – while emphasizing the following traits: (i) a *form that animals can be developed to reach* and (ii) *causally determined to maintain* through (iii) *development and maintenance* in an *exchange of matter with an environment*, and (iv) *ability to purposeful movement* ■

Analysis Predicate Prompt 14 `is_animal:` The domain analyser analyses "things" (ℓ) into an animal. The method can thus be said to provide the ***domain analysis prompt***:

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

`is_animal` is a method tool. The predicate `is_living_species(ℓ)` is a prerequisite for `is_animal(ℓ)`. We distinguish, motivated by [148], between humans and other.

4.5.2.2.1 Humans

Definition 68 . **Human** A *human* (a *person*) is an *animal*, cf. Definition 67 on the facing page, with the additional properties of having *language*, being *conscious* of *having knowledge* (of its own situation), and *responsibility* ▪

Analysis Predicate Prompt 15 `is_human:` The domain analyser analyses "things" (ℓ) into a human. The method can thus be said to provide the ***domain analysis prompt***:

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

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

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, that is, that these artefacts have human characteristics. You, the readers, 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 !

4.5.2.2.2 Other

We shall skip any treatment of other than human animals !

•••

`External Quality Analysis & Description First` is a method procedure.

4.6 Some Observations

Two observations must be made.

(i) The domain analyser cum describer procedures illustrated by the analysis functions `determine_Cartesian_parts`, `determine_same_sort_part_set` and `determine_alternative_sorts_part_set` yield names of endurant sorts. Some of these names may have already been encountered, i.e., discovered. That is, the domain analyser cum describer must carefully consider such possibilities.

(ii) Endurants are **not recursively definable** ! This appears to come as a surprise to many computer scientists. Immediately many suggest that "tree-like" endurants like a river, or, indeed, a tree, should be defined recursively. But we posit that that is not the case. A river, for example, has a delta, its "root" so-to-speak, but the sub-trees of a recursively defined river endurant has no such "deltas"! Instead we define such "tree-like" endurants as graphs with appropriate mereologies.

4.7 States

In our continued modelling we shall make good use of a concept of states.

Definition 69 . **State** By a *state* we shall understand any collection of one or more parts ■

In Chapter 5 Sect. **5.4** we introduce the notion of *attributes.* Among attributes there are the *dynamic attributes*. They model that internal part quality values may change dynamically. So we may wish, on occasion, to 'refine' our notion of state to be just those parts which have dynamic attributes.

4.7.1 State Calculation

Given any universe of discourse, uod:UoD, we can recursively calculate its "full" state, calc_parts({uod}).

35 Let e be any endurant. Let arg_parts be the parts to be calculated. Let res_parts be the parts calculated. Initialise the calculator with arg_parts={e} and res_parts={}. Calculation stops with arg_parts empty and res_parts the result.
36 If `is_Cartesian`(e)
37 then we obtain its immediate parts, determine_composite_part(e)
38 add them, as a set, to arg_parts, e removed from arg_parts and added to res_parts calculating the parts from that.
39 If `is_single_sort_part_set`(e)
40 then the parts, ps, of the single sort set are determined,
41 added to arg_parts and e removed from arg_parts and added to res_parts calculating the parts from that.
42 If `is_alternative_sorts_part_set`(e) then the parts, ((p1,_),(p2,_),...,(pn,_)), of the alternative sorts set are determined, added to arg_parts and e removed from arg_parts and added to res_parts calculating the parts from that.

```
value
35.  calc_parts: E-set → E-set → E-set
35.  calc_parts(arg_parts)(res_parts) ≡
35.    if arg_parts = {} then res_parts else
35.    let e • e ∈ arg_parts in
36.    is_Cartesian(e) →
37.       let ((e1,e2,...,en),_) = observe_Cartesian_part(e) in
38.       calc_parts(arg_parts\{e} ∪ {e1,e2,...,en})(res_parts ∪ {e}) end
39.    is_single_sort_part_set(e) →
40.       let ps = observe_single_sort_part_set(e) in
41.       calc_parts(arg_parts\{e}∪ ps)(res_parts ∪ {e}) end
42.    is_alternative_sort_part_set(e) →
42.       let ((p1,_),(p2,_),...,(pn,_)) = observe_alternative_sorts_part_set(e) in
```

```
42.         calc_parts(arg_parts\{e}∪{p1,p2,...,pn})(res_parts ∪ {e}) end
35.     end end
```

`calc_parts` is a method tool.

Method Principle 7 . ***Domain State:*** We have found, once all the state components, i.e., the endurant parts, have had their external qualities analysed, that it is then expedient to define the domain state. It can then be the basis for several concepts of internal qualities.

Example 40 . **Constants and States:**

43 Let there be given a universe of discourse, *rts*. The set {*rts*} is an example of a state.

From that state we can calculate other states.

44 The set of all hubs, *hs*.
45 The set of all links, *ls*.
46 The set of all hubs and links, *hls*.
47 The set of all automobiles, *as*.
48 The set of all parts, *ps*.

value
43 *rts*:UoD [43]
44 *hs*:H-**set** ≡ obs_sH(obs_SH(obs_RN(*rts*)))
45 *ls*:L-**set** ≡ obs_sL(obs_SL(obs_RN(*rts*)))
46 *hls*:(H|L)-**set** ≡ *hs*∪*ls*
47 *as*:A-**set** ≡ obs_As(obs_AA(obs_RN(*rts*)))
48 *ps*:(UoB|H|L|A)-**set** ≡ *rts*∪*hls*∪*as*

4.7.2 Update-able States

We shall, in Sect. **5.4**, introduce the notion of parts, having dynamic attributes, that is, having internal qualities that may change. To cope with the modelling, in particular of so-called *monitor-able* attributes, we present the *state* as a global variable:

variable σ := calc_parts({uod})

4.8 An External Analysis and Description Procedure

We have covered the individual analysis and description steps of our approach to the external qualities modelling of domain endurants. We now suggest a 'formal' description of the process of linking all these analysis and description steps.

4.8.1 An Analysis & Description State

Common to all the discovery processes is an idea of a *notice board*. A notice board, at any time in the development of a domain description, is a repository of the analysis and description process. We suggest to model the notice board in terms of three global variables. The **new** variable holds the **parts** yet to be described, The **ans** variable holds the **sort name of parts** that have so far been described, the **gen** variable holds the **parts** that have so far been described, and the **txt** variable holds the **RSL-Text** so far generated. We model the **txt** variable as a map from endurant identifier names to **RSL-Text**.

A Domain Discovery Notice Board

```
variable
    new := {uod} ,
    asn := { “UoD ”}
    gen := {} ,
    txt:RSL-Text := [ uid_UoD(uod) ↦ ⟨“type UoD”⟩ ]
```

4.8.2 A Domain Discovery Procedure, I

The discover_sorts pseudo program suggests a systematic way of proceeding through analysis, manifested by the is_··· predicates, to (→) description.

Some comments are in order. The e-set_a⊞e-set_b expression yields a set of endurants that are either in e-set_a, or in e-set_a, or in both, but such that two endurants, e_x and e_y which are of the same endurants type, say E, and are in respective sets is only represented once in the result; that is, if they are type-wise the same, but value-wise different they will only be included once in the result.

As this is the first time RSL-Text is put on the notice board we express this as:

- txt := txt ∪ [type_name(v) ↦ ⟨RSL-Text⟩]

Subsequent insertion of RSL-Text for internal quality descriptions and perdurants is then concatenated to the end of previously uploaded RSL-Text.

An External Qualities Domain Analysis and Description Process

```
value
discover_sorts: Unit → Unit
discover_sorts() ≡ while new ≠ {} do
    let v • v ∈ new in (new := new \ {v} || gen := gen ∪ {v} || ans := ans \ {type_of(v)}) ;
    is_atomic(v) → skip ,
    is_compound(v) →
      is_Cartesian(v) →
       let ((e1,...,en),(ηE1,...,ηEn))=analyse_composite_parts(v) in
       (ans := ans ∪ {ηE1,...,ηEn} || new := new ⊞ {e1,...,en}
         || txt := txt ∪ [type_name(v) ↦ ⟨calculate_composite_part_sorts(v)⟩]) end,
      is_part_set(v) →
       (is_single_sort_set(v) →
          let ({p1,...,pn},ηP)=analyse_single_sort_parts_set(v) in
          (ans := ans ∪ {ηP} || new := new ⊞ {p1,...,pn} ||
             txt := txt ∪ [type_name(v) ↦ calculate_single_sort_part_sort(v)]) end,
         is_alternative_sorts_set(v) →
          let ((p1,ηE1),...,(pn,ηEn))= observe_alternative_sorts_part_set(v) in
          (ans := ans ∪ {ηE1,...,En} || new := new ⊞ {p1,...,pn} ||
```

```
            txt := txt ∪ [type_name(v) ↦ calculate_alternative_sorts_part_sort(v)]) end)
   end end
```

`discover_sorts` is a method procedure.

4.9 Summary

We briefly summarise the main findings of this chapter. These are the main analysis predicates and functions and the main description functions. These, to remind the reader, are the *analysis*, the **is_**···, *predicates*, the *analysis*, the **determine_**···, *functions*, the *state calculation* function, the *description* functions, and the *domain discovery* procedure. They are summarised in this table:

External Qualities Predicates and Functions: Method Tools

- **Analysis Predicates:** These are the `is_···` functions. The domain scientist cum engineer, i.e., the domain analyser cum describer, applies this to entities being observed in the domain. The answer is a truth value. Dependent on the truth value that person then goes on to apply, again informally, either a subsequent predicate, or some function.
- **Analysis Functions:** These are the `determine_···` functions. They apply, respectively, to parts satisfying respective predicates.
- **State Calculation:** The state calculation function is given generally. The domain analyser cum describer must define this function for each domain studied.
- **Description Functions:** These calculation functions, in a sense, are the main "results" of this chapter.
- **Domain Discovery:** The procedure here being described, informally, guides the domain analyser cum describer to do the job !

#	Name	Introduced
	Analysis Predicates	
1	is_entity	page **38**
2	is_endurant	page **40**
3	is_perdurant	page **40**
4	is_solid	page **41**
5	is_fluid	page **42**
6	is_part	page **42**
7	is_atomic	page **43**
8	is_compound	page **43**
9	is_Cartesian	page **44**
10	is_single_sort_set	page **47**
11	is_alternative_sorts_set	page **47**
12	is_living_species	page **52**
13	is_plant	page **52**
14	is_animal	page **53**
15	is_human	page **53**
	Analysis Functions	
1	determine_Cartesian_parts	page **45**
3	determine_same_sort_part_set	page **47**
4	determine_alternative_sorts_part_set	page **47**
	State Calculation	
	calc_parts	page **54**
	Description Functions	
1	calc_Universe_of_Discourse	page **37**
2	calc_Cartesian_parts	page **45**
3	calc_single_sort_parts_sort	page **48**
4	calc_alternative_sort_part_sorts	page **48**
	Domain Discovery	
	discover_sorts	page **56**

• • •

Please consider Fig. 4.1 on page 39. This chapter has covered the tree-like structure to the left in Fig. 4.1. The next chapter covers the horisontal and vertical lines, also to the left in Fig. 4.1.

Chapter 5

Endurants: Internal and Universal Domain Qualities

Contents

Please consider Fig. 4.1 on page 39. The previous chapter covered the tree-like structure to the left in Fig. 4.1. This chapter covers the horisontal and vertical lines, also to the left in Fig. 4.1.

• • •

In this chapter we introduce the concepts of internal qualities of endurants and universal qualities of domains, and cover, first, the analysis and description of internal qualities: **unique identifiers** (Sect. **5.2** on page 62), **mereologies** (Sect. **5.3** on page 67) and **attributes** (Sect. **5.4** on page 72), There is, additionally, three universal qualities: **space**, **time** (Sect. **5.5** on page 83) and **intentionality** (Sect. **5.6** on page 89), where *intentionality* is “something” that expresses intention, design idea, purpose of artefacts – well, some would say, also of natural endurants.

As it turns out[64], to analyse and describe mereology we need to first analyse and describe unique identifiers; and to analyse and describe attributes we need to first analyse and describe mereologies. Hence:

---

[64] You, the first time reader cannot know this, i.e., the “turns out”. Once we have developed and presented the material of this chapter, then you can see it; clearly !

Method Procedure 1 . ***Sequential Analysis & Description of Internal Qualities:*** We advise that the domain analysis & description **first** analyse & describe **unique identification** of all endurant sorts; **then** analyse & describe **mereologies** of all endurant sorts; **finally** analyse & describe **attributes** of all endurant sorts.

5.1 Internal Qualities

We shall investigate the, as we shall call them, internal qualities of domains. That is the properties of the entities to which we ascribe internal qualities. The outcome of this chapter is that the reader will be able to model the internal qualities of domains. Not just for a particular domain instance, but a possibly infinite set of domain instances[65].

5.1.1 General Characterisation

External qualities of endurants of a manifest domain are, in a simplifying sense, those we can see and touch. They, so to speak, take form.

Internal qualities of endurants of a manifest domain are, in a less simplifying sense, those which we may not be able to see or "feel" when touching an endurant, but they can, as we now 'mandate' them, be reasoned about, as for **unique identifiers** and **mereologies**, or be measured by some **physical/chemical** means, or be "spoken of" by **intentional deduction**, and be reasoned about, as we do when we **attribute** properties to endurants.

5.1.2 Manifest Parts versus Structures

In [49] we covered a notion of *'structures'*. In this primer we shall treat the concept of 'structures' differently We do so by distinguishing between manifest parts and structures.

5.1.2.1 Definitions

Definition 70 . **Manifest Part** By a manifest part we shall understand a part which 'manifests' itself either in a physical, visible manner, "occupying" an 𝔸ℝ𝔼𝔸 or a 𝕍𝕆𝕃𝕌𝕄𝔼 and a ℙ𝕆𝕊𝕀𝕋𝕀𝕆ℕ in 𝕊ℙ𝔸ℂ𝔼, or in a conceptual manner forms an organisation in Your mind ! ■As we have already revealed, endurant parts can be transcendentally deduced into perdurant behaviours – with manifest parts indeed being so.

Definition 71 . **Structure** By a structure we shall understand an endurant concept that allows the domain analyser cum describer to rationally decompose a domain analysis and/or its description into manageable, logically relevant sections, but where these abstract endurants are not further reflected upon in the domain analysis and description. Structures are therefore not transcendentally deduced into perdurant behaviours.

---

[65] By this we mean: You are not just analysing a specific domain, say the one manifested around the corner from where you are, but any instance, anywhere in the world, which satisfies what you have described.

5.1.2.2 Analysis Predicates

Analysis Predicate Prompt 16 `is_manifest:` The method provides the ***domain analysis prompt***:

- `is_manifest` – where `is_manifest`(p) holds if p is to be considered manifest ▪

Analysis Predicate Prompt 17 `is_structure:` The method provides the ***domain analysis prompt***:

- `is_structure` – where `is_structure`(p) holds if p is to be considered a structure ▪

The obvious holds: `is_manifest(p)`$\equiv \neg$`is_structure(p)`.

5.1.2.3 Examples

Example 41 . Manifest Parts and Structures:
We refer to Example 35 on page 46: the Road Transport System. We shall consider all atomic parts: hubs, links and automobiles as being manifest. (They are physical, visible and in $\mathbb{SPACE}$.) We shall consider road nets and aggregates of automobiles as being manifest. Road nets are physical, visible and in $\mathbb{SPACE}$. Aggregates of automobiles are here considered conceptual. The road net manifest part, apart from it aggregates of hubs and links, can be thought of as “representing” a *Department of Roads*[66]. The automobile aggregate apart from its automobiles, can be thought of as “representing” a *Department of Vehicles*[67]. We shall consider hub and link aggregates and hub and link set as structures.

5.1.2.4 Modelling Consequence

In this chapter we introduce internal endurant qualities. If a part is considered manifest then we shall endow that part with all three kinds of internal qualities. If a part is considered a structure then we shall **not** endow that part with any of three kinds of internal qualities.

5.2 Unique Identification

The concept of parts having unique identifiability, that is, that two parts, if they are the same, have the same unique identifier, and if they are not the same, then they have distinct identifiers, that concept is fundamental to our being able to analyse and describe internal qualities of endurants. So we are left with the issue of ‘identity’ !

---

[66] – of some country, state, province, city or other.

[67] See above footnote.

5.2.1 On Uniqueness of Endurants

We therefore introduce the notion of unique identification of part endurants. We assume (i) that all part endurants, e, of any domain E, have *unique identifiers*, (ii) that *unique identifiers* (of part endurants e:E) are *abstract values* (of the *unique identifier* sort UI of part endurants e:E), (iii) that such that distinct part endurant sorts, E_i and E_j, have distinctly named *unique identifier* sorts, say UI_i and UI_j[68], and (iv) that all ui_i:UI_i and ui_j:UI_j are distinct.

Representation of Unique Identifiers: Unique identifiers are abstractions. When we endow two endurants (say of the same sort) distinct unique identifiers then we are simply saying that these two endurants are distinct. We are not assuming anything about how these identifiers otherwise come about. **Identifiability of Endurants:**From a philosophical point of view, and with basis in Kai Sørlander's Philosphy, cf. Paragraph **Identity, Difference and Relations** (Page 14), one can rationally argue that there are many endurants, and that they are unique, and hence uniquely identifiable. From an empirical point of view, and since one may eventually have a software development in mind, we may wonder how unique identifiablity can be accommodated.

Unique identifiability for solid endurants, even though they may be mobile, is straightforward: one can think of many ways of ascribing a unique identifier to any part; solid endurants do not "morph"[69]. Hence one can think of many such unique identification schemas.

Unique identifiability for fluids may seem a bit more tricky. For this monograph we shall not suggest to endow fluids with unique identification. We have simply not experimented with such part-fluids and fluid-parts domains – not enough – to suggest so.

5.2.2 Uniqueness Modelling Tools

The analysis method offers an observer function uid_E which when applied to part endurants, e, yields the unique identifier, *ui*:UI, of e.

Domain Description Prompt 5 `describe_unique_identifier(e)`**:** We can therefore apply the ***domain description prompt***:

- `describe_unique_identifier(e)`

to endurants e:E resulting in the analyser writing down the *unique identifier type and observer* domain description text according to the following schema:

4. `describe_unique_identifier(e)` **Observer**

❝ **Narration:**

[s] ... narrative text on unique identifier sort UI ...[70]
[u] ... narrative text on unique identifier observer uid_E ...
[a] ... axiom on uniqueness of unique identifiers ...

Formalisation:

type
[s] UI
value
[u] uid_E: E → UI ❞

---

[68] This restriction is not necessary, but, for the time, we can assume that it is.

[69] That is, our domain modelling method is not thought of as being applied to the physics situations of endurants going, for example, from states of being solid, via states of melting, to states of fluid.

`is_part(e)` is a prerequisite for `describe_unique_identifier(e)`.

The unique identifier type name, UI above, chosen, of course, by the *domain analyser cum describer*, usually properly embodies the type name, E, of the endurant being analysed and mereology-described. Thus a part of type-name E might be given the mereology type name EI. Generally we shall refer to these names by UI.

Observer Function Prompt 5 **`type_name, type_of, is_:`**
Given *description schema* 5 we have, so-to-speak "in-reverse", that

$$\forall\ e{:}E \bullet \mathrm{uid\_E}(e)=ui \Rightarrow \mathrm{type\_of}(ui)=\eta UI \wedge \mathrm{type\_name}(ui)=UI \wedge \mathrm{is\_UI}(ui)$$

ηUI is a variable of type $\eta\mathbb{T}$. $\eta\mathbb{T}$ is the type of all domain endurant, unique identifier, mereology and attribute type names. By the subsequent UI we refer to the unique identifier type name value of ηUI.

Example 42 . **Unique Identifiers:**

49 We assign unique identifiers to all parts.
50 By a road identifier we shall mean a link or a hub identifier.
51 Unique identifiers uniquely identify all parts.
  a All hubs have distinct [unique] identifiers.
  b All links have distinct identifiers.
  c All automobiles have distinct identifiers.
  d All parts have distinct identifiers.

type
49 H_UI, L_UI, A_UI
50 R_UI = H_UI | L_UI
value
51a uid_H: H → H_UI
51b uid_L: H → L_UI
51c uid_A: H → A_UI

5.2.3 The Unique Identifier State

Given a universe of discourse we can calculate the set of the unique identifiers of all its parts.

```
value
    calculate_all_unique_identifiers: UoD → UI-set
    calculate_all_unique_identifiers(uod) ≡
        let parts = calc_parts({uod})({}) in
        { uid_E(e) | e:E • e ∈ parts } end
```

70

5.2.4 The Unique Identifier State

We can speak of a unique identifier state:

variable
uod := ...
uid$_\sigma$:= discover_uids()
value
discover_uids: UoD → **Unit**
discover_uids(uod) ≡ calculate_all_unique_identifiers(uod)

Example 43 . **Unique Road Transport System Identifiers:**
We can calculate:

52 the *s*et, $h_{ui}s$, of *u*nique *h*ub *i*dentifiers;
53 the *s*et, $l_{ui}s$, of *u*nique *l*ink *i*dentifiers;
54 the *s*et, $r_{ui}s$, of all *u*nique hub and link, i.e., *r*oad *i*dentifiers;
55 the *m*ap, $hl_{ui}m$, from *u*nique *h*ub *i*dentifiers to the *s*et of *u*nique *l*ink *i*dentifiers of the links connected to the zero, one or more identified hubs,
56 the *m*ap, $lh_{ui}m$, from *u*nique *l*ink *i*dentifiersto the *s*et of *u*nique *h*ub *i*identifiers of the two hubs connected to the identified link;
57 the *s*et, $a_{ui}s$, of *u*nique *a*utomobile *i*dentifiers;

value
52 $h_{ui}s$:H_UI-**set** ≡ {uid_H(h)|h:H•h ∈ hs}
53 $l_{ui}s$:L_UI-**set** ≡ {uid_L(l)|l:L•l ∈ ls}
54 $r_{ui}s$:R_UI-**set** ≡ $h_{ui}s \cup l_{ui}s$
55 $hl_{ui}m$:(H_UI $\xrightarrow[m]{}$ L_UI-**set**) ≡
55 [h_ui↦luis|h_ui:H_UI,luis:L_UI-**set**•h_ui∈$h_{ui}s$∧(_,luis,_)=mereo_H(η(h_ui))]
56 $lh_{ui}m$:(L+UI $\xrightarrow[m]{}$ H_UI-**set**) ≡
56 [l_ui↦huis | h_ui:L_UI,huis:H_UI-**set** • l_ui∈$l_{ui}s$ ∧ (_,huis,_)=mereo_L(η(l_ui))]
57 $a_{ui}s$:A_UI-**set** ≡ {uid_A(a)|a:A•a ∈ as}

5.2.5 A Domain Law: Uniqueness of Endurant Identifiers

We postulate that the unique identifier observer functions are about the uniqueness of the postulated endurant identifiers, but how is that guaranteed ? We know, as *"an indisputable law of domains"*, that they are distinct, but our formulas do not guarantee that ! So we must formalise their uniqueness.

All Domain Parts have Unique Identifiers

A Domain Law: 1 All Domain Parts have Unique Identifiers:

58 All parts of a described domain have unique identifiers.

Example 44 . **Uniqueness of Road Net Identifiers:** We must express the following axioms:

59 All hub identifiers are distinct.

60 All link identifiers are distinct.
61 All automobile identifiers are distinct.
62 All part identifiers are distinct.

axiom
59 $\textbf{card}\, hs = \textbf{card}\, h_{ui}s$
60 $\textbf{card}\, ls = \textbf{card}\, l_{ui}s$
61 $\textbf{card}\, as = \textbf{card}\, a_{ui}s$
62 $\textbf{card}\, \{h_{ui}s \cup l_{ui}s \cup bc_{ui}s \cup b_{ui}s \cup a_{ui}s\}$
62 $\quad = \textbf{card}\, h_{ui}s + \textbf{card}\, l_{ui}s + \textbf{card}\, bc_{ui}s + \textbf{card}\, b_{ui}s + \textbf{card}\, a_{ui}s$ ■

We ascribe, in principle, unique identifiers to all endurants whether natural or artefactual. We find, from our many experiments, cf. the *Universes of Discourse* example, Page 36, that we really focus on those domain entities which are artefactual endurants and their behavioural "counterparts".

Example 45 . Rail Net Unique Identifiers:

63 With every rail net unit we associate a unique identifier.
64 That is, no two rail net units have the same unique identifier.
65 Trains have unique identifiers.
66 We let *tris* denote the set of all train identifiers.
67 No two distinct trains have the same unique identifier.
68 Train identifiers are distinct from rail net unit identifiers.

type
63. UI
value
63. uid_NU: NU → UI
axiom
64. ∀ ui_i,ui_j:UI • ui_i = ui_j ≡ uid_NU(ui_i)=uid_NU(ui_j)

5.2.5.1 Part Retrieval

Given the unique identifier, pi, of a part p, but not the part itself, and given the universe-of-discourse (uod) state σ, we can *retrieve* part, p, as follows:

value
  pi:PI, uod:UoD, σ
  retr_part: UI → P
  retr_part(ui) ≡ **let** p:P • p ∈ σ ∧ uid_P(p)=ui **in** p **end**
    pre: ∃ p:P • p ∈ σ ∧ uid_P(p)=ui

5.2.5.2 Unique Identification of Compounds

For structures we do not model their unique identification. But their components, whether the structures are *"Cartesian"* or *"sets"*, may very well be non-structures, hence be uniquely identifiable.

5.3 Mereology

This section is based on Sect. 5.3 of [49, Pages 112–119].

Definition 72 . **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 [65, 37].

5.3.1 Endurant Relations

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

(i) Physically two or more endurants may be topologically either adjacent to one another, like rails of a line, or within an endurant, like links and hubs of a road net, or an atomic part is conjoined to one or more fluids, or a fluid is conjoined to one or more parts. The latter two could also be considered conceptual "adjacencies".

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

5.3.2 Mereology Modelling Tools

When the domain analyser decides that some endurants are related in a specifically enunciated mereology, the analyser has to decide on suitable *mereology types* and *mereology observers* (i.e., endurant relations).

69 We may, to illustration, define a ***mereology type*** of an endurant e:E as a triplet type expression over set of unique [endurant] identifiers.
70 There is the identification of all those endurant sorts $E_{i_1}, E_{i_2}, ..., E_{i_m}$ where at least one of whose properties "`is_of_interest`" to parts e:E.
71 There is the identification of all those sorts $E_{io_1}, E_{io_2}, ..., E_{io_n}$ where at least one of whose properties "`is_of_interest`" to endurants e:E and vice-versa.
72 There is the identification of all those endurant sorts $E_{o_1}, E_{o_2}, ..., E_{o_o}$ for whom properties of e:E "`is_of_interest`" to endurants of sorts $E_{o_1}, E_{o_2}, ..., E_{o_o}$.
73 The mereology triplet sets of unique identifiers are disjoint and are all unique identifiers of the universe of discourse.

The triplet 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 endurants of that domain. We leave out further characterisation of the seemingly vague notion "`is_of_interest`".

type
70 iEI = iEI1 | iEI2 | ... | iEIm
71 ioEI = ioEI1 | ioEI2 | ... | ioEIn
72 oEI = oEI1 | oEI2 | ... | oEIo
69 MT = iEI-**set** × ioEI-**set** × oEI-**set**

axiom
73 ∀ (iset,ioset,oset):MT •
73   **card** iset + **card** ioset + **card** oset = **card** ∪{iset,ioset,oset}
73   ∪{iset,ioset,oset} ⊆ calc_all_unique_identifiers(uod)

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

- `describe_mereology`

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

5. `describe_mereology(e)` Observer

❝ **Narration:**
[t] ... narrative text on mereology type ...
[m] ... narrative text on mereology observer ...
[a] ... narrative text on mereology type constraints ...
Formalisation:
type
[t] MT = $\mathcal{M}(\mathsf{UI}_i,\mathsf{UI}_j,...,\mathsf{UI}_k)$
value
[m] mereo_P: P → MT
axiom [Well–formedness of Domain Mereologies]
[a] $\mathcal{A}$: $\mathcal{A}$(MT) ❞

The mereology type name, MT, chosen of course, by the *domain analyser cum describer*, usually properly embodies the type name, E, of the endurant being analysed and mereology-described. The mereology type expression $\mathcal{M}(\mathsf{UI}_i,\mathsf{UI}_j,...,\mathsf{UI}_k)$ is a type expression over unique identifiers.[71] Thus a part of type-name P might be given the mereology type name MP. $\mathcal{A}$(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 ▪

Example 46 . **Mereology of a Road Net:**

74 The mereology of hubs is a pair: (i) the set of all automobile identifiers[72], and (ii) the set of unique identifiers of the links that it is connected to and the set of all unique identifiers of all automobiles.[73]
75 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.
76 The mereology of an automobile is the set of the unique identifiers of all links and hubs[74].

We presently omit treatment of road net and automobile aggregate mereologies. For road net mereology we refer to Example 75, Item 162 on page 109.

type
74 H_Mer = V_UI-**set**×L_UI-**set**
75 L_Mer = V_UI-**set**×H_UI-**set**
76 A_Mer = R_UI-**set**
value
74 mereo_H: H → H_Mer

---

[71] We refer to Appendix Sect. **C.1.1** on page 173 for more on RSL types.

75 mereo_L: L → L_Mer
76 mereo_A: A → A_Mer

5.3.2.1 Invariance of Mereologies

For mereologies one can usually express some invariants. Such invariants express *"law-like properties"*, facts which are indisputable. We refer to Sect. **5.3.4** on the next page.

Example 47 . Invariance of Road Nets: The observed mereologies must express identifiers of the state of such for road nets:

axiom
74 ∀ (auis,luis):H_Mer • luis⊆$l_{ui}s$ ∧ auis=$a_{ui}s$
75 ∀ (auis,huis):L_Mer • auis=$a_{ui}s$ ∧ huis⊆$h_{ui}s$ ∧ **card** huis = 2
76 ∀ ruis:A_Mer • ruis=$r_{ui}s$

77 For all hubs, h, and links, l, in the same road net,
78 if the hub h connects to link l then link l connects to hub h.

axiom
77 ∀ h:H,l:L • h ∈ hs ∧ l ∈ ls ⇒
77 **let** (_,luis)=mereo_H(h), (_,huis)=mereo_L(l)
78 **in** uid_L(l)∈luis ≡ uid_H(h)∈huis **end**

79 For all links, l, and hubs, h_a,h_b, in the same road net,
80 if the l connects to hubs h_a and h_b, then h_a and h_b both connects to link l.

axiom
79 ∀ h_a,h_b:H,l:L • {h_a,h_b} ⊆ hs ∧ l ∈ ls ⇒
79 **let** (_,luis)=mereo_H(h), (_,huis)=mereo_L(l)
80 **in** uid_L(l)∈luis ≡ uid_H(h)∈huis **end**

5.3.2.2 Deductions made from Mereologies

Once we have settled basic properties of the mereologies of a domain we can, like for unique identifiers, cf. Example 42 on page 64, *"play around"* with that concept: 'the mereology of a domain'.

Example 48 . Consequences of a Road Net Mereology:

81 are there [isolated] units from which one can not "reach" other units ?
82 does the net consist of two or more "disjoint" nets ?
83 et cetera.

We leave it to the reader to narrate and formalise the above properly.

---

[71] This is just another way of saying that the meaning of hub mereologies involves the unique identifiers of all the vehicles that might pass through the hub `is_of_interest` to it.

[72] The link identifiers designate the links, zero, one or more, that a hub is connected to `is_of_interest` to both the hub and that these links is `interested` in the hub.

[73] — that the automobile might pass through

5.3.3 Formulation of Mereologies

The **`observe_mereology`** domain descriptor, Page 68, 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 had their unique identifiers defined.

5.3.4 Fixed and Varying Mereologies

The mereology of parts is not necessarily fixed.

Definition 73 . **Fixed Mereology** By a **fixed mereology** we shall understand a mereology of a part which remains fixed over time.

Definition 74 . **Varying Mereology** By a **varying mereology** we shall understand a mereology of a part which may vary over time.

Example 49 . **Fixed and Varying Mereology:** Let us consider a road net[74]. If hubs and links never change "affiliation", that is: hubs are in fixed relation to zero one or more links, and links are in a fixed relation to exactly two hubs then the mereology of Example 46 on page 68 is a *fixed mereology*. If, on the other hand hubs may be inserted into or removed from the net, and/or links may be removed from or inserted between any two existing hubs, then the mereology of Example 46 on page 68 is a *varying mereology*.

5.3.5 No Fluids Mereology

We comment on our decision, for this monograph, to not endow fluids with mereologies. A first reason is that we "restrict" the concept of mereology to part endurants, that is, to solid endurants – those with "more-or-less" *fixed extents*. Fluids can be said to normally not have fixed extents, that is, they can "morph" from small, fixed into spatially extended forms. For domains of part-fluid conjoins this is particularly true. The fluids in such domains flow through and between parts. Some parts, at some times, embodying large, at other times small amounts of fluid. Some proper, but partial amount of fluid flowing from one part to a next. Et cetera. It is for the same reason that we do not endow fluids with identity. So, for this monograph we decide to not suggest the modelling of fluid mereologies.

5.3.6 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 36, we have found no really interesting such cases. All our experimental case studies appears to focus on the mereology of artefacts. And,

---

[74] cf. Examples 27 on page 37, 35 on page 46, 36 on page 48, 38 on page 51, 41 on page 62, 42 on page 64, 44 on page 65, 45 on page 66, 46 on page 68 and 47 on the preceding page

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

Example 50 . **Rail Net Mereology:** We refer to Example 37 on page 49.

84 A linear rail unit is connected to exactly two distinct other rail net units of any given rail net.
85 A point unit is connected to exactly three distinct other rail net units of any given rail net.
86 A rigid crossing unit is connected to exactly four distinct other rail net units of any given rail net.
87 A single and a double slip unit is connected to exactly four distinct other rail net units of any given rail net.
88 A terminal unit is connected to exactly one distinct other rail net unit of any given rail net.
89 So we model the mereology of a railway net unit as a pair of sets of rail net unit unique identifiers distinct from that of the rail net unit.

```
value
89.  mereo_NU: NU → (UI-set×UI-set)
axiom
89.  ∀ nu:NU •
89.     let (uis_i,uis_o)=mereo_NU(nu) in
89.     case (card uis_i,card usi_o) =
84.         (is_LU(nu) → (1,1),
85.          is_PU(nu) → (1,2) ∨ (2,1),
86.          is_RU(nu)  → (2,2),
87.          is_SU(nu) → (2,2), is_DU(nu) → (2,2),
88.          is_TU(nu)  → (1,0) ∨ (0,1),
89.          _ → chaos) end
89.    ∧ uis_i∩uis_o={}
89.    ∧ uid_NU(nu) ∉ (uis_i ∪ uis_o)
89.     end
```

Figure 5.1 illustrates the mereology of four rail units.

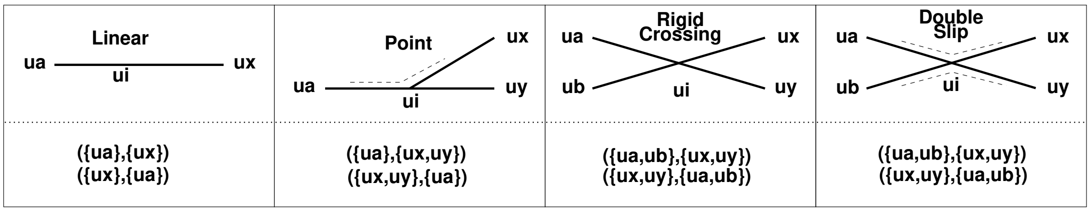


Fig. 5.1 Four Symmetric Rail Unit Mereologies

5.4 Attributes

This section is based on Sect. 5.4 of [49, Pages 119–139].

To recall: there are three sets of ***internal qualities***: unique identifiers, mereologies and attributes. Unique identifiers and mereologies are rather definite kinds of internal endurant qualities; attributes form more "free-wheeling" sets of ***internal qualities***. Whereas, for this monograph, we suggest to not endow fluids with unique identification and mereologies all endurants, i.e., including fluids, are endowed with attributes.

5.4.1 Inseparability of Attributes from Parts and Fluids

Parts and fluids are typically recognised because of their spatial form and are otherwise characterised by their intangible, but measurable attributes. That is, whereas endurants, whether solid (as are parts) or fluids, 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[75], or seen[76], but can be objectively measured[77]. 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 have *the same type of unique identifiers, the same type of mereologies, and the same types of attributes* — with one sort. Thus removing an internal 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) !

We can roughly distinguish between two kinds of attributes: those which can be motivated by **physical** (incl. chemical) **concerns**, and those, which, although they embody some form of 'physics measures', appear to reflect on **event histories**: *"if 'something', ϕ, has 'happened' to an endurant, e_a, then some 'commensurate thing', ψ, has 'happened' to another (one or more) endurants, e_b."* where the *'something'* and *'commensurate thing'* usually involve some 'interaction' between the two (or more) endurants. It can take some reflection and analysis to properly identify endurants e_a and e_b and commensurate events ϕ and ψ. Example 66 shall illustrate the, as we shall call it, **intentional pull** of event histories.

5.4.2 Attribute Modelling Tools

5.4.2.1 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.

---

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

[76] 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.

[77] 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.

5.4.2.2 Concrete Attribute Types

By a *concrete type* shall understand a sort (i.e., a type) which is defined in terms of some type expression: T = $\mathcal{T}$(...). This is referred to below as [=...].

5.4.2.3 Attribute Types and Functions

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

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

- `describe_attributes(e)`.

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:

```
let {$\eta$A$_1$, ..., $\eta$A$_m$} = analyse_attribute_type_names(e) in
❝ Narration:
        [t]   ... narrative text on attribute sorts ...
                some Ais may be concretely defined: [Ai=...]
        [o]   ... narrative text on attribute sort observers ...
        [p]   ... narrative text on attribute sort proof obligations ...
Formalisation:
        type
        [t]   A$_1$[=...] , ...,  A$_m$[=...]
        value
        [o]   attr_A$_1$: E→A$_1$, ..., attr_A$_m$: E→A$_m$
        proof obligation [Disjointness of Attribute Types]
        [p]   $\mathcal{PO}$: let P be any part sort in [the domain description]
        [p]           let a:(A$_1$|A$_2$|...|A$_m$) in is_A$_i$(a) $\neq$ is_A$_j$(a) [i$\neq$i, i,j:[1..m]] end end ❞
end
```

Let A$_1$, ..., A$_n$ be the set of all conceivable attributes of endurants $e{:}E$. (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$_m$, i.e., $m < n$. Across many domain models for *"more-or-less the same"* domain m varies and the attributes, A$_1$, ..., A$_m$, selected for one model may differ from those, A$'_1$, ..., A$'_{m'}$, chosen for another model.

The **type** definitions: A$_1$, ..., A$_m$, inform us that the domain analyser has decided to focus on the distinctly named A$_1$, ..., A$_m$ attributes.[78] The **value** clauses attr_A$_1$:P→A$_1$, ..., attr_A$_n$:P→A$_n$ are then "automatically" given: if an endurant, e:E, has an attribute A$_i$ then there is postulated, "by definition" [eureka] an attribute observer function attr_A$_i$:E→A$_i$ et cetera ▪

---

[78] The attribute type names are chosen by the domain analyser to reflect on domain phenomena.

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

5.4.2.4 Attribute Categories

Michael A. Jackson [107] has suggested a hierarchy of attribute categories: from static to dynamic values – and within the dynamic value category: inert values, reactive values, active values – and within the dynamic active value category: autonomous values, biddable values and programmable values. We now review these attribute value types. The review is based on [107, M.A.Jackson]. *Endurant attributes* are either constant, i.e., **static**, or varying, i.e., **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 ▪

Example 51 . **Static Attributes:** Let us exemplify road net attributes in this and the next examples. And let us assume the following attributes: year of first link construction and link length at that time. We may consider both to be static attributes: The year first established, seems an obvious static attribute and the length is fixed at the time the road was first built.

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 ▪

Example 52 . **Inert Attribute:** And let us now further assume the following link attribute: link name. We may consider it to be an inert attribute: the name is not “assigned” to the link by the link itself, but probably by some road net authority which we are not modelling.

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 either come from outside the domain of interest or from other endurants ▪

Example 53 . **Reactive Attributes:** Let us further assume the following two link attributes: “wear and tear”, respectively “icy and slippery”. We will consider those attributes to be reactive in that automobiles (another part) traveling the link, an external “force”, typically causes the “wear and tear”, respectively the weather (outside our domain) causes the “icy and slippery” property.

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*, or *biddable* or *programmable* attributes ■

Attribute Category 6 By an , 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” ■

Example 54 . **Autonomous Attributes:** We enlarge scope of our examples of attribute categories to now also include automobiles (on the road net). In this example we assume that an automobile is driven by a human [behaviour]. These are some automobile attributes: velocity, acceleration, and moving straight, or turning left, or turning right. We shall consider these three attributes to be autonomous. It is the driver, not the automobile, who decides whether the automobile should drive at constant velocity, including 0, or accelerate or decelerate, including stopping. And it is the driver who decides when to turn left or right, or not turn at all.

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* ■

Example 55 . **Biddable Attributes:** In the context of automobiles these are some biddable attributes: turning the wheel, to drive right at a hub – with the automobile failing to turn right; pressing the accelerator, to obtain a higher speed – with the automobile failing to really gaining speed; pressing the brake, to stop– with the automobile failing to halt ■

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 ■

Example 56 . **Programmable Attribute:** We continue with the automobile on the road net examples. In this example we assume that an automobile includes, as one inseparable entity, “the driver”. These are some automobile attributes: position on a link, velocity, acceleration (incl. deceleration), and direction: straight, turning left, turning right. We shall now consider these three attributes to be programmable.

Figure 5.2 captures an attribute value ontology.
Figure 5.2 hints at three categories of dynamic attributes: **monitorable only**, **biddable** and **programmable** attributes.

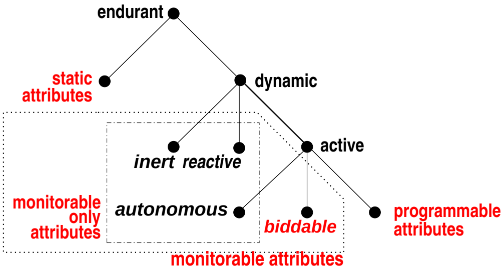


Fig. 5.2 Attribute Value Ontology

Attribute Category 9 By a ***monitorable only attribute***, a:A, `is_monitorable_only_attribute`(a), we shall understand a dynamic active attribute which is either *inert* or *reactive* or *autonomous*.

That is:

```
value
    is_monitorable_only: E → Bool
    is_monitorable_only(e) ≡ is_inert(e) ∨ is_reactive(e) ∨ is_autonomous(e)
```

Example 57 . **Road Net Attributes:**

We treat some attributes of the hubs of a road net.

90 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 *t*o 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.

91 There is a hub state space. It is a set of hub states. The current hub state must be in its state space. The meaning of the hub state space is that its states are all those the hub can attain.

92 Since we can think rationally about it, it can be described, hence we 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. Hub history is an *event history*.

```
type
90 HΣ = (L_UI×L_UI)-set
91 HΩ = HΣ-set
92 H_Traffic = (A_UI|B_UI) →m (𝕋𝕀𝕄𝔼 × VPos)*
axiom
90 ∀ h:H • obs_HΣ(h) ∈ obs_HΩ(h)
92 ∀ ht:H_Traffic,ui:(A_UI|B_UI) • ui ∈ dom ht ⇒ time_ordered(ht(ui))
value
90 attr_HΣ: H → HΣ
91 attr_HΩ: H → HΩ
92 attr_H_Traffic: H → H_Traffic
```

92 time_ordered: ($\mathbb{TIME} \times$ VPos)* → **Bool**
92 time_ordered(tvpl) ≡ ...

In Item 92 we model the time-ordered sequence of traffic as a discrete sampling, i.e., $\underset{m}{\rightarrow}$, rather than as a continuous function, →.

Example 58 . Invariance of Road Net Traffic States: We continue Example 57 on the preceding page.

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

axiom
93 ∀ h:H • h ∈ hs ⇒
93   **let** hσ = attr_HΣ(h) **in**
93   ∀ (l$_{ui}i$,li$_{ui}i'$):(L_UI×L_UI) • (l$_{ui}i$,l$_{ui}i'$) ∈ hσ ⇒ {l$_{ui_i}$,l$'_{ui_i}$} ⊆ l$_{ui}s$ **end**

You may skip Example 59 in a first reading.

Example 59 . Road Transport: Further Attributes:
Links:
We show just a few attributes.

94 There is a link state. It is a set of pairs, (h$_f$,h$_t$), of distinct hub identifiers, where these hub identifiers are in the mereology of the link. The meaning of a link state in which (h$_f$,h$_t$) is an element is that the link is open, **"green"**, for traffic $f$rom hub h$_f$ to hub h$_t$. Link states can have either 0, 1 or 2 elements.
95 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".
96 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.
97 The hub identifiers of link states must be in the set, $h_{ui}s$, of the road net's hub identifiers.

type
94 LΣ = H_UI-**set** [`programmable`, Df.8 Pg.75]
95 LΩ = LΣ-**set** [`static`, Df.1 Pg.74]
96 L_Traffic [`programmable`, Df.8 Pg.75]
96 L_Traffic = (A_UI|B_UI) $\underset{m}{\rightarrow}$ ($\mathbb{T}$×(H_UI×Frac×H_UI))*
96 Frac = **Real**, **axiom** frac:Fract • 0<frac<1
value
94 attr_LΣ: L → LΣ
95 attr_LΩ: L → LΩ
96 attr_L_Traffic: : → L_Traffic
axiom
94 ∀ lσ:LΣ•**card** lσ=2
94 ∀ l:L • obs_LΣ(l) ∈ obs_LΩ(l)
96 ∀ lt:L_Traffic,ui:(A_UI|B_UI)•ui ∈ **dom** ht ⇒ time_ordered(ht(ui))
97 ∀ l:L • l ∈ ls ⇒ **let** lσ = attr_LΣ(l) **in** ∀ (h$_{ui}i$,h$_{ui}i'$):(H_UI×K_UI) •
97   (h$_{ui}i$,h$_{ui}i'$) ∈ lσ ⇒ {h$_{ui_i}$,h$'_{ui_i}$} ⊆ h$_{ui}s$ **end**

Automobiles: We illustrate but a few attributes:

98 Automobiles have static number plate registration numbers.
99 Automobiles have dynamic positions on the road net:

- a either *at a hub* identified by some h_ui,
- b or *on a link*, some *fraction, frac:Fract* down an *identified link, l_ui*, from one of its *identified connecting hub*s, fh_ui, in the direction of the other *identified hub*, th_ui.
- c Fraction is a real properly between 0 and 1.

```
type
98   RegNo                          [static, Df.1 Pg.74]
99   APos == atHub | onLink       [programmable, Df.8 Pg.75]
99a  atHub :: h_ui:H_UI
99b  onLink :: fh_ui:H_UI × l_ui:L_UI × frac:Fract × th_ui:H_UI
99c  Fract = Real
axiom
99c  frac:Fract • 0<frac<1
value
98   attr_RegNo: A → RegNo
99   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. In Items 92 Pg. 76 and 96 Pg. 77, 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[79] ▪

5.4.2.5 Calculating Attribute Category Type Names

One can calculate sets of all attribute type names, of static, so-called monitorable and programmable attribute types of parts and fluids with the following ***domain analysis prompt***s:

- `analyse_attribute_type_names`,
- `sta_attr_types`,
- `mon_attr_types`, and
- `pro_attr_types`.

`analyse_attribute_type_names` applies to parts and yields a set of all attribute names of that part. `mon_attr_types` applies to parts and yields a set of attribute names of *monitorable* attributes of that part.[80]

---

[79] 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.

[80] $\eta\mathbb{A}$ is the type of all attribute types.

Observer Function Prompt 6 `analyse_attribute_types:`

```
value
    analyse_attribute_type_names: P → ηA-set
    analyse_attribute_type_names(p) as {ηA1,ηA,...,ηAm }
```

Observer Function Prompt 7 `sta_attr_types:`

```
value
    sta_attr_types: P → η𝔸×η𝔸×...×η𝔸
    sta_attr_types(p) as (ηA1,ηA2,...,ηAn)
        where: {ηA1,ηA2,...,ηAn} ⊆ analyse_attribute_type_names(p)
               ∧ let anms = analyse_attribute_type_names(p)
                  ∀ anm:η𝔸 • anm ∈ anms \ {ηA1,ηA2,...,ηAn}
                     ⇒ ∼ is_static_attribute{anm}
               ∧ ∀ anm:η𝔸 • anm ∈ {ηA1,ηA2,...,ηAn}
                     ⇒ is_static_attribute{anm} end
```

Observer Function Prompt 8 `mon_attr_types:`

```
value
    mon_attr_types: P → η𝔸×η𝔸×...×η𝔸
    mon_attr_types(p) as (ηA1,ηA2,...,ηAn)
        where: {ηA1,ηA2,...,ηAn} ⊆ analyse_attribute_type_names(p)
               ∧ let anms = analyse_attribute_type_names(p)
                  ∀ anm:η𝔸 • anm ∈ anms \ {ηA1,ηA2,...,ηAn}
                     ⇒ ∼ is_monitorable_attribute{anm}
               ∧ ∀ anm:η𝔸 • anm ∈ {ηA1,ηA2,...,ηAn}
                     ⇒ is_monitorable_attribute{anm} end
```

Observer Function Prompt 9 `pro_attr_types:`

```
value
    pro_attr_types: P → η𝔸×η𝔸×...×η𝔸
    pro_attr_types(p) as (ηA1,ηA2,...,ηAn)
        where: {ηA1,ηA2,...,ηAn} ⊆ analyse_attribute_type_names(p)
               ∧ let anms = analyse_attribute_type_names(p)
                  ∀ anm:η𝔸 • anm ∈ anms \ {ηA1,ηA2,...,ηAn}
                     ⇒ ∼ is_monitorable_attribute{anm}
               ∧ ∀ anm:η𝔸 • anm ∈ {ηA1,ηA2,...,ηAn}
                     ⇒ is_monitorable_attribute{anm} end
```

Some comments are in order. The analyse_attribute_type_names function is, as throughout, meta-linguistic, that is, informal, not-computable, but decidable by the domain analyser cum describer. Applying it to a part or fluid yields, at the discretion of the domain analyser cum describer, a set of attribute type names "freely" chosen by the domain analyser cum describer. The sta_attr_type_names, the mon_attr_type_names, and the pro_attr_type_names functions are likewise meta-linguistic; their definition here relies on the likewise meta-linguistic is_static, is_monitorable and is_programmable analysis predicates.

5.4.2.6 Calculating Attribute Values

Let (ηA1, ηA2, ... , ηAn) be a grouping of attribute types for part p (or fluid f). Then (attr_A1(p), attr_A2(p), ... ,attr_An(p)) (respectively f) yields (a1, a2, ... , an), the grouping of values for these attribute types.

We can "formalise" this conversion:

```
value
    types_to_values: $\eta\mathbb{A}_1 \times \eta\mathbb{A}_2 \times ... \times \eta\mathbb{A}_n \to A_1 \times A_2 \times ... \times A_n$
```

5.4.2.7 Calculating Attribute Names

The meta-linguistic, i.e., "outside" RSL proper, name for attribute type names is introduced here as $\eta\mathbb{A}$.

100 Given endurant e we can *meta-linguistically*[81] calculate names for its *static* attributes.
101 Given endurant e we can *meta-linguistically* calculate name for its *monitorable* attributes attributes.
102 Given endurant e we can *meta-linguistically* calculate names for its *programmable* attributes.
103 These four sets make up all the attributes of endurant e.

The type names ST, MA, PT designate mutually disjoint sets, ST, of names of static attributes, sets, MA, of names of monitoriable, i.e., monitorable-only and biddable, attributes, sets, PT, of names of programmable, i.e., fully controllable attributes.

```
type
100  ST = $\eta\mathbb{A}$-set
101  MA = $\eta\mathbb{A}$-set
102  PT = $\eta\mathbb{A}$-set
value
100  stat_attr_types: E → ST
101  moni_attr_types: E → MA
102  prgr_attr_types: E → PT

axiom
103  ∀ e:E •
100  let stat_nms = stat_attr_types(e),
101      moni_nms = moni_attr_types(e),
102      prgr_nms = prgr_types(e) in
103  card stat_nms + card moni_nms + card prgr_nms
103  = card(stat_nms ∪ mon_nms ∪ prgr_nms) end
```

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

104 Given endurant e we can *meta-linguistically* calculate its static attribute values, stat_attr_vals;
105 given endurant e we can *meta-linguistically* calculate its monitorable-only attribute values, moni_attr_vals; and
106 given endurant e we can *meta-linguistically* calculate its programmable attribute values, prgr_attr_vals.

---

[81] 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.

The type names sa1, ..., pap refer to the types denoted by the corresponding types name nsa1, ..., npap.

```
value
104  stat_attr_vals: E → SA1×SA2×...×SAs
104  stat_attr_vals(e) ≡
104    let {nsa1,nsa2,...,nsas} = stat_attr_types(e) in
104    (attr_sa1(e),attr_sa2(e),...,attr_sas(e)) end

105  moni_attr_vals: E → MA1×MA2×...×MAm
105  moni_attr_vals(e) ≡
105    let {nma1,nma2,...,nmam} = moni_attr_types(e) in
105    (attr_ma1(e),attr_ma2(e),...,attr_mam(e)) end


106  prgr_attr_vals: E → PA1×PA2×...×PAp
106  prgr_attr_vals(e) ≡
106    let {npa1,npa2,...,npap} = prgr_attr_types(e) in
106    (attr_pa1(e),attr_pa2(e),...,attr_pap(e)) end
```

The "ordering" of type values, (attr_sa1(e),...,attr_sas(e)), (attr_ma1(e),...,attr_mam(e)), et cetera, is arbitrary.

5.4.3 Operations on Monitorable Attributes of Parts

We remind the reader of the notions of states in general, Sect. **4.7** and updateable states, Sect. **4.7.2** on page 55. For every domain description there possibly is an updateable state. The is such a state if there is at least one part with at least one monitorable attribute. Below, as in Sect. **4.7.2**, we refer to the updateable states as σ.

Given a part, p, with attribute A, the simple operation attr_A(p) thus yields the value of attribute A for that part. But what if, what we have is just the global state σ, of the set of all monitorable parts of a given universe-of-discourse, uod, the unique identifier, uid_P(p), of a part of σ, and the name, ηA, of an attribute of p? Then how do we ascertain the attribute value for A of p, and, for *biddable* attributes A, "update" p, in σ, to some A value? Here is how we express these two issues.

5.4.3.1 Evaluation of Monitorable Attributes

107 Let pi:PI be the unique identifier of any part, p, with monitorable attributes, let A be a monitorable attribute of p, and let ηA be the name of attribute A.
108 Evaluation of the [current] attribute A value of p is defined by function read_A_from_P – retr_part(pi) is defined in Sect. **5.2.5.1** on page 66.

```
value
107.   pi:PI, a:A, ηA:η𝕋

108.   read_A_from_P: PI × 𝕋 → read σ
108.   read_A(pi,ηA) ≡ attr_A(retr_part(pi))
```

5.4.3.2 Update of Biddable Attributes

109 The update of a monitorable attribute A, with attribute name ηA of part p, identified by pi, to a new value **write**s to the global part state σ.
110 Part p is retrieved from the global state.
111 A new part, p′ is formed such that p′ is like part p:

- a same unique identifier,
- b same mereology,
- c same attributes values,
- d except for A.

112 That new p' replaces p in σ.

```
value
107.  σ, a:A, pi:PI, ηA:η𝕋

109.  update_P_with_A: PI × A × η𝕋 → write σ
109.  update_P_with_A(pi,a,ηA) ≡
110.     let p = retr_part(pi) in
111.     let p′:P •
111a.        uid_P(p′)=pi
111b.        ∧ mereo_P(p)=mereo_P(p′)
111c.        ∧ ∀ ηA’ in analyse_attribute_type_names(p) \ {ηA}
111c.                ⇒ attr_A(p)=attr_A(p′)
111d.        ∧ attr_A(p′)=a in
112.     σ := σ \ {p} ∪ {p′}
109.     end end
```

5.4.3.3 Stationary and Mobile Attributes

Endurants are either **stationary** or **mobile**.[82]

Definition 75 . **Stationary** An endurant is said to be stationary if it never moves ▪

Being stationary is a static attribute.

Analysis Predicate Prompt 18 `is_stationary:` The method provides the ***domain analysis prompt***:

- `is_stationary` – where `is_stationary(`e`)` holds if e is to be considered stationary ▪

Example 60 . **Stationary Endurants:** Examples of stationary endurants could be: (i) road hubs and links; (ii) container terminal stacks; (iii) pipeline units; and (iv) sea, lake and river beds ▪

Definition 76 . **Mobile** An endurant is said to be mobile if it is capable of being moved – whether by its own, or otherwise ▪

---

[82] This section was added on Sept. 17, 2022 !

Being mobile is a static attribute.

Analysis Predicate Prompt 19 `is_mobile:` The method provides the ***domain analysis prompt***:

- `is_mobile` – where `is_mobile(`e`)` holds if e is to be considered mobile ■

Example 61 . **Mobile Endurants:** Examples of mobile endurants are: (i) automobiles; (ii) container terminal vessels, containers, cranes and trucks; (iii) pipeline oil (or gas, or water, ...); (iv) sea, lake and river water ■

Being stationary or mobile is an attribute of any manifest endurant. Foe every manifest endurant, e, it is the case that `is_stationary(e)≡~is_mobile(e)`.

•••

Being stationary or, vice-versa, being mobile is often **tacitly assumed.** Having external or internal qualities of a certain kind is often also tacitly assumed. A major point of the domain analysis & description approach, of this primer, is to help the domain analyser cum describer – the domain engineer cum researcher – to unveil as many, if not all, these qualities. **Tacit understanding** would not be a common problem was it not for us to practice it "excessively" !

5.5 SPACE and TIME

The two concepts: **space** and **time** are not attributes of entities. In fact, they are not internal qualities of endurants. They are universal qualities of any world. As argued in Sect. **2.4.9** on page 17, SPACE and TIME are unavoidable concepts of any world. But we can ascribe spatial attributes to any concrete, manifest endurant. And we can ascribe attributes to endurants that record temporal concepts.

5.5.1 SPACE

Space is just there. So we do not define an observer, observe_space. For us – bound to model mostly artefactual worlds on this earth – there is but one space. Although SPACE, as a type, could be thought of as defining more than one space we shall consider these to be isomorphic ! SPACE is considered to consist of (an infinite number of) POINTs.

113 We can assume a point observer, observe_POINT, is a function which applies to endurants, e, and yield a point, pt : POINT

113. observe_POINT: E → POINT

At which "point" of an endurant, e, observe_POINT(e), is applied, or which of the (infinitely) many points of an endurant E, observe_POINT(e), yields we leave up to the domain analyser cum describer to decide !

We suggest, besides POINTs, the following spatial attribute possibilities:

114 EXTENT as a dense set of POINTs;

115 Volume, of concrete type, for example, m^3, as the "volume" of an $\mathbb{EXTENT}$ such that
116 $\mathbb{SURFACE}$s as dense sets of $\mathbb{POINT}$s have no volume, but an
117 Area, of concrete type, for example, m^2, as the "area" of a dense set of $\mathbb{POINT}$s;
118 $\mathbb{LINE}$ as dense set of $\mathbb{POINT}$s with no volume and no area, but
119 Length, of concrete type, for example, m.

For these we have that

120 the *intersection*, $\bigcap$, of two $\mathbb{EXTENT}$s is an $\mathbb{EXTENT}$ of possibly nil Volume,
121 the intersection, $\bigcap$, of two $\mathbb{SURFACE}$s may be either a possibly nil $\mathbb{SURFACE}$ or a possibly nil $\mathbb{LINE}$, or a combination of these.
122 the intersection, $\bigcap$, of two $\mathbb{LINE}$s may be either a possibly nil $\mathbb{LINE}$ or a $\mathbb{POINT}$.

Similarly we can define

123 the *union*, $\bigcup$, of two not-disjoint $\mathbb{EXTENT}$s,
124 the *union*, $\bigcup$, of two not-disjoint $\mathbb{SURFACE}$s,
125 the *union*, $\bigcup$, and of two not-disjoint $\mathbb{LINE}$s.

and:

126 the *[in]equality*, $\neq, =$, of pairs of $\mathbb{EXTENT}$, pairs of $\mathbb{SURFACE}$s, and pairs of $\mathbb{LINE}$s.

We invite the reader to first first express the signatures for these operations, then their preconditions, and finally, being courageous, appropriate fragments of axiom systems.
We leave it up to the reader to introduce, and hence define, functions that add, subtract, compare, etc., $\mathbb{EXTENT}$s, $\mathbb{SURFACE}$s, $\mathbb{LINE}$s, etc.

5.5.2 Mathematical Models of Space

Figure 5.3 on the facing page diagrams some mathematical models of space. We shall hint[83] at just one of these spaces.

5.5.2.1 Metric Spaces

Metric Space

Axiom System 1
A metric space is an ordered pair (M,d) where M is a set and d is a metric on M, i.e., a function:

$$d : M \times M \rightarrow \textbf{Real}$$

such that for any $x,y,z \in M$, the following holds:

$$d(x,y) = 0 \equiv x = y \quad \text{identity of indiscernibles} \tag{5.1}$$

$$d(x,y) = d(y,x) \quad \text{symmetry} \tag{5.2}$$

$$d(x,z) \leq d(x,y) + d(y,z) \quad \text{sub-additivity or triangle inequality} \tag{5.3}$$

Given the above three axioms, we also have that $d(x,y) \geq 0$ for any $x,y \in M$. This is deduced as follows:

---

[83] Figure 5.3 on the next page is taken from `https://en.wikipedia.org/wiki/Space_(mathematics)`.

$$d(x,y)+d(y,x) \geq d(x,x) \quad \text{triangle inequality} \tag{5.4}$$

$$d(x,y)+d(y,x) \geq d(x,x) \quad \text{by symmetry} \tag{5.5}$$

$$2d(x,y) \geq 0 \quad \text{identity of indiscernibles} \tag{5.6}$$

$$d(x,y) \geq 0 \quad \text{non-negativity} \tag{5.7}$$

The function d is also called distance function or simply distance. Often, d is omitted and one just writes M for a metric space if it is clear from the context what metric is used.

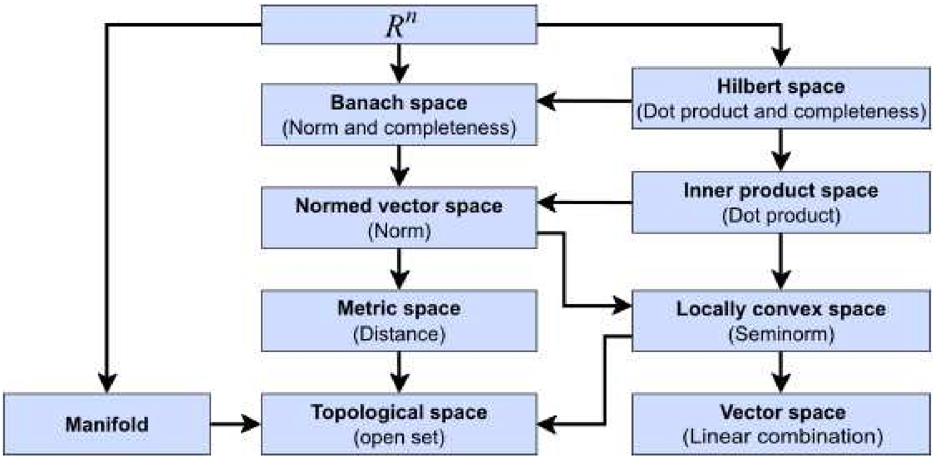


Fig. 5.3 Variety of Abstract Spaces. An arrow from space A to space B implies that A is also a kind of B.

5.5.3 TIME

a moving image of eternity;
the number of the movement in respect of the before and the after;
the life of the soul in movement as it passes
from one stage of act or experience to another;
a present of things past: memory,
a present of things present: sight,
and a present of things future: expectations[84]

This thing all things devours:
Birds, beasts, trees, flowers;
Gnaws iron, bites steel,
Grinds hard stones to meal;
Slays king, ruins town,
And beats high mountain down.[85]

Concepts of time continue to fascinate philosophers and scientists [153, 78, 119, 126, 130, 131, 132, 133, 134, 135, 137] and [80].

---

[84] Quoted from [4, Cambridge Dictionary of Philosophy]

[85] J.R.R. Tolkien, The Hobbit

`J.M.E. McTaggart` (1908, [119, 78, 137]) discussed theories of time around the notions of **"A-series":** with concepts like "past", "present" and "future", and **"B-series":** has terms like "precede", "simultaneous" and "follow". `Johan van Benthem` [153] and `Wayne D. Blizard` [63, 1980] relates abstracted entities to spatial points and time. A recent computer programming-oriented treatment is given in [80, `Mandrioli et al.`, 2013].

5.5.3.1 Time Motivated Philosophically

Definition 77 . **Indefinite Time** *We motivate, repeating from Sect.* **2.4.9.2***, the abstract notion of time as follows. 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* ▪

Definition 78 . **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 62 . **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 96 on page 77 and (ii) the time stamped hub traffic, cf. Item 92 on page 76* ▪

5.5.3.2 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 [80]. Similarly we shall not be concerned with any representation of time intervals.[86]

127 So there is an abstract type $\mathbb{T}$*ime*,
128 and an abstract type $\mathbb{TI}$: $\mathbb{T}$*ime*$\mathbb{I}$*nterval*.
129 There is no $\mathbb{T}$*ime* origin, but there is a "zero" $\mathbb{TI}$me interval.
130 One can add (subtract) a time interval to (from) a time and obtain a time.
131 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.
132 One can subtract a time from another time obtaining a time interval respecting that the subtrahend is smaller than or equal to the minuend.
133 One can multiply a time interval with a real and obtain a time interval.
134 One can compare two times and two time intervals.

```
type
127  𝕋
128  𝕋𝕀
value
129  0:𝕋𝕀
130  +,−: 𝕋 × 𝕋𝕀 → 𝕋
131  +,−: 𝕋𝕀 × 𝕋𝕀 →̃ 𝕋𝕀
132  −: 𝕋 × 𝕋 → 𝕋𝕀
133  ∗: 𝕋𝕀 × Real → 𝕋𝕀
134  <,≤,=,≠,≥,>: 𝕋 × 𝕋 → Bool
```

---

[86] – 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.

134 $<,\leq,=,\neq,\geq,>$: 𝕋𝕀 × 𝕋𝕀 → **Bool**

axiom
130 ∀ t:𝕋 • t+**0** = t

5.5.3.3 Temporal Observers

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

type
135 𝕋
value
135 **record**_𝕋𝕀𝕄𝔼(): **Unit** → 𝕋

The time recorder applies to nothing and yields a time. **record**_𝕋𝕀𝕄𝔼() can only occur in action, event and behavioural descriptions.

5.5.3.4 "Soft" and "Hard" Real-time

We loosely identify a spectrum of from "soft" to "hard" temporalities — through some informally worded texts. On that background we can introduce the term 'real-time'. And hence distinguish between 'soft' and 'hard' real-time issues. From an example of trying to formalise these in RSL, we then set the course for this chapter.

5.5.3.4.1 Soft Temporalities

You have often wished, we assume, that *"your salary never goes down, say between your ages of 25 to 65"*.

How to express that?

Taking into account other factors, you may additionally wish that *"your salary goes up."*

How do we express that?

Taking also into account that your job is a seasonal one, we may need to refine the above into *"between un-employments your salary does not go down"*.

How now to express that?

5.5.3.4.2 Hard Temporalities

The above quoted ("...") statements may not have convinced you about the importance of speaking precisely about time, whether narrating or formalising.

So let's try some other examples:

"The alarm clock must sound exactly at 6 am unless someone has turned it off sometime between 5am and 6 am the same morning."

"The gas valve must be open for exactly 20 seconds every 60 seconds."

"The sum total of time periods — during which the gas valve is open and there is no flame consuming the gas — must not exceed one twentieth of the time the gas valve is open."

"The time between pressing an elevator call button on any floor and the arrival of the cage and the opening of the cage door at that floor must not exceed a given time $t_{arrival}$*"*.

The next sections will hint at ways and means of speaking of time.

5.5.3.4.3 Soft and Hard Real-time

The informally worded temporalities of “soft real-time” can be said to involve time in a very “soft” way:

No explicit times (eg., 15:45:00), deadlines (eg., *“27’th February 2004”*), or time intervals (eg., *“within 2 hours”*), were expressed.

The informally worded temporalities of “hard real-time”, in contrast, can be said to involve time in a “hard” way: Explicit times were mentioned.

For pragmatic reasons, we refer to the former examples, the former “invocations” of ‘temporality’, as being representative of soft real-time, whereas we say that the latter invocations are typical of hard real-time.

Please do not confuse the issue of soft versus hard real-time: It is as much hard real-time if we say that something must happen two light years and five seconds from tomorrow at noon!

Example 63 . **Soft Real-Time Models Expressed in Ordinary RSL Logic:** Let us assume a salary data base SDB which at any time records your salary. In the conventional way of modelling time in RSL we assume that SDB maps time into Salary:

```
type
   Time, Sal
   SDB = Time →m Sal
value
   hi: (Sal×Sal)|(Time×Time) → Bool
   eq: (Sal×Sal)|(Time×Time) → Bool
   lo: (Sal×Sal)|(Time×Time) → Bool
axiom
   ∀ σ:SDB,t,t′:Time • {t,t′}⊆dom σ∧hi(t′,t)⇒∼lo(σ(t′),σ(t))
   ∀ t,t′:Time •
      (hi(t′,t)≡∼(eq(t′,t)∨lo(t′,t))) ∧
      (lo(t′,t)≡∼(eq(t′,t)∨hi(t′,t))) ∧
      (eq(t′,t)≡∼(lo(t′,t)∨hi(t′,t))) ... /∗ same for Sal ∗/
```

Example 64 . **Hard Real-Time Models Expressed in “Ordinary” RSL Logic:** To express hard real-time using just RSL we must assume a demon, a process which represents the clock:

```
type
   𝕋 = Real
value
   time: Unit → 𝕋
   time() as t
axiom
   time() ≠ time()
```

The axiom is informal: It states that no two invocations of the time function yields the same value. But this is not enough. We need to express that “immediately consecutive” invocations of the time function yields “adjacent” time points. 𝕋 provides a linear model of real-time.

```
variable
   t1,t2 : 𝕋
axiom
```

```
□ (t1 := time();
    t2 := time();
    t2 – t1 = /∗ infinitesimally small time interval: 𝕋𝕀∗/ ∧
    t2 > t1 ∧ ~∃ t:𝕋• t1 < t < t2 )
```

𝕋𝕀 provides a linear model of intervals of real-time.[87] The □ operator is here the "standard" RSL modal operator over states: Let P be a predicate involving globally declared variables. Then $\Box P$ asserts that P holds in any state (of these variables). But even this is not enough. Much more is needed ▪

5.6 Intentional Pull

In the next section we shall encircle the 'intention' concept by extensively quoting from Kai Sørlander's Philospohy [145, 146, 147, 148].

Intentionality[88] "expresses" conceptual, abstract relations between otherwise, or seemingly unrelated entities.

Intentional properties of a domain is not an internal quality of any (pair or group of) entities. They are potential, universal qualities of any world.

5.6.1 Issues Leading Up to Intentionality

5.6.1.1 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."

5.6.1.2 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 occur 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 movements. The first we call plants, the second we call animals."

---

[87] Of course, we really do not need make a distinction between 𝕋 and 𝕋𝕀, The former tries to model a real-time since time immemorial, i.e., the creation of the universe. If we always work with a time axis from "that started recently", i.e., a relative one, then we can "collapse" 𝕋 and 𝕋𝕀 into just 𝕋.

[88] The Oxford English Dictionary [116] characterises intentionality as follows: *"the quality of mental states (e.g. thoughts, beliefs, desires, hopes) which consists in their being directed towards some object or state of affairs"*.

5.6.1.3 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 organs *sensing among others the immediate purpose of its movement; they must have* means of motion *so that it can move; and they must have* instincts, incentives *and* feelings *as causal conditions that what it senses can drive it to movements. And all of this in accordance with the laws of physics."*

5.6.1.4 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 genomes. That is, animals are built from genomes which give them the* inner determination *to such building blocks for* instincts, *incentives and* feelings. *Similar kinds of deduction can be carried out with respect to plants. Transcendentally one can deduce basic principles of evolution but not its details."*

5.6.1.5 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."*

"Animals with higher social interaction uses signs, 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.*"*

5.6.1.6 Knowledge

"Humans must be conscious *of having knowledge of its concrete situation, and as such that humans can have knowledge about what they feel and eventually that humans can know whether what they feel is true or false. Consequently* a human can describe his situation correctly.*"*

5.6.1.7 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 further develop the theme of *living species: plants and animals*, thus excluding, most notably *humans*, in this chapter. We claim that the present chapter, due to its foundation in Kai Sørlander's Philosophy, provides a firm foundation within which we, or others, can further develop this theme: *analysis & description of living species.*

●●●

5.6.2 Intentionality

Intentionality as a philosophical concept is defined by the `Stanford Encyclopedia of Philosophy`[89] as *"the power of minds to be about, to represent, or to stand for, things, properties and states of affairs."*

5.6.2.1 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 65 . **Double Bookkeeping:** A classical example of intentional pull is found in double bookkeeping which states that every financial transaction has equal and opposite effects in at least two different accounts. It is used to satisfy the accounting equation: Assets = Liabilities + Equity. The intentional pull is then reflected in commensurate postings, for example: either in both debit and passive entries or in both credit and passive entries.

When a compound artefact is modelled as put together with a number of distinct sort endurants then it does have an intentionality and the components' individual intentionalities does, i.e., shall relate to that. The composite road transport system has intentionality of the road serving the automobile part, and the automobiles have the intent of being served by the roads, across "a divide", and vice versa, the roads of serving the automobiles.

Natural endurants, for example, rivers, lakes, seas[90] and oceans become, in a way, artefacts when mankind use them for transport; natural gas becomes an artefact when drilled for, exploited and piped; and harbours make no sense without artefactual boats sailing on the natural water.

5.6.2.2 The Type **Intent**

This, perhaps vague, concept of intentionality has yet to be developed into something of a theory. Despite that this is yet to be done, we shall proceed to define an *intentionality analysis function*. First we postulate a set of **intent designators**. An *intent designator* is really a further undefined quantity. But let us, for the moment, think of them as simple character strings, that is, literals, for example `""transport", "eating", "entertainment"`,etc.

type Intent

---

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

[90] Seas are smaller than oceans and are usually located where the land and ocean meet. Typically, seas are partially enclosed by land. The Sargasso Sea is an exception. It is defined only by ocean currents [oceanservice.noaa.gov/facts/oceanorsea.html].

5.6.2.3 Intentionalities

Observer Function Prompt 10 `analyse_intentionality:`
The domain analyser analyses an endurant as to the finite number of intents, zero or more, with which the analyser judges the endurant can be associated. The method provides the ***domain analysis prompt***:

- `analyse_intentionality` directs the domain analyser to observe a set of intents.

  value analyse_intentionality(e) ≡ {i_1,i_2,...,i_n}⊆Intent

Example 66 . **Intentional Pull: Road Transport:**
We simplify the link, hub and automobile histories – aiming at just showing an essence of the intentional pull concept.

136 With links, hubs and automobiles we can associate history attributes:
  a link history attributes time-stamped records, as an ordered list, the presence of automobiles;
  b hub history attributes time-stamped records, as an ordered list, the presence of automobiles; and
  c automobile history attributes time-stamped records, as an ordered list, their visits to links and hubs.

type
136a. LHist = AI $\overrightarrow{m}$ $\mathbb{TIME}^*$
136b. HHist = AI $\overrightarrow{m}$ $\mathbb{TIME}^*$
136c. AHist = (LI|HI) $\overrightarrow{m}$ $\mathbb{TIME}^*$

value
136a. attr_LHist: L → LHist
136b. attr_HHist: H → HHist
136c. attr_AHist: A → AHist

5.6.2.4 Wellformedness of Event Histories

Some observations must be made with respect to the above modelling of time-stamped event histories.

137 Each $\tau_\ell : \mathbb{TIME}^*$ is an indefinite list. We have not expressed any criteria for the recording of events: *all the time, continuously* ! (?)
138 Each list of times, $\tau_\ell : \mathbb{TIME}^*$, is here to be in decreasing, *continuous* order of times.
139 Time intervals from when an automobile enters a link (a hub) till it first time leaves that link (hub) must not overlap with other such time intervals for that automobile.
140 If an automobile leaves a link (a hub), at time τ, then it may enter a hub (resp. a link) and then that must be at time τ' where τ' is some infinitesimal, sampling time interval, quantity larger that τ. Again we refrain here from speculating on the issue of sampling !
141 Altogether, ensembles of link and hub event histories for any given automobile define routes that automobiles travel across the road net. Such routes must be in the set of routes defined by the road net.

As You can see, there is enough of interesting modelling issues to tackle !

5.6.2.5 Formulation of an Intentional Pull

142 An *intentional pull* of any road transport system, rts, is then if:

  a for any automobile, a, of rts, on a link, ℓ (hub, h), at time τ,
  b then that link, ℓ, (hub h) “records” automobile a at that time.

143 and:

  c for any link, ℓ (hub, h) being visited by an automobile, a, at time τ,
  d then that automobile, a, is visiting that link, ℓ (hub, h), at that time.

```
axiom
142a.  ∀ a:A • a ∈ as ⇒
142a.     let ahist = attr_AHist(a) in
142a.     ∀ ui:(LI|HI) • ui ∈ dom ahist ⇒
142b.        ∀ τ:TIME • τ ∈ elems ahist(ui) ⇒
142b.           let hist = is_LI(ui) → attr_LHist(retr_L(ui))(σ),
142b.                             _ → attr_HHist(retr_H(ui))(σ) in
142b.           τ ∈ elems hist(uid_A(a)) end end
143.   ∧
143c.  ∀ u:(L|H) • u ∈ ls∪hs ⇒
143c.     let uhist = attr(L|H)Hist(u) in
143d.     ∀ ai:AI • ai ∈ dom uhist ⇒
143d.        ∀ τ:TIME • τ ∈ elems uhist(ai) ⇒
143d.           let ahist = attr_AHist(retr_A(ai))(σ) in
143d.           τ ∈ elems uhist(ai) end end
```

Please note, that *intents* are not [thought of as] attributes. We consider *intents* to be a fourth, a comprehensive internal quality of endurants. They, so to speak, govern relations between the three other internal quality of endurants: the unique identifiers, the mereologies and the attributes. That is, they predicate them, “arrange” their comprehensiveness. Much more should be said about intentionality. It is a truly, I believe, worthy research topic of its own ▪

Example 67 . **Aspects of Comprehensiveness of Internal Qualities:** Let us illustrate the issues “at play” here.

- Consider a road transport system uod.
  - ∞ Applying `analyse_intentionality`(uod) may yield the set {`"transport", ...`}.
- Consider a financial service industry, fss.
  - ∞ Applying `analyse_intentionality`(fss) may yield the set {`"interest on deposit", ...`}.
- Consider a health care system, hcs.
  - ∞ Applying `analyse_intentionality`(hcs) may yield the set {`"cure diseases", ...`}.

What these analyses of intentionality yields, with respect to expressing intentional pull, is entirely of the discretion of the domain analysis & description ▪

We bring the above example, Example 67, to indicate, as the name of the example reveals, “Aspects of Comprehensiveness of Internal Qualities”. That the various components of artefactual systems relate in – further to be explored – ways. In this respect, performing domain analysis & description is not only an engineering pursuit, but also one of research. We leave it to the readers to pursue this research aspect of domain analysis & description.

5.6.3 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*.

5.6.4 Assignment of Attributes

So what can we deduce from the above, almost three pages ?

The attributes of **natural parts** and **natural fluids** are generally of such concrete types – expressible as some **real** with a dimension[91] of the International System of Units: `https://-physics.nist.gov/cuu/Units/units.html`. Attribute values usually enter into *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 68 . **Intentional Pull: General Transport:** 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.

5.6.5 Galois Connections

Galois Theory was first developed by Évariste Galois [1811-1832] around 1830[92]. Galois theory emphasizes a notion of **Galois connections**. We refer to standard textbooks on Galois Theory, e.g., [151, 2009].

5.6.5.1 Galois Theory: An Ultra-brief Characterisation

To us, an essence of Galois connections can be illustrated as follows:

- Let us observe[93] properties of a number of endurants, say in the form of attribute types.
- Let the function $\mathcal{F}$ map sets of entities to the set of common attributes.
- Let the function $\mathcal{G}$ map sets of attributes to sets of entities that all have these attributes.
- ($\mathcal{F}$,$\mathcal{G}$) is a Galois connection
  - if, when including more entities, the common attributes remain the same or fewer, and
  - if when including more attributes, the set of entities remain the same or fewer.
  - ($\mathcal{F}$,$\mathcal{G}$) is monotonously decreasing.

---

[91] Basic units are $m$eter, $k$ilo$g$ram, $s$econd, Ampere, Kelvin, mole, and $cand$ela. Some derived units are: Newton: $kg{\times}m{\times}s^{-2}$, W$eb$er: $kg \times m^2 \times s^{-2} \times A^{-1}$, etc.

[92] en.wikipedia.org/wiki/Galois_theory

[93] The following is an edited version of an explanation kindly provided by Asger Eir, e-mail, June 5, 2020 [75, 76, 53].

Example 69 . **LEGO Blocks:** We[94] have

- There is a collection of `LEGO™` blocks.
- From this collection, A, we identify the **red** square blocks, e.
- That is $\mathcal{F}(A)$ is $B = \{$attr_Color(e) = **red**,attr_Form(e)=**square**$\}$.
- We now add all the **blue** square blocks.
- And obtain A'.
- Now the common properties are their **squareness**: $\mathcal{F}(A')$ is $B' = \{$attr_Form(e)=**square**$\}$.
- More blocks as argument to $\mathcal{F}$ yields fewer or the same number of properties.
- The more entities we observe, the fewer common attributes they possess ▪

Example 70 . **Civil Engineering: Consultants and Contractors:** Less playful, perhaps more seriously, and certainly more relevant to our endeavour, is this next example.

- Let X be the set of civil engineering, i.e., building, consultants, i.e., those who, like architects and structural engineers design buildings – of whatever kind.
- Let Y be the set of building contractors, i.e., those firms who actually implement, i.e., build to, those designs.
- Now a subset, $X_{bridges}$ of X, contain exactly those consultants who specialise in the design of bridges, with a subset, $Y_{bridges}$, of Y capable of building bridges.
- If we change to a subset, $X_{bridges,tunnels}$ of X, allowing the design of both bridges **and** tunnels, then we obtain a corresponding subset, $Y_{bridges,tunnels}$, of Y.
- So when
  - we enlarge the number of properties from 'bridges' to 'bridges and tunnels',
  - we reduce, most likely, the number of contractors able to fulfill such properties,
  - and vice versa,
- then we have a Galois Connection[95] ▪

5.6.5.2 Galois Connections and Intentionality – A Possible Research Topic ?

We have a hunch[96] ! Namely that there are some sort of Galois Connections with respect to intentionality. We leave to the interested reader to pursue this line of inquiry.

5.6.6 Discovering Intentional Pulls

The analysis and description of a domain's external qualities and the internal qualities of unique identifiers, mereologies and attributes can be pursued systematically – endurant sort by sort. Not so with the discovery of a domain's possible intentional pulls. Basically *"what is going on"* here is that the domain analyser cum describer considers pairs, triples or more part "independent"[97] endurants and reflects on whether they stand in an *intentional pull* relation to one another. We refer to Sects. **5.6.2.2 – 5.6.2.3**.

---

[94] The E-mail, June 5, 2020, from Asger Eir

[95] This was, more formally, shown Dr. Asger Eir's PhD thesis [75].

[96] Hunch: a feeling or guess based on intuition rather than fact.

[97] By "independent" we shall here mean that these endurants are not 'derived' from one-another !

5.6.6.1 Identifying Intents

TO BE WRITTEN

5.6.6.2 Searching for Intentional Pulls

TO BE WRITTEN

5.6.6.3 Describing Intentional Pulls

TO BE WRITTEN

5.7 A Domain Discovery Procedure, II

This section is based on Sect. 5.8 of [49, Pages 146–147].

We continue from Sect. **4.8**.

5.7.1 The Process

We shall again emphasize some aspects of the domain analysis & description method. A **method procedures** is that of *exhaustively analyse & describe* all internal qualities of the domain under scrutiny. A **method technique** implied here is that sketched below. The **method tools** are here all the analysis and description prompts covered so far.

Please be reminded of *Discovery Schema 0*'s declaration of *Notice Board* variables (Page 56). In this section we collect (i) the *description of unique identifiers* of all parts of the state; (ii) the *description of mereologies* of all parts of the state; and (iii) the *description of attributes* of all parts of the state. (iii) We finally gather these into the *discover_internal_endurant_qualities* procedures.

An Endurant Internal Qualities Domain Analysis and Description Process

```
value
    discover_uids: Unit → Unit
    discover_uids() ≡
         for ∀ v • v ∈ gen
             do txt := txt † [type_name(v)↦txt(type_name(v))⁀⟨describe_unique_identifier(v)⟩] end
    discover_mereologies: Unit → Unit
    discover_mereologies() ≡
         for ∀ v • v ∈ gen
             do txt := txt † [type_name(v)↦txt(type_name(v))⁀⟨describe_mereology(v)⟩] end
    discover_attributes: Unit → Unit
    discover_attributes() ≡
         for ∀ v • v ∈ gen
             do txt := txt † [type_name(v)↦txt(type_name(v))⁀⟨describe_attributes(v)⟩] end
    discover_intentional_pulls: Unit → Unit
    discover_intentional_pulls() ≡
         for ∀ (v′,v″) • {v′,v″} ⊆ gen
             do txt := txt † [type_name(v′)↦txt(type_name(v′))⁀⟨describe_intentional_pull()⟩]
                          † [type_name(v″)↦txt(type_name(v″))⁀⟨describe_intentional_pull()⟩] end
```

```
     describe_intentional_pull: Unit → ...
     describe_intentional_pull() ≡ ...


value
     discover_internal_qualities: Unit → Unit
     discover_internal_qualities() ≡
          discover_uids() ;
               axiom [ all parts have unique identifiers ]
          discover_mereologies() ;
               axiom [ all unique identifiers are mentioned in sum total of ]
                     [ all mereologies and no isolated proper sets of parts ]
          discover_attributes() ;
               axiom [ sum total of all attributes span all parts of the state ]
          discover_intentional_pulls()
```

We shall comment on the axioms in the next section.

5.7.2 A Suggested Analysis & Description Approach, II

Figure 4.3 on page 51 possibly hints at an analysis & description order in which not only the external qualities of endurants are analysed & described, but also their internal qualities of unique identifiers, mereologies and attributes.

In Sect. **4.8** on page 55 we were concerned with the analysis & description order of endurants. We now follow up on the issue of (in Sect. **4.5.1.3** on page 50) on how compounds are treated: namely as both a "root" parts and as a composite of two or more "sibling" parts and/or fluids. The taxonomy of the road transport system domain, cf. Fig. 4.3 on page 51 and Example 35 on page 46, thus gives rise to many different analysis & description traversals. Figure 5.4 illustrates one such order.

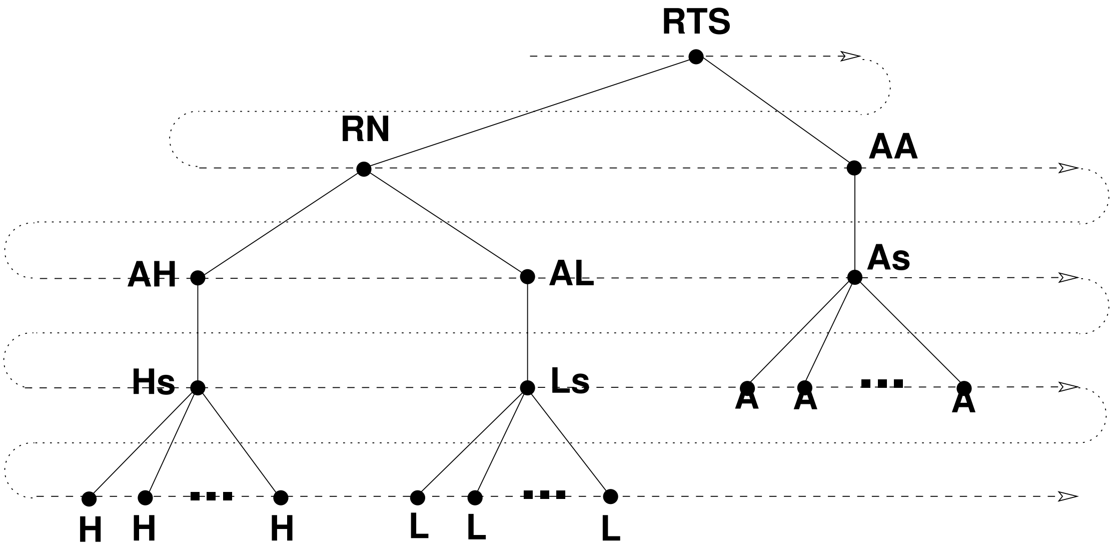


Fig. 5.4 A Breadth-First, Top-Down Traversal

Again, it is up to the domain engineer cum scientist to decide. If the domain analyser cum describer decides to not endow a compound "root" with internal qualities, then an 'internal qualities' traversal will not have to neither analyse nor describe those qualities.

5.8 Summary

Internal Qualities Predicates and Functions: Method Tools

- **Analysis Predicates:** As in Chapter 4 these predicates apply to endurants.
- **Attribute Analysis Predicates:** The predicates apply to attribute values.
- **Analysis Functions:** These functions yield appropriate values: unique identifiers and attribute type names.
- **Retrieval Function:** This function is generic. It applies to a unique part identifier and yields the part identified.
- **Description Functions:** There are three such functions: describing unique identifiers, mereologies and attributes.
- **Domain Discovery:** The procedure here being described, informally, guides the domain analyser cum describer to do the job !

#	Name	Introduced
	Analysis Predicates	
16	is_manifest	page **62**
17	is_structure	page **62**
	Attribute Analysis Predicates	
1	is_static_attribute	page **74**
2	is_dynamic_attribute	page **74**
3	is_inert_attribute	page **74**
4	is_reactive_attribute	page **74**
5	is_active_attribute	page **75**
6	is_autonomous_attribute	page **75**
7	is_biddable_attribute	page **75**
8	is_programmable_attribute	page **75**
9	is_monitorable_only_attribute	page **76**
	Analysis Functions	
	all_uniq_ids	page **65**
	calculate_all_unique_identifiers	page **64**
6	analyse_attribute_types	page **79**
7	sta_attr_types	page **79**
8	mon_attr_types	page **79**
9	pro_attr_types	page **79**
	Retrieval, Read and Write Functions	
	retr_part	page **66**
108	read_A_from_P	page **81**
109	update_P_with_A	page **82**
	Description Functions	
5	describe_unique_identifier	page **63**
6	describe_mereology	page **68**
7	describe_attributes	page **73**
	Domain Discovery	
	discover_uids	page **97**
	discover_mereologies	page **97**
	discover_attributes	page **97**
	discover_internal_qualities	page **97**

• • •

Please consider Fig. 4.1 on page 39. This chapter has covered the horisontal and vertical lines to the left in Fig. 4.1.

Chapter 6

Perdurants

Contents

Please consider Fig. 4.1 on page 39. The previous two chapters covered the left of Fig. 4.1. This chapter covers the right of Fig. 4.1.

•••

This chapter is a rather “drastic” reformulation and simplification of [49, *Chapter 7, i.e., pages 159–196*]. Besides, Sect. **6.5** is new.

In this chapter we transcendentally “morph” manifest **parts** into **behaviours,** that is: **endurants** into **perdurants**. We analyse that notion and its constituent notions of **actors**, **channels** and **communication**, **actions** and **behaviours**. We shall investigate the, as we shall call them, perdurants of domains. That is state and time-evolving domain phenomena. The outcome of this chapter is that the reader will be able to model the perdurants of domains. Not just for a particular domain instance, but a possibly infinite set of domain instances[98].

---

[98] By this we mean: You are not just analysing a specific domain, say the one manifested around the corner from where you are, but any instance, anywhere in the world, which satisfies what you have described.

6.1 Part Behaviours – An Analysis

We remind the reader of Sect. **2.1.2** on page 9.

6.1.1 Behaviour Definition Analysis

Parts co-exist; they do so endurantly as well as perdurantly: endure and perdure.

Part perdurants, i.e., behaviours, interact with their surroundings, that is, with other behaviours. This is true for both natural and man-made parts. The present domain modelling method is mainly focused on man-made parts, that is artefacts. So our next analysis will take its clues from artefactual parts.

We can, roughly, analyse part behaviours into three kinds.

- **Proactive Behaviours:** Behaviour B_i offers to synchronise and communicate values – *internal non-deterministically* with either of a definite number of distinct part sort behaviours B_a, B_b, ..., B_c:

  ```
  B(i)(args) ≡
        (... ch[{i,a}] ! a_val ; ... ; B(i)(args′))
     ⊓ (... ch[{i,b}] ! b_val ; ... ; B(i)(args″))
     ⊓ ...
     ⊓ (... ch[{i,c}] ! c_val ; ... ; B(i)(args‴))
  ```

  The tail-recursive invocation of B_i indicates a possible "update" of behaviour B_i arguments. More on this later.
- **Responsive Behaviours:** Behaviour B_i *external non-deterministically* expresses willingness to synchronisation with and accept values from either of a definite number of distinct part sort behaviours B_a, B_b, ..., B_c:

  ```
  B(i)(args) ≡
        (... let av = ch[{i,a}] ? in ...  B(i)(args′) end)
     ⌷ (... let bv = ch[{i,b}] ? in ... ; B(i)(args″) end)
     ⌷ ...
     ⌷ (... let cv = ch[{i,c}] ? in ... ; B(i)(args‴) end)
  ```

- **Mixed Behaviours:** Or behaviours, more generally, "are" an internal non-deterministic "mix" of the above:

  ```
  B(i)(args) ≡
        ((... ch[{i,a}] ! a_val ; ... ; B(i)(args′))
     ⊓  (... ch[{i,b}] ! b_val ; ... ; B(i)(args″))
     ⊓  ...
     ⊓  (... ch[{i,c}] ! c_val ; ... ; B(i)(args‴)))
     ⊓ ((... let av = ch[{i,a}] ? in ...  B(i)(args′) end)
     ⌷  (... let bv = ch[{i,b}] ? in ... ; B(i)(args″) end)
     ⌷  ...
     ⌷  (... let cv = ch[{i,c}] ? in ... ; B(i)(args‴) end))
  ```

- The "bodies" of the B_i behaviour definitions, i.e., "...", may contain interactions with [yet other] behaviours. Schematically for example:

```
ch[ {i,x} ] ! x_val
{ ch[ {i,z} ] ! z_val | z:{z1,z2,...,zm} }
let yv = ch[ {i,y} ] ? in ... end
let zv = ▯ { ch[ {i,z} ] ? | z:{z1,z2,...,zm} } in ... end
```

Etcetera. The full force of CSP with RSL is at play !

6.1.2 Channel Analysis

This is the first of two treatments of the concept of *channels*; the present treatment is informal, motivational, the second treatment, Sect. **6.2** (right next !), is more formal.

The CSP concept of *channel* is to be our way of expressing the "medium" in which behaviours interact. Channels is thus an abstract concept. Please do not think of it as a physical, an IT (information technology) device. As an abstract concept it is defined in terms of, roughly, the laws, the semantics, of CSP [100]. We write 'roughly' since the CSP we are speaking of, is "embedded" in RSL.

6.2 Domain Channel Description

We simplify the general treatment of channel declarations. Basically all we can say, for any domain, is that any two distinct part behaviours may need to communicate. Therefore we declare a vector of channels indexed by sets of two distinct part identifiers.

```
value
    discover_channels: Unit → Unit
    discover_channels() ≡
        ❝ channel { ch[ {ij,ik} ] | ij,ik:UI • {ij,ik}⊆ uid_σ ∧ ij≠ik } M ❞
```

Initially we shall leave the type of messages over channels further undefined. As we, laboriously, work through the definition of behaviours, we shall be able to make M precise. all_uniq_ids was defined in Sect. **5.2.4** on page 65.

6.3 Behaviour Definition Description

Behaviours have to be described. Behaviour definitions are in the form of function definitions and are here expressed in RSL relying, very much, on its CSP component. Behaviour definitions describe the type of the arguments the function, i.e., the behaviour, for which it is defined, that is, which kind of values it accepts. Behaviour definitions further describe

Thus there are two elements to a behaviour definition: the behaviour *signature* and the behaviour *body* definitions.

6.3.1 Behaviour Signatures

6.3.1.1 General

Function, F, signatures consists of two textual elements: the function name and the function type:

value F: A → B, or F: a:A → B

where A and B are the types of function ("input") arguments, respectively function ("output") values for such arguments. The first form F: A → B is what is normally referred to as the form for function signatures. The second form: F: a:A → B "anticipates" the general for for function F invocation: F(a).

6.3.1.2 Domain Behaviour Signatures

A schematic form of part (p) behaviour signatures is:

b: bi:BI→me:Mer→svl:StaV*→mvl:MonV*→prgl:PrgV* channels **Unit**

We shall motivate the general form of part behaviour, B, signatures, "step-by-step":

α.	b	the [chosen] name of part p behaviours.
β	U→V→...→W→Z:	The function signature is expressed in the `Schönfinkel/Curry`[99] style – corresponding to the invocation form F(u)(v)...(w)
γ.	bi:BI:	a general value and the type of part p unique identifier
δ.	me:Mer:	a general value and the type of part p mereology
ϵ.	svl:StaV*:	a general (possibly empty) list of values and types of part p's (possibly empty) list of static attributes
ζ.	mvl:MonV*:	a general list of names of types of part p's (possibly empty) list of monitorable attributes
η.	prgl:PrgV*:	a general list of values and types of part p's (possibly empty) list of programmable attributes
θ.	channels:	are usually of the form: {ch[{i,j}]|(i,j)∈I(me)} and express the subset of channels over which behaviour Bs interact with other behaviors
ι.	**Unit**:	designates the single value ()

In detail:

α. **Behaviour name:** In each domain description there are many sorts, B, of parts. For each sort there is a generic behaviour, whose name, here b. is chosen to suitably reflect B.

β. **Currying** is here used in the pragmatic sense of grouping "same kind of arguments", i.e., separating these from one-another, by means of the →s.

γ. The **unique identifier** of part sort B is here chosen to be BI. Its value is a constant.

δ. The **mereology** is a usually constant. For same part sorts it may be a variable.

Example 71 . **Variable Mereologies:** For a road transport system where we focus on the transport the mereology is a constant. For a road net where we focus on the development of the road net: building new roads: inserting and removing hubs and links, the mereology is

---

[99] Moses Schönfinkel (1888–1942) was a Russian logician and mathematician accredited with having invented combinatory logic [https://en.wikipedia.org/wiki/Moses_Schönfinkel]. Haskell B. Curry (1900–1982) was an American mathematician and logician known for his work in combinatory logic [https://en.wikipedia.org/wiki/Haskell_Curry]

a variable. Similar remarks apply to canal systems `www.imm.dtu.dk/~dibj/2021/Graphs/-Rivers-and-Canals.pdf`, pipeline systems [36], container terminals [44], assembly line systems [46], etc. ▪

ϵ. **Static attribute values** are constants. The use of static attribute values in behaviour body definitions is expressed by an identifier of the stvl list of identifiers.

ζ. **Monitorable attribute values** are generally, ascertainable, i.e., readable, cf. Sect. **5.4.3.1** on page 81. Some are *biddable*, can be changed by a, or the behaviour, cf. Sect. **5.4.3.2** on page 82, but there is no guarantee, as for programmable attributes, that they remain fixed.

The use of a[ny] monitorable attribute value in behaviour body definitions is expressed by a read_A_from_P(mv,bi) where mv is an identifier of the mvl list of identifiers and bi is the unique part identifier of the behaviour definition in which the read occurs.

The update of a biddable attribute value in behaviour body definitions is expressed by a update_P_with_A(bi,mv,a).

η. **Programmable attribute values** are just that. They vary as specified, i.e., "programmed", by the behaviour body definition. Tail-recursive invocations of behaviour B_i "replace" relevant programmable attribute argument list elements with "new" values.

θ. **channels:** $\mathcal{I}$(me) expresses a set of unique part identifiers different from bi, hence of behaviours, with which behaviour b(i) interacts.

ι. The **Unit** of the behaviour signature is a short-hand for the behaviour either **read**ing the value of a monitorable attribute, hence global state σ, or performing a **write**, i.e., an *update*, on σ.

6.3.1.3 Action Signatures

Actions come in any forms:

144 Some take no arguments, say action_a(), but read the global state component σ, and
145 others also take no arguments, say action_b(), but update the global state component σ.
146 Some take an argument, say, action_c(c), but do not "touch" a global state component,
147 while others both take an argument and deliver a value, say action_d(d) and also do not "touch" a global state component.
148 Et cetera !

type A, B, C, D, ...
value
144. action_a: **Unit** → **read** σ A
145. action_b: **Unit** → **write** σ B
146. action_c: C → **Unit**
147. action_d: D → E **Unit**
148. ...

An example of 146 are the CSP output: ch[...] ! c, and an example of 147 are the CSP input: **let** e = ch[...] ? **in** ... **end**.

6.3.2 Behaviour Invocation

The general form of behaviour invocation is shown below. The invocation follows the "Currying" of the behaviour type signature. [Normally one would write all this on one line: b(i)(m)(s)(m)(p) ≡.]

```
    behaviour_name
        (unique_identifier)
              (mereology)
                  (static_values)
                        (monitorable_attribute_names)
                            (programmable_variables) ≡
        ... body ...
```

When first “invoked”, that is, transcendentally deduced, i.e., “morphed”, from a manifest part, *p*, the invocation looks like:

```
    value
          discover_ behaviour_signature: P → RSL-Text
          discover_ behaviour_signature(p) ≡
          ❝ behaviour_name:
                          UId → Mereo → StaVL → MonVL → ProVL → channels Unit
              behaviour_name
                          (uid_B(p))
                              (mereo_B(p))
                                    (types_to_values(static_attribute_types(p)))
                                        (mon_attribute_types(p))
                                            (types_to_values(programmable_attribute_types(p))) ≡ ❞
                 pre: is_B(p) ∧ is_manifest(p)

          discover_ behaviour_signatures: Unit → RSL-Text
          discover_ behaviour_signatures() ≡
                 { discover_ behaviour_signature(p) | p ∈ σ ∧ is_manifest(p) }
```

6.3.3 Behaviour Definition Bodies

We remind the reader of Sect. **6.1.1** on page 100.

The general, “mixed”, form of behaviour definitions was given as:

```
    B(i)(args) ≡
          ( ( ... )
       ⊓   ( ... ch[{i,b}] ! b_val ; ... ; B(i)(args′′) )
       ⊓   ( ... ) )
       ⊓ ( ( ... )
       □   (... let bv = ch[{i,b}] ? in ... ; B(i)(args′′) end )
       □   ( ... ) )
```

We can express the same by separating the alternatives into invocations of separately defined behaviuors.

```
    B(i)(args) ≡
          ( ...
       ⊓   Bin_j(i)(args)
       ⊓    ... )
       ⊓ ( ...
       □   Bxn_k(i)(args)
       □    ...  )
```

where the internal don-deterministically invoked behaviours Bin_j(i)(args) and the external don-deterministically invoked behaviours Bin_k(i)(args) are then separately defined:

Bin_j(i)(args) ≡ (... Bin_j(i)(args′))
Bxn_k(i)(args) ≡ (... Bxn_k(i)(args″))

6.3.4 Discover Behaviour Definition Bodies

In other words, for current lack of a more definitive methodology for "discovering" the bodies of behaviour definitions we resort to ". . . "!

```
value
    discover_behaviour_definition: P → RSL-Text
    discover_behaviour_definition(p) ≡ ...

    discover_behaviour_definitions: Unit → RSL-Text
    discover_behaviour_definitions() ≡
        { discover_behaviour_definition(p) | p ∈ σ ∧ is_manifest(p) }
```

Example 72 . **Automobile Behaviour:** We remind the reader of the main, running example of this **tutorial**, the of *the road transport system* Example[100].

Signatures

149 automobile:
  a there is the usual "triplet" of arguments: unique identifier, mereology and static attributes;
  b then there are two programmable attributes: the automobile position (cf. Item 99 on page 78), and the automobile history (cf. Item 136c on page 92);
  c and finally there are the input/output channel references allowing communication between the automobile and the hub and link behaviours.

150 Similar for
  a link and
  b hub behaviours.

We omit the modelling of monitorable attributes (...).

```
value
149a,149a automobile: ai:AI → ((_,uis):AM) → ...
149b        → (apos:APos × ahist:AHist)
149c          in out {ch[{ai,ui}]|ai:AI,ui:(HI|LI) • ai∈ais ∧ ui ∈ uis} Unit
150a     link: li:LI → (his,ais):LM → LΩ → ...
150a        → (LΣ×L_Hist)
150a          in out {ch[{li,ui}]|li:LI,ui:(AI|HI)-set • ai∈ais ∧ li ∈lis∪his} Unit
```

---

[100] That is, examples 27 on page 37, 34 on page 44, 35 on page 46, 36 on page 48, 38 on page 51, 41 on page 62, 42 on page 64, 44 on page 65, 45 on page 66, 46 on page 68, 47 on page 69, 57 on page 76, 58 on page 77, 59 on page 77, and 66 on page 92.

```
150b   hub: hi:HI → ((_,ais):HM) → HΩ ...
150b      → (HΣ×H_Host)
150b         in out {ch[{ai,ui}]|hi:HI,ai:AI • ai∈ais ∧ hi ∈ uis} Unit
```

Definitions: Automobile at a Hub

151 We abstract automobile behaviour at a Hub (hi).

- a Either the automobile remains in the hub,
- b or, internally non-deterministically,
- c leaves the hub entering a link,
- d or, internally non-deterministically,
- e stops.

```
151  automobile(ai)(aai,uis)(...)(apos:atH(fli,hi,tli),ahist) ≡
151a     automobile_remains_in_hub(ai)(aai,uis)(...)(apos:atH(fli,hi,tli),ahist)
151b    ⊓
151c     automobile_leaving_hub(ai)(aai,uis)(...)(apos:atH(fli,hi,tli),ahist)
151d    ⊓
151e     automobile_stop(ai)(aai,uis)(...)(apos:atH(fli,hi,tli),ahist)
```

152 [151a] The automobile remains in the hub:

- a the automobile remains at that hub, “idling”,
- b informing (“first”) the hub behaviour.

```
152  automobile_remains_in_hub(ai)(aai,uis)(...)(apos:atH(fli,hi,tli),ahist) ≡
152     let τ = record_𝕋𝕀𝕄𝔼() in
152b     ch[ai,hi] ! τ ;
152a     automobile(ai)(aai,uis)(...)(apos,upd_hist(τ,hi)(ahist))
152     end

152a  upd_hist: (𝕋𝕀𝕄𝔼×I) → (AHist|LHist|HHist) → (AHist|LHist|HHist)
152a  upd_hist(τ,i)(hist) ≡ hist † [i ↦ ⟨τ⟩⁀hist(i)]
```

153 [151c] The automobile leaves the hub entering a link:

- a tli, whose “next” hub, identified by thi, is obtained from the mereology of the link identified by tli;
- b informs the hub it is leaving and the link it is entering,
- c “whereupon” the vehicle resumes (i.e., “while at the same time” resuming) the vehicle behaviour positioned at the very beginning (0) of that link.

```
153  automobile_leaving_hub(ai)(aai,uis)(...)(apos:atH(fli,hi,tli),ahist) ≡
153a     (let ({fhi,thi},ais) = mereo_L(retr_L(tli)(σ)) in assert: fhi=hi
153b      ( ch[ai,hi] ! τ || ch[ai,tli] ! τ ) ;
153c      automobile(ai)(aai,uis)(...)
153c            (onL(tli,(hi,thi),0),upd_hist(τ,tli)(upd_hist(τ,hi)(ahist))) end)
```

154 [151e] Or the automobile “disappears — off the radar” !

```
154  automobile_stop(ai)(aai,uis),(...)(apos:atH(fli,hi,tli),ahist) ≡ stop
```

Similar behaviour definitions can be given for *automobiles on a link*, for *link*s and for *hub*s. Together they must reflect, amongst other things: the time continuity of automobile flow, that automobiles follow routes, that automobiles, links and hubs together adhere to the intentional pull expressed earlier, et cetera. A specification of these aspects must be proved to adhere to these properties.

6.4 Domain Behaviour Initialisation

For every manifest part it must be described how its behaviour is initialised.

Example 73 . **The Road Transport System Initialisation:** We “wrap up” the main example of this **tutorial**: We omit treatment of monitorable attributes.

155 Let us refer to the system initialisation as an action.
156 All links are initialised,
157 all hubs are initialised,
158 all automobiles are initialised,
159 etc.

```
value
155. rts_initialisation: Unit → Unit
155. rts_initialisation() ≡
156.    ‖ { link(uid_L(l))(mereo_L(l))(attr_LEN(l),attr_LΩ(l))(attr_L_Traffic(l),attr_LΣ(l))| l:L • l ∈ ls }
157.    ‖ ‖ { hub(uid_H(l))(mereo_H(l))(attr_HΩ(l))(attr_H_Traffic(l),attr_HΣ(l))| h:H • h ∈ hs }
158.    ‖ ‖ { automobile(uid_A(a))(mereo_A(a))(attr_RegNo(a))(attr_APos(a)) | a:A • a ∈ as }
159.    ‖ ...
```

We have here omitted possible monitorable attributes. We refer to *ls*: Item 45 on page 55, *hs*: Item 46 on page 55, and *as*: Item 47 on page 55 ▪

6.5 Discrete Dynamic Domains

Up till now our analysis & description of a domain, has, in a sense, been *static:* in analysing a domain we considered its entities to be of a definite number. In this section we shall consider the case where the number of entities change: where new entities are *created* and existing entities are *destroyed*, that is: where new parts, and hence behaviours, arise, and existing parts, and hence behaviours, cease to exist.

6.5.1 Create and Destroy Entities

In the domain we can expect that its behaviours create and destroy entities.

Example 74 . **Creation and Destruction of Entities:** In the *road transport* domain new hubs, links and automobiles may be inserted into the road net, and existing links, hubs and automobiles may be removed from the road net. In a *container terminal* domain [22, 44] new containers are introduced, old are discarded; new container vessels are introduced, old are discarded; new ship-to-shore cranes are introduced, old are discarded; et cetera. In a *retailer* domain [47] new customers are introduced, old are discarded; new retailers are introduced, old are discarded; new merchandise is introduced, old is discarded; et cetera. In a *financial system* domain new customers are introduced, old are discarded; new banks are introduced, old are discarded; new brokers are introduced, old are discarded; et cetera ▪

The issue here is: When hubs and links are inserted or removed the mereologies of "neighbouring" road elements change, and so does the mereology of automobiles. When automobiles are inserted or removed The mereology of road elements have to be changed to take account of the insertions and removals, and so does the mereology of automobiles. And, some domain laws must be re-expressed: The domain part state, σ, must be updated[101], and so must the unique identifier state, uid_σ[102].

6.5.1.1 Create Entities

It is taken for granted here that there are behaviours, one or more, which take the initiative to and carry out the creation of specific entities. Let us refer to such a behaviour as the "creator". To create an entity implies the following three major steps [A.–C.] the step wise creation of the part and initialisation of the transduced behaviour, and [D.] the adjustment of all such part behaviours that might have their mereologies and attributes updated to accept such requests from creators.

A. To decide on the part sort – in order to create that part – that is

- to obtain a unique identifier – one hitherto not used;
- to obtain a mereology, one
  - according to the general mereology for parts of that sort,
  - and how the part specifically is to "fit" into its surroundings;
- to obtain an appropriate set of attributes:
  - again according to the attribute types for that part sort
  - and, more specifically, choosing initial attribute values.
- This part is then "joined" to the global part state, σ[103] and
- its unique identifier "joined" to the global unique identifier state, uid_σ[104].

B. Then to transcendentally deduce that part into a behaviour:

- initialised (according to Sect. **6.3.1**) with
  - the unique identifier,
  - the mereology, and
  - the attribute values
- This behaviour is then invoked and "joined" to the set of current behaviours, cf. Sect. **6.4** on the preceding page – i.e., just above!

C. Then, finally, to "adjust" the mereologies of topologically or conceptually related parts,

- that is, for each of these parts to update:
- their mereology and possibly some
- state and state space

arguments of their corresponding behaviours.

---

[101] Cf. Sect. **4.7.2** on page 55

[102] Cf. Sect. **5.2.4** on page 65

[103] Cf. Sect. **4.7.2** on page 55

[104] Cf. Sect. **5.2.4** on page 65

D. The update of the mereologies of already “running” behaviours requires the following:

- that, potentially all, behaviours offers to accept
- mereology update requests from the “creator” behaviour.

The latter means, practically speaking, that each part/behaviour which may be subject to mereology changes externally non-deterministically expresses an offer to accept such a change.

Example 75 . **Road Net Administrator:** We introduce the road net behaviour – based on the road net composite part, RN.

160 The road net has a programmable attribute: a *road net (development & maintenance) graph*.[105] The road net graph consists of a quadruple: a map that for each hub identifier records “all” the information that the road net administrator deems necessary[106] for the maintenance and development of road net hubs; a map that for each link identifier records “all” the information[107] that the road net administrator deems necessary for the maintenance and development of road net links; and a map from the hub identifiers to the set of identifiers of the links it is connected to, and the set of all automobile identifiers.
161 This graph is commensurate with the actual topology of the road net.

```
type
160.  G = (HI →m H_Info) × (LI →m L_Info) × (HI →m LI-set) × AI-set
value
160.  attr_G: RN → G
axiom
160.  ∀ (hi_info,li_info,map,ais):G •
160.     dom map = dom hi_info = his ∧ ∪ rng map = dom li_info = lis ∧
161.     ∀ hi:HI • hi ∈ dom hi_info ⇒
161.       let h:H • h ∈ σ ∧ uid_H(h)=hi in
161.       let (lis′,...) = mereo_H(h) in lis′ = map(hi)
161.       ais ⊆ ais ∧ ...
161.       end end
```

Please note the fundamental difference between the *road net (development & maintenance) graph* and the road net. The latter pretends to be “the real thing”. The former is “just” an abstraction thereof !

162 The road net mereology (“bypasses”) the hub and link aggregates, and comprises a set of hub identifiers and a set of link identifiers – of the road net[108].

```
type
162.  H_Mer = AI-set × LI-set
162.  mereo_RN: RN → RNMer
axiom
162.  ∀ rts:RTS • let (_,lis) = mereo_H(obs_RN(rts)) in lis ⊆ lis end
value
```

163 The road net [administrator] behaviour,

---

[105] The presentation of the road net Behaviour, rn, is simplified.

[106] We presently abstract from what this information is.

[107] See footnote 106.

[108] This is a repeat of the hub mereology given in Item 74 on page 68.

164 amongst other activities (. . .)
165 internal non-deterministically decides upon

  a either a hub insertion,
  b or a link insertion,
  c or a hub removal,
  d or a link removal;

These four sub-behaviours each resume being the road net behaviour.

value
163. rn: RNI → RNMer → G → **in**,**out**{ch[{i,j}]|{i,j}⊆uid_σ}
163. rn(rni)(rnmer)(g) ≡
164. ...
165a. ⊓ insert_hub(g)(rni)(rnmer)
165b. ⊓ insert_link(g)(rni)(rnmer)
165c. ⊓ remove_hub(g)(rni)(rnmer)
165d. ⊓ remove_link(g)(rni)(rnmer)

Details on the insert and remove actions are given below.

166 These road net sub-behaviours require information about

  a a hub to be inserted: its initial state, state space and [empty] traffic history, or
  b a link to be inserted: its length, initial state, state space and [empty] traffic history, or
  c a hub to be removed: its unique identifier, or
  d a link to be removed: its unique identifier.

type
166. Info == nHInfo | nLInfo | oHInfo | oLInfo
166. nHInfo :: HΣ × HΩ × H_Traffic
166. nLInfo :: LEN × LΣ × LΩ × L_Traffic
166. oHInfo :: HI
166. oLInfo :: LI ▪

Example 76 . **Road Net Development: Hub Insertion:** Road net development alternates between design, based on the *road net (development & maintenance) graph*, and actual, “real life”, construction taking place in the real surroundings of the road net.

167 If a hub insertion then the road net behaviour, based on the hub and link information and the road net layout in the *road net (development & maintenance) graph* selects

  a an initial mereology for the hub, h_mer,
  b an initial hub state, hσ, and
  c an initial hub state space, hω, and
  d an initial, i.e., empty hub traffic history;

168 updates its *road net (development & maintenance) graph* with information about the new hub,
169 and results in a suitable grouping of these.

value
167. design_new_hub: G → (nHInfo×G)
167. design_new_hub(g) ≡
167a. **let** h_mer:HMer = $\mathcal{M}_{ih}$(g),

```
167b.       h$\sigma$:H$\Sigma$ = $\mathcal{S}_{ih}$(g),
167c.       h$\omega$:H$\Omega$ = $\mathcal{O}_{ih}$(g),
167d.       h_traffic = [],
168.        g′ = $\mathcal{MSO}_{ih}$(g) in
169.     ((h_mer,h$\sigma$,h$\omega$,h_traffic),g′) end
```

We leave open, in Items 167a–167c, as to what the initial hub mereology, state and state space should be initialised, i.e., the $\mathcal{M}_{ih}$,$\mathcal{S}_{ih}$,$\mathcal{O}_{ih}$ and $\mathcal{MSO}_{ih}$ functions.

170 To insert a new hub the road net administrator

  a first designs the new hub,
  b then selects a hub part
  c which satisfies the design,
  whereupon it updates the global states
  d of parts σ,
  e of unique identifiers, and
  f of hub identifiers –

in parallel, and in parallel with
171 initiating a new hub behaviour
172 and resuming being the road net behaviour.

```
170.  insert_hub: G×RNI×RNMer → Unit
170.  insert_hub(g,rni,rnmer) ≡
170a.    let ((h_mer,h$\sigma$,h$\omega$,h_traffic),g′) = design_new_hub(g) in
170b.    let h:H • h$\notin\sigma$ •
170c.              mereo_H(h)=h_mer ∧ h$\sigma$=attr_H$\Sigma$(h) ∧
170c.              h$\omega$=attr_H$\Omega$(h) ∧ h_traffic=attr_HTraffic(h) in
170d.    $\sigma$ := $\sigma$ ∪ {h}
170e.  ∥ uid$_\sigma$ := uid$_\sigma$ ∪ {uid_H(h)}
170f.  ∥ $his$ := $his$ ∪ {uid_H(h)}
171.   ∥ hub(uid_H(h))(attr_H$\Sigma$(h),attr_H$\Omega$(h),attr_H$\Omega$(h))
172.   ∥ rn(rni)(rnmer)(g′)
170.     end end ▪
```

Example 77 . **Road Net Development: Link Insertion:**

173 If a link insertion then the road net behaviour based on the hub and link information and the road net layout in the *road net (development & maintenance) graph* selects

  a the mereology for the link, h_mer[109],
  b the (static) length (attribute),
  c an initial link state, lσ,
  d an initial link state space lω, and
  e and initial, i.e., empty, link traffic history;

174 updates its *road net (development & maintenance) graph* with information about the new link,
175 and results in a suitable grouping of these.

---

[109] that is, the two existing hub identifiers between whose hubs the new link is to be inserted

```
value
173.   design_new_link: G → (nLInfo×G)
173.   design_new_link(g) ≡
173a.      let l_mer:LMer = $\mathcal{M}_{il}$(g),
173b.          le:LEN = $\mathcal{L}_{il}$(g),
173c.          l$\sigma$:L$\Sigma$ = $\mathcal{S}_{il}$(g),
173d.          l$\omega$:L$\Omega$ = $\mathcal{O}_{il}$(g),
173e.          l_hist:L_Hist = [ ]
174.           g′:G = $\mathcal{MLSO}_{il}$(g) in
175.       ((l_mer,le,l$\sigma$,l$\omega$,l_hist),g′) end
```

We leave open, in Items 173a–173d, as to what the initial link mereology, state and state space should be initialised.

176 To insert a new link the road net administrator
  a first designs the new link,
  b then selects a link part
  c which satisfies the design,
    whereupon it updates the global states
  d of parts, σ,
  e of unique part identifiers, and
  f of link identifiers –

  in parallel, and in parallel with
177 initiating a new link behaviour and
178 updating the mereologies and possibly the state and the state space attributes of the connected hubs.

```
value
176.   insert_link: G → Unit
176.   insert_link(rni,l) ≡
176a.      let ((l_mer,le,l$\sigma$,l$\omega$,l_traffic_hist),g′) = design_new_link(g) in
176c.      let l:L • l$\notin\sigma$ • mereo_L(l)=l_mer ∧
176c.                  le=attr_LEN(l) ∧ l$\sigma$=attr_L$\Sigma$(l) ∧
176c.                  l$\omega$=attr_L$\Omega$(l) ∧ l_traffic_hist=attr_HTraffic(l) in
176d.      $\sigma$ := $\sigma$ ∪ {l}
176e.   || uid$_\sigma$ := uid$_\sigma$ ∪ {uid_L(l)}
176f.   || lis := list ∪ {}
177.    || link(uid_L(l))(l_mer)(le,l$\omega$)(l$\sigma$,l_traffic)
178.    || ch[{rni,hi1}] ! updH($\mathcal{M}_{il}$(g),$\Sigma_{il}$(g),$\Omega_{il}$(g))
178.    || ch[{rni,hi2}] !
176.      end end ▪
```

We leave undefined the mereology and the state σ and state space ω update functions.

6.5.1.2 Destroy Entities

The introduction to Sect. **6.5.1.1** on page 108 on the *creation of entities* outlined a number of creation issues ([A, B, C, D]). For the *destruction of entities* description matters are a bit simpler. It is, almost, simply a matter of designating, by its unique identifier, the entity: part and behaviour to be destroyed. Almost! The mereology of the destroyed entity must be such that the destruction does not leave "dangling" references!

Example 78 . Road Net Development: Hub Removal:

179 If a hub removal then the road net design_remove_hub behaviour, based on the *road net (development & maintenance) graph*, calculates the *unique hub identifier* of the "isolated" hub to be removed – that is, is not connected to any links,
180 updates the *road net (development & maintenance) graph*, and
181 results in a pair of these.

```
value
179. design_remove_hub: G → (HI×G)
179. design_remove_hub(g) as (hi,g′)
179.    let hi:HI • hi ∈ his ∧ let (_,lis) = mereo_H(retr_part(hi)) in lis={} end in
180.    let g′ = $\mathcal{M}_{rh}$(hi,g) in
181.    (hi,g′) end end
```

182 To remove a hub the road net administrator

- a first designs which old hub is to be removed
- b then removes the designated hub,
  whereupon it updates the global states
- c of parts σ,
- d of unique identifiers, and
- e of hub identifiers –

in parallel, and in parallel with
183 stopping the old hub behaviour
184 and resuming being a road net behaviour.

```
value
182. remove_hub: G → RNI → RNMer → Unit
182. remove_hub(g)(rni)(rnmer) ≡
182a.     let (hi,g′) = design_remove_hub(g) in
182b.     let h:H • uid_H(h)=hi ∧ ... in
182c.     $\sigma$ := $\sigma$ \ {h}
182d.  || uid$_\sigma$ := uid$_\sigma$ \ {hi}
182e.  || his := his \ {hi}
183.   || ch[{rni,hi}] ! mkStop()
184.   || rn(rni)(rnmer)(g′)
182.      end end ▪
```

6.5.2 Adjustment of Creatable and Destructable Behaviours

When an entity is created or destroyed its creation, respectively destruction affects the neurologically related parts and their behaviours. their mereology and possibly their programmable state attributes need be adjusted. And when entities are destroyed their behaviours are **stop**ped! These entities are "informed" so by the creator/destructor entity – as was shown in Examples 76–78. The next example will illustrate how such 'affected' entities handle such creator/destructor communication.

Example 79 . Hub Adjustments: We have not yet illustrated hub (nor link) behaviours. Now we have to!

185 The mereology of a hub is a triple: the identification of the set of automobiles that may enter the hub, the identification of the set of links that connect to the hub, and the identification of the road net.
186 The hub behaviour external non-deterministically (⌷) alternates between
187 doing “own work”,
188 or accepting a stop “command” from the road net administrator, or
189 or accepting mereology & state update information,
190 or other.

```
type
185.  HMer = AI-set × LI-set × RNI
value
185.  mereo_H: H → HMer
186.  hub: hi:HI → (auis,lis,rni):HMer → hω:HΩ → (hσ:HΣ×ht:HTraffic) →
186.         {ch[hi,ui]|ui:(RNI|AI) • ui=rni∨ui ∈ auis}  Unit
186.  hub(hi)(hm:(auis,lis,rni))(hω)(hσ,ht) ≡
187.        ...
188.     ⌷ let mkStop() = ch[hi,rni] ? in stop end
189.     ⌷ let mkUpdH(hm′,hσ′,hσ′) = ch[{rni,hi}] ? in
189.        hub(hi)(hm′)(hω′)(hσ′,ht) end
190.        ...
```

Observe from formula Item 188 that the hub behaviour ends, whereas “from” Item 189 it tail recurses! ▪

6.5.3 Summary on Creatable & Destructable Entities

We have sketched how we may model the dynamics of creating and destroying entities. It is, but a sketch. We should wish for a more methodological account. So, that is what we are working on – amongst other issues – at the moment.

6.6 Domain Engineering: Description and Construction

There are two meanings to the term ‘Domain Engineering’.

- the construction of *descriptions* of domains, and
- the construction of *domains*.

Most sections of Chapters 4–6 are “devoted” to the former; the previous section, Sect. **6.5** to the latter.

6.7 Domain Laws

The[110] issue of *domain laws* seems to be crucial. Inklings of *domain laws* have been hinted at: (i) intentional pulls, Sect. **5.6** and (ii) Galois Connections (?), Sect. **5.6.5**.

MORE TO COME

---

[110] This section is currently under consideration.

6.8 A Domain Discovery Procedure, III

The predecessors of this section are Sects. **4.8.2** on page 56 and **5.7** on page 96.

6.8.1 Review of the Endurant Analysis and Description Process

The discover_... functions below were defined in Sects. **4.8.2** on page 56 and **5.7** on page 96.

```
value
    endurant_analysis_and_description: Unit → Unit
    endurant_analysis_and_description() ≡
          discover_sorts();                     [Page 56]
          discover_internal_endurant_qualities()  [Page 96]
```

We are now to define a perdurant_analysis_and_description procedure – to follow the above endurant_analysis_and_description procedure.

6.8.2 A Domain Discovery Process, III

We define the perdurant_analysis_and_description procedure in the reverse order of that of Sect. **5.7** on page 96, first the full procedure, then its sub-procedures.

A Domain Endurant Analysis and Description Process

```
value
    perdurant_analysis_and_description: Unit → Unit
    perdurant_analysis_and_description() ≡
          discover_state();                  axiom ...   [ Note (a) ]
          discover_channels();               axiom ...   [ Note (b) ]
          discover_behaviour_signatures();   axiom ...   [ Note (c) ]
          discover_behaviour_definitions();  axiom ...   [ Note (d) ]
          discover_initial_system()          axiom ...   [ Note (e) ]
```

Notes:

- (a) **The States: σ and ui_σ** We refer to Sect. **4.7.2** on page 55 and Sect. **5.2.4** on page 65. The state calculation, as shown on Page 54, must be replicated, i.e., re-discovered, in any separate domain analysis & description. The purpose of the state, i.e., σ, is to formulate appropriate axiomatic constraints and domain laws.
- (b) **The Channels:** We refer to Sects. **6.1.2** on page 101 and **6.2** on page 101. Thus we indiscriminately declare a channel for each pair of distinct unique part identifiers whether the corresponding pair of part behaviours, if at all invoked, communicate or not.
- (c) **Behaviour Signatures:** We refer to Sect. **6.3.1.2** on page 102. We find it more productive to first settle on the signatures of all behaviours – careful thinking has to go into that – before tackling the far more time-consuming work on defining the behaviours:
- (d) **Behaviour Definitions:** We refer to Sect. **6.3.3** on page 104.
- (e) **The Running System:** We refer to Sect. **6.4** on page 107.

6.9 Summary

Perdurants: Analysis & Description: Method Tools

- **Domain Discovery:** The procedures being described here, informally, guides the domain analyser cum describer to do the job !

  We have basically finished our listings of the procedural steps of the domain engineering methodology of this **tutorial** !

#	Name	Introduced
	Discovery Functions	
	discover_channels	page **101**
	discover_behaviour_signatures	page **104**
	discover_behaviour_definitions	page **105**
	discover_initial_system	page **107**
	perdurant_analysis_and_description	page **115**

• • •

Please consider Fig. 4.1 on page 39. This chapter has covered the right of Fig. 4.1.

Chapter 7
Closing

Contents

7.1 Axioms, Well-formedness and Proof Obligations

The reader may have noticed that this **tutorial** hardly mentions the notion of verification, yet domain descriptions, as possibly any specification related to software, may require some form of verification. Yet this **tutorial** appears to skirt the issue. Indeed, we have, regrettably, omitted the issue. So we must refer to reader to relevant literature. We cannot, October 19, 2022, point to any definitive book on the topic. The field is under intense research. Instead we refer to such diverse papers as: [60, 82].

In *endurant description prompts* 2 on page 45, 3 on page 48 and 4 on page 49 we mention the concept of proof obligation. They are also mentioned in *attribute description prompt* 7 on page 73. In numerous other places we mention the concept of *axiom*: 58 on page 65 (uniqueness of unique identifiers), 6 on page 68 (mereology), 103 on page 80 (disjointedness of attribute categories), 130 on page 86 (property of time), And in some places we mention the concept of *well-formedness,* f.ex., Sect. **5.6.2.4** on page 92,

- **Axioms** express properties of endurants, whether external or internal qualities, that holds – as were they laws of the domain.
- **Well-formedness** predicates are defined where external or internal qualities of endurants are defined by concrete types in such ways as to warrant such predicates.
- **Proof obligations** are usually warranted where distinct sort definitions need be separated.

7.2 From Programming Language Semantics to Domain Models

In 1973–1974, at the *IBM Vienna Laboratory*, we, Peter Lucas, Hans Bekič, Cliff Jones, Wolfgang Henhapl and I researched & developed a formal description of the `PL/I` programming language [7]. In 1979–1984, at the *Dansk Datamatik Center, DDC*, under my leadership and with invaluable help from my colleague, Dr. Hans Bruun, and based on my MSc. lectures, seven M.Sc. students[111] developed formal descriptions of (and later full compilers for) the `CHILL` [89] and `Ada` [58] programming languages.

In a domain model we describe essential nouns and verbs of the "language spoken" by practitioners of the domain. The "extension" from the language "spoken by programmers" to that "spoken by domain practitioners" should be obvious.

In both cases, the descriptions, for realistic programming languages and for realistic domains, are not trivial. They are sizable. The `PL/I, CHILL` and `Ada` descriptions span from a hundred pages to several hundred pages,! Similarly, their implementation, in terms of interpreters and compilers, took many man-years. For the *DDC* `Ada Compiler` it took 44 man-years ! [88, 68]

From a description of realistic facets of a domain one can develop a number of more-or-less distinct requirements, and from these one can develop computing systems software and we can expect similar size efforts.

7.3 Domain Specific Languages

A domain specific language, generically referred to as a `DSL`, is a language whose basic syntactic elements directly reflect endurants and perdurants of a specific domain. *Actulus*, a language in which to express calculations of actuarian character [66], is a DSL.

The semantics of a `DSL`, obviously, must relate to a model for the domain in question. In fact, we advice, that `DSL`s be developed from the basis of relevant domain models.

7.4 The `RAISE` Specification Language, `RSL`

We refer to Sect. **1.10** on page 5. So we have used `RSL` in two ways in this **tutorial**: (i) informally, to explain the domain analysis & description method – in $\texttt{RSL}^+$, and (ii) formally, to present [fragments of] specific domain specifications. The latter always in enumerated examples.[112] Appendices **A**–**B** both exemplify formal uses of `RSL`. All the functions listed in Index Sects. **D.6**–**D.8** and their explication are using the informal $\texttt{RSL}^+$.

7.5 Two Issues

We single out to issues for a very brief mentioning.

---

[111] Jørgen Bundgaard, Ole Dommergaard, Peter L. Haff, Hans Henrik Løvengreen, Jan Storbank Petersen, Søren Prehn, Lennart Schulz

[112] These are: Examples 35 on page 46, 36 on page 48, 37 on page 49, 40 on page 55, 42 on page 64, 43 on page 65, 44 on page 65, 45 on page 66, 46 on page 68, 47 on page 69, 50 on page 71, 57 on page 76, 58 on page 77, 59 on page 77, 66 on page 92, 72 on page 105, 73 on page 107, 75 on page 109, 76 on page 110, 77 on page 111, 78 on page 113, and 79 on page 113

7.5.1 Rôle of Algorithms

In all of the function formulation of domain phenomena, in this **tutorial**, You have not seen a single, interesting algorithm ![113] We need not apologize for that. There is a reason. The reason is that we almost only describe properties. To that end we make use of classical mathematical notions such as set comprehension, for example: $\{ a \mid a{:}A \cdot \mathcal{P}(a) \}$. The search for a appropriate a such that $\mathcal{P}(a)$ holds is often what requires, often beautiful algorithms. We refer to [113, 93, *Knuth and Harel*]. The need for clever algorithms, usually, first arise when designing software. Not in requirements engineering (cf. Sect. 7.7), but in software design. Then requirements prescriptions, also usually expressed in terms of set, list or map comprehension, or corresponding quantifications, need efficient implementations; hence clever algorithms.

7.5.2 CSP versus PDEs

To model the behaviour of discrete dynamic domains, such as are the main focus of this **tutorial**, we use the CSP process concept [100]. To model the behaviour of continuous dynamic domains, which we really have not, we suggest that You use methods of analysis, to wit: *[Partial] Differential Equations, PDEs*. Perhaps also some *Fuzzy Logic* [162, 110]. That is: We see this as the "dividing line" between discrete and continuous dynamic systems modelling: *CSP versus DPEs*. Appendix **B**, pages 151–168, puts forward a domain whose continuous dynamics need be formalised, for example using PDEs [71]. Mathematical modelling such as based on *Adaptive Control Theory* [3], *Stochastic Control Theory* [112] or maybe *Fuzzy Control* [121], like algorithmics, first be required as possible techniques when issues of correct continuous dynamics and optimisation arise, as when implementing certain requirements.

7.6 Domain Facets

There are other, additional methodological domain modelling steps. In [49, Chapter 8, Pages 205–240] we cover the notion of *domain facets*. 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. We there list intrinsics, support technologies, rules & regulations, scripts, license languages, management & organisation, and human behaviour. as such facets. The referenced chapter ([49, Chapter 8, Pages 205–240]) is traditional, programming methodological, in the sense that there is no [semi-]formal calculi involved, as in this primer's Chapters 4–5, I could wish for that !

7.7 Requirements Engineering

Domain modelling, to repeat, can be pursued for two different, but related, reasons. (i) simply, without any concern for, or idea of possible software, in order to "just" understand a domain, or (ii) for reasons of subsequent software development. In the later case a step of *requirements*

---

[113] Algorithm: a process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.

engineering need be pursued. [49, Chapter 9, Pages 243–298] covers a notion of *requirements engineering*. In that chapter we show three stages of requirements development: (α) *domain requirements*, (β) *interface requirements*, and (γ) *machine requirements*. But first a definition of the term *'machine'*. By machine we shall understand a, or the, combination of hardware and software that is the target for, or result of the required computing systems development. By a *requirements* we shall understand (cf., IEEE Standard 610.12): *"A condition or capability needed by a user to solve a problem or achieve an objective."* ■ By a *domain requirements* we shall understand those requirements which can be expressed sôlely using terms of the domain ■ By an *interface requirements* we shall understand those requirements which can be expressed only using technical terms of both the domain and the machine ■ By a *machine requirements* we shall understand those requirements which, in principle, can be expressed sôlely using terms of the machine ■

The *domain requirements* stage of requirements development starts with a basis in the domain engineering's domain description. It is, so-to-speak, a first step in the development of a requirements prescription.[114] From there follows, according to [49, Chapter 9] a number of (five) steps: (1.) *projection*: By projection is meant a subset of the domain description, one which projects out all those endurants: parts and fluids, as well as perdurants: actions, events and behaviours that the stake-holders do not wish represented or relied upon by the machine ■ (2.) *instantiation*: By instantiation we mean a refinement of the partial domain requirements prescription (resulting from the projection step) in which the refinements aim at rendering more concrete, more specific the endurants: parts and fluids, as well as the perdurants: actions, events and behaviours of the domain requirements prescription ■ (3.) *determination*: By determination we mean a refinement of the partial domain requirements prescription, resulting from the instantiation step, in which the refinements aim at rendering less non-determinate, more determinate the endurants: parts and fluids, as well as the perdurants: functions, events and behaviours of the partial domain requirements prescription ■ (4.) *extension*: By extension we understand the introduction of endurants and perdurants 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 ■ (5.) *fitting*: By requirements fitting we mean a harmonisation of two or more domain requirements that have overlapping (shared) not always consistent parts and which results in n partial domain requirement, and m$m$shared domain requirement, that "fit into" two or more of the partial domain requirements ■

[49, Chapter 9] then goes on to outline interface and machine requirements steps.

So domain engineering is a sound basis, we claim, for software development.

How that basis harmonises with the approaches taken by *Axel van Lamsweerde* [114] and *Michael A. Jackson* [108] is, really, a worthwhile study in-and-by itself !

---

[114] The "passage" from domain description to requirements prescription marks a transcendental deduction. Domain descriptions designate that which is being described. Requirements prescriptions designate what is intended to be implemented by computing. Please note the distinction: At the end of the development of a domain description we have just that: a domain description. At the beginning of the development of a requirements prescription we consider the domain description to be the initial requirements prescription: Thus, seemingly bewildering, in one instance a document is considered a domain description, in the next instance, without that document having been textually changed, it is now considered a requirements prescription. The transition from domain description to requirements prescription also marks a transition from "no-design mode" description to "design-mode" prescription.

7.8 Possible [PhD] Research Topics

I list here a number of possible (PhD) research topics:

1 **Intentional Pull:** This topic is not treated to the depth it deserves in this **tutorial**. Try think of intentional pulls in several domains: (i) *money flow in financial institutions* (while domain modelling a fair selection of such: banks, credit card companies, brokers, stock exchanges [34], etc.); (ii) *railway systems* (study, for example, [59, 55, 18, 124, 52, 152, 136, 17, 54]); and (iii) *container terminals* (see [44]).

2 **Discrete vs. Continuous Endurants and Perdurants:** Take the example of (oil, gas, water) *pipelines*. See Appendix **B**. Try model the dynamic flow of liquid in pipes, valves, pumps, etc., that is "mix", as may be expected, differential equations with RSL formulas. Some have tried. No real progress seems attained. Se however [160, 161]. The pipeline example should illustrate the use of monitorable attributes, their "reading" and their "biddable updates".
The challenge here is threefold: (i) first the PDE etc. modelling of the flow for each kind of unit, including curved pipe units; (ii) then for their composition – for a specific layout, for example that hinted at in Fig. **B**.1 on page 151; and (iii) finally for the infinite collection of pipeline systems such as defined by the "abstract syntax" of Appendix. **B** Item 290 on page 151 (including its wellformedness).

3 **Towards a Calculus of Perdurants:** This **tutorial** has unveiled the beginnings of a *Calculus of Endurants*. (Yet, its real "calculus-orientation" has yet to emerge: its laws, etc.) Sect. **6.3.3** hints at what I have in mind. A systematic analysis which aims at uncovering a fixed number of behaviour patterns such as sketched in Sect. **6.3.3**.

4 **Modelling Human Interaction:** The "running example", summarised in Appendix **A**, illustrated a road net "populated" with automobiles driving "hither & dither". The current **tutorial** has not treated the interaction between humans and man-made artifacts, like, for example, drivers and their automobiles. You are to model, for example, such human actions as starting an automobile, accelerating, braking. turning left, turning right, and stopping. Doing so You will have to try out, experiment with the rôles of monitorable, including biddable automobile attributes. An aim, besides such a domain model, is to research method issues of modelling human interaction. Please disregard modelling issues of sentiments, feelings, etc.

5 **Transcendental Deduction:** In the philosophy of Kai Sørlander such as, for example, explained in Chapter 2, transcendental deduction is appealed to repeatably. In this **tutorial**, as in [49], transcendental deduction is appealed to only once ! Maybe research into possible calculi for perdurants, cf. Research Challenge 3, might yield some more examples of transcendental deductions.

6 **Formal Models of Domain Modelling Calculi:** In [38] I attempted a first formal model of the domain analysis & description calculi. With [49] and, especially, this **tutorial** as a background, perhaps a more thorough attempt should be made to bring the model of [38] up-to-date and complete !

7 **Kai Sørlander's Philosophy:** We refer to Chapter 2. It is here strongly suggested that this research project be based on [149], Kai Sørlander's most recent book.[115] The challenge, in a sense, has two elements: (i) the identification of Sørlander's use of transcendental deduction: painstakingly identifying **all** it uses, analysing each of these, studying whether one can characterise these uses into more than one common kind of deduction, or whether one might claim *"classes of deductions"*, not necessarily disjoint, but perhaps structured

---

[115] All of Sørlander's books [145, 146, 147, 148, 149, 1994–2022] are in Danish – so the researcher would either be able to read Danish, or, more preferably to me, to have a suitable (German, English, French, ...) translation at hand.

in some kind of taxonomy; and (ii) the analysis of this report's presentation of Sørlander's metaphysics.

7.9 Acknowledgments

In [49, *Preface/Acknowledgments, Page xiv*] I acknowledged the very many who, over my professional life, has inspired me. In "rewriting" this primer from [49] I have, again, attempted to "capture" Kai Sørlander's Philosophy, cf. Chapter 2. And again I wishes to deeply acknowledge that work and, hence, **Kai Sørlander**. Here I, additionally,wishes to acknowledge, with pleasure, **Laura Kovacs**, TU Wien. Laura invited me to lecture, in the fall of 2022[116], at TU Wien. This **tutorial** is the result of that invitation.

• • •

7.10 Epilogue

The first inklings, in my work on what is now the *Domain Science & Engineering* of this **tutorial** appeared in [10, 11, 12, 13, 14, 1995-1996]. The *UN University's International Institute for Software Technology*, `UNU/IIST`, of which I was the first and founding director, conducted several domain engineering-based research & development projects, most of them under the leaderships of (the late) Søren Prehn and Chris W. George [83]. [30, 2008] touched upon the concept of *Domain Facets*, not covered in this **tutorial**, but in [49, 2021]. Two papers [31, 35, 2010] suggested reasonably relevant properties of domain descriptions. It was not until [42, 2017] that the analysis & description calculi of this **tutorial** emerged, and were refined in [45, 2019].

---

[116] Well, an invitation for Covid-19 year 2021 had to be postponed !

Chapter 8
Bibliography

Contents

8.1 Bibliographical Notes

I have not read 20 of the 30 citations given in Footnote 16, Pages 7–8. But I have studied some of Kant's, Russel's, Wittgenstein's and Popper's writings. The dictionaries [4, 64, 103], as well as [116], have followed me for years.

8.2 References

1. Scott Aaronson. Quantum Computing since Democritus. Cambridge University Press, 2013.
2. Sara Ahbel-Rappe. Socrates: A Guide for the Perplexed. A&C Black (Bloomsbury), ISBN 978-0-8264-3325-1, 2011.
3. Karl Johan Åström and B. Wittenmark. Adaptive Control. Addison-Wesley Publishing Company, 1989.
4. Rober Audi. The Cambridge Dictionary of Philosophy. Cambridge University Press, The Pitt Building, Trumpington Street, Cambridge CB2 1RP, England, 1995.
5. Jonathan Barnes. The Presocratic Philosophers: The Natural Philosophy of Heraclitus. Routledge, Taylor & Francis Group, 1982. 43–62.
6. Jonathan Barnes, editor. Complete Works of Aristotle. Princeton University Press: Bollingen Series, 1984.
7. Hans Bekič, Dines Bjørner, Wolfgang Henhapl, Cliff B. Jones, and Peter Lucas. A Formal Definition of a PL/I Subset. Technical Report 25.139, Vienna, Austria, 20 September 1974.
8. Benjamain Berger and Daniel Whistler. The Schelling Reader. Bloomsbury Publishing PLC, 2020.
9. George Berkeley. Philosophical Works, Including the Works on Vision. Everyman edition, London, 1975 (1713).
10. Dines Bjøner, Chris W. George, and Søren Prehn. Domain Analysis — a Prerequisite for Requirements Capture. Technical Report 37, UNU/IIST, P.O.Box 3058, Macau, February 1995. .
11. Dines Bjørner. New Software Technology Development. Technical Report 46, UNU/IIST, P.O.Box 3058, Macau, November 1995. International Symposium: New IT for Governance and Publication Administration, Beijing, China; organized by UNDDSMS, June 1996.
12. Dines Bjørner. Software Systems Engineering — From Domain Analysis to Requirements Capture [— an Air Traffic Control Example]. Technical Report 48, UNU/IIST, P.O.Box 3058, Macau, November 1995. Keynote paper for the *Asia Pacific Software Engineering Conference*, APSEC'95, Brisbane, Australia, 6–9 December 1995. .

13. Dines Bjørner. Infrastructure Software Systems. Technical Report 58, UNU/IIST, P.O.Box 3058, Macau, Dec 1996. Presentation solicited for the Academia Europae (AE/CWI/SMC) Symposium, Amsterdam 11–12 April, 1996. .
14. Dines Bjørner. New Software Development. Administrative/Technical Report 59, UNU/IIST, P.O.Box 3058, Macau, January 1996. Special *Theme* paper: *New Software Technology Development*. Paper was first prepared in September 1995 for an International Symposium: *New IT Applications for Governance and Public Administration*, convened by the UN's Department for Development Support and Management Service: UNDDSMS, Beijing, November 9–14, 1995. This symposium was subsequently moved (tentatively) to June 1996, same venue. .
15. 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.
16. 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. `www2.imm.dtu.dk/~dibj/themarket.pdf`.
17. 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. `www2.imm.dtu.dk/~dibj/ifac-dynamics.pdf`.
18. Dines Bjørner. TRain: The Railway Domain — A "Grand Challenge" for Computing Science and Transportation Engineering. In Topical Days @ IFIP World Computer Congress 2004, IFIP Series. IFIP, Kluwer Academic Press, August 2004.
19. Dines Bjørner. Software Engineering, Vol. 1: Abstraction and Modelling. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. See [24, 27].
20. Dines Bjørner. Software Engineering, Vol. 2: Specification of Systems and Languages. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. Chapters 12–14 are primarily authored by Christian Krog Madsen. See [25, 28].
21. Dines Bjørner. Software Engineering, Vol. 3: Domains, Requirements and Software Design. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. See [26, 29].
22. Dines Bjørner. A Container Line Industry Domain. `www.imm.dtu.dk/ db/container-paper.pdf`. Techn. report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, June 2007.
23. Dines Bjørner. From Domains to Requirements `www.imm.dtu.dk/ dibj/2008/ugo/ugo65.pdf`. 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.
24. Dines Bjørner. Software Engineering, Vol. 1: Abstraction and Modelling. Qinghua University Press, 2008.
25. Dines Bjørner. Software Engineering, Vol. 2: Specification of Systems and Languages. Qinghua University Press, 2008.
26. Dines Bjørner. Software Engineering, Vol. 3: Domains, Requirements and Software Design. Qinghua University Press, 2008.
27. Dines Bjørner. **Chinese:** Software Engineering, Vol. 1: Abstraction and Modelling. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.
28. Dines Bjørner. **Chinese:** Software Engineering, Vol. 2: Specification of Systems and Languages. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.
29. Dines Bjørner. **Chinese:** Software Engineering, Vol. 3: Domains, Requirements and Software Design. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.
30. 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.
31. 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.
32. Dines Bjørner. On Development of Web-based Software: A Divertimento of Ideas and Suggestions. Technical, Technical University of Vienna, August–October 2010. `www.imm.dtu.dk/~dibj/wfdftp.pdf`.
33. Dines Bjørner. The Tokyo Stock Exchange Trading Rules `www.imm.dtu.dk/~db/todai/tse-1.pdf`, `www.imm.dtu.dk/~db/todai/tse-2.pdf`. R&D Experiment, Techn. Univ. of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, 2010.
34. Dines Bjørner. The Tokyo Stock Exchange Trading Rules `www.imm.dtu.dk/~db/todai/tse-1.pdf`, `www.imm.dtu.dk/~db/todai/tse-2.pdf`. R&D Experiment, Techn. Univ. of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, January, February 2010.
35. 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.

36. Dines Bjørner. Pipelines – a Domain www.imm.dtu.dk/~dibj/pipe-p.pdf. Experimental Research Report 2013-2, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013.
37. 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.
38. Dines Bjørner. Domain Analysis: Endurants – An Analysis & Description Process Model www.imm.dtu.dk/ dibj/2014/kanazawa/kanazawa-p.pdf. 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.
39. Dines Bjørner. A Credit Card System: Uppsala Draft www.imm.dtu.dk/~dibj/2016/credit/accs.pdf. Technical Report: Experimental Research, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, November 2016.
40. Dines Bjørner. Weather Information Systems: Towards a Domain Description www.imm.dtu.dk/~dibj/2016/wis/wis-p.pdf. Technical Report: Experimental Research, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, November 2016.
41. Dines Bjørner. A Space of Swarms of Drones. www.imm.dtu.dk/~dibj/2017/swarms/swarm-paper.pdf. Research Note, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, December 2017.
42. Dines Bjørner. Manifest Domains: Analysis & Description www.imm.dtu.dk/ dibj/2015/faoc/faoc-bjorner.pdf. Formal Aspects of Computing, 29(2):175–225, March 2017. Online: 26 July 2016.
43. Dines Bjørner. What are Documents ? www.imm.dtu.dk/~dibj/2017/docs/docs.pdf. Research Note, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, July 2017.
44. Dines Bjørner. Container Terminals. www.imm.dtu.dk/ dibj/2018/yangshan/maersk-pa.pdf. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, September 2018. An incomplete draft report; currently 60+ pages.
45. Dines Bjørner. Domain Analysis & Description – Principles, Techniques and Modeling Languages. www.imm.dtu.dk/ dibj/2018/tosem/Bjorner-TOSEM.pdf. ACM Trans. on Software Engineering and Methodology, 28(2):66 pages, March 2019.
46. Dines Bjørner. *An Assembly Plant Domain – Analysis & Description*, www.imm.dtu.dk/ dibj/2021/assembly/assembly-line.pdf. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, September 2019.
47. Dines Bjørner. A Retailer Market: Domain Analysis & Description. A Comparison Heraklit/DS&E Case Study. www.imm.dtu.dk/ dibj/2021/Retailer/BjornerHeraklit27January2021.pdf. Technical Report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, January 2021.
48. Dines Bjørner. Automobile Assembly Plants. www.imm.dtu.dk/~dibj/2021/assembly/assembly-line.pdf. Technical Report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, Summer 2021.
49. Dines Bjørner. Domain Science & Engineering – A Foundation for Software Development. EATCS Monographs in Theoretical Computer Science. Springer, 2021.
50. Dines Bjørner. Rivers and Canals. www.imm.dtu.dk/~dibj/2021/Graphs/Rivers-and-Canals.pdf. Technical Report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, March 2021.
51. Dines Bjørner. Shipping. www.imm.dtu.dk/~dibj/2021/ral/ral.pdf. Technical Report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, April 2021.
52. Dines Bjørner, Peter Chiang, Morten S.T. Jacobsen, Jens Kielsgaard Hansen, Michael P. Madsen, and Martin Penicka. Towards a Formal Model of CyberRail. In Topical Days @ IFIP World Computer Congress 2004, IFIP Series. IFIP, Kluwer Academic Press, August 2004.
53. 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.
54. 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. www2.imm.dtu.dk/~dibj/pasadena-25.pdf.
55. Dines Bjørner, C.W. George, B.Stig Hansen, H. Laustrup, and S. Prehn. A Railway System, Coordination'97, Case Study Workshop Example. Research Report 93, UNU/IIST, P.O.Box 3058, Macau, January 1997. .
56. Dines Bjørner and Cliff B. Jones, editors. The Vienna Development Method: The Meta-Language, volume 61 of LNCS. Springer, 1978.
57. Dines Bjørner and Cliff B. Jones, editors. Formal Specification and Software Development. Prentice-Hall, 1982.

58. Dines Bjørner and Ole N. Oest, editors. Towards a Formal Description of Ada, volume 98 of LNCS. Springer, 1980.
59. Dines Bjørner and Martin Pčnička. Towards a TRAIN Book for The RAIlway DomaiN. Techn. reports, www.railwaydomain.org/PDF/tb.pdf, The TRAIN Consortium, 2004.
60. Nikolaj Bjørner, Maxwell Levatich, Nuno P. Lopes, Andrey Rybalchenko, and Chandrasekar Vuppalapati. Supercharging plant configurations using Z3. In Peter J. Stuckey, editor, Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 18th International Conference, CPAIOR 2021, Vienna, Austria, July 5-8, 2021, Proceedings, volume 12735 of Lecture Notes in Computer Science, pages 1–25. Springer, 2021.
61. Dines Bjørner. Urban Planning Processes. `www.imm.dtu.dk/~dibj/2017/up/urban-planning.pdf`. Research Note, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, July 2017.
62. 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 .
63. Wayne D. Blizard. A Formal Theory of Objects, Space and Time. The Journal of Symbolic Logic, 55(1):74–89, March 1990.
64. 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.
65. Roberto Casati and Achille C. Varzi. Parts and Places: the structures of spatial representation. MIT Press, 1999.
66. 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, `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.
67. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí Oliet, José Meseguer, and Carolyn Talcott. Maude 2.6 Manual. Department of Computer Science, University of Illinois and Urbana-Champaign, Urbana-Champaign, Ill., USA, January 2011.
68. Geert Bagge Clemmensen and Ole N. Oest. Formal specification and development of an Ada compiler – a VDM case study. In Proc. 7th International Conf. on Software Engineering, 26.-29. March 1984, Orlando, Florida, pages 430–440. IEEE, 1984.
69. S. Marc Cohen. Aristotle's Metaphysics. In Stanford Encyclopedia of Philosophy. Center for the Study of Language and Information, Stanford University Stanford, CA, November 2018. The Metaphysics Research Lab.
70. Dirk L. Couprie and Radim Kocandrle. Anaximander: Anaximander on Generation and Destruction. x, Springer (Briefs in Philosophy Series).
71. Richard Courant and Fritz John. Introduction to Analysis and Calculus, I–II/1. Springer(Wiley 1974), December 1989, 1998. 'Classics in Mathematics' Series.
72. 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.
73. O.-J. Dahl, E.W. Dijkstra, and Charles Anthony Richard Hoare. Structured Programming. Academic Press, 1972.
74. René Descartes. Discours de la méthode. Texte et commentaire par Étienne Gilson. Paris: Vrin, 1987.
75. Asger Eir. Construction Informatics — issues in engineering, computer science, and ontology. PhD thesis, Dept. of Computer Science and Engineering, Institute of Informatics and Mathematical Modeling, Technical University of Denmark, Building 322, Richard Petersens Plads, DK–2800 Kgs.Lyngby, Denmark, February 2004.
76. Asger Eir. Formal Methods and Hybrid Real-Time Systems, chapter Relating Domain Concepts Intensionally by Ordering Connections, pages 188–216. Springer (LNCS Vol. 4700, Festschridt: Essays in Honour of Dines Bjørner and Zhou Chaochen on the Occasion of Their 70th Birthdays), 2007.
77. William David Ross et al. Plato's Theory of Ideas. Oxford University Press, 1963.
78. 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.
79. 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.
80. Carlo A. Furia, Dino Mandrioli, Angelo Morzenti, and Matteo Rossi. Modeling Time in Computing. Monographs in Theoretical Computer Science. Springer, 2012.

81. 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.
82. Kokichi Futatsugi. Advances of proof scores in CafeOBJ. Science of Computer Programming, 224, December 2022.
83. Chris George. Applicative modelling with RAISE. In Chris George, Zhiming Liu, and Jim Woodcock, editors, Domain Modeling and the Duration Calculus, International Training School, Shanghai, China, September 17-21. 2007, Advanced Lectures, volume 4710 of Lecture Notes in Computer Science, pages 51–118. Springer, 2007.
84. 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.
85. 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.
86. James Gosling and Frank Yellin. The Java Language Specification. Addison-Wesley & Sun Microsystems. ACM Press Books, 1996. 864 pp, ISBN 0-10-63451-1.
87. P. Guyer, editor. The Cambridge Companion to Kant. Cambridge Univ. Press, England, 1992.
88. P. Haff and A.V. Olsen. Use of VDM within CCITT. In VDM – A Formal Method at Work, eds. Dines Bjørner, Cliff B. Jones, Micheal Mac an Airchinnigh and Erich J. Neuhold, pages 324–330. Springer, Lecture Notes in Computer Science, Vol. 252, March 1987. Proc. VDM-Europe Symposium 1987, Brussels, Belgium.
89. Peter Haff. A Formal Definition of CHILL. A Supplement to the CCITT Recommendation Z.200. Technical report, Dansk Datamatik Center, Lyngby, Denmark, Dansk Datamatik Center, Lyngby, Denmark, 1980.
90. Michael Reichhardt Hansen and Hans Rischel. Functional Programming in Standard ML. Addison Wesley, 1997.
91. Michael Reichhardt Hansen and Hans Rischel. Functional Programming Using F#. Cambridge University Press, 2013.
92. G. H. Hardy, Edward M. Wright, and John Silvermann. An Introduction to the Theory of Numbers. Oxford University Press, England, 6th edition edition, 2008. Editor: Roger Heath Brown.
93. David Harel. Algorithmics —The Spirit of Computing. Addison-Wesley, 1987.
94. R. Harper, D. MacQueen, and R. Milner. Standard ML. Technical Report ECS-LFCS-86-2, Lab. f. Found. of Comp. Sci., Dept. of Comp. Sci., Univ. of Edinburgh, Scotland, 1986.
95. Georg Wilhelm Friedrich Hegel. Wissenschaft der Logik. Hofenberg, 2016 (1812–1816).
96. Martin Heidegger. Sein und Zeit (Being and Time). Oxford University Press, 1927, 1962.
97. Martin Heidegger. Parminedes. Indiana University Press, 1998.
98. Charles Anthony Richard Hoare. Notes on Data Structuring. In [73], pages 83–174, 1972.
99. Charles Anthony Richard Hoare. Communicating Sequential Processes. Communications of the ACM, 21(8), Aug. 1978.
100. Charles Anthony Richard Hoare. Communicating Sequential Processes. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985. Published electronically: `usingcsp.com/cspbook.pdf` (2004).
101. Charles Anthony Richard Hoare. Communicating Sequential Processes. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985.
102. Gerard J. Holzmann. The SPIN Model Checker, Primer and Reference Manual. Addison-Wesley, Reading, Massachusetts, 2003.
103. Ted Honderich. The Oxford Companion to Philosophy. Oxford University Press, Walton St., Oxford OX2 6DP, England, 1995.
104. David Hume. Enquiry Concerning Human Understanding. Squashed Editions, 2020 (1758).
105. Edmund Husserl. Ideas. General Introduction to Pure Phenomenology. Routledge, 2012.
106. Daniel Jackson. Software Abstractions: Logic, Language, and Analysis. The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-10114-9.
107. Michael A. Jackson. Software Requirements & Specifications: a lexicon of practice, principles and prejudices. ACM Press. Addison-Wesley, Reading, England, 1995.
108. 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.
109. David James and Gunter Zoller. Cambridge Companion to Fichte. Cambridge University Press, 2016.
110. James J. Buckley and Esfanidar Eslami. An Introduction to Fuzzy Logic and Fuzzy Sets. Springer, 2002.
111. Immanuel Kant. Critique of Pure Reason. Penguin Books Ltd, 2007 (1787).

112. Samuel Karlin and Howard M. Taylor. An Introduction to Stochastic Modeling. Academic Press, 1998. ISBN 0-12-684887-4.
113. D.E. Knuth. The Art of Computer Programming, 3 vols: 1: Fundamental Algorithms, 2: Seminumerical Algorithms, 3: Searching & Sorting. Addison-Wesley, Reading, Mass., USA, 1968, 1969, 1973; newly revised 2000.
114. Axel van Lamsweerde. Requirements Engineering: from system goals to UML models to software specifications. Wiley, 2009.
115. Paul Lindgreen. Systemanalyse og systembeskrivelse (System Analysis and System Description). Samfundslitteratur, 1983.
116. 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.
117. John Locke. An Essay Concerning Human Understanding. Penguin Classics, 1998 (1689).
118. Theodore McCombs. Maude 2,0 Primer. Department of Computer Science, University of Illinois and Urbana-Champaign, Urbana-Champaign, Ill., USA, August 2003.
119. J. M. E. McTaggart. The Unreality of Time. Mind, 18(68):457–84, October 1908. New Series. See also: [126].
120. J. Edward Mercer. The Mysticism Of Anaximenes And The Air. Kessinger Publishing, LLC, 2010.
121. Kai Michels, Frank Klawonn, Rudolf Kruse, and Andreas Nürnberger. Fuzzy Control: Fundamentals, Stability and Design of Fuzzy Controllers. Springer, 19 October 2010.
122. R. Milner, M. Tofte, and R. Harper. The Definition of Standard ML. The MIT Press, Cambridge, Mass., USA and London, England, 1990.
123. Patricia O'Grady. Thales of Miletus. Routledge (Western Philosophy Series), 2002.
124. Martin Penicka. From Railway Resource Planning to Train Operation. In Topical Days @ IFIP World Computer Congress 2004, IFIP Series. IFIP, Kluwer Academic Press, August 2004.
125. Benjamin Pierce. Types and Programming Languages. The MIT Press, 2002.
126. Robin Le Poidevin and Murray MacBeath, editors. The Philosophy of Time. Oxford University Press, 1993.
127. Karl R. Popper. Logik der Forschung. Julius Springer Verlag, Vienna, Austria, 1934 (1935). English version [128].
128. Karl R. Popper. The Logic of Scientific Dicovery. Hutchinson of London, 3 Fitzroy Square, London W1, England, 1959,. . . ,1979. Translated from [127].
129. Karl R. Popper. Conjectures and Refutations. The Growth of Scientific Knowledge. Routledge and Kegan Paul Ltd. (Basic Books, Inc.), 39 Store Street, WC1E 7DD, London, England (New York, NY, USA), 1963,. . . ,1981.
130. Arthur Prior. Changes in Events and Changes in Things, chapter in [126]. Oxford University Press, 1993.
131. Arthur N. Prior. Logic and the Basis of Ethics. Clarendon Press, Oxford, UK, 1949.
132. Arthur N. Prior. Formal Logic. Clarendon Press, Oxford, UK, 1955.
133. Arthur N. Prior. Time and Modality. Oxford University Press, Oxford, UK, 1957.
134. Arthur N. Prior. Past, Present and Future. Clarendon Press, Oxford, UK, 1967.
135. Arthur N. Prior. Papers on Time and Tense. Clarendon Press, Oxford, UK, 1968.
136. 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. `www2.imm.dtu.dk/~dibj/martin.pdf`.
137. 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.
138. Hartley R. Rogers. Theory of Recursive Functions and Effective Computability. McGraw-Hill, 1967.
139. A. W. Roscoe. Theory and Practice of Concurrency. C.A.R. Hoare Series in Computer Science. Prentice-Hall, 1997. `http://www.comlab.ox.ac.uk/people/bill.roscoe/publications/68b.pdf`.
140. Bertrand Russell. On Denoting. Mind, 14:479–493, 1905.
141. Bertrand Russell. The Problems of Philosophy. Home University Library, London, 1912. Oxford University Press paperback, 1959 Reprinted, 1971-2.
142. Bertrand Russell. Introduction to Mathematical Philosophy. George Allen and Unwin, London, 1919.
143. 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.
144. Peter Sestoft. Java Precisely. The MIT Press, 25 July 2002.
145. Kai Sørlander. Det Uomgængelige – Filosofiske Deduktioner [The Inevitable – Philosophical Deductions, with a foreword by Georg Henrik von Wright]. Munksgaard · Rosinante, Copenhagen, Denmark, 1994. 168 pages.
146. Kai Sørlander. Under Evighedens Synsvinkel [Under the viewpoint of eternity]. Munksgaard · Rosinante, Copenhagen, Denmark, 1997. 200 pages.

147. Kai Sørlander. Den Endegyldige Sandhed [The Final Truth]. Rosinante, Copenhagen, Denmark, 2002. 187 pages.
148. Kai Sørlander. Indføring i Filosofien [Introduction to The Philosophy]. Informations Forlag, Copenhagen, Denmark, 2016. 233 pages.
149. Kai Sørlander. Den rene fornufts struktur [The Structure of Pure Reason]. Ellekær, Slagelse, Denmark, 2022.
150. Baruch Spinoza. Ethics, Demonstrated in Geometrical Order. The Netherlands, 1677.
151. Steven Weintraub. Galois Theory. Springer, 2009.
152. 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. `www2.imm.dtu.dk/˜dibj/albena.pdf`.
153. 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.
154. 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.
155. N. Wirth. The Programming Language Oberon. Software — Practice and Experience, 18:671–690, 1988.
156. Ludwig Johan Josef Wittgenstein. Tractatus Logico–Philosophicus. Oxford Univ. Press, London, (1921) 1961.
157. Ludwig Johan Josef Wittgenstein. Philosophical Investigations. Oxford Univ. Press, 1958.
158. James Charles Paul Woodcock and James Davies. Using Z: Specification, Proof and Refinement. Prentice Hall International Series in Computer Science, 1996.
159. M.R. Wright. Empedokles: The Extant Fragments. Hackett Publishing Company, Inc., 1995.
160. WanLing Xie, ShuangQing Xiang, and HuiBiao Zhu. A UTP approach for rTiMo. Formal Ascpects of Computing, 30(6):713–738, 2018.
161. WanLing Xie, HuiBiao Zhu, and Xu QiWen. A process calculus BigrTiMo of mobile systems and its formal semantics. Formal Ascpects of Computing, 33(2):207–249, March 2021.
162. Lotfi A. Zadeh. Fuzzy sets. Information and Control, 8(3):338–353, 1965.

Appendix A
Road Transport

Contents

A.1 The Road Transport Domain

Our universe of discourse in this chapter is the road transport domain. Not a specific one, but "a generic road transport domain".

A.1.1 Naming

type RTS

A.1.2 Rough Sketch

The generic road transport domain that we have in mind consists of a road net (aggregate) and an aggregate of vehicles such that the road net serves to convey vehicles. We consider the road net to consist of hubs, i.e., street intersections, or just street segment connection points, and links, i.e., street segments between adjacent hubs. We consider the aggregate of vehicles to include in addition to vehicales, i.e., automobiles, a department of motor vehicles (DMVs), zero or more bus companies, each with zero, one or more buses, and vehicle associations, each with zero, one or more members who are owners of zero, one or more vehicles[1] ▪

A.2 External Qualities

A Road Transport System, I – Manifest External Qualities: Our intention is that the manifest external qualities of a road transport system are those of its roads, their **hub**s[2]i.e., road (or street) intersections, and their **link**s, i.e., the roads (streets) between hubs, and **vehicle**s, i.e., automobiles – that ply the roads – the buses, trucks, private cars, bicycles, etc. ▪

---

[1] This "rough" narrative fails to narrate what hubs, links, vehicles, DMVs, bus companies, buses and vehicle associations are. In presenting it here, as we are, we rely on your a priori understanding of these terms. But that is dangerous! The danger, if we do not painstakingly narrate and formalise what we mean by all these terms, then readers (software designers, etc.) may make erroneous assumptions.

[2] We have **highlighted** certain endurant sort names – as they will re-appear in rather many upcoming examples.

A.2.1 A Road Transport System, II – Abstract External Qualities

Examples of what could be considered abstract external qualities of a road transport domain are: the aggregate of all hubs and all links, the aggregate of all buses, say into bus companies, the aggregate of all bus companies into public transport, and the aggregate of all vehicles into a department of vehicles. Some of these aggregates may, at first be treated as abstract. Subsequently, in our further analysis & description we may decide to consider some of them as concretely manifested in, for example, actual departments of roads.

A.2.2 Transport System Structure

A transport system is modeled 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 modeled as set of hubs, respectively links.

We could have modeled the road net *structure* as a *composite part* with *unique identity, mereology* and *attributes* which could then serve to model a `road net authority`. And we could have modeled the automobile *structure* as a *composite part* with *unique identity, mereology* and *attributes* which could then serve to model a `department of vehicles` ▪

A.2.3 Atomic Road Transport Parts

From one point of view all of the following can be considered atomic parts: hubs, links[3], and automobiles.

A.2.4 Compound Road Transport Parts

A.2.4.1 The Composites

191 There is the *universe of discourse*, UoD.
It is structured into
192 a *road net*, RN, and
193 a *fleet of vehicles*, FV.

Both are structures. ....

type
191 UoD **axiom** ∀ uod:UoD • is_structure(uod)
192 RN **axiom** ∀ rn:RN • is_structure(rn).
193 FV **axiom** ∀ fv:FV • is_structure(fv).

value
192 obs_RN: UoD → RN
193 obs_FV: UoD → FV ▪

A.2.4.2 The Part Parts

194 The structure of hubs is a set, sH, of atomic hubs, H.
195 The structure of links is a set, sL, of atomic links, L.

---

[3] Hub ≡ street intersection; link ≡ street segments with no intervening hubs.

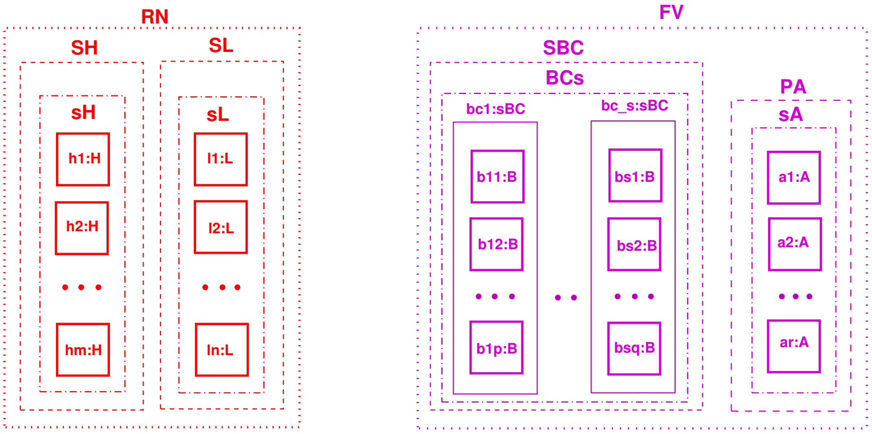


Fig. A.1 A Road Transport System Compounds and Structures

196 The structure of buses is a set, sBC, of composite bus companies, BC.
197 The composite bus companies, BC, are sets of buses, sB.
198 The structure of private automobiles is a set, sA, of atomic automobiles, A.

```
type
194  H, sH = H-set  axiom ∀ h:H • is_atomic(h)
195  L, sL = L-set  axiom ∀ l:L • is_atomic(l)
196  BC, BCs = BC-set  axiom ∀ bc:BC • is_composite(bc)
197  B, Bs = B-set  axiom ∀ b:B • is_atomic(b)
198  A, sA = A-set  axiom ∀ a:A • is_atomic(a)
value
194  obs_sH: SH → sH
195  obs_sL: SL → sL
196  obs_sBC: SBC → BCs
197  obs_Bs: BCs → Bs
198  obs_sA: SA → sA  ▪
```

A.2.5 The Transport System State

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

From that state we can calculate other states.

200 The set of all hubs, *hs*.
201 The set of all links, *ls*.
202 The set of all hubs and links, *hls*.
203 The set of all bus companies, *bcs*.
204 The set of all buses, *bs*.
205 The set of all private automobiles, *as*.
206 The set of all parts, *ps*.

```
value
199 rts:UoD    [43]
200 hs:H-set ≡:H-set ≡ obs_sH(obs_SH(obs_RN(rts)))
201 ls:L-set ≡:L-set ≡ obs_sL(obs_SL(obs_RN(rts)))
202 hls:(H|L)-set ≡ hs∪ls
203 bcs:BC-set ≡ obs_BCs(obs_SBC(obs_FV(obs_RN(rts))))
204 bs:B-set ≡ ∪{obs_Bs(bc)|bc:BC•bc ∈ bcs}
205 as:A-set ≡ obs_BCs(obs_SBC(obs_FV(obs_RN(rts))))
206 ps:(UoB|H|L|BC|B|A)-set ≡ rts∪hls∪bcs∪bs∪as
```

A.3 Internal Qualities

A.3.1 Unique Identifiers

207 We assign unique identifiers to all parts.
208 By a road identifier we shall mean a link or a hub identifier.
209 By a vehicle identifier we shall mean a bus or an automobile identifier.
210 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
207 H_UI, L_UI, BC_UI, B_UI, A_UI
208 R_UI = H_UI | L_UI
209 V_UI = B_UI | A_UI
value
210a uid_H: H → H_UI
210b uid_L: H → L_UI
210c uid_BC: H → BC_UI
210d uid_B: H → B_UI
210e uid_A: H → A_UI
```

A.3.1.1 Extract Parts from Their Unique Identifiers

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

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

A.3.1.2 All Unique Identifiers of a Domain

We can calculate:

212 the set, $h_{ui}s$, of *u*nique *h*ub *i*dentifiers;

213 the *s*et, $l_{ui}s$, of *u*nique *l*ink *i*dentifiers;
214 the *m*ap, $hl_{ui}m$, from *u*nique *h*ub *i*dentifiers to the *s*et of *u*nique *l*ink *i*identifiers of the links connected to the zero, one or more identified hubs,
215 the *m*ap, $lh_{ui}m$, from *u*nique *l*ink *i*dentifiers to the *s*et of *u*nique *h*ub *i*identifiers of the two hubs connected to the identified link;
216 the *s*et, $r_{ui}s$, of all *u*nique hub and link, i.e., *r*oad *i*dentifiers;
217 the *s*et, $bc_{ui}s$, of *u*nique *b*us *c*ompany *i*dentifiers;
218 the *s*et, $b_{ui}s$, of *u*nique *b*us *i*dentifiers;
219 the *s*et, $a_{ui}s$, of *u*nique private *a*utomobile *i*dentifiers;
220 the *s*et, $v_{ui}s$, of *u*nique bus and automobile, i.e., *v*ehicle *i*dentifiers;
221 the *m*ap, $bcb_{ui}m$, from *u*nique *b*us *c*ompany *i*dentifiers to the *s*et of its *u*nique *b*us *i*dentifiers; and
222 the (*b*ijective) *m*ap, $bbc_{ui}bm$, from *u*nique *b*us *i*dentifiers to their *u*nique *b*us *c*ompany *i*dentifiers.

value
212 $h_{ui}s$:H_UI-**set** ≡ {uid_H(h)|h:H•h ∈ hs}
213 $l_{ui}s$:L_UI-**set** ≡ {uid_L(l)|l:L•l ∈ ls}
216 $r_{ui}s$:R_UI-**set** ≡ $h_{ui}s \cup l_{ui}s$
214 $hl_{ui}m$:(H_UI $\underset{\overline{m}}{\rightarrow}$ L_UI-**set**) ≡
214 [h_ui↦luis|h_ui:H_UI,luis:L_UI-**set**•h_ui∈$h_{ui}s$∧(_,luis,_)=mereo_H(η(h_ui))] [cf. Item 229]
215 $lh_{ui}m$:(L+UI $\underset{\overline{m}}{\rightarrow}$ H_UI-**set**) ≡
215 [l_ui↦huis | h_ui:L_UI,huis:H_UI-**set** • l_ui∈$l_{ui}s$ ∧ (_,huis,_)=mereo_L(η(l_ui))] [cf. Item 230]
217 $bc_{ui}s$:BC_UI-**set** ≡ {uid_BC(bc)|bc:BC•bc ∈ bcs}
218 $b_{ui}s$:B_UI-**set** ≡ ∪{uid_B(b)|b:B•b ∈ bs}
219 $a_{ui}s$:A_UI-**set** ≡ {uid_A(a)|a:A•a ∈ as}
220 $v_{ui}s$:V_UI-**set** ≡ $b_{ui}s \cup a_{ui}s$
221 $bcb_{ui}m$:(BC_UI $\underset{\overline{m}}{\rightarrow}$ B_UI-**set**) ≡
221 [bc_ui ↦ buis | bc_ui:BC_UI, bc:BC • bc∈bcs ∧ bc_ui=uid_BC(bc) ∧ (_,_,buis)=mereo_BC(bc)]
222 $bbc_{ui}bm$:(B_UI $\underset{\overline{m}}{\rightarrow}$ BC_UI) ≡
222 [b_ui ↦ bc_ui | b_ui:B_UI,bc_ui:BC_ui • bc_ui=**dom**$bcb_{ui}m$∧b_ui∈$bcb_{ui}m$(bc_ui)]

A.3.1.3 Uniqueness of Road Net Identifiers

We must express the following axioms:

223 All hub identifiers are distinct.
224 All link identifiers are distinct.
225 All bus company identifiers are distinct.
226 All bus identifiers are distinct.
227 All private automobile identifiers are distinct.
228 All part identifiers are distinct.

axiom
223 **card** hs = **card** $h_{ui}s$
224 **card** ls = **card** $l_{ui}s$
225 **card** bcs = **card** $bc_{ui}s$
226 **card** bs = **card** $b_{ui}s$
227 **card** as = **card** $a_{ui}s$
228 **card** $\{h_{ui}s \cup l_{ui}s \cup bc_{ui}s \cup b_{ui}s \cup a_{ui}s\}$
228 = **card** $h_{ui}s$+**card** $l_{ui}s$+**card** $bc_{ui}s$+**card** $b_{ui}s$+**card** $a_{ui}s$ ▪

A.3.2 Mereology

A.3.2.1 Mereology Types and Observers

229 The mereology of hubs is a pair: (i) the set of all bus and automobile identifiers[4], and (ii) the set of unique identifiers of the links that it is connected to and the set of all unique identifiers of all vehicles (buses and private automobiles).[5]
230 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.
231 The mereology of a bus company is a set the unique identifiers of the buses operated by that company.
232 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[6].
233 The mereology of an automobile is the set of the unique identifiers of all links and hubs[7].

type
229 H_Mer = V_UI-**set**×L_UI-**set**
230 L_Mer = V_UI-**set**×H_UI-**set**
231 BC_Mer = B_UI-**set**
232 B_Mer = BC_UI×R_UI-**set**
233 A_Mer = R_UI-**set**

value
229 mereo_H: H → H_Mer
230 mereo_L: L → L_Mer
231 mereo_BC: BC → BC_Mer
232 mereo_B: B → B_Mer
233 mereo_A: A → A_Mer

A.3.2.2 Invariance of Mereologies

For mereologies one can usually express some invariants. Such invariants express *"law-like properties"*, facts which are indisputable.

A.3.2.2.1 Invariance of Road Nets

The observed mereologies must express identifiers of the state of such for road nets:

axiom
229 ∀ (vuis,luis):H_Mer • luis⊆$l_{ui}s$ ∧ vuis=$v_{ui}s$
230 ∀ (vuis,huis):L_Mer • vuis=$v_{ui}s$ ∧ huis⊆$h_{ui}s$ ∧ **card**huis=2
231 ∀ buis:H_Mer • buis = $b_{ui}s$
232 ∀ (bc_ui,ruis):H_Mer•bc_ui∈$bc_{ui}s$∧ruis=$r_{ui}s$
233 ∀ ruis:A_Mer • ruis=$r_{ui}s$

234 For all hubs, *h*, and links, *l*, in the same road net,
235 if the hub *h* connects to link *l* then link *l* connects to hub *h*.

axiom
234 ∀ h:H,l:L • h ∈ *hs* ∧ l ∈ *ls* ⇒

---

[4] This is just another way of saying that the meaning of hub mereologies involves the unique identifiers of all the vehicles that might pass through the hub `is_of_interest` to it.

[5] The link identifiers designate the links, zero, one or more, that a hub is connected to `is_of_interest` to both the hub and that these links is `interested` in the hub.

[6] — that the bus might pass through

[7] — that the automobile might pass through

```
234     let (_,luis)=mereo_H(h), (_,huis)=mereo_L(l)
235     in uid_L(l)∈luis ≡ uid_H(h)∈huis end
```

236 For all links, l, and hubs, h_a,h_b, in the same road net,
237 if the l connects to hubs h_a and h_b, then h_a and h_b both connects to link l.

```
axiom
236  ∀ h_a,h_b:H,l:L • {h_a,h_b} ⊆ hs ∧ l ∈ ls ⇒
236     let (_,luis)=mereo_H(h), (_,huis)=mereo_L(l)
237     in uid_L(l)∈luis ≡ uid_H(h)∈huis end
```

A.3.2.2.2 Possible Consequences of a Road Net Mereology

238 are there [isolated] units from which one can not "reach" other units ?
239 does the net consist of two or more "disjoint" nets ?
240 et cetera.

We leave it to the reader to narrate and formalise the above properly.

A.3.2.2.3 Fixed and Varying Mereology

Let us consider a road net. If hubs and links never change "affiliation", that is: hubs are in fixed relation to zero one or more links, and links are in a fixed relation to exactly two hubs then the mereology is a *fixed mereology*. If, on the other hand hubs may be inserted into or removed from the net, and/or links may be removed from or inserted between any two existing hubs, then the mereology is a *varying mereology*.

A.3.3 Attributes

A.3.3.1 Hub Attributes

We treat some attributes of the hubs of a road net.

241 There is a hub state. It is a set of pairs, $(\mathsf{l}_f,\mathsf{l}_t)$, of link identifiers, where these link identifiers are in the mereology of the hub. The meaning of the hub state in which, e.g., $(\mathsf{l}_f,\mathsf{l}_t)$ is an element, is that the hub is open, **"green"**, for traffic *f*rom link l_f *t*o 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.
242 There is a hub state space. It is a set of hub states. The current hub state must be in its state space. The meaning of the hub state space is that its states are all those the hub can attain.
243 Since we can think rationally about it, it can be described, hence we 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. Hub history is an *event history*.

```
type
241  HΣ = (L_UI×L_UI)-set
axiom
```

241 $\forall$ h:H • obs_HΣ(h) $\in$ obs_HΩ(h)
type
242 HΩ = HΣ-**set**
243 H_Traffic
243 H_Traffic = (A_UI|B_UI) $\underset{m}{\rightarrow}$ ($\mathbb{TIME}$ $\times$ VPos)*
axiom
243 $\forall$ ht:H_Traffic,ui:(A_UI|B_UI) •
243 ui $\in$ **dom** ht $\Rightarrow$ time_ordered(ht(ui))
value
241 attr_HΣ: H $\rightarrow$ HΣ
242 attr_HΩ: H $\rightarrow$ HΩ
243 attr_H_Traffic: H $\rightarrow$ H_Traffic
value
243 time_ordered: ($\mathbb{TIME}$ $\times$ VPos)* $\rightarrow$ **Bool**
243 time_ordered(tvpl) $\equiv$ ...

In Item 243 on the facing page we model the time-ordered sequence of traffic as a discrete sampling, i.e., $\underset{m}{\rightarrow}$, rather than as a continuous function, →.

A.3.3.2 Invariance of Traffic States

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

axiom
244 $\forall$ h:H • h $\in$ hs $\Rightarrow$
244 **let** hσ = attr_HΣ(h) **in**
244 $\forall$ (l$_{ui}i$,li$_{ui}i'$):(L_UI×L_UI) • (l$_{ui}i$,l$_{ui}i'$) $\in$ hσ $\Rightarrow$ {l$_{ui_i}$,l$'_{ui_i}$} $\subseteq$ l$_{ui}s$ **end**

A.3.3.3 Link Attributes

We show just a few attributes.

245 There is a link state. It is a set of pairs, (h$_f$,h$_t$), of distinct hub identifiers, where these hub identifiers are in the mereology of the link. The meaning of a link state in which (h$_f$,h$_t$) is an element is that the link is open, **"green"**, for traffic $f$rom hub h$_f$ to hub h$_t$. Link states can have either 0, 1 or 2 elements.
246 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".
247 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.
248 The hub identifiers of link states must be in the set, $h_{ui}s$, of the road net's hub identifiers.

type
245 LΣ = H_UI-**set**
axiom

245 ∀ lσ:LΣ•**card** lσ=2
245 ∀ l:L • obs_LΣ(l) ∈ obs_LΩ(l)
type
246 LΩ = LΣ**-set**
247 L_Traffic
247 L_Traffic = (A_UI|B_UI) $\xrightarrow[m]{}$ ($\mathbb{T}$×(H_UI×Frac×H_UI))*
247 Frac = **Real**, **axiom** frac:Fract • 0<frac<1
value
245 attr_LΣ: L → LΣ
246 attr_LΩ: L → LΩ
247 attr_L_Traffic: : → L_Traffic
axiom
247 ∀ lt:L_Traffic,ui:(A_UI|B_UI)•ui ∈ **dom** ht ⇒ time_ordered(ht(ui))
248 ∀ l:L • l ∈ ls ⇒
248 **let** lσ = attr_LΣ(l) **in** ∀ ($h_{ui}i$,$h_{ui}i'$):(H_UI×K_UI) •
248 ($h_{ui}i$,$h_{ui}i'$) ∈ lσ ⇒ {h_{ui_i},h'_{ui_i}} ⊆ $h_{ui}s$ **end**

A.3.3.4 Bus Company Attributes

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.

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

type
249 BusTimTbl
value
249 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.

A.3.3.5 Bus Attributes

We show just a few attributes.

250 Buses run routes, according to their line number, ln:LN, in the
251 bus time table, btt:BusTimTbl obtained from their bus company, and and keep, as inert attributes, their segment of that time table.
252 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 hub*s, fh_ui, in the direction of the other *identified hub*, th_ui.

253 Et cetera.

type
250 LN
251 BusTimTbl

```
252   BPos   == atHub | onLink
252a  atHub    :: h_ui:H_UI
252b  onLink   :: fh_ui:H_UI×l_ui:L_UI×frac:Fract×th_ui:H_UI
252b  Fract    = Real, axiom frac:Fract • 0<frac<1
253   ...
value
251   attr_BusTimTbl: B → BusTimTbl
252   attr_BPos: B → BPos
```

A.3.3.6 Private Automobile Attributes

We illustrate but a few attributes:

254 Automobiles have static number plate registration numbers.
255 Automobiles have dynamic positions on the road net:

> [252a] either *at a hub* identified by some h_ui,
> [252b] 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
254   RegNo
255   APos   == atHub | onLink
252a  atHub    :: h_ui:H_UI
252b  onLink   :: fh_ui:H_UI × l_ui:L_UI × frac:Fract × th_ui:H_UI
252b  Fract    = Real, axiom frac:Fract • 0<frac<1
value
254   attr_RegNo: A → RegNo
255   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. Observe that bus companies each have their own distinct *bus time table*, and that these are modeled as *programmable*, Item 249 on the preceding page, page 140. Observe then that buses each have their own distinct *bus time table*, and that these are model-led as *inert*, Item 251 on the facing page, page 140. In Items 92 Pg. 76 and 96 Pg. 77, 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.[8]

---

[8] 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.3.3.7 Intentionality

256 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;
257 seen from the point of view of a hub there is its own traffic history, H_Traffic Item 92 Pg. 76, which is a (time ordered) sequence of timed maps from automobile identities into automobile positions; and
258 seen from the point of view of a link there is its own traffic history, L_Traffic Item 96 Pg. 77, which is a (time ordered) sequence of timed maps from automobile identities into automobile positions.

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

259 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
256   A_Hi  = (𝕋 × APos)*
243   H_Trf = A_UI $\underset{m}{\rightarrow}$ (𝕋𝕀𝕄𝔼 × APos)*
247   L_Trf = A_UI $\underset{m}{\rightarrow}$ (𝕋𝕀𝕄𝔼×APos)*
259   AllATH=𝕋𝕀𝕄𝔼 $\underset{m}{\rightarrow}$(AUI $\underset{m}{\rightarrow}$APos)
259   AllHTH=𝕋𝕀𝕄𝔼 $\underset{m}{\rightarrow}$(AUI $\underset{m}{\rightarrow}$APos)
259   AllLTH =𝕋𝕀𝕄𝔼 $\underset{m}{\rightarrow}$(AUI $\underset{m}{\rightarrow}$APos)
axiom
259   let allA=mrg_AllATH({(a,attr_A_Hi(a))|a:A•a ∈ as}),
259       allH=mrg_AllHTH({attr_H_Trf(h)|h:H•h ∈ hs}),
259       allL =mrg_AllLTH({attr_L_Trf(l)|l:L•h ∈ ls}) in
259   allA = mrg_HLT(allH,allL) end
```

We leave the definition of the four merge functions to the reader ! 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[9]. 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 !*.

Intentional Pull – General Transport: 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* ■

A.4 Perdurants

In this section we transcendentally "morph" **parts** into **behaviours.** We analyse that notion and its constituent notions of **actors**, **channels** and **communication**, **actions** and **events**.

The main transcendental deduction of this chapter is that of associating with each part a behaviour. This section shows the details of that association. Perdurants are understood in terms of a notion of *state* and a notion of *time*.

---

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

State Values versus State Variables: Item 206 on page 134 expresses the **value** of all parts of a road transport system:

206. ps:(UoB|H|L|BC|B|A)-**set** ≡ $rts \cup hls \cup bcs \cup bs \cup as$.

260 We now introduce the set of variables, one for each part value of the domain being modeled.

260. { **variable** vp:(UoB|H|L|BC|B|A) | vp:(UoB|H|L|BC|B|A) • $vp \in ps$ }

Buses and Bus Companies A bus company is like a "root" for its fleet of "sibling" buses. But a bus company may cease to exist without the buses therefore necessarily also ceasing to exist. They may continue to operate, probably illegally, without, possibly. a valid bus driving certificate. Or they may be passed on to either private owners or to other bus companies. We use this example as a reason for not endowing a "block structure" concept on behaviours.

A.4.1 Channels and Communication

A.4.1.1 Channel Message Types

We ascribe types to the messages offered on channels.

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

type
262 H_L_Msg, L_H_Msg
261 HL_Msg = H_L_Msg | L_F_Msg
263 BC_B_Msg = T × BusTimTbl
264 V_R_Msg = T × (BPos|APos)

A.4.1.2 Channel Declarations

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

channel
265 { hl_ch[h_ui,l_ui]:H_L_Msg | h_ui:H_UI,l_ui:L_UI•i ∈ $h_{ui}s$∧j ∈ $lh_{ui}m$(h_ui) }
265 ∪
265 { hl_ch[h_ui,l_ui]:L_H_Msg | 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.

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

channel
266 { bc_b_ch[bc_ui,b_ui] | bc_ui:BC_UI, b_ui:B_UI • 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.

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

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

A.4.2 Behaviours

A.4.2.1 Road Transport Behaviour Signatures

We first decide on names of behaviours. In the translation schemas we gave schematic names to behaviours of the form $\mathcal{M}_P$. We now assign mnemonic names: from part names to names of transcendentally interpreted behaviours and then we assign signatures to these behaviours.

A.4.2.1.1 Hub Behaviour Signature

268 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
268 hub$_{h_{ui}}$:
268a h_ui:H_UI×(vuis,luis,_):H_Mer×HΩ
268b → (HΣ×H_Traffic)
268c → **in**,**out** { h_l_ch[h_ui,l_ui] | l_ui:L_UI•l_ui ∈ luis }
268d { ba_r_ch[h_ui,v_ui] | v_ui:V_UI•v_ui∈vuis } **Unit**
268a **pre**: vuis = $v_{ui}s$ ∧ luis = $l_{ui}s$

A.4.2.1.2 Link Behaviour Signature

269 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
269   link$_{l_{ui}}$:
269a  l_ui:L_UI×(vuis,huis,_):L_Mer×LΩ
269b     → (LΣ×L_Traffic)
269c     → in,out { h_l_ch[ h_ui,l_ui ] | h_ui:H_UI:h_ui ∈ huis }
269d          { ba_r_ch[ l_ui,v_ui ] | v_ui:(B_UI|A_UI)•v_ui∈vuis } Unit
269a     pre: vuis = $v_{ui}s$ ∧ huis = $h_{ui}s$
```

A.4.2.1.3 Bus Company Behaviour Signature

270 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
270   bus_company$_{bc_{ui}}$:
270a  bc_ui:BC_UI×(_,_,buis):BC_Mer
270b     → BusTimTbl
270c       in,out {bc_b_ch[ bc_ui,b_ui ]|b_ui:B_UI•b_ui∈buis} Unit
270a     pre: buis = $b_{ui}s$ ∧ huis = $h_{ui}s$
```

A.4.2.1.4 Bus Behaviour Signature

271 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
271   bus$_{b_{ui}}$:
271a  b_ui:B_UI×(bc_ui,_,ruis):B_Mer
271b     → (LN × BTT × BPOS)
271c     → out bc_b_ch[ bc_ui,b_ui ],
271d          {ba_r_ch[ r_ui,b_ui ]|r_ui:(H_UI|L_UI)•ui∈$v_{ui}s$} Unit
271a     pre: ruis = $r_{ui}s$ ∧ bc_ui ∈ $bc_{ui}s$
```

A.4.2.1.5 Automobile Behaviour Signature

272 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
272   automobile$_{a_{ui}}$:
272a  a_ui:A_UI×(_,_,ruis):A_Mer×rn:RegNo
272b    → apos:APos
272c      in,out {ba_r_ch[a_ui,r_ui]|r_ui:(H_UI|L_UI)•r_ui∈ruis} Unit
272a    pre: ruis = $r_{ui}s$ ∧ a_ui ∈ $a_{ui}s$ ▪
```

A.4.2.2 Behaviour Definitions

We only illustrate automobile, hub and link behaviours.

A.4.2.2.1 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.

273 We abstract automobile behaviour at a Hub (hui).
274 The vehicle remains at that hub, "idling",
275 informing the hub behaviour,
276 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,

277 or, again internally non-deterministically,
278 the vehicle "disappears — off the radar" !

```
273   automobile$_{a_{ui}}$(a_ui,({},(ruis,vuis),{}),rn)
273           (apos:atH(fl_ui,h_ui,tl_ui)) ≡
274     (ba_r_ch[a_ui,h_ui] ! (record_TIME(),atH(fl_ui,h_ui,tl_ui));
275      automobile$_{a_{ui}}$(a_ui,({},(ruis,vuis),{}),rn)(apos))
276   ⊓
276a     (let ({fh_ui,th_ui},ruis′)=mereo_L(℘(tl_ui)) in
276a           assert: fh_ui=h_ui ∧ ruis=ruis′
273      let onl = (tl_ui,h_ui,0,th_ui) in
276b      (ba_r_ch[a_ui,h_ui] ! (record_TIME(),onL(onl)) ||
276b       ba_r_ch[a_ui,tl_ui] ! (record_TIME(),onL(onl))) ;
276c       automobile$_{a_{ui}}$(a_ui,({},(ruis,vuis),{}),rn)
276c            (onL(onl)) end end)
277   ⊓
278      stop
```

A.4.2.2.2 Automobile Behaviour On a Link

279 We abstract automobile behaviour on a Link.

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

```
279         automobile$_{a_{ui}}$(a_ui,({},ruis,{}),rno)
279                        (vp:onL(fh_ui,l_ui,f,th_ui)) ≡
279(a)ii    (ba_r_ch[thui,aui]!atH(lui,thui,nxt_lui) ;
279(a)i      automobile$_{a_{ui}}$(a_ui,({},ruis,{}),rno)(vp))
279b        ⊓
279(b)i     (if not_yet_at_hub(f)
279(b)i        then
279(b)i1         (let incr = increment(f) in
273               let onl = (tl_ui,h_ui,incr,th_ui) in
279(b)i2          ba−r_ch[l_ui,a_ui] ! onL(onl) ;
279(b)i3          automobile$_{a_{ui}}$(a_ui,({},ruis,{}),rno)
279(b)i3                        (onL(onl))
279(b)i           end end)
279(b)ii        else
279(b)ii1         (let nxt_lui:L_UI•nxt_lui ∈ mereo_H(℘(th_ui)) in
279(b)ii2          ba_r_ch[thui,aui]!atH(l_ui,th_ui,nxt_lui) ;
279(b)ii3          automobile$_{a_{ui}}$(a_ui,({},ruis,{}),rno)
279(b)ii3                        (atH(l_ui,th_ui,nxt_lui)) end)
279(b)i      end)
279c        ⊓
279d         stop
279(b)i1    increment: Fract → Fract
```

A.4.2.2.3 Hub Behaviour

280 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 92 Pg. 76.

```
value
280  hub$_{h_{ui}}$(h_ui,(,(luis,vuis)),hω)(hσ,ht) ≡
280a    □
280b      { let m = ba_r_ch[h_ui,v_ui] ? in
280c            assert: m=(_,atHub(_,h_ui,_))
280d        let ht′ = ht † [h_ui ↦ ⟨m⟩⌢ht(h_ui)] in
280e        hub$_{h_{ui}}$(h_ui,(,(luis,vuis)),(hω))(hσ,ht′)
280f      | v_ui:V_UI•v_ui∈vuis end end }
280g   post: ∀ v_ui:V_UI•v_ui ∈ dom ht′⇒time_ordered(ht′(v_ui))
```

A.4.2.2.4 Link Behaviour

281 The link behaviour non-deterministically, externally offers
282 to accept timed vehicle positions —
283 which will be on the link, from some vehicle, v_ui.
284 The timed vehicle link position is appended to the front of that vehicle's entry in the link's traffic table;
285 whereupon the link proceeds as a link behaviour with the updated link traffic table.
286 The link behaviour offers to accept from any vehicle.
287 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 96 Pg. 77.

```
281  link$_{l_{ui}}$(l_ui,(_,(huis,vuis),_),lω)(lσ,lt) ≡
281    □
282      { let m = ba_r_ch[l_ui,v_ui] ? in
283            assert: m=(_,onLink(_,l_ui,_,_))
284        let lt′ = lt † [l_ui ↦ ⟨m⟩⌢lt(l_ui)] in
285        link$_{l_{ui}}$(l_ui,(huis,vuis),hω)(hσ,lt′)
286      | v_ui:V_UI•v_ui∈vuis end end }
287    post: ∀ v_ui:V_UI•v_ui ∈ dom lt′⇒time_ordered(lt′(v_ui))
```

A.5 System Initialisation

A.5.1 Initial States

value
hs:H-**set** ≡ ≡ obs_sH(obs_SH(obs_RN(*rts*)))
ls:L-**set** ≡ ≡ obs_sL(obs_SL(obs_RN(*rts*)))
bcs:BC-**set** ≡ obs_BCs(obs_SBC(obs_FV(obs_RN(*rts*))))

bs:B-**set** ≡ ∪{obs_Bs(bc)|bc:BC•bc ∈ *bcs*}
as:A-**set** ≡ obs_BCs(obs_SBC(obs_FV(obs_RN(*rts*))))

A.5.2 Initialisation

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

288 The system to be initialised is

  a the parallel compositions (‖) of
  b the distributed parallel composition (‖{...|...}) of all hub behaviours,
  c the distributed parallel composition (‖{...|...}) of all link behaviours,
  d the distributed parallel composition (‖{...|...}) of all bus company behaviours,
  e the distributed parallel composition (‖{...|...}) of all bus behaviours, and
  f the distributed parallel composition (‖{...|...}) of all automobile behaviours.

```
value
288   initial_system: Unit → Unit
288   initial_system() ≡
288b      ‖ { hub$_{h_{ui}}$(h_ui,me,hω)(htrf,hσ)
288b        | h:H•h ∈ hs, h_ui:H_UI•h_ui=uid_H(h), me:HMetL•me=mereo_H(h),
288b          htrf:H_Traffic•htrf=attr_H_Traffic_H(h),
288b          hω:HΩ•hω=attr_HΩ(h), hσ:HΣ•hσ=attr_HΣ(h)∧hσ ∈ hω }
288a      ‖
288c      ‖ { link$_{l_{ui}}$(l_ui,me,lω)(ltrf,lσ)
288c          l:L•l ∈ ls, l_ui:L_UI•l_ui=uid_L(l), me:LMet•me=mereo_L(l),
288c          ltrf:L_Traffic•ltrf=attr_L_Traffic_H(l),
288c          lω:LΩ•lω=attr_LΩ(l), lσ:LΣ•lσ=attr_LΣ(l)∧lσ ∈ lω }
288a      ‖
288d      ‖ { bus_company$_{bc_{ui}}$(bcui,me)(btt)
288d          bc:BC•bc ∈ bcs, bc_ui:BC_UI•bc_ui=uid_BC(bc), me:BCMet•me=mereo_BC(bc),
288d          btt:BusTimTbl•btt=attr_BusTimTbl(bc) }
288a      ‖
288e      ‖ { bus$_{b_{ui}}$(b_ui,me)(ln,btt,bpos)
288e          b:B•b ∈ bs, b_ui:B_UI•b_ui=uid_B(b), me:BMet•me=mereo_B(b), ln:LN:pln=attr_LN(b),
288e          btt:BusTimTbl•btt=attr_BusTimTbl(b), bpos:BPos•bpos=attr_BPos(b) }
288a      ‖
288f      ‖ { automobile$_{a_{ui}}$(a_ui,me,rn)(apos)
288f          a:A•a ∈ as, a_ui:A_UI•a_ui=uid_A(a), me:AMet•me=mereo_A(a),
288f          rn:RegNo•rno=attr_RegNo(a), apos:APos•apos=attr_APos(a) }  ■
```

Appendix B
Pipelines

B.1 Endurants: External Qualities

We follow the ontology of Fig. 4.1 on page 39, the lefthand dashed box labelled *External Qualities*.

B.1.1 Parts

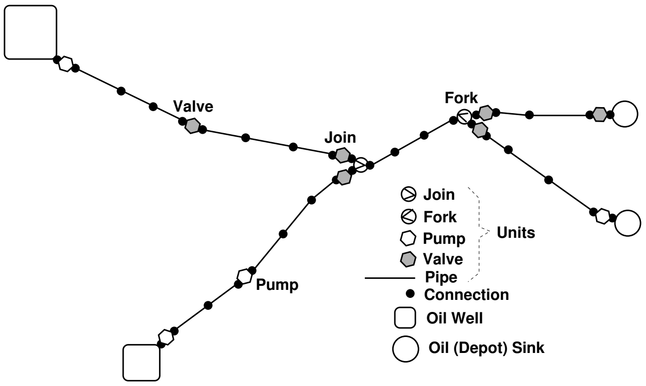


Fig. B.1 An example pipeline system

289 A pipeline system contains a set of pipeline units and a pipeline system monitor.
290 The well-formedness of a pipeline system depends on its mereology (cf. Sect. **B.2.2**) and the routing of its pipes (cf. Sect. **B.2.3.2**).
291 A pipeline unit is either a well, a pipe, a pump, a valve, a fork, a join, a plate[10], or a sink unit.
292 We consider all these units to be distinguishable, i.e., the set of wells, the set pipe, etc., the set of sinks, to be disjoint.

type

---

[10] A *plate* unit is a usually circular, flat steel plate used to "begin" or "end" a pipe segment.

```
289.  PLS′, U, M
290.  PLS = {| pls:PLS′•wf_PLS(pls) |}
value
290.  wf_PLS: PLS → Bool
290.  wf_PLS(pls) ≡
290.  wf_Mereology(pls)∧wf_Routes(pls)∧wf_Metrics(pls)[11]
289.  obs_Us: PLS → U-set
289.  obs_M: PLS → M
type
291.  U = We | Pi | Pu | Va | Fo | Jo | Pl | Si
292.  We :: Well
292.  Pi :: Pipe
292.  Pu :: Pump
292.  Va :: Valv
292.  Fo :: Fork
292.  Jo :: Join
292.  Pl :: Plate
292.  Si :: Sink
```

B.1.2 An Endurant State

293 For a given pipeline system
294 we exemplify an endurant state σ
295 composed of the given pipeline system and all its manifest units, i.e., without plates.

```
value
293.   pls:PLS
variable
294.   σ := collect_state(pls)
value
295.   collect_state: PLS
295.   collect_state(pls) ≡ {pls} ∪ obs_Us(pls) \ Pl
```

B.2 Endurants: Internal Qualities

We follow the ontology of Fig. 4.1 on page 39, the lefthand vertical and horisontal lines.

B.2.1 Unique Identification

296 The pipeline system, as such,
297 has a unique identifier, distinct (different) from its pipeline unit identifiers.
298 Each pipeline unit is uniquely distinguished by its unit identifier.

---

[11] wf_Mereology, wf_Routes and wf_Metrics will be explained in Sects. **B.2.2.2** on page 154, **B.2.3.2** on page 155, and **B.2.4.3** on page 159.

299 There is a state of all unique identifiers.

```
    type
297.    PLSI
298.    UI
    value
296.    pls:PLS
297.    uid_PLS: PLS → PLSI
298.    uid_U: U → UI
    variable
299.    $\sigma_{uid}$ := { uid_PLS(pls) } ∪ xtr_UIs(pls)
    axiom
298.    ∀ u,u′:U•{u,u′}⊆obs_Us(pls)⇒u≠u′⇒uid_UI(u)≠uid_UI(u′)
298.  ∧ uid_PLS(pls) ∉ {uid_UI(u)|u:U•u ∈ obs_Us(pls)}
```

300 From a pipeline system one can observe the set of all unique unit identifiers.

```
    value
300.  xtr_UIs: PLS → UI-set
300.  xtr_UIs(pls) ≡ {uid_UI(u)|u:U•u ∈ obs_Us(pls)}
```

301 We can prove that the number of unique unit identifiers of a pipeline system equals that of the units of that system.

```
    theorem:
301.    ∀ pls:PLS•card obs_Us(pl)=card xtr_UIs(pls)
```

B.2.2 Mereology

B.2.2.1 PLS Mereology

302 The mereology of a pipeline system is the set of unique identifiers of all the units of that system.

```
type
302.  PLS_Mer = UI-setiptyPLS_Merpls-mer-00
value
302.  mereo_PLS: PLS → PLS_Meripobmereo_PLSpls-mer-00
axiomiptyWellformed Mereologiespls-mer-00
302.  ∀ uis:PLS_Mer • uis = card xtr_UIs(pls)
```

B.2.2.2 Unit Mereologies

303 Each unit is connected to zero, one or two other existing input units and zero, one or two other existing output units as follows:

a A well unit is connected to exactly one output unit (and, hence, has no “input”).
b A pipe unit is connected to exactly one input unit and one output unit.

c A pump unit is connected to exactly one input unit and one output unit.
d A valve is connected to exactly one input unit and one output unit.
e A fork is connected to exactly one input unit and two distinct output units.
f A join is connected to exactly two distinct input units and one output unit.
g A plate is connected to exactly one unit.
h A sink is connected to exactly one input unit (and, hence, has no "output").

```
     type
303.     MER = UI-set × UI-set
     value
303.     mereo_U: U → MER
     axiom
303.      wf_Mereology: PLS → Bool
303.      wf_Mereology(pls) ≡
303.         ∀ u:U•u ∈ obs_Us(pls)⇒
303.            let (iuis,ouis) = mereo_U(u) in iuis ∪ ouis ⊆ xtr_UIs(pls) ∧
303.                case (u,(card uius,card ouis)) of
303a.                     (mk_We(we),(0,1)) → true,
303b.                     (mk_Pi(pi),(1,1)) → true,
303c.                     (mk_Pu(pu),(1,1)) → true,
303d.                     (mk_Va(va),(1,1)) → true,
303e.                     (mk_Fo(fo),(1,1)) → true,
303f.                    (mk_Jo(jo),(1,1)) → true,
303f.                    (mk_Pl(pl),(0,1)) → true, "begin"
303f.                    (mk_Pl(pl),(1,0)) → true, "end"
303h.                     (mk_Si(si),(1,1)) → true,
303.                     _ → false end end
```

B.2.3 Pipeline Concepts, I

B.2.3.1 Pipe Routes

304 A route (of a pipeline system) is a sequence of connected units (of the pipeline system).
305 A route descriptor is a sequence of unit identifiers and the connected units of a route (of a pipeline system).

```
     type
304.     R′ = U^ω
304.     R = {| r:Route′•wf_Route(r) |}
305.     RD = UI^ω
     axiom
305.     ∀ rd:RD • ∃ r:R•rd=descriptor(r)
     value
305.     descriptor: R → RD
305.     descriptor(r) ≡ ⟨uid_UI(r[i])|i:Nat•1≤i≤len r⟩
```

306 Two units are adjacent if the output unit identifiers of one shares a unique unit identifier with the input identifiers of the other.

```
   value
306.  adjacent: U × U → Bool
306.  adjacent(u,u′) ≡ let (,ouis)=mereo_U(u),(iuis,)=mereo_U(u′) in ouis ∩ iuis ≠ {} end
```

307 Given a pipeline system, *pls*, one can identify the (possibly infinite) set of (possibly infinite) routes of that pipeline system.

- a The empty sequence, ⟨⟩, is a route of *pls*.
- b Let u, u' be any units of *pls*, such that an output unit identifier of u is the same as an input unit identifier of u' then $\langle u, u' \rangle$ is a route of *pls*.
- c If r and r' are routes of *pls* such that the last element of r is the same as the first element of r', then r⌢**tl**r' is a route of *pls*.
- d No sequence of units is a route unless it follows from a finite (or an infinite) number of applications of the basis and induction clauses of Items 307a–307c.

```
   value
307.   Routes: PLS → RD-infset
307.   Routes(pls) ≡
307a.      let rs = ⟨⟩ ∪
307b.            {⟨uid_UI(u),uid_UI(u′)⟩|u,u′:U•{u,u′}⊆obs_Us(pls) ∧ adjacent(u,u′)}
307c.         ∪ {r⌢tl r′|r,r′:R•{r,r′}⊆rs}
307d.      in rs end
```

B.2.3.2 Well-formed Routes

308 A route is acyclic if no two route positions reveal the same unique unit identifier.

```
   value
308. is_acyclic_Route: R → Bool
308. is_acyclic_Route(r) ≡ ∼∃ i,j:Nat•{i,j}⊆inds r ∧ i≠j ∧ r[ i ]=r[ j ]
```

309 A pipeline system is well-formed if none of its routes are circular (and all of its routes embedded in well-to-sink routes).

```
   value
309. wf_Routes: PLS → Bool
309. wf_Routes(pls) ≡
309.    non_circular(pls) ∧ are_embedded_Routes(pls)

309. is_non_circular_PLS: PLS → Bool
309. is_non_circular_PLS(pls) ≡
309.    ∀ r:R•r ∈ routes(p)∧acyclic_Route(r)
```

310 We define well-formedness in terms of well-to-sink routes, i.e., routes which start with a well unit and end with a sink unit.

```
   value
310. well_to_sink_Routes: PLS → R-set
310. well_to_sink_Routes(pls) ≡
310.   let rs = Routes(pls) in
310.   {r|r:R•r ∈ rs ∧ is_We(r[ 1 ]) ∧ is_Si(r[ len r ])} end
```

311 A pipeline system is well-formed if all of its routes are embedded in well-to-sink routes.

```
311.  are_embedded_Routes: PLS → Bool
311.  are_embedded_Routes(pls) ≡
311.     let wsrs = well_to_sink_Routes(pls) in
311.     ∀ r:R • r ∈ Routes(pls) ⇒
311.        ∃ r′:R,i,j:Nat •
311.           r′ ∈ wsrs
311.        ∧ {i,j}⊆inds r′∧i≤j
311.        ∧ r = ⟨r′[k]|k:Nat•i≤k≤j⟩ end
```

B.2.3.3 Embedded Routes

312 For every route we can define the set of all its embedded routes.

```
      value
312.  embedded_Routes: R → R-set
312.  embedded_Routes(r) ≡ {⟨r[k]|k:Nat•i≤k≤j⟩ | i,j:Nat• i {i,j}⊆inds(r) ∧ i≤j}
```

B.2.3.4 A Theorem

313 The following theorem is conjectured:

- a the set of all routes (of the pipeline system)
- b is the set of all well-to-sink routes (of a pipeline system) and
- c all their embedded routes

```
      theorem:
313.  ∀ pls:PLS •
313.  let rs = Routes(pls),
313.     wsrs = well_to_sink_Routes(pls) in
313a.  rs =
313b.     wsrs ∪
313c.     ∪ {{r′|r′:R • r′ ∈ is_embedded_Routes(r″)} | r″:R • r″ ∈ wsrs}
312.  end
```

B.2.3.5 Fluids

314 The only fluid of concern to pipelines is the gas[12] or liquid[13] which the pipes transport[14].

```
      type
314.     GoL [ = M ]
      value
314.     obs_GoL: U → GoL
```

---

[12] Gaseous materials include: air, gas, etc.

[13] Liquid materials include water, oil, etc.

[14] The description of this document is relevant only to gas or oil pipelines.

B.2.4 Attributes

B.2.4.1 Unit Flow Attributes

315 A number of attribute types characterise units:

- a estimated current well capacity (barrels of oil, etc.),
- b pump height (a static attribute),
- c current pump status (not pumping, pumping; a programmable attribute),
- d current valve status (closed, open; a programmable attribute) and
- e flow (barrels/second, a biddable attribute).

```
    type
315a.   WellCap
315b.   Pump_Height
315c.   Pump_State == {|not_pumping,pumping|}
315d.   Valve_State == {|closed,open|}
315e.   Flow
```

316 Flows can be added and subtracted,
317 added distributively and
318 flows can be compared.

```
    value
316.   ⊕,⊖: Flow×Flow → Flow
317.   ⊕: Flow-set → Flow
318.   <,≤,=,≠,≥,>: Flow × Flow → Bool
```

319 Properties of pipeline units include

- a estimated current well capacity (barrels of oil, etc.) [a biddable attribute],
- b pipe length [a static attribute],
- c current pump height [a biddable attribute],
- d current valve open/close status [a programmable attribute],
- e current [𝓛aminar] in-flow at unit input [a monitorable attribute],
- f current 𝓛aminar] in-flow leak at unit input [a monitorable attribute],
- g maximum [𝓛aminar] guaranteed in-flow leak at unit input [a static attribute],
- h current [𝓛aminar] leak unit interior [a monitorable attribute],
- i current [𝓛aminar] flow in unit interior [a monitorable attribute],
- j maximum 𝓛aminar] guaranteed flow in unit interior [a monitorable attribute],
- k current [𝓛aminar] out-flow at unit output [a monitorable attribute],
- l current [𝓛aminar] out-flow leak at unit output [a monitorable attribute] and
- m maximum guaranteed 𝓛aminar out-flow leak at unit output [a static attribute.

```
type
319e  In_Flow = Flow
319f  In_Leak = Flow
319g  Max_In_Leak = Flow
319h  Body_Flow = Flow
319i  Body_Leak = Flow
319j  Max_Flow = Flow
319k  Out_Flow = Flow
319l  Out_Leak = Flow
319m  Max_Out_Leak = Flow
value
319a  attr_WellCap: We → WellCap
319b  attr_LEN: Pi → LEN
319c  attr_Height: Pu → Height
319d  attr_ValSta: Va → VaSta
319e  attr_In_Flow: U → UI → Flow
```

```
319f  attr_In_Leak: U → UI → Flow
319g  attr_Max_In_Leak: U → UI → Flow
319h  attr_Body_Flow: U → Flow
319i  attr_Body_Leak: U → Flow
319j  attr_Max_Flow: U → Flow
319k  attr_Out_Flow: U → UI → Flow
319l  attr_Out_Leak: U → UI → Flow
319m  attr_Max_Out_Leak: U → UI → Flow
```

320 Summarising we can define a two notions of flow:

  a static and
  b monitorable.

```
type
320a  Sta_Flows = Max_In_Leak×In_Max_Flow>Max_Out_Leak
320b  Mon_Flows = In_Flow×In_Leak×Body_Flow×Body_Leak×Out_Flow×Out_Leak
```

B.2.4.2 Unit Metrics

Pipelines are laid out in the terrain. Units have length and diameters. Units are positioned in space: have altitude, longitude and latitude positions of its one, two or three connection PoinTs[15].

321 length (a static attribute),
322 diameter (a static attribute) and
323 position (a static attribute).

```
type
321.  LEN
322.  ◯
323.  POS == mk_One(pt:PT) | mk_Two(ipt:PT,opt:PT)
323.          | mk_OneTwo(ipt:PT,opts:(lpt:PT,rpt:PT))
323.          | mk_TwoOne(ipts:(lpt:PT,rpt:PT),opt:PT)
323.  PT = Alt × Lon × Lat
323.  Alt, Lon, Lat = ...
value
321.  attr_LEN: U → LEN
322.  attr_◯: U → ◯
323.  attr_POS: U → POS
```

We can summarise the metric attributes:

324 Units are subject to either of four (mutually exclusive) metrics:

  a Length, diameter and a one point position.
  b Length, diameter and a two points position.
  c Length, diameter and a one+two points position.
  d Length, diameter and a two+one points position.

```
type
324.  Unit_Sta = Sta1_Metric | Sta2_Metric | Sta12_Metric | Sta21_Metric
324a  Sta1_Metric = LEN × Ø × mk_One(pt:PT)
324b  Sta2_Metric = LEN × Ø × mk_Two(ipt:PT,opt:PT)
324c  Sta12_Metric = LEN × Ø × mk_OneTwo(ipt:PT,opts:(lpt:PT,rpt:PT))
324d  Sta21_Metric = LEN × Ø × mk_TwpOne(ipts:(lpt:PT,rpt:PT),opt:PT)
```

---

[15] 1 for *wells*, *plates* and *sinks*; 2 for *pipes*, *pumps* and *valves*; 1+2 for *forks*, 2+1 for *joins*.

B.2.4.3 Wellformed Unit Metrics

The points positions of neighbouring units must “fit” one-another.

325 Without going into details we can define a predicate, wf_Metrics, that applies to a pipeline system and yields **true** iff neighbouring units must “fit” one-another.

```
value
325. wf_Metrics: PLS → Bool
325. wf_Metrics(pls) ≡ ...
```

B.2.4.4 Summary

We summarise the static, monitorable and programmable attributes for each manifest part of the pipeline system:

```
type
    PLS_Sta = PLS_net×...
    PLS_Mon = ...
    PLS_Prg = PLS_Σ×...
    Well_Sta = Sta1_Metric×Sta_Flows×Orig_Cap×...
    Well_Mon = Mon_Flows×Well_Cap×...
    Well_Prg = ...
    Pipe_Sta = Sta2_Metric×Sta_Flows×LEN×...
    Pipe_Mon = Mon_Flows×In_Temp×Out_Temp×...
    Pipe_Prg = ...
    Pump_Sta = Sta2_Metric×Sta_Flows×Pump_Height×...
    Pump_Mon = Mon_Flows×...
    Pump_Prg = Pump_State×...
    Valve_Sta = Sta2_Metric×Sta_Flows×...
    Valve_Mon = Mon_Flows×In_Temp×Out_Temp×...
    Valve_Prg = Valve_State×...
    Fork_Sta = Sta12_Metric×Sta_Flows×...
    Fork_Mon = Mon_Flows×In_Temp×Out_Temp×...
    Fork_Prg = ...
    Join_Sta = Sta21_Metric×Sta_Flows×...
    Join_Mon = Mon_Flows×In_Temp×Out_Temp×...
    Join_Prg = ...
    Sink_Sta = Sta1_Metric×Sta_Flows×Max_Vol×...
    Sink_Mon = Mon_Flows×Curr_Vol×In_Temp×Out_Temp×...
    Sink_Prg = ...
```

326 Corresponding to the above three attribute categories we can define “collective” attribute observers:

```
value
326. sta_A_We: We → Sta1_Metric×Sta_Flows×Orig_Cap×...
326. mon_A_We: We → ηMon_Flows×ηWell_Cap×ηIn_Temp×ηOut_Temp×...
326. prg_A_We: We → ...
326. sta_A_Pi: Pi → Sta2_Metric×Sta_Flows×LEN×...
326. mon_A_Pi: Pi → 𝒩Mon_Flows×ηIn_Temp×ηOut_Temp×...
326. prg_A_Pi: Pi → ...
```

326. sta_A_Pu: Pu → Sta2_Metric×Sta_Flows×LEN×...
326. mon_A_Pu: Pu → $\mathcal{N}$Mon_Flows×ηIn_Temp×ηOut_Temp×...
326. prg_A_Pu: Pu → Pump_State×...
326. sta_A_Va: Va → Sta2_Metric×Sta_Flows×LEN×...
326. mon_A_Va: Va → $\mathcal{N}$Mon_Flows×ηIn_Temp×ηOut_Temp×...
326. prg_A_Va: Va → Valve_State×...
326. sta_A_Fo: Fo → Sta12_Metric×Sta_Flows×...
326. mon_A_Fo: Fo → $\mathcal{N}$Mon_Flows×ηIn_Temp×ηOut_Temp×...
326. prg_A_Fo: Fo → ...
326. sta_A_Jo: Jo → Sta21_Metric×Sta_Flows×...
326. mon_A_Jo: Jo → Mon_Flows×ηIn_Temp×ηOut_Temp×...
326. prg_A_Jo: Jo → ...
326. sta_A_Si: Si → Sta1_Metric×Sta_Flows×Max_Vol×...
326. mon_A_Si: Si → $\mathcal{N}$Mon_Flows×ηIn_Temp×ηOut_Temp×...
326. prg_A_Si: Si → ...

326. $\mathcal{N}$Mon_Flows ≡ (ηIn_Flow,ηIn_Leak,ηBody_Flow,ηBody_Leak,ηOut_Flow,ηOut_Leak)

Monitored flow attributes are [to be] passed as arguments to behaviours *by reference* so that their monitorable attribute values can be sampled.

B.2.4.5 Fluid Attributes

Fluids, we here assume, oil, as it appears in the pipeline units have no unique identity, have not mereology, but does have attributes: hydrocarbons consisting predominantly of aliphatic, alicyclic and aromatic hydrocarbons. It may also contain small amounts of nitrogen, oxygen, and sulfur compounds

327 We shall simplify, just for illustration, crude oil fluid of units to have these attributes:

  a volume,
  b viscosity,
  c temperature,
  d paraffin content (%age),
  e naphtenes content (%age),

type	**value**
327. Oil	327b. obs_Oil: U → Oil
327a. Vol	327a. attr_Vol: Oil → Vol
327b. Visc	327b. attr_Visc: Oil → Visc
327c. Temp	327c. attr_Temp: Oil → Temp
327d. Paraffin	327d. attr_Paraffin: Oil → Paraffin
327e. Naphtene	327e. attr_Naphtene: Oil → Naphtene

B.2.4.6 Pipeline System Attributes

The “root” pipeline system is a compound. In its transcendentally deduced behavioral form it is, amongst other “tasks”, entrusted with the monitoring and control of all its units. To do so it must, as a basically static attribute possess awareness, say in the form of a net diagram of how these units are interconnected, together with all their internal qualities, by type and by value. Next we shall give a very simplified account of the possible pipeline system attribute.

328 We shall make use, in this example, of just a simple pipeline state, pls_ω.

The pipeline state, pls_ω, embodies all the information that is relevant to the monitoring and control of an entire pipeline system, whether static or dynamic.

```
type
328. PLS_$\Omega$
```

B.2.5 Pipeline Concepts, II: Flow Laws

329 "What flows in, flows out !". For $\mathcal{L}$aminar flows: for any non-well and non-sink unit the sums of input leaks and in-flows equals the sums of unit and output leaks and out-flows.

```
   Law:
329.     ∀ u:U\We\Si •
329.         sum_in_leaks(u) ⊕ sum_in_flows(u) =
329.         attr_body_Leak$_{\mathcal{L}}$(u) ⊕
329.         sum_out_leaks(u) ⊕ sum_out_flows(u)
```

```
value
    sum_in_leaks: U → Flow
    sum_in_leaks(u) ≡ let (iuis,) = mereo_U(u) in ⊕ {attr_In_Leak$_{\mathcal{L}}$(u)(ui)|ui:UI•ui ∈ iuis} end
    sum_in_flows: U → Flow
    sum_in_flows(u) ≡ let (iuis,) = mereo_U(u) in ⊕ {attr_In_Flow$_{\mathcal{L}}$(u)(ui)|ui:UI•ui ∈ iuis} end
    sum_out_leaks: U → Flow
    sum_out_leaks(u) ≡ let (,ouis) = mereo_U(u) in ⊕ {attr_Out_Leak$_{\mathcal{L}}$(u)(ui)|ui:UI•ui ∈ ouis} end
    sum_out_flows: U → Flow
    sum_out_flows(u) ≡ let (,ouis) = mereo_U(u) in ⊕ {attr_Out_Leak$_{\mathcal{L}}$(u)(ui)|ui:UI•ui ∈ ouis} end
```

330 "What flows out, flows in !". For $\mathcal{L}$aminar flows: for any adjacent pairs of units the output flow at one unit connection equals the sum of adjacent unit leak and in-flow at that connection.

```
   Law:
330. ∀ u,u′:U•adjacent(u,u′) ⇒
330.     let (,ouis)=mereo_U(u), (iuis′,)=mereo_U(u′) in
330.     assert: uid_U(u′) ∈ ouis ∧ uid_U(u) ∈ iuis ′
330.     attr_Out_Flow$_{\mathcal{L}}$(u)(uid_U(u′)) =
330.     attr_In_Leak$_{\mathcal{L}}$(u)(uid_U(u))⊕attr_In_Flow$_{\mathcal{L}}$(u′)(uid_U(u)) end
```

These "laws" should hold for a pipeline system without plates.

B.3 Perdurants

We follow the ontology of Fig. 4.1 on page 39, the right-hand dashed box labeled *Perdurants* and the right-hand vertical and horisontal lines.

B.3.1 State

We introduce concepts of *manifest* and *structure* endurants. The former are such compound endurants (Cartesians of sets) to which we ascribe internal qualities; the latter are such compound endurants (Cartesians of sets) to which we **do not** ascribe internal qualities. The distinction is pragmatic.

331 For any given pipeline system we suggest the state to consist of the manifest endurants of all its non-plate units.

value
331. σ = obs_Us(pls)

B.3.2 Channel

332 There is a [global] array channel indexed by a "set pair" of distinct manifest endurant part identifiers – signifying the possibility of the syncharonisation and communication between any pair of pipeline units and between these and the pipeline system, cf. last, i.e., bottom-most diagram of Fig. **B**.11 on page 170.

channel
332. { ch[{i,j}] | {i,j}:(PLSI|UI) • {i,j}$\subseteq\sigma_{id}$ }

B.3.3 Actions

These are, informally, some of the actions of a pipeline system:

333 **start pumping**: from a state of not pumping to a state of pumping "at full blast!".[16]
334 **stop pumping**: from a state of (full) pumping to a state of no pumping at all.
335 **open valve**: from a state of a fully closed valve to a state of fully open valve.[17]
336 **close valve**: from a state of a fully opened valve to a state of fully closed valve.

We shall not define these actions in this paper. But they will be referred to in the *pipeline_system* (Items 355a, 355b, 355c), the *pump* (Items 358a, 358b) and the *valve* (Items 361a, 361b) behaviours.

B.3.4 Behaviours

B.3.4.1 Behaviour Kinds

There are eight kinds of behaviours:

---

[16] – that is, we simplify, just for the sake of illustration, and do not consider "intermediate" states of pumping.
[17] – cf. Footnote 16.

337 the pipeline system behaviour;[18]
338 the [generic] well behaviour,
339 the [generic] pipe behaviour,
340 the [generic] pump behaviour,
341 the [generic] valve behaviour,
342 the [generic] fork behaviour,
343 the [generic] join behaviour,
344 the [generic] sink behaviour.

B.3.4.2 Behaviour Signatures

345 The *pipeline_system* behaviour, *pls*,
346 The *well* behaviour signature lists the unique well identifier, the well mereology, the static well attributes, the monitorable well attributes, the programmable well attributes and the channels over which the well [may] interact with the pipeline system and a pipeline unit.
347 The *pipe* behaviour signature lists the unique pipe identifier, the pipe mereology, the static pipe attributes, the monitorable pipe attributes, the programmable pipe attributes and the channels over which the pipe [may] interact with the pipeline system and its two neighbouring pipeline units.
348 The *pump* behaviour signature lists the unique pump identifier, the pump mereology, the static pump attributes, the monitorable pump attributes, the programmable pump attributes and the channels over which the pump [may] interact with the pipeline system and its two neighbouring pipeline units.
349 The *valve* behaviour signature lists the unique valve identifier, the valve mereology, the static valve attributes, the monitorable valve attributes, the programmable valve attributes and the channels over which the valve [may] interact with the pipeline system and its two neighbouring pipeline units.
350 The *fork* behaviour signature lists the unique fork identifier, the fork mereology, the static fork attributes, the monitorable fork attributes, the programmable fork attributes and the channels over which the fork [may] interact with the pipeline system and its three neighbouring pipeline units.
351 The *join* behaviour signature lists the unique join identifier, the join mereology, the static join attributes, the monitorable join attributes, the programmable join attributes and the channels over which the join [may] interact with the pipeline system and its three neighbouring pipeline units.
352 The *sink* behaviour signature lists the unique sink identifier, the sink mereology, the static sing attributes, the monitorable sing attributes, the programmable sink attributes and the channels over which the sink [may] interact with the pipeline system and its one or more pipeline units.

value
345. pls: plso:PLSI → pls_mer:PLS_Mer → PLS_Sta → PLS_Mon →
345. PLS_Prg → { ch[{plsi,ui}] | ui:UI • ui ∈ σ_{ui} } **Unit**
346. well: wid:WI → well_mer:MER → Well_Sta → Well_mon →
346. Well_Prgr → { ch[{plsi,ui}] | wi:WI • ui ∈ σ_{ui} } **Unit**
347. πipe: UI → pipe_mer:MER → Pipe_Sta → Pipe_mon →
347. Pipe_Prgr → { ch[{plsi,ui}] | ui:UI • ui ∈ σ_{ui} } **Unit**
348. pump: pi:UI → pump_mer:MER → Pump_Sta → Pump_Mon →
348. Pump_Prgr → { ch[{plsi,ui}] | ui:UI • ui ∈ σ_{ui} } **Unit**
349. valve: vi:UI → valve_mer:MER → Valve_Sta → Valve_Mon →
349. Valve_Prgr → { ch[{plsi,ui}] | ui:UI • ui ∈ σ_{ui} } **Unit**
350. fork: fi:FI → fork_mer:MER → Fork_Sta → Fork_Mon →

---

[18] This “PLS” behaviour summarises the either global, i.e., *SCADA*[19]-like behaviour, or the fully distributed, for example, manual, human-operated behaviour of the monitoring and control of the entire pipeline system.

[19] Supervisory Control And Data Acquisition

```
350.                        Fork_Prgr → { ch[{plsi,ui}] | ui:UI • ui ∈ $\sigma_{ui}$ } Unit
351. join: ji:JI → join_mer:MER → Join_Sta → Join_Mon →
351.                       Join_Prgr → { ch[{plsi,ui}] | ui:UI • ui ∈ $\sigma_{ui}$ } Unit
352. sink: si:SI → sink_mer:MER → Sink_Sta → Sink_Mon →
352.                       Sink_Prgr → { ch[{plsi,ui}] | ui:UI • ui ∈ $\sigma_{ui}$ } Unit
```

B.3.4.2.1 Behaviour Definitions

We show the definition of only three behaviours:

- the **pipe_line_system** behaviour,
- the **pump** behaviour and
- the **valve** behaviour.

B.3.4.2.2 The Pipeline System Behaviour

353 The pipeline system behaviour
354 calculates, based on its programmable state, its next move;
355 if that move is [to be] an action on a named
  a pump, whether to start or stop pumping, then the named pump is so informed, whereupon the pipeline system behaviour resumes in the new pipeline state; or
  b valve, whether to open or close the valve, then the named valve is so informed, whereupon the pipeline system behaviour resumes in the new pipeline state; or
  c unit, to collect its monitorable attribute values for monitoring, whereupon the pipeline system behaviour resumes in the further updated pipeline state;
  d et cetera;

```
value
353.  pls(plsi)(uis)(pls_msta)(pls_mon)(pls_$\omega$) ≡
354.    let (to_do,pls_$\omega'$) = calculate_next_move(plsi,pls_mer,pls_msta,pls_mon,pls_prgr) in
355.    case to_do of
355a       mk_Pump(pi,$\alpha$) →
355a           ch[{plsi,pi}] ! $\alpha$ assert: $\alpha$ ∈ {stop_pumping,pump};
355a           pls(plsi)(pls_mer)(pls_msta)(pls_mon)(pls_$\omega'$),
355b       mk_Valve(vi,$\alpha$) →
355b           ch[{plsi,vi}] ! $\alpha$ assert: $\alpha$ ∈ {open_valve,close_valve};
355b           pls(plsi)(pls_mer)(pls_msta)(pls_mon)(pls_$\omega'$),
355c       mk_Unit(ui,monitor) →
355c           ch[{plsi,ui}] ! monitor;
355c           pls(plsi)(pls_mer)(pls_msta)(pls_mon)(update_pls_$\omega$(ch[{plsi,ui}] ?,ui)(pls_$\omega'$)),
355d       ... end
353     end
```

We leave it to the reader to define the calculate_next_move function !

B.3.4.2.3 The Pump Behaviours

356 The [generic] pump behaviour internal non-deterministically alternates between
357 doing own work (...), or

358 accepting pump directives from the pipeline behaviour.

  a If the directive is either to start or stop pumping, then that is what happens – whereupon the pump behaviour resumes in the new pumping state.
  b If the directive requests the values of all monitorable attributes, then these are *gathered*, communicated to the pipeline system behaviour – whereupon the pump behaviour resumes in the "old" state.

```
value
356.  pump(π)(pump_mer)(pump_sta)(pump_mon)(pump_prgr) ≡
357.    ...
358.   ⊓ let α = ch[{plsi,π}] ? in
358.     case α of
358a.        stop_pumping ∨ pump
358a.           → pump(π)(pump_mer)(pump_sta)(pump_mon)(α)[20]end,
358b.        monitor
358b.           → let mvs = gather_monitorable_values(π,pump_mon) in
358b.             ch[{plsi,π}] ! mvs;
358b.             pump(π)(pump_mer)(pump_sta)(pump_mon)(pump_prgr) end
358.     end
```

We leave it to the reader to defined the gather_monitorable_values function.

B.3.4.2.4 The Valve Behaviours

359 The [generic] valve behaviour internal non-deterministically alternates between
360 doing own work (...), or
361 accepting valve directives from the pipeline system.

  a If the directive is either to open or close the valve, then that is what happens – whereupon the pump behaviour resumes in the new valve state.
  b If the directive requests the values of all monitorable attributes, then these are *gathered*, communicated to the pipeline system behaviour – whereupon the valve behaviour resumes in the "old" state.

```
value
359.  valve(vi)(valv_mer)(valv_sta)(valv_mon)(valv_prgr) ≡
360.    ...
361.   ⊓ let α = ch[{plsi,π}] ? in
361.     case α of
361a.        open_valve ∨ close_valve
361a.           → valve(vi)(val_mer)(val_sta)(val_mon)(α)[21]end,
361b.        monitor
361b.           → let mvs = gather_monitorable_values(vi,val_mon) in
361b.             ch[{plsi,π}] ! (vi,mvs);
361b.             valve(vi)(val_mer)(val_sta)(val_mon)(val_prgr) end
361.     end
```

---

[20] Updating the programmable pump state to either **stop_pumping** or **pump** shall here be understood to mean that the pump is set to not pump, respectively to pump.

[21] Updating the programmable valve state to either **open_valve** or **close_valve** shall here be understood to mean that the valve is set to open, respectively to closed position.

B.3.4.3 Sampling Monitorable Attribute Values

Static and programmable attributes are, as we have seen, *passed by value* to behaviours. Monitorable attributes "surreptitiously" change their values so, as a technical point, these are *passed by reference* – by *passing attribute type names*.

362 From the name, ηA, of a monitorable attribute and the unique identifier, u_i, of the part having the named monitorable attribute one can then, "dynamically", "on-the-fly", as the part behaviour "moves-on", retrieve the value of the monitorable attribute. This can be illustrated as follows:
363 The unique identifier u_i is used in order to retrieve, from the global parts state, σ, that identified part, p.
364 Then attr_A is applied to p.

```
value
362. retr_U: UI → Σ → U
362. retr_U(ui)(σ) ≡ let u:U • u ∈ σ∧uid_U(u)=ui in u end
363. retr_AttrVal: UI × ηA → Σ → A
364. retr_AttrVal(ui)(ηA)(σ) ≡ attr_A(retr_U(ui)(σ))
```

retr_AttrVal(...)(...)(...) can now be applied in the body of the behaviour definitions, for example in gather_monitorable_values.

B.3.4.4 System Initialisation

System initialisation means to "morph" all manifest parts into their respective behaviours, initialising them with their respective attribute values.

365 The *pipeline system* behaviour is initialised and "put" in parallel with the parallel compositions of
366 all initialised *well*,
367 all initialised *pipe*,
368 all initialised *pump*,
369 all initialised *valve*,
370 all initialised *fork*,
371 all initialised *join* and
372 all initialised *sink* behaviours.[22]

```
value
365. pls(uid_PLS(pls))(mereo_PLS(pls))((pls))((pls))((pls))
366. || || { well(uid_U(we))(mereo_U(we))(sta_A_We(we))(mon_A_We(we))(prg_A_We(we)) | we:Well • w ∈ σ }
367. || || { pipe(uid_U(pi))(mereo_U(pi))(sta_A_Pi(pi))(mon_A_Pi(pi))(prg_A_Pi(pi)) | pi:Pi • pi ∈ σ }
368. || || { pump(uid_U(pu))(mereo_U(pu))(sta_A_Pu(pu))(mon_A_Pu(pu))(prg_A_Pu(pu)) | pu:Pump • pu ∈ σ }
369. || || { valv(uid_U(va))(mereo_U(va))(sta_A_Va(va))(mon_A_Va(va))(prg_A_Va(va)) | va:Well • va ∈ σ }
370. || || { fork(uid_U(fo))(mereo_U(fo))(sta_A_Fo(fo))(mon_A_Fo(fo))(prg_A_Fo(fo)) | fo:Fork • fo ∈ σ }
371. || || { join(uid_U(jo))(mereo_U(jo))(sta_A_Jo(jo))(mon_A_J(jo))(prg_A_J(jo)) | jo:Join • jo ∈ σ }
372. || || { sink(uid_U(si))(mereo_U(si))(sta_A_Si(si))(mon_A_Si(si))(prg_A_Si(si)) | si:Sink • si ∈ σ }
```

The sta_..., mon_..., and prg_A... functions are defined in Items 326 on page 159.
Note: || { f(u)(...) | u:U • u ∈ {} } ≡ **()**.

B.4 Index

---

[22] Plates are treated as are structures, i.e., not "behaviourised"!

B.5 Illustrations of Pipeline Phenomena

Fig. B.2 **The Planned Nabucco Pipeline:** http://en.wikipedia.org/wiki/Nabucco_Pipeline

Fig. B.3 **Pipeline Construction**

Fig. B.4 **Pipe Segments**

Fig. B.5 **Valves**

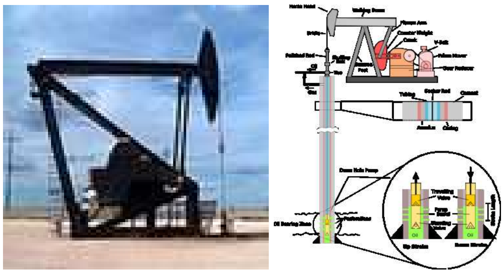


Fig. B.6 **Oil Pumps**

Fig. B.7 **Gas Compressors**

Fig. **B**.8 New and Old Pigs

Fig. **B**.9 Pig Launcher, Receiver

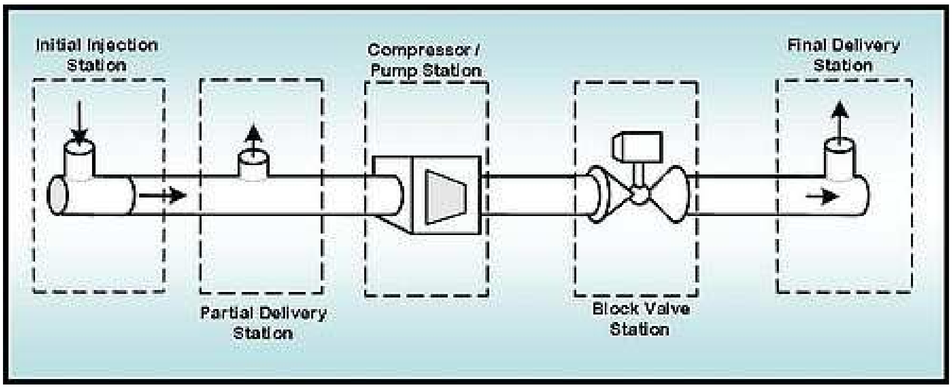


Fig. **B**.10 Leftmost: A **Well**. 2nd from left: a **Fork**. Rightmost: a **Sink**

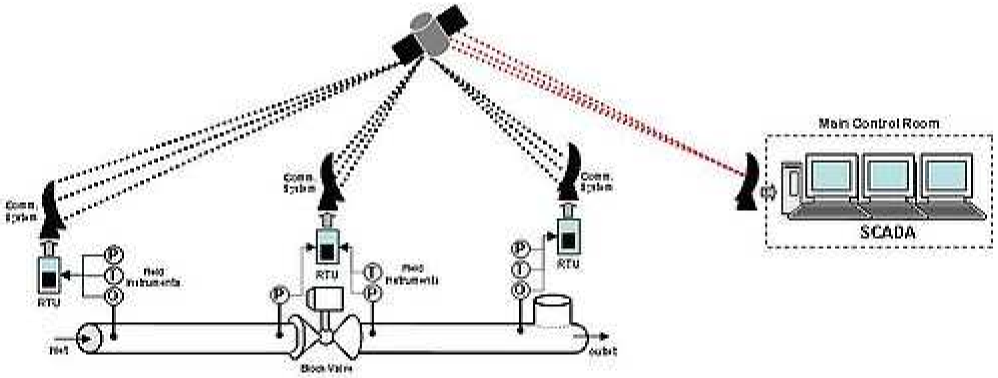


Fig. **B**.11 A SCADA [Supervisory Control And Data Acquisition] Diagram

Appendix C
A Raise Specification Language Primer

Contents

$\mathcal{I}$:[23] *We present an* `RSL Primer`*. Indented text, in slanted font, such as this, presents informal material and examples. Non-indented text, in* roman font*, presents narrative and formal explanation of RSL constructs.*

This `RSL Primer` *omits treatment of a number of language constructs, notably the RSL module concepts of schemes, classes and objects. Although we do cover the imperative language construct of [declaration of] variables and, hence, assignment, we shall omit treatment of structured imperative constructs like* **for** *..., **do** s **while** b, **while** b **do** s loops.*

Section **C.12** *on page 196 introduces additional language constructs, thereby motivation the* $^+$ *in the RSL*$^+$ *name*[24] ▪

---

[23] The letter $\mathcal{I}$ shall designate begin of informal text.

[24] The ▪ symbol shall designate end-of-informal text.

C.1 Types and Values

𝓘: Types are, in general, set-like structures[25] *of things, i.e., values, having common characteristics.*

A bunch of zero, one or more apples (type apples) may thus form a [sub]set of type Belle de Boskoop apples. A bunch of zero, one or more pears (type pears) may thus form a [sub]set of type Concorde pears. A union of zero, one or more of these apples and pears then form a [sub]set of entities of type fruits. ▪

C.1.1 Sort and Type Expressions

Sort and type expressions are expressions whose values are types, that is, possibly infinite set-like structures of values (of "that" type).

C.1.1.1 Atomic Types: Identifier Expressions and Type Values

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 [so-called] built-in atomic types. They are expressed in terms of literal identifiers. These are the **Bool**eans, **int**egers, **Nat**ural numbers, **Real**s, **Char**acters, and **Text**s. **Text**s are free-form texts and are more general than just texts of `RSL-like` formulas. `RSL`-**Text**'s will be introduced in Sect. **C.12** on page 196.

We shall not need the base types **Char**acters, nor the general type **Text**s for domain modelling in this primer. They will be listed below, but not mentioned further.

The base types are:

Basic Types

type
[1] **Bool**
[2] **Int**
[3] **Nat**
[4] **Real**
[5] **Char**
[6] **Text**

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″, ″bbb″, ...
6 The text type of character string values ″aa″, ″aaa″, ..., ″abc″, ...

---

[25] We shall not, in this primer, go into details as to the mathematics of types.

C.1.1.2 Composite Types: Expressions and Type Values

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 these are the composite types, hence, type expressions:

Composite Type Expressions

[7] A-**set**
[8] A-**infset**
[9] A × B × ... × C
[10] A^*
[11] A^ω
[12] $A \underset{m}{\rightarrow} B$
[13] $A \rightarrow B$
[14] $A \overset{\sim}{\rightarrow} B$
[15] A | B | ... | C
[16] mk_id(sel_a:A,...,sel_b:B)
[17] sel_a:A ... sel_b:B

The following are generic type expressions:

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 The postulated disjoint union of types A, B, . . . , and C.
16 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.
17 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.

Section **C.12** on page 196 introduces the extended RSL concepts of type name values and the type, $\mathbb{T}$, of type names.

C.1.2 Type Definitions

C.1.2.1 Sorts — Abstract Types

Types can be (abstract) sorts in which case their structure is not specified:

Sorts

```
type
   A, B, ..., C
```

C.1.2.2 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
```

RSL Example: **Sets.** **Narrative:** *H* stand for the domain type of street intersections – we shall call then hubs, and let *L* stand for the domain type of segments of streets between immediately neighboring hubs – we shall call then links. Then *Hs* and *Ls* are to designate the types of finite sets of zero, one or more hubs, respectively links. **Formalisation:**

type H, L, Hs=H-**set**, Ls=L-**set** •

RSL Example: **Cartesians.** **Narrative:** Let *RN* stand for the domain type of road nets consisting of hub aggregates, *HA*, and link aggregates, *LA*. Hub and link aggregates can be observed from road nets, and hub sets and link sets can be observed from hub, respectively link aggregates. **Formalisation:**

type RN = HA×LA, Hs, Ls
value obs_HA: RN→HA, obs_LA: RN– LA, obs_Hs: HA→Hs, obs_Ls: LA→Ls

Observer functions, obs_... are not further defined – beyond their signatures. They will (subsequently) be defined through axioms over their results •
Some schematic type definitions are:

Variety of Type Definitions

```
[18] Type_name = Type_expr /* without |s or subtypes */
[19] Type_name = Type_expr_1 | Type_expr_2 | ... | Type_expr_n
[20] 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)
[21] Type_name :: sel_a:Type_name_a ... sel_z:Type_name_z
[22] Type_name = {| v:Type_name′ • 𝒫(v) |}
```

where a form of [19–20] is provided by combining the types:

Record Types

```
[23] Type_name = A | B | ... | Z
[24] A == mk_id_1(s_a1:A_1,...,s_ai:A_i)
[25] B == mk_id_2(s_b1:B_1,...,s_bj:B_j)
[26] ...
[27] Z == mk_id_n(s_z1:Z_1,...,s_zk:Z_k)
```

Of these we shall almost exclusively make use of [23–27].

Disjoint Types. Narrative: A pipeline consists of a finite set of zero, one or more [interconnected][26] pipe units. Pipe units are either wells, or are pumps, or are valves, or are joins, or are forks, or are sinks. **Formalisation:**

```
type PL = P-set, P == WU|PU|VA|JO|FO|SI, Wu,Pu,Vu,Ju,Fu,Su
    WU::mkWU(swu:Wu), PU::mkPU(spu:Pu), VA::mkVU(svu:Vu),
    JO::mkJu(sju:Ju), FO::mkFu(sfu:Fu), SI::mkSi(ssu:Su)
```

where we leave types Wu, Pu, Vu, Ju, Fu and Su further undefined •
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.

C.1.2.3 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 $\mathcal{P}$, constitute the subtype A:

Subtypes

```
type
  A = {| b:B • 𝒫(b) |}
```

Subtype. Narrative: The subtype of even natural numbers.
Formalisation: **type** ENat = {| en | en:**Nat** • is_even_natural_number(en) |} •

C.2 The Propositional and Predicate Calculi

C.2.1 Propositions

$\mathcal{I}$: In logic, a proposition is the meaning of a declarative sentence. [A declarative sentence is a type of sentence that makes a statement] ▪

C.2.1.1 Propositional Expressions

*$\mathcal{I}$: Propositional expressions, informally speaking, are quantifier-free expressions having truth (or **chaos**) values. ∀, ∃ and ∃ ! are quantifiers, see below.*

Below, we will first treat propositional expressions all of whose identifiers denote truth values. As we progress, in sections on arithmetic, sets, list, maps, etc., we shall extend the range of propositional expressions ▪

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values (**true** or **false** [or **chaos**]). Then:

Propositional Expressions

false, **true**
a, b, ..., c ∼a, a∧b, a∨b, a⇒b, a=b, a≠b

are propositional expressions having Boolean values. ∼, ∧, ∨, ⇒, =, ≠ and □ are Boolean connectives (i.e., operators). They can be read as: **not**, **and**, **or**, **if then** (or **implies**), **equal**, **not equal** and **always**.

C.2.1.2 Propositional Calculus

$\mathcal{I}$: Propositional calculus is a branch of logic. It is also called propositional logic, statement logic, sentential calculus, sentential logic, or sometimes zeroth-order logic. It deals with propositions (which can be true or false) and relations between propositions, including the construction of arguments based on them. Compound propositions are formed by connecting propositions by logical connectives. Propositions that contain no logical connectives are called atomic propositions `[Wikipedia]` ▪

A simple two-value Boolean logic can be defined as follows:

```
type
   Bool
value
   true, false
   ∼: Bool → Bool
   ∧, ∨, ⇒, =, ≠, ≡: Bool × Bool → Bool
axiom
   ∀ b,b′:Bool •
      ∼b ≡ if b then false else true end
      b ∧ b′ ≡ if b then b′ else false end
      b ∨ b′ ≡ if b then true else b′ end
      b ⇒ b′ ≡ if b then b′ else true end
```

$b = b' \equiv$ **if** $(b \wedge b') \vee (\sim b \wedge \sim b')$ **then true else false end**
$(b \neq b') \equiv \sim(b = b')$
$(b \equiv b') \equiv (b = b')$

We shall, however, make use of a three-value Boolean logic. The model-theory explanation of the meaning of propositional expressions is now given in terms of the *truth tables* for the logic connectives:

∨, ∧, and ⇒ Syntactic Truth Tables

∨	true	false	chaos
true	true	true	true
false	true	false	chaos
chaos	chaos	chaos	chaos

∧	true	false	chaos
true	true	false	chaos
false	false	false	false
chaos	chaos	chaos	chaos

⇒	true	false	chaos
true	true	false	chaos
false	true	true	true
chaos	chaos	chaos	chaos

The two-value logic defined earlier 'transpires' from the **true,false** columns and rows of the above truth tables.

C.2.2 Predicates

$\mathcal{I}$: Predicates are mathematical assertions that contains variables, sometimes referred to as predicate variables, and may be true or false depending on those variables' value or values[27] ▪

C.2.2.1 Predicate Expressions

Let x, y, ..., z (or term expressions) designate non-Boolean values, and let $\mathcal{P}(x)$, $\mathcal{Q}(y)$ and $\mathcal{R}(z)$ be propositional or predicate expressions, then:

Simple Predicate Expressions

[28] $\forall x{:}X \bullet \mathcal{P}(x)$
[29] $\exists y{:}Y \bullet \mathcal{Q}(y)$
[30] $\exists ! z{:}Z \bullet \mathcal{R}(z)$

are quantified, i.e., predicate expressions. ∀, ∃ and ∃! are the quantifiers.

C.2.2.2 Predicate Calculus

They are "read" as:

[28] For all x (values in type X) the predicate $\mathcal{P}(x)$ holds – if that is not the case the expression yields truth value **false**.

[29] There exists (at least) one y (value in type Y) such that the predicate $\mathcal{Q}(y)$ holds – if that is not the case the expression yields truth value **false**.

[30] There exists a unique z (value in type Z) such that the predicate $\mathcal{R}(z)$ holds – if that is not the case the expression yields truth value **false**.

---

[27] https://calcworkshop.com/logic/predicate-logic/, and: predicate logic, first-order logic or quantified logic is a formal language in which propositions are expressed in terms of predicates, variables and quantifiers. It is different from propositional logic which lacks quantifiers https://brilliant.org/wiki/predicate-logic/.

[28–30] The predicates $\mathcal{P}(x)$, $\mathcal{Q}(y)$ or $\mathcal{R}(z)$ may yield **chaos** in which case the whole expression yields **chaos**.

C.3 Arithmetics

$\mathcal{I}$: RSL offers the usual set of arithmetic operators. From these the usual kind of arithmetic expressions can be formed. ▪

Arithmetic

```
type
   Nat, Int, Real
value
   +,−,∗: Nat×Nat→Nat | Int×Int→Int | Real×Real→Real
   /: Nat×Nat→̃Nat | Int×Int→̃Int | Real×Real→̃Real
   <,≤,=,≠,≥,> (Nat|Int|Real) → (Nat|Int|Real)
```

C.4 Comprehensive Expressions

$\mathcal{I}$: Comprehensive expressions are common in mathematics texts. They capture properties conveniently abstractly ▪

C.4.1 Set Enumeration and Comprehension

C.4.1.1 Set Enumeration

Let the below a's denote values of type A:

Set Enumerations

```
   {{}, {a}, {e₁,e₂,...,eₙ}, ...} ∈ A-set
   {{}, {a}, {e₁,e₂,...,eₙ}, ..., {e₁,e₂,...}} ∈ A-infset
```

C.4.1.2 Set Comprehension

The expression, last line below, to the right of the ≡, expresses set comprehension. The expression "builds" the set of values satisfying the given predicate. It is abstract in the sense that it does not do so by following a concrete algorithm.

Set Comprehension

```
type
   A, B
```

```
    P = A → Bool
    Q = A →̃ B
value
    comprehend: A-infset × P × Q → B-infset
    comprehend(s,P,Q) ≡ { Q(a) | a:A • a ∈ s ∧ P(a)}
```

C.4.1.3 Cartesian Enumeration

Let e range over values of Cartesian types involving $A, B, \ldots, C$, then the below expressions are simple Cartesian enumerations:

Cartesian Enumerations

```
type
    A, B, ..., C
    A × B × ... × C
value
    (e1,e2,...,en)
```

C.4.2 List Enumeration and Comprehension

C.4.2.1 List Enumeration

Let a range over values of type A, then the below expressions are simple list enumerations:

List Enumerations

```
    {⟨⟩, ⟨e⟩, ..., ⟨e1,e2,...,en⟩, ...} ∈ A*
    {⟨⟩, ⟨e⟩, ..., ⟨e1,e2,...,en⟩, ..., ⟨e1,e2,...,en,... ⟩, ...} ∈ Aω

    ⟨ a_i .. a_j ⟩
```

The last line above assumes a_i and a_j to be integer-valued expressions. It then expresses the set of integers from the value of e_i to and including the value of e_j. If the latter is smaller than the former, then the list is empty.

C.4.2.2 List Comprehension

The last line below expresses list comprehension.

List Comprehension

```
type
    A, B, P = A → Bool, Q = A →̃ B
value
```

```
    comprehend: $A^\omega \times P \times Q \overset{\sim}{\rightarrow} B^\omega$
    comprehend(l,P,Q) ≡ ⟨ Q(l(i)) | i in ⟨1..len l⟩ • P(l(i))⟩
```

C.4.3 Map Enumeration and Comprehension

C.4.3.1 Map Enumeration

Let (possibly indexed) *u* and *v* range over values of type *T*1 and *T*2, respectively, then the below expressions are simple map enumerations:

Map Enumerations

```
type
    T1, T2
    M = T1 $\underset{m}{\overrightarrow{\quad}}$ T2
value
    u,u1,u2,...,un:T1, v,v1,v2,...,vn:T2
    [], [u↦v], ..., [u1↦v1,u2↦v2,...,un↦vn] ∀ ∈ M
```

C.4.3.2 Map Comprehension

The last line below expresses map comprehension:

Map Comprehension

```
type
    U, V, X, Y
    M = U $\underset{m}{\overrightarrow{\quad}}$ V
    F = U $\overset{\sim}{\rightarrow}$ X
    G = V $\overset{\sim}{\rightarrow}$ Y
    P = U → Bool
value
    comprehend: M×F×G×P → (X $\underset{m}{\overrightarrow{\quad}}$ Y)
    comprehend(m,F,G,P) ≡ [ F(u) ↦ G(m(u)) | u:U • u ∈ dom m ∧ P(u) ]
```

C.5 Operations

C.5.1 Set Operations

C.5.1.1 Set Operator Signatures

Set Operator Signatures

value
18 ∈: A × A-**infset** → **Bool**
19 ∉: A × A-**infset** → **Bool**
20 ∪: A-**infset** × A-**infset** → A-**infset**
21 ∪: (A-**infset**)-**infset** → A-**infset**
22 ∩: A-**infset** × A-**infset** → A-**infset**
23 ∩: (A-**infset**)-**infset** → A-**infset**
24 \: A-**infset** × A-**infset** → A-**infset**
25 ⊂: A-**infset** × A-**infset** → **Bool**
26 ⊆: A-**infset** × A-**infset** → **Bool**
27 =: A-**infset** × A-**infset** → **Bool**
28 ≠: A-**infset** × A-**infset** → **Bool**
29 **card**: A-**infset** $\overset{\sim}{\to}$ **Nat**

C.5.1.2 Set Operation Examples

Set Operation Examples

examples
a ∈ {a,b,c}
a ∉ {}, a ∉ {b,c}
{a,b,c} ∪ {a,b,d,e} = {a,b,c,d,e}
∪{{a},{a,b},{a,d}} = {a,b,d}
{a,b,c} ∩ {c,d,e} = {c}
∩{{a},{a,b},{a,d}} = {a}
{a,b,c} \ {c,d} = {a,b}
{a,b} ⊂ {a,b,c}
{a,b,c} ⊆ {a,b,c}
{a,b,c} = {a,b,c}
{a,b,c} ≠ {a,b}
card {} = 0, **card** {a,b,c} = 3

C.5.1.3 Informal Set Operator Explication

The following is **not** a definition of RSL semantics. In RSL formulas we present an explication of RSL operators. Read, what appears as definitions, ≡, as [a kind of] identities.

18 ∈: The membership operator expresses that an element is a member of a set.
19 ∉: The nonmembership operator expresses that an element is not a member of a set.
20 ∪: 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.
21 ∪: 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.
22 ∩: The infix intersection operator. When applied to two sets, the operator gives the set whose members are in both of the two operand sets.
23 ∩: 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.
24 \: 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.
25 ⊆: The proper subset operator expresses that all members of the left operand set are also in the right operand set.
26 ⊂: 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.
27 =: The equal operator expresses that the two operand sets are identical.
28 ≠: The nonequal operator expresses that the two operand sets are not identical.
29 **card**: The cardinality operator gives the number of elements in a finite set.

C.5.1.4 Set Operator Explications

The set operations can be "equated" as follows:

Set Operator Explications

```
value
   s′ ∪ s″ ≡ { a | a:A • a ∈ s′ ∨ a ∈ s″ }
   s′ ∩ s″ ≡ { a | a:A • a ∈ s′ ∧ a ∈ s″ }
   s′ \ s″ ≡ { a | a:A • a ∈ s′ ∧ a ∉ s″ }
   s′ ⊆ s″ ≡ ∀ a:A • a ∈ s′ ⇒ a ∈ s″
   s′ ⊂ s″ ≡ s′ ⊆ s″ ∧ ∃ a:A • a ∈ s″ ∧ a ∉ s′
   s′ = s″ ≡ ∀ a:A • a ∈ s′ ≡ a ∈ s″ ≡ s⊆s′ ∧ s′⊆s
   s′ ≠ s″ ≡ s′ ∩ s″ ≠ {}
   card s ≡
      if s = {} then 0 else
      let a:A • a ∈ s in 1 + card (s \ {a}) end end
      pre s /∗ is a finite set ∗/
   card s ≡ chaos /∗ tests for infinity of s ∗/
```

C.5.2 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
```

C.5.3 List Operations

C.5.3.1 List Operator Signatures

List Operator Signatures

```
value
    hd: A^ω →̃ A
    tl: A^ω →̃ A^ω
    len: A^ω →̃ Nat
    inds: A^ω → Nat-infset
    elems: A^ω → A-infset
    .(.): A^ω × Nat →̃ A
    ⌢: A* × A^ω → A^ω
    =: A^ω × A^ω → Bool
    ≠: A^ω × A^ω → Bool
```

C.5.3.2 List Operation Examples

List Operation Examples

```
examples
    hd⟨a1,a2,...,am⟩=a1
    tl⟨a1,a2,...,am⟩=⟨a2,...,am⟩
    len⟨a1,a2,...,am⟩=m
    inds⟨a1,a2,...,am⟩={1,2,...,m}
    elems⟨a1,a2,...,am⟩={a1,a2,...,am}
    ⟨a1,a2,...,am⟩(i)=ai
```

```
⟨a,b,c⟩⁀⟨a,b,d⟩ = ⟨a,b,c,a,b,d⟩
⟨a,b,c⟩=⟨a,b,c⟩
⟨a,b,c⟩ ≠ ⟨a,b,d⟩
```

C.5.3.3 Informal List Operator Explication

The following is **not** a definition of RSL semantics. In RSL formulas we present an explication of RSL operators. Read, what appears as definitions, ≡, as [a kind of] identities.

- **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.
- ⁀: 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.
- ≠: The nonequal operator expresses that the two operand lists are not identical.

The operations can also be defined as follows:

C.5.3.4 List Operator Explications

The following is **not** a definition of RSL semantics. In RSL formulas we present an explication of RSL operators. Read, what appears as definitions, ≡, as [a kind of] identities.

List Operator Explications

```
value
    is_finite_list: A^ω → 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″)
```

C.5.4 Map Operations

C.5.4.1 Map Operator Signatures

Map Operator Signatures

```
value
[30] ·(·): M → A →̃ B
[31] dom: M → A-infset [domain of map]
[32] rng: M → B-infset [range of map]
[33] †: M × M → M [override extension]
[34] ∪: M × M → M [merge ∪]
[35] \: M × A-infset → M [restriction by]
[36] /: M × A-infset → M [restriction to]
[37] =,≠: M × M → Bool
[38] °: (A →m B) × (B →m C) → (A →m C) [composition]
```

C.5.4.2 Map Operation Examples

Map Operation Examples

```
value
[30] m(a) = b
[31] dom [a1↦b1,a2↦b2,...,an↦bn] = {a1,a2,...,an}
[32] rng [a1↦b1,a2↦b2,...,an↦bn] = {b1,b2,...,bn}
[33] [a↦b,a′↦b′,a″↦b″] † [a′↦b″,a″↦b′] = [a↦b,a′↦b″,a″↦b′]
[34] [a↦b,a′↦b′,a″↦b″] ∪ [a‴↦b‴] = [a↦b,a′↦b′,a″↦b″,a‴↦b‴]
[35] [a↦b,a′↦b′,a″↦b″]\{a} = [a′↦b′,a″↦b″]
[37] [a↦b,a′↦b′,a″↦b″]/{a′,a″} = [a′↦b′,a″↦b″]
[38] [a↦b,a′↦b′] ° [b↦c,b′↦c′,b″↦c″] = [a↦c,a′↦c′]
```

C.5.4.3 Informal 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.

C.5.4.4 Map Operator Explication

The following is **not** a definition of RSL semantics. In RSL formulas we present an explication of RSL operators. Read, what appears as definitions, ≡, as [a kind of] identities.
The map operations can also be defined as follows:

Map Operator Explications

```
value
  rng m ≡ { m(a) | a:A • a ∈ dom m }

  m1 † m2 ≡
    [ a↦b | a:A,b:B •
      a ∈ dom m1 \ dom m2 ∧ b=m1(a) ∨ a ∈ dom m2 ∧ b=m2(a) ]

  m1 ∪ m2 ≡ [ a↦b | a:A,b:B •
               a ∈ dom m1 ∧ b=m1(a) ∨ a ∈ dom m2 ∧ b=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
```

C.6 λ-Calculus + Functions

$\mathcal{I}$: The λ-Calculus is a foundation for the abstract specification language that RSL is ▪

C.6.1 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), ...
```

C.6.2 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).

C.6.3 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](λx•P)) ≡ λ y•P;
- **subst**([N/x](λ y•P)) ≡ λy• **subst**([N/x]P),

  if x≠y and y is not free in N or x is not free in P;
- **subst**([N/x](λy•P)) ≡ λz•**subst**([N/z]**subst**([z/y]P)),

  if y≠x and y is free in N and x is free in P
  (where z is not free in (N P)).

C.6.4 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) ≡ **subst**([N/x]M)

C.6.5 Function Signatures

For sorts we may want to postulate some functions:

Sorts and Function Signatures

```
type
   A, B, C
value
   obs_B: A → B,
   obs_C: A → C,
   gen_A: B×C → A
```

C.6.6 Function Definitions

Functions can be defined explicitly:

Explicit Function Definitions

```
value
   f: Arguments → Result
   f(args) ≡ DValueExpr
```

```
    g: Arguments ~→ Result
    g(args) ≡ ValueAndStateChangeClause
    pre P(args)
```

Or functions can be defined implicitly:

Implicit Function Definitions

```
value
    f: Arguments → Result
    f(args) as result
    post P1(args,result)

    g: Arguments ~→ Result
    g(args) as result
    pre P2(args)
    post P3(args,result)
```

The symbol $\stackrel{\sim}{\rightarrow}$ indicates that the function is partial and thus not defined for all arguments. Partial functions should be assisted by preconditions stating the criteria for arguments to be meaningful to the function.

C.7 Other Applicative Expressions

$\mathcal{I}$: RSL offers the usual collection of applicative constructs that functional programming languages (`Standard ML` [122, 122] or `F#` [91]) offer ▪

C.7.1 Simple let Expressions

Simple (i.e., nonrecursive) **let** expressions:

Let Expressions

let a = $\mathcal{E}_d$ **in** $\mathcal{E}_b$(a) **end**

is an “expanded” form of:

(λa.$\mathcal{E}_b$(a))($\mathcal{E}_d$)

C.7.2 Recursive let Expressions

Recursive **let** expressions are written as:

Recursive **let** Expressions

```
let f = λa:A • E(f) in B(f,a) end
```

is “the same” as:

```
let f = YF in B(f,a) end
```

where:

```
F ≡ λg•λa•(E(g)) and YF = F(YF)
```

C.7.3 Predicative let Expressions

Predicative **let** expressions:

Predicative **let** Expressions

```
let a:A • 𝒫(a) in ℬ(a) end
```

express the selection of a value a of type A which satisfies a predicate 𝒫(a) for evaluation in the body ℬ(a).

C.7.4 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,_,b⟩⌢ℓ = list in ... end

let [a↦b] ∪ m = map in ... end
let [a↦b,_] ∪ m = map in ... end
```

C.7.4.1 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
```

C.7.5 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⟩ ::= !
```

C.8 Imperative Constructs

𝓘: RSL offers the usual collection of imperative constructs that imperative programming languages (`Java` [86, 144] or `Oberon` (!) [155]) offer ■

C.8.1 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**.

C.8.2 Variables and Assignment

Variables and Assignment

```
0. variable v:Type := expression
1. v := expr
```

C.8.3 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
```

C.8.4 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
```

C.8.5 Iterative Conditionals

Iterative Conditionals

```
6. while expr do stm end
7. do stmt until expr end
```

C.8.6 Iterative Sequencing

Iterative Sequencing

```
8. for e in list_expr • P(b) do S(b) end
```

C.9 Process Constructs

𝓘: RSL offers several of the constructs that CS [100] offers ▪

C.9.1 Process Channels

As for channels we deviate from common RSL [84] in that we directly *declare* channels – and not via common RSL *objects* etc.

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).

C.9.2 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.

C.9.3 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.

C.9.4 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.

C.10 RSL Module Specifications

We shall not include coverage nor use of the RSL module concepts of *schemes, classes* and *objects*.

C.11 Simple RSL Specifications

Often, we do not want to encapsulate small specifications in schemas, 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
   ...
```

C.12 RSL$^+$: Extended RSL

Section **C.1** on page 173 covered standard RSL types. To them we now add two new types: Type names and RSL-Text.

We refer to Sect. **4.5.1.2.2** (the *An RSL Extension* box) Page 44 for a first introduction to extended RSL.

For uses of type name type and type name values and for the "generation" of RSL-Text to Sect. **4.5.1.2.2.1** (the `determine_Cartesian_parts` function), Sect. 2 (the `calc_Cartesian_parts` description prompt), Sect. **4.5.1.2.3.1** (the `determine_same_sort_parts_set` prompt), Sect. **4.5.1.2.3.2** (the `determine_alternative_sorts_part_set` function), Sect. **4.5.1.2.3.3** (the `calc_single_sort_parts_sort` description prompt), and Sect. **4.5.1.2.3.4** (the `calc_alternative_sort_parts_sort` prompt).

C.12.1 Type Names and Type Name Values

C.12.1.1 Type Names

- Let T be a type name.
- Then ηT is a type name value.
- And $\eta\mathbb{T}$ is the type of type names.

C.12.1.2 Type Name Operations

- η can be considered an operator.
    - It (prefix) applies, then, to type (T) identifiers and yields the name of that type.
    - Two type names, nT_i, nT_j, can be compared for equality: $\mathsf{nT}_i = \mathsf{nT}_j$ iff $i = j$.
- It, vice-versa, suffix applies to type name (nT) identifiers and yields the name, T, of that type: $\mathsf{nT}\eta = \mathsf{T}$.

C.12.2 RSL-Text

C.12.2.1 The RSL-Text Type and Values

- RSL-Text is the type name for ordinary, non-extended RSL texts.

We shall not here give a syntax for ordinary, non-extended RSL texts – but refer to [84].

C.12.2.2 RSL-Text Operations

- RSL-Texts can be compared and concatenated:
    - $\mathsf{rsl\text{-}text}_a = \mathsf{rsl\text{-}text}_b$
    - $\mathsf{rsl\text{-}text}_a \frown \mathsf{rsl\text{-}text}_b$

The ⁀ operator thus also applies, besides, lists (tuples), to RSL texts – treating RSL texts as (if they were) lists of characters.

C.13 Distributive Clauses

We clarify:

C.13.1 Over Simple Values

```
⊕ { a | a:A • a ∈ {a_1,a_2,...,a_n} } =
     if n>0 then a_1⊕a_1⊕...⊕a_n else
        case ⊕ of
             + → 0, − → 0, ∗ → 1, / → chaos, ∪ → {}, ∩ → {}, ...
```

end end

(f_1,f_2,...,f_n)(a) ≡ **if** n>0 **then** (f_1(a),f_2(a),...,f_n(a)) **else chaos end**

C.13.2 Over Processes

|| { p(i) | i:I • i ∈ {i_1,i_2,...,i_n} } ≡ **if** n>0 **then** p(i_1)||p(i_2)||...||p(i_n) **else** () **end**
⊓ { p(i) | i:I • i ∈ {i_1,i_2,...,i_n} } ≡ **if** n>0 **then** p(i_1)⊓p(i_2)⊓...⊓p(i_n) **else** () **end**
□ { p(i) | i:I • i ∈ {i_1,i_2,...,i_n} } ≡ **if** n>0 **then** p(i_1)□p(i_2)□...□p(i_n) **else** () **end**

Appendix D
Indexes

Contents

D.1 Definitions

D.2 **Concepts**

D.3 Method

Note: We have yet to index many more method principles, procedures, techniques and tools.

D.4 Symbols

D.5 Examples

D.6 Analysis Predicate Prompts

D.7 Analysis Function Prompts

D.8 Description Prompts

D.9 Attribute Categories

D.10 RSL Symbols