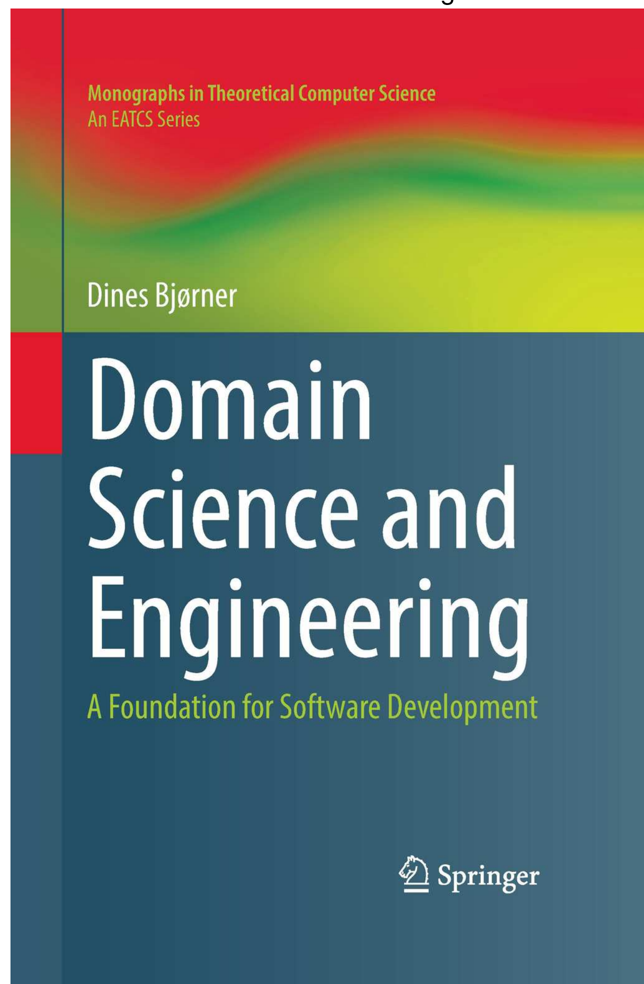
Dines Bjørner

Rigorous Domain Descriptions

A Compendium of Examples – a Torso

November 15, 2021: 16:12

The Domain Models are according to this book!



Publication date: 11.11.2021. ISBN 978-3-030-73483-1

www.imm.dtu.dk/˜db/2021/dd/dd.pdf

© Dines Bjørner

Dines Bjørner
Fredsvej 11
DK 2840 Holte
Denmark

Professor Emeritus
Technical University of Denmark
DK-2800 Kgs.Lyngby
Denmark

• This version is to **not** be distributed electronically.
• Please respect the © Dines Bjørner, 2021

Editorial Remarks as of November 15, 2021: 16:12 •

This compendium was collected and edited from 16 reports •
All of those were also accessible on the Internet •
The compendium editing started in July 2021 •
It is ongoing •
A target for a first completion is late Fall 2021 •
Sunday, August 1, 08:24am, 2021, I finished correcting undefined and multiple references •
Later I shall be properly editing each chapter text •
I added Chapter 8 on Sept. 24, 2021 •

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.

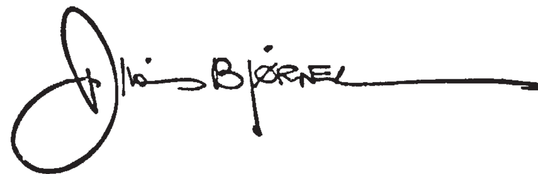
Domain Science & Engineering: In [55, Domain Science & Engineering – A Foundation for Software Development] we introduce the concept of domains and domain descriptions, and we present a method for analysing & describing domains. Studying the present compendium presumes that you are either reading or have read [55].

Examples: [55] proposes a rigorous approach to the analysis & description of domains, that is, [55, Chapters 4, 5 and 7], puts forward a pair of analysis & description calculi. In order to develop, hone and justify these calculi, I have, over the years, sketched a number of domain descriptions, some dates back from before the ideas of analysis & description calculi arose.

[55] is full of examples. Each illustrates a *methodological point*: a *principle*, a *technique* or a *tool*. But only by studying "across" an entire set of the *road transport* examples might the reader get a "feel" for the *software engineering* of a domain description.

This compendium then serves that purpose.

Caveat: The examples are "uneven". Many examples are not completed – remain a torso. Sections are set aside for narratives and formalisations, but these have yet to be done ! Some references may be erroneous ! ? Earlier examples do not fully reflect the "final" analysis & description calculi. Most recent examples do. We have annotated chapter titles with the approximate time of the conception of their domain description.



Dines Bjørner. November 15, 2021: 16:12
Fredsvej 11, DK-2840 Holte, Denmark

A Reading Guide

There are three parts:

- **Part I** contains just Chapter 0. It covers basic modeling techniques for such domains which are characterised by graph-like endurants.
- **Part II** contains Chapters 1–16. It covers a wide variety of domains – in the sense that we primarily aim at.
- **Part III** contains Chapters 17–18. The two domains covered here [Stock Exchange and an Internet based so-called *Virtual Shared Memory*] somehow fall outside a main characterisation of what domains are. Chapter 19 contains the bibliography.

Most chapters present the domain description in the following order:

```
Endurants                                   Perdurants
   External Qualities                          Channels
       Parts, obs_P                            Behaviours
       State                                       Signatures
    Internal Qualities                             Definitions
        Unique Identifiers                     System Initialisation
            uid_P
            all unique identifiers
            uniqueness of all parts
        Mereology
             mereo_P
             axiom: wellformedness
        Attributes
             attributes
             attr_A
```

Note the **obs_P, uid_P, mereo_P, attributes, attr_A** observer functions.

There are two Appendices:

- **A. A *Domain Analysis & Description* Primer**. If you do not have easy access to [55, Bjørner: *Domain Science & Engineering*, 2021] then this primer may help you.
- **B. An RSL Primer**. RSL is a primary tool of all pour domain descriptions.

• • •

A first sound **Foundation** for **Software Engineering:**

. [55, "The Monograph"]

Contents

Part I A Prelude “Domain” Description

0 Graphs [February 2021] 3
- 0.1 Introduction 4
- 0.2 Examples of Networks 5
- 0.3 Classical Mathematical Models 12
- 0.4 Our General Graph Model 16
- 0.5 The Nets Domain 34

Part II Main Examples

1 Rail Systems [1993–2007, 2020] 37
- 1.1 Endurants – Rail Nets and Trains 38
- 1.2 Transcendental Deduction 51
- 1.3 Perdurants 53
- 1.4 Closing 55

2 Road Transport [2007–2017] 57
- 2.1 The Road Transport Domain 58
- 2.2 External Qualities 58
- 2.3 Internal Qualities 61
- 2.4 Perdurants 68
- 2.5 System Initialisation 74

3 The Blue Skies [August 2021] 77
- 3.1 Introduction 77
- 3.2 Endurants 78
- 3.3 Perdurants 78
- 3.4 Conclusion 78

4 The 7 Seas [August 2021] 79
- 4.1 Introduction 80
- 4.2 Endurants 80
- 4.3 Perdurants 93
- 4.4 Conclusion 93

5 Pipelines [2008] 95
- 5.1 Photos of Pipeline Units and Diagrams of Pipeline Systems 96
- 5.2 Non-Temporal Aspects of Pipelines 96
- 5.3 State Attributes of Pipeline Units 105
- 5.4 Pipeline Actions 107
- 5.5 Connectors 110
- 5.6 On Temporal Aspects of Pipelines 112
- 5.7 A CSP Model of Pipelines 112
- 5.8 Conclusion 113

6 Simple Credit Card Systems [May 2016] ... 115
6.1 Introduction ... 115
6.2 Endurants ... 116
6.3 Perdurants ... 119

7 Weather Systems [November 2016] ... 125
7.1 On Weather Information Systems ... 126
7.2 Major Parts of a Weather Information System ... 127
7.3 Endurants ... 128
7.4 Perdurants ... 133
7.5 Conclusion ... 139

8 Automobile Assembly Lines [September 2021] ... 141
8.1 Introduction ... 143
8.2 A Domain Analysis & Description ... 144
8.3 Discussion ... 178
8.4 Conclusion ... 178

9 Document Systems [Summer 2017] ... 181
9.1 Introduction ... 182
9.2 A System for Managing, Archiving and Handling Documents ... 182
9.3 Principal Endurants ... 183
9.4 Unique Identifiers ... 183
9.5 Documents: A First View ... 184
9.6 Behaviours: An Informal, First View ... 186
9.7 Channels, A First View ... 187
9.8 An Informal Graphical System Rendition ... 188
9.9 Behaviour Signatures ... 188
9.10 Time ... 189
9.11 Behaviour "States" ... 190
9.12 Inter-Behaviour Messages ... 191
9.13 A General Discussion of Handler and Document Interactions ... 194
9.14 Channels: A Final View ... 195
9.15 An Informal Summary of Behaviours ... 195
9.16 The Behaviour Actions ... 198
9.17 Documents in Public Government ... 207
9.18 Documents in Urban Planning ... 207

10 Urban Planning [Fall 2017] ... 209
10.1 Structures and Parts ... 212
10.2 Unique Identifiers ... 215
10.3 Mereologies ... 220
10.4 Attributes ... 224
10.5 The Structure COMPILERs ... 234
10.6 Channel Analysis and Channel Declarations ... 236
10.7 The Atomic Part TRANSLATORs ... 240
10.8 Initialisation of The Urban Space Analysis & Planning System ... 255
10.9 Further Work ... 257

11 Swarms of Drones [November–December 2017] ... 261
11.1 An Informal Introduction ... 263
11.2 Entities, Endurants ... 264
11.3 Operations on Universe of Discourse States ... 280
11.4 Perdurants ... 283
11.5 Conclusion ... 300

12 Container Terminals [November 2017] 301
12.1 Introduction 304
12.2 Some Pictures 304
12.3 SECT 310
12.4 Main Behaviours 311
12.5 Endurants 313
12.6 Perdurants 332
12.7 Conclusion 358

13 Simple Retailer System [January 2021] 361
13.1 Two Approaches to Modeling 364
13.2 The retailer market Case Study 364
13.3 Endurants: External Qualities 368
13.4 Endurants: Internal Qualities 372
13.5 Merchandise 383
13.6 Perdurants 384
13.7 Conclusion 399

14 Shipping [Spring/Summer 2007, February–March 2021] 403
14.1 Informal Sketches of the Shipping Domain 404
14.2 Endurants: External Qualities 410
14.3 Endurants: Internal Qualities 412
14.4 Perdurants 421

15 Rivers [March–April 2021] 431
15.1 Introduction 431
15.2 External Qualities – The Endurants 433
15.3 Internal Qualities 435
15.4 Conclusion 438

16 Canals [March–April 2021] 439
16.1 Introduction 440
16.2 Visualisation of Canals 440
16.3 The Endurants 442
16.4 Conclusion 473

Part III Two Postlude “Domain” Examples

17 A Stock Exchange [January 2010] 477
17.1 Introduction 477
17.2 The Problem 477
17.3 A Domain Description 478
17.4 Tetsuo Tamai’s IEEE Computer Journal Paper 483

18 An “Extensible” Virtual Shared Memory [May–July 2010] 493
18.1 Introduction 494
18.2 XVSM Trees 497
18.3 XTree Operations 500
18.4 Indexing 505
18.5 Queries 507

Part IV Bibliography

19 Bibliography 513
19.1 Bibliographical Notes 513
19.2 References 513

Part V Appendix

A Domain Analysis & Description: A Primer 523
A.1 Domains 524
A.2 Endurants 524
A.3 Space, State and Time 535
A.4 Perdurants 537

B An RSL Primer 545
B.1 Types 547
B.2 The RSL Predicate Calculus 550
B.3 Concrete RSL Types: Values and Operations 551
B.4 λ-Calculus + Functions 559
B.5 Other Applicative Expressions 561
B.6 Imperative Constructs 563
B.7 Process Constructs 565
B.8 Simple RSL Specifications 566
B.9 RSL Module Specifications 567

Part I
A Prelude "Domain" Description

Chapter 0
Graphs [February 2021]

Contents

0.1 Introduction 4
0.1.1 Critique of Classical Mathematical Modeling of Nets 4
0.1.2 The Thesis 5
0.1.3 Structure of This Report 5
0.2 Examples of Networks 5
0.2.1 Overland Transport Nets 6
0.2.1.1 Road Nets 6
0.2.1.2 Rail Nets 6
0.2.1.3 Pipeline Nets 6
0.2.2 Natural Trees with Roots 7
0.2.3 Waterways 7
0.2.3.1 Rivers, Lakes, Deltas and Oceans 8
0.2.3.2 General 8
0.2.3.3 Visualisation of Rivers and Canals 10
0.2.3.3.1 Rivers 10
0.2.3.3.2 Deltas 11
0.2.3.3.3 Canals and Water Systems 11
0.2.3.3.4 Locks 11
0.2.4 Conclusion 12
0.3 Classical Mathematical Models 12
0.3.1 Graphs 14
0.3.1.1 General Graphs 14
0.3.1.1.1 Some Mathematics ! 14
0.3.1.1.2 Some Graphics ! 15
0.3.1.2 Unique Identification of Vertices and Edges 15
0.3.1.3 Paths 16
0.3.1.4 Directed Graphs 16
0.3.1.5 Acyclic Graphs 16
0.3.1.6 Connected Graphs and Trees 16
0.3.1.7 Vertex In- and Out-Degrees of Directed Graphs 16
0.4 Our General Graph Model 16
0.4.1 The External Qualities 16
0.4.1.1 A “Global” Graph 17
0.4.1.2 Varieties of Endurants 17
0.4.1.2.1 Road Net Endurants 17
0.4.1.2.2 Rail Endurants 17
0.4.1.2.3 Pipeline Endurants 18
0.4.1.2.4 River Net Endurants 18
0.4.2 Internal Qualities 19
0.4.2.1 Unique Identifiers 19
0.4.2.2 Auxiliary Functions 20
0.4.2.2.1 Extraction Functions: Unique Identifies 20
0.4.2.2.2 Retrieval Functions 20
0.4.2.3 Wellformedness 21

0.4.2.4 Unique Identifier Examples 21
0.4.2.4.1 Road Net Identifiers 21
0.4.2.4.2 Rail Net Identifiers 21
0.4.2.4.3 Pipeline Net Identifiers 22
0.4.2.4.4 River Net Identifiers 22
0.4.2.5 Mereologies 22
0.4.2.5.1 Mereology of Undirected Graphs 23
0.4.2.5.2 Wellformedness of Mereologies 23
0.4.2.5.3 Mereology of Directed Graphs 23
0.4.2.5.4 In- and Out-Degrees 24
0.4.2.5.5 Paths of Undirected Graphs 24
0.4.2.5.6 Paths of Directed Graphs 24
0.4.2.5.7 Connectivity 25
0.4.2.5.8 Acyclic Graphs, Trees and Forests 25
0.4.2.5.9 Forest 26
0.4.2.5.10 Mereology Examples 26
0.4.2.6 Attributes 30
0.4.2.6.1 Graph Labeling 30
0.4.2.6.2 General Net Attributes 30
0.4.2.6.3 Road Net Attributes 31
0.4.2.7 Summing Up 32
0.4.2.7.1 A Summary of The Example Endurant Models 32
0.4.2.7.2 Initial Conclusion on Labeled Graphs and Example Domains 33
0.5 The Nets Domain 34
0.5.1 Some Introductory Definitions 34

We study formalisations of graphs as they are found in the conventional *Graph Theory* literature, but as we would formalise graphs in the style of *Domain Analysis & Description* [55, Bjørner, 2021]. The title of this compendium, *A Graph Domain*, shall indicate that we shall present *graphs*, not in the conventional mathematical style, but according to the principles, techniques and tools of [55]. That is, both as mathematical entities and as, albeit abstract, i.e., not necessarily manifest, phenomena of the world. As such we shall endow vertices and edges of graphs with unique identifiablity, mereology – to model the edge/vertex relations, and attributes – to model vertex and edge labeling, i.e., to model properties of vertices and edges, including directedness! Appendix **A** (pages 523–544) presents an ultra-short introduction, a primer, to the *domain analysis & description calculi* underlying this compendium.

0.1 Introduction

0.1.1 Critique of Classical Mathematical Modeling of Nets

Classical mathematical modeling of (road and rail) transport nets, river systems, canal systems, etc., misses some, to us, important points.

The point being that the more-or-less individual elements of these systems, the links (edges) and hubs (nodes, vertices) each have their unique identity, their mereology and their attributes, and that it is these internal qualities of edges and nodes that capture the "real" meaning of the nets.

In the mathematical models graph edges and vertices have no internal qualities: they are treated merely as syntactic entities.

We strive, in *domain analysis & description* [55], to model *first* the **syntactic** properties of manifest phenomena, then the **semantic** properties. Naturally we cannot model their **pragmatics**!

0.1.2 The Thesis

The thesis of this compendium is that the *domain analysis & description* `principles, techniques` and `tools` as brought forward in [55, 48, 54, 51, 52] is a more proper way to model nets.

0.1.3 Structure of This Report

- In Sect. **0.2** we casually pictorialise a number of domains whose compositions basically amount to graphs. These examples are:
  - **Road Nets** [Sect. **0.2.1.1** on the following page],
  - **Railways** [Sect. **0.2.1.2** on the next page],
  - **Pipelines** [Sect. **0.2.1.3** on the following page],
  - **Rivers** [Sect. **0.2.3.1** on page 8] and
  - **Canals** [Sect. **0.2.3.3.3** on page 11].
- In Sect. **0.3** we prepare the ground by presenting a minimum account of graphs as they are usually first introduced in textbooks.

Correlated narratives and formalisations for these domains are shown, spread all over this compendium as follows:

- **Road Nets:** Sections:
  - **0.2.1.1** [Pictures],
  - **0.4.1.2.1** [Endurants],
  - **0.4.2.4.1** [Unique Identifiers],
  - **0.4.2.5.10** [Mereology] and
  - **0.4.2.6.3** [Attributes].
- **Railways:** In the compendium-proper we pictorialise railways in Sect. refnets-ex:Rail Nets [Pictures]. In all:
  - **0.2.1.2** [Pictures],
  - **1.1** [Endurants],
  - **0.4.2.4.2** [Unique Identifiers], and
  - **1.1.2.2** [Mereology].
- **Pipelines:** Sections:
  - **0.2.1.3** [Pictures],
  - **0.4.1.2.3** [Endurants],
  - **0.4.2.4.3** [Unique Identifiers], and
  - **0.4.2.5.10** [Mereology].
- **Rivers:** Sections:
  - **0.2.3.1** [Pictures],
  - **0.4.1.2.4** [Endurants],
  - **0.4.2.4.4** [Unique Identifiers], and
  - **0.4.2.5.10** [Mereology].
- **Canals:** Other than Sects. **0.2.3.3.3** this compendium does not yet illustrate a systematic canal system description.

0.2 Examples of Networks

We shall consider a widest set of networks,

0.2.1 Overland Transport Nets

By overland transport nets we mean such which are either placed on the ground, or underground, as tunnels, or through mountains, also as tunnels, or placed on bridges over valleys, etc.

0.2.1.1 Road Nets

Road nets are for the conveyance of automobiles: private cars, buses, trucks, etc.

Fig. 0.1 Left: The Netherlands. R: Scotland

Fig. 0.2 L & R: European Road Infrastructure

0.2.1.2 Rail Nets

Rail nets are for the conveyance of passenger and freight trains.

Rail nets and train traffic on these are narrated and formalised in Chapter 1:

0.2.1.3 Pipeline Nets

Pipelines are for the conveyance of fluids: water, natural gas, hydrogen, oil, etc.

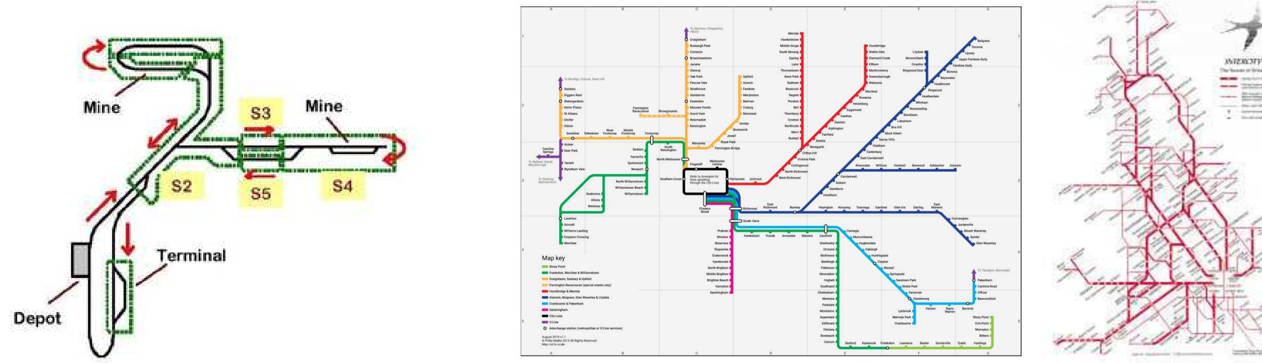


Fig. 0.3 Example Railway Nets

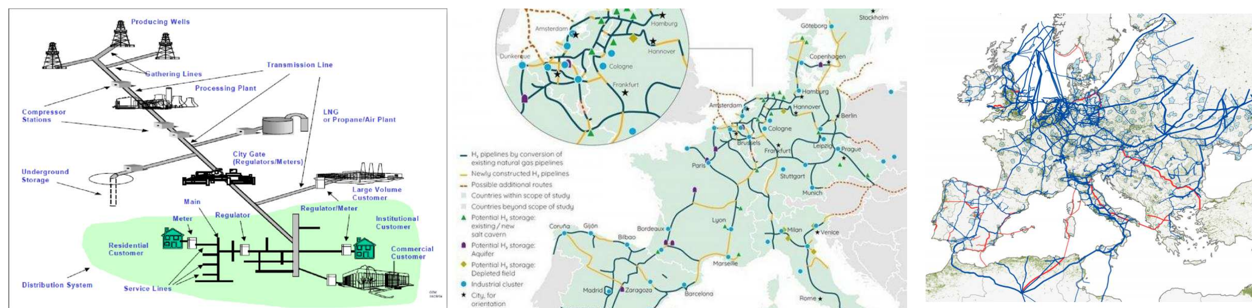


Fig. 0.4 Oil or Gas Field; European Gas and Hydrogen Pipelines

0.2.2 Natural Trees with Roots

0.2.3 Waterways

Canals are artificial or human-made channels or waterways that are used for navigation, transporting water, crop irrigation, or drainage purposes. Therefore, a canal can be considered an artificial version of a river. Canals are artificial or human-made channels or waterways that are used for navigation, transporting water, crop irrigation, or drainage purposes. Therefore, a canal can be considered an artificial version of a river.

Rivers, on the other hand, are naturally flowing watercourses, and typically flow until discharging their water into a lake, sea, ocean, or another river, while canals are constructed to connect existing rivers, seas, or lakes. However, occasionally some rivers do not discharge their water into lakes, seas, oceans, or other rivers. Rivers that do not empty into another body of water might flow into the ground or simply dry up before reaching another body of water. Additionally, small rivers can also be referred to as streams, rivulets, creeks, rills, or brooks.

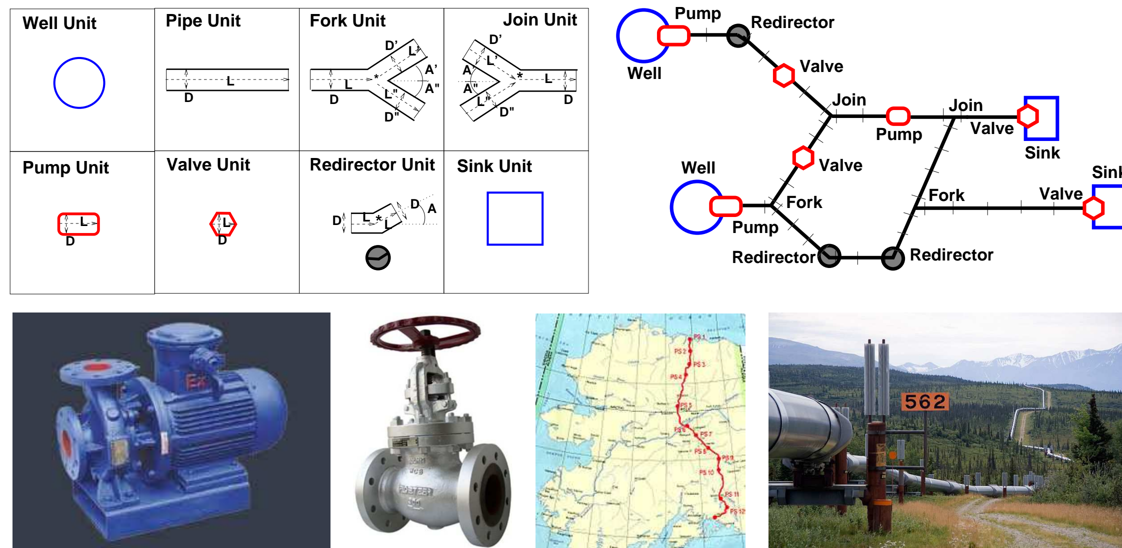


Fig. 0.5 Oil unit graphics; a simple oil pipeline.
A pump; a valve; the Trans-Alaska Pipeline System (TAPS); TAPS pipes, re-directors and 'heat pipes'.

Fig. 0.6 A Japanese Maple [Portland, Oregon, US] and an Angel Oak Tree [South Carolina, US]

0.2.3.1 Rivers, Lakes, Deltas and Oceans

By waterways we mean rivers, canals, lakes and oceans – such as are navigable by vessels: barges, boats and ships.

• • •

Disclaimer: At present ("great") lakes and the oceans (there are two !) are not included in this modeling effort.

0.2.3.2 General

Canals are artificial or human-made channels or waterways that are used for navigation, transporting water, crop irrigation, or drainage purposes. Therefore, a canal can be considered an artificial version of a river. Canals are artificial or human-made channels or waterways that are

Fig. 0.7 Drawings of Banyan Trees

Fig. 0.8 A Dragon Tree [Yemen] and an Aspen Tree Root [Colorado, US]

used for navigation, transporting water, crop irrigation, or drainage purposes. Therefore, a canal can be considered an artificial version of a river.

Rivers, on the other hand, are naturally flowing watercourses, and typically flow until discharging their water into a lake, sea, ocean, or another river, while canals are constructed to connect existing rivers, seas, or lakes. However, occasionally some rivers do not discharge their water into lakes, seas, oceans, or other rivers. Rivers that do not empty into another body of water might flow into the ground or simply dry up before reaching another body of water. Additionally, small rivers can also be referred to as streams, rivulets, creeks, rills, or brooks.

The natural water system of the earth includes 71% ocean with land continents being traversed by brooks, rivers, lakes and river deltas.

Headwaters are streams and rivers (tributaries) that are the source of a stream or river.

A tributary is a river or stream that flows into another stream, river, or lake.

A delta is a large, silty area at the mouth of a river at which the river splits into many different slow-flowing channels that have muddy banks. New land is created at deltas. Deltas are often triangular-shaped, hence the name (the Greek letter 'delta' is shaped like a triangle).

The trunk is the main course of river.

Confluence: In geography, a confluence (also: conflux) occurs where two or more flowing bodies of water join together to form a single flow. A confluence can occur in several configurations: at the point where a tributary joins a larger river (main stem); or where two streams meet to become the source of a river of a new name; or where two separated channels of a river (forming a river island) rejoin at the downstream end.

Towns and Harbours: In this report we model towns. That is, we therefore also model that towns have harbours – allowing river (and canal) vessels to berth (a place for mooring in a harbour) for cargo loading, unloading and resting.

0.2.3.3 Visualisation of Rivers and Canals

0.2.3.3.1 Rivers

Figures 0.9 and 0.10 illustrate a number of rivers.

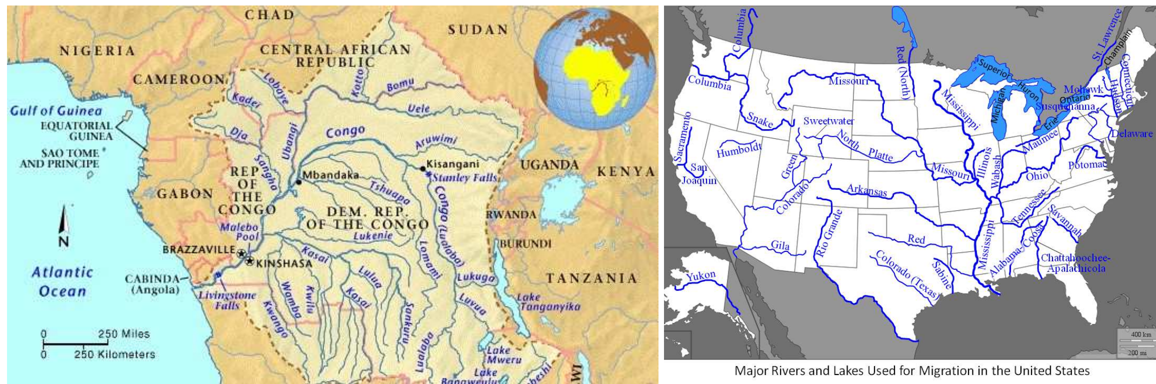


Fig. 0.9 The Congo and the US Rivers

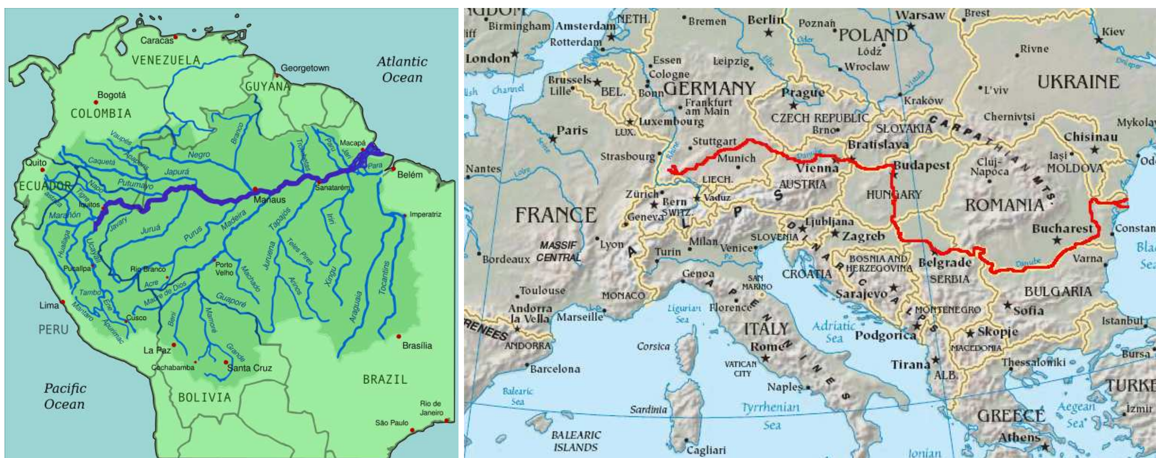


Fig. 0.10 The Amazon and The Danube Rivers

0.2.3.3.2 Deltas

We illustrate four deltas, Fig. 0.11:

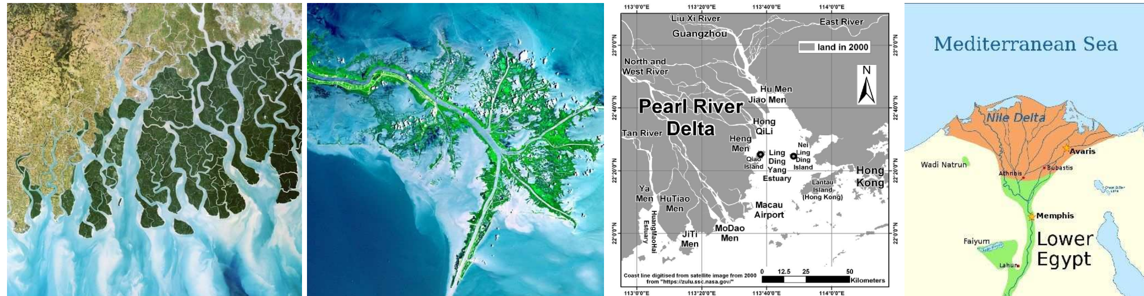


Fig. 0.11 The Ganges, Mississippi, Pearl and the Nile Deltas

0.2.3.3.3 Canals and Water Systems

We illustrate just four ship/barge/boat and water level control canal systems, Figs. 0.12, 0.13, 0.14 on the following page and 0.15 on page 13.

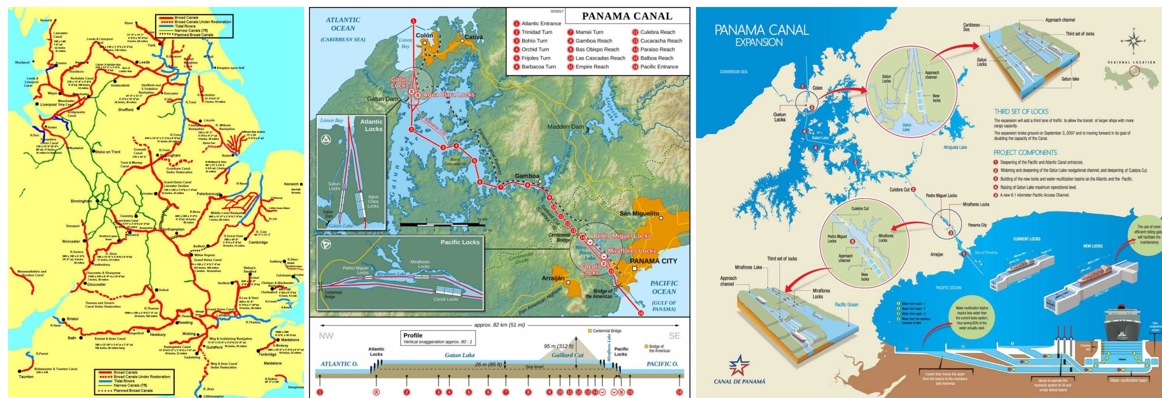


Fig. 0.12 UK Canals and The Panama Canal

The rightmost figure of Fig. 0.15 is from the Dutch *Rijkswaaterstaat*: www.rijkswaterstaat.nl/english/.

0.2.3.3.4 Locks

A lock is a device used for raising and lowering boats, ships and other watercraft between stretches of water of different levels on river and canal waterways. The distinguishing feature of a lock is a fixed chamber in which the water level can be varied. Locks are used to make a river more easily

Fig. 0.13 The Swedish Göta Kanal

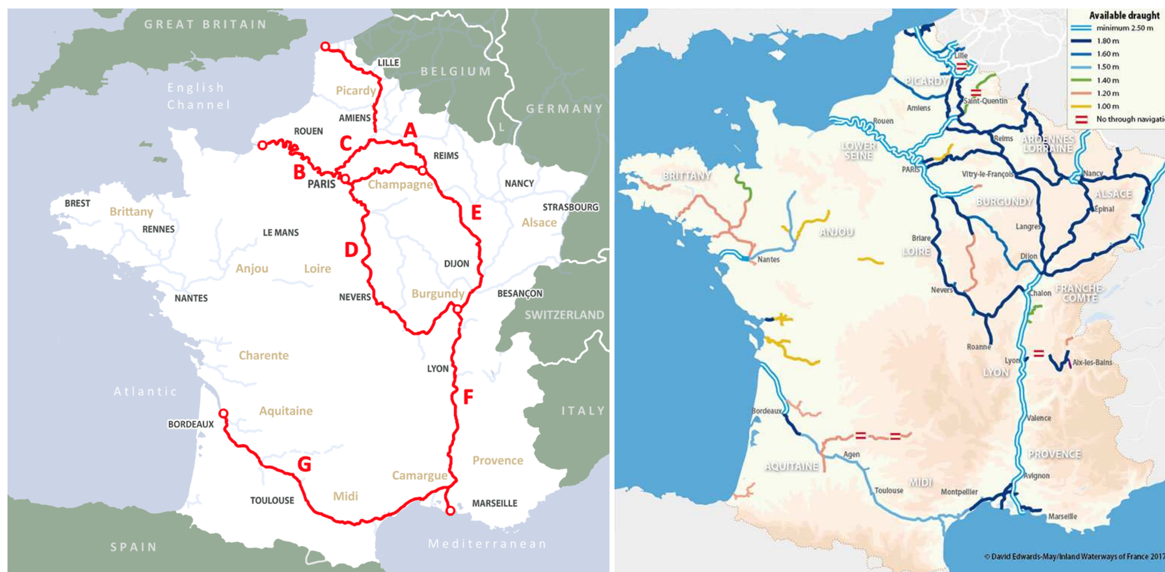


Fig. 0.14 French Rivers and Canals

navigable, or to allow a canal to cross land that is not level. Later canals used more and larger locks to allow a more direct route to be taken.[1]

We illustrate a number of locks: Figs. 0.16 on the facing page and 0.17 on page 14.

0.2.4 Conclusion

0.3 Classical Mathematical Models

We refer to standard textbooks in Graph Theory:

[1] https://en.wikipedia.org/wiki/Lock_(water_navigation)

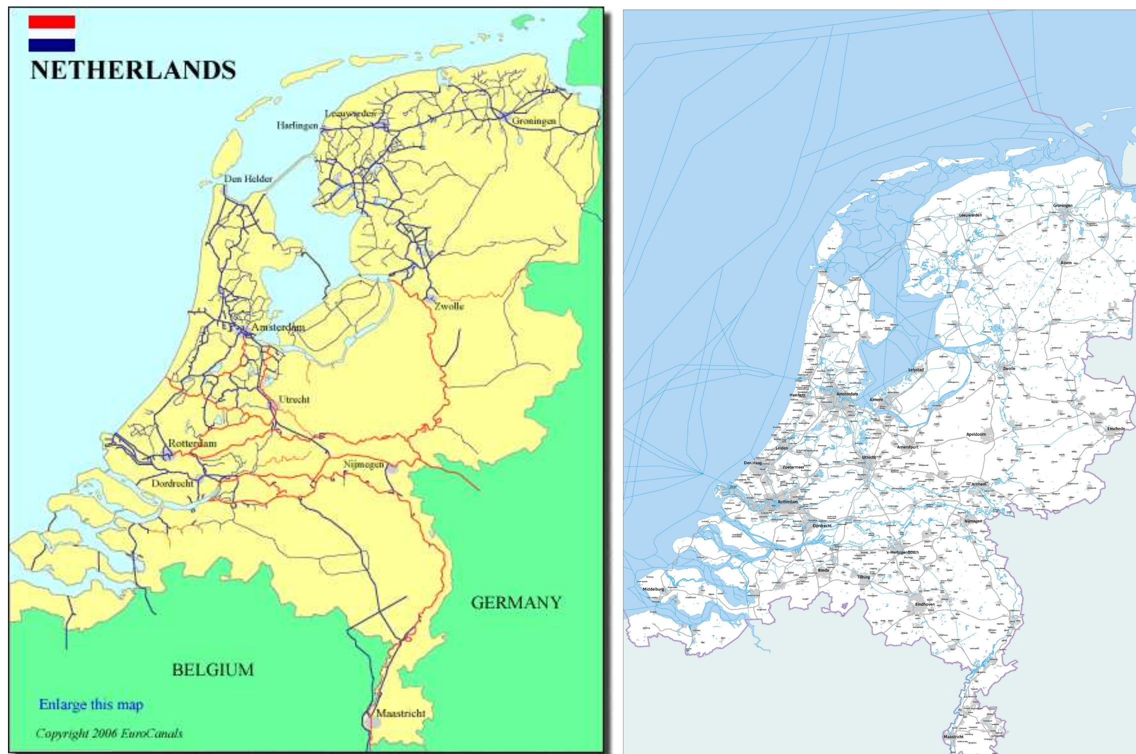


Fig. 0.15 Dutch Rivers and Canals

Fig. 0.16 Inland Canal Locks

- **Claude Berge: Graphs** [11, 12, 1958–1978, 1st–2nd ed.]
- **Oystein Ore: Graphs and their Uses** [127, 1963]
- **Frank Harrary: Graph Theory** [97, 1972]
- **J.A. Bondy and U.S.R. Murty: Graph Theory with Applications** [73, 1976]
- **S. Even: Graph Algorithms** [81, 1979]

or these `Wikipedia` Web pages:

Fig. 0.17 Harbour Canal Locks

a. **Graph Theory**
 en.m.wikipedia.org/wiki/Graph_theory
b. **Graphs: Discrete Mathematics**
 en.m.wikipedia.org/wiki/Graph_(discrete_mathematics)
c. **The Hamiltonian Path Problem**
 en.m.wikipedia.org/wiki/Hamiltonian_path_problem
d. **Glossary of Graph Theory**
 en.wikipedia.org/wiki/Glossary_of_graph_theory_terms

0.3.1 Graphs

0.3.1.1 General Graphs

We refer to en.wikipedia.org/wiki/Glossary_of_graph_theory_terms#A.

0.3.1.1.1 Some Mathematics !

From (a.): in one restricted but very common sense of the term,a graph is an ordered pair

- $G = (V, E)$, where
- V, is a set of vertices (also called nodes or points), and
- $E \subseteq \{\{x, y\} \mid x, y \in V\}$ is a set of edges (also called links or lines), which are unordered pairs of vertices.
- If $x = y$ then the edge s a *1-loop*, cf. upper leftmost edge of **G0** of Fig. 0.18 on the facing page.

To avoid ambiguity, this type of object may be called precisely an undirected simple graph, cf. graph **G0** of Fig. 0.18.

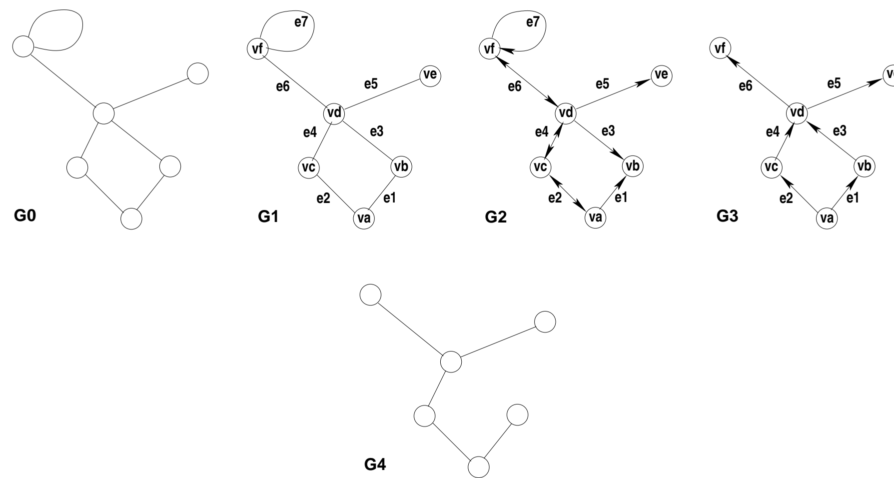


Fig. 0.18 Graphs

0.3.1.1.2 Some Graphics !

Figure 0.18 shows five similarly "shaped" graphs. Figure 0.19 shows how these could have been drawn differently.

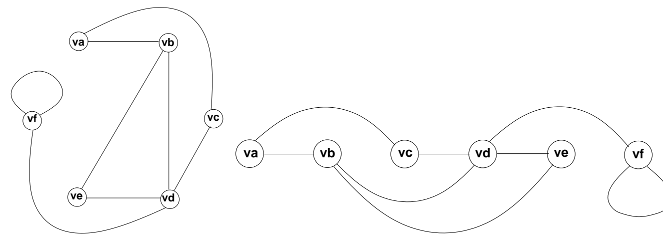


Fig. 0.19 Graphs

0.3.1.2 Unique Identification of Vertices and Edges

There is no way it can be avoided[2]. It simply makes no sense to not bring in that vertices and edges are uniquely identified. So we identify vertices and edges, cf. graph **G1** of Fig. 0.18. When, in classical graph theory, labeling of vertices and edges is introduced it is either for convenience of reference or for property attribution, as we shall later see.

With unique identification there is no problem with multiple edges between any pair of vertices.

[2] Sections 2.2.2.1 and 2.2.2.2 of [55, Bjørner] makes this clear: Unique identifiablity is an unavoidable fact of any world.

0.3.1.3 Paths

A *vertex path* is a sequence, $\langle \mathsf{v}_i, \mathsf{v}_j, ..., \mathsf{v}_k, \mathsf{v}_{k+1}, ..., \mathsf{v}_, \rangle$, of two or more vertices such that vertex v_k is *adjacent* to vertex v_{k+1} if there is an edge between them. Similar notion of *edge paths* and *vertex-edge-vertex paths* can be defined.

Graphs thus define possibly infinite sets of possibly infinite paths.

0.3.1.4 Directed Graphs

Directed graphs have directed edges, shown, in graph pictures, by affixing arrows to edges, see graph **G2** of Fig. 0.18 on the preceding page.

- G and V is as before, but
- $E \subseteq \{(x,y) \mid x,y \in V,\}$.

Directed graphs still define possibly infinite sets of possibly infinite paths. The vertex sequence $\langle \mathsf{v}_a, \mathsf{v}_c, \mathsf{v}_d, \mathsf{v}_f, \mathsf{v}_f \rangle$ is a path of graph **G2** of Fig. 0.18 on the previous page.

0.3.1.5 Acyclic Graphs

An *acyclic* graph is a graph none of whose vertex paths contain any vertex at most once. Graph **G3** of Fig. 0.18 on the preceding page is an acyclic graph.

0.3.1.6 Connected Graphs and Trees

A graph is *connected* if and only if for any two its vertices v_i, v_j there exists a path from v_i to v_j. A graph that is connected and is acyclic is a *tree*, cf. graph **G4** of Fig. 0.18 on the previous page.

0.3.1.7 Vertex In- and Out-Degrees of Directed Graphs

By the *in-degree* of a vertex of a [directed] graph is meant the number of edges incident upon that vertex. By the *out-degree* of a vertex of a [directed] graph is meant the number of edges emanating from that vertex. In an un-directed graph the in- and out-degrees of any vertex are identical. In an acyclic graph there necessarily must be one or more vertices whose in-degrees are zero. And in an acyclic graph there necessarily must be one or more vertices whose out-degrees are zero.

0.4 Our General Graph Model

0.4.1 The External Qualities

We refer to [55, Chapter 4].

1. Our domain is that of graphs.
2. From graphs one can observe sets of vertices,
3. and edges.

type

1. G
2. V
3. E

value
2. obs_Vs: G → V-**set**
3. obs_Es: G → E-**set**

Please notice that nothing is said about how vertices and edges relate. That is an issues of mereology, cf. [55, Sect. 5.3.1].

0.4.1.1 A “Global” Graph

4. For ease of reference we can postulate a[n arbitrary] graph.

value
4. g:G

0.4.1.2 Varieties of Endurants

Some domains warrant explication (e.g., renaming) of the vertices and edges or “collapsing” these into sets over a variety of units.

0.4.1.2.1 Road Net Endurants

5. A road as a pair of hubs and links.
6. Substitute vertices for *hubs, H*, i.e., street intersections,
7. and edges for *links, L*, i.e., street segment with no intersections.

type
5. RN = H-**set** × L-**set** [≃ G for Graphs]
6. H [≃ V for Graphs]
7. L [≃ E for Graphs]

0.4.1.2.2 Rail Endurants

8. So a graph, i.e., a railway net, RN, consists of a set of rail units.
9. A rail units is
   a. either a simple, linear [or curved] unit, LU,
   b. or a switch, SU,
   c. or a cross-over, XU,
   d. or a cross-over switch, CS,
   e. or ...

We refer to Fig. 1.1 on page 39 of Sect. **1.1.1.1** on page 38.

type
8. RN = RU-**set** [≃ G for Graphs]

9. RU == LU | SU | XU | XS | SC
9a. LU :: LiU
9b. SU :: SiU
9c. XU :: XiU
9d. CS :: CiS
9e. ...

Again; here we say nothing more about these units.

0.4.1.2.3 Pipeline Endurants

10. So a graph, i.e., a pipeline net, PN, consists of a set of pipeline units, PLU.
11. A pipeline units is
    a. either a source (a well), WU,
    b. or a pump, PU,
    c. or a pipe, LU,
    d. or a valve, VU,
    e. or a fork, FU,
    f. or a join, JU,
    g. or a sink, SU.
12. All pipeline units are distinct.

```
type
10. PN
11. PLU == WU | PU | LU | VU | FU | JU | SU
11a. WU :: W
11b. PU :: P
11c. LU :: L
11d. VU :: V
11e. FU :: F
11f. JU :: J
11g. SU :: S
value
11. obs_PLUs: PN → PLU-set
axiom
12. WU∩PU={} ∧ WU∩LU={} ∧ WU∩LU={} ∧ WU∩VU={}∧ WU∩FU={} ∧ WU∩JU={} ∧ WU∩SU={} ∧
12. PU∩LU={} ∧ PU∩VU={}∧ PU∩FU={} ∧ PU∩JU={} ∧ PU∩SU={} ∧
12. LU∩VU={} ∧ LU∩FU={} ∧ LU∩JU={} ∧ LU∩SU={} ∧
12. VU∩FU={} ∧ VU∩JU={} ∧ VU∩SU={} ∧
12. FU∩JU={} ∧ FU∩SU={} ∧
12. JU∩SU={}
```

Again; here we say nothing more about these units.

0.4.1.2.4 River Net Endurants

13. A river net is modeled as a graph, more specifically as a tree. The *root* of that river net tree is the mouth (or delta) of the river net. The *leaves* of that river net tree are the sources of respective trees. Paths from leaves to the root define *flows* of water.
14. We can thus, from a river net observe vertices
15. and edges.
16. River vertices model either a *source:* **so:SO**, a *mouth:* **mo:MO**, or possibly some *confluence:* **ko:KO**.

A river may thus be "punctuated" by zero or more confluences, k:KO.
A confluence defines the joining a 'main' river with zero[3] or more rivers into that 'main' river.
We can talk about the "upstream" and the "downstream" of rivers from their confluence.

17. River edges model *stretches:* st:ST.
A stretch is a linear sequences of simple, se:SE, or composite ce:CE, river elements.
18. River elements are either simple: (ch) river channels, which we shall call *river channels*: **CH**, or (la) lakes: **LA**, or (lo) locks: **LO**, or (wa) waterfalls (or *rapids*): **WA**, or (da) dams: **DA**, or (to) towns (cities, villages): **to:TO**[4]; or composite, ce:CE: a dam with a lock, **(da:DA,la:LA)**, a town with a lake, **(to:TO,la:LA)**, etcetera; even a town with a lake and a confluence, **to:TO,la:LA,ko:KO**. Etcetera.

```
type
13. RiN
14. V
15. E
16. SO, MO, KO
17. ST = (SE|CE)*
18. CH, LA, LO, WA, KO, DA, TO
18. SE = CH | LA | LO | WA | DA | TO
18. DaLo, WaLo, ToLa, ToLaKo, ...
18. CE = DaLo | WaLo | ToLa | ToLaKo | ...
value
16. obs_Vs: RiN → V-set
16.     axiom
16.        ∀ g:G,vs:V-set•vs ∈ obs_Vs(g) ⇒ vs≠{}
16.      ∧  ∀ v:V•v ∈ vs ⇒ is_SO(v) ∨ is_KO(v) ∨ is_MO(v)
17. obs_Es: RiN → E-set
17.     axiom
17.        ∀ g:G,es:E-set•es ∈ obs_Es(g) ⇒ es≠{}
17.      ∧  ∀ e:E•e ∈ es ⇒ is_ST(e)
17. obs_ST: E → ST
13. xtr_In_Degree_0_Vertices: RiN → SO-set
13. xtr_Out_Degree_0_Vertex: RiN → MO
```

0.4.2 Internal Qualities

We refer to [55, Chapter 5]

0.4.2.1 Unique Identifiers

We refer to [55, Sect. 5.2]

19. Each vertex has a unique identifier.
20. Each edge has a unique identifier.

[3] Normally, though, one would expect, not zero, but one

[4] Towns is here really a synonym for river harbours, places along the river (or a canal) where river vessels can stop (moor) for the loading and unloading of cargo and for resting.

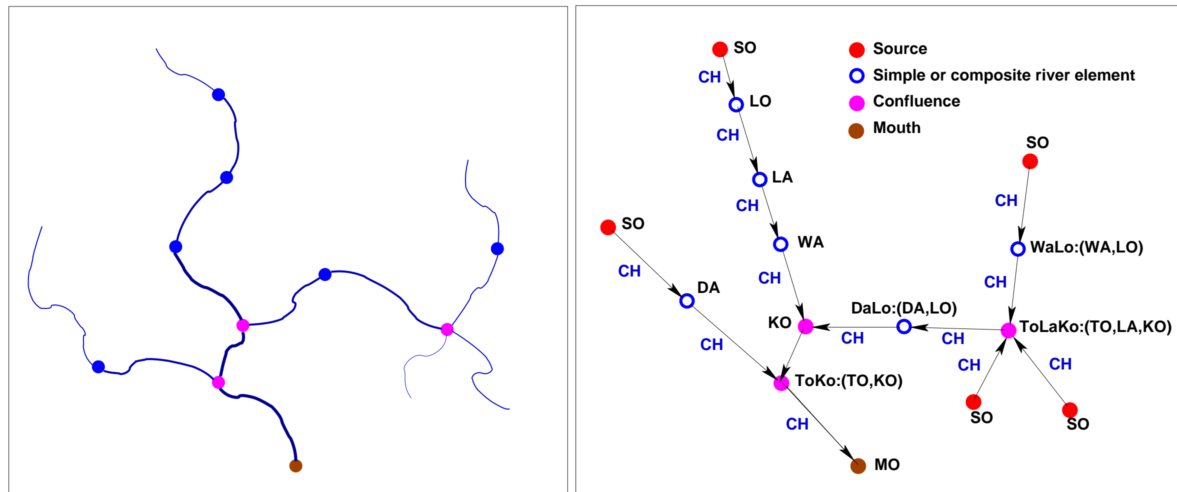


Fig. 0.20 The "Composition" of a River Net: Right Tree is an abstraction of the Left Tree

type
19. V_UI
20. E_UI
value
19. uid_V: V → V_UI
20. uid_E: E → E_UI

0.4.2.2 Auxiliary Functions

0.4.2.2.1 Extraction Functions: Unique Identifies

21. We can calculate the set of all unique vertex identifiers of a graph,
22. and all unique edge identifiers of a graph,
23. and all unique identifiers of vertices and edges of a graph.

value
21. xtr_V_UIs: G → V_UI-**set**, xtr_V_UIs(g) ≡ { uid_v(v) | v:V•v ∈ obs_Vs(g) }
22. xtr_E_UIs: G → EI-**set**, xtr_E_UIs(g) ≡ { uid_E(e) | e:E•e ∈ obs_Es(g) }
23. xtr_U_UIs: G → (VI|EI)-**set**, xtr_UIs(g) ≡ xtr_V_UIs(g) ∪ xtr_E_UIs(g)

0.4.2.2.2 Retrieval Functions

24. Given a unique vertex identifier of a graph one can retrieve, from the graph, the vertex of that identification.
25. Given a unique edge identifier of a graph one can retrieve, from the graph, the edge of that identification.

value

24. retr_V: V_UI → G $\stackrel{\sim}{\to}$ V
24. retr_V(v_ui)(g) ≡ **let** v:V • v ∈ obs_Vs(g) ∧ v_ui = uid_V(v) **in** v **end**, **pre**: e_ui ∈ xtr_E_UIs(g)
25. retr_E: EI → G $\stackrel{\sim}{\to}$ E
25. retr_E(ei)(g) ≡ **let** e:E • e ∈ obs_Es(g) ∧ e_ui = uid_E(e) **in** e **end**, **pre**: e_ui ∈ xtr_E_UIs(g)

0.4.2.3 Wellformedness

26. Vertex and edge identifiers are all distinct.
27. Each vertex and each edge has a distinct unique identifier.

axiom
26. ∀ g:G • xtr_V_UIs(g) ∩ xtr_E_UIs(g) = {}
27. **card** obs_Vs(g)=**card** xtr_V_UIs(g) ∧ **card** obs_Es(g)=**card** xtr_E_UIs(g)

0.4.2.4 Unique Identifier Examples

We giver four examples: roads, rails, pipelines and rivers.

0.4.2.4.1 Road Net Identifiers

Very simple,

28. substitute vertex identifiers, VI, with hub identifiers, HI, and
29. substitute edge identifiers, EI, with link identifiers, LI,

in type and unique observer function definitions.

type
28. HI [≡ VI for Graphs]
29. LI [≡ EI for Graphs]

0.4.2.4.2 Rail Net Identifiers

30. With every rail net unit we associate a unique identifier.
31. That is, no two rail net units have the same unique identifier.

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

0.4.2.4.3 Pipeline Net Identifiers

32. With pipeline units a type WU, PU, LU, VU, FU, JU and SU we associate a single unique identifier sort: UI.

32. UI == WU_UI | PU_UI | LU_UI | VU_UI | FU_UI | JU_UI | SU_UI

0.4.2.4.4 River Net Identifiers

We shall associate unique identifiers both with vertices, edges and vertex and edge elements.

33. River net vertices and edges have unique identifiers.
34. River net sources, confluences and mouths have unique identifiers.
35. River net stretches have unique identifiers.
36. River net channels, lakes, locks, waterfalls, dams and towns as well as combinations of these, that is, simple and composite river entities have unique identifiers.

type
33. V_UI, E_UI
34. SO_UI, KO_UI, MO_UI
35. ST_UI
36. CH_UI, LA_UI, LO_UI, WA_UI, DA_UI, TO_UI, DaLo_UI, WaLo_UI, ToLa_UI, ToLaKo_UI, ...
value
33. uid_V: V→V_UI, uid_E: E→ E_U
34. uid_SO: SO→SO_UI, uid_KO: KO→KO_UI, uid_MO: MO→MO_UI,
35. uid_ST: ST→ST_UI
36. uid_CH: CH→CH_UI, uid_LA: LA→LA_UI, uid_LO: LO→LO_UI, uid_WA: WA→WA_UI,
36. uid_DA: DA→DA_UI, uid_TO: TO→TO_UI,
36. uid_DaLo: DaLo→DaLo_UI, uid_WaLo: WaLo→WaLo_UI, uid_ToLa: ToLa→ToLa_UI,
36. uid_ToLaKo: ToLaKo→ToLaKo_UI, ...

37. All these identifiers are distinct.

The ⋒ operator takes the pairwise intersection of the types in its argument list and examines them for disjointedness.

axiom
37. ⋒(V_UI,E_UI,SO_UI,KO_UI,MO_UI,ST_UI,CH_UI,
37. LA_UI,LO_UI,WA_UI,DA_UI,TO_UI,DaLo_UI,WaLo_UI,ToLa_UI,ToLaKo_UI)

38. There are [many] other constraints, please state them !

38. [left as exercise to the reader !]

0.4.2.5 Mereologies

We refer to [55, Sect. 5.3]. We shall formalise a number of mereologies:

- of undirected graphs — typically road, air and sea transport nets,
- and "general" directed graphs —

0.4.2.5.1 Mereology of Undirected Graphs

39. The mereology of a vertex is the set of unique identifiers of the edges incident upon the vertex.
40. The mereology of an edges is the one-or two element set of the unique identifiers of the [1-loop] vertex, respectively the vertices which the edge is connecting.

```
type
39. V_Mer = E_UI-set
40. E_Mer = V_UI-set
value
39. mereo_V: V → V_Mer
40. mereo_E: E → E_Mer
axiom
39. ∀ g:G,v:V • v ∈ obs_Vs(g) ⇒ mereo_V(v)⊆xtr_E_UIs(g)
40. ∀ g:G,e:E • e ∈ obs_Es(g) ⇒ mereo_E(e)⊆xtr_V_UIs(g)
```

0.4.2.5.2 Wellformedness of Mereologies

41. The vertex mereology must record unique edge identifiers of the graph.
42. The edge mereology must record unique vertex identifiers of the graph.
43. If a vertex mereology identify edges then these edge mereologies must identify that vertex, and, vice versa
44. If an edge mereology identify vertices then these vertex mereologies must identify that edge.

```
axiom
41. ∀ g:G,v:V • v ∈ obs_Vs(g) ⇒ mereo_V(v)⊆xtr_EIs(g)
42. ∀ g:G,e:E • e ∈ obs_Es(g) ⇒ mereo_E(e)⊆xtr_VIs(g)
43. ∀ g:G,v:V • v ∈ obs_Vs(g) ⇒ ∀ ei ∈ mereo_V(v) ⇒ uid_V(v) ∈ mereo_E(e)
44. ∀ g:G,e:E • e ∈ obs_Es(g) ⇒ ∀ vi ∈ mereo_E(e) ⇒ uid_E(e) ∈ mereo_V(v)
```

0.4.2.5.3 Mereology of Directed Graphs

45. The mereology of a vertex is a pair of the set of unique identifiers of the edges incident upon the vertex and the set of unique identifiers of the edges emanating from the vertex –
46. and these must all be of the graph.
47. The mereology of an edge is a one or two element set of pairs of vertex identifiers –
48. and these must all be of the graph.

```
type
45. V_Mer = E_UI-set × E_UI-set
47. E_Mer = (V_UI × V_UI)-set
value
45. mereo_V: V → G → V_Mer
47. mereo_E: E → G → E_Mer
axiom
46. ∀ g:G,v:V • v ∈ obs_Vs(g) ⇒
46.     let (e_ui_s_i,e_ui_s_e) = V_Mer(v) in e_ui_s_i ∪ e_ui_s_e ⊆ xtr_E_UIs(g) end
48. ∀ g:G,e:E • e ∈ obs_Es(g) ⇒
48.     let p_v_ui_s = V_Mer(e) in
```

```
48.    let v_ui_s = { v_ui_i,v_ui_e | (v_ui_i,v_ui_e):(V_UI×V_UI)•(v_ui_i,v_ui_e)∈p_v_ui_s } in
48.    v_ui_s ⊆ xtr_V_UIs(g) end end
```

0.4.2.5.4 In- and Out-Degrees

49. The in-degree of a vertex of a directed graph is the number of edges incident upon that vertex.
50. The out-degree of a vertex of a graph is the number of edges emanating that vertex.

```
49. in_degee_V: V → G ~→ Nat
49. in_degree(v)(vs,es) ≡ let (uis_i,_)=mereo_V(v) in card uis_i end, pre v ∈ vs
50. out_degee_V: V → G ~→ Nat
50. out_degree(v)(vs,es) ≡ let (_,uis_e)=mereo_V(v) in card uis_e end, pre v ∈ vs
```

0.4.2.5.5 Paths of Undirected Graphs

We shall only illustrate *vertex-edge-vertex paths* for given graphs, g.

51. A vertex-edge-vertex path is a sequence of zero or more edges.
52. That is, the empty sequence, $\langle\rangle$, is a vertex-edge-vertex path, [the first basis clause].
53. If e is an edge of g, then the two elements $\langle(vi,ej,vk)\rangle$, $\langle(vk,ej,vi)\rangle$, where ej is the unique identifier of e whose mereology is $\{vi,vj\}$, are vertex-edge-vertex paths.
54. In $\langle(vi,ej,vk)\rangle$ we refer to vi is the first vertex identifier and vk as the second. Vice versa in $\langle(vk,ej,vi)\rangle$.

```
value
54. fVIfEP: EP → VI, fVIfEP(ep:⟨(vi,ej,vk)⟩⌢ep′) ≡ vi, pre: ep≠⟨⟩
54. lVIlEP: EP → VI, lVIlEP(ep:ep′⌢⟨(vi,ej,vk)⟩) ≡ vk, pre: ep≠⟨⟩
```

55. If p and p' are paths of g such that the last vertex identifier of the last element of p is the same as the first vertex identifier of the first element of p', then the sequence p followed by the sequence p' is a vertex-edge-vertex path of g [the inductive clause].
56. Only such paths which can be constructed by the above rules are edge paths [the extremal clause].

```
type
51. EP = E^ω
51. edge_paths: G → EP-set
51. edge_paths(g) ≡
52.     let ps = {⟨⟩}
53.         ∪ {⟨(vi,uid_E(e),vk)⟩,⟨(vk,uid_E(e),vi)⟩|e:E•e ∈ xtr_Es(g)∧{vi,vk}⊆mereo_E(e)}
55.         ∪ {p⌢p′|p,p′:EP•{p,p′}⊆ps∧lVIlEP(9)=fVIfEP(p′)} in
56.     ps end
```

0.4.2.5.6 Paths of Directed Graphs

51. A vertex-edge-vertex path is a sequence of zero or more edges.
52. That is, the empty sequence, $\langle\rangle$, is a vertex-edge-vertex path, [the first basis clause].

57. If e is an edge of g, and if (vi,vj) is in the mereology of e, then the $\langle(vi,ej,vk)\rangle$, where ej is the unique identifier of e is a vertex-edge-vertex path.
55. If p and p' are paths of g such that the last vertex identifier of the last element of p is the same as the first vertex identifier of the first element of p', then the sequence p followed by the sequence p' is a vertex-edge-vertex path of g [the inductive clause].
56. Only such paths which can be constructed by the above rules are edge paths [the extremal clause].

```
type
51. EP = E^ω
51. edge_paths: G → EP-set
51. edge_paths(g) ≡
52.    let ps = {⟨⟩}
57.        ∪ {⟨(vi,uid_E(e),vk)⟩|e:E•e ∈ xtr_Es(g)∧(vi,vk)∈mereo_E(e)}
55.        ∪ {p⌢p′|p,p′:EP•{p,p′}⊆ps∧lVlIEP(9)=fVIfEP(p′)} in
56.    ps end
```

Notice that the difference in the two definitions of (overload-named) edge_paths differ only in in the last terms of items 53 and 57.

0.4.2.5.7 Connectivity

58. For every pair of vertices we can calculate the set of all paths connecting these in a graph.

```
58. all_connected_paths: (V×V) → G → EP-set
58. all_connected_paths(vi,vj) ≡
58.    { ep | ep:EP • ep ∈ edge_paths(g) • ep[1] = (uid_V(vi),_,_), ep[len ep] = (_,_,uid_V(vj)) }
```

59. Two vertices, v_i,v_j, of a graph, g, *are_connected* if there is a path from v_i to v_j in g.

```
value
59. are_connected: (V×V) → G ~→ Bool
59. are_connected(vi,vj)(g) ≡ all_connected_paths(vi,vj) ≠ {}
```

60. A graph *is_connected* if there is a path from every vertex to every other vertex.

```
value
60. is_connected: G → Bool
60. is_connected(g) ≡ ∀ vi,vj:V • {vi,vj} ∈ obs_Vs(g) • are_connected(vi,vj)(g)
```

0.4.2.5.8 Acyclic Graphs, Trees and Forests

61. A cycle is a path which begins and ends at the same vertex.
62. An acyclic graph is a graph having no graph cycles.
63. A bipartite graph (or bi-graph) is a graph whose vertices can be divided into two disjoint and independent sets, V',V'', such that every edge connects a vertex in V' to one in V''. Acyclic graphs are bipartite.
64. By a tree we[5] shall understand a connected, acyclic graph such that there are no two distinct paths from any given pair of in-degree-0 and out-degree-0 vertices.

[5] Our definition is OK, but there are more encompassing definitions of trees.

65. A disjoint graph is a set of two or more graphs such that no two of these graphs, G, g′, have vertices in g with edges to g′.
66. A forest is a disconnected set of trees, hence form a disjoint graph of distinct trees.

62. is_a_cycle: EP → **Bool**
62. is_a_cycle(ep) ≡ **let** (vi,_,_)=ep[1], (_,_,vi′)=ep[**len** ep] **in** vi = vi′ **end**
62. is_acyclic: G → **Bool**
62. is_acyclic(g) ≡ !∃ ep:EP • ep ∈ edge_paths(g) ∧ is_a_cycle(ep)
63. is_bipartite: G → **Bool**
63. is_bipartite(g) ≡ ... [exercise for the reader]
64. is_a_tree: G → **Bool**
64. is_a_tree(g) ≡ ... [exercise for the reader]
65. is_disjoint_graph: G → **Bool**
65. is_disjoint_graph(g) ≡ ... [exercise for the reader]
66. is_a_forest: G → **Bool**
66. is_a_forest(g) ≡ ... [exercise for the reader]

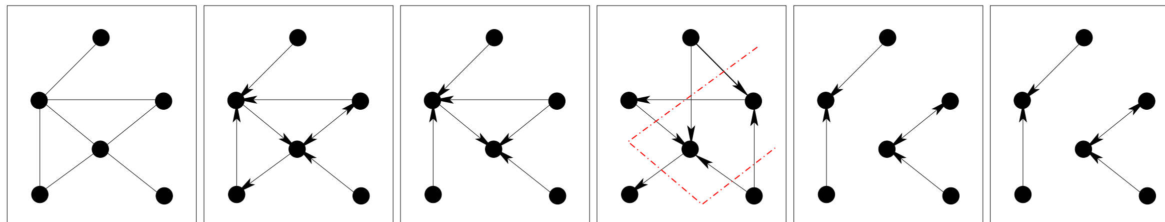


Fig. 0.21 Undirected, Directed, Acyclic, Bipartite, Tree and Disjoint Graphs

0.4.2.5.9 Forest

A forest is an undirected graph without cycles (a disjoint union of un-rooted trees), or a directed graph formed as a disjoint union of rooted trees.

0.4.2.5.10 Mereology Examples

We present mereology examples of both undirected and directed graphs.

Mereology of Undirected Graph Examples: We present mereology examples of road nets and railway tracks.

- **Road Nets**

  The mereology of road nets follow that of undirected graphs:

67. substitute V for H and VI for HI, and
68. substitute E for L and EI for LI.

We refer to Sect. **0.4.2.5.10** on the facing page.

67. H_Mer = L_UI-**set** × L_UI-**set**
68. L_Mer = (H_UI × H_UI)-**set**, **axiom** ∀ lm:L_Mer•**card**lm∈{0,1,2}

- **Rail Nets**

  We refer to Chapter 1.

Mereology of directed Graph Examples: In some circumstances we may model mereologies of directed graphs in terms of attributes. An example is that of road nets. Road nets, usually, can be considered undirected graphs. But discrete dynamically set and reset traffic signals as well as road signs may render streets and their intersection, i.e., links and hubs, "directed". We then model this "directedness", as we shall see, in Sect. **0.4.2.6.3** on page 31, in terms of *programmable attributes*.

- **Pipeline Nets**

  We refer to Sects. **0.2.1.3** on page 6, **0.4.1.2.3** on page 18 and **0.4.2.4.3** on page 22.

69. Wells have exactly one connection to an output unit – which is usually a pump.
70. Pipes, pumps, valves and re-directors have exactly one connection from an input unit and one connection to an output unit.
71. Forks have exactly one connection from an input unit and exactly two connections to distinct output units.
72. Joins have exactly two connections from distinct input units and one connection to an output unit.
73. Sinks have exactly one connection from an input unit – which is usually a valve.
74. Thus we model the mereology of a pipeline unit as a pair of disjoint sets of unique pipeline unit identifiers.

```
type
74  PM′=(UI-set×UI-set), PM={|(iuis,ouis):PM′•iuis ∩ ouis={}|}
value
74  mereo_PE: PE → PM
```

The well-formedness inherent in narrative lines 69–73 are formalised:

```
axiom [ Well−formedness of Pipeline Systems, PL (0) ]
    ∀ pl:PL,pe:PE • pe ∈ all_pipeline_uits(pl) ⇒
            let (iuis,ouis)=mereo_PE(pe) in
            case (card iuis,card ouis) of
69              (0,1) → is_We(pe),
70              (1,1) → is_Pi(pe)∨is_Pu(pe)∨is_Va(pe),
71              (1,2) → is_Fo(pe),
72              (2,1) → is_Jo(pe),
73              (1,0) → is_Si(pe), _ → false
            end end
```

To express full well-formedness we need express that pipeline nets are acyclic. To do so we first define a function which calculates all routes in a net.

Two pipeline units, pe_i with unique identifier π_i, and pe_j with unique identifier π_j, that are connected, such that an outlet marked π_j of p_i "feeds into" inlet marked π_i of p_j, are said to **share** the connection (modeled by, e.g., $\{(\pi_i,\pi_j)\}$)

75. The observed pipeline units of a pipeline system define a number of routes (or pipelines):
    Basis Clauses:
76. The null sequence, $\langle\rangle$, of no units is a route.

77. Any one pipeline unit, *pe*, of a pipeline system forms a route, $\langle pe \rangle$, of length one.
Inductive Clauses:
78. Let $r_i\widehat{\ }\langle pe_i\rangle$ and $\langle pe_j\rangle\widehat{\ }r_j$ be two routes of a pipeline system.
79. Let $pe_{i_{ui}}$ and $pe_{j_{ui}}$ be the unique identifiers pe_i, respectively pe_j.
80. If one of the output connectors of pe_i is $pe_{i_{ui}}$
81. and one of the input connectors of pe_j is $pe_{j_{ui}}$,
82. then $r_i\widehat{\ }\langle pe_i,pe_j\rangle\widehat{\ }r_j$ is a route of the pipeline system.
Extremal Clause:
83. Only such routes which can be formed by a finite number of applications of the clauses form a route.

```
type
75.  R = PE^ω
value
75   routes: PL ~→ R-infset
75   routes(ps) ≡
75     let cpes = pipeline_units(pl) in
76     let rs = {⟨⟩}
77            ∪ {⟨pe⟩|pe:PE•pe ∈ cpes} ∪
82            ∪ {ri⁀⟨pe_i⟩⁀⟨pe_j⟩⁀rj | pei,pej:PE • {pe_i,pe_j}⊆cpes
78                 ∧ ri⁀⟨pe_i⟩,⟨pe_j⟩⁀rj:R • {ri⁀⟨pe_i⟩,⟨pe_j⟩⁀rj}⊆rs
79,80               ∧ pe_i_ui = uid_PE(pe_i) ∧ pe_i_ui ∈ xtr_oUOs(pe_i)
79,81               ∧ pe_j_ui = uid_PE(pe_j) ∧ pe_j_ui ∈ xtr_iUIs(pe_j)} in
83     rs end end

xtr_iUIs: PE → UI-set, xtr_iUIs(u) ≡ let (iuis,_)=mereo_PE(pe) in iuis end
xtr_oUIs: PE → UI-set, xtr_oUIs(u) ≡ let (_,ouis)=mereo_PE(pe) in ouis end
```

84. The observed pipeline units of a pipeline system forms a net subject to the following constraints:
   a. unit output connectors, if any, are connected to unit input connectors;
   b. unit input connectors, if any, are connected to unit output connectors;
   c. there are no cyclic routes;
   d. nets has all their connectors connected, that is, “starts” with wells
   e. and “ends” with sinks.

```
value
84.  wf_Net: PL → Bool
84.  wf_Net(pl) ≡
84.    let cpes = all_pipeline_units{pl}  in
84.    ∀ pe:PE • pe ∈ cpes ⇒ let (iuis,ouis) = mereo_PE(pe) in
84.      axiom 69.–73.
84a.     ∧ ∀ pe_:UI•pe_ui ∈ iuis ⇒
84a.        ∃ pe′:PE•pe′≠pe∧pe′isin cpes∧uid_PE(pe′)=pe_ui∧pe_ui∈xtr_iUIs(pe′)
84b.     ∧ ∀ pe_ui:UI•pe_ui ∈ ouis ⇒
84b.        ∃ pe′:PE•pe′≠pe∧pe′isin cpes∧uid_PE(pe′)=pe_ui∧pe_ui∈xtr_oUIs(pe′)
84c.     ∧ ∀ r:R•r ∈ routes(pl) ⇒
84c.        ~∃ i,j:Nat•i≠j∧{i,j}∈ inds r∧r(i)=r(j)
84d.     ∧ ∃ we:We • we ∈ us ∧ r(1) = mkWe(we)
84e.     ∧ ∃ si:Si • si ∈ us ∧ r(len r) = mkSi(si)
75.    end end
```

- **River Nets**

85. The mereology of a river vertex is a pair: a set of unique identifiers, E_UI, of river edges, i.e., stretches, linear sequences of simple and composite river elements, incident upon the vertex, and a set of unique identifiers, E_UI, of river edges emanating from the vertex. If the vertex is a source then the first element of this pair is empty. If the vertex is a mount then the second element of this pair is empty. For a confluence vertex both elements of the pair are non-empty.
86. The mereology of a river edge, that is, the linear sequence of simple and composite river elements between two adjacent vertices, is a pair: the first element is a unique identifier of a river vertex and so is the second element of the pair.

We present the river net mereology in two forms. The first was with respect to its graph rendition. The second is with respect to its river element rendition.

87. The mereology of a source is just the single unique identifier of the first simple or composite river element of the stretch emanating from the source.
88. The mereology of a confluence is a triplet: the single unique identifier of the last simple or composite river element of the stretch of the main river incident upon the source, a set of unique identifier of the last simple or composite river element of the stretches of the tributary rivers incident upon the source, and the single unique identifier of the first simple or composite river element of the main river stretch emanating from the confluence.
89. The mereology of a mouth is just the single unique identifier of the last simple or composite river element of the stretch incident upon the mouth
90. The mereologies of simple and composite river elements are pairs: of the unique identifier of the river elements, including sources and confluences, upstream adjacent to the river element being "mereologised", and of the unique identifier of the river elements, including confluences and mouths, downstream adjacent to the river element being "mereologised".

85. Mer_V = E_UI-**set** × E_UI-**set**
86. Mer_E = V_UI × V_UI
87. Mer_SO = SE_UI | CE_UI
88. Mer_KO = (SE_UI|CE_UI) × (SE_UI|CE_UI)-**set** × (SE_UI|CE_UI)
89. Mer_MO = SE_UI | CE_UI
90. Mer_RE = (SO_UI|CO_UI|SE_UI|CE_UI) × (SE_UI|CE_UI|CO_UI|MO_UI)

91. The unique vertex and edge identifiers must be identifiers of the vertices and edges of a graph.
92. Similarly, the unique source, confluence and mouth identifiers must be identifiers of respective sources, confluences and mouths of a graph.
93. And likewise for simple and composite element identifiers.
94. No two sources, confluences, mouths, simple and composite elements have identical unique identifiers.
95. There are other constraints, please state them !

axiom

91. [left as exercise to the reader !]
92. [left as exercise to the reader !]
93. [left as exercise to the reader !]
94. [left as exercise to the reader !]
95. [left as exercise to the reader !]

0.4.2.6 Attributes

We refer to [55, Sect. 5.4]

Attributes of discrete endurants ascribe to them such properties that endow these, typical manifest entities with substance. External qualities of **endurants** allow us to reason about atomicity and compositions, whether as Cartesian-like products, as sets or as sequences; but not much more ! The *internal quality* of **unique identification** allows us to speak of, i.e., analyse and describe multiplicities of same sort endurants. The *internal quality* of **mereology** allows us to relate discrete endurants either topologically or otherwise. But it is the *internal quality* of possessing one or more **attributes**, i.e., properties — usually many more than we may actually care to define, that really sets different sort parts "apart" (!) and allows us to reason more broadly, more domain-specifically, about endurants.

0.4.2.6.1 Graph Labeling

It is quite common, in fact usually normal, to so-called "label" vertices and edges of graphs; that is, either none, or all, rarely only some proper subset. Such labeling is used for two distinct purposes: Either such labeling occur in only one of these forms, sometimes, though, in both; the situation is confusing. In our approach we clearly analyse labeling into two separate forms: the unique identification of distinct parts, and the ascription of attributes, sometimes the same, or "overlapping"[6] to more than one part, or even part sort. One should take care of the following: whereas distinct parts "receive" distinct unique identification, such distinct parts may be ascribed the same **attribute value.**

One may classify attributes in two different ways: into either *static, monitorable* and *programmable* as introduced by M.A. Jackson, [107], and as slightly "simplified" in [55, Sect. 5.4.2.3]; or as either measurable (by for example electro-, chemical or mechanical instruments) or referable (one can talk about histories of events) or both ! Anyone part may be ascribed attributes of any mix and composition of these classifications.

If you sense some uneasiness about the issue of graph labeling as it is treated in for example operations, where graphs are a stable work horse, then you are right !

0.4.2.6.2 General Net Attributes

Let us informally recall some general facts about the concept of attributes such as we introduce them in [55, Sect. 5.4].

96. We can speak of the set of **names** of attribute types. If A is the type of an attributes, the ηA is the name of that type.
97. For every part sort, P, we can thus speak of the set, or a suitably chosen, to be modeled, subset of attributes types in terms of their names.
98. Of course, different attribute names must designate distinct, i.e., non-overlapping attribute values of that type.

Likewise informally:

type
96. ηA, A
value
96. name_of_attribute: A → ηA,ηAname_of_attribute(A) ≡ ηA

[6] By "overlapping" assignment of attribute to different parts we mean that two or more parts may be assigned the same **attribute type.**

97. attributes: P → ηA-**set**
97. attributes(p) ≡ {ηA_i,ηA_j,...,ηA_k}
axiom
98. ∀ p:P, anms:{ηA_i,ηA_j,...,ηA_k}:ηA-**set**:anms⊆attributes(p)⇒∀i,j,...,k•$A_i \cap A_j$={}∧$A_j \cap A_j$={}∧...

0.4.2.6.3 Road Net Attributes

Link Attributes:

99. Standard "bookkeeping" link attributes are *road name, length,* name of *administrative authority* and others. These are static attributes.
100. Standard "control" attributes model the dynamically settable direction of flow along a link: its current link state as well as the space of all such link states.
101. Standard "event history" attributes model the time-stamped chronologically ordered sequence, for example latest first, of automobiles entering, stopping along (say parking) and leaving a link. A first element in such a list denotes "entering". The last element "leaving". Any element in-between, pairwise, "stopping" (for example for parking) and "starting" (resume driving).

Hub Attributes:

102. Standard "bookkeeping" hub attributes are *road intersection name,* name of *administrative authority* and others. These are static attributes.
103. Standard "control" attributes model the dynamically settable direction of flow along into and out of a hubs: its current hub state as well as the space of all such hub states.
104. Standard "event history" attributes model the time-stamped chronologically ordered sequence, for example latest first, of automobiles entering, stopping along (say parking) and leaving a hub. A first element in such a list denotes "entering". The last element "leaving". Any element in-between, pairwise, "stopping" (for example for parking) and "starting" (resume driving).
105. We assume a sort of automobile identifiers.

type
99. Road_Name, Length, Admin_Auth, ...
100. LΣ = (H_UI×H_UI)-**set**; **axiom** ∀ lσ:LΣ•**card** lσ ∈ {0,1,2}; LΩ = LΣ-**set**
101. L_History = A_UI $\overrightarrow{m}$ $\mathbb{TIME}^*$
102. Intersection_Name, Admin_Auth, ...
103. HΣ = (L_UI×L_UI)-**set**
103. HΩ = HΣ-**set**
104. H_History = A_UI $\overrightarrow{m}$ $\mathbb{TIME}^*$
105. A_UI
value
99. attr_Road_Name:: L→Road_Name, attr_Length: L→Length, attr_Admin_Auth: L→Admin_Auth, ...
100. attr_LΣ: L→LΣ, attr_LΩ: L→LΩ
101. attr_L_History: L→L_History
102. attr_Intersection_name:: H→Intersection_name, attr_Admin_Auth: H→Admin_Auth, ...
103. attr_HΣ: H→HΣ, attr_HΩ: H→HΩ
104. attr_H_History: H→H_History

We omit narrating and formalising attributes for `Road_Surface_Temperature`, `Road_Maintenance_Condition`, etc., etc.

Elucidation of Road Net History Attributes

The above was a terse rendition. Below we elucidate, in two steps.

- **All Events are Historized !**

The above "story" on road net history attributes was a "lead-in"! To get you started on the notion of event histories. They are not recorded by anyone. They do occur. That is a fact. We can talk about them. So they are attributes. But they occur without our consciously talking about them. So they are chronicled.

- **More Detailed Road Unit Histories**

  Also, the "story" was simplified. Here is a slightly more detailed history rendition of:

106. Attributed vents related to automobiles on roads.
107. Automobiles enter a link.
108. Automobiles stop along the link at a
109. fraction of the distance between the entered and the intended destination hubs.
110. Automobiles Restart.
111. Automobiles may make U-turns along a link at fraction of the distance between the entered and the originally intended destination hubs.
112. Eventually automobiles leave a link, entering a hub.
113. Same story for automobiles at a hub.

type
106. L_Hist = A_UI $\overrightarrow{m}$ (A_L_Event × $\mathbb{TIME}$)*
106. A_L_Event == Enter | Stop | ReStart | U_Turn | Leave
107. Enter :: H_UI
108. Stop :: H_UI × Frac × H_UI
109. Frac = **Real**; **axiom** ∀ f:Frac • 0<f<1
110. ReStart :: ...
111. U_Turn :: H_UI × Frac × H_UI
112. Leave :: H_UI

- **Requirements: Recording Events**

  So all events are chronicled. Not by the intervention of any device, but by "the sheer force of fate"! So be it — in the domain. But if you are to develop software for a road net application: be it a road pricing system, or a traffic control system, or other – something related to automobile and road events, then recording these events may be necessary. If so, you have to develop requirements from, for example, a domain description of this kind. We refer to [55, *Chapter 9:* Requirements]. More specifically you have to extend the domain, [55, *Sect. 9.4.4:* Domain Extension] – sensors that record the position of cars[7]. And this sensing may fail, and thus and implementation of the recording of hub and link histories may leave "holes" – and the requirements must then prescribe which kind of safeguards the thus extended road net system must provide.

0.4.2.7 Summing Up

0.4.2.7.1 A Summary of The Example Endurant Models

We summarise "the tip of the icebergs" by recording here the main domains, but now in a concrete form; that is, with concrete types for main sorts instead of abstract types with observers.

River nets form graphs. Similarly can be done for all the examples. First we recall graphs.

- **Graphs:** See Items. 1 on page 16, 2 on page 16, 3 on page 16, 19 on page 19, 20 on page 19, 39 on page 23, 45 on page 23, 40 on page 23 and 47 on page 23.

[7] These sensors may be photo-electric or electronic and placed at suitable points along the road net, or they may be satellite borne. To work properly we assume that automobiles emit such signals that let their identity be recorded.

type [Endurants]
1. G = V-**set** × E-**set**
2. V
3. E
type [Unique Identifiers]
19. V_UI
20. E_UI
type [Mereology]
39. V_Mer = E_UI-**set**; 45. V_Mer = E_UI-**set** × E_UI-**set** [Un−directed; Directed Graphs]
40. E_Mer = V_UI-**set**; 47. E_Mer = (V_UI × V_UI)-**set** [Un−directed; Directed Graphs]

- **Roads:** See Items 8 on page 17, 5 on page 17, 7 on page 17, 67 on page 26 and 68 on page 26.

type [Endurants]
5. RN = H-**set** × L-**set** [≃ G for Graphs]
6. H [≃ V for Graphs]
7. L [≃ E for Graphs]
type [Unique Identifiers]
28. HI [≡ VI for Graphs]
29. LI [≡ EI for Graphs]
type [Mereology]
67. H_Mer = LI-**set** × LI-**set** [≃ V_Mer for Graphs]
68. L_Mer = (HI × HI)-**set**, **axiom** ∀ lm:L_Mer•**card**lm∈{0,1,2} [≡ E_Mer for Graphs]

- **Rails:** See Chapter 1.
- **Pipelines:** See Items 10 on page 18, 11 on page 18, 32 on page 22 and 74 on page 27.

type [Endurants]
10. PN = PLU-**set** [≃ G for Graphs]
11. PLU == WU | PU | LU | VU | FU | JU | SU [≃ (V|E) for Graphs]
type [Unique Identifiers]
32. UI == WU_UI | PU_UI | LU_UI | VU_UI | FU_UI | JU_UI | SU_UI
type [Mereology]
74 PM′=(UI-**set**×UI-**set**), PM={|(iuis,ouis):PM′•iuis ∩ ouis={}|} [≃ V_Mer∪E_Mer for Graphs]

- **Rivers:** See Items 13 on page 18, 14 on page 18, 15 on page 18, Sect. **0.4.2.4.4** on page 22, 85 on page 29 and 86 on page 29.

type [Endurants]
13. RiN
14. V
15. E
type [Unique Identifiers]
33. V_UI, E_UI
type [Mereology]
85. Mer_V = E_UI-**set** × E_UI-**set**
86. Mer_E = V_UI × V_UI

0.4.2.7.2 Initial Conclusion on Labeled Graphs and Example Domains

We have shown basic models of abstract undirected and directed graphs. And we have shown four examples:

- road nets,
- rail nets,
- pipeline nets and
- river nets.

Road, rail, pipeline and river elements are all uniquely identified. The road and river nets were basically modeled as as graphs with vertices (hubs, respectively sources, confluences and mouths) and edges (links, respectively stretches of simple and composite river elements). The rail and pipeline nets we modeled as sets of rail and pipeline units with the mereology implying edges.

Labels, such as they are "practiced" in conventional graph theory, are introduced by may of attributes. Attributes were also used to model dynamically varying "directedness" of edges.

We can conclude the following

- There is now a firm foundation for the labeling of graphs:
  - *the origin of vertex and edge labeling is*
    - *the unique identifiers and/or*
    - *the attributes*

    of the endurant parts that vertices and edges designate; and
  - *there really can be no vertex or edge labeling unless the origin is motivated in*
    - *the unique identification and/or*
    - *the attribution*

    of the vertex and edge parts.

0.5 The Nets Domain

0.5.1 Some Introductory Definitions

Definition: By a **net domain**, or, for short, just a **net**, we shall understand a domain of the kind illustrated in Sect. **0.4**, that is, a domain the mereology of whose main parts model graphs ■

Definition: By a **dynamic net domain**, or, for short, just a **dynamic net**, we shall understand a *net* whose *mereology* – or a corresponding attribute notion – may change ■

Definition: By a **nets domain**, or, for short, just **nets**, (notice the suffix 's', we shall understand a domain each of whose instances is a *dynamic net domain* ■

MORE TO COME

Part II
Main Examples

Chapter 1

Rail Systems [1993–2007, 2020]

Contents

1.1 Endurants – Rail Nets and Trains 38
1.1.1 External Qualities 38
1.1.1.1 Rail Nets 38
1.1.1.1.1 The Endurants 38
1.1.1.1.2 All Net Units 39
1.1.1.2 Trains 39
1.1.1.2.1 The Endurants 39
1.1.1.2.2 All Trains 39
1.1.2 Internal Qualities 40
1.1.2.1 Unique Identifiers 40
1.1.2.1.1 All Net Unit Unique Identifiers 40
1.1.2.1.2 Trains 40
1.1.2.1.3 Retrieve Net Units 41
1.1.2.2 Mereology 41
1.1.2.2.1 Rail Units 41
1.1.2.2.2 Well-formed Mereologies 42
1.1.2.2.3 Trains 42
1.1.2.2.4 Routes 42
1.1.2.2.4.1 Route Types 42
1.1.2.2.4.2 Initial Routes 43
1.1.2.2.4.3 Next Route Elements 43
1.1.2.2.4.4 Previous Route Elements 43
1.1.2.2.4.5 All Routes 44
1.1.2.2.4.6 Isolated Rail Net Units 44
1.1.2.2.4.7 A Delineation: Train Stations 45
1.1.2.2.4.8 All Stations of a Railway System 45
1.1.2.2.4.9 Rail Lines 46
1.1.2.3 Attributes 47
1.1.2.3.1 Rail Nets 47
1.1.2.3.2 Open Routes 48
1.1.2.3.3 Station Names 48
1.1.2.3.4 Trains 49
1.1.2.3.5 An Intentional Pull 49
1.1.2.3.6 History Attributes 50
1.1.2.3.7 The Intentional Pull Revisited 50
1.2 Transcendental Deduction 51
1.2.1 General 51
1.2.2 A Note on $\mathbb{TIME}$ 51
1.2.3 Train Traffic 51
1.2.3.1 Well-formed Train Traffics 52
1.3 Perdurants 53
1.3.1 Channels 53
1.3.2 Behaviour Signatures 54
1.3.3 Behaviour Definitions 54

1.3.3.1 Rail Unit Behaviours 54
1.3.3.2 Train Behaviour 55
1.4 Closing 55

This model evolved over many years. A first, beautiful model was developed in 1993 by the late Søren Prehn[8]. Over the years variations of this model went into several papers [68, 69, 19, 61, 21, 22, 23, 130, 24, 151, 131]. We refer to *Railways* – a compendium `imm.dtu.dk/~dibj/train-book.pdf`. The current model is a complete rewrite of earlier models. These earlier models were not based on the endurant/perdurant, the atomic/compound [set and composite] externalities and the unique identifier, mereology and attribute paradigms. The present model is.

The example is quite extensive. Anything smaller really makes no sense: does not bring across the issues of what it takes to describe a domain nor the scale of domain descriptions.

The example is that of a railway system's net of rail units and trains.

1.1 Endurants – Rail Nets and Trains

1.1.1 External Qualities

1.1.1.1 Rail Nets

1.1.1.1.1 The Endurants

114. The example is that of a railway system.
115. We focus on the railway net [and, later, trains]. They can be observed from the railway system.
116. The railway net embodies a set of [railway] net units.
117. A net unit is either a straight or curved **linear** unit, or a simple switch, i.e., a **turnout**, unit[9] 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.

We refer to Figure 1.1 on the next page.

type
114. RS
115. RN
value
115. obs_RN: RS → RN
type
116. NUs = NU-**set**
116. NU = LU | PU | RU | SU | DU | TU
value
117. obs_NUs: RN → NU-**set**

[8] Søren Prehn was a brilliant student of mine 1975–1980. He became a leading member of Dansk Datamatik Center [62], and later the CR company in Denmark, from 1980 onward. He spent a 2 year sabbatical from CR with me at the UNU/IIST, the United Nations International Institute for Software Technology in Macau, 1992–1994. Sadly he passed away in the spring of 2006.

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

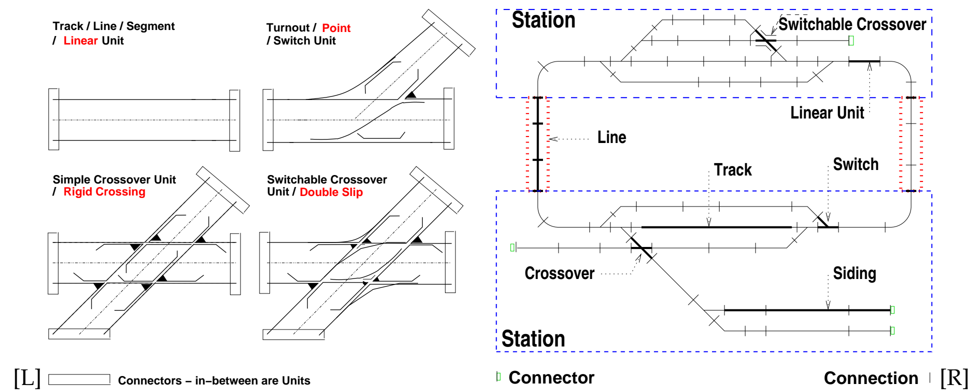


Fig. 1.1 Left: Four net units; Right: A railway net

1.1.1.1.2 All Net Units

118. From a railway system net one can observe, i.e., extract, all the rail net units.
119. We let *rs* denote the value of of an arbitrary chosen railway system,
120. and we let *nus* denote the value of the set of all railway units.

value
118. xtr_NUs: RS → NU-**set**
118. xtr_NUs(rs) ≡ obs_NUs(obs_RN(rs))

119. *rs*:RS
120. *nus* = obs_NUs(*rs*)

1.1.1.2 Trains

1.1.1.2.1 The Endurants

121. We shall, simplifying, consider trains as atomic parts.
122. From a railway system one can observe a finite, let us decide, non-empty set of trains.

type
121. Train
122. TS = Train-**set**
value
122. obs_TS: RS → TS
axiom
122. ∀ rs:RS • obs_TS(rs)≠{}

1.1.1.2.2 All Trains

123. We let *trains* denote the value of the set of all trains.

value
123. *trains* = { t | t:Train • obs_TS(*rs*) }

1.1.2 Internal Qualities

1.1.2.1 Unique Identifiers

Rail Units

124. With every rail net unit we associate a unique identifier.
125. That is, no two rail net units have the same unique identifier.

```
type
124.  UI
value
124.  uid_NU: NU → UI
axiom
125.  ∀ ui_i,ui_j:UI • ui_i = ui_j ≡ uid_NU(ui_i)=uid_NU(ui_j)
```

1.1.2.1.1 All Net Unit Unique Identifiers

126. From a railway system net one can observe, i.e., extract, the set of all the unique rail unit identifiers of all the rail net units.
127. We let *uis* denote the set of all railway units of the arbitrarily chosen railway system cum railway net.

```
value
126. xtr_UIs: RS → UI-set
126. xtr_UIs(rs) ≡ { uid_NU(nu) | nu:NU • nu ∈ obs_NUs(obs_RN(rs)) }
127. uis = xtr_UIs(rs)
```

1.1.2.1.2 Trains

128. Trains have unique identifiers.
129. We let *tris* denote the set of all train identifiers.
130. No two distinct trains have the same unique identifier.
131. Train identifiers are distinct from rail net unit identifiers.

```
type
128.  TI
value
128.  uid_Train: Train → TI
129.  tris = { uid_Train(t) | t:Train • t ∈ trains }
axiom
130.  either: card trains = card tris
130.  or: ∀ rs:RS •
130.        ∀ train_a,train_b:Train • {train_a,train_b}⊆obs_TS(rs) ⇒
130.        train_a≠train_b ⇒ uid_Train(train_a)≠uid_Train(train_b)
131.  uis ∩ tris = {}
```

1.1.2.1.3 Retrieve Net Units

132. Given a net unit unique identifier and a railway net one can retrieve the net unit with that identifier.

```
value
132. retr_NU: UI → RS →~ NU
132. retr_NU(ui)(rs) ≡ let nu:NU • nu ∈ xtr_NUs(rs) ∧ uid_NU(nu)=ui in nu end
132.    pre: ui ∈ xtr_UIs(rs)
```

1.1.2.2 Mereology

1.1.2.2.1 Rail Units

The mereology of a rail net unit expresses its topological relation to other rail net units and trains.

133. Every rail unit is conceptually related to every train.
134. A linear rail unit is connected to exactly two distinct other rail net units of any given rail net.
135. A point unit is connected to exactly three distinct other rail net units of any given rail net.
136. A rigid crossing unit is connected to exactly four distinct other rail net units of any given rail net.
137. A single and a double slip unit is connected to exactly four distinct other rail net units of any given rail net.
138. A terminal unit is connected to exactly one distinct other rail net unit of any given rail net.
139. 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.
140. Trains can run on every rail unit of any rail system.

Fig. 1.2 Four Symmetric Mereologies

```
type
139. Unit_Mereo = (UI-set×UI-set) × TI-set
value
139. mereo_NU: NU → Unit_Mereo
axiom
139. ∀ nu:NU •
139.   let ((uis_i,uis_o),tris)=mereo_NU(nu) in
133.   tris = tris ∧
139.   case (card uis_i,card usi_o) =
134.      (is_LU(nu) → (1,1),
135.       is_PU(nu) → (1,2) ∨ (2,1),
```

```
136.        is_RU(nu)  → (2,2),
137.        is_SU(nu) → (2,2), is_DU(nu) → (2,2),
138.        is_TU(nu)  → (1,0) ∨ (0,1),
139.        _ → chaos) end
139.   ∧ uis_i∩uis_o={}
139.   ∧ uid_NU(nu) ∉ (uis_i ∪ uis_o)
139.    end
```

1.1.2.2.2 Well-formed Mereologies

141. The unique identifiers of any rail unit mereology of a rail net must be of rail units of that net and
142. the set of train identifiers of any rail unit mereology of a rail net must be the set of all train identifiers of that railway system.

```
value
141.  wf_Mereology: RS → Bool
141.  wf_Mereology(rs) ≡
141.    let (nus,uis) = (xtr_NUs,xtr_UIs)(rs) in
141.    ∀ nu:NU • nu ∈ nus •
141.       let ui = uid_NU(nu), ((iuis,ouis),tris) = mereo_NU(nu) in
141.       ui ∉ iuis ∪ ouis ∧ iuis ∩ ouis ={}∧ iuis ∪ ouis ⊆ uis
142.       ∧ tris = tris
141.    end end
```

1.1.2.2.3 Trains

143. Trains can run on every rail unit of any rail system.

We omit consideration of trains communicating with other trains as well as with net management. We leave such “completions” to the reader.

```
type
143.  Train_Mereo = UI-set
value
143.  mereo_Train_Mereo: Train → Train_Mereo
axiom
143.  ∀ rs:RS • ∀ train:Train
143.     ∀ train:Train • train ∈ obs_TS(rs) ⇒ mereo_Train_Mereo(train)=retr_UIs(rs)
```

1.1.2.2.4 Routes

We decompose the analysis into several preparatory steps.

1.1.2.2.4.1 Route Types

144. A route is a finite or infinite sequence of one or more route elements.

145. A route element is a [route] triple of three distinct net unit identifiers, the net unit identifier of an immediately preceding rail unit, the net unit identifier of the present rail unit, the net unit identifier of an immediately succeeding rail unit, irrespective of whether the preceding and succeeding units are actually in the route as analysed.

```
type
144.  R   = TUI^ω
145.  TUI = UI×UI×UI
axiom
145.  ∀ (pui,ui,sui):TUI • card{pui,ui,sui}=3
144.  ∀ r:R • ∀ i:Nat • {i,i+1}⊆inds r ⇒
144.     let (pui,ui,sui)=r[i], (pui',ui',sui')=r[i+1] in
144.     sui = pui' ∧ ui≠ui' ∧ pui ≠ ... end
```

1.1.2.2.4.2 Initial Routes

146. We define an auxiliary function which, for any given railway system, calculates the finite set of all its initial routes – where an initial route is a one element route triplet of a non-terminal net unit.

```
value
146.  initial_routes: RS → R-set
146.  initial_routes(rs) ≡
146.    let (nus,uis) = (retr_NUs,retr_UIs)(rs) in
146.    { ⟨(pui,ui,sui)⟩, ⟨(sui,ui,pui)⟩
146.     | nu:NU • nu ∈ nus ∧ ∼is_TU(nu) ∧
146.       let (ui,(puis,suis)) = (uid_NU,mereo_NU)(nu) in
146.       pui ∈ puis ∧ sui ∈ suis end }
146.    assert: [ there are up to eight triplets in the above set ]
146.    end
```

1.1.2.2.4.3 Next Route Elements

147. Give a route element, i.e., a triplet (pui,ui,sui), one can calculate the set of one or two next route triplet designating the net unit with identifier sui.

```
value
147.  next_route_elements: TUI → RS → R-set
147.  next_route_elements(_,ui,sui)(rs) ≡
147.    let (puis ∪ {ui},suis) = mereo_NU(retr_NU(sui)(rs)) in
147.    { ⟨(pui,uid_NU(retr_NU(sui)(rs)),sui')⟩ | pui:UI•pui∈puis∧sui'∈suis }
147.    assert: [ there are either one or two triplets in the set above. ]
147.    end
```

1.1.2.2.4.4 Previous Route Elements

148. Give a route element, i.e., a triplet (pui,ui,sui), one can calculate the set of one or two previous route triplet designating the net unit with identifier sui.

```
value
148.  previous_route_elements: TUI → RS → R-set
148.  previous_route_elements(pui,ui,_)(rs) ≡
148.     let (puis,suis ∪ {ui}) = mereo_NU(retr_NU(pui)(rs)) in
148.     { ⟨(pui',uid_NU(retr_NU(pui)(rs)),pui)⟩ | pui' ∈ puis ∪ suis }
148.     assert: [ there are either one or two triplets in the set above ]
148.     end
```

1.1.2.2.4.5 All Routes

149. A route is a finite or infinite sequence of triplets.
150. The analysis function routes calculates a potentially infinite set of routes.
151. The set rs is recursively defined.
     It is the smallest set, i.e., fix-point, satisfying the equation.
     rs is initialised, i.e., the base step, with the set of initial routes of the railway system.
152. The induction step (152–155) "adds"
153. next, nr, and
154. previous, pr, triplets
155. to an arbitrarily selected route (so far calculated).
156. The pr⁀udr⁀nr element of formula line 152 need not be included as it will be calculated in some subsequent recursion.

```
value
150.  routes: RS → R-infset
150.  routes(rs) ≡
151.     let all_routes = irs ∪
152.              { udr⁀nr, pr⁀udr, pr⁀udr⁀nr
153.              | nr ∈ next_route_elements(udr[len udr])(rs) ∧
154.                pr ∈ previous_route_elements(udr[1])(rs) ∧
155.                udr:R • udr ∈ all_routes } end
```

1.1.2.2.4.6 Isolated Rail Net Units

We wish to analyse a rail net for the following property: can one reach every rail unit from any given rail unit ? The analysis function isolated decides on that !

157. Given two distinct net unit identifiers, *ui′* and *ui″*, of a railway net, *ui″* is isolated from *ui′* if there is no route in the railway net from *ui′* to *ui″*.

```
value
157. isolated: UI × UI → RS → Bool
157. isolated(uif,uit)(rs) ≡
157.    let all_routes = routes(rs) in
157.    ∼∃ r:Route•r∈all_routes⇒∃ i,j:Nat•{i,j}⊆inds r∧i<j∧r(i)=(_,uif,_)∧r(j)=(_,uit,_) end
157.    pre {uif,uit}⊆xtr_UIs(rs)
```

1.1.2.2.4.7 A Delineation: Train Stations

In preparation for our later introduction of a notion of trains we shall attempt to delineate a notion of train station. By a train station we shall understand a largest set of connected rail units all designated as being in that station.

158. We shall therefore, presently, introduce a predicate: in_station that applies to a rail unit and yields **true** if it is a designated train station, **false** otherwise.
159. Based on a rail unit, *nu*, that satisfies in_station, i.e., in_station(*nu*) and on the mereology of stations, i.e., the connected rail units, beginning with *nu*, we define an analysis function which calculates the "full" station from *nu*.
160. Finally we define an analysis function station which, given a station rail unit calculates the largest set of rail units belonging to the same station.

```
type
160.  Station = NU-set
value
158.  in_station: NU → Bool
axiom
160.  ∀ st:Station, ∀ nu:NU•nu ∈ st ⇒ in_station(nu)
value
159.  station: NU → RS → NU-set
159.  station(inu)(rs) ≡
159.    let st = {inu} ∪
159.      { nu |
159.        stnu:NU • stnu ∈ st ∧
159.        let (iuis,ouis) = mereo_NU(stnu) in
159.        let cnus = { get_NU(ui)(rs) | ui:UI • ui ∈ iuis ∪ ouis } in
159.        nu ∈ cnus ∧ in_station(nu) end end }
159.    in st end
159.    pre: in_station(nu)
```

How we may determine whether a rail unit is a station is left undefined. That is, we refrain from any (speculation) as to whether stations can be characterised by certain topological features of rail unit connections.

1.1.2.2.4.8 All Stations of a Railway System

161. We define an analysis function, all_stations, which calculates, from a railway system its set of two or more stations.
162. We calculate, snus, the set of all station rail units.
163. For each of these we calculate the station to which these station rail units belong.

```
value
161.  all_stations: RS → Station-set
161.  all_stations(rs) ≡
162.    let snus = { nu | nu:NU • nu ∈ xtr_NUs(rs) ∧ in_station(nu) } in
163.    { station(nu)(rs) | nu:NU • nu ∈ snus } end
axiom
161.  card all_stations(rs) ≥ 2
```

Two or more rail units, nu, of line 163 may calculate the same station.

1.1.2.2.4.9 Rail Lines

164. By a trail line we mean a route that connects two neighbouring stations.
165. is_connected_stations: Given two stations it may be that there are no routes connecting them.
166. connecting_line: We can calculate a line, ln, that does connect two connected stations.
Given two stations that are connected there will be a number of rail units in both stations that can serve as end points of their connecting rail line. We would then say that these end point rail units designate respective station platforms from and to where trains depart, respectively arrive.
167. is_immediately_connecting_line: We can inquire as to whether there is an immediately connecting line between two given stations of a railway system.

```
type
164.  LN = R
axiom
164.  ∀ rs:RS • ∀ ln:LN • ln ∈ routes(rs) ⇒
164.     let (_,1ui,_) = hd ln, (_,nui,_) = ln[len ln] in
164.     let 1nu = get_NU(1ui)(rs), nnu = get_NU(nnu)(rs) in
164.     in_station(1nu) ∧ in_station(nni) end end
value
165.  is_connected_stations: Station × Station → RS → Bool
165.  is_connected_stations(fs,ts)(rs) ≡
165.     let all_routes = routes(rs) in
165.     ∃ ln:R • ln ∈ all_routes •
164.        let (_,1ui,_) = hd ln, (_,nui,_) = ln[len ln] in
164.        let 1nu = get_NU(1ui)(rs), nnu = get_NU(nnu)(rs) in
164.        fs = 1nu ∧ ts = nnu end end
165.     end
165.     pre: {fs,ts}⊆all_stations(rs)

166.  connecting_line: Station × Station → RS → LN
166.  connecting_line(fs,ts)(rs) ≡
165.     let all_routes = routes(rs) in
166.     let ln:R • ln ∈ all_routes •
164.           let (_,1ui,_) = hd ln, (_,nui,_) = ln[len ln] in
164.           let 1nu = get_NU(1ui)(rs), nnu = get_NU(nnu)(rs) in
164.           fs = 1nu ∧ ts = nnu end end
166.     ln end end
166.     pre: is_connected_stations(fs,ts)(rs)

167.  is_immediately_connecting_line:  Station × Station → RS → Bool
167.  is_immediately_connecting_line(fs,ts)(rs) ≡
167.     let ln = connecting_line(fs,ts)(rs) in
167.     ∀ (_,ui,_):TUI • (_,ui,_) ∈ inds ln ⇒
167.        let s =  get_RU(ui)(rs) in
167.        s ∈ fs ∪ ts ∨ ∼in_station(s) end end
167.     pre: is_connected_stations(fs,ts)(rs)
```

We leave it to the reader to define analysis functions that yield the set of all [immediately] connecting lines between two stations of a railway system.

1.1.2.3 Attributes

Attributes are either static, [STA], or monitorable, [MON], or programmable, [PRG].

1.1.2.3.1 Rail Nets

We treat attributes of rail units.

168. A rail unit is either in a station or is not, STA.
169. A rail unit is in some state – where a state is a possibly empty set of pairs of unique identifiers of connected rail units – with these being in respective set of the pair of sets making up the mereology of the rail unit, PRG.
Figure 1.3 shows the twelve possible state of a point.

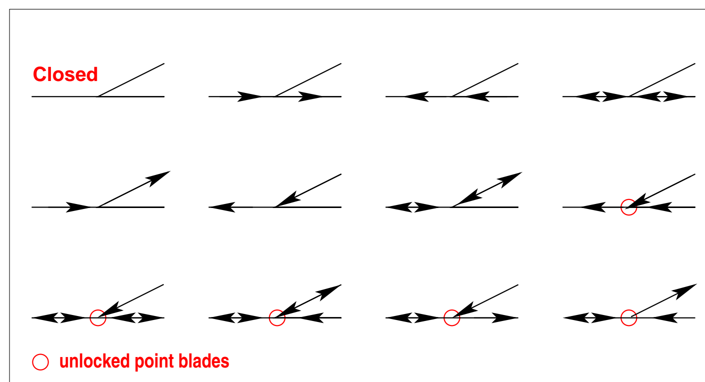


Fig. 1.3 The 12 Possible States of a Turnout Point

If a switch in unlocked, a train coming from either of the converging directs will pass through the points onto the narrow end, regardless of the position of the points, as the vehicle's wheels will force the points to move. Passage through a switch in this direction is known as a trailing-point movement.

axiom
∀ pu:PU • pu ∈ xtr_NUs(ps) ⇒ **let** ({i},{o1,o2})=mereo_RU(pu), ω=attr_RUΩ(pu) **in**
ω = {{}, {{(i,o1)}}, {(o1,i)}, {{(i,o1),(o1,i)}},
{{(i,o2)}}, {(o2,i)}, {(i,o2),(o2,i)}, {(i,o2),(o2,i),(o1,i)},
{{(i,o1),(o1,i),(o2,i)}}, {{(i,o2),(o2,i),(o1,i)}}, {{(i,o1),(o2,i)}}, {{(i,o2),(o2,i)}}}
end

170. A rail unit has a state space – consisting of all the states that a rail unit may attain, STA.
171. A point or a slip is either un-locked or locked, that is, its blades can be pressed to move, or cannot.
172. A rail unit has a length, STA.
173. A rail unit is either occupied by (a section of) an identified train or is not, PRG.
174. Et cetera.

type
168. RU_In_St = **Bool** [STA]
169. RUΣ = (UI×UI)-**set** [PRG]
170. RUΩ = RΣ-**set** [STA]

```
170. Lock_Status = ″un-locked″ | ″locked″ [PRG]
172. RU_Len [STA]
173. RU_Train == TI | ‶nil″ [PRG]
174. ...
value
168. attr_RU_In_st: RU → RU_In_St
169. attr_RUΣ: RU → RUΣ
170. attr_RUΩ: RU → RUΩ
170. attr_(PU|RU|SU|DU)_Lock_Status: (PU|RU|SU|DU)→Lock_Status
172. attr_RU_Len: RU → RU_Len
173. attr_Train: RU → RU_Train
axiom
169. ∀ rs:RS,ru:RU • ru∈retr_NUs(rs) ⇒
169.   let uis = retr_UIs(rs), (iuis,ouis) = mereo_RU(ru), σ = attr_Σ(ru) in
169.   ∀ (iui,oui):(UI×UI) • (iui,oui)∈σ ⇒ iui∈iuis∧oui∈ouis∧{iui,oui}⊆uis
170.  ∧ σ ∈ attr_Ω(ru) end
173.  ∧ (attr_Train(ru) ∈ tris ∨ attr_Train(ru) = ‶nil″)
174.  ...
```

For any given switch the state space may be a proper subset of the set of all possible states.

1.1.2.3.2 Open Routes

175. A route is said to be open if all pairs of the first and last element of route triplets are in the current state of the rail unit designated by the second element of these route triplets.

```
value
175. is_open_route: R → RS → Bool
175. is_open_route(r)(rs)
175.   ∀ (iu,ui,ou):TUI • (iu,ui,ou) ∈ elems r ⇒
175.     let ru = get_RU(ui)(rs) in let σ = attr_RUΣ(ru) in (iu,ou) ∈ σ end end
175.   pre: r ∈ routes(rs)
```

1.1.2.3.3 Station Names

176. All rail units of a station has the same station name.
177. No two distinct stations have the same name.

```
value
177. station_name: Station → Station_Name
177. station_name(st) ≡ let ru:RU • ru ∈ st in attr_Name(ru) end
axiom
176. ∀ rs:RS • let rn = obs_RN(rs) in
176.   ∀ st,st′:Station • {st,st′}⊆stations(rn) ⇒
176.     ∀ ru,ru′:RU • {ru,ru′}⊆ ∈ st ⇒ attr_Name(ru) = attr_Name(ru′)
177.   st≠st′ ⇒ station_name(st)≠station_name(st′)
177. end
```

1.1.2.3.4 Trains

178. Trains have length with those of a given name having not necessarily the same length.
179. Trains [are expected to] follow a route, Train_Route, and to be, at any time, at a Train_Position.
  A Train_Route is a sequence of zero, one or more timed triplets, TUIT, of rail unit identifiers.
  A Train_Position is a train attribute. It consists of three elements. Two train routes, ptr (past train route) and ntr (next train route), and a [current] timed triplet, TUIT, of rail unit identifiers. The meaning of a Train_Position is that the train has passed the past route, is at the current timed triplet, and can next enter the next route.
180. No two distinct trains occupy overlapping routes on the net.
181. Trains have a speed and acceleration (or deceleration).
182. ...

```
type
178. Train_Length [STA]
179. TUIT = TUI×𝕋𝕀𝕄𝔼
179. Train_Route = TUIT*
179. Train_Position = ptr:Train_Route × TUIT × ntr:Train_Route [PRG]
181. Train_Speed, Train_Acceleration, Train_Deceleration [MON]
182. ...
value
178. attr_Train_Length: Train → Train_Length
179. attr_Train_Position: Train → Train_Position
181. attr_Train_Speed: Train → Train_Speed
181. attr_Train_Acceleration: Train → Train_Acceleration
181. attr_Train_Deceleration: Train → Train_Deceleration
182. ...
axiom
179. ∀ rs:RS •
179.   ∀ tr,tr′:Train • {tr,tr′} ⊆ obs_TS(rs) ∧ tr ≠ tr′
179.     ⇒ is_open_route(attr_Train_Position(train))(rs)
179.     ∧ let (trp,trp′) = attr_Train_Position(tr,tr′) in
179.       {rui|(_,(rui,_),_):TTUIT•(_,(rui,_),_) ∈ elemens trr}
179.       ∩
179.       {rui|(_,(rui′,_),_):TUI•(_,(rui′,_),_) ∈ elemens trr′} = {}
179.       end
```

1.1.2.3.5 An Intentional Pull

183. For every railway system it is the case that
184. for every rail unit in that system which "records", as an attribute, a train, there is exactly one train that in its route position records exactly that rail unit,
185. and vice versa.

```
axiom
183. ∀ rs:RS •
184.   ∀ ru:RU • ru ∈ retr_NUs(rs) ⇒
184.     if attr_RU_Train(ru) ≠ "nil" ⇒
184.       ∃! tr:Train • tr ∈ trains(rs) ∧
184.         uid_NU(ru) ∈ {ui|(_,ui,_):TUI • (_,ui,_) ∈ elems attr_Train_Position(tr)}
183.   ∧
```

```
185.     ∀ tr:Train • tr ∈ trains(rs) ⇒
185.         ∀ (_,ui,_):TUI • (_,ui,_) ∈ elems attr_Train_Position(tr) ⇒
185.             attr_RU_Train(get_NU(ui)(rs)) = uid_Train(tr)
183. end
```

1.1.2.3.6 History Attributes

The attributes and axioms over them – covered above do not relate to time; they are time-independent. We now treat time-dependent attributes and axioms over them. By $\mathbb{TIME}$ we mean absolute times, like *November 15, 2021: 16:12*, and by $\mathbb{TI}$ we man time intervals, like *two hours, three minutes and five seconds*. We shall here consider $\mathbb{TIME}$ to span a definite "period" of time, say from *January 1, 2020, 00:00am* to *December 31, 2020, 24:00*.

186. Of a road unit we can speak of its history as a time-decreasing, ordered sequence of time-stamped train identifiers.
187. Of a train we can speak of its history as a time-decreasing, ordered sequence of time-stamped rail unit identifiers.

We could have considered other properties to form or be included in event histories, but abstain.

```
type
186. RU_Hist = (𝕋𝕀𝕄𝔼 × TI)* [PRG]
187. TR_Hist = (𝕋𝕀𝕄𝔼 × UI)* [PRG]
value
186. attr_RU_Hist: RU → RU_Hist
187. attr_TR_Hist: Train → TR_Hist
axiom
186. [ descending times in rail unit history ]
187. [ descending times in train history ]
```

1.1.2.3.7 The Intentional Pull Revisited

188. For every railway system it is the case that
189. for every rail unit,
190. if at any time it records a train,
191. then that train's event history records that rail unit in the route it is occupying at that time, and
192. for every train, if at any time it records a route
193. then exactly the rail units of that route record that train.

... below function has to be redefined ...

```
axiom
188. ∀ rs:RS •
189.     ∀ ru:RU • ru ∈ retr_NUs(rs) ⇒
189.         let ruh = attr_RU_Hist(ru) in
190.         ∀ time:dom ruh • ruh(time) ≠ {} ⇒
191.             let {ti} = ruh(time) in
191.             let trh = attr_TR_Hist(get_Train(ti)(rs)) in
191.             trh(time) ≠ {} ∧
191.             let {r} = trh(time) in
```

```
191.        ∃ (_,ui,_):TUI • (_,ui,_) ∈ elems(r) ⇒ ruh = get_RU(ui)(rs)
191.      end end end end
192.    et cetera
193.    et cetera
```

1.2 Transcendental Deduction

1.2.1 General

By a transcendental deduction parts can be "morphed" into behaviours. We consider the following railway system parts:

- all the railway net units and
- all the trains.

That is, we shall not here consider the railway net management, the train operator, the passenger and [freight] shipper parts as behaviours.

1.2.2 A Note on $\mathbb{TIME}$

194. We shall consider $\mathbb{TIME}$ to stand for a time in a definite interval of times, for example from *January 1, 2020, 00:00 am* to *December 31, 2020, 23:59:59*.
195. That is, $\mathbb{TIME}$-interval, is the set of all the designated times in the interval.
196. The operators $\mathcal{F}$[irst] and $\mathcal{L}$[ast] applied to the $\mathbb{TIME}$-interval interval yields the first and last times of the interval $\mathbb{TIME}$-interval.
197. We shall introduce a time interval quantity, $\delta\tau$:$\mathbb{TI}$ – and shall consider $\delta\tau$ to be, if not infinitesimal small, then at least "small", say, in the context of train traffic, *1 second !*
198. We shall, loosely, introduce the operator $\mathcal{D}$, applied to the interval $\mathbb{TIME}$interval, to yield the definite set of times such that if τ is in $\mathbb{TIME}$-interval and τ is not $\mathcal{L}$($\mathbb{TIME}$-interval)then the next time in $\mathbb{TIME}$interval is $\tau+\delta\tau$.

```
type
194.  𝕋𝕀𝕄𝔼
195.  𝕋𝕀𝕄𝔼−interval
196.  ℱ: 𝕋𝕀𝕄𝔼−interval → 𝕋𝕀𝕄𝔼
196.  ℒ: 𝕋𝕀𝕄𝔼−interval → 𝕋𝕀𝕄𝔼
value
197.  δτ:𝕋𝕀 [ say 1 second ]
198.  𝒟: 𝕋𝕀𝕄𝔼−interval → 𝕋𝕀𝕄𝔼-set
```

1.2.3 Train Traffic

199. By train traffic we shall understand a discrete function, in RSL [92] expressed as a map, over a closed interval of time from time to trains and their route position.
We model this as shown in formula line 199.

Here we have taken the liberty of modeling the traffic as being discrete over infinitesimal small time intervals $\delta\tau$.

```
type
199. TrainTraffic = TI →m (𝕋𝕀𝕄𝔼 →m R)
```

1.2.3.1 Well-formed Train Traffics

200. For every railway system a train traffic is well-formed
   a. if all trains cover the same time period;
   b. if all train traffics occur on routes of the railway system;
   c. if two or more trains do not have overlapping routes at any time; and
   d. if each train traffic progresses monotonically.

```
axiom
200.  ∀ rs:RS •
200.     ∀ trtr:TrainTraffic •
200a.        same_time_period(trtr)
200b.     ∧ routes_of_rs(trtr)(rs)
200c.     ∧ disjoint_routes(trtr)(rs)
200d.     ∧ monotonic(trtr)(rs)

200a.  same_time_period: TrainTraffic → Bool
200a.  same_time_period(trtr) ≡ ∀ time,time′:𝕋𝕀𝕄𝔼 • 𝔻𝕆𝕄𝔸𝕀ℕ(time)=𝔻𝕆𝕄𝔸𝕀ℕ(time′)

value
200b.  routes_of_rs: TrainTraffic → RS → Bool
200b.  routes_of_rs(trtr)(rs)
200b.     ∀ ti:TI • ti ∈ dom trtr ⇒
200b.        ∀ time:𝕋𝕀𝕄𝔼 • time ∈ dom ti
200b.           route_of((trtr(ti))(time))(rs)
200b.  route_of: R → RS → Bool
200b.  route_of(r)(rs) ≡ r ∈ routes(rs)

value
200c.  disjoint_routes: TrainTraffic → RS → Bool
200c.  disjoint_routes(trtr)(rs) ≡
200c.     ∀ ti,ti′:TI • {ti,ti′}≤=dom trtr ∧ ti≠ti′ ⇒
200c.           ∀ time:𝕋𝕀𝕄𝔼 • time ∈ dom ti ⇒
200c.              disjoint_routes((trtr(ti))(time),(trtr(ti′))(time))
200c.  disjoint_routes: R × R → Bool
200c.  disjoint_routes(r,r′) ≡
200c.     {ui|(_,ui,_):TUI•(_,ui,_)∈ elems r}∩{ui|(_,ui,_):TUI•(_,ui,_)∈ elems r′} = {}
```

For a traffic to be monotonic it must be the case that

201. for all trains
202. for two “closely adjacent” times in the domain of that train’s traffic
203. the route positions, r, r' of any train (at these times) must

204. either be the same. i.e. $r = r'$,
205. or truncated by at most the first element, i.e. r'=**tl** r (being a route of the system),
206. or amended by at most one element, i.e., $r'=r\frown\langle$tui$\rangle$ (being a route of the system),
207. or both, i.e., r'=**tl** $r\frown\langle$tui$\rangle$ (being a route of the system).

```
value
200d.  monotonic: TrainTraffic → RS → Bool
200d.  monotonic(trtr)(rs) ≡
202.    ∀ ti:TI • ti ∈ dom trtr ⇒ in
202.      ∀ time,time′:𝕋𝕀𝕄𝔼 •
202.        {time,time′} ⊆ 𝔻𝕆𝕄𝔸𝕀ℕ(trtr(ti))
202.        time′>time ∧ time′−time=δτ  ∧
203.        let (r,r′) = ((trtr(ti))(time),(trtr(ti))(time′)) in
204.        (r′= r) ∨
205.        (r′= tl r ∧ tl r ∈ routes(rs)) ∨
206.        (r′= r⌢⟨tui⟩ ∧ r⌢⟨tui⟩ ∈ routes(rs)) ∨
207.        (r′= tl r⌢⟨tui⟩ ∧ tl r⌢⟨tui⟩ ∈ routes(rs))
203.        end
```

1.3 Perdurants

To every part, that is,

208. linear unit,
209. turn out,
210. rigid crossing,
211. slip (crossing),
212. double (crossing),
213. terminal unit, and
214. train

we associate, by a transcendental deduction, a never ending train behaviour which, as a function, takes some arguments ...→... and otherwise goes on forever (**Unit**).

```
value
208. linear_unit: ... → ... Unit
209. turn_out: ... → ... Unit
210. rigid: ... → ... Unit
211. slip: ... → ... Unit
212. double: ... → ... Unit
213. terminal: ... → ... Unit
214. train: ... → ... Unit
```

The **Unit** does not refer to the railway units of the domain, but is an RSL ... in effect designating never ending processes.

1.3.1 Channels

215. Trains and rail net units exchange messages, NT_Msg.
These message will eventually be further defined.
216. Trains potentially communicate with all rail net units.
Rail net units potentially communicate with all trains.

```
type
215.  NT_Msg
channel
216.  { ch[{ui,tri}]:NT_Msg | ui:UI, tri:TRI • ui ∈ uis ∧ tri ∈ trus }
```

In a more realistic railway system domain description a rail net management would monitor trains and control (set) switches etc.

1.3.2 Behaviour Signatures

We continue sketching some of the railway system behaviour signatures. Rail net unit and train identifiers become [first] parameters; mereology attributes become [second set of] parameters; static attributes become [third set of] parameters; programmable attributes become [fourth] parameters; and channel references become "last" parameters.

value
208. linear_unit: ui:UI × (_,tris):Unit_Mereo × (RUΩ×RU_Len) → (RUΣ×RU_Hist)
→ **in**,**out** {ch[{ui,ti}]|ti:TI•ti∈tris} **Unit**
209. turn_out: ui:UI × (_,tris):Unit_Mereo × (RUΩ×RU_Len) → (RUΣ×RU_Hist)
→ **in**,**out** {ch[{ui,ti}]|ti:TI•ti∈tris} **Unit**
210. rigid: ui:UI × (_,tris):Unit_Mereo × (RUΩ×RU_Len) → (RUΣ×RU_Hist)
→ **in**,**out** {ch[{ui,ti}]|ti:TI•ti∈tris} **Unit**
211. slip: ui:UI × (_,tris):Unit_Mereo × (RUΩ×RU_Len) → (RUΣ×RU_Hist)
→ **in**,**out** {ch[{ui,ti}]|ti:TI•ti∈tris} **Unit**
212. double: ui:UI × (_,tris):Unit_Mereo × (R`UOmega×RU_Le) → (RUΣ×RU_Hist)
→ **in**,**out** {ch[{ui,ti}]|ti:TI•ti∈tris} **Unit**
213. terminal: ui:UI × (_,tris):Unit_Mereo × (RUΩ×RU_Len) → (RUΣ×RU_Hist)
→ **in**,**out** {ch[{ui,ti}]|ti:TI•ti∈tris} **Unit**

214. train: ti:TI× uis:Train_Mereo × (TRΩ×Train_Length) → (Train_Position×(TRΣ×TR_Hist))
→ **in**,**out** {ch[{ui,ti}]|ui:UI•ii∈uis} **Unit**

1.3.3 Behaviour Definitions

We shall illustrate only a narrow aspect of trains on rails. Namely that of the "simulation" of train traffic as per pre-planned routes. That is we shall not model actual train traffic as per set time tables – that would entail numerous more formulas than we now show. So it is only an illustration of how rail and train behaviours might look.

1.3.3.1 Rail Unit Behaviours

We shall only exemplify linear rail unit behaviours.

217. Rail unit behaviours all have in common what we now model as the linear rail unit behaviour.
218. Non-deterministically, external choice, the rail units offers to accept communication from passing trains, ti, as to the time they are passing by –
219. with this information being added to the rail unit history as the rail unit behaviour resumes.

value
217. linear_unit(ui,(ruω,...),(_,tris))(ruσ,ruh) ≡
218. **let** Msg_TR_RU(time,ti) = ⟦⟧ {ch[{ui,ti}] ? | ti ti:TI • ti ∈ tris} **in**
219. linear_unit(ui,(ruω,...),(_,tris))(ruσ,⟨(time,ti)⟩⌢ruh)
217. **end**
217. **pre**: ruσ ∈ ruω

1.3.3.2 Train Behaviour

We focus, in our description of train behaviours sôlely on the un-aided movement of trains and, further, on an "idealised" description.

220. There are two train positions of interest when describing train movement:
    a. the general situation where the train has not yet reached its final destination, and
    b. the special situation where the train has indeed reached its final destination
221. In the former (Item 220a.) the train position, at time τ, is at rail unit *ui*, with the first next unit being *ui′* (and where *aui*=*aui′*).
222. If elapsed time is less than planned time τ,
223. then the train informs the rail unit behaviour designated by *ui* that it is currently passing it.
224. and moves on within the current unit *ui*, having updated its history;
225. else, when elapsed time is up, i.e., equals planned time τ, the train informs the rail unit it is now entering that it is so,
226. updates its history accordingly and moves on to the next unit, *ui′*

```
value
221. train(ti,sta,uis)(pr,((bui,ui,aui),τ),⟨((aui′,ui′,nui),τ′)⟩⁀nr),(trσ,trh) ≡
221.     let time = record_𝕋𝕀𝕄𝔼 in
222.     if time < τ
223.     then ch[{ui,ti}] ! Msg_TR_RU(time,ti) ;
224.         train(ti,sta,uis)(pr,((bui,ui,aui),τ),⟨((aui′,ui′,nui),τ′)⟩⁀nr),(trσ,⟨(time,ui)⟩⁀trh)
225.     else ch[{aui′,ti}] ! Msg_TR_RU(time,ti) ; assert: time = τ
226.         train(ti,sta,uis)(tp⁀⟨(τ,(bui,ui,aui))⟩,((aui′,ui′,nui),τ′),nr),(trσ,⟨(time,ui′)⟩trh)
221.     end end
221.     pre: trσ ∈ trω ∧ aui=aui′ ∧ τ < τ′
```

227. In the other position (Item 220b.) the train, at time τ, is at rail unit *ui*, with their bing no next units to enter.
228. If elapsed time is less than planned time, τ,
229. then the train informs the rail unit behaviour designated by *ui* that it is currently passing it
230. and moves on within the current unit *ui*, having updated its history;
231. else the train journey has ended and the train behaviour "stops", i.e., ceases to exist !

```
227. train(ti,sta,uis)((pr,((bui,ui,aui),τ),⟨⟩),(trσ,trh)) ≡
227.     let time = record_𝕋𝕀𝕄𝔼 in
228.     if time < τ
229.     then ch[{ui,ti}] ! Msg_TR_RU(time,ti) ;
230.         train(ti,sta,uis)((pr,((bui,ui,aui),τ),⟨⟩),(trσ,⟨(time,ui)⟩⁀trh))
231.     else skip assert: time = τ
227.     end end
227.     pre: trσ ∈ trω
```

1.4 Closing

We end our example here. To analyse & describe a proper railway system we would have to introduce some rail net and train management. Rail net management would monitor the rails, and, according to train time tables issued by train management, set switches. Train management

would establish train time tables, pass these onto rail net management, and would monitor and control trains. We have given, we think, enough clues as how to analyse & describe such railway systems.

Chapter 2

Road Transport [2007–2017]

Contents

2.1 The Road Transport Domain 58
2.1.1 Naming 58
2.1.2 Rough Sketch 58
2.2 External Qualities 58
2.2.1 A Road Transport System, II – Abstract External Qualities 58
2.2.2 Transport System Structure 59
2.2.3 Atomic Road Transport Parts 59
2.2.4 Compound Road Transport Parts 59
2.2.4.1 The Composites 59
2.2.4.2 The Part Parts 59
2.2.5 The Transport System State 60
2.3 Internal Qualities 61
2.3.1 Unique Identifiers 61
2.3.1.1 Extract Parts from Their Unique Identifiers 61
2.3.1.2 All Unique Identifiers of a Domain 61
2.3.1.3 Uniqueness of Road Net Identifiers 62
2.3.2 Mereology 62
2.3.2.1 Mereology Types and Observers 62
2.3.2.2 Invariance of Mereologies 63
2.3.2.2.1 Invariance of Road Nets 63
2.3.2.2.2 Possible Consequences of a Road Net Mereology 64
2.3.2.2.3 Fixed and Varying Mereology 64
2.3.3 Attributes 64
2.3.3.1 Hub Attributes 64
2.3.3.2 Invariance of Traffic States 65
2.3.3.3 Link Attributes 65
2.3.3.4 Bus Company Attributes 66
2.3.3.5 Bus Attributes 66
2.3.3.6 Private Automobile Attributes 66
2.3.3.7 Intentionality 67
2.4 Perdurants 68
2.4.1 Channels and Communication 68
2.4.1.1 Channel Message Types 68
2.4.1.2 Channel Declarations 69
2.4.2 Behaviours 69
2.4.2.1 Road Transport Behaviour Signatures 69
2.4.2.1.1 Hub Behaviour Signature 70
2.4.2.1.2 Link Behaviour Signature 70
2.4.2.1.3 Bus Company Behaviour Signature 70
2.4.2.1.4 Bus Behaviour Signature 71
2.4.2.1.5 Automobile Behaviour Signature 71
2.4.2.2 Behaviour Definitions 71
2.4.2.2.1 Automobile Behaviour at a Hub 71
2.4.2.2.2 Automobile Behaviour On a Link 72

2.4.2.2.3 Hub Behaviour 73
2.4.2.2.4 Link Behaviour 73
2.5 System Initialisation 74
2.5.1 Initial States 74
2.5.2 Initialisation 74

2.1 The Road Transport Domain

Our universe of discourse in this chapter is the road transport domain.

2.1.1 Naming

type RTS

2.1.2 Rough Sketch

The road transport system that we have in mind consists of a road net and a set 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 vehicles to additionally include departments of motor vehicles (DMVs), 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[10] □

2.2 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[11]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. □

2.2.1 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,

[10] 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.

[11] We have **highlighted** certain endurant sort names – as they will re-appear in rather many upcoming examples.

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.

2.2.2 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` □

2.2.3 Atomic Road Transport Parts

From one point of view all of the following can be considered atomic parts: hubs, links[12], and automobiles.

2.2.4 Compound Road Transport Parts

2.2.4.1 The Composites

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

It is structured into

233. a *road net*, RN, and
234. a *fleet of vehicles*, FV.

Both are structures. ....

type
232 UoD **axiom** $\forall$ uod:UoD • is_structure(uod).
233 RN **axiom** $\forall$ rn:RN • is_structure(rn).
234 FV **axiom** $\forall$ fv:FV • is_structure(fv).

value
233 obs_RN: UoD $\to$ RN
234 obs_FV: UoD $\to$ FV □

2.2.4.2 The Part Parts

235. The structure of hubs is a set, sH, of atomic hubs, H.
236. The structure of links is a set, sL, of atomic links, L.
237. The structure of buses is a set, sBC, of composite bus companies, BC.
238. The composite bus companies, BC, are sets of buses, sB.
239. The structure of private automobiles is a set, sA, of atomic automobiles, A.

[12] Hub ≡ street intersection; link ≡ street segments with no intervening hubs.

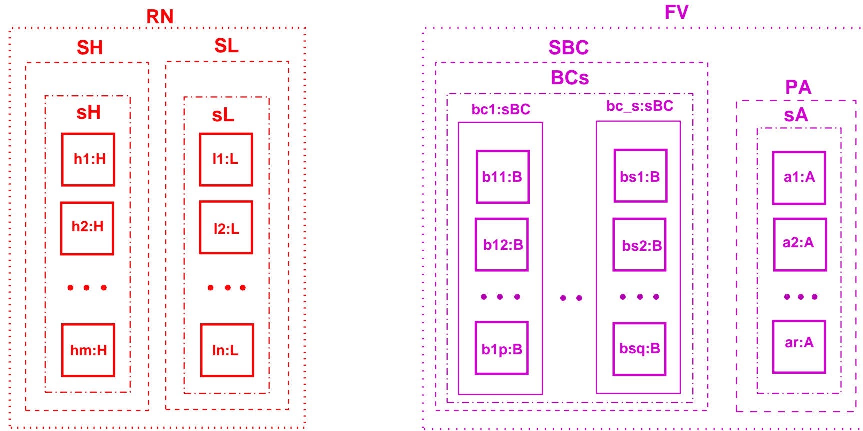


Fig. 2.1 A Road Transport System Compounds and Structures

```
235  H, sH = H-set  axiom ∀ h:H • is_atomic(h)
236  L, sL = L-set  axiom ∀ l:L • is_atomic(l)
237  BC, BCs = BC-set  axiom ∀ bc:BC • is_composite(bc)
238  B, Bs = B-set  axiom ∀ b:B • is_atomic(b)
239  A, sA = A-set  axiom ∀ a:A • is_atomic(a)
value
235  obs_sH: SH → sH
236  obs_sL: SL → sL
237  obs_sBC: SBC → BCs
238  obs_Bs: BCs → Bs
239  obs_sA: SA → sA   □
```

2.2.5 The Transport System State

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

    From that state we can calculate other states.

241. The set of all hubs, *hs*.
242. The set of all links, *ls*.
243. The set of all hubs and links, *hls*.
244. The set of all bus companies, *bcs*.
245. The set of all buses, *bs*.
246. The set of all private automobiles, *as*.
247. The set of all parts, *ps*.

```
value
240  rts:UoD    [240]
241  hs:H-set ≡:H-set ≡ obs_sH(obs_SH(obs_RN(rts)))
242  ls:L-set ≡:L-set ≡ obs_sL(obs_SL(obs_RN(rts)))
243  hls:(H|L)-set ≡ hs∪ls
244  bcs:BC-set ≡ obs_BCs(obs_SBC(obs_FV(obs_RN(rts))))
245  bs:B-set ≡ ∪{obs_Bs(bc)|bc:BC•bc ∈ bcs}
246  as:A-set ≡ obs_BCs(obs_SBC(obs_FV(obs_RN(rts))))
247  ps:(UoB|H|L|BC|B|A)-set ≡ rts∪hls∪bcs∪bs∪as
```

2.3 Internal Qualities

2.3.1 Unique Identifiers

248. We assign unique identifiers to all parts.
249. By a road identifier we shall mean a link or a hub identifier.
250. By a vehicle identifier we shall mean a bus or an automobile identifier.
251. 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
248  H_UI, L_UI, BC_UI, B_UI, A_UI
249  R_UI = H_UI | L_UI
250  V_UI = B_UI | A_UI
value
251a  uid_H: H → H_UI
251b  uid_L: H → L_UI
251c  uid_BC: H → BC_UI
251d  uid_B: H → B_UI
251e  uid_A: H → A_UI
```

2.3.1.1 Extract Parts from Their Unique Identifiers

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

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

2.3.1.2 All Unique Identifiers of a Domain

We can calculate:

253. the *set*, $h_{ui}s$, of *u*nique *h*ub *i*dentifiers;
254. the *set*, $l_{ui}s$, of *u*nique *l*ink *i*dentifiers;
255. the *map*, $hl_{ui}m$, 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,
256. the *map*, $lh_{ui}m$, 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;
257. the *set*, $r_{ui}s$, of all *u*nique hub and link, i.e., *r*oad *i*dentifiers;
258. the *set*, $bc_{ui}s$, of *u*nique *b*us *c*ompany *i*dentifiers;
259. the *set*, $b_{ui}s$, of *u*nique *b*us *i*dentifiers;
260. the *set*, $a_{ui}s$, of *u*nique private *a*utomobile *i*dentifiers;
261. the *set*, $v_{ui}s$, of *u*nique bus and automobile, i.e., *v*ehicle *i*dentifiers;
262. the *map*, $bcb_{ui}m$, from *u*nique *b*us *c*ompany *i*dentifiers to the *s*et of its *u*nique *b*us *i*dentifiers; and
263. the (*b*ijective) *map*, $bbc_{ui}bm$, from *u*nique *b*us *i*dentifiers to their *u*nique *b*us *c*ompany *i*dentifiers.

```
value
253  h_uis:H_UI-set ≡ {uid_H(h)|h:H•h ∈ hs}
254  l_uis:L_UI-set ≡ {uid_L(l)|l:L•l ∈ ls}
257  r_uis:R_UI-set ≡ h_uis∪l_uis
255  hl_uim:(H_UI →m L_UI-set) ≡
255      [h_ui↦luis|h_ui:H_UI,luis:L_UI-set•h_ui∈h_uis∧(_,luis,_)=mereo_H(η(h_ui))]  [cf. Item 270]
```

```
256 lh_ui m:(L+UI →m H_UI-set) ≡
256    [l_ui↦huis | h_ui:L_UI,huis:H_UI-set • l_ui∈l_ui s ∧ (_,huis,_)=mereo_L(η(l_ui))] [cf. Item 271]
258 bc_ui s:BC_UI-set ≡ {uid_BC(bc)|bc:BC•bc ∈ bcs}
259 b_ui s:B_UI-set ≡ ∪{uid_B(b)|b:B•b ∈ bs}
260 a_ui s:A_UI-set ≡ {uid_A(a)|a:A•a ∈ as}
261 v_ui s:V_UI-set ≡ b_ui s ∪ a_ui s
262 bcb_ui m:(BC_UI →m B_UI-set) ≡
262    [ bc_ui ↦ buis | bc_ui:BC_UI, bc:BC • bc∈bcs ∧ bc_ui=uid_BC(bc) ∧ (_,_,buis)=mereo_BC(bc) ]
263 bbc_ui bm:(B_UI →m BC_UI) ≡
263    [ 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) ]
```

2.3.1.3 Uniqueness of Road Net Identifiers

We must express the following axioms:

264. All hub identifiers are distinct.
265. All link identifiers are distinct.
266. All bus company identifiers are distinct.
267. All bus identifiers are distinct.
268. All private automobile identifiers are distinct.
269. All part identifiers are distinct.

```
axiom
264 card hs = card h_ui s
265 card ls = card l_ui s
266 card bcs = card bc_ui s
267 card bs = card b_ui s
268 card as = card a_ui s
269 card {h_ui s∪l_ui s∪bc_ui s∪b_ui s∪a_ui s}
269    = card h_ui s+card l_ui s+card bc_ui s+card b_ui s+card a_ui s  □
```

2.3.2 Mereology

2.3.2.1 Mereology Types and Observers

270. The mereology of hubs is a pair: (i) the set of all bus and automobile identifiers[13], 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).[14]
271. 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.
272. The mereology of a bus company is a set the unique identifiers of the buses operated by that company.
273. 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[15].
274. The mereology of an automobile is the set of the unique identifiers of all links and hubs[16].

```
type
270 H_Mer = V_UI-set×L_UI-set
271 L_Mer = V_UI-set×H_UI-set
272 BC_Mer = B_UI-set
```

```
273  B_Mer = BC_UI×R_UI-set
274  A_Mer = R_UI-set
value
270  mereo_H: H → H_Mer
271  mereo_L: L → L_Mer
272  mereo_BC: BC → BC_Mer
273  mereo_B: B → B_Mer
274  mereo_A: A → A_Mer
```

2.3.2.2 Invariance of Mereologies

For mereologies one can usually express some invariants. Such invariants express *"law-like properties"*, facts which are indisputable.

2.3.2.2.1 Invariance of Road Nets

The observed mereologies must express identifiers of the state of such for road nets:

```
axiom
270  ∀ (vuis,luis):H_Mer • luis⊆l_uis ∧ vuis=v_uis
271  ∀ (vuis,huis):L_Mer • vuis=v_uis ∧ huis⊆h_uis ∧ card huis=2
272  ∀ buis:H_Mer • buis = b_uis
273  ∀ (bc_ui,ruis):H_Mer•bc_ui∈bc_uis∧ruis=r_uis
274  ∀ ruis:A_Mer • ruis=r_uis
```

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

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

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

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

[13] 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.

[14] 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.

[15] — that the bus might pass through

[16] — that the automobile might pass through

2.3.2.2.2 Possible Consequences of a Road Net Mereology

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

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

2.3.2.2.3 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*.

2.3.3 Attributes

2.3.3.1 Hub Attributes

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

282. 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.
283. 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.
284. 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
282 HΣ = (L_UI×L_UI)-set
axiom
282 ∀ h:H • obs_HΣ(h) ∈ obs_HΩ(h)
type
283 HΩ = HΣ-set
284 H_Traffic
284 H_Traffic = (A_UI|B_UI) →m (𝕋𝕀𝕄𝔼 × VPos)*
axiom
284 ∀ ht:H_Traffic,ui:(A_UI|B_UI) •
284     ui ∈ dom ht ⇒ time_ordered(ht(ui))
value
282 attr_HΣ: H → HΣ
283 attr_HΩ: H → HΩ
284 attr_H_Traffic: H → H_Traffic
value
284  time_ordered: (𝕋𝕀𝕄𝔼 × VPos)* → Bool
284  time_ordered(tvpl) ≡ ...
```

In Item 284 on the facing page we model the time-ordered sequence of traffic as a discrete sampling, i.e., $\overrightarrow{m}$, rather than as a continuous function, →.

2.3.3.2 Invariance of Traffic States

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

axiom
285 ∀ h:H • h ∈ hs ⇒
285 **let** hσ = attr_HΣ(h) **in** ∀ (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**

2.3.3.3 Link Attributes

We show just a few attributes.

286. 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.
287. 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".
288. 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.
289. The hub identifiers of link states must be in the set, $h_{ui}s$, of the road net's hub identifiers.

type
286 LΣ = H_UI-**set**
axiom
286 ∀ lσ:LΣ•**card** lσ=2
286 ∀ l:L • obs_LΣ(l) ∈ obs_LΩ(l)
type
287 LΩ = LΣ-**set**
288 L_Traffic
288 L_Traffic = (A_UI|B_UI) $\overrightarrow{m}$ ($\mathbb{T}$×(H_UI×Frac×H_UI))*
288 Frac = **Real**, **axiom** frac:Fract • 0<frac<1
value
286 attr_LΣ: L → LΣ
287 attr_LΩ: L → LΩ
288 attr_L_Traffic: : → L_Traffic
axiom
288 ∀ lt:L_Traffic,ui:(A_UI|B_UI)•ui ∈ **dom** ht ⇒ time_ordered(ht(ui))
289 ∀ l:L • l ∈ ls ⇒
289 **let** lσ = attr_LΣ(l) **in** ∀ (h$_{ui}i$,h$_{ui}i'$):(H_UI×K_UI) •
289 (h$_{ui}i$,h$_{ui}i'$) ∈ lσ ⇒ {h$_{ui_i}$,h$'_{ui_i}$} ⊆ h$_{ui}s$ **end**

2.3.3.4 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.

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

```
type
290  BusTimTbl
value
290  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.

2.3.3.5 Bus Attributes

We show just a few attributes.

291. Buses run routes, according to their line number, ln:LN, in the
292. bus time table, btt:BusTimTbl obtained from their bus company, and and keep, as inert attributes, their segment of that time table.
293. 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.
294. Et cetera.

```
type
291   LN
292   BusTimTbl
293   BPos   == atHub | onLink
293a  atHub    :: h_ui:H_UI
293b  onLink   :: fh_ui:H_UI×l_ui:L_UI×frac:Fract×th_ui:H_UI
293b  Fract    = Real, axiom frac:Fract • 0<frac<1
294   ...
value
292   attr_BusTimTbl: B → BusTimTbl
293   attr_BPos: B → BPos
```

2.3.3.6 Private Automobile Attributes

We illustrate but a few attributes:

295. Automobiles have static number plate registration numbers.
296. Automobiles have dynamic positions on the road net:
    [293a] either *at a hub* identified by some h_ui,

[293b] 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.

```
type
295   RegNo
296   APos   == atHub | onLink
293a  atHub    :: h_ui:H_UI
293b  onLink   :: fh_ui:H_UI × l_ui:L_UI × frac:Fract × th_ui:H_UI
293b  Fract     = Real, axiom frac:Fract • 0<frac<1
value
295   attr_RegNo: A → RegNo
296   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 290 on the preceding page, page 66. Observe then that buses each have their own distinct *bus time table*, and that these are model-led as *inert*, Item 292 on the facing page, page 66. In Items 284 Pg. 64 and 288 Pg. 65, 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.[17]

2.3.3.7 Intentionality

297. 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;
298. seen from the point of view of a hub there is its own traffic history, H_Traffic Item 284 Pg. 64, which is a (time ordered) sequence of timed maps from automobile identities into automobile positions; and
299. seen from the point of view of a link there is its own traffic history, L_Traffic Item 288 Pg. 65, which is a (time ordered) sequence of timed maps from automobile identities into automobile positions.

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

300. 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
297   A_Hi  = (𝕋 × APos)*
284   H_Trf = A_UI →m (𝕋𝕀𝕄𝔼 × APos)*
288   L_Trf = A_UI →m (𝕋𝕀𝕄𝔼×APos)*
300   AllATH=𝕋𝕀𝕄𝔼 →m(AUI →mAPos)
```

[17] 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.

```
300  AllHTH=TIME →m (AUI →m APos)
300  AllLTH =TIME →m (AUI →m APos)
axiom
300  let allA=mrg_AllATH({(a,attr_A_Hi(a))|a:A•a ∈ as}),
300      allH=mrg_AllHTH({attr_H_Trf(h)|h:H•h ∈ hs}),
300      allL =mrg_AllLTH({attr_L_Trf(l)|l:L•h ∈ ls}) in
300  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[18]. 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* ■

2.4 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*.

State Values versus State Variables: Item 247 on page 60 expresses the **value** of all parts of a road transport system:

247. *ps*:(UoB|H|L|BC|B|A)-**set** ≡ *rts*∪*hls*∪*bcs*∪*bs*∪*as*.

301. We now introduce the set of variables, one for each part value of the domain being modeled.

301. { **variable** *vp*:(UoB|H|L|BC|B|A) | *vp*:(UoB|H|L|BC|B|A) • *vp*∈*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.

2.4.1 Channels and Communication

2.4.1.1 Channel Message Types

We ascribe types to the messages offered on channels.

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

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

```
type
303 H_L_Msg, L_H_Msg
302 HL_Msg = H_L_Msg | L_F_Msg
304 BC_B_Msg = T × BusTimTbl
305 V_R_Msg = T × (BPos|APos)
```

2.4.1.2 Channel Declarations

306. This justifies the channel declaration which is calculated to be:

```
channel
306  { hl_ch[ h_ui,l_ui ]:H_L_Msg
306       | h_ui:H_UI,l_ui:L_UI•i ∈ h_uis∧j ∈ lh_uim(h_ui) }
306  ∪
306  { hl_ch[ h_ui,l_ui ]:L_H_Msg
306       | h_ui:H_UI,l_ui:L_UI•l_ui ∈ l_uis∧i ∈ lh_uim(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.

307. This justifies the channel declaration which is calculated to be:

```
channel
307  { bc_b_ch[ bc_ui,b_ui ] | bc_ui:BC_UI, b_ui:B_UI
307    • bc_ui ∈ bc_uis ∧ b_ui ∈ b_uis }: 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.

308. This justifies the channel declaration which is calculated to be:

```
channel
308  { v_r_ch[ v_ui,r_ui ] | v_ui:V_UI,r_ui:R_UI
308    • v_ui∈ v_uis∧r_ui∈ r_uis }: V_R_Msg
```

2.4.2 Behaviours

2.4.2.1 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.

2.4.2.1.1 Hub Behaviour Signature

309. $\mathsf{hub}_{h_{ui}}$:
    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
309   hub_{h_ui}:
309a  h_ui:H_UI×(vuis,luis,_):H_Mer×HΩ
309b    → (HΣ×H_Traffic)
309c    → in,out { h_l_ch[ h_ui,l_ui ] | l_ui:L_UI•l_ui ∈ luis }
309d         { ba_r_ch[ h_ui,v_ui ] | v_ui:V_UI•v_ui∈vuis } Unit
309a     pre: vuis = v_ui s ∧ luis = l_ui s
```

2.4.2.1.2 Link Behaviour Signature

310. $\mathsf{link}_{l_{ui}}$:
    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
310   link_{l_ui}:
310a  l_ui:L_UI×(vuis,huis,_):L_Mer×LΩ
310b    → (LΣ×L_Traffic)
310c    → in,out { h_l_ch[ h_ui,l_ui ] | h_ui:H_UI:h_ui ∈ huis }
310d         { ba_r_ch[ l_ui,v_ui ] | v_ui:(B_UI|A_UI)•v_ui∈vuis } Unit
310a    pre: vuis = v_ui s ∧ huis = h_ui s
```

2.4.2.1.3 Bus Company Behaviour Signature

311. $\mathsf{bus_company}_{bc_{ui}}$:
    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
311   bus_company_{bc_ui}:
311a  bc_ui:BC_UI×(_,_,buis):BC_Mer
311b    → BusTimTbl
311c       in,out {bc_b_ch[ bc_ui,b_ui ]|b_ui:B_UI•b_ui∈buis} Unit
311a    pre: buis = b_ui s ∧ huis = h_ui s
```

2.4.2.1.4 Bus Behaviour Signature

312. $\text{bus}_{b_{ui}}$:
    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
312   bus_{b_ui}:
312a   b_ui:B_UI×(bc_ui,_,ruis):B_Mer
312b     → (LN × BTT × BPOS)
312c     → out bc_b_ch[bc_ui,b_ui],
312d          {ba_r_ch[r_ui,b_ui]|r_ui:(H_UI|L_UI)•ui∈v_ui s} Unit
312a     pre: ruis = r_ui s ∧ bc_ui ∈ bc_ui s
```

2.4.2.1.5 Automobile Behaviour Signature

313. $\text{automobile}_{a_{ui}}$:
    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
313   automobile_{a_ui}:
313a   a_ui:A_UI×(_,_,ruis):A_Mer×rn:RegNo
313b     → apos:APos
313c       in,out {ba_r_ch[a_ui,r_ui]|r_ui:(H_UI|L_UI)•r_ui∈ruis} Unit
313a     pre: ruis = r_ui s ∧ a_ui ∈ a_ui s   □
```

2.4.2.2 Behaviour Definitions

We only illustrate automobile, hub and link behaviours.

2.4.2.2.1 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.

314. We abstract automobile behaviour at a Hub (hui).
315. The vehicle remains at that hub, "idling",
316. informing the hub behaviour,
317. 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,

318. or, again internally non-deterministically,
319. the vehicle “disappears — off the radar” !

```
314  automobile_aui(a_ui,({},(ruis,vuis),{}),rn)
314          (apos:atH(fl_ui,h_ui,tl_ui)) ≡
315      (ba_r_ch[a_ui,h_ui] ! (record_TIME(),atH(fl_ui,h_ui,tl_ui));
316       automobile_aui(a_ui,({},(ruis,vuis),{}),rn)(apos))
317     ⊓
317a    (let ({fh_ui,th_ui},ruis')=mereo_L(℘(tl_ui)) in
317a          assert: fh_ui=h_ui ∧ ruis=ruis'
314     let onl = (tl_ui,h_ui,0,th_ui) in
317b     (ba_r_ch[a_ui,h_ui] ! (record_TIME(),onL(onl)) ||
317b      ba_r_ch[a_ui,tl_ui] ! (record_TIME(),onL(onl))) ;
317c      automobile_aui(a_ui,({},(ruis,vuis),{}),rn)
317c            (onL(onl)) end end)
318     ⊓
319      stop
```

2.4.2.2.2 Automobile Behaviour On a Link

320. 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” !

```
320     automobile_aui(a_ui,({},ruis,{}),rno)
320                   (vp:onL(fh_ui,l_ui,f,th_ui)) ≡
320(a)ii   (ba_r_ch[thui,aui]!atH(lui,thui,nxt_lui) ;
320(a)i     automobile_aui(a_ui,({},ruis,{}),rno)(vp))
320b     ⊓
320(b)i   (if not_yet_at_hub(f)
320(b)i      then
320(b)i1       (let incr = increment(f) in
314             let onl = (tl_ui,h_ui,incr,th_ui) in
```

```
320(b)i2       ba−r_ch[l_ui,a_ui] ! onL(onl) ;
320(b)i3       automobile_a_ui(a_ui,({},ruis,{}),rno)
320(b)i3                  (onL(onl))
320(b)i       end end)
320(b)ii    else
320(b)ii1     (let nxt_lui:L_UI•nxt_lui ∈ mereo_H(℘(th_ui)) in
320(b)ii2      ba_r_ch[thui,aui]!atH(l_ui,th_ui,nxt_lui) ;
320(b)ii3      automobile_a_ui(a_ui,({},ruis,{}),rno)
320(b)ii3                 (atH(l_ui,th_ui,nxt_lui)) end)
320(b)i    end)
320c   ⊓
320d      stop
320(b)i1  increment: Fract → Fract
```

2.4.2.2.3 Hub Behaviour

321. 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 284 Pg. 64.

```
value
321  hub_h_ui(h_ui,(,(luis,vuis)),hω)(hσ,ht) ≡
321a    ⫿
321b      { let m = ba_r_ch[h_ui,v_ui] ? in
321c           assert: m=(__,atHub(__,h_ui,__))
321d         let ht′ = ht † [h_ui ↦ ⟨m⟩⌢ht(h_ui)] in
321e         hub_h_ui(h_ui,(,(luis,vuis)),(hω))(hσ,ht′)
321f      | v_ui:V_UI•v_ui∈vuis end end }
321g    post: ∀ v_ui:V_UI•v_ui ∈ dom ht′⇒time_ordered(ht′(v_ui))
```

2.4.2.2.4 Link Behaviour

322. The link behaviour non-deterministically, externally offers
323. to accept timed vehicle positions —
324. which will be on the link, from some vehicle, v_ui.
325. The timed vehicle link position is appended to the front of that vehicle's entry in the link's traffic table;
326. whereupon the link proceeds as a link behaviour with the updated link traffic table.
327. The link behaviour offers to accept from any vehicle.
328. 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 288 Pg. 65.

```
322  link_l_ui(l_ui,(_,(huis,vuis),_),lω)(lσ,lt) ≡
322    ⌈⌉
323      { let m = ba_r_ch[ l_ui,v_ui ] ? in
324            assert: m=(_,onLink(_,l_ui,_,_))
325         let lt′ = lt † [ l_ui ↦ ⟨m⟩⌢lt(l_ui) ] in
326         link_l_ui(l_ui,(huis,vuis),hω)(hσ,lt′)
327      | v_ui:V_UI•v_ui∈vuis end end }
328    post: ∀ v_ui:V_UI•v_ui ∈ dom lt′⇒time_ordered(lt′(v_ui))
```

2.5 System Initialisation

2.5.1 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))))
```

2.5.2 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.

329. 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
329   initial_system: Unit → Unit
329   initial_system() ≡
329b     ‖ { hub_h_ui(h_ui,me,hω)(htrf,hσ)
329b        | h:H•h ∈ hs, h_ui:H_UI•h_ui=uid_H(h), me:HMetL•me=mereo_H(h),
329b          htrf:H_Traffic•htrf=attr_H_Traffic_H(h),
329b          hω:HΩ•hω=attr_HΩ(h), hσ:HΣ•hσ=attr_HΣ(h)∧hσ ∈ hω }
329a     ‖
329c     ‖ { link_l_ui(l_ui,me,lω)(ltrf,lσ)
329c          l:L•l ∈ ls, l_ui:L_UI•l_ui=uid_L(l), me:LMet•me=mereo_L(l),
329c          ltrf:L_Traffic•ltrf=attr_L_Traffic_H(l),
329c          lω:LΩ•lω=attr_LΩ(l), lσ:LΣ•lσ=attr_LΣ(l)∧lσ ∈ lω }
```

```
329a    ||
329d    || { bus_company_{bc_ui}(bcui,me)(btt)
329d         bc:BC•bc ∈ bcs, bc_ui:BC_UI•bc_ui=uid_BC(bc), me:BCMet•me=mereo_BC(bc),
329d         btt:BusTimTbl•btt=attr_BusTimTbl(bc) }
329a    ||
329e    || { bus_{b_ui}(b_ui,me)(ln,btt,bpos)
329e         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),
329e         btt:BusTimTbl•btt=attr_BusTimTbl(b), bpos:BPos•bpos=attr_BPos(b) }
329a    ||
329f    || { automobile_{a_ui}(a_ui,me,rn)(apos)
329f         a:A•a ∈ as, a_ui:A_UI•a_ui=uid_A(a), me:AMet•me=mereo_A(a),
329f         rn:RegNo•rno=attr_RegNo(a), apos:APos•apos=attr_APos(a) }    □
```

Chapter 3
The Blue Skies [August 2021]

Contents

3.1 Introdution 77
3.2 Endurants 78
3.2.1 External Qualities 78
3.2.1.1 Parts and Fluids 78
3.2.1.2 The Air State 78
3.2.2 Internal Qualities 78
3.2.2.1 Unique Identifiers 78
3.2.2.1.1 Obervers 78
3.2.2.1.2 All Unique Identifiers 78
3.2.2.1.3 Axioms 78
3.2.2.2 Mereology 78
3.2.2.2.1 Obervers 78
3.2.2.2.2 Axioms 78
3.2.2.3 Attributes 78
3.3 Perdurants 78
3.3.1 Channels 78
3.3.2 Behaviours 78
3.3.3 Signatures 78
3.3.4 Definitions 78
3.3.5 System 78
3.4 Conclusion 78

3.1 Introdution

Some early work on this domain was reported in 1995 [18]. From Appendix B of [55] we "lift" Fig. B.1 Page 349, cf. Fig. 3.1 on the following page.

The aim of this chapter is to [eventually] present a model of the air traffic domain hinted at in Fig. 3.1 on the next page.

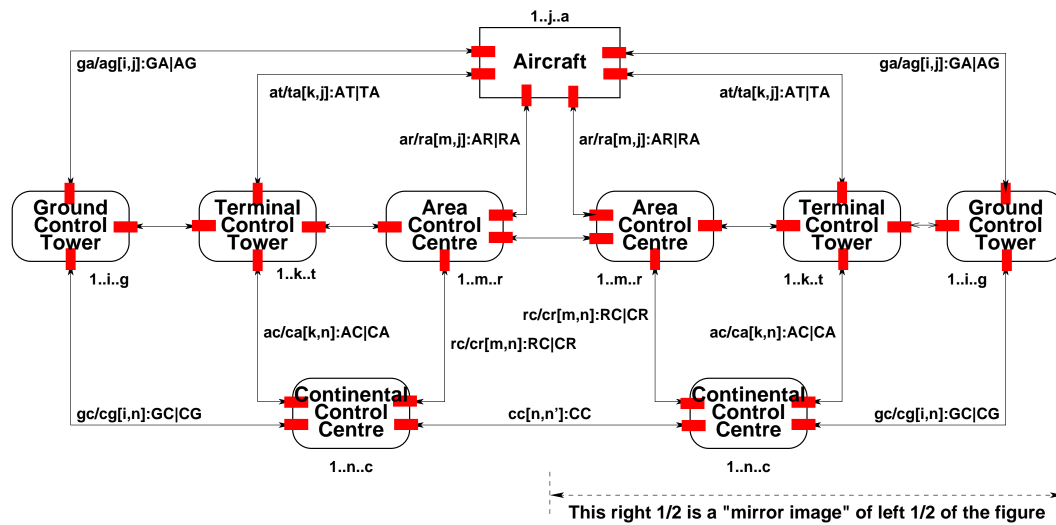


Fig. 3.1 A schematic air traffic system

3.2 Endurants

3.2.1 External Qualities

3.2.1.1 Parts and Fluids

3.2.1.2 The Air State

3.2.2 Internal Qualities

3.2.2.1 Unique Identifiers

3.2.2.1.1 Obervers

3.2.2.1.2 All Unique Identifiers

3.2.2.1.3 Axioms

3.2.2.2 Mereology

3.2.2.2.1 Obervers

3.2.2.2.2 Axioms

3.2.2.3 Attributes

3.3 Perdurants

3.3.1 Channels

3.3.2 Behaviours

3.3.3 Signatures

3.3.4 Definitions

3.3.5 System

Chapter 4
The 7 Seas [August 2021]

Contents

4.1 Introduction 80
4.2 Endurants 80
4.2.1 External Qualities 80
4.2.1.1 Informal Introduction 80
4.2.1.2 Formal Introduction 85
4.2.1.2.1 Parts and Fluids 85
4.2.1.2.2 The 7 Seas State 85
4.2.2 Internal Qualities 86
4.2.2.1 Unique Identifiers 86
4.2.2.1.1 Observers 86
4.2.2.1.2 All Unique Identifiers 86
4.2.2.1.3 Axiom 86
4.2.2.1.4 Extraction of Atomic Elements 86
4.2.2.2 Mereology 87
4.2.2.2.1 Types, Observers and Axioms 87
4.2.2.2.1.1 Seas: 87
4.2.2.2.1.2 Rivers: 87
4.2.2.2.1.3 Canals and Straits: 87
4.2.2.2.1.4 Continents: 88
4.2.2.2.1.5 Harbours: 88
4.2.2.2.1.6 Vessels: 88
4.2.2.2.2 A Remark 89
4.2.2.2.3 A Domain Axiom 89
4.2.2.3 Attributes 89
4.2.2.3.1 Seas 89
4.2.2.3.2 Rivers 90
4.2.2.3.3 Canals and Straits 90
4.2.2.3.4 Continents 91
4.2.2.3.5 Harbours 91
4.2.2.3.6 Vessels 92
4.3 Perdurants 93
4.3.1 Channels 93
4.3.2 Behaviours 93
4.3.3 Signatures 93
4.3.4 Definitions 93
4.3.5 System 93
4.4 Conclusion 93

4.1 Introduction

In this model we shall treat waterways, not as fluids, but as solids ! That is, we may considers waterways as parts, and hence, by transcendental deductions, as possibly having behaviours. Similarly we shall consider many composite endurants, not as elements of structures, but as parts, while not considering their internal qualities, that is, not considering their possible behaviours.

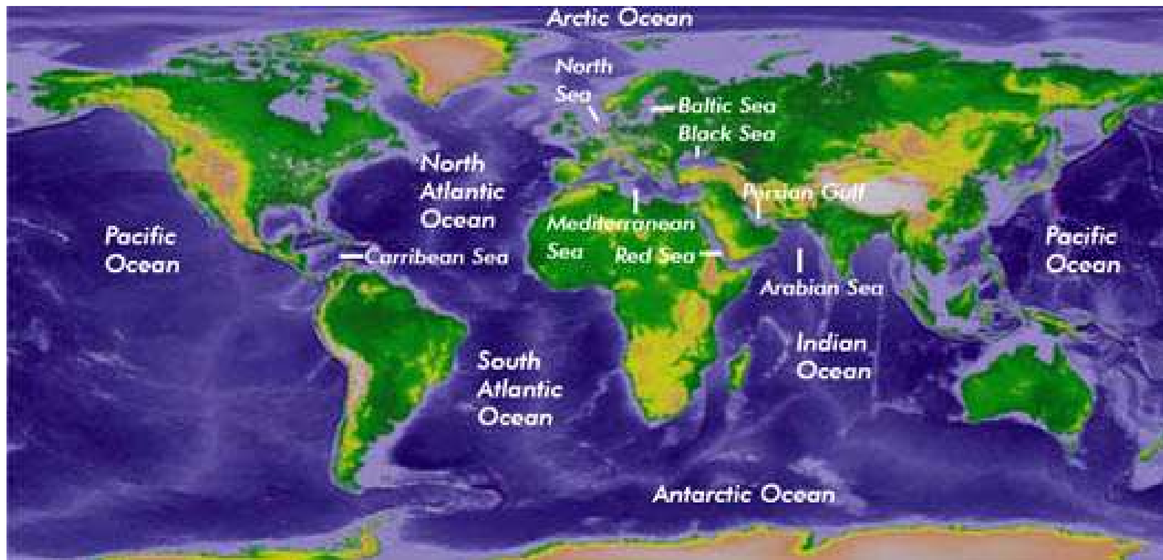
4.2 Endurants

4.2.1 External Qualities

4.2.1.1 Informal Introduction

- Waterways include seas, rivers and navigable "k"anals.
- One can take the view that there are the following eight seas: the *Arctic Ocean*, the *North Atlantic Ocean*, the *South Atlantic Ocean*, the *Indian Ocean*, the *North Pacific Ocean*, the *South Pacific Ocean*, the *Southern* (or *Antarctic*) *Ocean*, and the *Kaspian Sea*. Another view "collapses" the north and south into one, leaving just 6 oceans and seas. Yet a third view is that there are just 2 oceans and seas: The *Kaspian Sea* and the others – since they are all "tightly" connected ! The *Kaspian Sea* cannot be reached by ship or boat from the ocean[s] ! *The Mediterranean* and *The*

Black Seas are both considered segments of *The Atlantic Ocean*. *The Arab Sea* is considered a segment of *The Indian Ocean*. Etcetera.



A World Map of Oceans and Seas

The Mediterranean and Arab Seas

The Black Sea and the Kaspian Ocean

- By navigable rivers, “k”anals and status mean such rivers, “k”anals and straits that are connected to the seas and can be navigated by boats and ships. Such areas of rivers and “k”anals that are not navigable by ocean-going boats and ships are area-wise elements of “their” continents. Notice that we “lump” “k”anals and straits:

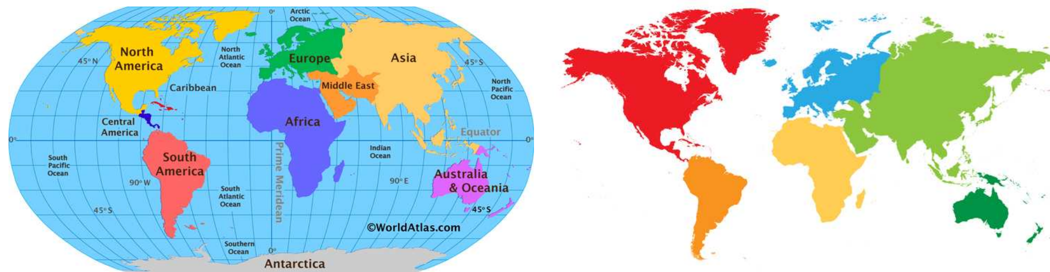
The Mississippi and the Amazon Rivers

The Yang Tse and the Danube Rivers

The Panama and Suez Canals

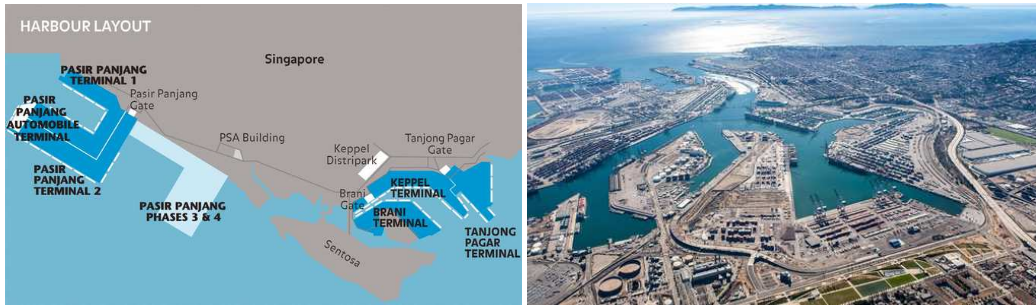
The Gibraltar and Malacca Straits

- By continents we loosely mean some connected land area.

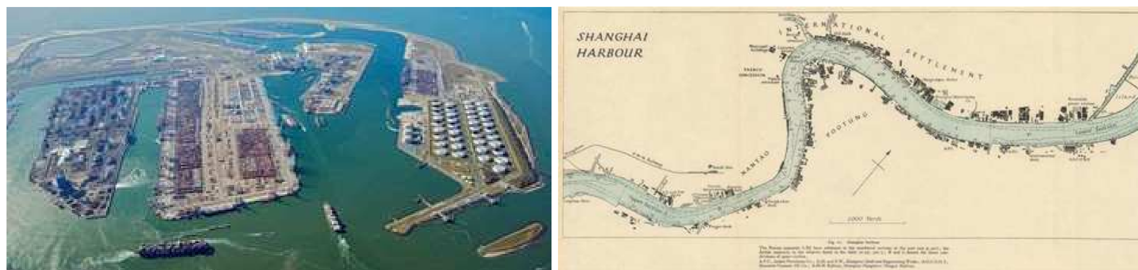


The left map counts *Central America, The Caribbean* and *Middle East* as continents !

- By harbours we mean places at the edge of continents, seas, rivers, “k”anals and straits where vessels can berth, unload and load cargo and/or passengers.

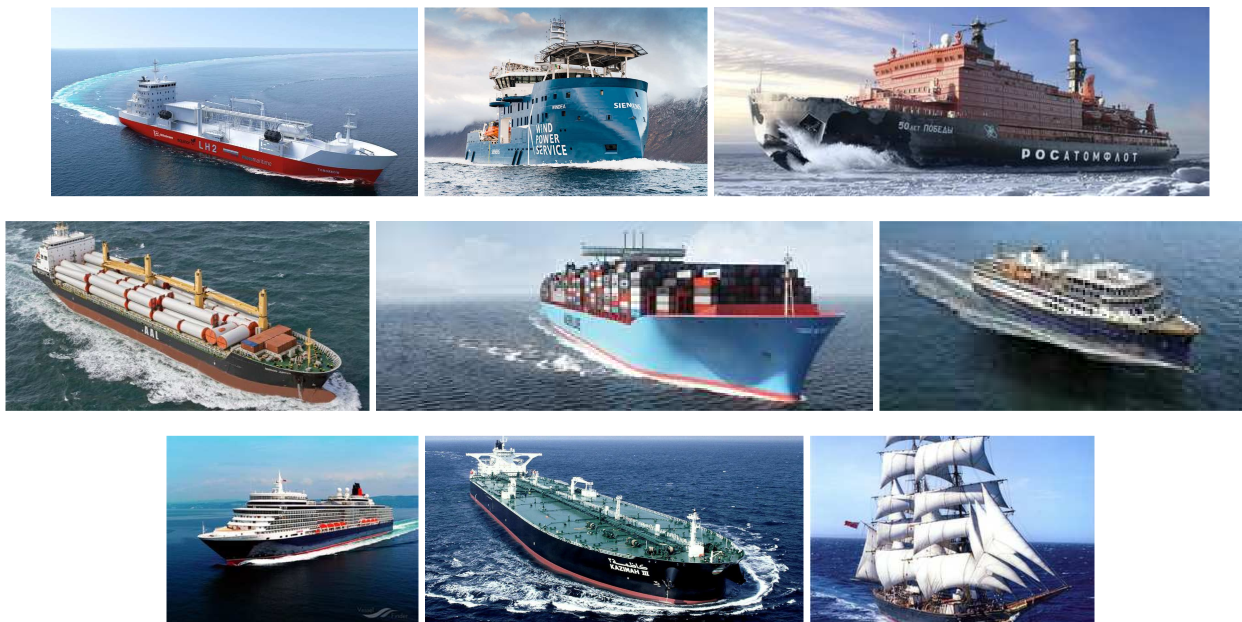


Singapore and Los Angeles Harbours



Rotterdam and Shanghai Harbours

- By vessels we mean ocean-going ships and boats. Without loss of generality we omit consideration of such vessels as floats, barges, etc.



Miscellaneous Vessels

4.2.1.2 Formal Introduction

4.2.1.2.1 Parts and Fluids

330. "The 7 Seas" is a structure composite of the waterways, the continents, the harbours and the vessels.
331. The waterways aggregate consists of an structure composite of a fluids: seas, rivers and "k"anal/straits aggregates.
332. The seas aggregate is a set of seas.
333. The rivers aggregate is a set of [atomic] rivers.
334. The "k"anal/straits aggregate is a set of [atomic] "k"anals and straits.
335. The continents aggregate is a set of [atomic] continents.
336. The harbour aggregate is a set of [atomic] harbours.
337. The Vessel aggregate is a set of [atomic] vessels.

```
type
330.  7Seas, WA, CA, HA, VA
331.  SA, RA, KA
332.  Ss = S-set
333.  Rs = R-set
334.  Ks = K-set
335.  Cs = C-set
336.  Hs = H-set
337.  Vs = V-set
value
330.  obs_WA: 7Seas→WA, obs_CA: 7Seas→CA, obs_HA: 7Seas→HA, obsVA: 7Seas→VA
331.  obs_SA: WA → SA, obs_RA: WA → RA, obs_KA: WA → KA
332.  obs_Ss: SA → Ss
333.  obs_Rs: RA → Rs
334.  obs_Ks: KA → Ks
335.  obs_Cs: CA → Cs
336.  obs_Hs: HA → Hs
337.  obs_Vs: VA → Vs
```

4.2.1.2.2 The 7 Seas State

338. By "The 7 Seas state" we mean the collection of all atomic "The 7 Seas" endurants – a collection which is the distributed union of all continents, rivers, canals, continents, harbours and vessels.

```
value
330.  7seas:7Seas
332.  ss:Ss = obs_Ss(obs_SA(obs_WA(7seas)))
333.  rs:Rs = obs_Rs(obs_RA(obs_WA(7seas)))
334.  ks:Ks = obs_Ks(obs_KA(obs_WA(7seas)))
335.  cs:Cs = obs_Cs(obs_CA(7seas))
336.  hs:Hs = obs_Hs(obs_HA(7seas))
337.  vs:Vs = obs_Vs(obs_VA(7seas))
338.  7σ:(S|R|K|C|H|V)-set = ss ∪ rs ∪ ks ∪ cs ∪ hs ∪ vs
```

Please not the *type font* names for the state values.

4.2.2 Internal Qualities

4.2.2.1 Unique Identifiers

4.2.2.1.1 Observers

339.

type
339. SI, RI, KI, CI, HI, VI
value
339. uid_S: S→SI, uid_R: R→RI, uid_K: K→KI, uid_C: C→CI, uid_H: H→HI, uid_V: V→VI

4.2.2.1.2 All Unique Identifiers

340. We can calculate the sets of all sea, river, canal, continent, harbor and vessel identifiers,
341. as well as the set of all atomic part and fluid identifiers of the 7 Seas domain.

value
340. *sis*:SI-**set** = {uid_S(s)|s:S•s ∈ *ss*}
340. *ris*:RI-**set** = {uid_R(r)|r:R•r ∈ *rs*}
340. *kis*:KI-**set** = {uid_K(k)|k:K•k ∈ *ks*}
340. *cis*:CI-**set** = {uid_C(c)|c:C•c ∈ *cs*}
340. *his*:HI-**set** = {uid_H(h)|h:H•h ∈ *hs*}
340. *vis*:VI-**set** = {uid_V(v)|v:V•v ∈ *vs*}
341. 7*is*:(SI|RI|KI|CI|HI|VI)-**set** = *sis*∪*ris*∪*kis*∪*cis*∪*his*∪*vis*

4.2.2.1.3 Axiom

342. All atomic parts and separate fluids have unique identifiers.

axiom
342. **card** 7σ = **card** ais

4.2.2.1.4 Extraction of Atomic Elements

343. From a sea identifier we can, via the domain state *ss*, obtain the seal.
344. From a river identifier we can, via the domain state *rs*, obtain the river.
345. From a canal identifier we can, via the domain state *ks*, obtain the canal.
346. From a continent identifier we can, via the domain state *cs*, obtain the continent.
347. From a harbour identifier we can, via the domain state *hs*, obtain the harbour.
348. From a vessel identifier we can, via the domain state *vs*, obtain the vessel.

value
343. xtr_S: SI→S; xtr_S(si) ≡ **let** s:S • s ∈ *ss* ∧ uid_S(s) = si **in** s **end**
344. xtr_R: RI→R; xtr_R(ri) ≡ **let** r:R • r ∈ *rs* ∧ uid_R(r) = ri **in** r **end**
345. xtr_K: KI→K; xtr_K(ki) ≡ **let** k:K • k ∈ *ks* ∧ uid_K(k) = ki **in** k **end**

346. xtr_C: CI→C; xtr_C(ci) ≡ **let** c:C • c ∈ *cs* ∧ uid_C(c) = ci **in** c **end**
347. xtr_H: HI→H; xtr_H(hi) ≡ **let** h:H • h ∈ *hs* ∧ uid_H(h) = hi **in** h **end**
348. xtr_V: VI→V; xtr_V(vi) ≡ **let** v:V • v ∈ *vs* ∧ uid_V(v) = vi **in** v **end**

4.2.2.2 Mereology

4.2.2.2.1 Types, Observers and Axioms

4.2.2.2.1.1 Seas:

349. The mereology of a sea is a triplet of the sets of unique identifiers of
    - the vessels that may sail on it,
    - the continents that borders it and
    - the harbours that confront it.

type
349. MS = VI-**set** × CI-**set** × HI-**set**
value
349. mereo_S: S → MS
axiom
349. ∀ s:S: s ∈ ss ⇒ **let** (vis,cis,his) = mereo_S(s) **in** vis ⊆ *vis* ∧ cis ⊆ *cis* ∧ his ⊆ *his* **end**

4.2.2.2.1.2 Rivers:

350. The mereology of a river is the triplet of
    - the non-empty set of unique identifiers of the continents it is embedded in,
    - the [one] unique identifier of the sea (or ocean) it is connected to, and
    - the set of unique identifiers of the vessels that may sail on that river.

type
350. MR = CI-**set** × SI × VI-**set**
value
350. mereo_R: R → MR
axiom
350. ∀ r:R: r ∈ rs ⇒ **let** (cis,si,vis) = mereo_R(r) **in** {} ≠ cis ⊆ *cis* ∧ si ∈ *sis* ∧ vis ⊆ *vis* **end**

4.2.2.2.1.3 Canals and Straits:

351. The mereology of a canal or a strait is the triplet of
    - a set of one or two unique identifiers of the seas that the canal or strait connects,
    - the set of unique identifiers of the harbours it offers,
    - the set of unique identifiers of the vessels that may sail through the canal or strait.

type
351. MK = SI-**set** × HI_set × VI-**set**
value

351. mereo_K: K → MK
axiom
351. ∀ r:K: k ∈ ks ⇒ **let** (sis,cis,vis) =
mereo_K(k) **in** 1 ≤**card** sis ≤ 2 ∧ sis ⊆ *sis* ∧ his ∈ *his* ∧ vis ⊆ *vis* **end**

4.2.2.2.1.4 Continents:

352. The mereology of a continent is the triplet of
   - the set of unique identifiers of the [other[19]] continents that the continent borders with,
   - the set of unique identifiers of the harbours on that continent, and
   - the set of unique identifiers of the rivers flowing through that continent.

type
352. MC = CI-**set** × HI_set × RI-**set**
value
352. mereo_C: C → MC
axiom
352. ∀ c:C: c ∈ *cs* ⇒ **let** (cis,his,ris) = mereo_C(c) **in** cis ⊆ *cis* ∧ his ⊆ *his* ∧ ris ⊆ *ris* **end**

4.2.2.2.1.5 Harbours:

353. The mereology of a harbour is the triplet of
   - the unique identifier of the continent to which the harbour belongs, and
   - the set of unique identifiers of the vessels that may berth at that harbour.

type
353. MH = CI × VI-**set**
value
353. mereo_H: H → MH
axiom
353. ∀ h:H • h ∈ *hs* ⇒ **let** (ci,vis) = mereo_H(j) **in** ci ∈ *cis* ∧ vis ∈ *vis* **end**

4.2.2.2.1.6 Vessels:

354. The mereology of a vessel is the pair of
   - the set of unique identifiers of the seas on which the vessel may sail, and
   - the set of unique identifiers of the harbours at which the vessel may berth,

type
354. MV = SI-**set** × HI-**set**
value
354. mereo_V: V → MV
axiom
354. ∀ v:V • v ∈ *vis* ⇒ **let** (sis,his) = mereo_V(v) **in** sis ⊆ *sis* ∧ his ⊆ *his* **end**

[19] The **axiom** (351) does not model "the other" clause !

4.2.2.2.2 A Remark

Please note that we have not [yet] had a need to describe the sea and land 𝔸ℝ𝔼𝔸s of seas and continents.

4.2.2.2.3 A Domain Axiom

The axioms of Sect. **4.2.2.2.1** pertains to the individual atomic elements of the domain, not to their occurrence in the context of the aggregates to which they are elements.

355. The mereology of a sea of a domain states the unique identifiers of the vessels that may sail on it, so we must, vice-versa, expect that the mereology of the identified vessels likewise identify that sea as one on which it may sail.

```
axiom
355.  ∀ s:S • s ∈ ss ⇒
355.     let (vis,cis,his) = mereo_S(s) in
355.     ∀ vi:VI • vi ∈ vis ⇒
355.        let v:V • v = xtr_V(vi) in
355.        let (sis,his) = mereo_V(v) in
355.        uid_S(s) ∈ sis end end end
```

We leave it to the reader to narrate and formalise similar "cross-mereology" axioms for [all other] relevant "pairs" of different sort atomic elements of the domain.

4.2.2.3 Attributes

Seas, rivers, canals, continents and harbours have spatial attributes of kind 𝕊𝕌ℝ𝔽𝔸ℂ𝔼, 𝕃𝕀ℕ𝔼 and ℙ𝕆𝕀ℕ𝕋. We refer to [55, Sect. 3.4].

4.2.2.3.1 Seas

356. We ascribe names to seas.
357. Seas spread over contiguous surface (𝕊𝕌ℝ𝔽𝔸ℂ𝔼).
358. Seas have borders/edges (𝕃𝕀ℕ𝔼).
359.
360.
361.
362.

```
type
356.  SeaName
357.  SeaSurface = 𝕊𝕌ℝ𝔽𝔸ℂ𝔼
358.  SeaBorder = 𝕃𝕀ℕ𝔼
359.
360.
361.
value
356.  attr_SeaName: S → SeaName
357.  attr_SeaSurface: S → SeaSurface
```

358. attr_SeaBorder: S → SeaBorder
359. attr_: →
360. attr_: →
361. attr_: →

4.2.2.3.2 Rivers

363.
364.
365.
366.
367.
368.
369.

type
363.
364.
365.
366.
367.
368.
value
363. attr_: →
364. attr_: →
365. attr_: →
366. attr_: →
367. attr_: →
368. attr_: →

4.2.2.3.3 Canals and Straits

370.
371.
372.
373.
374.
375.
376.

type
370.
371.
372.
373.
374.
375.
value
370. attr_: →

371. attr_: →
372. attr_: →
373. attr_: →
374. attr_: →
375. attr_: →

4.2.2.3.4 Continents

377.
378.
379.
380.
381.
382.
383.

type
377.
378.
379.
380.
381.
382.
value
377. attr_: →
378. attr_: →
379. attr_: →
380. attr_: →
381. attr_: →
382. attr_: →

4.2.2.3.5 Harbours

384.
385.
386.
387.
388.
389.
390.

type
384.
385.
386.
387.
388.
389.
value

384. attr_: →
385. attr_: →
386. attr_: →
387. attr_: →
388. attr_: →
389. attr_: →

4.2.2.3.6 Vessels

391. Vessels have names.
392. Vessels have kind: passenger, ordinary freight, crude oil, container, ...
393. Vessels, at any one "point" in time has a position.
394. Vessels, when sailing, follow a route.
395. Vessel positions are well-formed if they are on the current route.
396. Vessels have a speed
397. and a velocity.
398. A vessel is **on course** if its position (at some time) is on that vessel's route.

```
type
391.  VesselName
392.  VesselKind = ...
393.  VesselPos = 𝕋𝕀𝕄𝔼 × ℙ𝕆𝕊𝕀𝕋𝕀𝕆ℕ
394.  VesselRoute = BezierCurve
396.  VesselSpeed
396.  VesselVelocity
value
391. attr_VesselName: V → VesselName
392. attr_VesselKind: V → VesselKind
393. attr_VesselPos: V → VesselPos
394. attr_VesselRoute: V → VesselRoute
396. attr_VesselSpeed: V → Speed
397. attr_VesselVelocity: V → Velocity
398. Vessel_on_course: V → Bool
398. Vessel_on_course(v) ≡ let (vp,_) = attr_VesselPos(v) in Position_on_curve(vp,attr_VesselRoute(v)) end
398. Position_on_curve: ℙ𝕆𝕊𝕀𝕋𝕀𝕆ℕ × Bezier → Bool
```

4.3 Perdurants

4.3.1 Channels

4.3.2 Behaviours

4.3.3 Signatures

4.3.4 Definitions

4.3.5 System

4.4 Conclusion

Chapter 5
Pipelines [2008]

Fig. 5.1 The Planned Nabucco Pipeline: http://en.wikipedia.org/wiki/Nabucco_Pipeline

- Named after Verdi's opera
- Gas pipeline
- 3300 kms
- 2011–2014, first gas flow: 2014; 2017–2019, more pipes
- 8 billion Euros
- Max flow: 31 bcmy: billion cubic meters a year
- http://www.nabucco-pipeline.com/

Fig. 5.2 The Planned Nabucco Pipeline: http://en.wikipedia.org/wiki/Nabucco_Pipeline

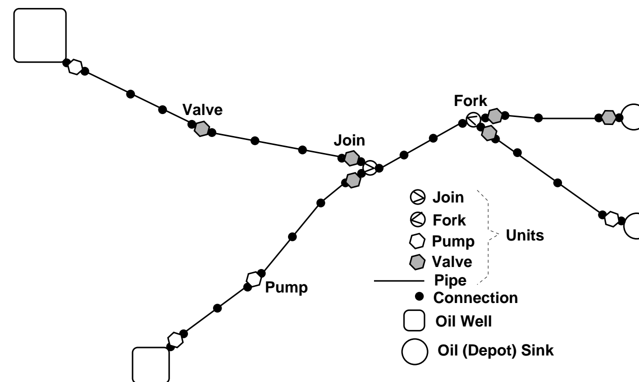


Fig. 5.3 An oil pipeline system

5.1 Photos of Pipeline Units and Diagrams of Pipeline Systems

When combining joins and forks we can construct sitches. Figure 5.7 on page 99 shows some actual switches.

Figure 5.8 on page 100 diagrams a generic switch.

5.2 Non-Temporal Aspects of Pipelines

These are some non-temporal aspects of pipelines. nets and units: wells, pumps, pipes, valves, joins, forks and sinks; net and unit attributes; and units states, but not state changes. We omit, in early (i.e., next) chapters, consideration of "pigs" and "pig"-insertion and "pig"-extraction units.

5.2.1 Nets of Pipes, Valves, Pumps, Forks and Joins

399. We focus on nets, $n:N$, of pipes, $\pi:\Pi$, valves, $v:V$, pumps, $p:P$, forks, $f:F$, joins, $j:J$, wells, $w:W$ and sinks, $s:S$.

[19] See http://en.wikipedia.org/wiki/Nabucco_Pipeline

Fig. 5.4 Pipes

Fig. 5.5 Valves

400. Units, $u : U$, are either pipes, valves, pumps, forks, joins, wells or sinks.
401. Units are explained in terms of disjoint types of PIpes, VAlves, PUmps, FOrks, JOins, WElls and SKs.[20]

type

[20] This is a mere specification language technicality.

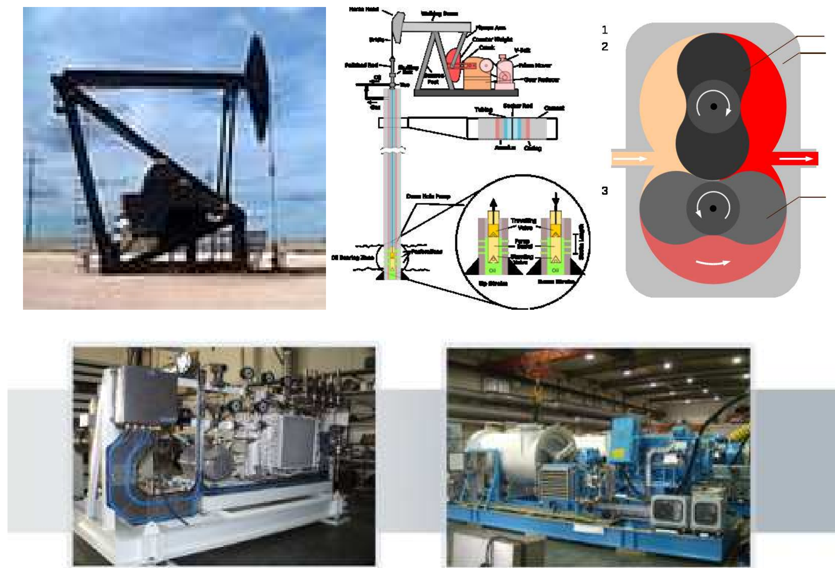


Fig. 5.6 Oil Pumps and Gas Compressors

```
399 N, PI, VA, PU, FO, JO, WE, SK
400 U = Π | V | P | F | J | S| W
400 Π == mkΠ(pi:PI)
400 V == mkV(va:VA)
400 P == mkP(pu:PU)
400 F == mkF(fo:FO)
400 J == mkJ(jo:JO)
400 W == mkW(we:WE)
400 S == mkS(sk:SK)
```

5.2.2 Unit Identifiers and Unit Type Predicates

402. We associate with each unit a unique identifier, *ui* : *UI*.
403. From a unit we can observe its unique identifier.
404. From a unit we can observe whether it is a pipe, a valve, a pump, a fork, a join, a well or a sink unit.

```
type
  402 UI
value
  403 obs_UI: U → UI
  404 is_Π: U → Bool, is_V: U → Bool, ..., is_J: U → Bool
    is_Π(u) ≡ case u of mkPI(_) → true, _ → false end
    is_V(u) ≡ case u of mkV(_) → true, _ → false end
    ...
    is_S(u) ≡ case u of mkS(_) → true, _ → false end
```

Fig. 5.7 Oil and Gas Switches

5.2.3 Unit Connections

A connection is a means of juxtaposing units. A connection may connect two units in which case one can observe the identity of connected units from “the other side”.

405. With a pipe, a valve and a pump we associate exactly one input and one output connection.
406. With a fork we associate a maximum number of output connections, *m*, larger than one.
407. With a join we associate a maximum number of input connections, *m*, larger than one.
408. With a well we associate zero input connections and exactly one output connection.
409. With a sink we associate exactly one input connection and zero output connections.

value
405 obs_InCs,obs_OutCs: Π|V|P → {|1:**Nat**|}

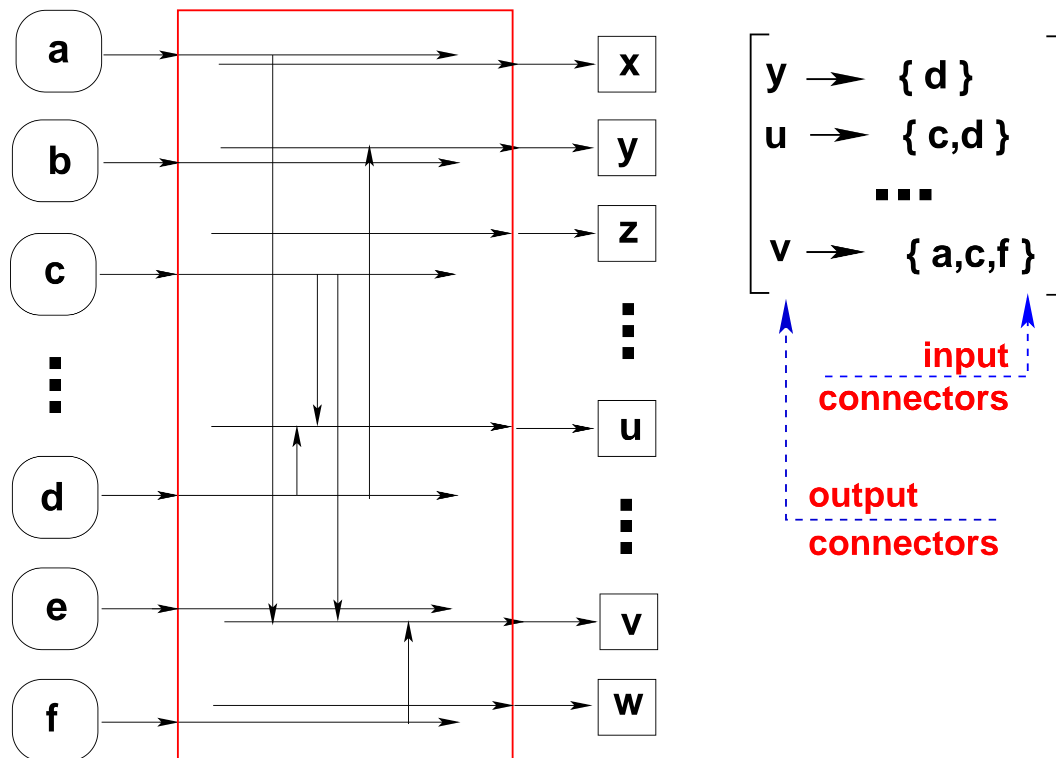


Fig. 5.8 A Switch Diagram

406 obs_inCs: F → {|1:**Nat**|}, obs_outCs: F → **Nat**
407 obs_inCs: J → **Nat**, obs_outCs: J → {|1:**Nat**|}
408 obs_inCs: W → {|0:**Nat**|}, obs_outCs: W → {|1:**Nat**|}
409 obs_inCs: S → {|1:**Nat**|}, obs_outCs: S → {|0:**Nat**|}
axiom
406 ∀ f:F • obs_outCs(f) ≥ 2
407 ∀ j:J • obs_inCs(j) ≥ 2

If a pipe, valve or pump unit is input-connected [output-connected] to zero (other) units, then it means that the unit input [output] connector has been sealed. If a fork is input-connected to zero (other) units, then it means that the fork input connector has been sealed. If a fork is output-connected to n units less than the maximum fork-connectability, then it means that the unconnected fork outputs have been sealed. Similarly for joins: "the other way around".

5.2.4 Net Observers and Unit Connections

410. From a net one can observe all its units.
411. From a unit one can observe the the pairs of disjoint input and output units to which it is connected:
    a. Wells can be connected to zero or one output unit — a pump.
    b. Sinks can be connected to zero or one input unit — a pump or a valve.

Fig. 5.9 **To be treated in a later version of this report:** Pig Launcher, Receiver and New and Old Pigs

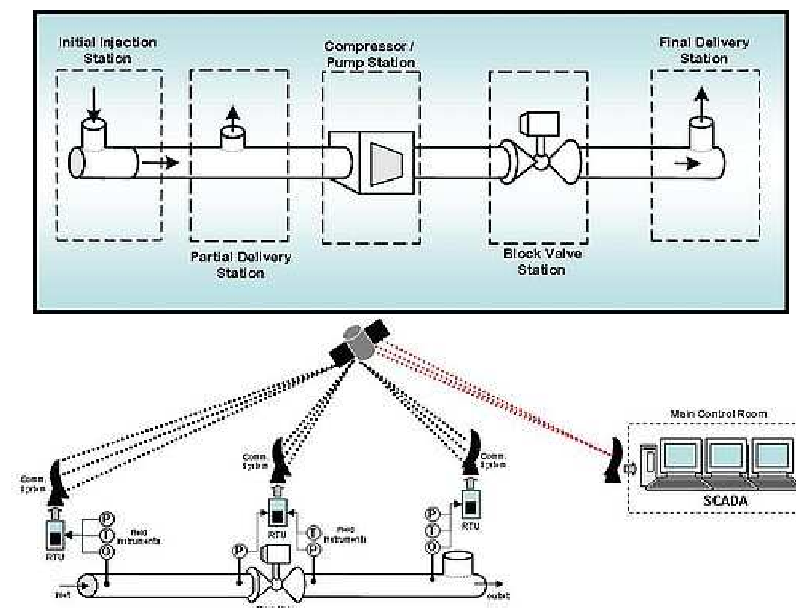


Fig. 5.10 Pipeline Diagrams

c. Pipes, valves and pumps can be connected to zero or one input units and to zero or one output units.
d. Forks, f, can be connected to zero or one input unit and to zero or n, $2 \leq n \leq$obs_Cs(f) output units.
e. Joins, j, can be connected to zero or n, $2 \leq n \leq$obs_Cs(j) input units and zero or one output units.

```
value
  410  obs_Us: N → U-set
  411  obs_cUIs: U → UI-set × UI-set
       wf_Conns: U → Bool
       wf_Conns(u) ≡
           let (iuis,ouis) = obs_cUIs(u) in iuis ∩ ouis = {} ∧
           case u of
  411a  mkW(_) → card iuis ∈ {0} ∧ card ouis ∈ {0,1},
  411b  mkS(_) → card iuis ∈ {0,1} ∧ card ouis ∈ {0},
  411c  mkΠ(_) → card iuis ∈ {0,1} ∧ card ouis ∈ {0,1},
  411c  mkV(_) → card iuis ∈ {0,1} ∧ card ouis ∈ {0,1},
  411c  mkP(_) → card iuis ∈ {0,1} ∧ card ouis ∈ {0,1},
  411d  mkF(_) → card iuis ∈ {0,1} ∧ card ouis ∈ {0}∪{2..obs_inCs(j)},
  411e  mkJ(_) → card iuis ∈ {0}∪{2..obs_inCs(j)} ∧ card ouis ∈ {0,1}
           end end
```

5.2.5 Well-formed Nets, Actual Connections

412. The unit identifiers observed by the obs_cUIs observer must be identifiers of units of the net.

```
axiom
  412  ∀ n:N,u:U • u ∈ obs_Us(n) ⇒
  412    let (iuis,ouis) = obs_cUIs(u) in
  412    ∀ ui:UI • ui ∈ iuis ∪ ouis ⇒
  412      ∃ u′:U • u′ ∈ obs_Us(n) ∧ u′≠u ∧ obs_UI(u′)=ui end
```

5.2.6 Well-formed Nets, No Circular Nets

413. By a route we shall understand a sequence of units.
414. Units form routes of the net.

```
type
  413  R = UI^ω
value
  414  routes: N → R-infset
  414  routes(n) ≡
  414    let us = obs_Us(n) in
  414    let rs = {⟨u⟩|u:U•u ∈ us} ∪ {r⌢r′|r,r′:R• {r,r′}⊆rs∧adj(r,r′)} in
  414    rs end end
```

415. A route of length two or more can be decomposed into two routes
416. such that the least unit of the first route “connects” to the first unit of the second route.

```
value
  415    adj: R × R → Bool
  415    adj(fr,lr) ≡
  415      let (lu,fu)=(fr(len fr),hd lr) in
```

```
416      let (lui,fui)=(obs_UI(lu),obs_UI(fu)) in
416      let ((_,luis),(fuis,_))=(obs_cUIs(lu),obs_cUIs(fu)) in
416      lui ∈ fuis ∧ fui ∈ luis end end end
```

417. No route must be circular, that is, the net must be acyclic.

```
value
  417  acyclic: N → Bool
  417    let rs = routes(n) in
  417    ∼∃ r:R•r ∈ rs⇒∃ i,j:Nat•{i,j}⊆inds r∧i≠j∧r(i)=r(j) end
```

5.2.7 Well-formed Nets, Special Pairs, wfN_SP

418. We define a "special-pairs" well-formedness function.
   a. Fork outputs are output-connected to valves.
   b. Join inputs are input-connected to valves.
   c. Wells are output-connected to pumps.
   d. Sinks are input-connected to either pumps or valves.

```
value
  418  wfN_SP: N → Bool
  418  wfN_SP(n) ≡
  418    ∀ r:R • r ∈ routes(n) in
  418      ∀ i:Nat • {i,i+1}⊆inds r ⇒
  418        case r(i) of ∧
  418a         mkF(_) → ∀ u:U•adj(⟨r(i)⟩,⟨u⟩) ⇒ is_V(u),_→true end ∧
  418        case r(i+1) of
  418b         mkJ(_) → ∀ u:U•adj(⟨u⟩,⟨r(i)⟩) ⇒ is_V(u),_→true end ∧
  418        case r(1) of
  418c         mkW(_) → is_P(r(2)),_→true end ∧
  418        case r(len r) of
  418d         mkS(_) → is_P(r(len r−1))∨is_V(r(len r−1)),_→true end
```

The **true** clauses may be negated by other **case** distinctions' is_V or is_V clauses.

5.2.8 Special Routes, I

419. A pump-pump route is a route of length two or more whose first and last units are pumps and whose intermediate units are pipes or forks or joins.
420. A simple pump-pump route is a pump-pump route with no forks and joins.
421. A pump-valve route is a route of length two or more whose first unit is a pump, whose last unit is a valve and whose intermediate units are pipes or forks or joins.
422. A simple pump-valve route is a pump-valve route with no forks and joins.
423. A valve-pump route is a route of length two or more whose first unit is a valve, whose last unit is a pump and whose intermediate units are pipes or forks or joins.
424. A simple valve-pump route is a valve-pump route with no forks and joins.

425. A valve-valve route is a route of length two or more whose first and last units are valves and whose intermediate units are pipes or forks or joins.
426. A simple valve-valve route is a valve-valve route with no forks and joins.

```
value
  419-426 ppr,sppr,pvr,spvr,vpr,svpr,vvr,svvr: R → Bool
          pre {ppr,sppr,pvr,spvr,vpr,svpr,vvr,svvr}(n): len n≥2

  419 ppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_P(fu) ∧ is_P(lu) ∧ is_πfjr(ℓ)
  420 sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)
  421 pvr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_P(fu) ∧ is_V(r(len r)) ∧ is_πfjr(ℓ)
  422 sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)
  423 vpr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_V(fu) ∧ is_P(lu) ∧ is_πfjr(ℓ)
  424 sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)
  425 vvr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_V(fu) ∧ is_V(lu) ∧ is_πfjr(ℓ)
  426 sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)

  is_πfjr,is_πr: R → Bool
  is_πfjr(r) ≡ ∀ u:U•u ∈ elems r⇒is_Π(u)∨is_F(u)∨is_J(u)
  is_πr(r) ≡ ∀ u:U•u ∈ elems r⇒is_Π(u)
```

5.2.9 Special Routes, II

Given a unit of a route,
427. if they exist (∃),
428. find the nearest pump or valve unit,
429. “upstream” and
430. “downstream” from the given unit.

```
value
  427 ∃UpPoV: U × R → Bool
  427 ∃DoPoV: U × R → Bool
  429 find_UpPoV: U × R ~→ (P|V), pre find_UpPoV(u,r): ∃UpPoV(u,r)
  430 find_DoPoV: U × R ~→ (P|V), pre find_DoPoV(u,r): ∃DoPoV(u,r)
  427 ∃UpPoV(u,r) ≡
  427   ∃ i,j Nat•{i,j}⊆inds r∧i≤j∧{is_V|is_P}(r(i))∧u=r(j)
  427 ∃DoPoV(u,r) ≡
  427   ∃ i,j Nat•{i,j}⊆inds r∧i≤j∧u=r(i)∧{is_V|is_P}(r(j))
  429 find_UpPoV(u,r) ≡
  429   let i,j:Nat•{i,j}⊆indsr∧i≤j∧{is_V|is_P}(r(i))∧u=r(j) in r(i) end
  430 find_DoPoV(u,r) ≡
  430   let i,j:Nat•{i,j}⊆indsr∧i≤j∧u=r(i)∧{is_V|is_P}(r(j)) in r(j) end
```

5.3 State Attributes of Pipeline Units

By a state attribute of a unit we mean either of the following three kinds: (i) the open/close states of valves and the pumping/not_pumping states of pumps; (ii) the maximum (laminar) oil flow characteristics of all units; and (iii) the current oil flow and current oil leak states of all units.

431. Oil flow, $\phi : \Phi$, is measured in volume per time unit.
432. Pumps are either pumping or not pumping, and if not pumping they are closed.
433. Valves are either open or closed.
434. Any unit permits a maximum input flow of oil while maintaining laminar flow. We shall assume that we need not be concerned with turbulent flows.
435. At any time any unit is sustaining a current input flow of oil (at its input(s)).
436. While sustaining (even a zero) current input flow of oil a unit leaks a current amount of oil (within the unit).

```
type
  431  Φ
  432  PΣ == pumping | not_pumping
  432  VΣ == open | closed
value
        −,+: Φ × Φ → Φ, <,=,>: Φ × Φ → Bool
  432    obs_PΣ: P → PΣ
  433    obs_VΣ: V → VΣ
  434–436  obs_LamiΦ.obs_CurrΦ,obs_LeakΦ: U → Φ
  is_Open: U → Bool
    case u of
      mkΠ(_)→true,mkF(_)→true,mkJ(_)→true,mkW(_)→true,mkS(_)→true,
      mkP(_)→obs_PΣ(u)=pumping,
      mkV(_)→obs_VΣ(u)=open
    end
  acceptable_LeakΦ, excessive_LeakΦ: U → Φ
axiom
  ∀ u:U • excess_LeakΦ(u) > accept_LeakΦ(u)
```

5.3.1 Flow Laws

The sum of the current flows into a unit equals the the sum of the current flows out of a unit minus the (current) leak of that unit. This is the same as the current flows out of a unit equals the current flows into a unit minus the (current) leak of that unit. The above represents an interpretation which justifies the below laws.

437. When, in Item 435, for a unit u, we say that at any time any unit is sustaining a current input flow of oil, and when we model that by obs_CurrΦ(u) then we mean that obs_CurrΦ(u) - obs_LeakΦ(u) represents the flow of oil from its outputs.

```
value
  437   obs_inΦ: U → Φ
  437   obs_inΦ(u) ≡ obs_CurrΦ(u)
  437   obs_outΦ: U → Φ
law:
  437   ∀ u:U • obs_outΦ(u) = obs_CurrΦ(u)−obs_LeakΦ(u)
```

438. Two connected units enjoy the following flow relation:
   a. If
      i. two pipes, or
      ii. a pipe and a valve, or
      iii. a valve and a pipe, or
      iv. a valve and a valve, or
      v. a pipe and a pump, or
      vi. a pump and a pipe, or
      vii. a pump and a pump, or
      viii. a pump and a valve, or
      ix. a valve and a pump

      are immediately connected
   b. then
      i. the current flow out of the first unit's connection to the second unit
      ii. equals the current flow into the second unit's connection to the first unit

law:

438a ∀ u,u′:U • {is_Π,is_V,is_P,is_W}(u′|u″) ∧ adj(⟨u⟩,⟨u′⟩)
438a is_Π(u)∨is_V(u)∨is_P(u)∨is_W(u) ∧
438a is_Π(u′)∨is_V(u′)∨is_P(u′)∨is_S(u′)
438b ⇒ obs_outΦ(u)=obs_inΦ(u′)

A similar law can be established for forks and joins. For a fork output-connected to, for example, pipes, valves and pumps, it is the case that for each fork output the out-flow equals the in-flow for that output-connected unit. For a join input-connected to, for example, pipes, valves and pumps, it is the case that for each join input the in-flow equals the out-flow for that input-connected unit. We leave the formalisation as an exercise.

5.3.2 Possibly Desirable Properties

439. Let r be a route of length two or more, whose first unit is a pump, p, whose last unit is a valve, v and whose intermediate units are all pipes: if the pump, p is pumping, then we expect the valve, v, to be open.
440. Let r be a route of length two or more, whose first unit is a pump, p, whose last unit is another pump, p' and whose intermediate units are all pipes: if the pump, p is pumping, then we expect pump p'', to also be pumping.
441. Let r be a route of length two or more, whose first unit is a valve, v, whose last unit is a pump, p and whose intermediate units are all pipes: if the valve, v is closed, then we expect pump p, to not be pumping.
442. Let r be a route of length two or more, whose first unit is a valve, v', whose last unit is a valve, v'' and whose intermediate units are all pipes: if the valve, v' is in some state, then we expect valve v'', to also be in the same state.

desirable properties:

439 ∀ r:R • spvr(r) ∧
439 **spvr_prop(r):** obs_PΣ(**hd** r)=pumping ⇒ obs_PΣ(r(**len** r))=open

440 ∀ r:R • sppr(r) ∧
440 **sppr_prop(r):** obs_PΣ(**hd** r)=pumping⇒obs_PΣ(r(**len** r))=pumping

441 ∀ r:R • svpr(r) ∧
441 **svpr_prop(r):** obs_PΣ(**hd** r)=open⇒obs_PΣ(r(**len** r))=pumping

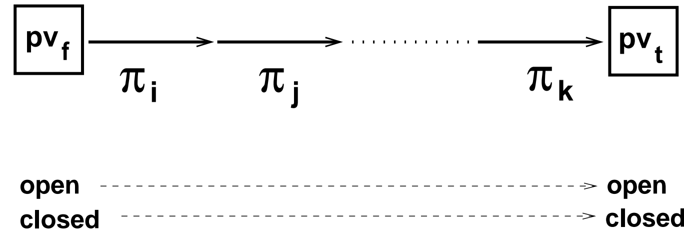


Fig. 5.11 pv: Pump or valve, π: pipe

```
442 ∀ r:R • svvr(r) ∧
442   svvr_prop(r): obs_PΣ(hd r)=obs_PΣ(r(len r))
```

5.4 Pipeline Actions

5.4.1 Simple Pump and Valve Actions

443. Pumps may be set to pumping or reset to not pumping irrespective of the pump state.
444. Valves may be set to be open or to be closed irrespective of the valve state.
445. In setting or resetting a pump or a valve a desirable property may be lost.

```
value
  443 pump_to_pump, pump_to_not_pump: P → N → N
  444 valve_to_open, valve_to_close: V → N → N
```

```
value
  443 pump_to_pump(p)(n) as n′
  443   pre p ∈ obs_Us(n)
  443   post let p′:P•obs_UI(p)=obs_UI(p′) in
  443        obs_PΣ(p′)=pumping∧else_equal(n,n′)(p,p′) end
  443 pump_to_not_pump(p)(n) as n′
  443   pre p ∈ obs_Us(n)
  443   post let p′:P•obs_UI(p)=obs_UI(p′) in
  443        obs_PΣ(p′)=not_pumping∧else_equal(n,n′)(p,p′) end
  444 valve_to_open(v)(n) as n′
  443   pre v ∈ obs_Us(n)
  444   post let v′:V•obs_UI(v)=obs_UI(v′) in
  443        obs_VΣ(v′)=open∧else_equal(n,n′)(v,v′) end
  444 valve_to_close(v)(n) as n′
  443   pre v ∈ obs_Us(n)
  444   post let v′:V•obs_UI(v)=obs_UI(v′) in
  443        obs_VΣ(v′)=close∧else_equal(n,n′)(v,v′) end
```

```
value
  else_equal: (N×N) → (U×U) → Bool
  else_equal(n,n′)(u,u′) ≡
```

$$\begin{array}{l} \quad \text{obs_UI}(u)=\text{obs_UI}(u') \\ \wedge\ u \in \text{obs_Us}(n) \wedge u' \in \text{obs_Us}(n') \\ \wedge\ \text{omit_}\Sigma(u)=\text{omit_}\Sigma(u') \\ \wedge\ \text{obs_Us}(n)\backslash\{u\}=\text{obs_Us}(n)\backslash\{u'\} \\ \wedge\ \forall\ u'':U \bullet u'' \in \text{obs_Us}(n)\backslash\{u\} \equiv u'' \in \text{obs_Us}(n')\backslash\{u'\} \end{array}$$

$\text{omit_}\Sigma$: U $\rightarrow$ U_{no_state} --- "magic" function

=: $U_{no_state} \times U_{no_state} \rightarrow$ **Bool**

axiom

$\forall$ u,u′:U•omit_Σ(u)=omit_Σ(u′) $\equiv$ obs_UI(u)=obs_UI(u′)

5.4.2 Events

5.4.2.1 Unit Handling Events

446. Let n be any acyclic net.
446. If there exists p,p',v,v', pairs of distinct pumps and distinct valves of the net,
446. and if there exists a route, r, of length two or more of the net such that
447. all units, u, of the route, except its first and last unit, are pipes, then
448. if the route "spans" between p and p' and the *simple desirable property*, sppr(r), does not hold for the route, then we have a possibly undesirable event — that occurred as soon as sppr(r) did not hold;
449. if the route "spans" between p and v and the *simple desirable property*, spvr(r), does not hold for the route, then we have a possibly undesirable event;
450. if the route "spans" between v and p and the *simple desirable property*, svpr(r), does not hold for the route, then we have a possibly undesirable event; and
451. if the route "spans" between v and v' and the *simple desirable property*, svvr(r), does not hold for the route, then we have a possibly undesirable event.

events:

```
446 ∀ n:N • acyclic(n) ∧
446   ∃ p,p′:P,v,v′:V • {p,p′,v,v′}⊆obs_Us(n)⇒
446     ∧ ∃ r:R • routes(n) ∧
447       ∀ u:U • u ∈ elems(r)\{hd r,r(len r)} ⇒ is_Π(i) ⇒
448         p=hd r∧p′=r(len r) ⇒ ∼sppr_prop(r) ∧
449         p=hd r∧v=r(len r) ⇒ ∼spvr_prop(r) ∧
450         v=hd r∧p=r(len r) ⇒ ∼svpr_prop(r) ∧
451         v=hd r∧v′=r(len r) ⇒ ∼svvr_prop(r)
```

5.4.2.2 Foreseeable Accident Events

A number of foreseeable accidents may occur.

452. A unit ceases to function, that is,
    a. a unit is clogged,
    b. a valve does not open or close,

c. a pump does not pump or stop pumping.

453. A unit gives rise to excessive leakage.
454. A well becomes empty or a sunk becomes full.
455. A unit, or a connected net of units gets on fire.
456. Or a number of other such “accident”.

5.4.3 Well-formed Operational Nets

457. A well-formed operational net
458. is a well-formed net
    a. with at least one well, w, and at least one sink, s,
    b. and such that there is a route in the net between w and s.

```
value
  457  wf_OpN: N → Bool
  457  wf_OpN(n) ≡
  458    satisfies axiom 412 on page 102 ∧ acyclic(n): Item 417 on page 103 ∧
  458    wfN_SP(n): satisfies flow laws, 437 on page 105 and 438 on page 106 ∧
  458a    ∃ w:W,s:S • {w,s}⊆obs_Us(n) ⇒
  458b      ∃ r:R• ⟨w⟩⁀r⁀⟨s⟩ ∈ routes(n)
```

5.4.4 Orderly Action Sequences

5.4.4.1 Initial Operational Net

459. Let us assume a notion of an initial operational net.
460. Its pump and valve units are in the following states
    a. all pumps are not_pumping, and
    b. all valves are closed.

```
value
  459  initial_OpN: N → Bool
  460  initial_OpN(n) ≡ wf_OpN(n) ∧
  460a    ∀ p:P • p ∈ obs_Us(n) ⇒ obs_PΣ(p)=not_pumping ∧
  460b    ∀ v:V • v ∈ obs_Us(n) ⇒ obs_VΣ(p)=closed
```

5.4.4.2 Oil Pipeline Preparation and Engagement

461. We now wish to prepare a pipeline from some well, $w : W$, to some sink, $s : S$, for flow.
    a. We assume that the underlying net is operational wrt. w and s, that is, that there is a route, r, from w to s.
    b. Now, an orderly action sequence for engaging route r is to “work backwards”, from s to w
    c. setting encountered pumps to pumping and valves to open.

In this way the system is well-formed wrt. the desirable sppr, spvr, svpr and svvr properties. Finally, setting the pump adjacent to the (preceding) well starts the system.

```
value
  461   prepare_and_engage: W × S → N ~→ N
  461   prepare_and_engage(w,s)(n) ≡
  461a    let r:R • ⟨w⟩⁀r⁀⟨s⟩ ∈ routes(n) in
  461b    action_sequence(⟨w⟩⁀r⁀⟨s⟩)(len⟨w⟩⁀r⁀⟨s⟩)(n) end
  461    pre ∃ r:R • ⟨w⟩⁀r⁀⟨s⟩ ∈ routes(n)

  461c  action_sequence: R → Nat → N → N
  461c  action_sequence(r)(i)(n) ≡
  461c    if i=1 then n else
  461c    case r(i) of
  461c      mkV(_) → action_sequence(r)(i−1)(valve_to_open(r(i))(n)),
  461c      mkP(_) → action_sequence(r)(i−1)(pump_to_pump(r(i))(n)),
  461c      _ → action_sequence(r)(i−1)(n)
  461c    end end
```

5.4.5 Emergency Actions

462. If a unit starts leaking excessive oil
    a. then nearest up-stream valve(s) must be closed,
    b. and any pumps in-between this (these) valves and the leaking unit must be set to not_pumping — following an orderly sequence.
463. If, as a result, for example, of the above remedial actions, any of the desirable properties cease to hold
    a. then — a ha !
    b. Left as an exercise.

5.5 Connectors

The interface , that is, the possible "openings", between adjacent units have not been explored. Likewise the for the possible "openings" of "begin" or "end" units, that is, units not having their input(s), respectively their "output(s)" connected to anything, but left "exposed" to the environment. We now introduce a notion of connectors: abstractly you may think of connectors as concepts, and concretely as "fittings" with bolts and nuts, or "weldings", or "plates" inserted onto "begin" or "end" units.

464. There are connectors and connectors have unique connector identifiers.
465. From a connector one can observe its uniwue connector identifier.
466. From a net one can observe all its connectors
467. and hence one can extract all its connector identifiers.
468. From a connector one can observe a pair of "optional" (distinct) unit identifiers:
    a. An optional unit identifier is
    b. either a unit identifier of some unit of the net

c. or a ‘‘nil’’ “identifier”.

469. In an observed pair of “optional” (distinct) unit identifiers
   - there can not be two ‘‘nil’’ “identifiers”.
   - or the possibly two unit identifiers must be distinct

```
type
    464  K, KI
value
    465  obs_KI: K → KI
    466  obs_Ks: N → K-set
    467  xtr_KIS: N → KI-set
    467  xtr_KIs(n) ≡ {obs_KI(k)|k:K•k ∈ obs_Ks(n)}
type
    468  oUIp′ = (UI|{|nil|})×(UI|{|nil|})
    468  oUIp = {|ouip:oUIp′•wf_oUIp(ouip)|}
value
    468  obs_oUIp: K → oUIp
    469  wf_oUIp: oUIp′ → Bool
    469  wf_oUIp(uon,uon′) ≡
    469    uon=nil⇒uon′≠nil∨uon′=nil⇒uon≠nil∨uon≠uon′
```

470. Under the assumption that a fork unit cannot be adjacent to a join unit
471. we impose the constraint thet no two distinct connectors feature the same pair of actual (distinct) unit identifiers.
472. The first proper unit identifier of a pair of “optional” (distinct) unit identifiers must identify a unit of the net.
473. The second proper unit identifier of a pair of “optional” (distinct) unit identifiers must identify a unit of the net.

```
axiom
   470   ∀ n:N,u,u′:U•{u.u′}⊆obs_Us(n)∧adj(u,u′)⇒ ∼(is_F(u)∧is_J(u′))

   471   ∀ k,k′:K•obs_KI(k)≠obs_KI(k′)⇒
             case (obs_oUIp(k),obs_oUIp(k′)) of
                 ((nil,ui),(nil,ui′)) → ui≠ui′,
                 ((nil,ui),(ui′,nil)) → false,
                 ((ui,nil),(nil,ui′)) → false,
                 ((ui,nil),(ui′,nil)) → ui≠ui′,
                 _ → false
             end


   ∀ n:N,k:K•k ∈ obs_Ks(n) ⇒
       case obs_oUIp(k) of
    472      (ui,nil) → ∃UI(ui)(n)
    473      (nil,ui) → ∃UI(ui)(n)
    472-473  (ui,ui′) → ∃UI(ui)(n)∧∃UI(ui′)(n)
       end
value
   ∃UI: UI → N → Bool
   ∃UI(ui)(n) ≡ ∃ u:U•u ∈ obs_Us(n)∧obs_UI(u)=ui
```

5.6 On Temporal Aspects of Pipelines

The else_qual(u,u′)(n,n′) function definition represents a gross simplification. It ignores the actual flow which changes as a result of setting alternate states, and hence the net state. We now wish to capture the dynamics of flow. We shall do so using the Duration Calculus — a continuous time, integral temporal logic that is semantically and proof system "integrated" with RSL:

Zhou ChaoChen and Michael Reichhardt Hansen
Duration Calculus: A Formal Approach to Real-time Systems
Monographs in Theoretical Computer Science
The EATCS Series
Springer 2004

5.7 A CSP Model of Pipelines

We recapitulate Sect. **5.5** — now adding connectors to our model:

474. From an oil pipeline system one can observe units and connectors.
475. Units are either well, or pipe, or pump, or valve, or join, or fork or sink units.
476. Units and connectors have unique identifiers.
477. From a connector one can observe the ordered pair of the identity of the two from-, respectively to-units that the connector connects.

type
474 OPLS, U, K
476 UI, KI
value
474 obs_Us: OPLS → U-**set**, obs_Ks: OPLS → K-**set**
475 is_WeU, is_PiU, is_PuU, is_VaU,
475 is_JoU, is_FoU, is_SiU: U → **Bool** [mutually exclusive]
476 obs_UI: U → UI, obs_KI: K → KI
477 obs_UIp: K → (UI|{nil}) × (UI|{nil})

Above, we think of the types OPLS, U, K, UI and KI as denoting semantic entities. Below, in the next section, we shall consider exactly the same types as denoting syntactic entities !

478. There is given an oil pipeline system, opls.
479. To every unit we associate a CSP behaviour.
480. Units are indexed by their unique unit identifiers.
481. To every connector we associate a CSP channel.
Channels are indexed by their unique "k"onnector identifiers.
482. Unit behaviours are cyclic and over the state of their (static and dynamic) attributes, represented by u.
483. Channels, in this model, have no state.
484. Unit behaviours communicate with neighbouring units — those with which they are connected.
485. Unit functions, $\mathcal{U}_i$, change the unit state.
486. The pipeline system is now the parallel composition of all the unit behaviours.

Editorial Remark: Our use of the term unit and the RSL literal **Unit** may seem confusing, and we apologise. The former, unit, is the generic name of a well, pipe, or pump, or valve, or join, or fork, or sink. The literal **Unit**, in a function signature, before the → "announces" that the function takes

no argument.[21] The literal **Unit**, in a function signature, after the → "announces", as used here, that the function never terminates.

```
value
478  opls:OPLS
channel
481  {ch[ ki ]|k:KI,k:K•k ∈ obs_Ks(opls)∧ki=obs_KI(k)} M
value
486  pipeline_system: Unit → Unit
486  pipeline_system() ≡
479    || {unit(ui)(u)|u:U•u ∈ obs_Us(opls)∧ui=obs_UI(u)}

480  unit: ui:UI → U →
484       in,out {ch[ ki ]|k:K,ki:KI•k ∈ obs_Ks(opls)∧ki=obs_KI(k)∧
484                  let (ui′,ui′′)=obs_UIp(k) in ui ∈{ui′,ui′′}\{nil} end}  Unit
482  unit(ui)(u) ≡ let u′ = 𝒰ᵢ(ui)(u) in unit(ui)(u′) end

485  𝒰ᵢ: ui:UI → U →
485       in,out {ch[ ki ]|k:K,ki:KI•k ∈ obs_Ks(opls)∧ki=obs_KI(k)∧
485                  let (ui′,ui′′)=obs_UIp(k) in ui ∈{ui′,ui′′}\{nil} end}  U
```

5.8 Conclusion

We have shown draft sketches of aspects of gas/oil pipelines. From a comprehensive such domain description we can systematically "derive" a set of complementary or alternative requirements prescriptions for the monitoring and control of individual pipe units, as well as of consolidated pipelines. Etcetera !

[21] **Unit** is a type name; () is the only value of type **Unit**.

Chapter 6
Simple Credit Card Systems [May 2016]

Contents

6.1 Introduction 115
6.2 Endurants 116
6.2.1 Credit Card Systems 116
6.2.2 Credit Cards 117
6.2.3 Banks 117
6.2.4 Shops 118
6.3 Perdurants 119
6.3.1 Behaviours 119
6.3.2 Channels 119
6.3.3 Behaviour Interactions 120
6.3.4 Credit Card 121
6.3.5 Banks 122
6.3.6 Shops 124

We present an attempt at a model of a simple credit card system of credit card holders, shops and banks.[22]

6.1 Introduction

We present a domain description of an abstracted credit card system. The narrative part of the description is terse, perhaps a bit too terse.

Credit cards are moving from simple plastic cards to smart phones. Uses of credit cards move from their mechanical insertion in credit card terminals to being swiped. Authentication (hence not modelled) moves from keying in security codes to eye iris "prints", and/or finger prints or voice prints or combinations thereof.

This document abstracts from all that in order to understand a bare, minimum essence of credit cards and their uses. Based on a model, such as presented here, the reader should be able to extend/refine the model into any future technology – for requirements purposes.

[22] This model evolved during a PhD course at the University of Uppsala, Sweden.

6.2 Endurants

6.2.1 Credit Card Systems

487. Credit card systems, *ccs*:*CCS*, [23]consists of three kinds of parts:
488. an assembly, *cs*:*CS*, of credit cards[25],
489. an assembly, *bs*:*BS*, of banks, and
490. an assembly, *ss*:*SS*, of shops.

```
type
487  CCS
488  CS
489  BS
490  SS
value
488  obs_CS: CCS → CS
489  obs_BS: CCS → BS
490  obs_SS: CCS → SS
```

491. There are credit cards, *c*:*C*, banks *b*:*B*, and shops *s*:*S*.
492. The credit card part, *cs*:*CS*, abstracts a set, *soc*:*Cs*, of card.
493. The bank part, *bs*:*BS*, abstracts a set, *sob*:*Bs*, of banks.
494. The shop part, *ss*:*SS*, abstracts a set, *sos*:*Ss*, of shops.

```
type
491  C, B, S
492  Cs = C-set
493  Bs = B-set
494  Ss = S-set
value
492  obs_CS: CS → Cs, obs_Cs: CS → Cs
493  obs_BS: BS → Bs, obs_Bs: BS → Bs
494  obs_SS: SS → Ss, obs_Ss: SS → Ss
```

495. Assembliers of credit cards, banks and shops have unique identifiers, *csi*:*CSI*, *bsi*:*BSI*, and *ssi*:*SSI*.
496. Credit cards, banks and shops have unique identifiers, *ci*:*CI*, *bi*:*BI*, and *si*:*SI*.
497. One can define functions which extract all the
498. unique credit card,
499. bank and
500. shop identifiers from a credit card system.

```
495  CSI, BSI, SSI
496  CI, BI, SI
value
495  uid_CS: CS→CSI, uid_BS: BS→BSI, uid_SS: SS→SSI,
```

[23] The composite part *CS* can be thought of as a credit card company, say VISA[24]. The composite part *BS* can be thought of as a bank society, say BBA: British Banking Association. The composite part *SS* can be thought of as the association of retailers, say bira: British Independent Retailers Association. The model does not prevent "shops" from being airlines, or car rental agencies, or dentists, or consultancy firms. In this case *SS* would be some appropriate association.

[25] We "equate" credit cards with their holders.

```
496  uid_C: C→CI, uid_B: B→BI, uid_S: S→SI,
498  xtr_CIs: CCS → CI-set
498  xtr_CIs(ccs) ≡ {uid_C(c)|c:C•c ∈ obs_Cs(obs_CS(ccs))}
499  xtr_BIs: CCS → BI-set
499  xtr_BIs(ccs) ≡ {uid_B(s)|b:B•b ∈ obs_Bs(obs_BS(ccs))}
500  xtr_SIs: CCS → SI-set
500  xtr_SIs(ccs) ≡ {uid_S(s)|s:S•s ∈ obs_Ss(obs_SS(ccs))}
```

501. For all credit card systems it is the case that
502. all credit card identifiers are distinct from bank identifiers,
503. all credit card identifiers are distinct from shop identifiers,
504. all shop identifiers are distinct from bank identifiers,

```
axiom
501  ∀ ccs:CCS •
501    let cis=xtr_CIs(ccs), bis=xtr_BIs(ccs), sis = xtr_SIs(ccs) in
502    cis ∩ bis = {}
503  ∧ cis ∩ sis = {}
504  ∧ sis ∩ bis = {} end
```

6.2.2 Credit Cards

505. A credit card has a mereology which “connects” it to any of the shops of the system and to exactly one bank of the system,
506. and some attributes — which we shall presently disregard.
507. The wellformedness of a credit card system includes the wellformedness of credit card mereologies with respect to the system of banks and shops:
508. The unique shop identifiers of a credit card mereology must be those of the shops of the credit card system; and
509. the unique bank identifier of a credit card mereology must be of one of the banks of the credit card system.

```
type
505.   CM = SI-set × BI
value
505.   obs_mereo_CM: C → CM
507    wf_CM_of_C: CCS → Bool
507    wf_CM_of_C(ccs) ≡
505      let bis=xtr_BIs(ccs), sis=xtr_SIs(ccs) in
505       ∀ c:C•c ∈ obs_Cs(obs_CS(ccs)) ⇒
505         let (ccsis,bi)=obs_mereo_CM(c) in
508         ccsis ⊆ sis
509        ∧ bi ∈ bis
505      end end
```

6.2.3 Banks

Our model of banks is (also) very limited.

510. A bank has a mereology which "connects" it to a subset of all credit cards and a subset of all shops,
511. and, as attributes:
512. a cash register, and
513. a ledger.
514. The ledger records for every card, by unique credit card identifier,
515. the current balance: how much money, credit or debit, i.e., plus or minus, that customer is owed, respectively has borrowed from the bank,
516. the dates-of-issue and -expiry of the credit card, and
517. the name, address, and other information about the credit card holder.
518. The wellformedness of the credit card system includes the wellformedness of the banks with respect to the credit cards and shops:
519. the bank mereology's
520. must list a subset of the credit card identifiers and a subset of the shop identifiers.

```
type
510   BM = CI-set × SI-set
512   CR = Bal
513   LG = CI ₘ→ (Bal×DoI×DoE×...)
515   Bal = Int
value
510   obs_mereo_B: B → BM
512   attr_CR: B → CR
513   attr_LG: B → LG
518      wf_BM_B: CCS → Bool
518      wf_BM_B(ccs) ≡
518         let allcis = xtr_CIs(ccs), allsis = xtr_SIs(ccs) in
518         ∀ b:B • b ∈ obs_Bs(obs_BS(ccs)) in
519            let (cis,sis) = obs_mereo_B(b) in
520            cis ⊆ ∀ cis ∧ sis ⊆ allsis end end
```

6.2.4 Shops

521. The mereology of a shop is a pair: a unique bank identifiers, and a set of unique credit card identifiers.
522. The mereology of a shop
523. must list a bank of the credit card system,
524. band a subset (or all) of the unique credit identifiers.

We omit treatment of shop attributes.

```
type
521   SM = CI-set × BI
value
521   obs_mereo_S: S → SM
522   wf_SM_S: CCS → Bool
522   wf_SM_S(ccs) ≡
522      let allcis = xtr_CIs(ccs), allbis = xtr_BIs(ccs) in
522      ∀ s:S • s ∈ obs_Ss(obs_SS(ccs)) ⇒
522         let (cis,bi) obs_mereo_S(s) in
```

```
523         bi ∈ allbis
524       ∧ cis ⊆ allcis
522     end end
```

6.3 Perdurants

6.3.1 Behaviours

525. We ignore the behaviours related to the *CCS*, *CS*, *BS* and *SS* parts.
526. We therefore only consider the behaviours related to the *Cs*, *Bs* and *Ss* parts.
527. And we therefore compile the credit card system into the parallel composition of the parallel compositions of all the credit card, *crd*, all the bank, *bnk*, and all the shop, *shp*, behaviours.

```
value
525  ccs:CCS
525  cs:CS = obs_CS(ccs),
525  uics:CSI =uid_CS(cs),
525  bs:BS = obs_BS(ccs),
525  uibs:BSI =uid_BS(bs),
525  ss:SS = obs_SS(ccs),
525  uiss:SSI =uid_SS(ss),
526  socs:Cs = obs_Cs(cs),
526  sobs:Bs = obs_Bs(bs),
526  soss:Ss = obs_Ss(ss),
```

```
value
527  sys: Unit → Unit,
525  sys() ≡
527        cards_uics(obs_mereo_CS(cs),...)
527     || || {crd_uid_C(c)(obs_mereo_C(c))|c:C•c ∈ socs}
527     || banks_uibs(obs_mereo_BS(bs),...)
527     || || {bnk_uid_B(b)(obs_mereo_B(b))|b:B•b ∈ sobs}
527     || shops_uiss(obs_mereo_SS(ss),...)
527     || || {shp_uid_S(s)(obs_mereo_S(s))|s:S•s ∈ soss},
525  cards_uics(...) ≡ skip,
525  banks_uibs(...) ≡ skip,
525  shops_uiss(...) ≡ skip

axiom   skip || behaviour(...) ≡ behaviour(...)
```

6.3.2 Channels

528. Credit card behaviours interact with bank (each with one) and many shop behaviours.
529. Shop behaviours interact with bank (each with one) and many credit card behaviours.
530. Bank behaviours interact with many credit card and many shop behaviours.
The inter-behaviour interactions concern:

531. between credit cards and banks: withdrawal requests as to a sufficient, mk_Wdr(am), balance on the credit card account for buying am:AM amounts of goods or services, with the bank response of either is_OK() or is_NOK(), or the revoke of a card;
532. between credit cards and shops: the buying, for an amount, am:AM, of goods or services: mk_Buy(am), or the refund of an amount;
533. between shops and banks: the deposit of an amount, am:AM, in the shops' bank account: mk_Depost(ui,am) or the removal of an amount, am:AM, from the shops' bank account: mk_Removl(bi,si,am)

```
channel
528 {ch_cb[ ci,bi ]|ci:CI,bi:BI•ci ∈ cis ∧ bi ∈ bis}:CB_Msg
529 {ch_cs[ ci,si ]|ci:CI,si:SI•ci ∈ cis ∧ si ∈ sis}:CS_Msg
530 {ch_sb[ si,bi ]|si:SI,bi:BI•si ∈ sis ∧ bi ∈ bis}:SB_Msg
531 CB_Msg == mk_Wdrw(am:aM) | is_OK() | is_NOK() | ...
532 CS_Msg == mk_Buy(am:aM) | mk_Ref(am:aM) | ...
533 SB_Msg == Depost | Removl | ...
533 Depost == mk_Dep((ci:CI|si:SI),am:aM) |
533 Removl == mk_Rem(bi:BI,si:SI,am:aM)
```

6.3.3 Behaviour Interactions

534. The credit card initiates
   a. buy transactions
      i. [1.Buy] by enquiring with its bank as to sufficient purchase funds (am:aM);
      ii. [2.Buy] if NOK then there are presently no further actions; if OK
      iii. [3.Buy] the credit card requests the purchase from the shop – handing it an appropriate amount;
      iv. [4.Buy] finally the shop requests its bank to deposit the purchase amount into its bank account.
   b. refund transactions
      i. [1.Refund] by requesting such refunds, in the amount of am:aM, from a[ny] shop; whereupon
      ii. [2.Refund] the shop requests its bank to move the amount am:aM from the shop's bank account
      iii. [3.Refund] to the credit card's account.

Thus the three sets of behaviours, crd, bnk and shp interact as sketched in Fig. 6.1 on the facing page.

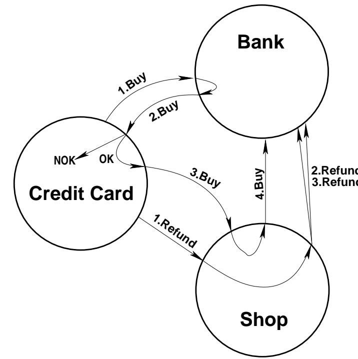


Fig. 6.1 Credit Card, Bank and Shop Behaviours

[1.Buy]	Item 540, Pg.121	card ch_cb[ci,bi]!mk_Wdrw(am) (shown as ... three lines down) and
	Item 549, Pg.123	bank mk_Wdrw(ci,am)=⌈⌉{ch_cb[bi,bi]?|ci:CI•ci ∈ cis}.
[2.Buy]	Items 542-543, Pg.122	bank ch_cb[ci,bi]!is_[N]OK() and
	Item 540, Pg.121	shop (...;ch_cb[ci,bi]?).
[3.Buy]	Item 542, Pg.122	card ch_cs[ci,si]!mk_Buy(am) and
	Item 564, Pg.124	shop mk_Buy(am)=⌈⌉{ch_cs[ci,si]?|ci:CI•ci∈ cis}.
[4.Buy]	Item 565, Pg.124	shop ch_sb[si,bi]!mk_Dep(si,am) and
	Item 554, Pg.123	bank mk_Dep(si,am)=⌈⌉{ch_cs[ci,si]?|si:SI•si∈sis}.
[1.Refund]	Item 546, Pg.122	card ch_cs[ci,si]!mk_Ref((ci,si),am) and
	Item 565, Pg.124	shop (si,mk_Ref(ci,am))=⌈⌉{si′,ch_sb[si,bi]?|si,si′:SI•{si,si′}⊆sis∧si=si′}.
[2.Refund]	Item 569, Pg.124	shop ch_sb[si,cbi]!mk_Ref(cbi,(ci,si),am and
	Item 558, Pg.123	bank (si,mk_Ref(cbi,(ci,am)))=⌈⌉{(si′,ch_sb[si,bi]?)|si,si′:SI•{si,si′}⊆sis∧si=si′}.
[3.Refund]	Item 570, Pg.124	shop ch_sb[si,sbi]!mk_Wdr(si,am)) **end** and
	Item 559, Pg.123	bank (si,mk_Wdr(ci,am))=⌈⌉{(si′,ch_sb[si,bi]?)|si,si′:SI•{si,si′}⊆sis∧si=si′}

6.3.4 Credit Card

535. The credit card behaviour, crd, takes the credit card unique identifier, the credit card mereology, and attribute arguments (omitted). The credit card behaviour, crd, accepts inputs from and offers outputs to the bank, bi, and any of the shops, si∈sis.
536. The credit card behaviour, crd, non-deterministically, internally “cycles” between buying and getting refunds.

value
535 $\mathsf{crd}_{ci:CI}$: (bi,sis):CM → **in**,**out** ch_cb[ci,bi],{ch_cs[ci,si]|si:SI•si ∈ sis} **Unit**
535 crd_{ci}(bi,sis) ≡ (buy(ci,(bi,sis)) ⌈⌉ ref(ci,(bi,sis))) ; crd_{ci}(ci,(bi,sis))

537. By am:AM we mean an amount of money, and by si:SI we refer to a shop in which we have selected a number or goods or services (not detailed) costing am:AM.
538. The buyer action is simple.
539. The amount for which to buy and the shop from which to buy are selected (arbitrarily).
540. The credit card (holder) withdraws am:AM from the bank, if sufficient funds are available[26].

[26] First the credit card [holder] requests a withdrawal. If sufficient funds are available, then the withdrawal takes place, otherwise not – and the credit card holder is informed accordingly.

541. The response from the bank
542. is either OK and the credit card [holder] completes the purchase by buying the goods or services offered by the selected shop,
543. or the response is "not OK", and the transaction is skipped.

```
type
537 AM = Int
value
538 buy: ci:CI × (bi,sis):CM →
538    in,out ch_cb[ci,bi] out {ch_cs[ci,si]|si:SI•si ∈ sis} Unit
538 buy(ci,(bi,sis)) ≡
539    let am:aM • am>0, si:SI • si ∈ sis in
540    let msg = (ch_cb[ci,bi]!mk_Wdrw(am);ch_cb[ci,bi]?) in
541    case msg of
542       is_OK() → ch_cs[ci,si]!mk_Buy(am),
543       is_NOK() → skip
538    end end end
```

544. The refund action is simple.
545. The credit card [handler] requests a refund am:AM
546. from shop si:SI.
This request is handled by the shop behaviour's sub-action *ref*, see lines 562.–571. page 124.

```
value
544 rfu: ci:CI × (bi,sis):CM → out {ch_cs[ci,si]|si:SI•si ∈ sis} Unit
544 rfu(ci,(bi,sis)) ≡
545       let am:AM • am>0, si:SI • si ∈ sis in
546       ch_cs[ci,si]!mk_Ref(bi,(ci,si),am)
544       end
```

6.3.5 Banks

547. The bank behaviour, bnk, takes the bank's unique identifier, the bank mereology, and the programmable attribute arguments: the ledger and the cash register. The bank behaviour, bnk, accepts inputs from and offers outputs to the any of the credit cards, *ci*∈*cis*, and any of the shops, *si*∈*sis*.
548. The bank behaviour non-deterministically externally chooses to accept either 'withdraw'al requests from credit cards or 'deposit' requests from shops or 'refund' requests from credit cards.

```
value
547 bnk_{bi:BI}: (cis,sis):BM → (LG×CR) →
547       in,out {ch_cb[ci,bi]|ci:CI•ci ∈ cis} {ch_sb[si,bi]|si:SI•si ∈ sis} Unit
547 bnk_{bi}((cis,sis))(lg:(bal,doi,doe,...),cr) ≡
548       wdrw(bi,(cis,sis))(lg,cr)
548       [] depo(bi,(cis,sis))(lg,cr)
548       [] refu(bi,(cis,sis))(lg,cr)
```

549. The 'withdraw' request, wdrw, (an action) non-deterministically, externally offers to accept input from a credit card behaviour and marks the only possible form of input from credit cards, mk_Wdrw(ci,am), with the identity of the credit card.
550. If the requested amount (to be withdrawn) is not within balance on the account
551. then we, at present, refrain from defining an outcome (**chaos**), whereupon the bank behaviour is resumed with no changes to the ledger and cash register;
552. otherwise the bank behaviour informs the credit card behaviour that the amount can be withdrawn; whereupon the bank behaviour is resumed notifying a lower balance and 'withdraws' the monies from the cash register.

```
value
548 wdrw: bi:BI × (cis,sis):BM → (LG×CR) → in,out {ch_cb[bi,ci]|ci:CI•ci ∈ cis} Unit
548 wdrw(bi,(cis,sis))(lg,cr) ≡
549     let mk_Wdrw(ci,am) = ⌈⌉ {ch_cb[ci,bi]?|ci:CI•ci ∈ cis} in
548     let (bal,doi,doe) = lg(ci) in
550     if am>bal
551        then (ch_cb[ci,bi]!is_NOK(); bnk_bi(cis,sis)(lg,cr))
552        else (ch_cb[ci,bi]!is_OK(); bnk_bi(cis,sis)(lg†[ci↦(bal−am,doi,doe)],cr−am)) end
547     end end
```

The ledger and cash register attributes, lg,cr, are programmable attributes. Hence they are modeled as separate function arguments.

553. The deposit action is invoked, either by a shop behaviour, when a credit card [holder] buy's for a certain amount, am:AM, or requests a refund of that amount. The deposit is made by shop behaviours, either on behalf of themselves, hence am:AM, is to be inserted into the shops' bank account, si:SI, or on behalf of a credit card [i.e., a customer], hence am:AM, is to be inserted into the credit card holder's bank account, si:SI.
554. The message, ch_cs[ci,si]?, received from a credit card behaviour is either concerning a buy [in which case *i* is a ci:CI, hence sale, or a refund order [in which case *i* is a si:SI].
555. In either case, the respective bank account is "upped" by am:AM – and the bank behaviour is resumed.

```
value
553 deposit: bi:BI × (cis,sis):BM → (LG×CR) →
554     in,out {ch_sb[bi,si]|si:SI•si ∈ sis} Unit
553 deposit(bi,(cis,sis))(lg,cr) ≡
554     let mk_Dep(si,am) = ⌈⌉ {ch_cs[ci,si]?|si:SI•si ∈ sis} in
553     let (bal,doi,doe) = lg(si) in
555     bnk_bi(cis,sis)(lg†[si↦(bal+am,doi,doe)],cr+am)
553     end end
```

556. The refund action
557. non-deterministically externally offers to either
558. non-deterministically externally accept a mk_Ref(ci,am) request from a shop behaviour, si, or
559. non-deterministically externally accept a mk_Wdr(ci,am) request from a shop behaviour, si. The bank behaviour is then resumed with the
560. credit card's bank balance and cash register incremented by am and the
561. shop' bank balance and cash register decremented by that same amount.

```
value
556 rfu: bi:BI × (cis,sis):BM → (LG×CR) → in,out {ch_sb[bi,si]|si:SI•si ∈ sis} Unit
556 rfu(bi,(cis,sis))(lg,cr) ≡
```

```
558     (let (si,mk_Ref(cbi,(ci,am))) = ⌈⌉ {(si′,ch_sb[si,bi]?)|si,si′:SI•{si,si′}⊆sis∧si=si′} in
556      let (balc,doic,doec) = lg(ci) in
560      bnk_bi(cis,sis)(lg†[ci↦(balc+am,doic,doec)],cr+am)
556      end end)
557   ⌈⌉
559     (let (si,mk_Wdr(ci,am)) = ⌈⌉ {(si′,ch_sb[si,bi]?)|si,si′:SI•{si,si′}⊆sis∧si=si′} in
556      let (bals,dois,does) = lg(si) in
561      bnk_bi(cis,sis)(lg†[si↦(bals−am,dois,does)],cr−am)
556      end end)
```

6.3.6 Shops

562. The shop behaviour, shp, takes the shop's unique identifier, the shop mereology, etcetera.
563. The shop behaviour non-deterministically, externally
either
564. offers to accept a Buy request from a credit card behaviour,
565. and instructs the shop's bank to deposit the purchase amount.
566. whereupon the shop behaviour resumes being a shop behaviour;
567. or
568. offers to accept a refund request in this amount, am, from a credit card [holder].
569. It then proceeds to inform the shop's bank to withdraw the refund from its ledger and cash register,
570. and the credit card's bank to deposit the refund into its ledger and cash register.
571. Whereupon the shop behaviour resumes being a shop behaviour.

```
value
562 shp_si:SI: (CI-set×BI)×...→in,out: {ch_cs[ci,si]|ci:CI•ci ∈ cis},{ch_sb[si,bi′]|bi′:BI•bi′isin bis} Unit
562 shp_si((cis,bi),...) ≡
564      (sal(si,(bi,cis),...)
567      ⌈⌉
568      ref(si,(cis,bi),...)):

562 sal: SI×(CI-set×BI)×...→in,out: {cs[ci,si]|ci:CI•ci ∈ cis},sb[si,bi] Unit
562 sal(si,(cis,bi),...) ≡
564     let mk_Buy(am) = ⌈⌉{ch_cs[ci,si]?|ci:CI•ci ∈ cis} in
565     ch_sb[si,bi]!mk_Dep(si,am) end ;
566     shp_si((cis,bi),...)

562 ref: SI×(CI-set×BI)×...→in,out: {ch_cs[ci,si]|ci:CI•ci ∈ cis},{ch_sb[si,bi′]|bi′:BI•bi′isin bis} Unit
568 ref(si,(cis,sbi),...) ≡
568     let mk_Ref((ci,cbi,si),am) = ⌈⌉{ch_cs[ci,si]?|ci:CI•ci ∈ cis} in
569     (ch_sb[si,cbi]!mk_Ref(cbi,(ci,si),am)
570   ∥ ch_sb[si,sbi]!mk_Wdr(si,am)) end ;
571     shp_si((cis,sbi),...)
```

Chapter 7
Weather Systems [November 2016]

Contents

7.1 On Weather Information Systems 126
7.1.1 On a Base Terminology 126
7.1.2 Some Illustrations 127
7.1.2.1 Weather Stations 127
7.1.2.2 Weather Forecasts 127
7.1.2.3 Forecast Consumers 127
7.2 Major Parts of a Weather Information System 127
7.3 Endurants 128
7.3.1 Parts and Materials 128
7.3.2 Unique Identifiers 129
7.3.3 Mereologies 130
7.3.4 Attributes 130
7.3.4.1 Clock, Time and Time-intervals 130
7.3.4.2 Locations 131
7.3.4.3 Weather 131
7.3.4.4 Weather Stations 132
7.3.4.5 Weather Data Interpreter 132
7.3.4.6 Weather Forecasts 133
7.3.4.7 Weather Forecast Consumer 133
7.4 Perdurants 133
7.4.1 A WIS Context 133
7.4.2 Channels 134
7.4.3 WIS Behaviours 134
7.4.4 Clock 135
7.4.5 Weather Station 135
7.4.6 Weather Data Interpreter 136
7.4.6.1 collect_wd 136
7.4.6.2 calculate_wf 137
7.4.6.3 disseminate_wf 137
7.4.7 Weather Forecast Consumer 138
7.5 Conclusion 139
7.5.1 Reference to Similar Work 139
7.5.2 What Have We Achieved ? 139
7.5.3 What Needs to be Done Next ? 139
7.5.4 Acknowledgements 139

This document reports a class exercise from a PhD course at the University of Bergen, Norway, November 2016.[27] We show an example domain description. It is developed and presented

[27] I thank my host, Prof. Magne Haveraaen for the invitation. The occasion was that of a visit by Mme. Dooren Tuheirwe from Makerere University, Uganda, and her work with the universty and the Norwegian Meteorological Institute rm on a joint project on a Weather Information System for Uganda.

as outlined in [48]. The domain being described is that of a generic weather information system. Four main endurants (i.e., aspects) of a generic weather information system are those of the weather, weather stations (collecting weather data), weather data interpretation (i.e., metereological institute[s]), and weather forecast consumers. There are, correspondingly, two kinds of weather information: the weather data, and the weather forecasts. These forms of weather information are acted upon: the weather data interpreter (i.e., a metereological institute) is gathering weather data; based on such interpretations the metereological institute is "calculating" weather forecasts; and and weather forecast consumers are requesting and further "interpreting" (i.e., rendering) such forecasts. Thus weather data is communicated from weather stations to the weather data interpreter; and weather forecasts are communicated from the weather data interpreter to the weather forecast consumers. It is the dual purpose of this technical report to present a domain description of the essence of generic weather information systems, and to add to the "pile" [38, 37, 42, 41, 43, 46, 45, 47] of technical reports that illustrate the use[fulness] of the principles, techniques and tools of [48].

7.1 On Weather Information Systems

7.1.1 On a Base Terminology

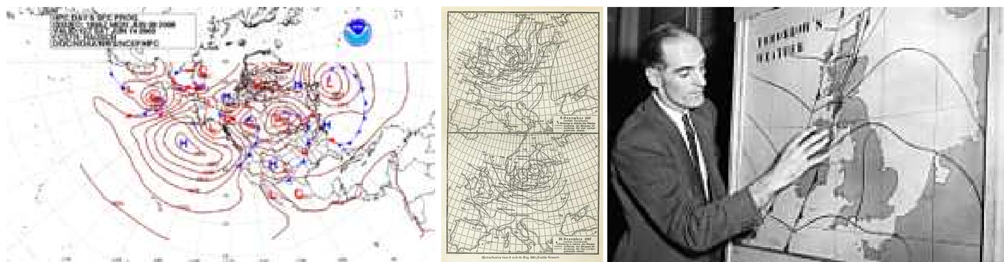
From Wikipedia:

572. **Weather** is the state of the atmosphere, to the degree that it is hot or cold, wet or dry, calm or stormy, clear or cloudy, atmospheric (barometric) pressure: high or low.
573. So weather is characterized by **temperature, humidity** (incl. **rain**, **wind** (direction, velocity, center, incl. its possible mobility), **atmospheric pressure**, etcetera.
574. By **weather information** we mean
    - either weather data that characterizes the weather as defined above (Item 572),
    - or weather forecast, i.e., a prediction of the state of the atmosphere for a given location and time or time interval.
575. Weather data are collected by **weather stations**. We shall here not be concerned with technical means of weather data collection.
576. **Weather forecasts** are used by forecast consumers, anyone: you and me.
577. Weather data interpretation (i.e., **forecasting**) is the science and technology of creating weather forecasts based on **time-** or **time interval-stamped weather data** and **locations**. Weather data interpretation is amongst the charges of meteorological institutes.
578. **Meteorology** is the interdisciplinary scientific study of the atmosphere.
579. An **atmosphere** (from Greek $\alpha\tau\mu o\zeta$ (atmos), meaning "vapour", and $\sigma\phi\alpha\iota\rho\alpha$ (sphaira), meaning "sphere") is a layer of gases surrounding a planet or other material body, that is held in place by the gravity of that body.
580. Meteorological institutes work together with the World Meteorological Organization (WMO). Besides weather forecasting, meteorological institutes (and hence WMO) are concerned also with aviation, agricultural, nuclear, maritime, military and environmental meteorology, hydrometeorology and renewable energy.
581. Agricultural meteorologists, soil scientists, agricultural hydrologists, and agronomists are persons concerned with studying the effects of weather and climate on plant distribution, crop yield, water-use efficiency, phenology of plant and animal development, and the energy balance of managed and natural ecosystems. Conversely, they are interested in the rôle of vegetation on climate and weather.

7.1.2 Some Illustrations

7.1.2.1 Weather Stations

7.1.2.2 Weather Forecasts



7.1.2.3 Forecast Consumers

7.2 Major Parts of a Weather Information System

We think of the following parts as being of concern in the kind of weather information systems that we shall analyse and describe: Figure 7.1 on the next page shows one **weather** (dashed rounded corner all embracing rectangle), one central **weather data interpreter** (cum meteorological institute) seven **weather stations** (rounded corner squares), nineteen **weather forecast consumers**, and one global **clock**. All are distributed, as hinted at, in some geographical space. Figure 7.2 shows "an orderly diagram" of "the same" weather information system as Figure 7.1. The lines between pairs of the various parts shall indicate means communication between the pairs of (thus) connected parts. Dashed lines "crossing" bundles of these communication lines are labeled ch_xy. These

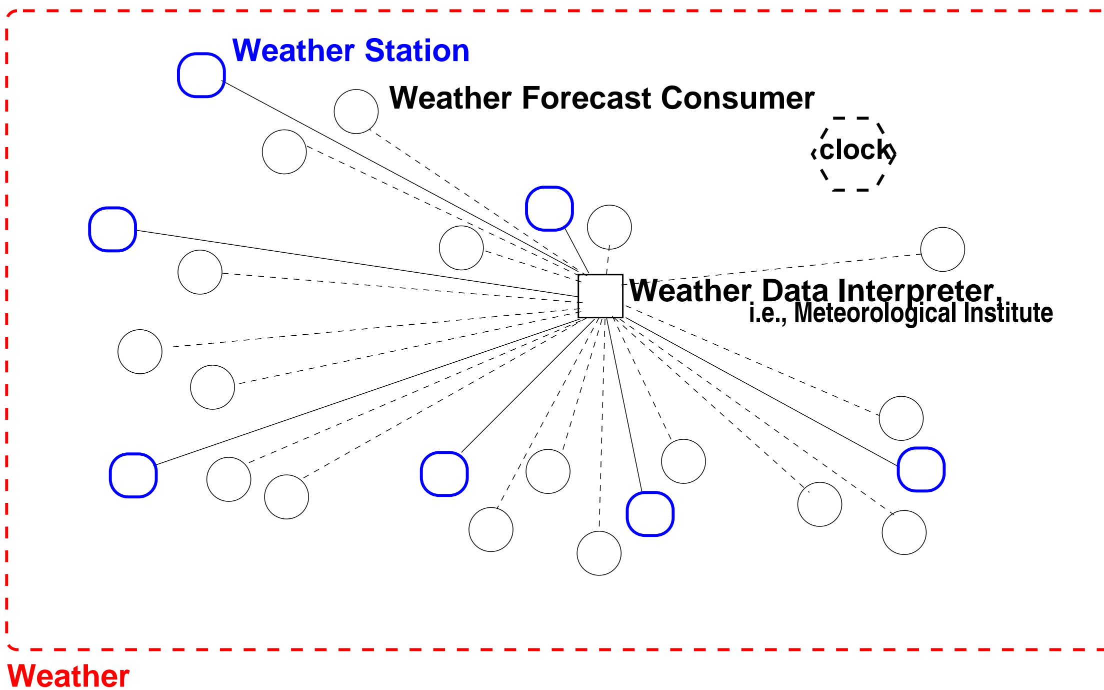


Fig. 7.1 A Weather Information System

labels, ch_xy, designated CSP-like channels. An input, by a weather station (wsi), of weather data from the weather (wi), is designated by the CSP expression ch_ws[wi,wsi] ?. An output, say from the weather data interpreter (wdi) to a weather forecast consumer (fci), of a forecast f, is designated by ch_ic[wdii,fci] ! f

7.3 Endurants

7.3.1 Parts and Materials

582. The WIS domain contains a number of sub-domains:
    a. the weather, W, which we consider a material,
    b. the weather stations sub-domain, WSS (a composite part),
    c. the weather data interpretation sub-domain, WDIS (an atomic part),
    d. the weather forecast consumers sub-domain, WFCS (a composite part), and
    e. the ("global") clock (an atomic part).

type
582 WIS
582a W
582b WSS
582c WDIS
582d WFCS
582e CLK
value

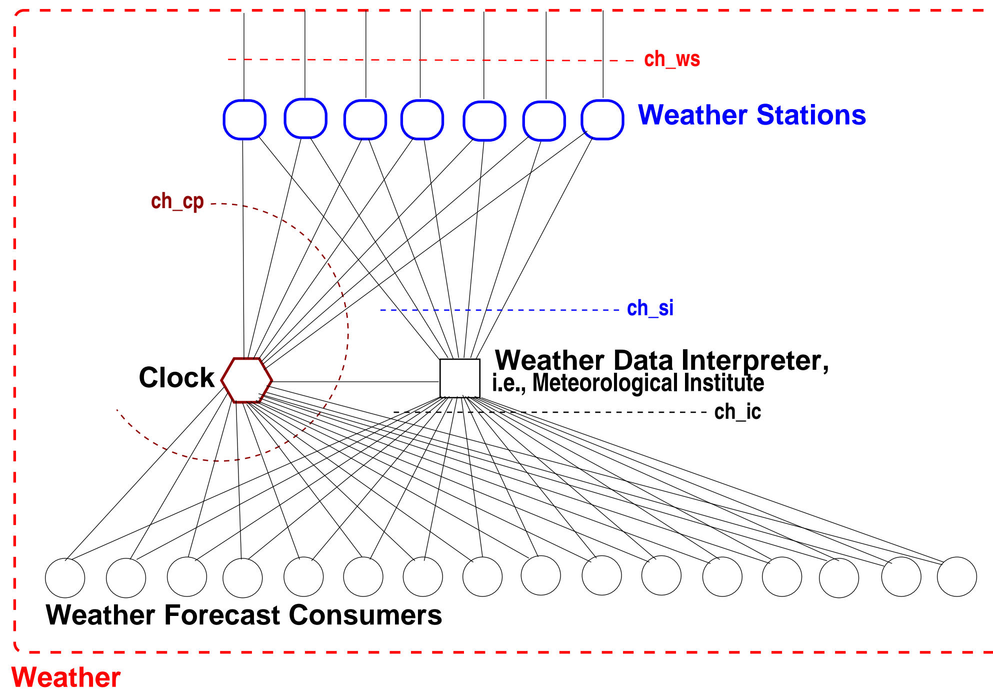


Fig. 7.2 A Weather Information System Diagram

582a obs_material_W: WIS → W
582b obs_part_WSS: WIS → WSS
582c obs_part_WDIS: WIS → WDIS
582d obs_part_WFCS: WIS → WFCS
582e obs_part_CLK: WIS → CLK

583. The weather station sub-domain, WSS, consists of a set, WSs,
584. of atomic weather stations, WS.
585. The weather forecast consumers sub-domain, WFCS, consists of a set, WFCs,
586. of atomic weather forecast consumers, WFC.

type
583 WSs = WS-**set**
584 WS
585 WFCs = WFC-**set**
586 WFC
value
583 obs_part_WSs: WSS → WSs
585 obs_part_WFCs: WFCS → WFCs

7.3.2 Unique Identifiers

We shall consider only atomic parts.

587. Every single weather station has a unique identifier.
588. The weather data interpretation (i.e., the weather forecast "creator") has a unique identifier.
589. Every single weather forecast consumer has a unique identifier.
590. The global clock has a unique identifier.

type
587 WSI
588 WDII
589 WFCI
590 CLKI
value
587 uid_WSI: WS → WSI
588 uid_WDII: WDIS → WDII
589 uid_WFCI: WFC → WFCI
589 uid_CLKI: CLK → CLKI

7.3.3 Mereologies

We shall restrict ourselves to consider the mereologies only of the atomic parts.

591. The mereology of weather stations is the pair of the unique clock identifier and the unique identifier of the weather data interpreter.
592. The mereology of weather data interpreter is the triple of the unique clock identifier, set of unique identifiers of all the weather stations and the set of unique identifiers of all the weather forecast consumers.
593. The mereology of weather forecast consumer is the the pair of the unique clock identifier and the unique identifier of the weather data interpreter.
594. The mereology of the global clock is the triple of the set of all the unique identifiers of weather stations, the unique identifier of the weather data interpreter, and the set of all the unique identifiers of weather forecast consumers.

type
591 WSM = CLKI × WDII
592 WDIM = CLKI × WSI-**set** × WFCI-**set**
593 WFCM = CLKI × WDII
594 CLKM = CLKI × WDGI-**set** × WDII × WFCI-**set**
value
591 mereo_WSM: WS → WSM
592 mereo_WDI: WDI → WDIM
593 mereo_WFC: WFC → WFCM
594 mereo_CLK: CLK → CLKM

7.3.4 Attributes

7.3.4.1 Clock, Time and Time-intervals

595. The global clock has an autonomous time attribute.

596. Time values are further undefined, but times are considered absolute in the sense as representing some intervals since "the birth of time", an example, concrete time could be NOVEMBER 15, 2021: 16:12 .
597. Time intervals are further undefined, but time intervals can be considered relative in the sense of representing a quantity elapsed between two times, examples are: 1 day 2 hours and 3 minutes, etc. When a time interval, *ti*, is specified it is always to be understood to designate the times from now, or from a specified time, *t*, until the time $t + ti$.
598. We postulate ⊕, ⊖, and can postulate further "arithmetic" operators, and
599. we can postulate relational operators.

```
type
595  TIME
596  TI
value
595  attr_TIME: CLK → TIME
598  ⊕: TIME×TI→TIME, TI×TI→TI
598  ⊖: TIME×TI→TIME, TIME×TIME→TI
599  =, ≠, <, ≤, ≥, >: TIME×TIME→Bool, TI×TI→Bool, ...
```

We do not here define these operations and relations.

7.3.4.2 Locations

600. Locations are metric, topological spaces and can thus be considered dense spaces of three dimensional points.
601. We can speak of one location properly contained (⊂) within, or contained or equal (⊆), or equal (=), or not equal (≠) to another location.

```
type
600.  LOC
value
601.  ⊂, ⊆, =, ≠: LOC × LOC → Bool
```

7.3.4.3 Weather

602. The weather material is considered a dense, infinite set of weather point volumes WP. Some dense, infinite subsets (still proper volumes) of such points may be liquid, i.e., rain, water in rivers, lakes and oceans. Other dense, infinite subsets (still proper volumes) of such points may be gaseous, i.e., the air, or atmosphere. These two forms of proper volumes "border" along infinite subsets (curved planes, surfaces) of weather points.
603. From the material weather one can observe its location.

```
type
602  W = WP-infset
602  WP
value
603  attr_LOC: W → LOC
```

604. Some meteorological quantities are:

a. *Hu*midity,
b. *Te*mperature,
c. *Wi*nd and
d. *Ba*rometric pressure.

605. The weather has an indefinite number of attributes at any one time.
    a. Humidity distribution, at level (above sea) and by location,
    b. Temperature distribution, at level (above sea) and by location,
    c. Wind direction, velocity and mobility of wind center, and by location,
    d. Barometric pressure, and by location,
    e. etc., etc.

```
type
604a  Hu
604b  Te
604c  Wi
604d  Ba
605a  HDL = LOC →m Hu
605b  TDL = LOC →m Te
605c  WDL = LOC →m Wi
605d  BPL = LOC →m Ba
605e  ...
value
605a  attr_HDL: W → HDL
605b  attr_TDL: W → TDL
605c  attr_WDL: W → WDL
605d  attr_APL: W → BPL
605e  ...
```

7.3.4.4 Weather Stations

606. Weather stations have static location attributes.
607. Weather stations sample the weather gathering humidity, temperature, wind, barometric pressure, and possibly other data, into time and location stamped weather data.

```
value
606  attr_LOC: WS → LOC
type
607  WD :: mkWD((TIME×LOC)×(TDL×HDL×WDL×BPL×...))
```

7.3.4.5 Weather Data Interpreter

608. There is a programmable attribute: weather data repository, wdr:WDR, of weather data, wd:WD, collected from weather stations.
609. And there is programmable attribute: weather forecast repository, wfr:WFR, of forecasts, wf:WF, disseminate-able to weather forecast consumers.
    These repositories are updated when
610. received from the weather stations, respectively when
611. calculated by the weather data interpreter.

```
type
608  WDR
609  WFR
value
610  update_wdr: TIME × WD → WDR → WDR
611  update_wfr: TIME × WF → WFR → WFR
```

It is a standard exercise to define these two functions (say algebraically).

7.3.4.6 Weather Forecasts

612. Weather forecasts are weather forecast format-, time- and location-stamped quantities, the latter referred to as wefo:WeFo.
613. There are a definite number ($n \geq 1$) of weather forecast formats.
614. We do not presently define these various weather forecast formats.
615. They are here thought of as being requested, mkWFReq, by weather forecast consumers.

```
type
612  WF = WFF × (TIME×TI) × LOC × WeFo
613  WFF = WFF1 | WFF2 | ... | WFFn
614  WFF1, WFF2, ..., WFFn
615  WFReq :: mkWFReq(s_wff:WFF,s_ti:(TIME×TI),s_loc:LOC)
```

7.3.4.7 Weather Forecast Consumer

616. There is a programmable attribute, d:D, D for display (!).
617. Displays can be rendered (RND): visualized, tabularised, made audible, translated (between languages and language dialects, ...), etc.
618. A rendered display can be "abstracted back" into its basic form.
619. Any abstracted rendered display is identical to its abstracted form.

```
type
616  D
617  RND
value
616  attr_D: WFC → D
617  rndr_D: RND × D → D
618  abs_D: D → D
axiom
619  ∀ d:D, r:RND • abs_D(rndr(r,d)) = d
```

7.4 Perdurants

7.4.1 A WIS Context

620. We postulate a given system, wis:WIS. That system is characterized by
621. a dynamic weather
622. and its unique identifier,
623. a set of weather stations
624. and their unique identifiers,
625. a single weather data interpreter
626. and its unique identifier,
627. a set of weather forecast consumers
628. and their unique identifiers, and
629. a single clock
630. and its unique identifier.

631. Given any specific wis:WIS there is [therefore] a full set of part identifiers, is, of weather, clock, all weather stations, the weather data interpreter and all weather forecast consumers.

We list the above-mentioned values. They will be referenced by the channel declarations and the behaviour definitions of this section.

```
value
620  wis:WIS
621  w:W = obs_material_W(wis)
622  wi:WI = uid_WI(w)
623  wss:WSs = obs_part_WSs(obs_part_WSS(wis))
624  wsis:WDGI-set = {uid_WSI(ws)|ws:WS•ws ∈ wss}
625  wdi:WDI = obs_part_WDIS(wis)
626  wdii:WDII = uid_WDII(wdi)
627  wfcs:WFCs = obs_part_WFCs(obs_part_WFCS(wis))
628  wfcis:WFI-set = {uid_WFCI(wfc)|wfc:WFC•wfc ∈ wfcs}
629  clk:CLK = obs_part_CLK(wis)
630  clki:CLKI = uid_CLKI(clk)
631  is:(WI|WSI|WDII|WFCI)-set = {wi}∪wsis∪{wdii}∪wfcis
```

7.4.2 Channels

632. Weather stations share weather data, WD, with the weather data interpreter — so there is a set of channels, one each, "connecting" weather stations to the weather data interpreter.
633. The weather data interpreter shares weather forecast requests, WFReq, and interpreted weather data (i.e., forecasts), WF, with each and every forecast consumer — so there is a set of channels, one each, "connecting" the weather data interpreter to the interpreted weather data (i.e., forecast) consumers.
634. The clock offers its current time value to each and every part, except the weather, of the WIS system.

```
channel
632  { ch_si[ wsi,wdii ]:WD | wsi:WSI•wsi ∈ wsis }
633  { ch_ic[ wdii,fci ]:(WFReq|WF) | fci:FCI•fci ∈ fcis }
634  { ch_cp[ clki,i ]:TIME | i:(WI|CLKI|WSI|WDII|WFCI)•i ∈ is }
```

7.4.3 WIS Behaviours

635. WIS behaviour, wis_beh, is the
636. parallel composition of all the weather station behaviours, in parallel with the
637. weather data interpreter behaviour, in parallel with the
638. parallel composition of all the weather forecast consumer behaviours, in parallel with the
639. clock behaviour.

value

```
635  wis_beh: Unit → Unit
635  wis_beh() ≡
636    || { ws_beh(uid_WSI(ws),mereo_WS(ws),...) | ws:WS•ws ∈ wss } ||
637    || wdi_beh(uid_WDI(wdi),mereo_WDI(wdi),...)(wd_rep,wf_rep) ||
638    || { wfc_beh(uid_WFCI(wfc),mereo_WDG(wfc),...) | wfc:WFC•wfc ∈ wfcs } ||
639       clk_beh(uid_CLKI(clk),mereo_CLK(clk),...)("November 15, 2021: 16:12 ")
```

7.4.4 Clock

640. The clock behaviour has a programmable attribute, t.
641. It repeatedly offers its current time to any part of the WIS system.
It nondeterministically internally "cycles" between
642. retaining its current time, or
643. increment that time with a "small" time interval, δ, or
644. offering the current time to a requesting part.

```
value
640.  clk_beh: clki:CLKI × clkm:CLKM → TIME →
641.      out {ch_cp[ clki,i ]|i:(WSI|WDII|WFCI)•i ∈wsis∪{wdii}∪wfcis }  Unit
640.  clk_beh(clki,is)(t) ≡
642.        clk_beh(clki,is)(t)
643.     ⊓ clk_beh(clki,is)(t⊕δ)
644.     ⊓ ( ⌈⌉⌊⌋{ ch_cp[ clki,i ] ! t | i:(WSI|WDII|WFCI)•i ∈ is } ; clk_beh(clki,is)(t) )
```

7.4.5 Weather Station

645. The weather station behaviour communicates with the global clock and the weather data interpreter.
646. The weather station behaviour simply "cycles" between sampling the weather, reporting its findings to the weather data interpreter and resume being that overall behaviour.
647. The weather station time-stamp "sample' the weather (i.e., meteorological information).
648. The meteorological information obtained is analysed with respect to temperature (distribution etc.),
649. humidity (distribution etc.),
650. wind (distribution etc.),
651. barometric pressure (distribution etc.), etcetera,
652. and this is time-stamp and location aggregated (mkWD) and "sent" to the (central ?) weather data interpreter,
653. whereupon the weather data generator behaviour resumes.

```
value
645  ws_beh: wsi:WSI × (clki,wi,wdii):WDGM × (LOC × ...) →
645     in ch_cp[ clki,wsi ] out ch_gi[ wsi,wdii ]  Unit
646  ws_beh(wsi,(clki,wi,wdii),(loc,...)) ≡
648     let tdl = attr_TDL(w),
649         hdl = attr_HDL(w),
650         wdl = attr_WDL(w),
```

```
651       bpl = attr_BPL(w), ... in
652     ch_gi[wsi,wdii] ! mkWD((ch_cp[clki,wsi] ?,loc),(tdl,hdl,wdl,bpl,...)) end ;
653     wdg_beh(wsi,(clki,wi,wdii),(loc,...))
```

7.4.6 Weather Data Interpreter

654. The weather data interpreter behaviour communicates with the global clock, all the weather stations and all the weather forecast consumers.
655. The weather data interpreter behaviour non-deterministically internally (⊓) chooses to
656. either collect weather data,
657. or calculate some weather forecast,
658. or disseminate a weather forecast.

```
value
654  wdi_beh: wdii:WDII×(clki,wsis,wfcis):WDIM×...→(WD_Rep×WF_Rep)→
654         in ch_cp[clki,wdii], { ch_si[wsi,wdii] | wsi:WSI•wsi ∈ wsis },
654         out { ch_ic[wdii,wfci] | wfci:WFCI•wfci ∈ wfcis }  Unit
654  wdi_beh(wdii,(clki,wsis,wfcis),...)(wd_rep,wf_rep) ≡
656     collect_wd(wdii,(clki,wsis,wfcis),...)(wd_rep,wf_rep)
655     ⊓
657     calculate_wf(wdii,(clki,wsis,wfcis),...)(wd_rep,wf_rep)
655     ⊓
658     disseminate_wf(wdii,(clki,wsis,wfcis),...)(wd_rep,wf_rep)
```

7.4.6.1 collect_wd

659. The collect weather data behaviour communicates with the global clock and all the weather stations – but "passes-on" the capability to communicate with all of the weather forecast consumers.
660. The collect weather data behaviour
661. non-deterministically externally offers to accept weather data from some weather station,
662. updates the weather data repository with a time-stamped version of that weather data,
663. and resumes being a weather data interpreter behaviour, now with an updated weather data repository.

```
value
659  collect_wd: wdii:WDII×(clki,wsis,wfcis):WDIM×...
659         → (WD_Rep×WF_Rep) →
659         in ch_cp[clki,wdii], { ch_si[wsi,wdii] | wsi:WSI•wsi ∈ wsis },
659         out { ch_ic[wdii,wfci] | wfci:WFCI•wfci ∈ wfcis }  Unit
660  collect_wd(wdii,(clki,wsis,wfcis),...)(wd_rep,wf_rep) ≡
661     let ((ti,loc),(hdl,tdl,wdl,bpl,...)) = ⊓⊓{wsi[wsi,wdii]?|wsi:WSI•wsi∈wsis} in
662     let wd_rep′ = update_wdr(ch_cp[clki,wdii]?,((ti,loc),(hdl,tdl,wdl,bpl,...)))(wd_rep) in
663     wdi_beh(wdii,(clki,wsis,wfcis),...)(wd_rep′,wf_rep) end end
```

7.4.6.2 calculate_wf

664. The calculate forecast behaviour communicates with the global clock – but "passes-on" the capability to communicate with all of weather stations and the weather forecast consumers.
665. The calculate forecast behaviour
666. non-deterministically internally chooses a forecast type from among a indefinite set of such,
667. and a current or "future" time-interval,
668. whereupon it calculates the weather forecast and updates the weather forecast repository,
669. and then resumes being a weather data interpreter behaviour now with the weather forecast repository updated with the calculated forecast.

```
value
664 calculate_wf: wdii:WDII×(clki,wsis,wfcis):WDIM×...→(WD_Rep×WF_Rep)→
664         in ch_cp[clki,wdii], { ch_si[wsi,wdii] | wsi:WSI•wsi ∈ wsis },
664         out { ch_ic[wdii,wfci] | wfci:WFCI•wfci ∈ wfcis }  Unit
665 calculate_wf(wdii,(clki,wsis,wfcis),...)(wd_rep,wf_rep) ≡
666     let tf:WWF = ft1 ⊓ ft2 ⊓ ... ⊓ ftn,
667         ti:(TIME×TIVAL) • toti≥ch_cp[clki,wdii] ? in
668     let wf_rep′ = update_wfr(calc_wf(tf,ti)(wf_rep)) in
669     wdi_beh(wdii,(clki,wsis,wfcis),...)(wd_rep,wf_rep′) end end
```

670. The calculate_weather forecast function is, at present, further undefined.

```
value
670. calc_wf: WFF × (TIME×TI) → WFRep → WF
670. calc_wf(tf,ti)(wf_rep) ≡ ,,,
```

7.4.6.3 disseminate_wf

671. The disseminate weather forecast behaviour communicates with the global clock and all the weather forecast consumers – but "passes-on" the capability to communicate with all of weather stations.
672. The disseminate weather forecast behaviour non-deterministically externally offers to received a weather forecast request from any of the weather forecast consumers, wfci, that request is for a specific format forecast, tf, and either for a specific time or for a time-interval, toti, as well as for a specific location, loc.
673. The disseminate weather forecast behaviour retrieves an appropriate forecast and
674. sends it to the requesting consumer –
675. whereupon the disseminate weather forecast behaviour resumes being a weather data interpreter behaviour

```
value
671 disseminate_wf: wdii:WDII×(clki,wsis,wfcis):WDIM×...→(WD_Rep×WF_Rep)→
671         in ch_cp[clki,wdii] in,out { ch_ic[wdii,wfci] | wfci:WFCI•wfci ∈ wfcis }  Unit
671 disseminate_wf(wdii,(clki,wsis,wfcis),...)(wd_rep,wf_rep) ≡
672     let mkReqWF((tf,toti,loc),wfci) = [] {ch_ic[wdii,wfci] ? | wfci:WFCI•wfci∈wfcis} in
673     let wf = retr_WF((tf,toti,loc),wf_rep) in
674     ch_ic[wdii,wfci] ! wf ;
675     disseminate_wf(wdii,(clki,wsis,wfcis),...)(wd_rep,wf_rep) end end
```

676. The retr_WF((tf,toti,loc),wf_rep) function invocation retrieves the weather forecast from the weather forecast repository most "closely" matching the format, tf, time, toti, and location of the request received from the weather forecast consumer. We do not define this function.

```
676. retr_WF: (WFF×(TIME×TI)×LOC) × WFRep → WF
676. retr_WF((tf,toti,loc),wf_rep) ≡ ...
```

We could have included, in our model, the time-stamping of receipt (formula Item 672) of requests, and the time-stamping of delivery of requested forecast in which case we would insert ch_cp[clki,wdii]? at respective points in formula Items 672 and 674.

7.4.7 Weather Forecast Consumer

677. The weather forecast consumer communicates with the global clock and the weather data interpreter.
678. The weather forecast consumer behaviour
679. nondeterministically internally either
680. selects a suitable weather cast format, tf,
681. selects a suitable location, loc′, and
682. selects, toti, a suitable time (past, present or future) or a time interval (that is supposed to start when forecast request is received by the weather data interpreter.
683. With a suitable formatting of this triple, mkReqWF(tf,loc′,toti), the weather forecast consumer behaviour "outputs" a request for a forecast to the weather data interpreter (first "half" of formula Item 682) whereupon it awaits (;) its response (last "half" of formula Item 682) which is a weather forecast, wf,
684. whereupon the weather forecast consumer behaviour resumes being that behaviour with it programmable attribute, d, being replaced by the received forecast suitably annotated;
679 or the weather forecast consumer behaviour
685. edits a display
686. and resumes being a weather forecast consumer behaviour with the edited programmable attribute, d′.

```
value
677 wfc_beh: wfci:WFCI × (clki,wdii):WFCM × (LOC × ...) → D →
677       in ch_cp[ clki,wfci ],
677       in,out { ch_ic[ wdii,wfci ] | wfci:WFCI•wfci ∈ wfcis } Unit
678 wfc_beh(wfci,(clki,wdii),(loc,...))(d) ≡
680     let tf = tf1 ⊓ tf2 ⊓ ... ⊓ tfn,
681         loc′:LOC • loc′=loc∨loc′≠loc,
682         (t,ti):(TIME×TI) • ti≥0 in
683     let wf = (ch_ic[ wdii,wfci ] ! mkReqWF(tf,loc′,(t,ti))) ; ch_ic[ wdii,wfci ] ? in
684     wfc_beh(wfci,(clki,wdii),(loc,...))((tf,loc′,(t,ti)),wf) end end
679   ⊓
685     let d′:D {\EQ} rndr\_D(d,{\DOTDOTDOT}) in
686     wfc_beh(wfci,(clki,wdii),(loc,...))(d′) end
```

The choice of location may be that of the weather forecast consumer location, or it may be one different from that. The choice of time and time-interval is likewise a non-deterministic internal choice.

7.5 Conclusion

7.5.1 Reference to Similar Work

As far as I know there are no published literature nor, to our knowledge, institutional or private works on the subject of modelling weather data collection, interpretaion and weather forecast delivery systems.

7.5.2 What Have We Achieved ?

TO BE WRITTEN

7.5.3 What Needs to be Done Next ?

TO BE WRITTEN

7.5.4 Acknowledgements

This technical cum experimental research report was begun in Bergen, Wednesday, November 9, 2016 – inspired by a presentation by Ms. Doreen Tuheirwe, Makarere University, Kampala, Uganda. I thank her, and Profs. Magne Haveraaen and Jaakko Järvi of BLDL: the Bergen Language Design Laboratory, Dept. of Informatics, University of Bergen (Norway), for their early comments, and Prof. Haveraaen for inviting me to give PhD lectures there in the week of Nov. 6–12, 2016.

Chapter 8

Automobile Assembly Lines [September 2021]

Contents

8.1 Introduction 143
8.2 A Domain Analysis & Description 144
8.2.1 An Initial Domain Sketch 144
8.2.2 Endurants 146
8.2.2.1 External Qualities 146
8.2.2.1.1 Parts 146
8.2.2.1.2 On Main Elements 149
8.2.2.1.2.1 General: 149
8.2.2.1.2.2 Assembly Line Element Types: 149
8.2.2.1.3 Automobile Manufacturing: A Wider Context 149
8.2.2.1.4 An Assembly Plant Taxonomy 150
8.2.2.1.5 Aggregate, Set, Core and Sibling Parts 151
8.2.2.1.5.1 Atomic and Compound Parts: 151
8.2.2.1.5.2 Aggregates and Sets: 151
8.2.2.1.5.3 Cores [Roots] and Siblings: 151
8.2.2.1.6 The Core State 152
8.2.2.1.6.1 State Narrative: 152
8.2.2.1.6.2 Endurant State Formalisation: 152
8.2.2.1.7 Invariant: External Qualities 154
8.2.2.2 Internal Qualities 155
8.2.2.2.1 Unique Identifiers 155
8.2.2.2.1.1 Common Unique Identifier Observer: 155
8.2.2.2.1.2 The Unique Identifier State: 156
8.2.2.2.1.3 An Invariant: 156
8.2.2.2.1.4 Part Retrieval: 156
8.2.2.2.1.5 The Unique Identifier Indexed Endurant State: 157
8.2.2.2.1.6 Taxonomy Map with Unique Identifier Labels: 157
8.2.2.2.1.7 Unique Identifier State Expressions: 157
ι **687.** AP, Assembly Plants 157
ι **688.** ALA, Assembly Line Aggregates 158
ι **689.** MAL, Main Assembly Lines 159
ι **690.** SALA, Supply Assembly Line Aggregates 159
ι **691.** SALs=SAL-set, Supply Assembly Line Sets 159
ι **692.** SAL, Supply Assembly Lines 160
ι **693.** SA, Station Aggregates 160
ι **694.** Ss=S-set, Station Set 160
ι **695.** S, Stations 160
ι **696.** ME, Main Elements 161
ι **697.** RA, Robot Aggregates 161

ι 698. Rs=R-set, Robot Sets . . . 161
ι 699. R, Robots . . . 161
ι 700. ES, Element Supplies . . . 161
ι 701. Es=E-set, Element Supply Sets . . 161
ι 702. E, Elements . . . 161
8.2.2.2.2 Mereology . . . 162
8.2.2.2.2.1 ι 687. AP: Assembly Plant: . . . 162
8.2.2.2.2.2 ι 688. ALA: Assembly Line Aggregate: 162
8.2.2.2.2.3 ι 689. MAL: Main Assembly Line: . . . 163
8.2.2.2.2.4 ι 690. SALA: Supply Assembly Line Aggregate: . . . 163
8.2.2.2.2.5 ι 691. SALs=SAL-set: Simple Assembly Line Set: . . . 164
8.2.2.2.2.6 ι 692. SAL: Simple Assembly Lines: . . . 164
8.2.2.2.2.7 ι 693. SA: Station Aggregate: . . . 165
8.2.2.2.2.8 ι 694. Ss = S-set: Station Sets: . . . 165
8.2.2.2.2.9 ι 695. S: Station: . . . 165
8.2.2.2.2.10 ι 696. ME: Main Elements: . . . 168
8.2.2.2.2.11 ι 697. RA: Robot Aggregate: . . . 168
8.2.2.2.2.12 ι 698. Rs=R-set: Robot Set: . . . 169
8.2.2.2.2.13 ι 699. R: Robot: . . . 169
8.2.2.2.2.14 ι 700. ES: Element Supply: . . . 169
8.2.2.2.2.15 ι 701. Es=E-set: Element Supply Set: . . 170
8.2.2.2.2.16 ι 702. E: Elements: . . . 170
Comments on the Mereology Presentation . . . 170
Distances of Stations from Outlet . . . 170
8.2.2.2.3 Attributes . . . 172
8.2.2.2.3.1 ι 687. AP: Assembly Plant: . . . 172
8.2.2.2.3.2 ι 688. ALA: Assembly Line Aggregate: 172
8.2.2.2.3.3 ι 689. MAL: Main Assembly Line: . . . 173
8.2.2.2.3.4 ι 690. SALA: Supply Assembly Line Aggregate: . . . 173
8.2.2.2.3.5 ι 691. SALs: Supply Assembly Line Set: . . . 173
8.2.2.2.3.6 ι 692. SAL: Supply Assembly Lines: . . . 173
8.2.2.2.3.7 ι 693. SA: Station Aggregate: . . . 173
8.2.2.2.3.8 ι 694. Ss=S-set: Station Set: . . . 174
8.2.2.2.3.9 ι 695. S: Station: . . . 174
8.2.2.2.3.10 ι 696. ME: Main Element: . . . 175
8.2.2.2.3.11 ι 697. RA: Robot Aggregate: . . . 175
8.2.2.2.3.12 ι 698. Rs=R-set: Robot Set: . . . 175
8.2.2.2.3.13 ι 699. R: Robot: . . . 175
8.2.2.2.3.14 ι 700. ES: Element Supply: . . . 176
8.2.2.2.3.15 ι 701. Es=E-set: Element Supply Set: . . 176
8.2.2.2.3.16 ι 702. E: Elements: . . . 176
8.2.2.3 Comments wrt. [70] . . . 176
8.2.3 Perdurants . . . 177
8.2.3.1 From Parts to Behaviours . . . 177
8.2.3.2 Channels . . . 178
8.2.3.3 Actors . . . 178
8.2.3.3.1 Actions and Events . . . 178
8.2.3.3.2 Behaviours . . . 178
8.2.3.4 System Initialisation . . . 178
8.3 Discussion . . . 178
8.4 Conclusion . . . 178
8.4.1 Models and Axioms . . . 178
8.4.2 Learning Forwards, Practicing In Reverse . . . 178
8.4.3 Diagrammatic Reasoning . . . 179
8.4.4 The Management of Domain Modeling . . . 179
8.4.5 ... one more section 180
8.4.6 ... a last section (?) 180
8.4.7 Acknowledgments . . . 180

We interpret Sect. 2 of [70]. That is, we present the domain description of a generic, assembly line manufacturing plant, like, for example, an automobile plant. The description is in the style of, i.e., according to the dogma of [55]. It is an aim of this report to (i) classify the various notions of [70] in their relationship to domain analysis & description notions of [55]: endurants and perdurants, external and internal endurant qualities: unique identifiers, mereologies and attributes, as well as domain versus requirements specifications, i.e., descriptions vs. prescriptions.

Caveat

The topic of this report is currently being studied and writing progresses according ly. I have not checked all item (etc.) references, but will, one day I have a printed copy to work from ! I have also left many stubs to be resolved. Various sections represent "diverse" modeling attempts. It will be interesting to see which will "survive" ! Since this report will be updated on the net daily You may wish to not download-copy it, but to reload it, from day-to-day, if need be.

November 15, 2021: 16:12 : "Progress"

- Mereology "finished".
- "Finished" first round of Attributes.
- Speculating on robot tasks.
- Unfinished "business" wrt. parts and robot operations.

A Development Document

This report cum paper, may look like a paper. But it is not. It is a report on *"work in progress"*.

It expresses, in its current form, the way we would, sequentially, develop en experimental domain model, such as mentioned in Sect. **8.4.4** on page 179, in the item labeled **Experiment** on Page 179.

8.1 Introduction

The current author has put forward a theory and a methodology of domain engineering [48, 52, 55]. That methodology is the result of 30 years of experimental development of analyses & descriptions of numerous domains. See the bibliography entry for [60] to see the variety of domains so studied. Isolated aspects of the domain of assembly line manufacturing has been a topic of study, also in computing science, for some years. See, for example, `https://en.wikipedia.org/wiki/Cellular_manufacturing`. These computing science studies have, however, focused, less on overall assembly lines, and more on their individual manufacturing cells – in this report referred to as operators (or stations (?)). So when I heard of and read [70] I was ready to myself tackle the domain analysis & description of an "entire" production line, i.e., a single assembly line complex of a main and possibly several supply assembly lines.

MORE TO COME

8.2 A Domain Analysis & Description

8.2.1 An Initial Domain Sketch

We refer to Fig. 8.5 on page 146. In this section we shall give an informal sketch of the domain. The domain is that of the generic *assembly line* "core" of a manufacturing plant – think of an automobile factory ![28]

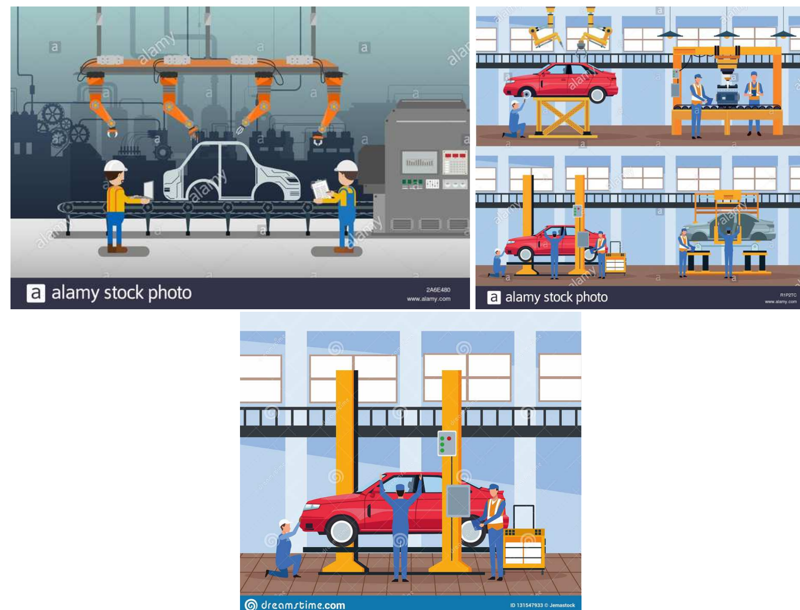


Fig. 8.1 Aspects of an Automobile Assembly Line, I

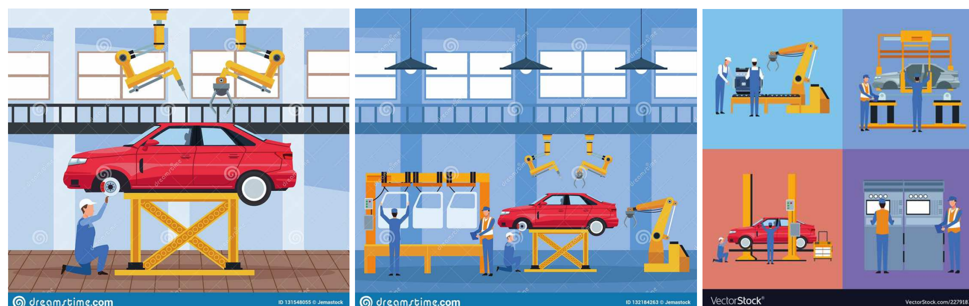


Fig. 8.2 Aspects of an Automobile Assembly Line, II

[28] For the specific case of automobile factories the assembly line focus thus omits consideration of number of major components: the motor foundry etc., the paint shop, etc.

Fig. 8.3 Aspects of an Automobile Assembly Line,III

Fig. 8.4 Aspects of an Automobile Assembly Line,IV

We thus focus sôlely on *assembly lines*[29,30]. Figure 8.5 shows an idealised layout of an assembly line. It shows one *main assembly line* and three *supply assembly lines*. Assembly lines assemble, as we shall call them, *elements*.[31] Assembly of elements, from other, the constituent, elements are performed by *robots*[32] at stations. *Stations* are linearly ordered within an assembly line. Assembly lines has a *flow* direction, i.e., the direction in which increasingly "bigger" elements "flow". Each station consists of one or more *robots*. Robots direct their work at a *main element*, and apply their grips to elements supplied from an *element supply*,[33] or to a "larger" assembly "fetched" from a *supply assembly line* incident at that *station* !

[29] https://en.wikipedia.org/wiki/Assembly_line: *An* ***assembly line*** *is a manufacturing process (often called a progressive assembly) in which parts (usually interchangeable parts) are added as the semi-finished assembly moves from workstation to workstation where the parts are added in sequence until the final assembly is produced. By mechanically moving the parts to the assembly work and moving the semi-finished assembly from work station to work station, a finished product can be assembled faster and with less labor than by having workers carry parts to a stationary piece for assembly.*

Assembly lines are common methods of assembling complex items such as automobiles and other transportation equipment, household appliances and electronic goods.

[30] Example supply assembly lines are: (i) engine assembly (where the start of such lines are supplied with already prepared engine blocks (from a non-assembly line engine foundry and machining shop), (ii-v) four left and right front and rear door assemblies, (vi-ix) body interior left and right front and rear sofa, and panel assemblies.

[31] Other, perhaps more common terms are: products or parts. The term 'part' is used in our domain analysis & description method, [48, 52, 55], for quite other purposes –so that is "out !"

[32] Robots are either humans assisted by various machine tools, as in Charlie Chaplin's movie: *'Modern Times'* (1936), or are, indeed, robots.

[33] That is, a station local storage of elements that are to be joined, at a station, by the help of robots, to the main element. How the supply elements are introduced to the supply is currently left unspecified.

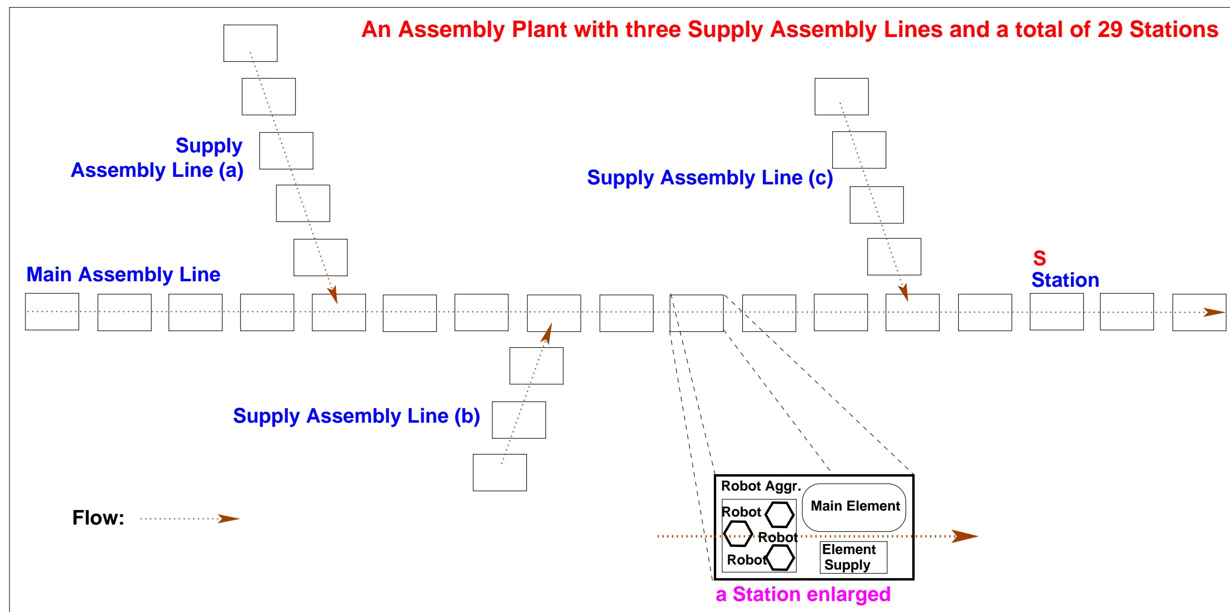


Fig. 8.5 A simplified *Assembly Plant* diagram

8.2.2 Endurants

The endurant analysis & description is according to the ontology graph of Fig. 8.6 on the facing page. The analysis & description is otherwise according to either of [48, 52, 55]. It suffices to have studied [52].

8.2.2.1 External Qualities

The *domain analyser cum describer*[34], who is assumed fully familiar with the domain analysis & description method, [55], starts with analysing and describing external qualities of the domain. In the case of an assembly plant these are the solid endurants, or, as they are called in [55], the parts. The domain analyser cum describer, from being familiar with the method, therefore is, all the time, aware that these (described) parts will, in the transition to the analysis & description of perdurants, be transcendentally deduced, i.e., “morphed” into behaviours. It is this a-priory knowledge that guides the analyser cum describer’s determination as to whether parts are to be modeled as atomic or as compounds, and the decomposition of compound parts into atomic, aggregate and set parts.

8.2.2.1.1 Parts

The domain, that is, the universe of discourse, is that of an **assembly plant** – say for automobiles, for machinery or for electronic gadgets.

687. In an **assembly plant**, AP, we can observe

[34] The notion of *’domain analyser cum describer’* covers one, individually (as the author of this paper) working person, or a well-managed group of two or more persons, all “equally” familiar with the method of [55].

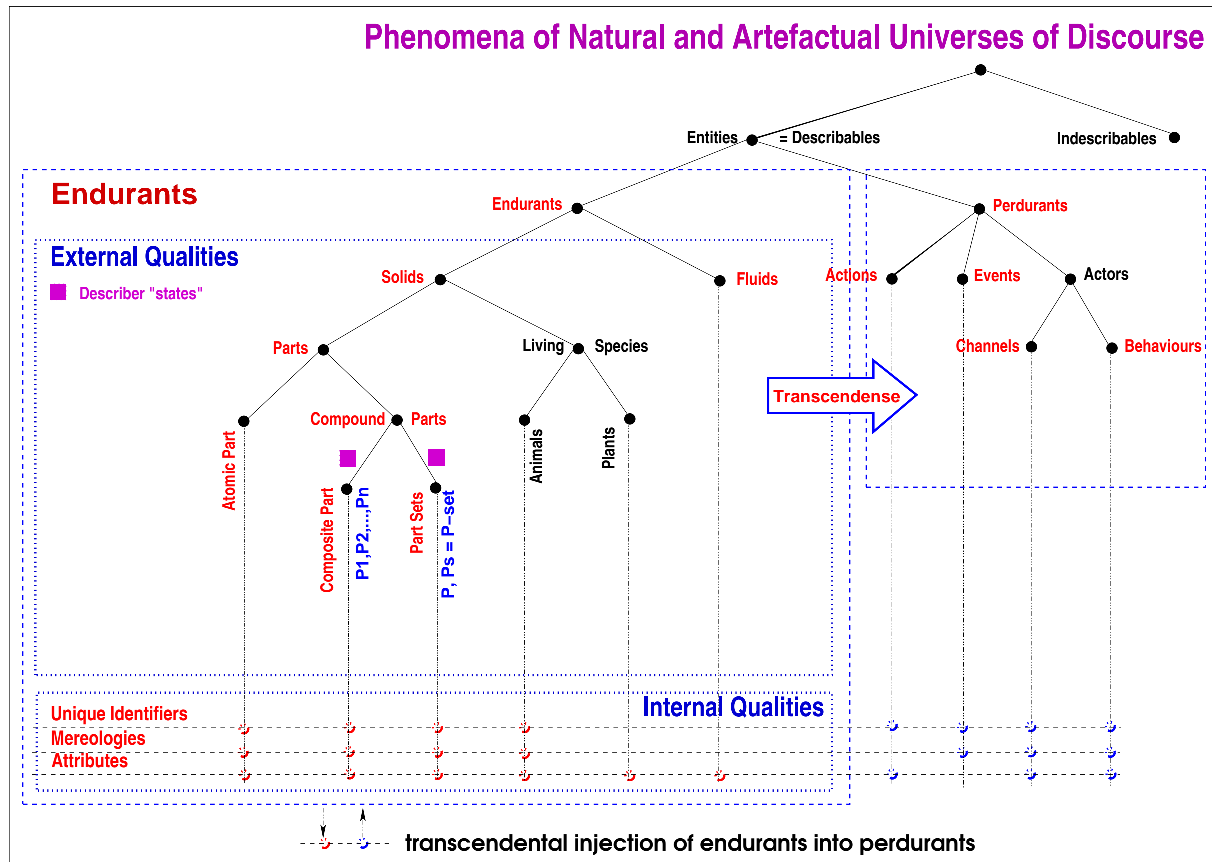


Fig. 8.6 The simple *Ontology Graph* underlying our *Analysis & Description*

688. an assembly line aggregate, ALA[35].
689. From an assembly line aggregate one can observe a composite of a main assembly line, MAL,
690. and aggregate of supply assembly lines, SALA,
691. with the latter being sets SALs = SAL-**set**.
692. of supply assembly lines,
693. From main and supply assembly lines one can observe aggregates of stations, SA,
694. which are sets Ss = S-**set**[36]
695. of two or more stations S. From a station one can observe
696. a main element, ME, (an assembly[37]),
697. an aggregate robot, RA, which is
698. a set, Rs = R-**set**,[38] of
699. one or more robots, R, and

[35] We omit observations of *motor works (foundry, machining, etc.), body shop (pressing, etc.), paint shop*, etc.

[36] **Linear Lines:** The mereology of sect. **8.2.2.2.2** will order these in a linear sequence

[37] https://www.merriam-webster.com/dictionary/assembly: ***Assembly:*** *the fitting together of manufactured elements into a complete machine, structure, or unit of a machine*

[38] [70]: **Our interpretation of 'operator':** robot perform processes which consists of tasks. These are perdurants, that is, an operator will, subsequently, in this report be transcendentally "morphed" into s set of one or more concurrent processes. These processes are then subject, in the domain model, to invariants, and in a subsequent requirements "model" into constraints.

700. an aggregate, ES, of [“supply”] elements[39],
701. which are sets, ESs = E-**set**,
702. of manufacturing elements E.

type
687. AP
688. ALA
689. MAL
690. SALA
691. SALs = SAL-**set**
692. SAL
693. SA
694. Ss = S-**set**
695. S
696. ME
697. RA
698. Rs = R-**set**
699. R
700. ES
701. Es = E-**set**
702. E
value
688. obs_ALA: AP → ALA
689. obs_MAL: ALA → MAL
690. obs_SALA: ALA → SALA
691. obs_SALs: SALA → SALs
693. obs_SA: (MAL|SAL) → SA
694. obs_Ss: SA → Ss
696. obs_ME: S → ME
697. obs_RA: S → RA
698. obs_Rs: RA → Rs
700. obs_ES: S → ES
701. obs_Es: ES → Es
axiom
694. ∀ ss:Ss • **card** ss > 1
698. ∀ rs:Rs • **card** rs > 0

Figure 8.7 repeats Fig. 8.5 on page 146, but now marked with the names of composite sorts introduced in Items 687–701.

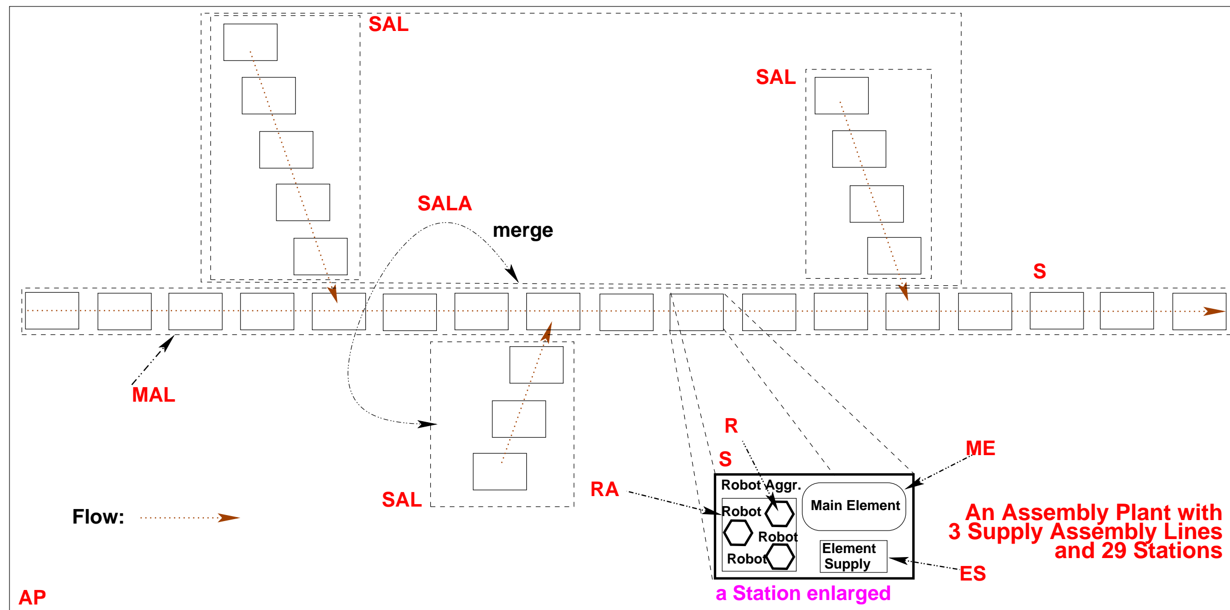


Fig. 8.7 A simplified *Assembly Plant* diagram marked with composite endurant sort names

[39] These are local storage, usually simple, mostly atomic solid or fluid elements such as bolts & nuts, glue, etc.

8.2.2.1.2 On Main Elements

This section is an aside.

In this section we shall discuss what is meant here by a main element, that is, "what is in store" – what will/might come up later on.

8.2.2.1.2.1 General:

The *main element* is here modeled as a solid endurant. It is a *"place-holder"* for "the thing for which the manufacturing plant" is intended. The plan is to endow main elements with an attribute [Sect. **8.2.2.2.3.10** on page 175]: That attribute may itself be thought of as being a solid endurant. We shall then use the term *part*[40] Robots, then, perform operations on the main element. These operations are functions, which are attributes of robots. As functions they take the main element [main] part attribute and a set of element supply elements and yield an updated main element part. So You may think of the main element as a "container" for that main part. There may be no contents of the container, in which case the main element's part attribute is "`nil`". Its content is "received" from the main element of the previous station, if there is one, else from an element supply. A content that a station can no longer contribute to is "passed" on to the next station, or "fused" in, if from a supply assembly line, as a supply element, to a main assembly line elements, or, to an outside, the "ready product!"

8.2.2.1.2.2 Assembly Line Element Types:

The type of main elements is a pair: the type that is stroven for, that is, the assembly line type, and the type of the element "currently residing in" the main elements. So each station is particularly typed by its "current" main element type.

8.2.2.1.3 Automobile Manufacturing: A Wider Context

These are, roughly the principal components of automobile manufacturing[41]:

- **Chassis:** The chassis of the car is the baseline component. All other parts are integrated on, or within the chassis. This is typically a welded frame that's initially attached to a conveyor that moves along a production line. As the frame progresses, the car is literally "built from the frame up" to create a final product. Parts that are sequentially applied to the chassis include the engine, front and rear suspension, gas tank, rear-end and half-shafts, transmission, drive shaft, gear box, steering box, wheel drums and the brake system.
- **Body:** Once the "running gear" is integrated within the frame, the body is constructed as a secondary process. First, the floor pan is positioned properly, then the left and right quarter panels are positioned and welded to the floor structure. This step is followed by adding the front/rear door pillars, the body side panels, rear deck, hood and roof. The entire process is typically executed by robotic machines.
- **Paint:** Before painting the vehicle, a quality control team inspects the body as it sits. Skilled workers look for dents, abrasives or other deformations that could create a finishing problem when undergoing the painting process. Once this step is completed, the car is automatically "dipped" with primer, followed by a layer of undercoat and dried in a heated paint bay. Once

[40] Not to be confused with the Design Analysis & Description concept of parts, i.e., solid endurants.

[41] This account is taken, *ad verbatim*, from: `https://itstillruns.com/car-manufacturing-process-5575669.html`.

the primer/undercoat process is finished, the car is again "dipped" with the base coat and again dried before moving the assembly to the next stage.

- **Interior:** After the structure is entirely painted, the body is moved to the interior department in the plant. There, all of the internal components are integrated with the body. These components include: instrumentation, wiring systems, dash panels, interior lights, seats, door/trim panels, headliner, radio, speakers, glass, steering column, all weather-striping, brake and gas pedals, carpeting and front/rear fascias.
- **Chassis/Body Mating:** The two central major assemblies are next mated for final setup and roll-out. Again, this process is executed via computer and control machines to ensure speed, and perfect the fit between the body assembly and the chassis. Once the car is rolling on its own, it's driven to the final quality control point, inspected and placed in a waiting line for transportation to its final dealer destination.

8.2.2.1.4 An Assembly Plant Taxonomy

Figure 8.8 "graphs" the composition of solid endurants of an assembly plant according to the endurant composition of Items 687– 701 on page 148.

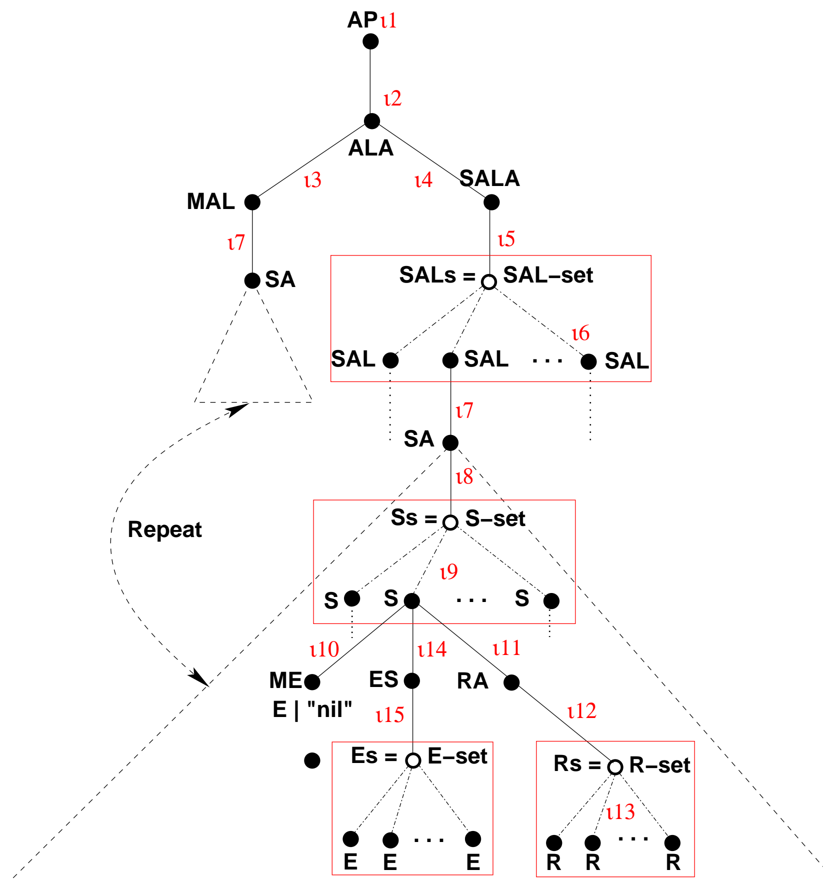


Fig. 8.8 The Composition of Solid Endurants of an Assembly Plant (AP)
Larger **red** framed boxes designate set parts.
$\iota i, \pi p$ refer to item i page 148.

Some diagram explications: (i) the top left dashed triangle shall show an endurant composition as does the main, large, dashed triangle; (ii) the vertical dotted lines "hanging down" from two SALs "hint" at the "tree" emanating down from the "middle" SAL; and (iii) the horisontal dots, . . . , in SAL, S, R and E "lines" hint at any number, 0 or more, of these endurants !

8.2.2.1.5 Aggregate, Set, Core and Sibling Parts

We review[42], as an aside, the [55, Monograph] concepts of *atomic, compound, aggregates, sets*, *core* or *root*, and *sibling* parts.

8.2.2.1.5.1 Atomic and Compound Parts:

Atomic parts are solid endurants whose possibly "internal" composition is ignored. Compound parts are solid endurants which we further analyse into *core* (or, equivalently, *root*) and *sibling* parts.

8.2.2.1.5.2 Aggregates and Sets:

Compound parts are either *composites* of one or more parts of different sorts, i.e., like Cartesians, but where we avoid modeling a composite as a Cartesian of a definite number of parts – since we may, "later", wish to "add" additional parts, or are [finite] *sets* of zero, one or more parts of the same sort.
We use the term aggregate to cover both kind of compounds. Usually, however, we use aggregates for composites, and sets for sets !

8.2.2.1.5.3 Cores [Roots] and Siblings:

With compound parts we distinguish between the core part and the sibling parts.

The core part is understood as follows: It is to be considered a "proper" part although it may sometime be more of an abstraction than a solid ! Consider the following example: a car, as seen from the point of view of an automobile plant, is a composite, with a core, the car as a whole, as "somehow" embodied in the overall software that monitors and co-controls various of the car's siblings; these siblings are then further aggregates, each with their cores and siblings. Immediate car siblings could be the chassis, the motor, the engine train, the body. The chassis, as an aggregate, has, usually, four wheels, etc. The body, as an aggregate, has, perhaps, four doors, a trunk and a hood. And each of these, the *chassis, motor, engine train, body,* etc., has their cores.

We now "formalise" the notion of the core of a compound.

- We do so by introducing an otherwise not further defined [binary or distributive] operator, $\ominus$.
- It applies to a pair of parts:
  - one is an aggregate, cp, and
  - the other, sp, is
    - either a single one of its siblings, p_i,
    - or a set of these, $\{p_i,p_j,..,p_k\}$,
  - i.e., $cp \ominus ps$, "somehow removes" ps from p.

We now apply the $\ominus$ systematically to all components of our domain.

[42] The treatment of part *cores* (in [55] called *roots*) is here augmented with the $\ominus$ operation – not mentioned in [55].

- The *'core'* of an atomic part is that part.
- The *'core'* of a composite part is that part *"minus"* (⊖) its *"sibling"* parts:
  - Let p be a composite part,
  - then p_1 = obs_P_1(p), p_2 = obs_$P_2(p)$, ..., p_m = obs_$P_m(p)$ are its sibling parts
  - [where obs_P_1, obs_P_2, ..., obs_P_m are the observers of parts p (of type P)].
  - The 'core' of p, i.e., core_P(p), is then $p \ominus \{p_1, p_2, ..., p_m\} : P_\kappa$.
- The *'core'* of a set of parts is that part *"minus"* (⊖) its *"sibling"* parts:
  - Let ps be a set part (of type $Ps = Q-$**set**),
  - 'core' of ps, i.e., core_Ps(ps), is then $ps \ominus \{q_1, q_2, ..., q_n\} : Ps_\kappa$.

Subsequently introduced *unique identifier, mereology* and *attribute observers* apply to core parts as they do to non-core parts.

8.2.2.1.6 The Core State

To encircle the notion of domain core states we need characterise:

- the state narratively, Sect. **8.2.2.1.6.1**; and
- the state formally, Sect. **8.2.2.1.6.2**.

8.2.2.1.6.1 State Narrative:

We shall now narrate the assembly plant domain state. We start by referring to Fig. 8.9 on the facing page.

703. We shall model the assembly plant state, σ, by a set of κore parts composed as follows:

ι 687 π 146	[43] (AP) the *a*ssembly *p*lant κore	ap_κ:AP$_\kappa$;
ι 688 π 147	(ALA) the *a*ssembly *l*ine *a*ggregate κore	ala_κ:ALA$_\kappa$;
ι 689 π 147	(MAL) the *m*ain *a*ssembly *l*ine κore,	mal_κ:MAL$_\kappa$;
ι 690 π 147	(SALA) the *a*ggregate of *s*upply *a*ssembly *l*ines κore,	$sala_\kappa$:SALA$_\kappa$;
ι 691 π 147	(SALs) the *c*onsolidated[44] *s*et of all sets of *a*ssembly *l*ine κores,	$csals_\kappa$:ALs$_\kappa$;
ι 693 π 147	(SA) the *c*onsolidated[45] *s*et of all *s*tation *a*ggregates	$cssa_\kappa$:SA$_\kappa$-**set**;
ι 694 π 147	(Ss) the *c*onsolidated *s*et of all assembly lines' *s*tation set κores,	css_κ:Ss$_\kappa$;
ι 695 π 147	(S) the *c*onsolidated *s*et of all assembly lines' *s*tation κores,	cs_κ:S$_\kappa$;
ι 696 π 147	(ME) the *c*onsolidated *s*et of all *m*ain *e*lement κores,	$csme_\kappa$:ME$_\kappa$;
ι 697 π 147	(RA) the *c*onsolidated *s*et of all *r*obot *a*ggregates κores,	$csra_\kappa$:RA$_\kappa$-**set**;
ι 698 π 147	(Rs) the *c*onsolidated *s*et of all *r*obot *s*et κores,	$csrs_\kappa$:Rs$_\kappa$;
ι 699 π 147	(R) the *c*onsolidated *s*et of all *r*obot κores,	csr_κ:Rs$_\kappa$;
ι 700 π 148	(ES) the *c*onsolidated *s*et of all *e*lement *s*upply κores,	$cses_\kappa$:ES$_\kappa$-**set**;
ι 701 π 148	(Es) the *c*onsolidated *s*et of all κore *e*lements,	cse_κ:E$_\kappa$-**set**;

704. as a set, σ_{s_κ}.

8.2.2.1.6.2 Endurant State Formalisation:

705. We consolidate the main and the supply assembly lines into one kind of assembly line, AL,
706. and their corresponding [consolidated] sets.

[43] ι *item* π *page* labels refers to narrative *item* on *page*; the corresponding formalisation is found on page[s] 148–148.
[44] – from both the main assembly line and from all the supply assembly lines
[45] Henceforth, by consolidated, we mean as in footnote 44.

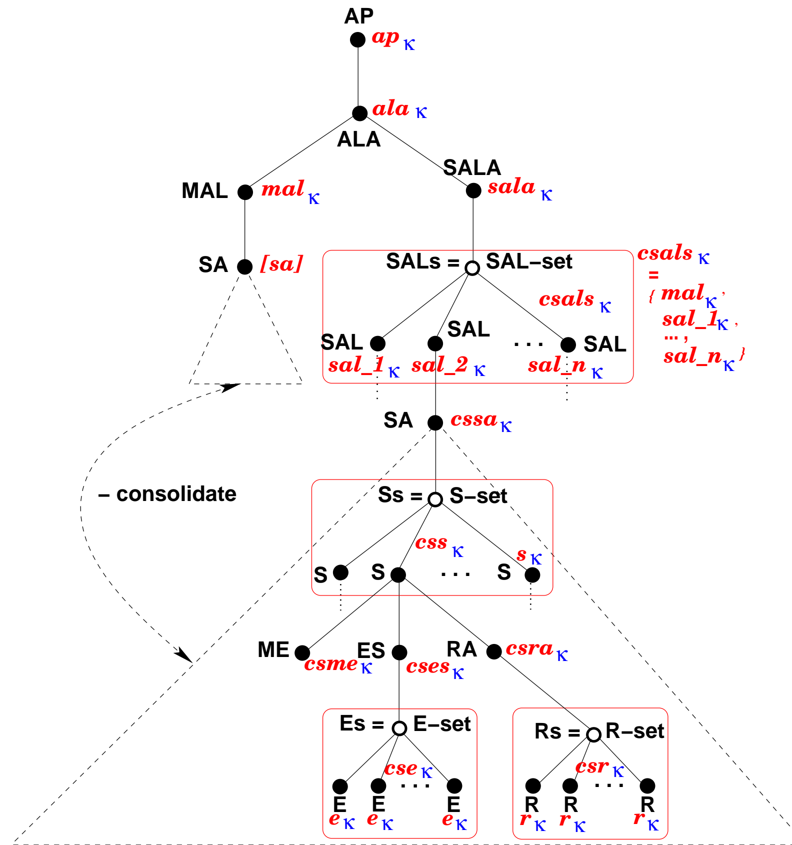


Fig. 8.9 Red/Blue text labels designate contributions to domain state

type
705. AL = MAL | SAL
706. ALs = AL-**set**
vaue
ι 687 π 146. *ap*:AP
ι 687 π 146. ap_κ:AP_κ = core_AP(*ap*)

ι 688 π 147. *ala*:ALA = obs_ALA(*ap*)
ι 688 π 147. ala_κ:ALA_κ = core_ALA(*ala*)

ι 689 π 147. *mal*:MAL = obs_MAL(*ala*)
ι 689 π 147. mal_κ:MAL_κ = core_MAL(*ala*)

ι 690 π 147. *sala*:SALA = obs_SALA(*ala*)
ι 690 π 147. $sala_\kappa$:SALA_κ = core_SALA(*sala*)

ι 691 π 147. *csals*:AL-**set** = {*mal*} ∪ obs_SALs(*sala*)
ι 691 π 147. $csals_\kappa$:AL-$\mathbf{set}_\kappa$ = ∪{core_AL(al)|al:AL•al ∈ *csal*}

ι 693 π 147. *cssa*:SA-**set** = {obs_SA(al)|al:AL•al ∈ *csal*}
ι 693 π 147. $cssa_\kappa$:SA-**set**$_\kappa$ = ∪{core_SA(sa)|sa:SA•sa ∈ *cssa*}

ι 694 π 147. *css*:S-**set** = ∪{obs_Ss(sa)|sa:SA•sa ∈ *cssa*}
ι 694 π 147. css_κ:S-**set**$_\kappa$ = ∪{core_Ss(obs_Ss(s))|s:S•s ∈ *css*}

ι 695 π 147. *cs*:S-**set** = ∪ *css*
ι 695 π 147. cs_κ:S-**set**$_\kappa$ = {core_S(s)|s:S•s ∈ *cs*}

ι 696 π 147. *csme*:ME-**set** = ∪{obs_ME(s)|s:S•s ∈ *css*}
ι 696 π 147. $csme_\kappa$:ME-**set**$_\kappa$ = core_ME(*csme*)

ι 697 π 147. *csra*:RA-**set** = ∪{obs_RA(s)|s:S•s ∈ *css*}
ι 697 π 147. $csra_\kappa$:RA-**set**$_\kappa$ = core_RA(*csra*)

ι 698 π 147. *csrs*:R-**set** = ∪{obs_Rs(ra)|ra:RA•ra ∈ *csra*}
ι 698 π 147. $csrs_\kappa$:$_\kappa$ = core_R(*csrs*)

ι 699 π 147. *crs*:R-**set** = ∪{obs_Rs(ra)|ra:RA•ra ∈ *csra*}
ι 699 π 147. crs_κ:$_\kappa$ = core_R(*csrs*)

ι 700 π 148. *cses*:ES = ∪{obs_ES(s)|s:S•s ∈ *csra*}
ι 700 π 148. $cses_\kappa$:$_\kappa$ = core_ES(*cses*)

ι 701 π 148. *cse*:E-**set** = ∪{obs_Es(es)|es:ES•s ∈ *cses*}
ι 701 π 148. cse_κ:$_\kappa$ = core_E(*cse*)

704. σ_s:(AP|ALA|MAL|SALA|SALs|Ss|S|ME|RA|Rs|R|ES|Es|E)-**set** =
704. $\{ap_\kappa\} \cup \{ala_\kappa\} \cup \{mal_\kappa\} \cup \{sala_\kappa\} \cup csals_\kappa \cup cssa_\kappa \cup$
704. $css_\kappa \cup cs_\kappa \cup csme_\kappa \cup csra_\kappa \cup csrs_\kappa \cup csr_\kappa \cup cses_\kappa \cup cse_\kappa$

8.2.2.1.7 Invariant: External Qualities

707. No two assembly lines, whether main or supply, are equal;
708. no two stations in same or different assembly lines are equal;
709. no two robots in different stations are equal;
710. no two main elements are equal;
711. no two element supplies in different stations are equal;
712. etc.

707. ∀ al_i,al_j:AL•{al_i,al_j}⊆*csal* ⇒
707.
707. to come
707.
708. ∀ s_i,s_j:S•{s_i,s_j}⊆*csal* ⇒
708.
708. to come
708.
709. ∀ r_i,r_j:R•{r_i,r_j}⊆*csr* ⇒
709.

709. to come
709.
710. ∀ me_i,me_j:ME•{me_i,me_j}⊆*csme* ⇒
710.
710. to come
710.
711. ∀ es_i,es_j:ES•{es_i,es_j}⊆*cses* ⇒
711.
711. to come
711.
712. ...

8.2.2.2 Internal Qualities

External qualities can be said to represent manifestation: that an endurant can be seen and touched. Internal qualities gives "contents" to the manifests in three ways:

- by the obvious endowment of solid endurants with *unique identification* [55, Sect. 5.2],
- by stating relations between solid endurants, whether topological or conceptual, e.g., operational, in the form of *mereologies* [55, Sect. 5.3], and
- by giving "flesh & blood, body & soul" to these endurants in the form of wide ranging *attributes* [55, Sect. 5.4].

8.2.2.2.1 Unique Identifiers

We shall show that many of the concerns of [70] have their "root" in the unique identification of solid endurants of the domain.

713. All parts, whether compound or atomic, have unique identifiers.

type
713. API, ALAI, MALI, SALAI, SALsI, SALI, SAI, Ss, SI, MEI, RAI, RsI, RI, ESI, EsI, EI

value
713. uid_AP: AP→API,
713. uid_ALA: AL→ALAI,
713. uid_MAL: MAL→MALI,
713. uid_SALA: SALA→SALAI,
713. uid_SALs: SALs→SALsI,
713. uid_SAL: SAL→SALI,
713. uid_SA: SA→SAI,
713. uid_Ss: Ss→SsI,
713. uid_S: S→SI,
713. uid_ME: ME→MEI,
713. uid_RA: RA→RAI,
713. uid_Rs: Rs→RsI,
713. uid_R: R→RI,
713. uid_ES: ES→EI,
713. uid_E: E→EI

8.2.2.2.1.1 Common Unique Identifier Observer:

714. Given that is_$P(p)$ holds if p is of type P, and is false otherwise, we can define a common unique identifier observer function for all assembly plant types.

type
714. P = AP|ALA|MAL|SALA|SALs|SA|Ss|S|ME|RA|Rs|R|ES|Es|E
714. PI = API|ALAI|MALI|SALAI|SALsI|SAI|SsI|SI|MEI|RAI|RsI|RI|ESI|EsI|EI
value
714. uid: P → PI

value
714. uid(p) ≡
714. is_AP(p)→uid_AP(p),
714. is_ALA(p)→uid_ALA(p),
714. is_MAL(p)→uid_MAL(p),
714. is_SALA(p)→uid_SALA(p),
714. is_SALs(p)→uid_SALs(p),
714. is_SA(p)→uid_SA(p),
714. is_Ss(p)→uid_Ss(p),
714. is_S(p)→uid_S(p),
714. is_ME(p)→uid_ME(p),
714. is_RA(p)→uid_RA(p),
714. is_Rs(p)→uid_Rs(p),
714. is_R(p)→uid_R(p),
714. is_ES(p)→uid_ES(p),
714. is_Es(p)→uid_Es(p),
714. is_E(p)→uid_E(p),
714. _→**false**

8.2.2.2.1.2 The Unique Identifier State:

As for endurant parts, cf. Sect. **8.2.2.1.6.2** on page 152, we can define a state of all endurant parts' unique identifiers. To do so we make use of the uid_E observers as also being distributive, that is, if uid_E is applied to a set of solid endurants, say $\{e_1,e_2,...,e_n\}$, then the result is $\{uid_E(e_1),uid_E(e_2),...,uid_E(e_n)\}$.

687_{uid}. *uid_ap* = uid_AP(*ap*)
688_{uid}. *uid_ala* = uid_ALA(*ala*)
689_{uid}. *uid_mal* = uid_MAL(*mal*)
690_{uid}. *uid_sala* = uid_SALA(*sala*)
691_{uid}. *uid_csal* = ∪{uid_SAL(sal)|sal:SAL•sal ∈ *csal*}
692_{uid}. *uid_csals* = ∪{uid_SALs(sals)|sals:SALs•sals ∈ *csals*}
693_{uid}. *uid_cssa* = ∪{uid_SA(sa)|sa:SA•sa ∈ *cssa*}
694_{uid}. *uid_css* = ∪{uid_Ss(ss)|ss:Ss•ss ∈ *css*}
695_{uid}. *uid_cs* = ∪{uid_S(s)|s:S•s ∈ *css*}
696_{uid}. *uid_csme* = ∪{uid_ME(me)|me:ME•me ∈ *csme*}
697_{uid}. *uid_csra* = ∪{uid_RA(ra)|ra:RA•ra ∈ *csra*}
698_{uid}. *uid_csrs* = ∪{uid_Rs(rs)|rs:Rs•rs ∈ *csrs*}
699_{uid}. *uid_csr* = ∪{uid_R(R)|r:R•r ∈ *csr*}
700_{uid}. *uid_cses* = ∪{uid_ES(es)|es:ES•es ∈ *cses*}
701_{uid}. *uid_cse* = ∪{uid_Es(es)|es:Es•es ∈ *cse*}

704_{uid}. *uid_*s_σ:(AP|AL|MAL|SALA|SALs|SAL|SA|Ss|S|ME|RA|Rs|R|ES|Es)-**set** =
704_{uid}. {*uid_ap*} ∪ {*uid_ala*} ∪ {*uid_mal*} ∪ {*uid_sala*} ∪ *uid_csal* ∪ *uid_cssa* ∪
704_{uid}. *uid_css* ∪ *uid_csme* ∪ *uid_csra* ∪ *uid_csr* ∪ *uid_cses* ∪ *uid_cse*

8.2.2.2.1.3 An Invariant:

715. All parts are uniquely identified, cf. Item 704 on page 154 and Item 704_{uid} on page 156.

715. **card** s_σ = **card** *uid_*s_σ

8.2.2.2.1.4 Part Retrieval:

716. From a unique identifier of a domain and the domain endurant state we can obtain the identified endurant.

value
716. retr_End: UI → P-**set** → P
716. retr_End(ui)(σ) ≡ **let** p•p ∈ σ ∧ uid(p) = ui **in** p **end**
axiom
716. σ = s_σ ∧ ui ∈ *uid*_s_σ ∧ p ∈ s_σ

8.2.2.2.1.5 The Unique Identifier Indexed Endurant State:

We can define a map from unique identifiers of endurant parts to these.

value
704. σ_{uid} =
687_σ. [*uid_ap* ↦ *ap*,
688_σ. *uid_ala* ↦ *ala*,
689_σ. *uid_mal* ↦ *mal*,
690_σ. *uid_sala* ↦ *sala*,
691_σ. *uid_csal* ↦ *csal*,
692_σ. *uid_csals* ↦ *csals*,
693_σ. *uid_cssa* ↦ *cssa*,
694_σ. *uid_css* ↦ *css*,
695_σ. *uid_cs* ↦ *cs*,
696_σ. *uid_csme* ↦ *csme*,
697_σ. *uid_csra* ↦ *csra*,
698_σ. *uid_csrs* ↦ *csrs*,
699_σ. *uid_csr* ↦ *csr*,
700_σ. *uid_cses* ↦ *cses*,
701_σ. *uid_cse* ↦ *cse*]

We leave it to the reader to state the type of the σ_{uid} value !

8.2.2.2.1.6 Taxonomy Map with Unique Identifier Labels:

Figure 8.10 on the following page[46] repeats Figs. 8.8 on page 150 and 8.9 on page 153. In Fig. 8.10 lines are now labeled with appropriate unique identifiers. This leads up to Fig. 8.11 on page 159.

Figure 8.11 on page 159 is a first, a graphical, two-dimensional expression. We shall comment on the graphics.

First one may say that Fig. 8.11 shows "horisontally" what Figs. 8.8–8.10 shows "vertically".

Then we note that compound composites and compound sets are expressed as maps from unique part identifiers to parts (which include these unique identifiers).

And finally we note that each compound part is expressed as a pair: pκ,map, the pκ labels the upper left outside of the map – such that the parentheses of the pair, (pκ,map), is shown just before pκ and ends after map.

8.2.2.2.1.7 Unique Identifier State Expressions:

We now present the proper *Unique Identifier State Expression* formula sketched in Fig. 8.11. It will be defined in terms of the *generate unique identifier state expression* function g_uise.

[687 on page 146] AP, Assembly Plants

717. The *unique identifier state expression* for the *a*ssembly *p*lant is the pair of the assembly plant core, *ap*κ, and the unique identifier state expression for the *a*ssembly *l*ine *a*ggregate.

value
717. g_uise(*ap*) ≡ (*ap*κ,g_uise(*ala*))
717. **where:** *ap*κ=core_AP(*ap*) ∧ *ala*=obs_ALA(*ap*)

[46] A difference between Fig. 8.10 and Figs. 8.8–8.9 is that in Fig. 8.10 we have "moved" the left **MAL** taxonomy triangle a level down, to "level"with the right **SAL** triangles.

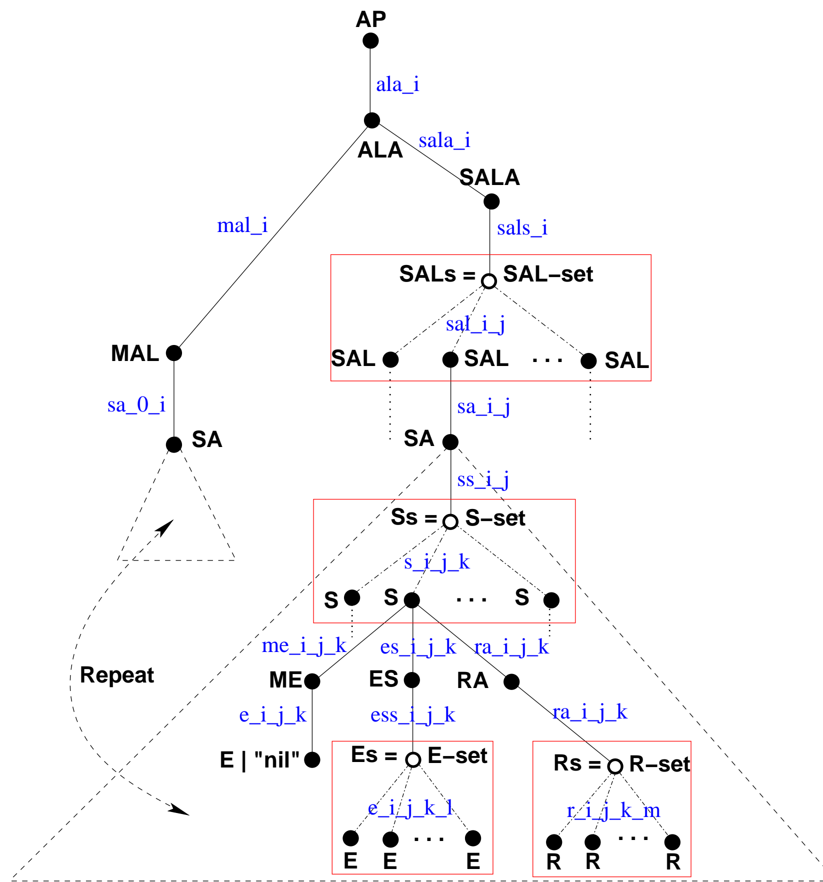


Fig. 8.10 Taxonomy with Unique Identifier Labels

[688 on page 147] **ALA, Assembly Line Aggregates**

718. The *unique identifier state expression* for the *a*ssembly *l*ine *a*ggregate, *ala*, is the pair of the assembly line aggregate core, *alaκ*, and the singleton map from the unique identifier of the *assembly line aggregate* to that aggregate – expressed as a core-part annotated map.

value
718. g_uise(*ala*) ≡
718.′ (*alaκ*,
718.′ [uid_ALA(*ala*)↦
718.′ [uid_MAL(*mal*)↦g_uise(*mal*),
718.′ uid_SALA(*sala*)↦g_uise(*sala*)]])

The one-liner, Item 718′, just above, is too "complex", better, we think, is the 4 liner just below, i.e., Items 718″–718⁗.

value
718. g_uise(*ala*) ≡
718.″ (*alaκ*,
718.‴ [uid_ALA(*ala*)
718.⁗ ↦ [uid_MAL(*mal*) ↦ g_uise(*mal*),

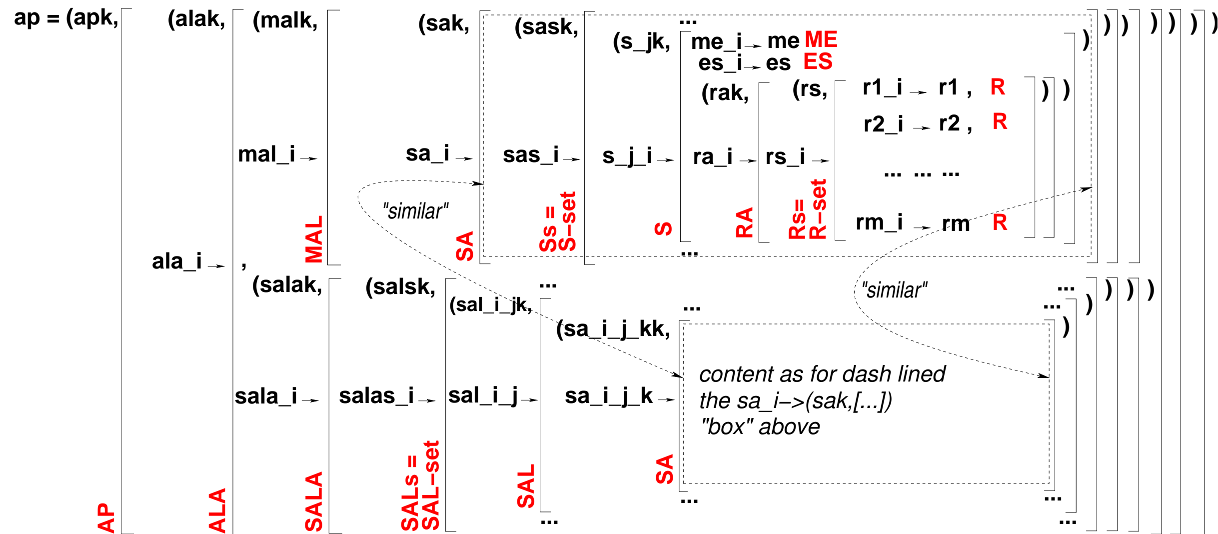


Fig. 8.11 Unique Identifier State Expression

718.′′′′′ uid_SALA($sala$) ↦ g_uise($sala$)]])
718. **where:** $ala\kappa$=core_AP(ala) ∧ mal = obs_MAL(ala) ∧ $sala$ = obs_SALA(ala)

[689 on page 147] **MAL, Main Assembly Lines**

719. The *unique identifier state expression* for the *m*ain *a*ssembly *l*ine, *mal*, is the pair of the main assembly line core, $mal\kappa$, and the map from the unique identifier of its station assembly, *sa*, and the unique identifier state expression for the station assembly.

value
719. g_uise(mal) ≡
719. ($mal\kappa$,
719. [uid_MAL(mal) ↦ [uid_SA(sa) ↦ g_uise(sa)]])
719. **where:** $mal\kappa$=core_MAL(mal) ∧ sa = obs_SA(mal)

[690 on page 147] **SALA, Supply Assembly Line Aggregates**

720. The *unique identifier state expression* for the *s*upply *a*ssembly *l*ine *a*ggregate, *sala*, is the pair of the supply assembly line aggregate core, $sala\kappa$, and the singleton map from the unique identifier of the *set of assembly lines* to that set – expressed as a core-part annotated map.

value
720. g_uise($sala$) ≡
720. ($sala\kappa$,
720. [uid_SALA($sala$) ↦ [uid_SALs($sals$) ↦ g_uise($sals$)]])
720. **where:** $sals$ = obs_SALs($sala$) ∧

[691 on page 147] **SALs=SAL-set, Supply Assembly Line Sets**

721. The *unique identifier state expression* for the *s*upply *a*ssembly *l*ine *s*et, *sals*, is the pair of the supply assembly line set core, *sals*κ, and the map from the unique identifier of each of the *supply assembly lines* to that set – expressed as a core-part annotated map.

value
721. g_uise(*sals*) ≡
721. (*sals*κ,
721. [uid_SAL(sal) ↦ g_uise(sal) | sal:SAL • sal ∈ *sals*])
721. **where:**

[692 on page 147] SAL, Supply Assembly Lines

722. The *unique identifier state expression* for the *s*upply *a*ssembly *l*ine, *sal*, is the pair of the main assembly line core, *sal*κ, and the singleton map from the unique identifier of its station assembly, *sa*, and the unique identifier state expression for that station assembly.

value
722. g_uise(*sal*) ≡
722. (*sal*κ ,
722. [uid_SAL(*sal*) ↦ [uid_SA(*sa*) ↦ g_uise(*sa*)]])
722. **where:** *sal*κ=core_SAL(*sal*) ∧ *sa* = obs_SA(*sal*)

[693 on page 147] SA, Station Aggregates

723. The *unique identifier state expression* for the *s*tation *a*ggregate, *sa*, is the pair of the station aggregate core, *sal*κ, and the map from the unique identifier of each of the *stations* to that set – expressed as a core-part annotated map.

value
723. g_uise(*sa*) ≡
723. (*sa*κ ,
723. [uid_SA(*sa*) ↦ g_uise(*ss*)])
723. **where:** *sa*κ = core_SA(*sa*) ∧ *ss* = obs_Ss(*sa*)

[694 on page 147] Ss=S-set, Station Set

724. The *unique identifier state expression* for a set of *s*tations, *ss*, is the pair of the station set core, *ss*κ, and the singleton map from the unique identifier of each of the *stations* to that set – expressed as a core-part annotated map.

value
724. g_uise(*ss*) ≡
724. (*ss*κ ,
724. [uid_Ss(*s*) ↦ g_uise(*s*) | *s*:S • *s* ∈ *ss*])
724. **where:**

[695 on page 147] S, Stations

725. The *unique identifier state expression* for *s*tations, *s*, is the pair of the station core, *s*κ, and the map from

- the unique identifier of that stations' *main element* to that main element, considered an atomic,

- the unique identifier of that stations' *element supply* to that element supply, here considered an "atomic" (!), and
- the unique identifier of that stations' *robot aggregate* to the unique identifier state expression for that robot aggregate.

value
725. g_uise(s) ≡
725. ($s\kappa$,
725. [uid_ME(me) ↦ me ,
725. uid_ES(es) ↦ es ,
725. uid_RA(ra) ↦ g_uise(ra)])
725. **where:** $s\kappa$ = core_S(s) ∧ me = obs_ME(s) ∧ es = obs_ES(s) ∧ ra = obs_RA(s)

[696 on page 147] ME, Main Elements

Item 725 above expresses that g_uise(me) = me.

[697 on page 147] RA, Robot Aggregates

726. The *unique identifier state expression* for robot aggregates, ra, is a pair of the robot aggregate core, $ra\kappa$, and the singleton map from unique identifier of the robot aggregate to the unique identifier state expression for the set of robots, rs, of that aggregate.

value
726. g_uise(ra) ≡
726. ($ra\kappa$,
726. [uid_Rs(rs) ↦ g_uise(rs)])
726. **where:** rs = obs_Rs(ra)

[698 on page 147] Rs=R-set, Robot Sets

727. The *unique identifier state expression* for robot sets, rs, is a pair of the robot set core and the map from the unique identifiers of the robots of the set to these robots.

value
727. g_uise(rs) ≡
727. ($rs\kappa$,
727. [uid_R(r) ↦ r | r:R • r ∈ rs])

[699 on page 147] R, Robots

Item 727 expresses that g_uise(r) = r.

[700 on page 148] ES, Element Supplies

Item 725 on the facing page above expresses that g_uise(es) = es.

[701 on page 148] Es=E-set, Element Supply Sets

Item 725 on the facing page hence expresses that g_uise(ess) = ess.

[702 on page 148] E, Elements

Item 725 on the facing page hence expresses that g_uise(e) = e.

8.2.2.2.2 Mereology

Observation of endurant parts does not itself leave any trace as to their taxonomy, nor does the identification of observed parts.

Mereology is what brings forth the taxonomy structures that is rendered, one way or another, in all the figures shown so far!

We shall show that many of the concerns of [70] have their "root" in mereology-properties of the domain; and we shall show that the topological aspects of the mereology "supports" `Microsoft`'s *Automated Graph Layout Tool* [125].

We express the mereology properties as relations between the mereology of the endurant being inquired, some or all elements of the mereology of the "ancestor" endurant, and some or all elements of the mereology of the "descendant" endurant(s).

Common to all mereo_P observers we "retrieve" the "predecessor" part, from the overall endurant state, and observe its mereology, while also "retrieving" the "descendant" parts, also from the overall endurant state, given their identifiers from the mereology of the part under observation, and then correlate them.

We then end up with a a set of mereology types, a set of corresponding mereology observer signatures [not definitions], and a set of corresponding axioms. For any given domain the mereology expresses some property that holds and that property transpires as the fix-point solution to the mutually [but not recursively] – sort-of simultaneous[ly] – expressed axioms [in the form of equations].

The overall property of the mereologies presented here is to secure that no two parts have identical mereologies.

That should be a provable property of what is presented below.

- The following numbered paragraphs start with the ι**tem number** of the endurant, whose **name** is given next. The item numbers are formally defined on page 148.

8.2.2.2.2.1 ι 687. AP: Assembly Plant:

728. The mereology of an assembly plant is
   - the unique identifier of its assembly line aggregate – such that
     a. the successor part's mereology identifies the assembly plant.

```
type
728. AP_Mer = ALAI
value
728. mereo_AP: AP → AP_Mer
axiom
728. let alai = mereo_AP(ap) in
728a.let (api,_) = mereo_ALA(retr_ALA(alai)) in retr_AP(api) = ap end end
```

8.2.2.2.2.2 ι 688. ALA: Assembly Line Aggregate:

729. The mereology of an assembly line aggregate is a pair
   - of the unique identifier of the main assembly line
   - and the unique identifier of the supply assembly line aggregate – such that
     a. the [assembly plant's, i.e., the] predecessor's successor is that assembly line aggregate and
     b. the two successors' ancestor are likewise.

```
type
729.  ALA_Mer = API × (MALI×SALAI)
value
729.  mereo_ALA: ALA → ALA_Mer
axiom
729.  let (api,(mali,salai)) = mereo_ALA(ala) in
729a.  let alai = mereo_AP(retr_AP(api)),
729b.      (alai′,_) = mereo_MAL(retr_MAL(mali)),
729b.      (alai″,_) = mereo_SALA(retr_SALA(salai)) in
729a.  alai = uid_ALA(ala) ∧
729b.  alai = alai′ = alai″ end end
```

8.2.2.2.2.3 ι 689. MAL: Main Assembly Line:

730. The mereology of a main assembly line aggregate is
   - the pair of the unique identifier of an assembly line aggregate and
   - the unique identifier of a station aggregate – such that
     a. the main assembly line's unique identifier is the same as the [assembly line aggregate] ancestor's successor and
     b. [station aggregate] successor's ancestor.

```
type
730.   MAL_Mer = ALAI × SAI
value
730.   mereo_MAL: MAL → MAL_Mer
axiom
730.   let (alai,sai) = mereo_MAL(mal), mali = uid_MAL(mal) in
730a.  let (_,(mali′,_)) = mereo_ALA(retr_ALA(alai′)),
730b.      (mali″,_) = mereo_SA(retr_SA(sai)) in
730a.  mali = mali′ ∧
730b.  mali = mali″ end end
```

8.2.2.2.2.4 ι 690. SALA: Supply Assembly Line Aggregate:

731. The mereology of a supply assembly line aggregate is
   - the unique identifier of an assembly line aggregate and
   - a pair of the unique identifier of a supply assembly line set – such that
     a. the [assembly line aggregate] predecessor's successor and
     b. the [supply line set] successor's predecessor

     supply line aggregate identifiers are the same.

```
type
731.   SALA_Mer = ALAI × SALsI
value
731.   mereo_SALA: SALA → SALA_Mer
axiom
731.   let (alai,salsi) = mereo_SALA(sala), salai=uid_SALA(sala) in
731.   let (_,(_,salai′)) = mereo_ALA(retr_ALA(alai)),
```

```
731.      (salai″,_) = mereo_SALs(retr_SALs(salsi)) in
731a.  salai = salai′ ∧
731b.  salai = salai′ end end
```

8.2.2.2.2.5 ι 691. SALs=SAL-set: Simple Assembly Line Set:

732. The mereology of a set of simple assembly lines is a pair of
    - the unique identifier of a supply assembly line aggregate and
    - a set of the unique identifiers of station aggregates – such that
      a. the [supply line aggregate] predecessor's successor and
      b. each individual simple assembly line's predecessor

      supply line set identifiers are the same.

```
type
732.  SALs_Mer = SALAI × SAI-set
value
732.  mereo_SALs: SALs → SALAI × SAI-set
axiom
732.  let (salai,sais) = mereo_SALs(sals), salsi = uid_SALs(sals) in
732.  let (_,salsi′) = mereo_SALA(retr_SALA(salai)) in
732a.  salsi = salsi′ ∧
732b.  ∀ sai:SAI•sai ∈ sais ⇒ let (sals″,_) = mereo_SA(retr_SA(sai)) in salsi=sals″ end
732.   end end
```

8.2.2.2.2.6 ι 692. SAL: Simple Assembly Lines:

733. The mereology of a simple assembly line is a pair of
    - the unique identifier of a [predecessor] supply assembly line set and
    - the unique identifier of a [successor] station assembly – such that
      a. the [supply assembly line set] predecessor's and
      b. the [station assembly] successor's

      simple assembly line identifiers are the same and that of the simple assembly line being observed.

```
type
733.  SAL_Mer= SALsI × SAI
value
733.  mereo_SAL: SAL → SAL_Mer
axiom
733.  let (salsi,sai) = mereo_SAL(sal), sali = uid_SAL(sal) in
733.  let (_,sali′) = mereo_SALs(retr_SALs(salsi)), (sali′i,_) = mereo_SA(retr_SA(sai)) in
733a. sali = sali′ ∧
733b. sali = sali″
733.  end end
```

8.2.2.2.2.7 ι 693. SA: Station Aggregate:

734. The mereology of a station aggregate is a pair of
   - the unique identifier of the [simple assembly line] predecessor and
   - the unique identifier of the [station set] successor – such that
     a. their station aggregate (successor), respectively (predecessor) station aggregate identifiers are the same as that of the station aggregate being observed.

```
type
734. SA_Mer = SALI × SsI
value
734. mereo_SA: SA → SA_Mer
axiom
734. let (sali,ssi) = mereo_SA(sa), sai = uid_SA(sa) in
734. let (_,sai′) = mereo_SAL(retr_SAL(sali)), (sai″,_) = mereo_Ss(retr_Ss(ssi)) in
734a. sai = sai′ = sai″ end end
```

8.2.2.2.2.8 ι 694. Ss = S-set: Station Sets:

735. The mereology of a station set is a pair of
   - the unique identifier of a [predecessor] station aggregate and
   - a set of unique identifiers of [successor] stations – such that
     a. that station aggregate's successor and
     b. that each successor station's predecessor

   unique identifiers are the same as that of the observed station set.

```
type
735. Ss_Mer = SAI × SI-set
value
735. mereo_Ss: Ss → Ss_Mer
axiom
735. let (sai,sis) = mereo_Ss(ss), ssi = uid_Ss(ss) in
735a. let (_,ssi′) = mereo_(retr_SA(sai)) in ssi = ssi′ end
735b. ∀ si:SI • si ∈ ssi • let (ssi″,_) = mereo_S(retr_S(si)) in ssi = ssi″ end end
```

8.2.2.2.2.9 ι 695. S: Station:

For all but stations the mereologies of solid endurants have modeled the part-hood relation *"part of"* (in the sense of *"sub-part of"*). All taxonomy figures[47] show this "sub-part" relation by means of the *lines* connection the •s. Figure 8.5 on page 146 show two additional [topological] part-hood relations: *"adjacent to"* and *"incident upon"*. Two stations of a simple assembly line may be adjacent to one another. The last station of a supply assembly line is incident upon a station of a main assembly line. The first station of any assembly line has no predecessor. The last station of a main assembly line has no successor.

736. Thus the mereology of a station, *s* identified by *si*, is a pair of,

[47] Figs. 8.6 on page 147, 8.7 on page 148, 8.8 on page 150, 8.9 on page 153 and 8.10 on page 158

a. *first* a pair, modeling *"part of"*:
  i. the unique identifier of a station set, *ssi*, the predecessor of *s*,
  ii. the unique identifiers of a triplet [(*mei*,*esi*,*rai*)] of successors of *s*:
    1. a main element *mei*,
    2. an element supply, *esi*, and
    3. a robot aggregate *rai*,
  and
b. *then* a pair, (*nsi*,*psi*), modeling, *nsi "[next] adjacent to"*, and, *psi "[previous] incident upon"* such that,
  i. for the first of the pair, i.e., *nsi*, is
    1. either "nil" for the "last" station, the outlet, of a main line,
    2. or is the next station of a main or supply line,
    3. or, for *s* being the "downstream last" of a supply line station, identifies a mainline station.
  ii. for the second of the pair, *psi* is [again] a pair: (*plsi*,*lslsi*)), where
    1. *plsi* is the *s*tation identifier of a station of the line to which *s* belongs, where
      - *plsi* is "nil", if *s* is the "first, upstream", station of its line, or
      - *plsi* properly identifies an "upstream" immediately previous station,
    and where *lslsi* is
    2. either "nil", ie., station *s* is not one incident upon by a supply assembly line,
    3. or is the proper identifier of a supply assembly line's "downstream, last" station such that
      - no two main line stations have the same supply assembly line incident upon them, and
      - where the number of supply assembly lines exactly equal the number of main line stations that have supply assembly lined incident upon them.

All of the above must satisfy the following invariants:

c. the unique identifier, *si*, of *s*, is in the the set of unique station identifiers of the predecessor,

and such that the unique identifiers of

d. the main element's,
e. the element supply's, and
f. the robot aggregate's

predecessors are all the same as that of the station under observation – and such that

g. the stations, *ss*, of the ancestor station set do indeed form a linear sequence;
h. *si*′ and *si*′′ [in (*si*′,*si*′′)] are indeed station identifiers of that sequence – or one is that of the next-but-last station of a supply assembly line and the other is that of a station of a main assembly line;
i. *si*′ [in (`"nil"`,*si*′)] is indeed a station identifier of that sequence; and
j. *si*′ [in (*si*′,`"nil"`)] is indeed a station identifier of that sequence.

We model the notion of linear sequences [here of stations].

k. Let ls:LS=S* stand for a linear sequence of two or more stations S.
l. Let ss stand for a set of two or more stations, i.e., ss∈Ss=S-**set**.
m. Then let linear_Ss be the function which "converts" ss to ls.[48]

[48] It is not a matter of whether or not an ss∈Ss=S-**set** may form a linear sequence. They simply do ! An assembly plant's assembly lines simply are linear ! Constellations of stations not forming linear sequences do not contribute to a proper assembly plant !

```
type
736.  S_Mer = (SsI × (MEI × ESI × RAI)) × ((opt_SI×opt_SI)×opt_SI)
736.  opt_SI = ({|″nil″|} | SI)

value
736.  S_Mer: S → S_Mer
axiom
736.  let ((ssi,(mei,esi,rai)),((si_b,si_a),si_sl)) = S_Mer(s), si = uid_S(s) in
736.  let (_,sis) = mereo_Ss(retr_Ss(ssi)), (si′,_) = mereo_ME(retr_ME(mei)),
736.      (si″,_) = mereo_ES(retr_ES(esi)), (si‴,_) = mereo_RA(retr_RA(rai)) in
736(b)ii.  si ∈ sis ∧
736d.  si = si′ ∧
736e.  si = si″ ∧
736f.  si = si‴  end end

736(a)i.   ∀ s:S • let (_,(b_si,a_si)) = S_Mer(s) in
736(a)i.      b_si≠″nil″∧a_si≠″nil″ ∨
736(a)i.      b_si=″nil″∧a_si≠″nil″ ∨
736(a)i.      b_si≠″nil″∧a_si=″nil″ end
736(a)ii.
736(a)ii1.
736(a)ii2.
736(a)ii3.
736(b)i.
736(b)i1.
736(b)i2.
736(b)i3.
736(b)ii.
736(b)ii1.
736(b)ii2.

type
736k.  LS = S*

axiom
736k.  ∀ ls:LS • len ls > 1

736k.  is_linear: LS → Bool
736k.  is_linear(ls) ≡
736k.     ∧ let (_,(null,_)) = mereo_S(ls[1]) in null = ″nil″ end
736k.     ∧ ∀ i:Nat • {i,i+1}⊆inds ls ⇒
736k.          let (_,(_,si_a)) = S_Mer(ls[i]),
736k.              (_,(si_b,_)) = S_Mer(ls[i+1]) in si_a = uid_S(ls[i]) = si_b end
736k.     ∧ let (_,(,s_uid)) = mereo_S(ls[len ls]) in
736k.     ∧ (s_uid = ″nil″ ∨ is_MAL_S(retr_S(s_uid))) end

value
736k.  is_MAL_S: S → Bool
736k.  is_MAL_S(s) ≡
736k.     let ((ssi,_),_) = mereo_S(s) in
736k.     let (sai,_)   = mereo_Ss(retr_Ss(ssi)) in
736k.     let ali = mereo_SA(retr_SA(ssi)) in
```

```
736k.     is_MALI(ali) end end end

736l.  ss:Ss, axiom card ss > 1
736g.  linear_Ss: S-set → S*
736g.  linear_Ss(ss) ≡
736g.     let ls:LS • elems ls = ss ∧
736g.        ∀ i:Nat • {i,i+1}⊆inds ls ⇒
736g.           let (_,(_,a_si)) = mereo_S(ls[i])
736g.           let (_,(b_si,_)) = mereo_S(ls[i+1]) in
736g.           a_si = b_si end
736g.     ls end end
```

8.2.2.2.2.10 ι696. ME: Main Elements:

737. The mereology of a main element is a singleton
   - of the unique identifier of its predecessor station – such that
     a. that station identifies that main element.

```
type
737.  ME_Mer = SI
value
737.  mereo_ME: ME → ME_Mer
axiom
737a.  let si = mereo_ME(me), mei = uid_ME(me) in
737a.  let (_,(mei′,_,_)) = mereo_S(retr_S(si)) in
737a.  mei = mei′ end end
```

8.2.2.2.2.11 ι697. RA: Robot Aggregate:

738. The mereology of a robot aggregate is
   - a pair of the unique identifier of a station (the predecessor) and
   - a unique identifier of a robot set (the successors) – such that
     a. the station predecessor identifies the robot aggregate, and
     b. the identified robot set identifies the same robot aggregate.

```
type
738.  RA_Mer = SI × RsI
value
738.  mereo_RA: RA → RA_Mer
axiom
738.  let (si,rsi) = mereo_RA(ra), rai = uid_RA(ra) in
738.  let (_,(rai′,_,_),_) = mereo_S(retr_S(si)),
738.     (rai″,_) = mereo_Rs(retr_Rs(rsi)) in
738a.  rai = rai′ ∧
738b.  rai = rai″ end end
```

8.2.2.2.2.12 ι 698. Rs=R-set: Robot Set:

739. The mereology of a robot set is a pair of
   - the unique identifier of a robot aggregate and
   - a set of unique identifiers of robots – such that
     a. the identified robot aggregate identifies the robot set, and
     b. all the identified robots also identifies that robot set.

```
type
739.  Rs_Mer = RAI × RI-set
value
739.  mereo_Rs: Rs → Rs_Mer
axiom
739.  let (rai,ris) = mereo_Rs(rs), rsi = uid_Rs(rs) in
739a. let (_,rsi′) = mereo_RA(retr_RA(rai)) in rsi = rsi′ end
739a. ∀ ri:RI • ri ∈ ris ⇒ let rsi″ = mereo_R(retr_R(ri)) in rsi = rsi″ end end
```

8.2.2.2.2.13 ι 699. R: Robot:

740. The mereology of a robot is
   - a singleton of the unique identifier of a robot set – such that.
     a. that robot set identifies the robot.

```
type
740.  R_Mer = RsI
value
740.  mereo_Rs: Rs → Rs_Mer
axiom
740.  let rsi = mereo_R(r), ri = uid_R(r) in
740a. let (_,ris) = mereo_Rs(retr_Rs(rsi)) in ri ∈ ris end end
```

8.2.2.2.2.14 ι 700. ES: Element Supply:

741. The mereology of an element supply is a pair of
   - the unique identifier of a station and
   - the unique identifier of an element supply set – such that
     a. the identified station identifies the element supply, and
     b. the identified element supply set identifies the element supply.

```
type
741.  ES_Mer = SI × EsI
value
741.  mereo_ES: ES → ES_Mer
axiom
741.  let (si,esi) = mereo_ES(es), esi = uid_ES(es) in
741.  let ((_,(_,esi′,_)),_) = mereo_ES(retr_ES(esi)), esi″ = uid_ES(es) in
741a. esi = esi′ ∧
741b. esi = esi″ end end
```

8.2.2.2.2.15 ι 701. Es=E-set: Element Supply Set:

742. The mereology of an element supply set is a pair of
   - the unique identifier of an element supply aggregate and
   - a set of unique identifiers of elements – such that
     a. the identified element supply aggregate identifies the element supply set and
     b. the all the element identifiers identifies the element supply set.

```
type
742.  Es_Mer = ESI × EI-set
value
742.  mereo_Es: Es → Es_Mer
axiom
742.  let (esi,eis) = mereo_Es(es), es_i = uid_Es(es) in
742a.  let (_,es_j) = mereo_Es(retr_Es(esi)) in es_i = es_j end ∧
742b.  ∀ ei:EI•ei ∈ eis ⇒ let es_k = mereo_E(retr_E(ei)) in es_i = es_k end end
```

8.2.2.2.2.16 ι 702. E: Elements:

743. The mereology of an element is
   - a singleton of the unique identifier of an element supply set – such that
     a. this identifier identifies the element's supply set.

```
type
743.  E_Mer = EsI
value
743.  mereo_E: E → ES_Mer
axiom
743.  let esi = mereo_E(e), eis = uid_E(e) in
743a. let (_,eis′) = mereo_Es(retr_Es(esi)) in eis = eis′ end end
```

Comments on the Mereology Presentation

It is all very tedious: Mereology after mereology – of each and all of the solid endurants. Their narratives and formalisations, expression-wise, all follow the same "pattern", and the "contents" follow, almost mechanical, from the taxonomy figures[49] and, wrt. stations, from Figs. 8.5 on page 146 and 8.7 on page 148.

I have not followed a strict narrative for the 16 mereology presentations, and even the formulas differ slightly. Once I get time I will probably device a LaTeX macro so as to generate consistent narratives.

Distances of Stations from Outlet

Paths:

We shall examine an ordering, ⪯, on stations. To this end we introduce the notion of paths. A path is a sequence of station identifiers such that

744. A path is a non-empty sequence of station identifiers such that
745. the first identifier is that of the first station of a main assembly line,

[49] Figs. 8.6 on page 147, 8.7 on page 148, 8.8 on page 150, 8.9 on page 153 and 8.10 on page 158

746. and such that
747. adjacent identifiers of a path are those of neighbouring stations,
   a. whether of the same assembly line,
   b. or of
      i. the first station of a supply assembly line
      ii. and of the station of the main assembly line onto which supply assembly line is joined.

```
type
744.     Path = SI*
axiom [ paths of an assembly plant ]
744.     ∀ p:Path • len p > 0 ∧
745.        let ⟨si⟩⌢p′ = p, ss:Ss = obs_Ss(obs_SA(mal)) in
745.        let s:S • s ∈ ss ∧ let (,(,nil))=mereo_S(s) in nil=″nil″∧si=uid_S(s) end end
746.        ∧
747.        ∀ i:Nat • {i,i+1}⊆inds p ⇒
747.        let (_,(pi,_)) = mereo_S(retr_S(i)), (_,(_,si)) = mereo_S(retr_S(i+1)) in
747a.        ( pi=p(i+1) ∧ si=p(i) )
746.        ∨
747(b)i.     (
747(b)i.     ∧
747(b)ii.     ... )
744.        end end
```

Set of all Paths:

From an assembly plant we can then generate the set of all paths.

748.
749.
750.
751.
752.
753.

```
748.
749.
750.
751.
752.
753.
```

Distance:

Given any station of an assembly plant we can then calculate its distance from the main line outlet.

754.
755.
756.
757.
758.
759.

```
754.
755.
756.
```

757.
758.
759.

The ⪯ Relation:

760. Given any two stations of an assembly plant we can then express which of the two "precedes" the other wrt. distance from the main line outlet.

760.

8.2.2.2.3 Attributes

General

The real action of an assembly line is focused in the stations. The robots apply elements to the contents of the main element. So, in treating now the attributes of assembly lines, we shall in this early version of this project report, focus on the rôle of elements.

Elements and Parts

The term *'part'* is a main term of the *domain analysis & description* method [55] that we use. It is not to be confused with the same term, i.e., `part`, used, normally, in connection with machine parts, part assembly, etc. The term Main Element is used to name the solid endurant of a station, namely that which, so-to-speak, "holds" the main object of concern: the thing being assembled. We shall think of main elements to be some form of manifest "carrier". We shall then ascribe such main elements an attribute, and here we shall switch to the use of the term part, namely a main part. So element supplies, which we hitherto explained as containing elements for use in the assembly of main parts, could, as well be called parts. Whereas solid endurants such as stations and robot will, later, be "morphed", i.e., transcendentally deduced, into behaviours, we shall not morph main parts into behaviours – not as long, at least, as they stay within the assembly lines. Once a main part has left a main assembly line, "from" its last station, then it may, in some other domain model, attain "life" in the form of a behaviour.[50]

Relationship to [70]

We shall show that many of the concerns of [70] have their "root" in attribute-properties of the domain.

Specifics

8.2.2.2.3.1 ι 687. AP: Assembly Plant:

We omit treatment of assembly plant attributes.

8.2.2.2.3.2 ι 688. ALA: Assembly Line Aggregate:

We presently omit treatment of assembly line aggregate attributes.

[50] The main parts leaving the main assembly line of an automobile factory, in an orderly fashion, may then, as an automobile, be able to leave by its own means !

8.2.2.2.3.3 ι 689. MAL: Main Assembly Line:

With every supply assembly line we associate the attributes

761. that it is a main assembly line, and that
762. the main elements of its stations contain parts of a specific (to be finalised) element type.

```
type
761.  AL_Typ = ″Main″
762.  ME_Typ = E_Typ
value
761.  attr_AL_Typ: MAL → AL_Typ
762.  attr_ME_Typ: MAL → ME_Typ
```

8.2.2.2.3.4 ι 690. SALA: Supply Assembly Line Aggregate:

We presently omit treatment of supply assembly line aggregate attributes.

8.2.2.2.3.5 ι 691. SALs: Supply Assembly Line Set:

We presently omit treatment of supply assembly line set attributes.

8.2.2.2.3.6 ι 692. SAL: Supply Assembly Lines:

With every supply assembly line we associate the attributes that

763. it is a supply assembly line[51],
764. the main elements of its stations contain parts of a specific (to be finalised) element type,
765. it "feeds" into an identified main line station, and that
766. it is either "feeding" into the main line at the ″left″ or at the ″right″ !

```
type
763.  AL_Typ = ″Supply″
764.  ME_Typ = E_Typ
766.  Feed == ″Left″ | ″Right″
value
763.  attr_AL_Typ: SAL → AL_Typ
764.  attr_ME_Typ: SAL → ME_Typ
765.  attr_MAL_S: SAL → SI
766.  attr_Feed: SAL → Feed
```

8.2.2.2.3.7 ι 693. SA: Station Aggregate:

We presently omit treatment of station aggregate attributes.

[51] – where main assembly lines are ″Main″ !

8.2.2.2.3.8 ι 694. Ss=S-set: Station Set:

We presently omit treatment of station set attributes.

8.2.2.2.3.9 ι 695. S: Station:

We first discuss some of the rôles played by the robots, main element part and element supply of a station.

- Robots of a station are **cap**able, at any one time of performing one of a set of one or more operations. Robots and their operations have names, RNm respectively OpNm.
  So we can attribute a station with

  type
  767. CAP = RNm $\overrightarrow{m}$ OpNm-**set**

  We allow two or more robots of any one station to "feature" the same, named operation!
- Operations, OP, are functions from a main element part and
  - either a single part provided by a supply line, if the operation is performed at a main line station, and
    - either
    - or a set of elements provided by that stations element supply
  - to an updated main element part.

  type
  768. OP = ME_Part × (ME_Part|E-**set**) → ME_Part

- So a Station can be given the following attribute:

  type
  768. OPS = OpNm $\overrightarrow{m}$ OP

  Two or more differently named operations may, in fact, designate identical operations!
- Operations have types:

  type
  OpTyp = ME_Part_Typ × (ME_Part_Typ | E_Typ*) × ME_Part_Typ

  So we assume that there are (meta-) functions like:

  value
  type_of: E → E−Typ, ME_Part → ME_Part_Typ, is_of_type: E×E_Typ → **Bool**, etc.

We now "return" to our attribute "ascription story" proper!

With a station we can associate the following attributes:

767. The named operations that can be performed by it robots;
768. the catalogue of these operations;
769. the area of the assembly floor covered by the station;
770. the identified *zones* (sub-areas) into which the station is divided;

type
767. RNm, OpNm
767. CAP = RNm $\overrightarrow{m}$ OpNm-**set**
768. OPS = OpNm $\overrightarrow{m}$ OP
768. OP = ME_Part × (ME_Part|E-**set**) → ME_Part
769. Sta_Area = $\mathbb{AREA}$
770. Zones = ZId $\overrightarrow{m}$ Zone
770. Zone = Zone_Area
770. Zone_Area = $\mathbb{AREA}$
value
767. attr_CAP: S → CAP
768. attr_OPS: S → OPS
769. attr_StaArea: S → StaArea
770. attr_Zones: S → Zones
axiom
770. [∪ **of** zone areas ≡ station area]

8.2.2.2.3.10 ι696. ME: Main Element:

With main elements we associate the following programmable attribute:

771. the main part, mp:ME_Part and
772. the types of the main part before, during and after robot operations, i.e., as it enters the station, during its stay at the station, and as it leaves the station.

type
771. ME_Part
772. ME_Part_Types = E_Typ*
value
771. attr_Part: ME → ME_Part
772. attr_ME_Part_Types ME → ME_Part_Types

Caveat: The above type model is a bit simplified ! Shall/must be reviewed !

8.2.2.2.3.11 ι697. RA: Robot Aggregate:

Caveat: It seems that either stations or robot aggregates must have some form of awareness, expressed in the form of an attribute, of the **task**s to be collectively, co-operatively performed by the ensemble of robots. **I am currently contemplating such a model !**

8.2.2.2.3.12 ι698. Rs=R-set: Robot Set:

We presently omit treatment of robot set attributes.

8.2.2.2.3.13 ι699. R: Robot:

With a robot we can associate the following attributes:

773. the zone to which it is allocated;

774. the operations it can perform and their type;
775. where we leave unspecified these element (i.e., part) types.
776. ...

```
type
773.  R_Zone = Zone
774.  R_Ops = OpNm →m OpTyp
774.  OpTyp = ME_Part_Typ × (ME_Part_Typ|E_Typ*) × ME_Part_Typ
775.  ME_Part_Typ, E_Typ
776.  ...
value
773.  attr_RZone: R → RZone
774.  attr_ROps: R → ROps
775.  ...
776.  ...
```

8.2.2.2.3.14 ι 700. ES: Element Supply:

An element supply can be characterised by

777. a catalogue of element *"quantities on hand"* and their type.

```
type
777.  ES_QoH_Typ = E_Typ × Nat
value
777.  attr_ES_QoH_Typ: ES → ES_QoH_Typ
```

8.2.2.2.3.15 ι 701. Es=E-set: Element Supply Set:

We presently omit treatment of element set attributes.

8.2.2.2.3.16 ι 702. E: Elements:

An element (i.e., a part) can be characterised by

778. its type

```
type
778.  E_Typ
value
778.  attr_E_Typ: E → E_Typ
```

8.2.2.3 Comments wrt. [70]

We shall now relate the various segments of our model to [70].

TO BE WRITTEN

8.2.3 Perdurants

8.2.3.1 From Parts to Behaviours

We refer the reader to Figs. 8.5 on page 146 and 8.8 on page 150 – summarised in Fig. 8.12.

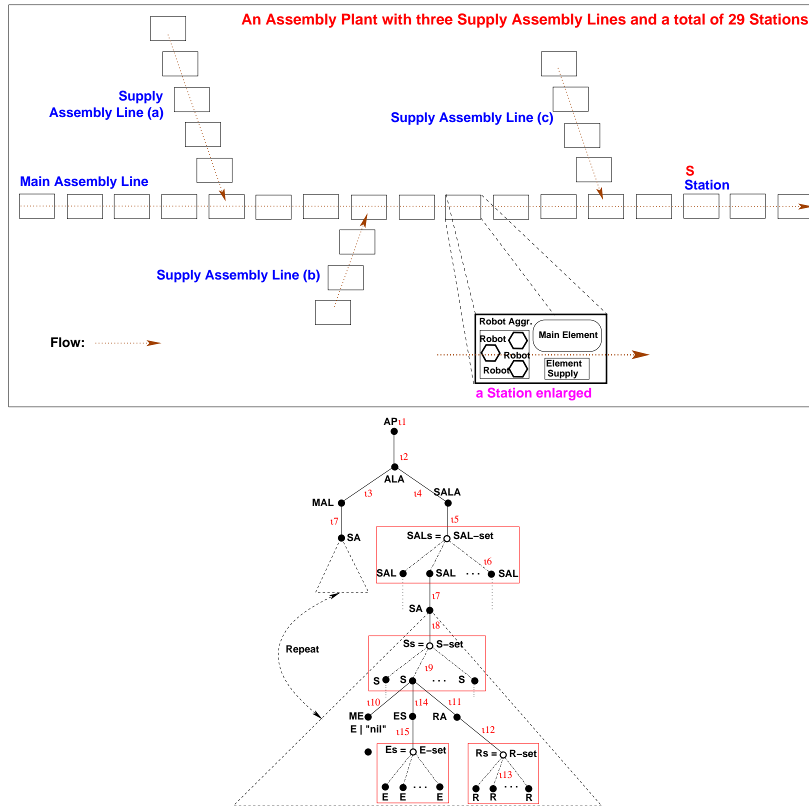


Fig. 8.12

- By transcendental deduction, see [55, Chapter 5], we "morph" core parts, p_κ, i.e., including atomic parts, into behaviours β_p.
- Behaviour β_{ap} coordinates behaviour β_{ala} with the rest of the manufacturing plant – remember: the assembly line complex is only one among several factory elements.
- Behaviour β_{ala} coordinates the main assembly line behaviours with that of the behaviour of the supply assembly lines aggregate.
- Behaviour β_{mal} coordinates the main assembly line's stations.
- Behaviour β_{sala} coordinates the total of all supply lines.
- Behaviours β_{sal} coordinates the specific supply assembly line's stations.
- Behaviour β_{sa} coordinates the interaction between the stations of an assembly line.
- Behaviour β_s coordinates the specific station's elements (main element, robots and element supply) as well as that station's interaction with neighbouring stations.
- Behaviour β_{me} participates in the main elements interaction with its station's robots.
- Behaviour β_{es} responds to its station's robots' requests for supply elements.
- Behaviour β_{ra} coordinates the specific station's robots.

- Behaviours β_r interacts with its station's main element, its other robots, and its element supply.
- Behaviours β_e – is presently left unspecified.

8.2.3.2 Channels

8.2.3.3 Actors

8.2.3.3.1 Actions and Events

8.2.3.3.2 Behaviours

8.2.3.4 System Initialisation

8.3 Discussion

We shall relate the model of Sect. **8.2** to [70]. To us [70] both describes and prescribes: describes some aspects of the problem domain and prescribes some requirements.

TO BE WRITTEN

8.4 Conclusion

We shall discuss whether the kind of work reported in [70] could be supported, made easier, made more complete, given that their domain is first properly described.

8.4.1 Models and Axioms

TO BE WRITTEN

8.4.2 Learning Forwards, Practicing In Reverse

The Danish philosopher Søren Kierkegaard (1813–1855) is quoted as saying

> *Life can only be understood backwards; but it must be lived forwards.*

Now, why do we bringing that quote here ? ! We do so for the following, slightly, if not radically less "lofty" reason: We learn forward, bit-by-bit, not seeing the overall picture before at the end. Then, when we shall practice what we have been taught, what we have learnt, we apply that knowledge, so-to-speak, backwards, knowing where what we shall end up with from the start of that "doing it".

When You study [55] You learn the subject forward. But having hopefully understood the domain modeling discipline, You You practice it "sort-of" in reverse.

My reason for bring the Søren Kierkegaard quote is to make You remember "that" !

8.4.3 Diagrammatic Reasoning

One, of many, observations of this report, are the examples of what I shall refer to as *diagrammatic reasoning*[52].

One way in which this is manifested, in this compendium is in Figs. 8.5 on page 146, 8.6 on page 147, 8.7 on page 148, 8.8 on page 150, 8.9 on page 153 and 8.10 on page 158. You may think that the number of these figures is a bit high. Very well, but they helped this "seasoned domain engineer" to come to grips with the seeming complexities of the domain being modeled. The internal relationships between these figures is obvious, *"when You look at them !"*, and their "external" relations to the *narration & formalisation* items should also be "obvious" !

8.4.4 The Management of Domain Modeling

A Domain Modeling Development Plan

We outline a plan for the commercial/professional development of a domain model for a "real" [say automobile] assembly plant:

- **Study:**
  - A domain is suggested.
  - One or two seasoned domain engineers cum scientists , *the initiation team*, make inquiries about the domain:
    - Visit one or more such domain sites.
    - Search the Internet for reliable accounts on the domain.
    - Read technical/scientific papers about the domain.
  - At some point the initiation team decides to do one or more experimental domain modeling efforts.
- **Experiment:**
  - They follow the dogma of [55] – "strictly".
  - (This report is an example of such an experimental research and engineering development.)
  - They may waver along different paths, maybe abandon/abort certain modeling directions, eventually reaching some, usually, incomplete domain analysis & description documentation.
  - They may decide to do another, and, perhaps, subsequently yet another experimental research and engineering development.
  - Eventually they **either abandon** the attempt to go after a fully complete, professional domain model, **or they conclude that a satisfactory, complete modeling project is professionally and commercial viable.**
- **Apply:**
  - The first step in a professional and commercial domain modeling project is that of creating a staff plan:
    - An outcome of a final domain modeling experiment is that the main taxonomy of the domain has been settled upon.

52

- *Gerard Allwein and Jon Barwise (ed.) (1996). Logical Reasoning with Diagrams. Oxford University Press.*
- `https://en.wikipedia.org/wiki/Diagrammatic_reasoning`.

  - For each of the main categories of endurants one or two domain engineers [cum scientists] are then to be allocated to the project.
  - A development graph[53] is developed.
  - A budget is established.
  - Negotiations with customer finally establish the financial foundation for the project[54].
- The commercial development project starts.
- First the endurant aspects are modeled – with
  - external qualities being first modeled [55, Chapter 4], then with
  - internal qualities:
    - unique identification [55, Sect. 5.2],
    - mereologies [55, Sect. 5.3], and
    - attributes [55, Sect. 5.4] – including notably *intentional pull* – which has not been illustrated in this report [55, Sect. 5.5].
- Then perdurants:
  - states [55, Sect. 7.2],
  - channels [55, Sect. 7.5],
  - actor, i.e., action, event and behaviour, signatures [55, Sect. 7.6],
  - their definitions [55, Sect. 7.7], and
  - system initialisation [55, Sect. 7.8].
- Etcetera !
- Each project member either "sticks" to the initially assigned endurant (hence perdurant) area throughout the project, or members have their subject areas "rotated".

Special circumstances may mandate variations to the above development plan.

For a reasonably "complete", i.e., covering essential aspects of, say an automobile manufacturing plant's assembly lines, it is roughly estimated that a group of well-educated domain engineers cum scientists would number 8–10, and that it would take 18–24 months to do the "Apply" phase of a domain modeling development project.

8.4.5 ... one more section ...

8.4.6 ... a last section (?) ...

8.4.7 Acknowledgments

TO BE WRITTEN

[53] For the notion of *Development Graphs* see [15, 16, 17].

[54] One cannot assume that the customer explicitly funds the Study and Experiment phases of the project.

Chapter 9
Document Systems [Summer 2017]

I had, over the years, since mid 1990s, reflected upon the idea of *"what is a document?"*. A most recent version, as I saw it in 2017, was "documented" in Chapter 7 [58]. But, preparing for my work, at TongJi University, Shanghai, September 2017, see Chapter 10, I reworked my earlier notes [58] into what is now this chapter.

Contents

9.1 Introduction . . . 182
9.2 A System for Managing, Archiving and Handling Documents . . . 182
9.3 Principal Endurants . . . 183
9.4 Unique Identifiers . . . 183
9.5 Documents: A First View . . . 184
9.5.1 Document Identifiers . . . 184
9.5.2 Document Descriptors . . . 184
9.5.3 Document Annotations . . . 185
9.5.4 Document Contents: Text/Graphics . . . 185
9.5.5 Document Histories . . . 185
9.5.6 A Summary of Document Attributes . . . 185
9.6 Behaviours: An Informal, First View . . . 186
9.7 Channels, A First View . . . 187
9.8 An Informal Graphical System Rendition . . . 188
9.9 Behaviour Signatures . . . 188
9.10 Time . . . 189
9.10.1 Time and Time Intervals: Types and Functions . . . 189
9.10.2 A Time Behaviour and a Time Channel . . . 190
9.10.3 An Informal RSL Construct . . . 190
9.11 Behaviour "States" . . . 190
9.12 Inter-Behaviour Messages . . . 191
9.12.1 Management Messages with Respect to the Archive . . . 191
9.12.2 Management Messages with Respect to Handlers . . . 192
9.12.3 Document Access Rights . . . 192
9.12.4 Archive Messages with Respect to Management . . . 193
9.12.5 Archive Message with Respect to Documents . . . 193
9.12.6 Handler Messages with Respect to Documents . . . 193
9.12.7 Handler Messages with Respect to Management . . . 194
9.12.8 A Summary of Behaviour Interactions . . . 194
9.13 A General Discussion of Handler and Document Interactions . . . 194
9.14 Channels: A Final View . . . 195
9.15 An Informal Summary of Behaviours . . . 195
9.15.1 The Create Behaviour: Left Fig. 9.3 on page 196 . . . 195
9.15.2 The Edit Behaviour: Right Fig. 9.3 on page 196 . . . 195
9.15.3 The Read Behaviour: Left Fig. 9.4 on page 196 . . . 196
9.15.4 The Copy Behaviour: Right Fig. 9.4 on page 196 . . . 196
9.15.5 The Grant Behaviour: Left Fig. 9.5 on page 197 . . . 197
9.15.6 The Shred Behaviour: Right Fig. 9.5 on page 197 . . . 197
9.16 The Behaviour Actions . . . 198

9.16.1 Management Behaviour 198
9.16.1.1 Management Create Behaviour: Left Fig. 9.3 on page 196 198
9.16.1.2 Management Copy Behaviour: Right Fig. 9.4 on page 196 199
9.16.1.3 Management Grant Behaviour: Left Fig. 9.5 on page 197 200
9.16.1.4 Management Shared Behaviour: Right Fig. 9.5 on page 197 201
9.16.2 Archive Behaviour 201
9.16.2.1 The Archive Create Behaviour: Left Fig. 9.3 on page 196 201
9.16.2.2 The Archive Copy Behaviour: Right Fig. 9.4 on page 196 202
9.16.2.3 The Archive Shred Behaviour: Right Fig. 9.5 on page 197 202
9.16.3 Handler Behaviours 203
9.16.3.1 The Handler Create Behaviour: Left Fig. 9.3 on page 196 203
9.16.3.2 The Handler Edit Behaviour: Right Fig. 9.3 on page 196 203
9.16.3.3 The Handler Read Behaviour: Left Fig. 9.4 on page 196 204
9.16.3.4 The Handler Copy Behaviour: Right Fig. 9.4 on page 196 204
9.16.3.5 The Handler Grant Behaviour: Left Fig. 9.5 on page 197 205
9.16.4 Document Behaviours 205
9.16.4.1 The Document Edit Behaviour: Right Fig. 9.3 on page 196 205
9.16.4.2 The Document Read Behaviour: Left Fig. 9.4 on page 196 206
9.16.4.3 The Document Shred Behaviour: Right Fig. 9.5 on page 197 206
9.16.5 Conclusion 207
9.17 Documents in Public Government 207
9.18 Documents in Urban Planning 207

We domain analyse and suggest a description of a domain of documents. We emphasize that the model is one of several possible. Common to these models is that we model "all" we can say about documents – irrespective of whether it can also be "implemented"! The model(s) are not requirements prescriptions – but we can develop such from our domain description.

You may find that the model is overly detailed with respect to a number of "operations" and properties of documents. We find that these operations must be part of the very basis of a document domain in order to cope with documents such as they occur in, for example, public government, see Appendix sect. **9.17**, or in urban planning, see Appendix Sect. **9.18**.

9.1 Introduction

We analyse a notion of documents. Documents such as they occur in daily life. What can we say about documents – regardless of whether we can actually provide compelling evidence for what we say! That is: we model documents, not as electronic entities — which they are becoming, more-and-more, but as if they were manifest entities. When we, for example, say that *"this document was recently edited by such-and-such and the changes of that editing with respect to the text before is such-and-such"*, then we can, of course, always claim so, even if it may be difficult or even impossible to verify the claim. It is a fact, although maybe not demonstrably so, that there was a version of any document before an edit of that document. It is a fact that some handler did the editing. It is a fact that the editing took place at (or in) exactly such-and-such a time (interval), etc. We model such facts.

This research note unravels its analysis &[55] description in stages.

9.2 A System for Managing, Archiving and Handling Documents

The title of this section: *A System for Managing, Archiving and Handling Documents* immediately reveals the major concepts: That we are dealing with a *system* that **manages**, **archives** and

[55] We use the logo gram & between two terms, A & B, when we mean to express one meaning.

handles documents. So what do we mean by **managing, archiving** and **handling** documents, and by **documents** ? We give an ultra short survey. The survey relies on your prior knowledge of what you think documents are ! **Management** decides[56] to direct **handler**s to work on **document**s. **Management** first directs the document archive to **create document**s. The document **archive creates document**s, as requested by **management**, and informs management of the **unique document identifiers** (by means of which handlers can handle these documents). **Management** then **grant**s its designated **handler**(s) **access rights** to **document**s, these access rights enable handlers to **edit, read** and **copy** documents. The **handler**s' **edit**ing and **read**ing of **document**s is accomplished by the **handler**s "working directly" with the **document**s (i.e., synchronising and communicating with **document behaviour**s). The **handler**s' **copy**ing of **document**s is accomplished by the **handler**s requesting **management**, in collaboration with the **archive** behaviour, to do so.

9.3 Principal Endurants

By an *endurant* we shall understand *"an entity that can be observed or conceived and described as a "complete thing" at no matter which given snapshot of time."* Were we to "freeze" time we would still be able to observe the entire endurant. This characterisation of what we mean by an 'endurant' is from [48, Manifest Domains: Analysis & Description]. We begin by identifying the principal endurants.

779. From document handling systems one can observe aggregates of handlers and documents. We shall refer to 'aggregates of handlers' by M, for management, and to 'aggregates of documents' by A, for archive.
780. From aggregates of handlers (i.e., M) we can observe sets of handlers (i.e., H).
781. From aggregates of documents (i.e., A) we can observe sets of documents (i.e., D).

```
type
779  S, M, A
value
779  obs_M: S → M
779  obs_A: S → A
type
780  H, Hs = H-set
781  D, Ds = D-set
value
780  obs_Hs: M → Hs
781  obs_Ds: A → Ds
```

9.4 Unique Identifiers

The notion of unique identifiers is treated, at length, in [48, Manifest Domains: Analysis & Description].

782. We associate unique identifiers with aggregate, handler and document endurants.
783. These can be observed from respective parts[57].

[56] How these decisions come about is not shown in this research note – as it has nothing to do with the essence of document handling, but, perhaps, with 'management'.

[57] [48, Manifest Domains: Analysis & Description] explains how 'parts' are the discrete endurants with which we associate the full complement of properties: unique identifiers, mereology and attributes.

type
782 MI[58], AI[59], HI, DI
value
783 uid_MI[60]: M → MI
783 uid_AI[61]: A → AI
783 uid_HI: H → HI
783 uid_DI: D → DI

As reasoned in [48, Manifest Domains: Analysis & Description], the unique identifiers of endurant parts are indeed unique: No two parts, whether composite, as are the aggregates, or atomic, as are handlers and documents, can have the same unique identifiers.

9.5 Documents: A First View

A document is a written, drawn, presented, or memorialized representation of thought. The word originates from the Latin documentum*, which denotes a "teaching" or "lesson".*[62] We shall, for this research note, take a document in its written and/or drawn form. In this section we shall survey the concept a documents.

9.5.1 Document Identifiers

Documents have *unique identifiers*. If two or more documents have the same document identifier then they are the same, one (and not two or more) document(s).

9.5.2 Document Descriptors

With documents we associate *document descriptors*. We do not here stipulate what document descriptors are other than saying that when a document is **create**d it is provided with a descriptor and this descriptor "remains" with the document and never changes value. In other words, it is a static attribute.[63] We do, however, include, in document descriptors, that the document they describe was initially based on a set of zero, one or more documents – identified by their unique identifiers.

---

[58] We shall not, in this research note, make use of the (one and only) management identifier.

[59] We shall not, in this research note, make use of the (one and only) archive identifier.

[60] Cf. Footnote 58: hence we shall not be using the uid_MI observer.

[61] Cf. Footnote 59: hence we shall not be using the uid_AI observer.

[62] From: https://en.wikipedia.org/wiki/Document

[63] You may think of a document descriptor as giving the document a title; perhaps one or more authors; perhaps a physical address (of, for example, these authors); an initial date; as expressing whether the document is a research, or a technical report, or other; who is issuing the document (a public institution, a private firm, an individual citizen, or other); etc.

9.5.3 Document Annotations

With documents we also associate *document annotations*. By a document annotation we mean a programmable attribute, that is, an attribute which can be 'augmented' by document handlers. We think of document annotations as "incremental", that is, as "adding" notes "on top of" previous notes. Thus we shall model document annotations as a repository: notes are added, i.e., annotations are augmented, previous notes are not edited, and no notes are deleted. We suggest that notes be time-stamped. The notes (of annotations) may be such which record handlers work on documents. Examples could be: *"November 15, 2021: 16:12 : This is version V."*, *"This document was released on November 15, 2021: 16:12 ."*, *"November 15, 2021: 16:12 : Section X.Y.Z of version III was deleted."*, *"November 15, 2021: 16:12 : References to documents* doc_i *and* doc_j *are inserted on Pages* p *and* q*, respectively."* and *"November 15, 2021: 16:12 : Final release."*

9.5.4 Document Contents: Text/Graphics

The main idea of a document, to us, is the *written* (i.e., text) and/or *drawn* (i.e., graphics) *contents*. We do not characterise any format for this *contents*. We may wish to insert, in the *contents*, references to locations in the *content*s of other documents. But, for now, we shall not go into such details. The main operations on documents, to us, are concerned with: their **creation, editing, reading, copying** and **shredding**. The **editing** and **reading** operations are mainly concerned with document *annotations* and *text/graphics*.

9.5.5 Document Histories

So documents are **create**d, **edit**ed, **read**, **copied** and **shred**ed. These operations are initiated by the management (**create**), by the archive (**create**), and by handlers (**edit, read, copy**), and at specific times.

9.5.6 A Summary of Document Attributes

784. As separate attributes of documents we have document descriptors, document annotations, document contents and document histories.
785. Document annotations are lists of document notes.
786. Document histories are lists of time-stamped document operation designators.
787. A document operation designator is either a create, or an edit, or a read, or a copy, or a shred designator.
788. A create designator identifies
    a. a handler and a time (at which the create request first arose), and presents
    b. elements for constructing a document descriptor, one which
        i. besides some further undefined information
        ii. refers to a set of documents (i.e., embeds reference to their unique identifiers),
    c. a (first) document note, and
    d. an empty document contents.
789. An edit designator identifies a handler, a time, and specifies a pair of edit/undo functions.

790. A read designator identifies a handler.
791. A copy designator identifies a handler, a time, the document to be copied (by its unique identifier, and a document note to be inserted in both the master and the copy document.
792. A shred designator identifies a handler.
793. An edit function takes a triple of a document annotation, a document note and document contents and yields a pair of a document annotation and a document contents.
794. An undo function takes a pair of a document note and document contents and yields a triple of a document annotation, a document note and a document contents.
795. Proper pairs of (edit,undo) functions satisfy some inverse relation.

There is, of course, no need, in any document history, to identify the identifier of that document.

```
type
784   DD, DA, DC, DH
value
784   attr_DD: D → DD
784   attr_DA: D → DA
784   attr_DC: D → DC
784   attr_DH: D → DH
type
785   DA = DN*
786   DH = (TIME × DO)*
787   DO == Crea | Edit | Read | Copy | Shre
788   Crea :: (HI × TIME) × (DI-set × Info) × DN × {|"empty_DC"|}
788(b)i   Info = ...
value
788(b)ii   embed_DIs_in_DD: DI-set × Info → DD
axiom
788d   "empty_DC" ∈ DC
type
789   Edit :: (HI × TIME) × (EDIT × UNDO)
790   Read :: (HI × TIME) × DI
791   Copy :: (HI × TIME) × DI × DN
792   Shre :: (HI × TIME) × DI
793   EDIT = (DA × DN × DC) → (DA × DC)
794   UNDO = (DA × DC) → (DA × DN × DC)
axiom
795   ∀ mkEdit(_,(e,u)):Edit •
795      ∀ (da,dn,dc):(DA×DN×DC) •
795         u(e(da,dn,dc))=(da,dn,dc)
```

9.6 Behaviours: An Informal, First View

In [48, Manifest Domains: Analysis & Description] we show that we can associate behaviours with parts, where parts are such discrete endurants for which we choose to model all its observable properties: unique identifiers, mereology and attributes, and where behaviours are sequences of actions, events and behaviours.

- The overall document handler system behaviour can be expressed in terms of the parallel composition of the behaviours

796. of the system core behaviour,
797. of the handler aggregate (the management) behaviour
798. and the document aggregate (the archive) behaviour,

with the (distributed) parallel composition of

799. all the behaviours of handlers and,

the (distributed) parallel composition of

800. at any one time, zero, one or more behaviours of documents.

- To express the latter

801. we need introduce two "global" values: an indefinite set of handler identifiers and an indefinite set of document identifiers.

```
value
801  his:HI-set, dis:DI-set

796     sys(...)
797  || mgtm(...)
798  || arch(...)
799  || ||{hdlr_i(...)|i:HI•i∈his}
800  || ||{docu_i(dd)(da,dc,dh)|i:DI•i∈dis}
```

For now we leave undefined the arguments, (...) etc., of these behaviours. The arguments of the document behaviour, (dd)(da,dc,dh), are the static, respectively the three programmable (i.e., dynamic) attributes: *document descriptor, document annotation, document contents* and *document history*. The above expressions, Items 797–800, do not define anything, they can be said to be "snapshots" of a "behaviour state". Initially there are no document behaviours, docu_i(dd)(da,dc,dh), Item 800. Document behaviours are "started" by the archive behaviour (on behalf of the management and the handler behaviours). Other than mentioning the system (core) behaviour we shall not model that behaviour further.

9.7 Channels, A First View

Channels are means for behaviours to synchronise and communicate values (such as unique identifiers, mereologies and attributes).

802. The management behaviour, mgtm, need to (synchronise and) communicate with the archive behaviour, arch, in order, for the management behaviour, to request the archive behaviour
    - to **create** (ab initio or due to **copy**ing)
    - or **shred** document behaviours, docu_j,

    and for the archive behaviour

    - to inform the management behaviour of the identity of the document(behaviour)s that it has created.

```
channel
802  mgtm_arch_ch:MA
```

803. The management behaviour, mgtm, need to (synchronise and) communicate with all handler behaviours, $hdlr_i$ and they, in turn, to (synchronised) communicate with the handler management behaviour, mgtm. The management behaviour need to do so in order
   - to inform a handler behaviour that it is granted access rights to a specific document, subsequently these access rights may be modified, including revoked.

   channel
   803 {mgtm_hdlr_ch[i]:MH|i:HI•i ∈ his}

804. The document archive behaviour, arch, need (synchronise and) communicate with all document behaviours, $docu_j$ and they, in turn, to (synchronise and) communicate with the archive behaviour, arch.

   channel
   804 {arch_docu_ch[j]:AD|h:DI•j ∈ dis}

805. Handler behaviours, $hdlr_i$, need (synchronise and) communicate with all the document behaviours, $docu_j$, with which it has operational allowance to so do so[64], and document behaviours, $docu_j$, need (synchronise and) communicate with potentially all handler behaviours, $hdlr_i$, namely those handler behaviours, $hdlr_i$ with which they have ("earlier" synchronised and) communicated.

   channel
   805 {hdlr_docu_ch[i,j]:HD|i:HI,j:DI•i ∈ his∧j ∈ dis}

806. At present we leave undefined the type of messages that are communicated.

   type
   806 MA, MH, AD, HD

9.8 An Informal Graphical System Rendition

Figure 9.1 on the facing page is an informal rendition of the "state" of a number of behaviours: a single management behaviour, a single archive behaviour, a fixed number, n_h, of one or more handler behaviours, and a variable, initially zero number of document behaviours, with a maximum of these being n_d. The figure also indicates, again rather informally, the channels between these behaviours: one channel between the management and the archive behaviours; n_h channels (n_h is, again, informally indicated) between the management behaviour and the n_h handler behaviours; n_d channels (n_d is, again, informally indicated) between the archive behaviour and the n_d document behaviours; and $n_h \times n_d$ channels ($n_d \times n_d$ is, again, informally indicated) between the n_h handler behaviours and the n_d document behaviours

9.9 Behaviour Signatures

807. The mgtm behaviour (synchronises and) communicates with the archive behaviour and with all of the handler behaviours, $hdlr_i$.
808. The archive behaviour (synchronises and) communicates with the mgtm behaviour and with all of the document behaviours, $docu_j$.

[64] The notion of operational allowance will be explained below.

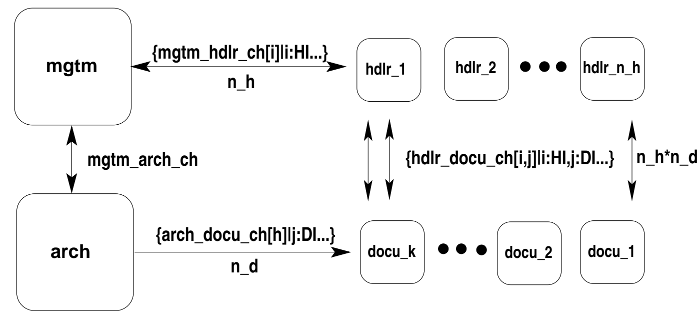


Fig. 9.1 An Informal Snapshot of System Behaviours

809. The signature of the generic handler behaviours, hdlr_i expresses that they [occasionally] receive "orders" from management, and otherwise [regularly] interacts with document behaviours.
810. The signature of the generic document behaviours, docu_j expresses that they [occasionally] receive "orders" from the archive behaviour and that they [regularly] interacts with handler behaviours.

value
807 mgtm: ... → **in**,**out** mgtm_arch_ch, {mgtm_hdlr_ch[i]|i:HI•i ∈ his} **Unit**
808 arch: ... → **in**,**out** mgtm_arch_ch, {arch_docu_ch[j]|j:DI•j ∈ dis} **Unit**
809 hdlr_i: ... → **in** mgtm_hdlr_ch[i], **in**,**out** {hdlr_docu_ch[i,j]|j:DI•j∈dis} **Unit**
810 docu_j: ... → **in** mgtm_arch_ch, **in**,**out** {hdlr_docu_ch[i,j]|i:HI•i ∈ his} **Unit**

9.10 Time

9.10.1 Time and Time Intervals: Types and Functions

811. We postulate a notion of time, one that covers both a calendar date (from before Christ up till now and beyond). But we do not specify any concrete type (i.e., format such as: YY:MM:DD, HH:MM:SS).
812. And we postulate a notion of (signed) time interval — between two times (say: ±YY:MM:DD:HH:MM:SS).
813. Then we postulate some operations on time: Adding a time interval to a time obtaining a time; subtracting one time from another time obtaining a time interval, multiplying a time interval with a natural number; etc.
814. And we postulate some relations between times and between time intervals.

type
811 TIME
812 TIME_INTERVAL
value
813 add: TIME_INTERVAL × TIME → TIME
813 sub: TIME × TIME → TIME_INTERVAL
813 mpy: TIM_INTERVALE × **Nat** → TIME_INTERVAL
814 <,≤,=,≠,≥,>: ((TIME×TIME)|(TIME_INTERVAL×TIME_INTERVAL)) → **Bool**

9.10.2 A Time Behaviour and a Time Channel

815. We postulate a[n "ongoing"] time behaviour: it either keeps being a time behaviour with unchanged time, t, or – internally non-deterministically – chooses being a time behaviour with a time interval incremented time, t+ti, or – internally non-deterministically – chooses to [first] offer its time on a [global] channel, time_ch, then resumes being a time behaviour with unchanged time., t
816. The time interval increment, ti, is likewise internally non-deterministically chosen. We would assume that the increment is "infinitesimally small", but there is no need to specify so.
817. We also postulate a channel, time_ch, on which the time behaviour offers time values to whoever so requests.

```
value
815  time: TIME → time_ch  TIME  Unit
815  time(t) ≡ (time(t) ⊓ time(t+ti) ⊓ time_ch!t ; time(t))
816  ti:TIME_INTERVAL ...
channel
817  time_ch:TIME
```

9.10.3 An Informal RSL Construct

The formal-looking specifications of this report appear in the style of the RAISE [93] Specification Language, RSL [92]. We shall be making use of an informal language construct:

- **wait** ti.

wait is a keyword; ti designates a time interval. A typical use of the wait construct is:

- ... *ptA* ; **wait** ti; *ptB* ; ...

If at specification text point *ptA* we may assert that time is t, then at specification text point *ptB* we can assert that time is t+ti.

9.11 Behaviour "States"

We recall that the endurant parts, Management, Archive, Handlers, and Documents, have properties in the form of *unique identifiers, mereologies* and *attributes*. We shall not, in this research note, deal with possible mereologies of these endurants. In this section we shall discuss the endurant attributes of mgtm (management), arch (archive), hdlrs (handlers), and docus (documents). Together the values of these properties, notably the attributes, constitute states – and, since we associate behaviours with these endurants, we can refer to these states also a behaviour states. Some attributes are static, i.e., their value never changes. Other attributes are dynamic.[65] Document handling systems are rather conceptual, i.e., abstract in nature. The dynamic attributes, therefore, in this modeling "exercise", are constrained to just the *programmable* attributes. Programmable attributes are those whose value is set by "their" behaviour. For a behaviour β we shall show the static attributes as one set of parameters and the programmable attributes as another set of parameters.

[65] We refer to Sect. 3.4 of [48], and in particular its subsection 3.4.4.

value β: Static → Program → ... **Unit**

818. For the management endurant/behaviour we focus on one programmable attribute. The management behaviour needs keep track of all the handlers it is charged with, and for each of these which zero, one or more documents they have been granted access to (cf. Sect. **9.12.3** on the next page). Initially that management directory lists a number of handlers, by their identifiers, but with no granted documents.
819. For the archive behaviour we similarly focus on one programmable attribute. The archive behaviour needs keep track of all the documents it has used (i.e., created), those that are available (and not yet used), and of those it has shredded. Initially all these three archive directory sets are empty.
820. For the handler behaviour we similarly focus on one programmable attribute. The handler behaviour needs keep track of all the documents it has been charged with and its access rights to these.
821. Document attributes we mentioned above, cf. Items 784–787.

```
type
818  MDIR = HI →m (DI →m ANm-set)
819  ADIR = avail:DI-set × used:DI-set × gone:DI-set
820  HDIR = DI →m ANm-set
821  SDATR = DD, PDATR = DA × DC × DH
axiom
819  ∀ (avail,used,gone):ADIR • avail ∩ used = {} ∧ gone ⊆ used
```

We can now "complete" the behaviour signatures. We omit, for now, static attributes.

```
value
807  mgtm:  MDIR  → in,out mgtm_arch_ch, {mgtm_hdlr_ch[i]|i:HI•i ∈ his}  Unit
808  arch:   ADIR  → in,out mgtm_arch_ch, {arch_docu_ch[j]|j:DI•j ∈ dis}  Unit
809  hdlr_i:  HDIR  → in mgtm_hdlr_ch[i], in,out {hdlr_docu_ch[i,j]|j:DI•j∈dis}  Unit
810  docu_j: SDATR → PDATR  → in mgtm_arch_ch, in,out {hdlr_docu_ch[i,j]|i:HI•i ∈ his}  Unit
```

9.12 Inter-Behaviour Messages

Documents are not "fixed, innate" entities. They embody a "history", they have a "past". Somehow or other they "carry a trace of all the "things" that have happened/occurred to them. And, to us, these things are the manipulations that management, via the archive and handlers perform on documents.

9.12.1 Management Messages with Respect to the Archive

822. Management **create** documents. It does so by requesting the archive behaviour to allocate a document identifier and initialize the document "state" and start a document behaviour, with initial information, cf. Item 788 on page 185:
    a. the identity of the initial handler of the document to be created,
    b. the time at which the request is being made,

c. a document descriptor which embodies a (finite) set of zero or more (used) document identifiers (dis),
d. a document annotation note dn, and
e. an initial, i.e., "empty" contents, "empty_DC".

type
788. Crea :: (HI × TIME) × (DI-**set** × Info) × DN × {|"empty_DC"|} [cf. formula Item 788, Page 186]

823. The management behaviour passes on to the archive behaviour, requests that it accepts from handlers behaviours, for the copying of document:

823 Copy :: DI × HI × TIME × DN [cf. Item 833 on page 194]

824. Management **schred**s documents by informing the archive behaviour to do so.

type
824 Shred :: TIME × DI

9.12.2 Management Messages with Respect to Handlers

825. Upon receiving, from the archive behaviour, the "feedback" the identifier of the created document (behaviour):

type
825. Create_Reply :: NewDocID(di:DI)

826. the management behaviour decides to **grant** access rights, acrs:ACRS[66], to a document handler, hi:HI.

type
826 Gran :: HI × TIME × DI × ACRS

9.12.3 Document Access Rights

Implicit in the above is a notion of document access rights.

827. By document access rights we mean a set of action names.
828. By an action name we mean such tokens that indicate either of the document handler operations indicate above.

type
827 ACRS = ANm-**set**
828 ANm = {|"edit","read","copy"|}

[66] For the concept of access rights see Sect. **9.12.3**.

9.12.4 Archive Messages with Respect to Management

To create a document management provides the archive with some initial information. The archive behaviour selects a document identifier that has not been used before.

829. The archive behaviour informs the management behaviour of the identifier of the created document.

type
829 NewDocID :: DI

9.12.5 Archive Message with Respect to Documents

830. To shred a document the archive behaviour must access the designated document in order to **stop** it. No "message", other than a symbolic `"stop"`, need be communicated to the document behaviour.

type
830 Shred :: {|"stop"|}

9.12.6 Handler Messages with Respect to Documents

Handlers, generically referred to by hdlr_i, may perform the following operations on documents: **edit, read** and **copy**. (Management, via the archive behaviour, **create**s and **shred**s documents.)

831. To perform an **edit** action handler hdlr_i must provide the following:
   - the document identity – in the form of a (i:HI,j:DI) channel hdlr_docu_ch index value,
   - the handler identity, i,
   - the time of the edit request,
   - and a pair of functions: one which performs the editing and one which un-does it !

   type
   831 Edit :: DI × HI × TIME × (EDIT × UNDO)

832. To perform a **read** action handler hdlr_i must provide the following information:
   - the document identity – in the form of a di:DI channel hdlr_docu_ch index value,
   - the handler identity and
   - the time of the read request.

   type
   832 Read :: DI × HI × TIME

9.12.7 Handler Messages with Respect to Management

833. To perform a **copy** action, a handler, hdlr_i, must provide the following information to the management behaviour, mgtm:
   - the document identity,
   - the handler identity – in the form of an hi:HI channel mgtm_hdlr_ch index value,
   - the time of the copy request, and
   - a document note (to be affixed both the master and the copy documents).

833 Copy :: DI × HI × TIME × DN [cf. Item 823 on page 192]

How the handler, the management, the archive and the "named other" handlers then enact the copying, etc., will be outlined later.

9.12.8 A Summary of Behaviour Interactions

Figure 9.2 summarises the sources, **out**, resp. !, and the targets, **in**, resp. ?, of the messages covered in the previous sections.

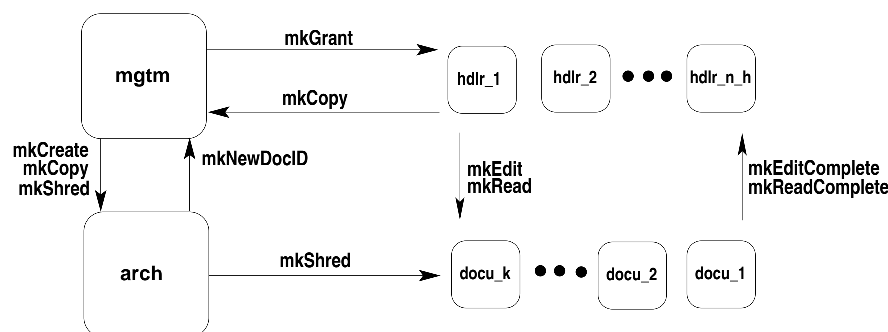


Fig. 9.2 A Summary of Behaviour Interactions

9.13 A General Discussion of Handler and Document Interactions

We think of documents being manifest. Either a document is in paper form, or it is in electronic form. In paper form we think of a document as being in only one – and exactly one – physical location. In electronic form a document is also in only one – and exactly one – physical location. No two handlers can access the same document at the same time or in overlapping time intervals. If your conventional thinking makes you think that two or more handlers can, for example, read the same document "at the same time", then, in fact, they are reading either a master and a copy of that master, or they are reading two copies of a common master.

9.14 Channels: A Final View

We can now summarize the types of the various channel messages first referred to in Items 802, 803, 804 and 805.

type
802 MA = Create (Item 822 on page 191)
802 | Shred (Item 822d on page 192)
802 | NewDocID (Item 829 on page 193)
803 MH = Grant (Item 822c on page 192)
803 | Copy (Item 833 on the preceding page)
804 AD = Shred (Item 830 on page 193)
805 HD = Edit (Item 831 on page 193)
805 | Read (Item 832 on page 193)
805 | Copy (Item 833 on the preceding page)

9.15 An Informal Summary of Behaviours

9.15.1 The Create Behaviour: Left Fig. 9.3 on the following page

834. [1] The management behaviour, at its own volition, initiates a create document behaviour. It does so by offering a create document message to the archive behaviour.
    a. [1.1] That message contains a meaningful document descriptor,
    b. [1.2] an initial document annotation,
    c. [1.3] an "empty" document contents and
    d. [1.4] a single element document history.

    (We refer to Sect. **9.12.1** on page 191, Items 822–822e.)
835. [2] The archive behaviour offers to accept that management message. It then selects an available document identifier (here shown as k), henceforth marking k as used.
836. [3] The archive behaviour then "spawns off" document behaviour docu_k – here shown by the "dash–dotted" rounded edge square.
837. [4] The archive behaviour then offers the document identifier k message to the management behaviour.
    (We refer to Sect. **9.12.4** on page 193, Item 829.)
838. [5] The management behaviour then
    a. [5.1] selects a handler, here shown as i, i.e., hdlr_i,
    b. [5.2] records that that handler is granted certain access rights to document k,
    c. [5.3] and offers that granting to handler behaviour i.

    (We refer to Sect. **9.12.2** on page 192, Item 826 on page 192.)
839. [6] Handler behaviour i records that it now has certain access rights to document i.

9.15.2 The Edit Behaviour: Right Fig. 9.3 on the next page

1 Handler behaviour i, at its own volition, initiates an edit action on document j (where i has editing rights for document j). Handler i, optionally, provides document j with a(annotation) note.

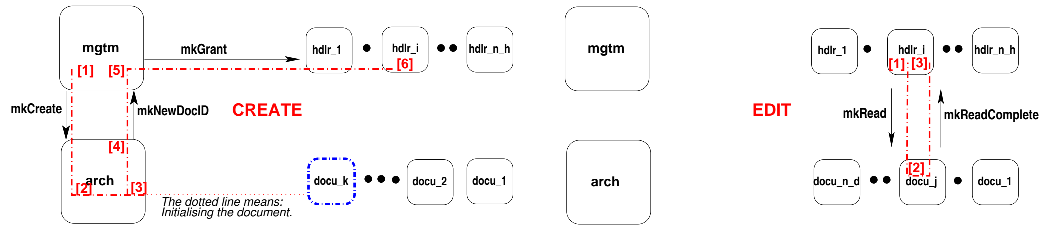


Fig. 9.3 Informal Snapshots of Create and Edit Document Behaviours

While editing document *j* handler *i* also "selects" an appropriate pair of *edit/undo* functions for document *j*.

2 Document behaviour *j* accepts the editing request, enacts the editing, optionally appends the (annotation) note, and, with handler *i*, completes the editing, after some time interval ti.

3 Handler behaviour *i* completes its edit action.

9.15.3 The Read Behaviour: Left Fig. 9.4

1 Handler behaviour *i*, at its own volition, initiates a read action on document *j* (where *i* has reading rights for document *j*). Handler *i*, optionally, provides document *j* with a(annotation) note.

2 Document behaviour *j* accepts the reading request, enacts the reading by providing the handler, *i*, with the document contents, and optionally appends the (annotation) note, and, with handler *i*, completes the reading, after some time interval ti.

3 Handler behaviour *i* completes its read action.

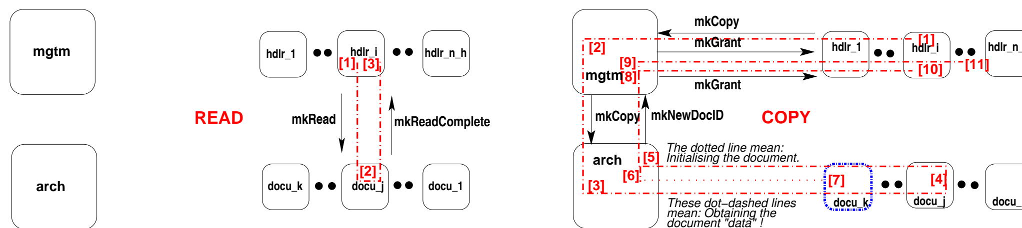


Fig. 9.4 Informal Snapshots of Read and Copy Document Behaviours

9.15.4 The Copy Behaviour: Right Fig. 9.4

1 Handler behaviour *i*, at its own volition, initiates a copy action on document *j* (where *i* has copying rights for document *j*). Handler *i*, optionally, provides master document *j* as well as the copied document (yet to be identified) with respective (annotation) notes.

2 The management behaviour offers to accept the handler message. As for the create action, the management behaviour offers a combined *copy and create* document message to the archive behaviour.
3 The archive behaviour selects an available document identifier (here shown as k), henceforth marking k as used.
4 The archive behaviour then obtains, from the master document j its *d*ocument *d*escriptor, dd_j, its *d*ocument *a*nnotations, da_j, its *d*ocument *c*ontents, dc_j, and its document history, dh_j.
5 The archive behaviour informs the management behaviour of the identifier, k, of the (new) document copy,
6 while assembling the attributes for that (new) document copy: its *d*ocument *d*escriptor, dd_k, its *d*ocument *a*nnotations, da_k, its *d*ocument *c*ontents, dc_k, and its document history, dh_k, from these "similar" attributes of the master document j,
7 while then "spawning off" document behaviour docu_k – here shown by the "dash–dotted" rounded edge square.
8 The management behaviour accepts the identifier, k, of the (new) document copy, recording the identities of the handlers and their access rights to k,
9 while informing these handlers (informally indicated by a "dangling" dash-dotted line) of their grants,
10 while also informing the master copy of the copy identity (et cetera).
11 The handlers granted access to the copy record this fact.

9.15.5 The Grant Behaviour: Left Fig. 9.5

This behaviour has its

1 Item [1] correspond, in essence, to Item [9] of the copy behaviour – see just above – and
2 Item [2] correspond, in essence, to Item [11] of the copy behaviour.

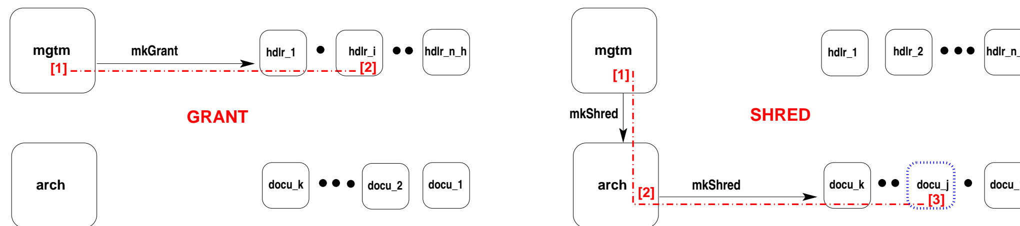


Fig. 9.5 Informal Snapshots of Grant and Shred Document Behaviours

9.15.6 The Shred Behaviour: Right Fig. 9.5

1 The management, at its own volition, selects a document, j, to be shredded. It so informs the archive behaviour.
2 The archive behaviour records that document j is to be no longer in use, but shredded, and informs document j's behaviour.

3 The document j behaviour accepts the shred message and **stop**s (indicated by the dotted rounded edge box).

9.16 The Behaviour Actions

To properly structure the definitions of the four kinds of (management, archive, handler and document) behaviours we single each of these out "across" the six behaviour traces informally described in Sects. **9.15.1–9.15.6**. The idea is that if behaviour β is involved in τ traces, $\tau_1, \tau_2, ..., \tau_\tau$, then behaviour β shall be defined in terms of τ non-deterministic alternative behaviours named $\beta_{\tau_1}, \beta_{\tau_2}, ..., \beta_{\tau_\tau}$.

9.16.1 Management Behaviour

840. The management behaviour is involved in the following action traces:

a. **create** — Fig. 9.3 on page 196 Left
b. **copy** — Fig. 9.4 on page 196 Right
c. **grant** — Fig. 9.5 on the preceding page Left
d. **shred** — Fig. 9.5 on the previous page Right

```
value
840   mgtm: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
840   mgtm(mdir) ≡
840a       mgtm_create(mdir)
840b     ⨅ mgtm_copy(mdir)
840c     ⨅ mgtm_grant(mdir)
840d     ⨅ mgtm_shred(mdir)
```

9.16.1.1 Management Create Behaviour: Left Fig. 9.3 on page 196

841. The **management create** behaviour
842. initiates a create document behaviour (i.e., a request to the archive behaviour),
843. and then awaits its response.

```
value
841   mgtm_create: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
841   mgtm_create(mdir) ≡
842   [1]   let hi = mgtm_create_initiation(mdir) ;        [Left Fig. 9.3 on page 196]
843   [5]   mgtm_create_awaits_response(mdir)(hi) end  [Left Fig. 9.3 on page 196]
```

The **management create initiation** behaviour

844. selects a handler on behalf of which it requests the document creation,
845. assembles the elements of the create message:
   - by embedding a set of zero or more document references, dis, with some information, info, into a document descriptor, adding
   - a document note, dn, and

- and initial, that is, empty document contents, "empty_DC",

846. offers such a create document message to the archive behaviour, and
847. yields the identifier of the chosen handler.

```
value
842 mgtm_create_initiation: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} HI
842 mgtm_create_initiation(mdir) ≡
844          let hi:HI • hi ∈ dom mdir,
845 [1.2–.4]      (dis,info):(DI-set×Info),dn:DN • is_meaningful(embed_DIs_in_DD(dis,info))(mdir) in
846 [1.1]     mgtm_arch_ch ! mkCreate(embed_DIs_in_DD(ds,info),dn,″empty_DC″)
847          hi end

845 is_meaningful: DD → MDIR → Bool [left further undefined]
```

The **management create awaits response** behaviour

848. starts by awaiting a reply from the archive behaviour with the identity, *di*, of the document (that that behaviour has created).
849. It then selects suitable access rights,
850. with which it updates its handler/document directory
851. and offers to the chosen handler
852. whereupon it resumes, with the updated management directory, being the management behaviour.

```
value
843 mgtm_create_awaits_response: MDIR → HI → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
843 mgtm_create_awaits_response(mdir) ≡
848 [5]      let mkNewDocID(di) = mgtm_arch_ch ? in
849 [5.1]     let acrs:ANm-set in
850 [5.2]     let mdir′ = mdir † [hi ↦ [di ↦ acrs]] in
851 [5.3]     mgtm_hdlr_ch[hi] ! mkGrant(di,acrs)
852          mgtm(mdir′) end end end
```

9.16.1.2 Management Copy Behaviour: Right Fig. 9.4 on page 196

853. The **management copy** behaviour
854. accepts a copy document request from a handler behaviour (i.e., a request to the archive behaviour),
855. and then awaits a response from the archive behaviour;
856. after which it grants access rights to handlers to the document copy.

```
value
853 mgtm_copy: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
853 mgtm_copy(mdir) ≡
854 [2] let hi = mgtm_accept_copy_request(mdir) in
855 [8] let di = mgtm_awaits_copy_response(mdir)(hi) in
856 [9] mgtm_grant_access_rights(mdir)(di) end end
```

857. The **management accept copy** behaviour non-deterministically externally (⫿) awaits a copy request from a[ny] handler (*i*) behaviour –
858. with the request identifying the master document, *j*, to be copied.

859. The management accept copy behaviour forwards (!) this request to the archive behaviour –
860. while yielding the identity of the requesting handler.

```
857.  mgtm_accept_copy_request: MDIR →
857.      in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} HI
857.  mgtm_accept_copy_request(mdir) ≡
858.      let mkCopy(di,hi,t,dn) = ⌈⌉{mgtm_hdlr_ch[i]?|i:HI•i ∈ his} in
859.      mgtm_arch_ch ! mkCopy(di,hi,t,dn) ;
859.      hi end
```

The **management awaits copy response** behaviour

861. awaits a reply from the archive behaviour as to the identity of the newly created copy (*di*) of master document *j*.
862. The management awaits copy response behaviour then informs the 'copying-requesting' handler, *hi*, that the copying has been completed and the identity of the copy (*di*) –
863. while yielding the identity, *di*, of the newly created copy.

```
840b.  mgtm_awaits_copy_response: MDIR → HI →
840b.      in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} DI
840b.  mgtm_awaits_copy_response(mdir)(hi) ≡
861.  [8]  let mkNewDocID(di) = mgtm_arch_ch ? in
862.       mgtm_hdlr_ch[hi] ! mkCopy(di) ;
863.       di end
```

The **management grants access rights** behaviour

864. selects suitable access rights for a suitable number of selected handlers.
865. It then offers these to the selected handlers.

```
856.  mgtm_grant_access_rights: MDIR → DI →
856.     in,out {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
856.  mgtm_grant_access_rights(mdir)(di) ≡
864.     let diarm = [hi↦acrs|hi:HI,arcs:ANm-set• hi ∈ dom mdir∧arcs⊆(diarm(hi))(di)] in
865.     || {mgtm_hdlr_ch[hi]!mkGrant(hi,time_ch?,di,acrs) |
865.        hi:HI,acrs:ANm-set•hi ∈ dom diarm∧acrs⊆(diarm(hi))(di)} end
```

9.16.1.3 Management Grant Behaviour: Left Fig. 9.5 on page 197

The **management grant** behaviour

866. is a variant of the mgtm_grant_access_rights function, Items 864–865.
867. The management behaviour selects a suitable subset of known handler identifiers, and
868. for these a suitable subset of document identifiers from which
869. it then constructs a map from handler identifiers to subsets of access rights.
870. With this the management behaviour then issues appropriate grants to the chosen handlers.

```
type
    MDIR = HI →m (DI →m ANm-set)
value
866  mgtm_grant: MDIR → in,out {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
866  mgtm_grant(mdir) ≡
867     let his ⊆ dom dir in
```

```
868    let dis ⊆ ∪{dom mdir(hi)|hi:HI•hi ∈ his} in
869    let diarm = [ hi↦acrs|hi:HI,di:DI,acrs:ANm-set• hi ∈ his∧di ∈ dis∧acrs⊆(diarm(hi))(di) ] in
870    ||{mgtm_hdlr_ch[ hi ]!mkGrant(di,acrs) |
870      hi:HI,di:DI,acrs:ANm-set•hi ∈ dom diarm∧di ∈ dis∧acrs⊆(diarm(hi))(di)}
866    end end end
```

9.16.1.4 Management Shared Behaviour: Right Fig. 9.5 on page 197

The **management shred** behaviour

871. initiates a request to the archive behaviour.
872. First the management shred behaviour selects a document identifier (from its directory).
873. Then it communicates a shred document message to the archive behaviour;
874. then it notes the (to be shredded) document in its directory
875. whereupon the management shred behaviour resumes being the management behaviour.

```
value
871  mgtm_shred: MDIR → out mgtm_arch_ch  Unit
871  mgtm_shred(mdir) ≡
872      let di:DI • is_suitable(di)(mdir) in
873  [ 1 ]  mgtm_arch_ch ! mkShred(time_ch?,di) ;
874      let mdir′ = [ hi↦mdir(hi)\{di}|hi:HI•hi ∈ dom mdir ] in
875      mgtm(mdir′) end end
```

9.16.2 Archive Behaviour

876. The archive behaviour is involved in the following action traces:
    a. **create** — Fig. 9.3 on page 196 Left
    b. **copy** — Fig. 9.4 on page 196 Right
    c. **shred** — Fig. 9.5 on page 197 Right

```
type
819  ADIR = avail:DI-set × used:DI-set × gone:DI-set
axiom
819  ∀ (avail,used,gone):ADIR • avail ∩ used = {} ∧ gone ⊆ used
value
876  arch: ADIR → in,out mgmt_arch_ch, {arch_docu_ch[ di ]|di:DI•di ∈ dis} Unit
876a  arch(adir) ≡
876a       arch_create(adir)
876b     ⊓ arch_copy(adir)
876c     ⊓ arch_shred(adir)
```

9.16.2.1 The Archive Create Behaviour: Left Fig. 9.3 on page 196

The **archive create** behaviour

877. accepts a request, from the management behaviour to create a document;

878. it then selects an available document identifier;
879. communicates this new document identifier to the management behaviour;
880. while initiating a new document behaviour, docu_{di}, with the document descriptor, *dd*, the initial document annotation being the singleton list of the note, *an*, and the initial document contents, *dc* – all received from the management behaviour – and an initial document history of just one entry: the date of creation, all
881. in parallel with resuming the archive behaviour with updated programmable attributes.

```
876a.  arch_create: AATTR → in,out mgmt_arch_ch, {arch_docu_ch[di]|di:DI•di ∈ dis} Unit
876a.  arch_create(avail,used,gone) ≡
877.   [2] let mkCreate((hi,t),dd,an,dc) = mgmt_arch_ch ? in
878.         let di:DI•di ∈ avail in
879.   [4] mgmt_arch_ch ! mkNewDocID(di) ;
880.   [3] docu_di(dd)(⟨an⟩,dc,<(date_of_creation)>)
881.         || arch(avail\{di},used∪{di},gone)
876a.        end end
```

9.16.2.2 The Archive Copy Behaviour: Right Fig. 9.4 on page 196

The **archive copy** behaviour

882. accepts a copy document request from the management behaviour with the identity, *j*, of the master document;
883. it communicates (the request to obtain all the attribute values of the master document, *j*) to that document behaviour;
884. whereupon it awaits their communication (i.e., (*dd*,*da*,*dc*,*dh*));
885. (meanwhile) it obtains an available document identifier,
886. which it communicates to the management behaviour,
887. while initiating a new document behaviour, docu_{di}, with the master document descriptor, *dd*, the master document annotation, and the master document contents, *dc*, and the master document history, *dh* (all received from the master document),
888. in parallel with resuming the archive behaviour with updated programmable attributes.

```
876b.  arch_copy: AATTR → in,out mgmt_arch_ch, {arch_docu_ch[di]|di:DI•di ∈ dis} Unit
876b.  arch_copy(avail,used,gone) ≡
882.   [3] let mkDocID(j,hi) = mgtm_arch_ch ? in
883.         arch_docu_ch[j] ! mkReqAttrs() ;
884.         let mkAttrs(dd,da,dc,dh) = arch_docu_ch[j] ? in
885.         let di:DI • di ∈ avail in
886.         mgtm_arch_ch ! mkCopyDocID(di) ;
887.  [6,7] docu_di(augment(dd,′′copy′′,j,hi),
887.                augment(da,′′copy′′,hi),dc,
887.                augment(dh,(′′copy′′,date_and_time,j,hi)))
888.       || arch(avail\{di},used∪{di},gone)
876b.       end end end
```

where we presently leave the [overloaded] augment functions undefined.

9.16.2.3 The Archive Shred Behaviour: Right Fig. 9.5 on page 197

The **archive shred** behaviour

889. accepts a shred request from the management behaviour.
890. It communicates this request to the identified document behaviour.
891. And then resumes being the archive behaviour, noting however, that the shredded document has been shredded.

```
876c.  arch_shred: AATTR → in,out mgmt_arch_ch, {arch_docu_ch[di]|di:DI•di ∈ dis} Unit
876c.  arch_shred(avail,used,gone) ≡
889.   [2] let mkShred(j) = mgmt_arch_ch ? in
890.       arch_docu_ch[j] ! mkShred() ;
891.       arch(avail,used,gone∪{j})
876c.      end
```

9.16.3 Handler Behaviours

892. The handler behaviour is involved in the following action traces:
   a. **create** — Fig. 9.3 on page 196 Left
   b. **edit** — Fig. 9.3 on page 196 Right
   c. **read** — Fig. 9.4 on page 196 Left
   d. **copy** — Fig. 9.4 on page 196 Right
   e. **grant** — Fig. 9.5 on page 197 Left

```
value
892   hdlr_hi: HATTRS → in,out mgtm_hdlr_ch[hi],{hdlr_docu_ch[hi,di]|di:DI•di∈dis} Unit
892   hdlr_hi(hattrs) ≡
892a       hdlr_create_hi(hattrs)
892b    ⌈⌉ hdlr_edit_hi(hattrs)
892c    ⌈⌉ hdlr_read_hi(hattrs)
892d    ⌈⌉ hdlr_copy_hi(hattrs)
892e    ⌈⌉ hdlr_grant_hi(hattrs)
```

9.16.3.1 The Handler Create Behaviour: Left Fig. 9.3 on page 196

893. The **handler create** behaviour offers to accept the granting of access rights, acrs, to document di.
894. It according updates its programmable hattrs attribute;
895. and resumes being a handler behaviour with that update.

```
892a  hdlr_create_hi: HATTRS × HHIST → in,out mgtm_hdlr_ch[hi] Unit
892a  hdlr_create_hi(hattrs,hhist) ≡
893    let mkGrant(di,acrs) = mgtm_hdlr_ch[hi] ? in
894    let hattrs′ = hattrs † [hi ↦ acrs] in
895    hdlr_create_hi(hattrs′,augment(hhist,mkGrant(di,acrs))) end end
```

9.16.3.2 The Handler Edit Behaviour: Right Fig. 9.3 on page 196

896. The handler behaviour, on its own volition, decides to edit a document, *di*, for which it has editing rights.

897. The handler behaviour selects a suitable (...) pair of *e*dit/*u*ndo functions and a suitable (annotation) note.
898. It then communicates the desire to edit document *di* with (*e*,*u*) (at time *t*=time_ch?).
899. Editing take some time, *ti*.
900. We can therefore assert that the time at which editing has completed is *t*+*ti*.
901. The handler behaviour accepts the edit completion message from the document handler.
902. The handler behaviour can therefore resume with an updated document history.

```
892b  hdlr_edit_hi: HATTRS × HHIST → in,out {hdlr_docu_ch[hi,di]|di:DI•di∈dis} Unit
892b  hdlr_edit_hi(hattrs,hhist) ≡
896 [1]  let di:DI • di ∈ dom hattrs ∧ ′′edit′′ ∈ hattrs(di) in
897 [1]  let (e,u):(EDIT×UNDO) • ... , n:AN • ... in
898 [1]  hdlr_docu_ch[hi,di] ! mkEdit(hi,t=time_ch?,e,u,n) ;
899 [2]  let ti:TIME_INTERVAL • ... in
900 [2]  wait ti ; assert: time_ch? = t+ti
901 [3]  let mkEditComplete(ti′,...) = hdlr_docu_ch[hi,di] ? in assert ti′ ≅ ti
902        hdlr_hi(hattrs,augment(hhist,(di,mkEdit(hi,t,ti,e,u))))
892b       end end end end
```

9.16.3.3 The Handler Read Behaviour: Left Fig. 9.4 on page 196

903. The **handler behaviour**, on its own volition, decides to read a document, *di*, for which it has reading rights.
904. It then communicates the desire to read document *di* with at time *t*=time_ch? – with an annotation note (*n*).
905. Reading take some time, *ti*.
906. We can therefore assert that the time at which reading has completed is *t*+*ti*.
907. The handler behaviour accepts the read completion message from the document handler.
908. The handler behaviour can therefore resume with an updated document history.

```
892c  hdlr_edit_hi: HATTRS × HHIST → in,out {hdlr_docu_ch[hi,di]|di:DI•di∈dis} Unit
892c  hdlr_edit_hi(hattrs,hhist) ≡
903 [1]  let di:DI • di ∈ dom hattrs ∧ ′′read′′ ∈ hattrs(di), n:N • ... in
904 [1]  hdlr_docu_ch[hi,di] ! mkRead(hi,t=time_ch?,n) ;
905 [2]  let ti:TIME_INTERVAL • ... in
906 [2]  wait ti ; assert: time_ch? = t+ti
907 [3]  let mkReadComplete(ti,...) = hdlr_docu_ch[hi,di] ? in
908        hdlr_hi(hattrs,augment(hhist,(di,mkRead(di,t,ti))))
892c       end end end
```

9.16.3.4 The Handler Copy Behaviour: Right Fig. 9.4 on page 196

909. The **handler [copy] behaviour**, on its own volition, decides to copy a document, *di*, for which it has copying rights.
910. It communicates this copy request to the management behaviour.
911. After a while the handler [copy] behaviour receives acknowledgment of a completed copying from the management behaviour.
912. The handler [copy] behaviour records the request and acknowledgment in its, thus updated whereupon the handler [copy] behaviour resumes being the handler behaviour.

```
892d  hdlr_copy_hi: HATTRS × HHIST → in,out mgtm_hdlr_ch[ hi ] Unit
892d  hdlr_copy_hi(hattrs,hhist) ≡
909    [ 1 ] let di:DI • di ∈ dom hattrs ∧ ″copy″ ∈ hattrs(di) in
910    [ 1 ] mgtm_hdlr_ch[ hi ] ! mkCopy(di,hi,t=time_ch?) ;
911   [ 10 ] let mkCopyComplete(di′,di) = mgtm_hdlr_ch[ hi ] ? in
912   [ 10 ] hdlr_hi(hattrs,augment(hhist,time_ch?,(mkCopy(di,hi,,t),mkCopyComplete(di′))))
892d        end end
```

9.16.3.5 The Handler Grant Behaviour: Left Fig. 9.5 on page 197

913. The **handler [grant] behaviour** offers to accept grant permissions from the management behaviour.
914. In response it updates its handler attribute while resuming being a handler behaviour.

```
892e  hdlr_grant_hi: HATTRS × HHIST → in,out mgtm_hdlr_ch[ hi ] Unit
892e  hdlr_grant_hi(hattrs,hhist) ≡
913   [ 2 ] let mkGrant(di,acrs) = mgtm_hdlr_ch[ hi ] ? in
914   [ 2 ] hdlr_hi(hattrs†[ di↦acrs ],augment(hhist,time_ch?,mkGrant(di,acrs)))
892e       end
```

9.16.4 Document Behaviours

915. The document behaviour is involved in the following action traces:
   a. **edit** Fig. 9.3 on page 196 Right
   b. **read** Fig. 9.4 on page 196 Left
   c. **shred** Fig. 9.5 on page 197 Right

```
value
915  docu_di: DD × (DA × DC × DH) →
915     in,out arch_docu_ch[ di ], {hdlr_docu_ch[ hi,di ]|hi:HI•hi∈his} Unit
915  docu_di(dattrs) ≡
915a         docu_edit_di(dd)(da,dc,dh)
915b      ⊓ docu_read_di(dd)(da,dc,dh)
915c      ⊓ docu_shred_di(dd)(da,dc,dh)
```

9.16.4.1 The Document Edit Behaviour: Right Fig. 9.3 on page 196

916. The **document [edit] behaviour** offers to accept edit requests from document handlers.
   a. The document contents is edited, over a time interval of *ti*, with respect to the handlers edit function (*e*),
   b. the document annotations are augmented with respect to the handlers note (*n*), and
   c. the document history is augmented with the fact that an edit took place, at a certain time, with a pair of *edit*/*undo* functions.
917. The *e*dit (etc.) function(s) take some time, *ti*, to do.
918. The handler behaviour is notified, mkEditComplete(...) of the completion of the edit, and

919. the document behaviour is then resumed with updated programmable attributes.

```
value
915a  docu_edit_di: DD × (DA × DC × DH) → in,out {hdlr_docu_ch[hi,di]|hi:HI•hi∈his} Unit
915a  docu_edit_di(dd)(da,dc,dh) ≡
916   [2] let mkEdit(hi,t,e,u,n) = ⌈⌉{hdlr_docu_ch[hi,di]?|hi:HI•hi∈his} in
916a  [2] let dc′ = e(dc),
916b        da′ = augment(da,((hi,t),(″edit″,e,u),n)),
916c        dh′ = augment(dh,((hi,t),(″edit″,e,u))) in
917       let ti = time_ch? − t in
918       hdlr_docu_ch[hi,di] ! mkEditComplete(ti,...) ;
919       docu_di(dd)(da′,dc′,dh′)
915a      end end end
```

9.16.4.2 The Document Read Behaviour: Left Fig. 9.4 on page 196

920. The The **document [read] behaviour** offers to receive a read request from a handler behaviour.
921. The reading takes some time to do.
922. The handler behaviour is advised on completion.
923. And the document behaviour is resumed with appropriate programmable attributes being updated.

```
value
915b  docu_read_di: DD × (DA × DC × DH) → in,out {hdlr_docu_ch[hi,di]|hi:HI•hi∈his} Unit
915b  docu_read_di(dd)(da,dc,dh) ≡
920   [2] let mkRead(hi,t,n) = {hdlr_docu_ch[hi,di]?|hi:HI•hi∈his} in
921   [2] let ti:TIME_INTERVAL • ... in
921   [2] wait ti ;
922   [2] hdlr_docu_ch[hi,di] ! mkReadComplete(ti,...) ;
923   [2] docu_di(dd)(augment(da,n),dc,augment(dh,(hi,t,ti,″read″)))
915b      end end
```

9.16.4.3 The Document Shred Behaviour: Right Fig. 9.5 on page 197

924. The **document [shred] behaviour** offers to accept a document shred request from the archive behaviour –
925. whereupon it **stop**s!

```
value
915c  docu_shred_di: DD × (DA × DC × DH) → in,out arch_docu_ch[di] Unit
915c  docu_shred_di(dd)(da,dc,dh) ≡
924   [3] let mkShred(...) = arch_docu_ch[di] ? in
925       stop
915c  [3] end
```

9.16.5 Conclusion

This completes a first draft version of this document. The date time is: November 15, 2021: 16:12 . Many things need to be done. First a careful checking of all types and functions: that all used names have been defined. The internal non-deterministic choices in formula Items 840 on page 198, 876 on page 201, 892 on page 203 and 915 on page 205, need be checked. I suspect there should, instead, be some mix of both internal and external non-deterministic choices. Then a careful motivation for all the other non-deterministic choices.

9.17 Documents in Public Government

Public government, in the spirit of *Charles-Louis de Secondat, Baron de La Brède et de Montesquieu* (or just *Montesquieu*), has three branches:

- the **legislative**,
- the **executive**, and
- the **judicial**.

Our interpretation of these, with respect to documents, are as follows.

- The **legislative** branch produces laws, i.e., documents. To do so many preparatory documents are created, edited, read, copied, etc. Committees, subcommittees, individual lawmakers and ministry law office staff handles these documents. Parliament staff and legislators are granted limited or unlimited access rights to these documents. Finally laws are put into effect, are amended, changed or abolished.
  The legislative branch documents refer to legislative, executive and judicial branch documents.
- The **executive** branch produces guide lines, i.e., documents. Instructions on interpretation and implementation of laws; directives to ministry services on how to handle the laws; et cetera.
  These executive branch documents refer to legislative, executive and judicial branch documents.
- The **judicial** branch produces documents. Police cite citizens and enterprises for breach of law. Citizens and enterprise sue other citizens and/or enterprises. Attorneys on behalf of the governments, or citizens or enterprises prepare statements. Court proceedings are recorded. Justices pass verdicts.
  The judicial branch documents refer to legislative, executive and judicial branch documents.

9.18 Documents in Urban Planning

A separate research note [71, Urban Planning Processes] analyses & describes a domain of urban planning. There are the geographical documents:

- geodetic,
- geotechnic,
- meteorological,
- and other types of geographical documents.

In order to perform an informed urban planning further documents are needed:

- auxiliary documents which
- requirements documents which

Auxiliary documents presents such information that "fill in" details concerning current ownership of the land area, current laws affecting this ownership, the use of the land, et cetera. Requirements documents express expectations about the (base) urban plans that should result from the base urban planning. As a first result of base urban planning we see the emergence of the following kinds of documents:

- base urban plans
- and ancillary notes.

The base urban plans deal with

- cadestral,
- cartographic and
- zoning

issues. The ancillary notes deal with such things as insufficiencies in the base plans, things that ought be improved in a next iteration base urban planning, etc. The base plans and ancillary notes, besides possible re-iteration of base urban planning, lead on to "derived urban planning" for

- light, medium and heavy industry zones,
- mixed shopping and residential zones,
- apartment building zones,
- villa zones,
- recreational zones,
- et cetera.

After these "first generation" derived urban plans are well underway, a "second generation" derived urban planning can start:

- transport infrastructure,
- water and waste resource management,
- electricity, natural gas, etc., infrastructure,
- et cetera.

And so forth. Literally "zillions upon zillions" of strongly and crucially interrelated documents accrue.

Urban planning evolves and revolves around documents.

Documents are the only "tangible" results or urban planning.[67]

[67] Once urban plans have been agreed upon by all relevant authorities and individuals, then urban development ("build") and, finally, "operation" of the developed, new urban "landscape". For development, the urban plans form one of the "tangible" inputs. Others are of financial and human and other resource nature.

Chapter 10

Urban Planning [Fall 2017]

Contents

10.1 Structures and Parts 212
10.1.1 The Urban Space, Clock, Analysis & Planning Complex 212
10.1.2 The Analyser Structure and Named Analysers 212
10.1.3 The Planner Structure 213
10.1.4 Atomic Parts 213
10.1.5 Preview of Structures and Parts 214
10.1.6 Planner Names 214
10.1.7 Individual and Sets of Atomic Parts 215
10.2 Unique Identifiers 215
10.2.1 Urban Space Unique Identifier 216
10.2.2 Analyser Unique Identifiers 216
10.2.3 Master Planner Server Unique Identifier 216
10.2.4 Master Planner Unique Identifier 216
10.2.5 Derived Planner Server Unique Identifier 217
10.2.6 Derived Planner Unique Identifier 217
10.2.7 Derived Plan Index Generator Identifier 217
10.2.8 Plan Repository 217
10.2.9 Uniqueness of Identifiers 218
10.2.10 Indices and Index Sets 218
10.2.11 Retrieval of Parts from their Identifiers 219
10.2.12 A Bijection: Derived Planner Names and Derived Planner Identifiers 219
10.3 Mereologies 220
10.3.1 Clock Mereology 220
10.3.2 Urban Space Mereology 221
10.3.3 Analyser Mereology 221
10.3.4 Analysis Depository Mereology 221
10.3.5 Master Planner Server Mereology 222
10.3.6 Master Planner Mereology 222
10.3.7 Derived Planner Server Mereology 222
10.3.8 Derived Planner Mereology 223
10.3.9 Derived Planner Index Generator Mereology 223
10.3.10 Plan Repository Mereology 223
10.4 Attributes 224
10.4.1 Clock Attribute 224
10.4.1.1 Time and Time Intervals and their Arithmetic 224
10.4.1.2 The Attribute 224
10.4.2 Urban Space Attributes 225
10.4.2.1 The Urban Space 225
10.4.2.2 The Urban Space Attributes 225
10.4.2.2.1 Main Part and Attributes 226
10.4.2.2.2 Urban Space Attributes – Narratives and Formalisation 226
10.4.2.2.3 General Form of Attribute Models 226

10.4.2.2.4 Geodetic Attribute[s] ... 227
10.4.2.2.5 Cadastral Attribute[s] ... 227
10.4.2.2.6 Geotechnical Attribute[s] ... 227
10.4.2.2.7 Meteorological Attribute[s] ... 228
10.4.2.2.8 Socio-Economic Attribute[s] ... 228
10.4.2.2.9 Law Attribute[s]: State, Province, Region, City and District Ordinances ... 229
10.4.2.2.10 Industry and Business Economics ... 229
10.4.2.2.11 Etcetera ... 229
10.4.2.2.12 The Urban Space Attributes – A Summary ... 229
10.4.2.2.13 Discussion ... 229
10.4.3 Scripts ... 230
10.4.4 Urban Analysis Attributes ... 230
10.4.5 Analysis Depository Attributes ... 230
10.4.6 Master Planner Server Attributes ... 231
10.4.7 Master Planner Attributes ... 231
10.4.8 Derived Planner Server Attributes ... 232
10.4.9 Derived Planner Attributes ... 232
10.4.10 Derived Planner Index Generator Attributes ... 232
10.4.11 Plan Repository Attributes ... 233
10.4.12 A System Property of Derived Planner Identifiers ... 233
10.5 The Structure Compilers ... 234
10.5.1 A Universe of Discourse Compiler ... 234
10.5.2 The Analyser Structure Compiler ... 234
10.5.3 The Planner Structure Compiler ... 235
10.5.3.1 The Master Planner Structure Compiler ... 235
10.5.3.2 The Derived Planner Structure Compiler ... 235
10.5.3.3 The Derived Planner Pair Structure Compiler ... 235
10.6 Channel Analysis and Channel Declarations ... 236
10.6.1 The clk_ch Channel ... 236
10.6.2 The tus_a_ch Channel ... 237
10.6.3 The tus_mps_ch Channel ... 237
10.6.4 The a_ad_ch Channel ... 237
10.6.5 The ad_s_ch Channel ... 238
10.6.6 The mps_mp_ch Channel ... 238
10.6.7 The p_pr_ch Channel ... 238
10.6.8 The p_dpxg_ch Channel ... 239
10.6.9 The pr_s_ch Channel ... 239
10.6.10 The dps_dp_ch Channel ... 240
10.7 The Atomic Part Translators ... 240
10.7.1 The clock Translator ... 240
10.7.1.1 The Translate_CLK Function ... 240
10.7.1.2 The clock Behaviour ... 240
10.7.2 The Urban Space Translator ... 241
10.7.2.1 The Translate_TUS Function ... 241
10.7.2.2 The urb_spa Behaviour ... 241
10.7.3 The Analyser$_{anm_i}$, i:[1 : n] Translator ... 243
10.7.3.1 The Translate_A$_{anm_j}$ Function ... 243
10.7.3.2 The analyser$_{ui_j}$ Behaviour ... 243
10.7.4 The Analysis Depository Translator ... 244
10.7.4.1 The Translate_AD Function ... 244
10.7.4.2 The ana_dep Behaviour ... 245
10.7.5 The Derived Planner Index Generator Translator ... 245
10.7.5.1 The Translate_DPXG($dpxg$) Function ... 245
10.7.5.2 The dpxg Behaviour ... 246
10.7.6 The Plan Repository Translator ... 246
10.7.6.1 The Translate_PR Function ... 246
10.7.6.2 The plan_rep Behaviour ... 247
10.7.7 The Master Server Translator ... 247
10.7.7.1 The Translate_MPS Function ... 247
10.7.7.2 The master_server Behaviour ... 248
10.7.8 The Master Planner Translator ... 248
10.7.8.1 The Translate_MP Function ... 248

10.7.8.2 The Master urban_planning Function 249
10.7.8.3 The master_planner Behaviour 250
10.7.8.4 The initiate derived servers and derived planners Behaviour 250
10.7.9 The DERIVED SERVER$_{nm_i}$, i:[1 : p] TRANSLATOR 251
10.7.9.1 The TRANSLATE_DPS$_{nm_j}$ Function 251
10.7.9.2 The derived_server Behaviour 252
10.7.10 The DERIVED PLANNER$_{nm_i}$, i:[1 : p] TRANSLATOR 253
10.7.10.1 The TRANSLATE_DPdp_{nm_j} Function 253
10.7.10.2 The derived_urban_planning Function 253
10.7.10.3 The derived_planner$_{nm_j}$ Behaviour 254
10.8 Initialisation of The Urban Space Analysis & Planning System 255
10.8.1 Summary of Parts and Part Names 255
10.8.2 Summary of of Unique Identifiers 255
10.8.3 Summary of Channels 256
10.8.4 The Initial System 256
10.8.5 The Derived Planner System 257
10.9 Further Work 257
10.9.1 Reasoning About Deadlock, Starvation, Live-lock and Liveness 257
10.9.2 Document Handling 257
10.9.2.1 Urban Planning Documents 257
10.9.2.2 A Document Handling System 258
10.9.3 Validation and Verification (V&V) 258
10.9.4 Urban Planning Project Management 258
10.9.4.1 Urban Planning Projects 258
10.9.4.2 Strategic, Tactical and Operational Management 259
10.9.4.2.1 Project Resources 259
10.9.4.2.2 Strategic Management 259
10.9.4.2.3 Tactical Management 259
10.9.4.2.4 Operational Management 259
10.9.4.3 Urban Planning Management 260

ENDURANTS

By an *entity* we shall understand a phenomenon, i.e., something that can be observed, i.e., be seen or touched by humans, or that can be conceive d as an abstraction of an entity. We further demand that an entity can be objectively described

By an *endurant* we shall understand an entity that can be observed or conceived and described as a "complete thing" at no matter which given snapshot of time. Were we to "freeze" time we would still be able to observe the entire endurant.

By a *discrete endurant* we shall understand an endurant which is separate, individual or distinct in form or concept.

By a *part* we shall understand a discrete endurant which the domain engineer chooses to endow with *internal qualities*[68] such as *unique identification*, *mereology*, and one or more *attributes*. We shall define these three categories in Sects. **10.2**, **10.3**, respectively Sect. **10.4**. We refer in general to [48].

In this, a major section of this report, we shall cover

- Sect. **10.1**: Parts,
- Sect. **10.2**: Unique Identifiers,
- Sect. **10.3**: Mereology, and
- Sect. **10.4**: Attributes.

[68] – where by *external qualities* of an endurant we mean whether it is discrete of *continuous*, whether it is a parts, or a *component* – such as these are defined in [48].

10.1 Structures and Parts

From an epistemological[69] point of view a study of the parts of a universe of discourse is often the way to understand *"who the players"* of that domain are. From the point of view of [48] *knowledge about parts lead to knowledge about behaviours.* This is the reason, then, for our interest in parts.

10.1.1 The Urban Space, Clock, Analysis & Planning Complex

The domain-of-interest, i.e., the universe of discourse for this report is that of *the urban space analysis & planning complex* – where the ampersand, '&', shall designate that we consider this complex as 'one' !

926. The *universe of discourse*, UoD, is here seen as a *structure* of four elements:
   a. a *clock*, CLK,
   b. *the urban space*, TUS,
   c. an *analyser aggregate*, AA,
   d. the *planner aggregate*, PA,

type
926 UoD, CLK, TUS, AAG, PA

value
926a obs_CLK: UoD → CLK
926b obs_TUS: UoD → TUS
926c obs_AAG: UoD → AAG
926d obs_PA: UoD → PA

The clock and the urban space are here considered *atomic*, the *analyser aggregate*, AA, and the the *planner aggregate*, PA, are here seen as *structures*.

10.1.2 The Analyser Structure and Named Analysers

927. The *analyser structure* consists of
   a. a *structure*, AC, which consists of two elements:
      i. a *structure* of an indexed set, hence named *analysers*,
      ii. A_{anm_1}, A_{anm_2}, ..., A_{anm_n},

   and
928. an *atomic analysis depository*, AD.

   There is therefore defined a set, ANms, of

929. *analyser names*: $\{anm_1, anm_2, \ldots, anm_n\}$, where $n \geq 0$.

type
927 AA, AC, A, AD
927(a)i $A = A_{anm_1} \mid A_{anm_2} \mid \ldots \mid A_{anm_n}$

[69] Epistemology is the branch of philosophy concerned with the theory of knowledge.

929 ANms = $\{|anm_1, anm_2, ..., anm_n|\}$
value
927a obs_AC: AA → AC
927(a)ii obs_AC$_{anm_i}$: AC → A$_{anm_i}$, i:[1..n]
928 obs_AD: AA → AD

Analysers and the *analysis depository* are here seen as atomic parts.

10.1.3 The Planner Structure

930. The composite *planner structure* part, consists of
   a. a *master planner structure*, MPA, which consists of
      i. an atomic *master planner server*, MPS, and
      ii. an atomic *master planner*, MP, and
   b. a *derived planner structure*, DPA, which consists of
      i. a *structure* in the form of an indexed set of (hence named) *derived planner structures*, DPC$_{nm_j}$, j : [1..p], which each consists of
         1. a atomic *derived planner servers*, DPS$_{nm_j}$, j : [1..p], and
         2. a atomic *derived planners*, DP$_{nm_j}$, j : [1..p];
   c. an atomic *plan repository*, PR, and
   d. an atomic *derived planner index generator*, DPXG.

type
930 PA, MPA, MPS, MP, DPA, DPC$_{nm_j}$, DPS$_{nm_j}$, DP$_{nm_j}$, i:[1..p]

value
930a obs_MPA: PA → MPA
930(a)i obs_MPS: MPA → MPS
930(a)ii obs_MP: MPA → MP
930b obs_DPA: PA → DPA
930(b)i obs_DPC$_{nm_j}$: DPA → DPC$_{nm_j}$, i:[1..p]
930(b)i1 obs_DPS$_{nm_j}$: DPC$_{nm_j}$ → DPS$_{nm_j}$, i:[1..p]
930(b)i2 obs_DP$_{nm_j}$: DPC$_{nm_j}$ → DP$_{nm_j}$, i:[1..p]
930c obs_PR: PA → PR
930d obs_DPXG: → DPXG

We have chosen to model as *structures* what could have been modeled as *composite* parts. If we were to domain analyse & describe *management & organisation* facets of the urban space analysis & planning domain then we might have chosen to model some of these *structures* instead as *composite parts.*

10.1.4 Atomic Parts

The following are seen as atomic parts:

- *clock,*
- *urban space,*
- *analysis deposit,*
- each *analyser* in the indexed set of *analyser*$_{anm_i}$s,
- *master planner server,*
- *master planner,*
- each *server* in the indexed set of *derived planner server*$_{nm_j}$s,
- each *planner* in the indexed set of *derived planner*$_{nm_j}$s,
- *derived planner index generator.*
- *plan repository* and

We shall return to the these atomic part sorts when we explore their properties: *unique identifiers, mereologies* and *attributes*.

10.1.5 Preview of Structures and Parts

Let us take a preview of the parts, see Fig. 10.1.

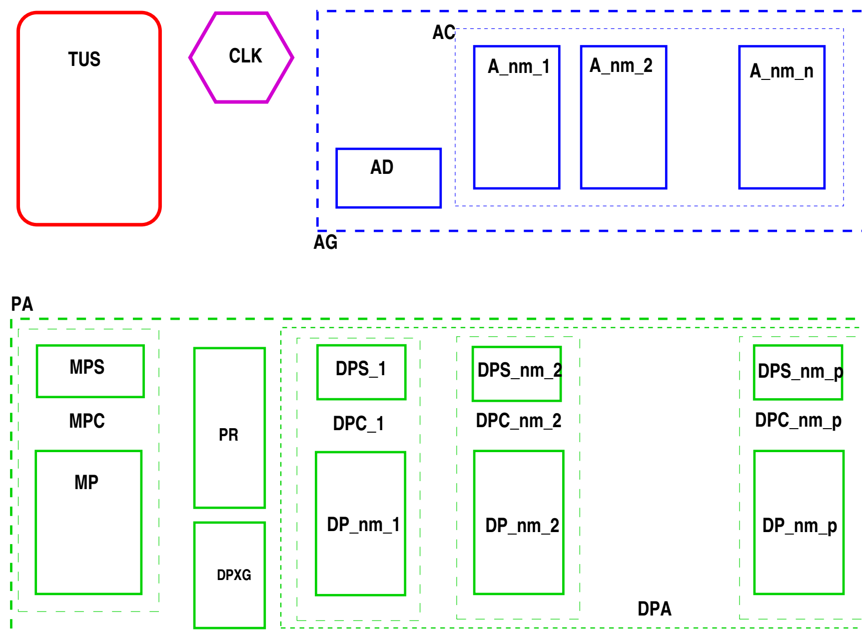


Fig. 10.1 The Urban Analysis and Planning System: **Structures and Atomic Parts**

10.1.6 Planner Names

931. There is therefore defined identical sets of *derived planner aggregate names, derived planner server names,* and *derived planner names*: $\{dnm_1, dnm_2, \ldots, dnm_p\}$, where $g \geq 0$.

type
931 DNms = $\{|dnm_1, dnm_2, ..., dnm_p|\}$

10.1.7 Individual and Sets of Atomic Parts

In this closing section of Sect. **10.1.7** we shall identify individual and sets of atomic parts.

932. We postulate an arbitrary *universe of discourse*, uod:UoD and let that be a constant value from which we the calculate a number of individual and sets of atomic parts.
933. There is the *clock*, *clk*:CLK,
934. *the urban space*, *tus*:TUS,
935. the set of *analysers*, a_{anm_i}:A_{anm_i}, *i*:[1..*n*],
936. the *analysis depository*, *ad*,
937. the *master planner server*, *mps*:MPS,
938. the *master planner*, *mp*:MP,
939. the set of *derived plannner servers*, {dps_{nm_i}:DPS_{nm_i} | *i*:[1..*p*]},
940. the set of *derived planners*, {dp_{nm_i}:DP_{nm_i}| *i*:[1..*p*]},
941. the *derived plan index generator*, *dpxg*,
942. the *plan repository*, *pr*, and
943. the set of pairs of *derived server* and *derived planner*s, *sps*.

```
value
932 uod : UoD
933 clk : CLK = obs_CLK(uod)
934 tus : TUS = obs_TUS(uod)
935 ans : A_anm_i-set, i:[1..n] =
935         { obs_A_anm_i(aa) | aa∈(obs_AA(uod)), i:[1..n]}
936 ad : AD = obs_AD(obs_AA(uod))
937 mps : MPS = obs_MPS(obs_MPA(uod))
938 mp : MP = obs_MP(obs_MPA(uod))
939 dpss : DPS_nm_i-set, i:[1..p] =
939         { obs_DPS_nm_i(dpc_nm_i) |
939               dpc_nm_i:DPC_nm_i•dpc_nm_i∈obs_DPCS_nm_i(obs_DPA(uod)), i:[1..p] }
940 dps : DP_nm_i-set, i:[1..p] =
940         { obs_DP_nm_i(dpc_nm_i) |
940               dpc_nm_i:DPC_nm_i•dpc_nm_i∈obs_DPCS_nm_i(obs_DPA(uod)), i:[1..p] }
941 dpxg : DPXG = obs_DPXG(uod)
942 pr : PR = obs_PR(uod)
943 spsps : (DPS_nm_i×DP_nm_i)-set, i:[1..p] =
943         { (obs_DPS_nm_i(dpc_nm_i),obs_DP_nm_i(dpc_nm_i)) |
943               dpc_nm_i:DPC_nm_i•dpc_nm_i∈ obs_DPCS_nm_i(obs_DPA(uod)), i:[1..p] }
```

10.2 Unique Identifiers

We introduce a notion of unique identification of parts. We assume (i) that all parts, *p*, of any domain *P*, have unique identifiers, (ii) that unique identifiers (of parts *p*:*P*) are abstract values (of the unique identifier, π, sort *Π_UI* of parts *p*:*P*), (iii) such that distinct part sorts, P_i and P_j, have distinctly named unique identifier sorts, say Π_UI_i and Π_UI_j, (iv) that all π:Π_UI_i and π_j:Π_UI_j are distinct, and (v) that the observer function uid_*P* applied to *p* yields the unique identifier, say π:*Π_UI*, of *p*.

The analysis & description of unique identification is a prerequisite for talking about mereologies of universes of discourse, and the analysis & description of mereologies are a means for understanding how parts relate to one another.

Since we model as *structures* what elsewhere might have been modeled as *composite parts* we shall only deal with *unique identifiers* of *atomic parts*.

10.2.1 Urban Space Unique Identifier

944. The urban space has a unique identifier.

type
944 TUS_UI
value
944 uid_TSU: TSU → TUS_UI

10.2.2 Analyser Unique Identifiers

945. Each analyser has a unique identifier.
946. The analysis depository has a unique identifier.

type
945 A_UI = A_UI$_{anm_1}$ | A_UI$_{anm_2}$ | ... | A_UI$_{anm_n}$
946 AD_UI
value
945 uid_A: A$_{nm_i}$ → A_UI$_{nm_i}$, $i:[1..n]$
946 uid_AD: AD → AD_UI
axiom
945 ∀ a$_{nm_i}$:A$_{nm_i}$ •
945 **let** a_ui$_{nm_i}$ = uid_A(a$_{nm_i}$) **in** a_ui$_{nm_i}$ ≃ nm_i **end**

The mathematical symbol ≃ (in this report) denotes *isomorphy*.

10.2.3 Master Planner Server Unique Identifier

947. The *unique identifier* of the *master planner server*.

type
947 MPS_UI
value
947 uid_MPS: MPS → MPS_UI

10.2.4 Master Planner Unique Identifier

948. The *unique identifier* of the *master planner*.

type
948 MP_UI

value
948 uid_MP: MP → MP_UI

10.2.5 Derived Planner Server Unique Identifier

949. The *unique identifiers* of *derived planner servers*.

type
949 DPS_UI = DPS_UI$_{nm_1}$ | DPS_UI$_{nm_2}$ | ... | DPS_UI$_{nm_p}$
value
949 uid_DPS: DPS$_{nm_i}$ → DPS_UI$_{nm_i}$, $i:[1..p]$
axiom
949 ∀ dps$_{nm_i}$:DPS$_{nm_i}$ •
949 **let** dps_ui$_{nm_i}$ = uid_DPS(dps$_{nm_i}$) **in** dps_ui$_{nm_i}$ $\simeq nm_i$ **end**

10.2.6 Derived Planner Unique Identifier

950. The *unique identifiers* of *derived planners*.

type
950 DP_UI = DP_UI$_{nm_1}$ | DP_UI$_{nm_2}$ | ... | DP_UI$_{nm_p}$
value
950 uid_DP: DP$_{nm_i}$ → DP_UI$_{nm_i}$, $i:[1..p]$
axiom
950 ∀ dp$_{nm_i}$:DP$_{nm_i}$ •
950 **let** dp_ui$_{nm_i}$ = uid_DP(dp$_{nm_i}$) **in** dp_ui$_{nm_i}$ $\simeq nm_i$ **end**

10.2.7 Derived Plan Index Generator Identifier

951. The *unique identifier* of *derived plan index generator*:

type
951 DPXG_UI
value
951 uid_DPXG: DPXG → DPXG_UI

10.2.8 Plan Repository

952. The *unique identifier* of *plan repository*:

type
952 PR_UI

value
952 uid_PR: PR → PR_UI

10.2.9 Uniqueness of Identifiers

953. The identifiers of all analysers are distinct.
954. The identifiers of all derived planner servers are distinct.
955. The identifiers of all derived planners are distinct.
956. The identifiers of all other atomic parts are distinct.
957. And the identifiers of all atomic parts are distinct.

953 $\mathbf{card}\, ans = \mathbf{card}\, a_{ui}s$
954 $\mathbf{card}\, dpss = \mathbf{card}\, dps_{ui}s$
955 $\mathbf{card}\, dps = \mathbf{card}\, dp_{ui}s$
956 $\mathbf{card}\{ clk_{ui}, tus_{ui}, ad_{ui}, mps_{ui}, mp_{ui}, dpxg_{ui}, plas_{ui}\} = 7$
957 $\cap(ans, dpss, dps, \{clk_{ui}, tus_{ui}, ad_{ui}, mps_{ui}, mp_{ui}, dpxg_{ui}, plas_{ui}\}) = \{\}$

10.2.10 Indices and Index Sets

It will turn out to be convenient, in the following, to introduce a number of index sets.

958. There is the *clock* identifier, clk_{ui}:CLK_UI.
959. There is *the urban space* identifier, tus_{ui}:TUS_UI.
960. There is the set, $a_{ui}s$:A_UI-**set**, of the identifiers of all *analysers*.
961. The *analysis depository* identifier, ad_{ui}.
962. There is the *master planner server* identifier, mps_{ui}:MPS_UI.
963. There is the *master planner* identifier, mp_{ui}:MP_UI.
964. There is the set, $dps_{ui}s$:DPS_UI-**set**, of the identifiers of all *derived planner servers*.
965. There is the set, $dp_{ui}s$:DP_UI-**set**, of the identifiers of all derived planners.
966. There is the *derived plan index generator* identifier, $dpxg_{ui}$:DPXG_UI.
967. And there is the *plan repository* identifier, pr_{ui}:PR_UI.

value
958 clk_{ui} : CLK_UI = uid_CLK(uod)
959 tus_{ui} : TUS_UI = uid_TUS(uod)
960 $a_{ui}s$: A_UI-**set** = {uid_A(a)|a:A•a ∈ ans}
961 ad_{ui} : AD_UI = uid_AD(ad)
962 mps_{ui} : MPS_UI = uid_MPS(mps)
963 mp_{ui} : MP_UI = uid_MP(mp)
964 $dps_{ui}s$: DPS_UI-**set** = {uid_DPS(dps)|dps:DPS•dps ∈ $dpss$}
965 $dp_{ui}s$: DP_UI-**set** = {uid_DP(dp)|dp:DP•dp ∈ dps}
966 $dpxg_{ui}$: DPXG_UI = uid_DPXG($dpxg$)
967 pr_{ui} : PR_UI = uid_PR(pr)

968. There is also the set of identifiers for all servers: $ps_{ui}s$:(MPS_UI|DPS_UI)-**set**,
969. there is then the set of identifiers for all planners: $ps_{ui}s$:(MP_UI|DP_UI)-**set**,
970. there is finally the set of pairs of paired *derived planner server* and *derived planner* identifiers.

971. there is a map from the unique derived server identifiers to their "paired" unique derived planner identifiers, and
972. there is finally the reverse map from planner to server identifiers.

value
968 $s_{ui}s$: (MPS_UI|DPS_UI)-**set** = $\{mps_{ui}\} \cup dps_{ui}s$
969 $p_{ui}s$: (MP_UI|DP_UI)-**set** = $\{mp_{ui}\} \cup dp_{ui}s$
970 *sips* : (DPS_UI×DP_UI)-**set** = {(uid_DPS(dps),uid_DP(dp))|(dps,dp):(DPS×DP)•(dps,dp)∈*sps*}
971 *si_pi_m* : DPS_UI $\overrightarrow{m}$ DP_UI = [uid_DPS(dps)↦uid_DP(dp)|(dps,dp):(DPS×DP)•(dps,dp)∈*sps*]
972 *pi_si_m* : DP_UI $\overrightarrow{m}$ DPS_UI = [uid_DP(dp)↦uid_DPS(dps)|(dps,dp):(DPS×DP)•(dps,dp)∈*sps*]

10.2.11 Retrieval of Parts from their Identifiers

973. Given the global set *dpss*, cf. 939 on page 215, i.e., the set of all derived servers, and given a unique planner server identifier, we can calculate the derived server with that identifier.
974. Given the global set *dps*, cf. 940 on page 215, the set of all derived planners, and given a unique derived planner identifier, we can calculate the derived planner with that identifier.

value
973 c_s: *dpss* → DPS_UI → DPS
973 c_s(*dpss*)(dps_ui) ≡ **let** dps:DPS•dps ∈*dpss*∧uid_DPS(dps)=dps_ui **in** dps **end**
974 c_p: *dps* → DP_UI → DP
974 c_p(*dps*)(dp_ui) ≡ **let** dp:DP•dp ∈*dps*∧uid_DPS(dp)=dp_ui **in** dp **end**

10.2.12 A Bijection: Derived Planner Names and Derived Planner Identifiers

We can postulate a unique relation between the names, dn:DNm-**set**, i.e., the names dn∈DNms, and the unique identifiers of the named planners:

975. We can claim that there is a function, extr_DNm, from the unique identifiers of derived planner servers to the names of these unique identifiers.
976. Similarly can claim that there is a function, extr_DNm, from the unique identifiers of derived planners to the names of these unique identifiers.

value
975 extr_Nm: DPS_UI → DNm
975 extr_Nm(dps_ui) ≡ ...
976 extr_Nm: DP_UI → DNm
976 extr_Nm(dp_ui) ≡ ...
axiom
975 ∀ dps_ui1,dps_ui2:DPS_ui • dps_ui1≠dps_ui2 ⇒ extr_Nm(dps_ui1) ≠ extr_Nm(dps_ui1)
976 ∀ dp_ui1,dp_ui2:DP_ui • dp_ui1≠dp_ui2 ⇒ extr_Nm(dp_ui1) ≠ extr_Nm(dp_ui1)

977. Let dps_ui_dnm:DPS_UI_DNm, dp_ui_dnm:DP_UI_DNm stand for maps from derived planner server, respectively derived planner unique identifiers to derived planner names.
978. Let nm_dp_ui:Nm_DP_UI, nm_dp_ui:Nm_DP_UI stand for the reverse maps.
979. These maps are bijections.

```
type
977  DPS_UI_DNm:  DPS_UI ⇸ DP_Nm
977  DP_UI_DNm:   DP_UI ⇸ DP_Nm
978  DNm_DPS_UI:  DP_Nm ⇸ DP_UI
978  DNm_DP_UI:   DP_Nm ⇸ DP_UI
axiom
979  ∀ dps_ui_dnm:DPS_UI_DNm • dps_ui_dnm⁻¹·dps_ui_dnnm = λx.x
979  ∀ dp_ui_dnm:DP_UI_DNm • dp_ui_dnm⁻¹·dp_ui_dnnm = λx.x
979  ∀ dnm_dps_ui:DNm_DPS_UI • dnm_dps_ui⁻¹·dnm_dps_ui = λx.x
979  ∀ dnm_dp_ui:DNm_DP_UI • dp_ui_dnm⁻¹·dnm_dps_ui = λx.x
      that is:
979  ∀ dps_ui_dnm:DPS_UI_DNm, dp_ui_dnm:DP_UI_DNm, dps_ui:DPS_UI •
979         dps_ui ∈ dom dps_ui_dnm ⇒ dp_ui_dnm(dps_ui_dnm(dps_ui)) = dps_ui
      et cetera !
```

980. The function mk_DNm_DUI takes the set of all derived planner servers, respectively derived planners and produces bijective maps, dnm_dps_ui, respectively dnm_dp_ui.
981. Let *dnm_dps_ui*:DNm_DPS_UI and
982. *dnm_dp_ui*:DNm_DP_UI

stand for such [global] maps.

```
value
980  mk_Nm_DPS_UI: DPS_nm_i-set → DNm_DPS_UI
980  mk_Nm_DPS_UI(dpss) ≡ [uid_DPS(dps)↦extr_Nm(uid_DPS(dps))|dps:DPS•dps ∈ dpss]
980  mk_Nm_DP_UI: DP_nm_i-set → DNm_DP_UI
980  mk_Nm_DP_UI(dps) ≡ [uid_DP(dp)↦extr_Nm(uid_DP(dp))|dp:DP•dps ∈ dps]

981  nm_dps_ui:Nm_DPS_UI = mk_Nm_DPS_UI(dps)
982  nm_dp_ui:Nm_DP_UI = mk_Nm_DP_UI(dps)
```

10.3 Mereologies

Mereology (from the Greek $\mu\epsilon\rho o\varsigma$ 'part') is the *theory of part-hood relations:* of *the relations of part to whole* and *the relations of part to part within a whole*[70].

Part mereologies inform of how parts relate to other parts. As we shall see in the section on *perdurants*, mereologies are the basis for analysing & describing communicating between part behaviours.

Again: since we model as *structures* what is elsewhere modeled as *composite parts* we shall only consider *mereologies* of *atomic parts*.

10.3.1 Clock Mereology

983. The clock is related to all those parts that create information, i.e., documents of interest to other parts. Time is then used to *time-stamp* those documents. These other parts are: the *urban space*, the *analysers*, the *planner servers* and the *planners*.

[70] Achille Varzi: Mereology, http://plato.stanford.edu/entries/mereology/ 2009 and [75].

type
983 CLK_Mer = TSU_UI×A_UI-**set**×MPS_UI×MP_UI×DPS_UI-**set**×DP_UI-**set**
value
983 mereo_CLK: CLK → Clk_Mer
axiom
983 mereo_CLK(uod) = (tus_{ui},$a_{ui}s$,mps_{ui},mp_{ui},$dps_{ui}s$,$dp_{ui}s$)

10.3.2 Urban Space Mereology

The urban space stands in relation to those parts which consume urban space information: the *clock* (in order to time stamp urban space information), the *analyser*s and the *master planner server*.

984. The mereology of the urban space is a triple of the clock identifier, the identifier of the master planner server and the set of all analyser identifiers. all of which are provided with urban space information.
985. The constraint here is expressed in the 'the': for the universe of discourse it must be the master planner aggregate unique identifier and the set of exactly all the analyser unique identifiers for that universe.

type
984 TUS_Mer = CLK_UI × A_UI-**set** × MPS_UI
value
984 mereo_TUS: TUS → TUS_Mer
axiom
985 mereo_TUS(tus) = (clk_{ui},$a_{ui}s$,mps_{ui})

10.3.3 Analyser Mereology

986. The mereology of a[ny] *analyser* is that of a triple: the *clock identifier, the urban space identifier*, and the *analysis depository identifier*.

type
986 A_Mer = CLK_UI × TUS_UI × AD_UI
value
986 mereo_A: A → A_Mer

10.3.4 Analysis Depository Mereology

987. The mereology of the *analysis depository* is a triple: the *clock identifier*, the *master planner server identifier*, and the set of *derived planner server identifier*s.

type
987 AD_Mer = CLK_UI × MPS_UI × DPS_UI-**set**
value
987 mereo_AD: AD → AD_Mer

10.3.5 Master Planner Server Mereology

988. The *master planner server* mereology is a quadruplet of the clock identifier (time is used to time stamp input arguments, prepared by the server, to the planner), the urban space identifier, the analysis depository and the master planner identifier.
989. And for all universes of discourse these must be exactly those of that universe.

type
988 MPS_Mer = CLK_UI × TUS_UI × AD_UI × MP_UI
value
988 mereo_MPS: MPS → MPS_Mer
axiom
989 mereo_MPS(mps) = (clk_{ui},tus_{ui},ad_{ui},mp_{ui})

10.3.6 Master Planner Mereology

990. The mereology of the *master planner* is a triple of: the clock identifier[71], master server identifier[72], derived planner index generator identifier[73], and the plan repository identifier[74].

type
990 MP_Mer = CLK_UI × MPS_UI × DPXG_UI × PR_UI
value
990 mereo_MP: MP → MP_Mer
axiom
990 mereo_MP(mp) = (clk_{ui},mps_{ui},$dpxg_{ui}$,pr_{ui})

10.3.7 Derived Planner Server Mereology

991. The *derived planner server* mereology is a quadruplet of:

the clock identifier[75], the set of all analyser identifiers[76], the plan repository identifier,[77] and the derived planner identifier[78].

type
991 DPS_Mer = CLK_UI × AD_UI × PLAS_UI × DP_UI
value

[71] From the clock the planners obtain the time with which they stamp all information assembled by the plannner.
[72] from which the master planner obtains essential input arguments
[73] in collaboration with which the master planner obtains a possibly empty set of derived planning indices
[74] with which it posits and from which it obtains summaries of all urban planning plans produced so far.
[75] From the clock the servers obtain the time with which they stamp all information assembled by the servers.
[76] From the analysers the servers obtain analyses.
[77] In collaboration with the plan repository the planners deposit plans etc. and obtains summaries of all urban planning plans produced so far
[78] The server provides its associated planner with appropriate input arguments.

```
991   mereo_DPS: DPS → DPS_Mer
axiom
991   ∀ (dps,dp):(DPS×DP) • (dps,dp)∈sps ⇒
991       mereo_DPS(dps) = (clk_ui,ad_ui,plas_ui,uid_DP(dp))
```

10.3.8 Derived Planner Mereology

992. The *derived planner* mereology is a quadruplet of:

the clock identifier, the derived plan server identifier, the derived plan index generator identifier, and the plan repository identifier.

```
type
992   DP_Mer = CLK_UI × DPS_UI × DPXG_UI × PR_UI
value
992   mereo_DP: DP → DP_Mer
axiom
992   ∀ (dps,dp):(DPS×DP) • (dps,dp)∈sps ⇒
992       mereo_DP(dp) = (clk_ui,uid_DPS(dps),dpxg_ui,pr_ui)
```

10.3.9 Derived Planner Index Generator Mereology

993. The mereology of the derived planner index generator is the set of all planner identifiers: master and derived.

```
type
993   DPXG_Mer = (MP_UI|DP_UI)-set
value
993   mereo_DPXG: DPXG → DPXG_Mer
axiom
993   mereo_DPXG(dpxg) = ps_uis
```

10.3.10 Plan Repository Mereology

994. The plan repository mereology is the set of all planner identifiers: master and derived.

```
994   PR_Mer = (MP_UI|DP_UI)-set
value
994   mereo_PR: PR → PR_Mer
axiom
994   mereo_PR(pr) = ps_uis
```

10.4 Attributes

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

10.4.1 Clock Attribute

10.4.1.1 Time and Time Intervals and their Arithmetic

995. Time is modeled as a continuous entity.
996. One can subtract two times and obtain a time interval.
997. There is an "infinitesimally" smallest time interval, δt:T.
998. Time intervals are likewise modeled as continuous entities.
999. One can add or subtract a time interval to, resp. from a time and obtain a time.
1000. One can compare two times, or two time intervals.
1001. One can add and subtract time intervals.
1002. One can multiply time intervals with real numbers.

```
type
995  T
996  TI
value
996  sub: T × T → TI
997  δt:TI
999  add,sub: TI × T → T
1000  <,≤,=,≥,>: ((T×T)|(TI×TI)) → Bool
1001  add,sub: TI × TI → TI
1002  mpy: TI × Real → TI
```

10.4.1.2 The Attribute

1003. The only attribute of a clock is time. It is a programmable attribute.

```
type
1003  T
value
1003  attr_T: CLK → T
axiom
1003  ∀ clk:CLK •
1003     let (t,t′) = (attr_CLK(clk);attr_CLK(clk)) in
1003     t≤t′ end
```

The ‘;’ in an expression (a;b) shall mean that first expression a is evaluated, then expression b.

10.4.2 Urban Space Attributes

10.4.2.1 The Urban Space

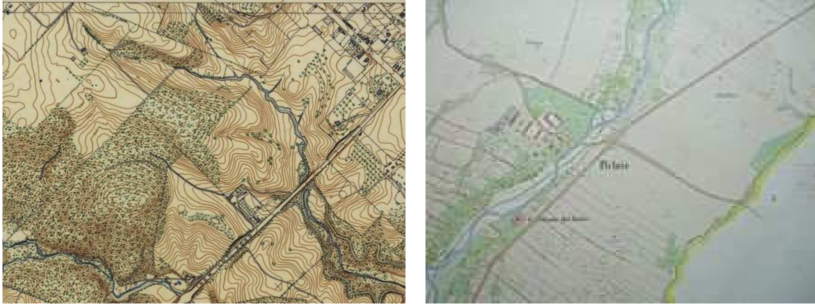
1004. We shall assume a notion of *the urban space*, tus:TUS, from which we can observe the attribute:
1005. an infinite, compact Euclidean set of points.
1006. By a *point* we shall understand a further undefined atomic notion.
1007. By an *area* we shall understand a concept, related to the urban space, that allows us to speak of *"a point being in an area"* and *"an area being equal to or properly within another area"*.
1008. To an[y] *urban space* we can associate an area; we may think of an area being an *attribute* of the urban space.

```
type
1004  TUS
1005  PtS = Pt-infset
value
1004  attr_PtS: TUS → Pt-infset
type
1006  Pt
1007  Area
value
1008  attr_Area: TUS → Area
1007  is_Pt_in_Area: Pt × (TUS|Area) → Bool
1007  is_Area_within_Area: Area × (TUS|Area) → Bool
```

10.4.2.2 The Urban Space Attributes

By *urban space attributes* we shall here mean the facts by means of which we can characterize that which is subject to urban planning: the land, what is in and on it: its geodetics, its cadastra[79], its meteorology, its socio-economics, its rule of law, etc. As such we shall consider ‘the urban space’ to be a *part* in the sense of [48]. And we shall consider the *geodetic, cadastral, geotechnical, meteorological, "the law"* (i.e., *state, province, city* and *district ordinances*) and *socio-economic* properties as *attributes*.

[79] Cadastra: A Cadastra is normally a parcel based, and up-to-date land information system containing a record of interests in land (e.g. rights, restrictions and responsibilities). It usually includes a geometric description of land parcels linked to other records describing the nature of the interests, the ownership or control of those interests, and often the value of the parcel and its improvements. See http://www.fig.net/



Left: geodetic map, right: cadastral map.

10.4.2.2.1 Main Part and Attributes

One way of observing *the urban space* is presented: to the left, in the framed box, we ***narrate*** the story; to the right, in the framed box, we ***formalise*** it.

1009. The Urban Space (TUS) has the following
   a. PointSpace attributes,
   b. Geodetic attributes,
   c. Cadastre attributes,
   d. Geotechnical attributes,
   e. Meteorological attributes,
   f. Law attributes,
   g. Socio-Economic attributes, etcetera.

type
1009 TUS, PtS, GeoD, Cada, GeoT, Met, Law, SocEco, ...

value
1009a attr_Pts: TUS → PtS
1009b attr_GeoD: TUS → GeoD
1009c attr_Cada: TUS → Cada
1009d attr_GeoT: TUS → GeoT
1009e attr_Met: TUS → Met
1009f attr_Law: TUS → Law
1009g attr_SocEco: TUS → SocEco

The attr_A: P → A is the **signature** of a postulated *attribute (observer) function.* From parts of type P it **observes** attributes of type A. attr_A are postulated functions. They express that we can always observe attributes of type A of parts of type P.

10.4.2.2.2 Urban Space Attributes – Narratives and Formalisation

We describe attributes of the domain of urban spaces. As they are, in real life. Not as we may record them or represent them (on paper or within the computer). We can "freely" model that reality as we think it is. If we can talk about and describe it, then it is so! For meteorological attributes it means that we describe precipitation, evaporation, humidity and atmospheric pressure as these physical phenomena "really" are: continuous over time! Similar for all other attributes. Etcetera.

10.4.2.2.3 General Form of Attribute Models

1010. We choose to model the *General Form of Attributes*, such as geodetical, cadastral, geotechnical, meteorological, socio-economic, legal, etcetera, as [continuous] functions from time to maps from points or areas to the specific properties of the attributes.
1011. The points or areas of the properties maps must be in, respectively within, the area of the urban space whose attributes are being specified.

type
1010 GFA = T → ((Pt|Area) $\overrightarrow{m}$ Properties)
value
1011 wf_GFA: GFA × TUS → **Bool**

```
1011  wf_GFA(gfa,tus) ≡
1011    let area = attr_Area(tus) in
1011    ∀ t:T • t ∈ 𝒟 gfa ⇒
1011      ∀ pt:Pt • pt ∈ dom gfa(t) ⇒ is_Pt_in_Area(pt,area)
1011     ∧ ∀ ar:Area • ar ∈ dom gfa(t) ⇒ is_within_Area(ar,area)
1011    end
```

$\mathcal{D}$ is a hypothesized function which applies to continuous functions and yield their domain !

10.4.2.2.4 Geodetic Attribute[s]

1012. Geodetic attributes map points to
   a. land elevation and what kind of land it is; and (or) to
   b. normal and current water depths and what kind of water it is.
1013. Geodetic attributes also includes road nets and what kind of roads;
1014. etcetera,

```
type
1012   GeoD = T → (Pt →m ((Land|Water) × RoadNet × ...))
1012a  Land = Elevation × (Farmland|Urban|Forest|Wilderness|Meadow|Swamp|...)
1012b  Water = (NormDepth × CurrDepth) × (Spring|Creek|River|Lake|Dam|Sea|Ocean|...)
1013   RoadNet = ...
1014   ...
```

10.4.2.2.5 Cadastral Attribute[s]

A cadastre is a public register showing details of ownership of the real property in a district, including boundaries and tax assessments.

1015. Cadastral maps shows the boundaries and ownership of land parcels. Some cadastral maps show additional details, such as survey district names, unique identifying numbers for parcels, certificate of title numbers, positions of existing structures, section or lot numbers and their respective areas, adjoining and adjacent street names, selected boundary dimensions and references to prior maps.
1016. Etcetera.

```
type
1015  Cada = T → (Area →m (Owner × Value × ...))
1016  ...
```

10.4.2.2.6 Geotechnical Attribute[s]

1017. Geotechnical attributes map points to
   a. top and lower layer soil etc. composition, by depth levels,
   b. ground water occurrence, by depth levels,
   c. gas, oil occurrence, by depth levels,
   d. etcetera.

type
1017 GeoT = (Pt $\overrightarrow{m}$ Composition)
1017a Composition = VerticalScaleUnit × Composite*
1017b Composite = (Soil|GroundWater|Sand|Gravel|Rock|...|Oil|Gas|...)
1017c Soil,Sand,Gravel,Rock,...,Oil,Gas,... = [chemical analysis]
1017d ...

10.4.2.2.7 Meteorological Attribute[s]

1018. Meteorological information records, for points (of an area) precipitation, evaporation, humidity, etc.;
   a. precipitation: the amount of rain, snow, hail, etc.; that has fallen at a given place and at the time-stamped moment[80], expressed, for example, in milimeters of water;
   b. evaporation: the amount of water evaporated (to the air);
   c. atmospheric pressure;
   d. air humidity;
   e. etcetera.

1018 Met = T → (Pt $\overrightarrow{m}$ (Precip × Evap × AtmPress × Humid × ...))
1018a Precip = MMs [milimeters]
1018b Evap = MMs [milimeters]
1018c AtmPress = MB [milibar]
1018d Humid = Percent
1018e ...

10.4.2.2.8 Socio-Economic Attribute[s]

1019. Socio-economic attributes include time-stamped area sub-attributes:
   a. income distribution;
   b. housing situation, by housing category: apt., etc.;
   c. migration (into, resp. out of the area);
   d. social welfare support, by citizen category;
   e. health status, by citizen category;
   f. etcetera.

type
1019 SocEco = T → (Area $\overrightarrow{m}$ (Inc×Hou×Mig×SoWe×Heal×...))
1019a Inc = ...
1019b Hou = ...
1019c Mig = {|"in","out"|} $\overrightarrow{m}$ ({|"male","female"|} $\overrightarrow{m}$ (Agegroup × Skills × HealthSumm × ...))
1019d SoWe = ...
1019e CommHeal = ...
1019f ...

[80] – that is within a given time-unit

10.4.2.2.9 Law Attribute[s]: State, Province, Region, City and District Ordinances

1020. By the law we mean any state, province, region, city, district or other 'area' ordinance[81].
1021. ...

```
type
1020  Law
value
1020  attr_Law: TUS → Law
type
1020  Law = Area ⇸ Ordinances
1021  ...
```

10.4.2.2.10 Industry and Business Economics

TO BE WRITTEN

10.4.2.2.11 Etcetera

TO BE WRITTEN

10.4.2.2.12 The Urban Space Attributes – A Summary

Summarising we can model the aggregate of urban space attributes as follows.

1022. Each of these attributes can be given a name.
1023. And the aggregate can be modelled as a map (i.e., a function) from names to appropriately typed attribute values.

```
type
1022 TUS_Attr_Nm = {|"pts","ged","cad","get","law","eco",...|}
1023 TUSm = TUS_Attr_Nm ⇸ TUS_Attr
axiom
1023 ∀ tusm:TUSm • ∀ nm:TUS_Attr_Nm • nm ∈ dom tusm ⇒
1023     case (nm,mtusm(nm)) of
1023       (("pts",v) → is_PtS(v), "ged",v) → is_GeoD(v), ("cad",v) → is_CaDa(v),
1023        ("get",v) → is_GeoT(v), ("law",v) → is_Law(v), ("eco",v) → is_Eco(v), ...
1023     end
```

10.4.2.2.13 Discussion

TO BE WRITTEN

[81] Ordinance: a law set forth by a governmental authority; specifically a municipal regulation: for ex.: *A city ordinance forbids construction work to start before 8 a.m.*

10.4.3 Scripts

The concept of *scripts* is relevant in the context of *analysers* and *planners*.

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

Those who *contract* urban analyses and urban plannings may wish to establish that some procedural steps are taken. Examples are: the vetting of urban space information, the formulation of requirements to what the analysis must contain, the vetting of that and its "quality", the order of procedural steps, etc. We refer to [50, 57].

A[ny] *script*, as implied above, is *"like a program"*, albeit to be "computed" by humans.

Scripts may typically be expressed in some notation that may include: graphical renditions that, for example, illustrate that two or more independent groups of people, are expected to perform a number of named and more-or-less loosely described actions, expressed in, for example, the technical (i.e., domain) language of urban analysis, respectively urban planning.

The design of urban analysis and of urban planning scripts is an experimental research project with fascinating prospects for further understanding *what urban analysis* and *urban planning* is.

10.4.4 Urban Analysis Attributes

1024. Each *analyser* is characterised by a script, and
1025. the set of master and/or derived planner server identifiers – meaning that their "attached" planners might be interested in its analysis results.

type
1024 A_Script = A_Script$_{anm_1}$ | A_Script$_{anm_2}$ | ... | A_Script$_{anm_n}$
1025 A_Mer = (MPS_UI|DPS_UI)-**set**
value
1024 attr_A_Script: A → A_Scripts
1025 attr_A_Mer: A → A_Mer
axiom
1025 ∀ a:A•a ∈ ans ⇒ attr_A_Mer(a) ⊆ $ps_{ui}s$

10.4.5 Analysis Depository Attributes

The purpose of the *analysis depository* is to *accept, store* and *distribute* collections of *analyses*; it *accepts* these analysis from the analysers. it *stores* these analyses "locally"; and it *distributes* aggregates of these analyses to *plan servers*.

1026. The *analysis depository* has just one attribute, AHist. It is modeled as a map from *analyser names* to *analysis histories*.
1027. An *analysis history* is a time-ordered sequence, of time stamped analyses, most recent analyses first.

type
1026 AHist = ANm $\overrightarrow{m}$ (s_T:T × s_Anal:Anal$_{anm_i}$)*
value
1026 attr_AHist: AD → AHist

axiom
1027 ∀ ah:AHist, anm:ANm • anm ∈ **dom** ah ⇒
1027 ∀ i:**Nat** • {i,i+1}⊆**inds** ah(anm) ⇒
1027 s_T((ah(nm))[i]) > s_T((ah(nm))[i+1])

10.4.6 Master Planner Server Attributes

The *planner servers*, whether for *master planners* or *derived planners*, assemble arguments for their associated (i.e., 'paired') planners. These arguments include information *auxiliary* to other arguments, such as urban space information for the master planner, and analysis information for all planners; in addition the server also provides *requirements* that are resulting planner plans are expected to satisfy. For every iteration of the planner behaviour the pair of *auxiliary* and *requirements* information is to be renewed and the renewed pairs must somehow *"fit"* the previously issued pairs.

1028. The *programmable* attributes of the master planner server are those of aux:AUXiliaries and req:REQuirements.
1029. We postulate a predicate function, fit_mAux_mReq, which takes a pair of pairs auxiliary and requirements arguments, and yields a truth value.

type
1028 mAUX, mREQ
value
1028 attr_mAUX: MPS → mAUX
1028 attr_mREQ: MPS → mREQ
1029 fit_mAUX_mReq: (mAUX×mREQ)×(mAUX×mREQ) → **Bool**
1029 fit_mAUX_mReq(arg_prev,arg_new) ≡ ...

10.4.7 Master Planner Attributes

The *master planner* has the following attributes:

1030. a *master planner script* which is a *static attribute*;
1031. an aggregate of *script "counters"*, a *programmable attribute*; the aggregate designates *pointers* in the *master script* where resumption of *master planning* is to take place in a resumed planning;
1032. a set of *names* of the *analysers* whose analyses the master planner is, or may be interested in, a *static attribute*; and
1033. a set of *identifiers* of the *derived planners* which the master planner may initiate *static attribute*.

type
1030 MP_Script
1031 MP_Script_Pt
1031 MP_Script_Pts = MP_Script_pt-**set**
1032 ANms = ANm-**set**
1033 DPUIs = DP_UI-**set**
value
1030 attr_MP_Script: MP → MP_Script
1031 attr_Script_Pts: MP → MP_Script_Pts
1032 attr_ANms: MP → ANms
1033 attr_DPUIs: MP → DPUIs
axiom
1032 attr_ANms(*mp*) ⊆ ANms
1033 attr_DPNms(*mp*) ⊆ DNms

10.4.8 Derived Planner Server Attributes

1034. The *programmable* attributes, of the derived planner servers are those of aux:AUXiliaries and req:REQuirements, one each of an indexed set.
1035. We postulate an indexed predicate function, fit_mAux_mReq, which takes a pair of pairs auxiliary and requirements arguments, and yields a truth value.

type
1028 $\mathsf{dAUX} = \mathsf{dAUX}_{dnm_1} \mid \mathsf{dAUX}_{dnm_2} \mid \ldots \mid \mathsf{dAUX}_{dnm_p}$
1028 $\mathsf{dREQ} = \mathsf{dREQ}_{dnm_1} \mid \mathsf{dREQ}_{dnm_2} \mid \ldots \mid \mathsf{dREQ}_{dnm_p}$
value
1034 $\mathsf{attr_dAUX}_{dnm_i}\colon \mathsf{MPS}_{dnm_i} \to \mathsf{dAUX}_{dnm_i}$
1034 $\mathsf{attr_dREQ}_{dnm_i}\colon \mathsf{MPS}_{dnm_i} \to \mathsf{dREQ}_{dnm_i}$
1035 $\mathsf{fit_dAUX_dReq}_{dnm_i}\mathsf{_dReq}_{dnm_i}\colon (\mathsf{dAUX}_{dnm_i} \times \mathsf{dREQ}_{dnm_i}) \times (\mathsf{dAUX}_{dnm_i} \times \mathsf{dREQ}dnm_i) \to \mathbf{Bool}$
1035 $\mathsf{fit_dAUX_dReq}_i(\mathsf{arg_prev}_{dnm_i}, \mathsf{arg_new}_{dnm_i}) \equiv \ldots$

10.4.9 Derived Planner Attributes

1036. a *derived planner script* which is a *static attribute*;
1037. an aggregate of *script "counters"*, a *programmable attribute*; the aggregate designates *points* in the *derived planner script* where resumption of *derived planning* is to take place in a resumed planning;
1038. a set of *identifiers* of the *analysers* whose analyses the master planner is, or may be interested in, a *static attribute*; and
1039. a set of *identifiers* of the *derived planners* which any specific derived planner may initiate, a *static attribute*.

type
1036 DP_Script
1037 DP_Script_pt
1037 DP_Script_Pts = DP_Script_pt*
1038 ANms
1039 DNms
value
1036 attr_MP_Script: MP → MP_Script
1037 attr_Script_Pts: MP → Script_Pts
1038 attr_ANms: MP → ANms
1039 attr_DNms: MP → DNms
axiom
1038 attr_AUIs(mp) ⊆ ANms
1039 attr_DPUIs(mp) ⊆ DNms

10.4.10 Derived Planner Index Generator Attributes

The *derived planner index generator* has two attributes:

1040. the set of all derived planner identifiers (a static attribute), and
1041. a set of already used planner identifiers (a programmable attribute).

type
1040 All_DPUIs = DP_UI-**set**
1041 Used_DPUIs = DP_UI-**set**
value
1040 attr_All_DPUIs: DPXG →
All_DPUIs

```
1041  attr_Used_DPUIs: DPXG →
   Used_DPUIs
axiom
1040  attr_All_DPUIs(dpxg) = dp_uis
1041  attr_Used_DPUIs(dpxg) ⊆ dp_uis
```

10.4.11 Plan Repository Attributes

The rôle of the *plan repository* is to keep a record of all master and derived plans. There are two plan repository attributes.

1042. A bijective map between derived planner identifiers and names, and
1043. a pair of a list of time-stamped master plans and a map from derived planner names to lists of time-stamped plans, where the lists are sorted in time order, most recent time first.

```
type
1042  NmUIm = DNm ↔m DP_UI
1043  PLANS = ((MP_UI|DP_UI) ↔m(s_t:T×s_pla:PLA)*)
value
1042  attr_NmUIm: PR → NmUIm
axiom
1042  ∀ bm:NmUIm • bm⁻¹(bm) ≡ λx.x
value
1042  attr_PLANS: PR → PLANS
axiom
1043  let plans = attr_PLANS(pr) in
1043  dom plans ⊆ {mp_ui}∪dp_uis
1043  ∀ pui:(MP_UI|DP_UI)•pui ∈ {mp_ui}∪dp_uis ⇒ time_ordered(plans(pui))
1043  end
value
1043  time_ordered: (s_t:T×s_pla:PLA)* → Bool
1043  time_ordered(tsl) ≡ ∀ i:Nat•{i,i+1}⊆inds tsl ⇒ s_t(sl(i)) > s_t(tsl(i+1))
```

10.4.12 A System Property of Derived Planner Identifiers

Let there be given the set of derived planners *dps*.

1044. The function reachable identifiers is the one that calculates all derived planner identifiers reachable from a given such identifier, dp_ui:DP_UI, in *dps*.
   a. We calculate the derived planner, dp:DP, from dp_ui.
   b. We postulate a set of unique identifiers, uis, initialised with those that can are in the attr_DPUIs(dp) attribute.
   c. Then we recursively calculate the derived planner identifiers that can be reached from any identifier, ui, in uis.
   d. The recursion reaches a fix-point when there are no more identifiers "added" to uis in an iteration of the recursion.
1045. A derived planner must not "circularly" refer to itself.

```
value
1044   reachable_identifiers: DP-set × DP_UI → DP_UI-set
1044   (dps)(dp_ui) ≡
1044a      let dp = c_p(dps)(dp_ui) in
1044b      let uis = attr_DPUIs(dp) ∪
1044c            {ui|ui:DP_UI•ui ∈ uis ∧ ui ∈ reachable_identifiers(dps)(ui)}
1044d      in uis end end

1045   ∀ ui:DP_UI • ui ∈ dp_uis ⇒ ui ∉ names(dps)(ui)
```

The seeming "endless recursion" ends when an iteration of the dns construction and its next does not produce new names for dns — a least fix-point has been reached.

10.5 The Structure COMPILERs

10.5.1 A UNIVERSE OF DISCOURSE COMPILER

In this section, i.e., all of Sect. **10.5.1**, we omit complete typing of behaviours.

1046. The universe of discourse, *uod*, **BEHAVIOUR_SIGNATURE**$_s$ and TRANSLATEs into the of its four elements:
   a. the translation of the atomic clock, see Item **10.7.1** on page 240,
   b. the translation of the atomic urban space, see Item **10.7.2** on page 241,
   c. the compilation of the analyser structure, see Item **10.5.2**,
   d. the compilation of planner structure. see Item **10.5.3** on the facing page,

```
value
1046   BEHAVIOUR_SIGNATURE_UoD(uod) ≡
1046a     TRANSLATE_CLK(clk),
1046b     TRANSLATE_TUS(tus),
1046c     BEHAVIOUR_SIGNATURE_AA(obs_AA(uod)),
1046d     BEHAVIOUR_SIGNATURE_PA(obs_PA(uod))
```

The COMPILER apply to, as here, *structures*, or composite parts. The TRANSLATOR apply to atomic parts. In this section, i.e., Sect. **10.5.1**, we will explain the obvious meaning of these functions: we will not formalise their type, and we will make some obvious short-cuts.

10.5.2 The ANALYSER STRUCTURE COMPILER

1047. Compiling the analyser structure results in an RSL-**Text** which expresses the separate
   a. translation of each of its *n* analysers, see Item **10.7.3** on page 243, and
   b. the translation of the analysis depository, see Item **10.7.4** on page 244.

```
1047   BEHAVIOUR_SIGNATURE_AA(aa) ≡
1047a     { TRANSLATE_A_anm_i(obs_A_anm_i(aa)) | i:[1..n] },
1047b     TRANSLATE_AD(obs_AD(aa))
```

10.5.3 The PLANNER STRUCTURE COMPILER

1048. The *planner structure*, pa:PA, compiles into four elements:
   a. the compilation of the *master planner structure*, see Item **10.5.3.1**,
   b. the translation of the *derived server index generator*, see Item **10.7.5** on page 245,
   c. the translation of the *plan repository*, see Item **10.7.6** on page 246, and
   d. the compilation of the *derived server structure*, see Item **10.5.3.2**.

```
1048  BEHAVIOUR_SIGNATURE_PA(pa) ≡
1048a     BEHAVIOUR_SIGNATURE_MPA(obs_MPA(pa)),
1048b     TRANSLATE_DPXG(obs_DPXG(pa)),
1048c     TRANSLATE_PR(obs_PR(pa)),
1048d     BEHAVIOUR_SIGNATURE_DPA(obs_DPA(pa))
```

10.5.3.1 The MASTER PLANNER STRUCTURE COMPILER

1049. Compiling the *master planner structure* results in an RSL-**Text** which expresses the separate translations of the
   a. the atomic *master planner server*, see Item **10.7.7** on page 247 and
   b. the atomic *master planner*, see Item **10.7.8** on page 248.

```
1049  BEHAVIOUR_SIGNATURE_MPA(mpa) ≡
1049a     TRANSLATE_MPS(obs_MPS(mpa)),
1049b     TRANSLATE_MP(obs_MP(mpa))
```

10.5.3.2 The DERIVED PLANNER STRUCTURE COMPILER

1050. The compilation of the *derived planner structure* results in some RSL-**Text** which expresses the set of separate compilations of each of the *derived planner pair structures*, see Item **10.5.3.3**.

1050 BEHAVIOUR_SIGNATURE_DPA(dpa) ≡ { BEHAVIOUR_SIGNATURE(obs_DPC$_{nm_j}$(pa)) | j:[1..p] }

10.5.3.3 The DERIVED PLANNER PAIR STRUCTURE COMPILER

1051. The compilation of the derived planner pair structure results in some RSL-**Text** which expresses
   a. the results of translating the *derived planner server*, see Item **10.7.9** on page 251 and
   b. the results of translating the *derived planner*, see Item **10.7.10** on page 253.

1051 BEHAVIOUR_SIGNATURE_DPC$_{nm_j}$(dpc$_{nm_j}$), i:[1..p] ≡
1051a TRANSLATE_DPS$_{nm_j}$(obs_DPS$_{nm_j}$(dpc$_{nm_j}$)),
1051b TRANSLATE_DP$_{nm_j}$(obs_DP$_{nm_j}$(dpc$_{nm_j}$))

10.6 Channel Analysis and Channel Declarations

The *transcendental interpretation* of *parts* as *behaviours* implies *existence* of *means of communication & synchronisation* of between and of these behaviours. We refer to Fig. 10.2 for a summary of the channels of the urban space analysis and urban planning system.

MORE TO COME

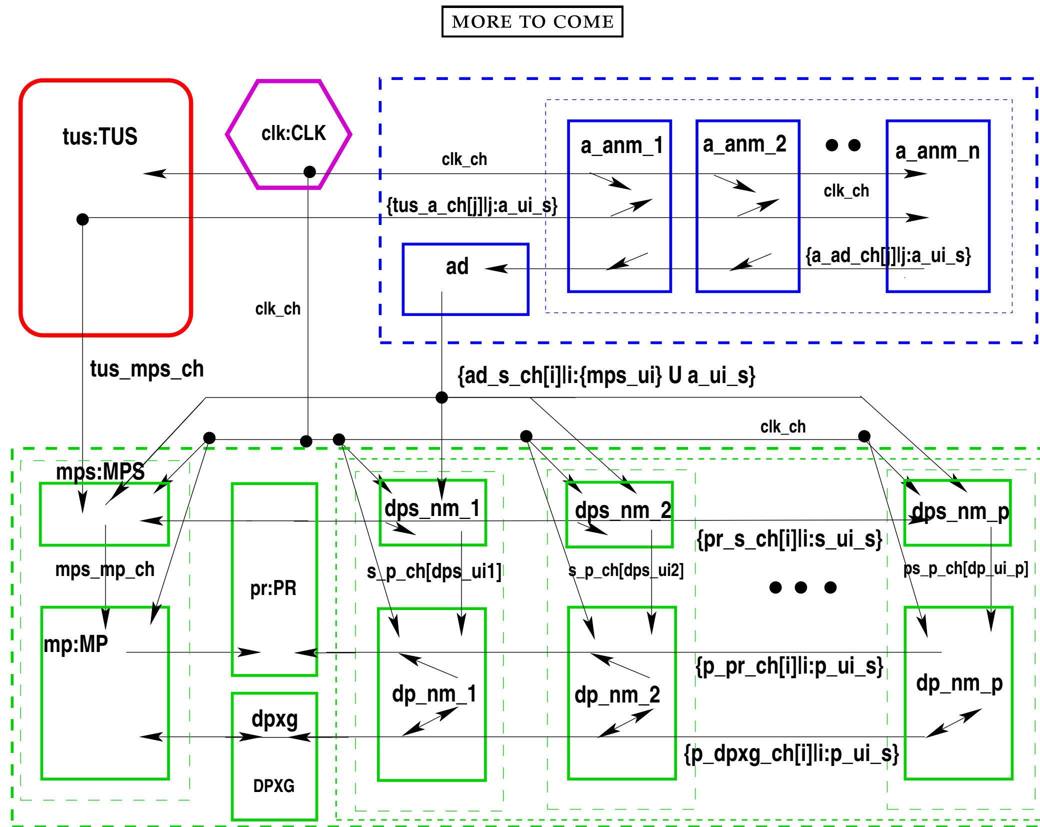


Fig. 10.2 The Urban Space and Analysis Channels and Behaviours

10.6.1 The clk_ch Channel

The purpose of the clk_ch channel is, for the clock, to propagate Time to such entities who inquire. We refer to Sects. **10.3.1** on page 220, **10.3.2** on page 221, **10.3.3** on page 221, **10.3.5** on page 222, **10.3.6** on page 222, **10.3.7** on page 222 and **10.3.8** on page 223 for the mereologies that help determine the indices for the clk_ch channel.

1052. There is declared a (single) channel clk_ch
1053. whose messages are of type CLK_MSG (for Time).

The clk_ch is single. There is no need for enquirers to provide their identification. The clock "freely" dispenses of "its" time.

type
1052 CLK_MSG = T

channel
1053 clk_ch:CLK_MSGch-clk-010

10.6.2 The tus_a_ch Channel

The purpose of the tus_a_ch channel is, for the the urban space, to propagate urban space attributes to analysers. We refer to Sects. **10.3.2** and **10.3.3** for the mereologies that help determine the indices for the tus_a_-ch channel.

1054. There is declared an array channel tus_a_ch whose messages are of
1055. type TUS_MSG (for a *time stamped* aggregate of *urban space attributes*, TUSm, cf. Item 1023 on page 229).

type
1055 TUS_MSG = T × TUSm
channel
1054 {tus_a_ch[a_ui]:TUS_MSG|a_ui:A_UI•a_ui ∈ $a_{ui}s$} ch-tus-a-000

The tus_a_ch channel is to offer urban space information to all analysers. Hence it is an array channel over indices ANms, cf. Item 929 on page 212.

10.6.3 The tus_mps_ch Channel

The purpose of the tus_mps_ch channel is, for the the urban space, to propagate urban space attributes to the master planner server. We refer to Sects. **10.3.2** and **10.3.5** for the mereologies that help determine the indices for the tus_mps_ch channel.

1056. There is declared a channel tus_mps_ch whose messages are of
1055 type TUS_MSG (for a *time stamped* aggregate of *urban space attributes*, TUSm, cf. Item 1023 on page 229).

type
1055 TUS_MSG = T × TUSm
channel
1056 tus_mps_ch:TUS_MSGch-tus-mps-000

The tus_s_ch channel is to offer urban space information to just the master server. Hence it is a single channel.

10.6.4 The a_ad_ch Channel

The purpose of the a_ad_ch channel is, for analysers to propagate analysis results to the analysis depository. We refer to Sects. **10.3.3** and **10.3.4** for the mereologies that help determine the indices for the a_ad_ch channel.

1057. There is declared a channel a_ad_ch whose *time stamped* messages are of
1058. type A_MSG (for *analysis message*).

type
1058 $\mathsf{A_MSG}_{anm_i}$ = (s_T:T × s_A:$\mathsf{Analysis}_{anm_i}$), i:[1:$n$]
1058 A_MSG = $\mathsf{A_MSG}_{anm_1}$|$\mathsf{A_MSG}_{anm_2}$|...|$\mathsf{A_MSG}_{anm_n}$
channel
1057 {a_ad_ch[a_ui]:A_MSG|a_ui:A_UI•a_ui ∈ $a_{ui}s$}ch-a-ad-000

10.6.5 The ad_s_ch Channel

The purpose of the ad_s_ch channel is, for the analysis depository to propagate histories of analysis results to the server. We refer to Sects. **10.3.4**, **10.3.5** and **10.3.7** for the mereologies that help determine the indices for the ad_s_ch channel.

1059. There is declared a channel ad_s_ch whose messages are of
1060. type AD_MSG (defined as A_Hist for a *histories of analyses*), see Item 1026 on page 230.

type
1060 AD_MSG = A_Hist
channel
1059 {ad_s_ch[s_ui]|s_ui:(MPS_UI|DPS_UI)•s_ui ∈{mps_{ui}}∪$dps_{ui}s$}:AD_MSGch-ad-dps-000

The ad_s_ch channel is to offer urban space information to the *master* and *derived servers*. Hence it is an array channel.

10.6.6 The mps_mp_ch Channel

The purpose of the mps_mp_ch channel is for the master server to propagate comprehensive master planner input to the master planner. We refer to Sects. **10.3.5** and **10.3.6** for the mereologies that help determine the indices for the mps_mp_ch channel.

1061. There is declared a channel mps_mp_ch whose messages are of
1062. type MPS_MSG which are quadruplets of time stamped urban space information, TUS_MSG, see Item 1055 on the preceding page, analysis histories, A_Hist, see Item 1060, *master planner auxiliary* information, mAUX, and *master plan requirements*, mREQ.

type
1062 MPS_MSG = TUS_MSG×AD_MSG×mAUX×mREQ
channel
1061 mps_mp_ch:MPS_MSGch-mps-mp-000

The mps_mp_ch channel is to offer MPS_MSG information to just the *master server*. Hence it is a single channel.

10.6.7 The p_pr_ch Channel

The purpose of the p_pr_ch channel is, for master and derived planners to deposit and retrieve master and derived plans to the plan repository. We refer to Sects. **10.3.6** and **10.3.10** for the mereologies that help determine the indices for the p_pr_ch channel.

1063. There is declared a channel p_pr_ch whose messages are of
1064. type PLAN_MSG – for *time stamped master plans*.

type
1064 PLAN_MSG = T × PLANS
channel
1063 {p_pr_ch[p_ui]:PLAN_MSG|p_ui:(MP_UI|DP_UI)•p_ui ∈ $p_{ui}s$} ch-mp-pr-000

The p_pr_ch channel is to offer comprehensive records of all current plans to all the the *planners*. Hence it is an array channel.

10.6.8 The p_dpxg_ch Channel

The purpose of the p_dpxg_ch channel is, for planners to request and obtain derived planner index names of, respectively from the derived planner index generator. We refer to Sects. **10.3.6** and **10.3.9** for the mereologies that help determine the indices for the mp_dpxg_ch channel.

1065. There is declared a channel p_dpxg_ch whose messages are of
1066. type DPXG_MSG. DPXG_MSG messages are
   a. either *request* from the *planner* to the *index generator* to provide zero, one or more of an indicated set of *derived planner names*,
   b. or to accept such a (*response*) set from the *index generator*.

type
1066 DPXG_MSG = DPXG_Req | DPXG_Rsp
1066a DPXG_Req :: DNm-**set**
1066b DPXG_Rsp :: DNm-**set**
channel
1065 {p_dpxg_ch[ui]:DPXG_MSG|ui:(MP_UI|DP_UI)•ui ∈ $p_{ui}s$} ch-mp-ix-000

10.6.9 The pr_s_ch Channel

The purpose of the pr_s_ch channel is, for the plan repository to provide master and derived plans to the derived planner servers. We refer to Sects. **10.3.10** and **10.3.7** for the mereologies that help determine the indices for the pr_dps_ch channel.

1067. There is declared a channel pr_dps_ch whose messages are of
1068. type PR_MSGd, defined as PLAp, cf. Item 1043 on page 233.

type
1068 PR_MSG = PLANS
channel
1067 {pr_s_ch[ui]:PR_MSGd|ui:(MPS_UI|DPS_UI)•ui ∈ $s_{ui}s$}ch-pr-dps-000

10.6.10 The dps_dp_ch Channel

The purpose of the dps_dp_ch channel is, for derived planner servers to provide input to the derived planners. We refer to Sects. **10.3.7** and **10.3.8** for the mereologies that help determine the indices for the dps_dp_ch channel.

1069. There is declared a channel dps_dp_ch[ui_nm_j], one for each *derived planner* pair.
1070. The channel messages are of type DPS_MSG$_{nm_j}$. These DPS_MSG$_{nm_i}$ messages are quadruplets of *analysis* aggregates, AD_MSG, *urban plan* aggregates, PLANS, *derived planner auxiliary information*, dAUX$_{nm_j}$, and *derived plan requirements*, dAUX$_{nm_j}$.

type
1070 DPS_MSG$_{nm_j}$ = AD_MSG×PLANS×dAUX$_{nm_j}$×dREQ$_{nm_j}$, j:[1..p]
channel
1069 {dps_dp_ch[ui]:DPS_MSG$_{nm_j}$|ui:DPS_UI•ui ∈ $dps_{ui}s$}ch-dps-dp-000

10.7 The Atomic Part Translators

10.7.1 The clock Translator

We refer to Sect. **10.4.1.2** for the attributes that play a rôle in determining the clock signature.

10.7.1.1 The Translate_CLK Function

1071. The Translate_CLK(*clk*) results in three text elements:
   a. the **value** keyword,
   b. the *signature* of the clock definition,
   c. and the *body* of that definition.

The clock signature contains the *unique identifier* of the clock; the *mereology* of the clock, cf. Item **10.3.1** on page 220; and the *attributes* of the clock, in some form or another: the programmable time attribute and the channel over which the clock offers the time.

value
1071 Translate_CLK(*clk*) ≡
1071a " **value**
1071b clock: T → **out** clk_ch **Unit**
1071c clock(uid_CLK(*clk*),mereo_CLK(*clk*))(attr_T(*clk*)) ≡ ... "

10.7.1.2 The clock Behaviour

The purpose of the clock is to show the time. The "players" that need to know the time are: the urban space when informing requestors of aggregates of urban space attributes, the analysers when submitting analyses to the analysis depository, the planners when submitting plans to the plan repository.

1072. We see the clock as a behaviour.

1073. It takes a programmable input, the *current* time, t.
1074. It repeatedly emits the some *next* time on channel clk_ch.
1075. Each iteration of the clock it non-deterministically, internally increments the *current* time by either nothing or an infinitisimally small time interval δ ti, cf. Item 997 on page 224.
1076. In each iteration of the clock it either offers this *next* time, or skips doing so;
1077. whereupon the clock resumes being the clock albeit with the new, i.e., *next* time.

```
value
1074  δti:TI = ... cf. Item 997 on page 224
1072  clock: T → out clk_ch Unit
1073  clock(uid_clk,mereo_clk)(t) ≡
1075     let t′ = (t+δti) ⊓ t in
1076     skip ⊓ clk_ch!t′ ;
1077     clock(uid_clk,mereo_clk)(t′) end
1077  pre: uid_clk = clk_ui ∧
1077      mereo_clk = (tus_ui,a_ui s,mps_ui,mp_ui,dps_ui s,dp_ui s)
```

10.7.2 The URBAN SPACE TRANSLATOR

We refer to Sect. **10.4.2.2** for the attributes that play a rôle in determining the urban space signature.

10.7.2.1 The TRANSLATE_TUS Function

1078. The TRANSLATE_TUS(*tus*) results in three text elements:
   a. the **value** keyword
   b. the *signature* of the urb_spa definition,
   c. and the *body* of that definition.

   The urban space signature contains the *unique identifier* of the urban space, the *mereology* of the urban space, cf. Item **10.3.2** on page 221, the static point space *attribute*.

```
value
1078   TRANSLATE_TUS(tus) ≡
1078a     " value
1078b         urb_spa: TUS_UI × TUS_Mer → Pts →
1078b             out ... Unit
1078c         urb_spa(uid_TUS(tus),mereo_TUS(tus))(attr_Pts(tus)) ≡ ...  "
```

We shall detail the urb_spa signature and the urb_spa body next.

10.7.2.2 The urb_spa Behaviour

The urban space can be seen as a behaviour. It is "visualized" as the rounded edge box to the left in Fig. 10.3 on the next page. It is a "prefix" of Fig. 10.3 on the following page. In this section we shall refer to many other elements of our evolving specification. To grasp the seeming complexity of the urban space, its analyses and its urban planning functions, we refer to Fig. 10.3 on the next page.

1079. To every observable part, like tus:TUS, there corresponds a behaviour, in this case, the urb_spa.

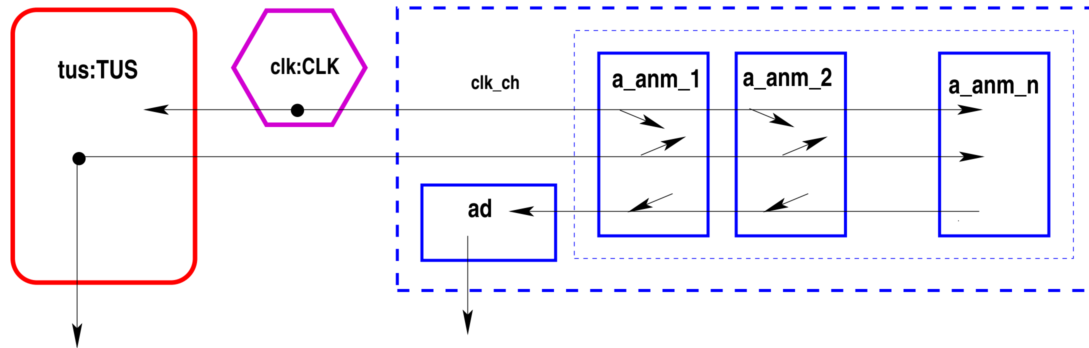


Fig. 10.3 The Urban Space and Analysis Behaviours

1080. The urb_spa behaviour has, for this report, just one static attribute, the point space, Pts.
1081. The urb_spa behaviour has the following biddable and programmable attributes, the Cadastral, the Law and the SocioEconomic attributes. The biddable and programmable attributes "translate" into behaviour parameters.
1082. The urb_spa behaviour has the following dynamic, non-biddable, non-programmable attributes, the GeoDetic, GeoTechnic and the Meterological attributes The non-biddable, non-programmable dynamic attributes "translate", in the conversion from parts to behaviours, to input channels etc.

the_urb_spa behaviour offers its attributes, upon demand,

1083. to a urban space analysis behaviours, tus_ana_i and one master urban server.
1084. The urb_spa otherwise behaves as follows:
   a. it repeatedly "assembles" a tuple, tus, of all attributes;
   b. then it external non-deterministically either offers the tus tuple
   c. to either any of the urban space analysis behaviours,
   d. or to the master urban planning behaviour;
   e. in these cases it resumes being the urb_spa behaviour;
   f. or internal-non-deterministically chooses to
   g. update the law, the cadastral, and the socio-economic attributes;
   h. whereupon it resumes being the urb_spa behaviour.

```
channel
1082   attr_Pts_ch:Pts, attr_GeoD_ch:GeoD, attr_GeoT_ch:GeoT, attr_Met_ch:Met
1083   tus_mps_ch:TUSm
1083   {tus_a_ch[ai]|ai ∈ a_ui s}:TUSm
value
1079   urb_spa: TUS_UI × TUS_Mer →
1080       Pts →
1081           (Cada×Law×Soc_Eco×...) →
1082               in attr_Pts_ch, attr_GeoD_ch, attr_GeoT_ch, attr_Met_ch →
1083               out tus_mps_ch, {tus_ana_ch[ai]|ai ∈ [a_1...a_a]} → Unit
1084   urb_spa(pts)(pro) ≡
1084a      let geo = ["pts"↦attr_Pts_ch?"ged"↦attr_GeoD_ch?,"cad"↦cada,"get"↦attr_geoT_ch?,
1084a                 "met"↦attr_Met_ch?,"law"↦law,"eco"↦eco,...] in
1084c      ((⊞ {tus_a_ch[ai]!geo|ai ∈ a_ui s}
1084b       ⊞
```

```
1084d      tus_mps_ch!geo) ;
1084e     urb_spa(pts)(pro)) end
1084f    ⊓
1084g     let pro′:(Cada×Law×Soc_Eco×...)•fit_pro(pro,pro′) in
1084h     urb_spa(pts)(pro′) end

1084g  fit_pro: (Cada×Law×Soc_Eco×...) × (Cada×Law×Soc_Eco×...) → Bool
```

We leave the *fitness predicate* fit_pro further undefined. It is intended to ensure that the biddable and programmable attributes evolve in a commensurate manner.

10.7.3 The ANALYSER$_{anm_i}$, i:[1 : n] TRANSLATOR

We refer to Sect. **10.4.4** for the attributes that play a rôle in determining the analyser signature.

10.7.3.1 The TRANSLATE_A$_{anm_j}$ Function

1085. The TRANSLATE_A$_{anm_j}$(a_{anm_j}) results in three text elements:
   a. the **value** keyword,
   b. the *signature* of the analysera_{anm_j} definition,
   c. and the *body* of that definition.

The analyser$_{anm_j}$ signature contains the *unique identifier* of the analyser, the *mereology* of the analyser, cf. Item **10.3.3** on page 221, and the *attributes*, here just the programmable attribute of the most recent analysisa_{anm_j} performed by the analyser$_{anm_j}$.

```
type
1085  Analysis = Analysis_nm1|Analysis_nm2|...|Analysis_nmn
value
1085  TRANSLATE_A_nmi(a_nmi):
1085   ” value
1085       analyser_nmi: (uid_A×mereo_A) →
1085         Analysis_nmi →
1085         in  tus_a_ch[uid_A(a_nmi)]
1085         out a_ad_ch[uid_A(a_nmi)]
1085       analyser_uij(uid_A(a_nmi),mereo_A(a_nmi))(ana_nmi) ≡ ...  ”
```

10.7.3.2 The analyser$_{ui_j}$ Behaviour

Analyses, or various kinds, of the urban space, is an important prerequisite for urban planning. We therefore introduce a number, n, of urban space analysis behaviours, analysis$_{anm_i}$ (for anm_i in the set $\{anm_1,...,anm_a\}$. The indexing designates that each analysis$_{anm_i}$ caters for a distinct kind of urban space analysis, each analysis with respect to, i.e., across existing urban areas: ..., (a_i) traffic statistics, (a_j) income distribution, ..., (a_k) health statistics, (a_ℓ) power consumption, ..., (a_a) ... . We shall model, by an indexed set of behaviours, ana$_i$, the urban [space] analyses that are an indispensable prerequisite for urban planning.

1086. Urban [space] analyser, tus_ana$_i$, for $a_i \in [a_1...a_a]$, performs analysis of an urban space whose attributes, except for its point set, it obtains from that urban space – via channel tus_ana_ch and
1087. offers analysis results to the mp_beh and the n derived behaviours.
1088. Urban analyser, ana$_{a_i}$, otherwise behaves as follows:
   a. The analyser obtains, from the urban space, its most recent set of attributes.
   b. The analyser then proceeds to perform the specific analysis as "determined" by its index a_i.
   c. The result, tus_ana$_{a_i}$, is communicated whichever urban, the master or the derived, planning behaviour inquires.
   d. Whereupon the analyser resumes being the analyser, improving and/or extending its analysis.

```
type
1085  Analysis = Analysis_anm1|Analysis_anm2|...|Analysis_anmn
value
1088  analyser_nmi(a_ui,a_mer)(analysis_nmi) ≡
1088a     let tusm = tus_a_ch[a_ui] ? in
1088b     let analysis′_nmi = perform_analysis_nmi(tusm)(analysis) in
1088c     ⌈⌉ a_ad_ch[a_ui] ! (clk_ck?,analysis′_nmi) ;
1088d     analyser_i(a_ui,a_mer)(analysis′_nmi) end end

1088b  perform_analysis_anm_i: TUSm → Analysis_anm_i → Analysis_anm_i
1088b  perform_analysis_anm_i(tusm)(analysis_anm_i) ≡ ...
```

10.7.4 The Analysis Depository Translator

We refer to Sect. **10.4.5** for the attributes that play a rôle in determining the analysis depository signature.

10.7.4.1 The Translate_AD Function

1089. The Translate_AD(ad) results in three text elements:
   a. the **value** keyword
   b. the *signature* of the ana_dep definition,
   c. and the *body* of that definition.

The ana_dep signature essentially contains the *unique identifier* of the analyser, the *mereology* of the analyser, cf. Item **10.3.4** on page 221, and the *attributes*, in one form or another: the programmable attribute, a_hist, see Item 1026 on page 230, the channels over which ana_dep either accepts time stamped *analyses*, Analysis$_{a_{u_i}}$, from *analyser*$_{anm_i}$, or offers a_hists to either the *master planner server* or the *derived planner servers*.

```
value
1089  Translate_AD(ad) ≡
1089a   " value
1089b       ana_dep: (A_UI × A_Mer) → AHist →
1089b         in {a_ad_ch[i]|i:A_UI•i ∈ a_ui s}
1089b         out {ad_s_ch[i]|i:A_UI•i ∈ s_ui s} Unit
1089c       ana_dep(ui_A(ad),mereo_A(ad))(attr_AHist(ad)) ≡ ...  "
```

10.7.4.2 The ana_dep Behaviour

The definition of the *analysis depository* is as follows.

1090. The behaviour of ana_dep is as follows: non-deterministically, externally (⌷), ana_dep
1091. either (⌷, line 1093) offers to accept a time stamped analysis *from some* analyser (⌷{ ... | ... }),
    a. receiving such an analyses it "updates" its history,
    b. and resumes being the ana_dep behaviour with that updated history;
1092. or offers the analysis history *to the* master planner server
    and resumes being the ana_dep behaviour;
1093. or offers the analysis history
    a. *to whichever* (⌷{ ... | ... }) planner server offers to accept a history
    b. and resumes being the ana_dep behaviour with that updated history.

```
value
1090   ana_dep(a_ui,a_mer)(ahist) ≡
1091     ⌷ { (let ana = a_ad_ch[ i ] ? in
1091a         let ahist′ = ahist†[ i↦⟨ana⟩⁀(ahist(i)) ] in
1091b         ana_dep(a_ui,a_mer)(ahist′) end end)
1091b       | i:A_UI•i∈ a_uis }
1092     ⌷ (ad_mps_ch!ahist ; ana_dep(a_ui,a_mer)(ahist))
1093     ⌷
1093a       ({ ad_s_ch[ j ]!ahist
1093a        | j:(MPS_UI|DPS_UI)•j∈s_uis};
1093b         ana_dep(a_ui,a_mer)(ahist))
```

10.7.5 The DERIVED PLANNER INDEX GENERATOR TRANSLATOR

We refer to Sect. **10.4.10** for the attributes that play a rôle in determining the derived planner index generator signature.

10.7.5.1 The TRANSLATE_DPXG(*dpxg*) Function

1094. The TRANSLATE_DPXG(*dpxg*) results in three text elements:
    a. the **value** keyword
    b. the *signature* of the dpxg behaviour definition,
    c. and the *body* of that definition.

The signature of the dpxg behaviour definition has many elements: the *unique identifier* of the dpxg behaviour, the *mereology* of the dpxg behaviour, cf. Item **10.3.9** on page 223, and the *attributes* in some form or another:the *unique identifier*, the *mereology*, and the *attributes*, in some form or another: the programmable attribute All_DPUIs, cf. Item 1040 on page 232, the programmable attribute Used_DPUIs, cf. Item 1041 on page 232, the mp_dpxg_ch input/output channel, and the dp_dpxg_ch input/output array channel.

```
value
1094   TRANSLATE_DPXG(dpxg) ≡
1094a     " value
```

```
1094b     dpxg_beh: (DPXG_UI×DPXG_Mer) →
1094b        (All_DPUIs×UsedDPUIS) →
1094b        in,out {p_dpxg_ch[i]|i:(MP_UI|DP_UI)•i∈p_uis} Unit
1094c     dpxg_beh(uid_DPXG(dpxg),mereo_DPXG(dpxg))(all_dpuis,used_dpuis) ≡ ... ”
```

10.7.5.2 The dpxg Behaviour

1095. The index generator otherwise behaves as follows:
   a. It non-deterministically, externally, offers to accept requests from any planner, whether master or server. The request suggests the names, req, of some derived planners.
   b. The index generator then selects a suitable subset, sel_dpuis, of these suggested derived planners from those that are yet to be started.
   c. It then offers these to the requesting planner.
   d. Finally the index generator resumes being an index generator, now with an updated used_dpuis programmable attribute.

```
value
1095   dpxg: (DPXG_UI×DPXG_Mer) → (All_DPUIs×Used_DPUIs) →
1095      in,out mp_dpxg_ch,
1095             {p_dpxg_ch[j]|j:(MP_UI|DP_UI)•j∈{p_uis}} Unit
1095   dpxg(dpxg_ui,dpxg_mer)(all_dpuis,used_dpuis) ≡
1095a      ⫿ { let req = p_dpxg_c[j] ? in
1095b          let sel_dpuis = all_dpuis \ used_dpuis • sel_dpuis ⊆ req_dpuis in
1095c          dp_dpxg_ch[j] ! sel_dpuis ;
1095d          dpxg(dpxg_ui,dpxg_mer)(all_dpuis,used_dpuis∪sel_dpuis) end end
1095        | j:(MP_UI|DP_UI)•j∈p_uis }
```

10.7.6 The Plan Repository Translator

We refer to Sect. **10.4.11** for the attributes that play a rôle in determining the plan repository signature.

10.7.6.1 The Translate_PR Function

1096. The Translate_PR(*pr*) results in three text elements:
   a. the **value** keyword,
   b. the *signature* of the plan repository definition,
   c. and the *body* of that definition.

The plan repository signature contains the *unique identifier* of the plan repository, the *mereology* of the plan repository, cf. Item **10.3.10** on page 223, and the *attributes*: the *programmable* plans, cf. 1043 on page 233, and the *input/out channel* p_pr_ch.

```
value
1096   Translate_PR(pr) ≡
1096a      ” value
1096b          plan_rep: PLANS →
```

```
1096b          in  {p_pr_ch[i]|i:(MP_UI|DP_UI)•i∈p_uis}
1096b          out {s_pr_ch[i]|i:(MP_UI|DP_UI)•i∈s_uis}  Unit
1096c       plan_rep(plans)(attr_AllDPUIs(pr),attr_UsedDPUIs(pr)) ≡ ...  ”
```

10.7.6.2 The plan_rep Behaviour

1097. The plan repository behaviour is otherwise as follows:
   a. The plan repository non-deterministically, externally chooses between
      i. offering to accept time-stamped plans from a planner, p_{ui}, either the master planner or anyone of the derived planners,
      ii. from whichever planner so offers,
      iii. inserting these plans appropriately, i.e., at p_{ui}, as the new head of the list of “there”,
      iv. and then resuming being the plan repository behaviour appropriately updating its programmable attribute;
   b. or
      i. offering to provide a full copy of its plan repository map
      ii. to whichever server requests so,
      iii. and then resuming being the plan repository behaviour.

```
value
1097a     plan_rep(pr_ui,ps_uis)(plans) ≡
1097(a)i     ⌈⌉⌊⌋ { let (t,plan) = p_pr_ch[i] ? in assert: i ∈ dom plans
1097(a)iii          let plans′ = plans † [i↦⟨(t,plan)⟩⌒plans(i)] in
1097(a)iv           plan_rep(pr_ui,ps_uis)(plans′) end end
1097(a)ii        | i:(MP_UI|DP_UI)•i∈p_uis }
1097b        ⌈⌉⌊⌋
1097(b)i     ⌈⌉⌊⌋ { s_pr_ch[i] ! plans ; assert: i ∈ dom plans
1097(b)iii          plan_rep(pr_ui,ps_uis)(plans)
1097(b)ii        | i:(MP_UI|DP_UI)•i∈p_uis }
```

10.7.7 The MASTER SERVER TRANSLATOR

We refer to Sect. **10.4.6** for the attributes that play a rôle in determining the master server signature.

10.7.7.1 The TRANSLATE_MPS Function

1098. The TRANSLATE_MPS(*mps*) results in three text elements:
   a. the **value** keyword,
   b. the *signature* of the master_server definition,
   c. and the *body* of that definition.

The master_server signature contains the *unique identifier* of the master server, the *mereology* of the master server, cf. Item **10.3.5** on page 222, and the *dynamic attributes* of the master server: the most recently, previously produced *auxiliary* information, the most recently, previously produced plan *requirements* information, the clock channel, the urban space channel, the analysis depository channel, and the master planner channel.

```
value
1098   TRANSLATE_MPS(mps) ≡
1098a      " value
1098b          master_server: (mAUX×mREQ) →
1098b              in   clk_ch, tus_m_ch, ad_s_ch[ uid_MPS(mps) ]
1098b              out mps_mp_ch  Unit
1098c          master_server(uid_MPS(mps),mereo_MPS(mps))(attr_mAUX(mps),attr_mREQ(mps)) ≡ ...   "
```

10.7.7.2 The master_server Behaviour

1099. The master_server obtains time from the clock, see Item 1100c, information from the urban space, and the most recent analysis history, assembles these together with "locally produced"
   a. *auxiliary* planner information and
   b. plan *requirements*

   as input, MP_ARG, to the master planner.
1100. The master server otherwise behaves as follows:
   a. it obtains latest urban space information and latest analysis history, and
   b. then produces auxiliary planning and plan requirements commensurate, i.e., fit, with the most recently, i.e., previously produced such information;
   c. it then offers a time stamped compound of these kinds of information to the master planner,
   d. whereupon the master server resumes being the master server, albeit with updated programmable attributes.

```
type
1099a  mAUX
1099b  mREQ
1099   mARG = (T × ((mAUX × mREQ) × (TUSm × AHist)))
value
1100   master_server(uid,mereo)(aux,req) ≡
1100a    let tusm = tus_m_ch ? , ahist = ad_s_ch[ mps_ui ] ? ,
1100b         maux:mAUX, mreq:mREQ • fit_AuxReq((aux,req),(maux,mreq)) in
1100c    s_p_ch[ uid ] ! (clk_ch?,((maux,mreq),(tusm,ahist))) ;
1100d    master_server(uid,mereo)(maux,mreq)
1100      end

1100b  fitAuxReq: (mAUX×mREQ)×(mAUX×mREQ) → Bool
1100b  fitAuxReq((aux,req),(maux,mreq)) ≡ ...
```

10.7.8 The MASTER PLANNER TRANSLATOR

We refer to Sect. **10.4.7** for the attributes that play a rôle in determining the master planner signature.

10.7.8.1 The TRANSLATE_MP Function

1101. The TRANSLATE_MP(*mp*) results in three text elements:

a. the **value** keyword,
b. the *signature* of the master_planner definition,
c. and the *body* of that definition.

The master_planner signature contains the *unique identifier* of the master planner, the *mereology* of the master planner, cf. Item **10.3.6** on page 222, and the *attributes* of the master planner: the script, cf. Sect. **10.4.3** on page 230 and Item 1024 on page 230, a set of script pointers, cf. Item 1031 on page 231, a set of analyser names, cf. Item 1032 on page 231, a set of planner identifiers, cf. Item 1033 on page 231, and the channels as implied by the master planner mereology.

```
value
1101   TRANSLATE_MP(mp) ≡
1101a     " value
1101b         master_planner: Mmp_ui:P_UI×MP_Mer×(Script×ANms×DPUIs) →
1101b             Script_Pts →
1101b                in      clk_ch, mps_mp_ch, ad_ps_ch[ mp_ui ]
1101b                out     p_pr_ch[ mp_ui ]
1101b                in,out p_dpxg_ch[ mp_ui ]  Unit
1101c         master_planner(uid_MP(mp),mereo_MP(mp),
1101c             (attr_Script(mp),attr_ANms(mp),attr_DPUIs(mp)))(attr_Script_Ptrs(mp)) ≡ ...   "
```

10.7.8.2 The Master urban_planning Function

1102. The core of the master_planner behaviour is the master_urban_planning function.
1103. It takes as arguments: the script, a set of analyser names, a set of derived planner identifiers, a set of script pointers, and the time-stamped master planner argument, cf. Item 1099 on the preceding page;
1104. and delivers, i.e., yields, a set of "remaining" derived planner identifiers, an updated set of script pointers, and a master result:M_RES, i.e., a master plan, mp:M_PLAN together with the time stamped master argument from which the plan was constructed.
1105. The master urban planning function is not defined by other than a predicate:

a. the "remaining" derived planner identifiers is a subset of the arguments derived planner identifiers;
b. the "resulting" master argument is the same as the input master argument, i.e., it is "carried forward";
c. the arguments: the script, the analyser names, the derived planner identifiers, the set of script pointers, the time-stamped master planner argument, and the result plan otherwise satisfies a predicate $\mathcal{P}$(script,anms,dpuis,ptrs,marg)(mplan) expressing that the result mplan is an appropriate plan in view of the other arguments.

```
type
1104   M_PLAN
1104   M_RES = M_PLAN × DPUI-set × M_ARG
value
1103   master_urban_planning:
1103       Script × ANm-set × DP_UI-set × Script_Ptr-set × M_ARG
1104           → (DP_UI-set × Script_Ptr-set) × M_RES
1102   master_urban_planning(script,anms,dpuis,ptrs,marg)
1105a      as ((dpuis′,ptrs′),(mplan,marg′))
1105a         dpuis′ ⊆ dpuis
1105b      ∧ marg′ = marg
```

```
1105c      ∧ 𝒫(script,anms,dpuis,ptrs,marg)(mplan)
1102  𝒫: ((Script×ANM-set×DP_UI-set×Script_Ptr-set×M_ARG×MPLAN×Script_Ptr-set)
1102            × (DP_UI-set×Script_Ptr-set×M_ARG×MPLAN)) → Bool
1102  𝒫((script,anms,dpuis,ptrs,marg,mplan,ptrs),(dpuis′,ptrs′,marg,mplan)) ≡ ...
```

10.7.8.3 The master_planner Behaviour

1106. The master_planner behaviours is otherwise as follows:
   a. The master_planner obtains, from the master server, its time stamped master argument, cf. Item 1099 on page 248;
   b. it then invokes the master urban planning function;
   c. the time-stamped result is offered to the plan repository;
   d. if the result is OK as a final result,
   e. then the behaviour is stopped;
   f. otherwise
      i. the master planner inquires the derived planner index generator as for such derived planner identifiers which are not used;
      ii. the master planner behaviour is the resumed with the appropriately updated programmable script pointer attribute, in parallel with
      iii. the distributed parallel composition of the parallel behaviours of the derived servers
      iv. and the derived planners
      v. designated by the derived planner identifiers transcribed into (*nm_dps_ui*) derived server, respectively into (*nm_dp_ui*) derived planner names. For these transcription maps we refer to Sect. **10.2.12** on page 219, Item 981 on page 220.

```
value
1106     master_planner(uid,mereo,(script,anms,puis))(ptrs) ≡
1106a     let (t,((maux,mreq),(tusm,ahist))) = mps_mp_ch ? in
1106b     let ((dpuis′,ptrs′),mres) = master_urban_planning(script,anms,dpuis,ptrs) in
1106c     p_pr_ch[uid] ! mres ;
1106d     if completed(mres) assert: ptrs′ = {}
1106e        then init_der_serv_planrs(uid,dpuis′)
1106f        else
1106(f)i          init_der_serv_plans(ui,dpuis)
1106(f)ii      || master_planner(uid,mereo,(script,anms,puis))(ptrs′)
1106     end end end
```

10.7.8.4 The initiate derived servers and derived planners Behaviour

The init_der_serv_planrs behaviour plays a central rôle. The outcome of the urban planning functions, whether for master or derived planners, result in a possibly empty set of derived planner identifiers, dpuis. If empty then that shall mean that the planner, in the iteration, of the planner behaviour is suggesting that no derived server/derived planner pairs are initiated. If dpuis is not empty, say consists of the set $\{dp_{ui_i}, dp_{ui_j}, ..., dp_{ui_k}\}$ then the planner behaviour is suggesting that derived server/derived planner pairs whose planner element has one of these unique identifiers, be appropriately initiated.

1107. The init_der_serv_planrs behaviour takes the unique identifier, uid, of the "initiate issuing" planner and a suggested set of derived planner identifiers, dpuis.

1108. It then obtains, from the *derived planner index generator*, dpxg, a subset, dpuis′, that may be equal to dpuis.

It then proceeds with the parallel initiation of

1109. derived servers (whose names are extracted, extr_Nm, from their identifiers, cf. Item 975 on page 219),
1110. and planners (whose names are extracted, extr_Nm, from their identifiers, cf. Item 976 on page 219)
1111. for every dp_ui in the set dpuis′.

However, we must first express the selection of appropriate arguments for these server and planner behaviours.

1112. The selection of the server and planner parts, making use of the identifier to part mapping *nms_dp_ui* and *nm_dp_ui*, cf. Items 981– 982 on page 220;
1113. the selection of respective identifiers,
1114. mereologies, and
1115. auxiliary and
1116. requirements attributes.

```
value
1107  init_der_serv_planrs: uid:(DP_UI|MP_UI) × DP_UI-set → in,out pr_dpxg[uid] Unit
1107  init_der_serv_planrs(uid,dpuis) ≡
1108     let dpuis′ = (pr_dpxg_ch[uid] ! dpuis ; pr_dpxg_ch[uid] ?) in
1112     || { let p = c_p(dp_ui), s = c_s(nms_dp_ui(dp_ui)) in
1113          let ui_p = uid_DP(p), ui_s = uid_DPS(s),
1114              me_p = mereo_DP(p), me_s = mereo_DPS(s),
1115              aux_p = attr_sAUX(p), aux_s = attr_sAUX(s),
1116              req_p = attr_sREQ(p), req_s = attr_sREQ(s) in
1109          derived_server_extr_Nm(dp_ui)(ui_s,me_s,(aux_s,req_s)) ||
1110          derived_planner_extr_Nm(dp_ui)(ui_p,me_p,(aux_p,req_p))
1111       | dp_ui:DP_UI•dpui ∈ dpuis′ end end }
1107     end
```

10.7.9 The DERIVED SERVER$_{nm_i}$, i:[1 : p] TRANSLATOR

We refer to Sect. **10.4.8** for the attributes that play a rôle in determining the derived server signature.

10.7.9.1 The TRANSLATE_DPS$_{nm_j}$ Function

1117. The TRANSLATE_DPS(dps_{nm_j}) results in three text elements:
   a. the **value** keyword,
   b. the *signature* of the derived_server definition,
   c. and the *body* of that definition.

The derived_server$_{nm_j}$ signature of the derived server contains the *unique identifier*; the *mereology*, cf. Item **10.3.7** on page 222 – used in determining channels: the dynamic clock identifier, the analysis depository identifier, the derived planner identifier; and the *attributes* which are: the auxiliary, dAUX$_{nm_j}$ and the plan requirements, dREQ$_{nm_j}$.

```
value
1117   TRANSLATE_DPS(dps_nm_j) ≡
1117a    ” value
1117b       derived_server_nm_j:
1117b          DPS_UI_nm_j×DPS_Mer_nm_j → (DAUX_nm_j×dREQ_nm_j) →
1117b             in clk_ch, ad_s_ch[ uid_DPS(dps_nm_j) ]
1117b             out s_p_ch[ uid_DPS(dps_nm_j) ]  Unit
1117c       derived_server_nm_j
1117c             (uid_DPS(dps_nm_j),mereo_DPS(dps_nm_j)),(attr_dAUX(dps_nm_j),attr_dREQ(dps_nm_j)) ≡ ...   ”
```

10.7.9.2 The derived_server Behaviour

The derived_server is almost identical to the master server, cf. Sect. **10.7.7.2**, except that *plans* replace *urban space* information.

1118. The derived_server obtains time from the clock, see Item 1119c, , and the most recent analysis history, assembles these together with "locally produced"
   a. *auxiliary* planner information and
   b. plan *requirements*

   as input, MP_ARG, to the master planner.
1119. The master server otherwise behaves as follows:
   a. it obtains latest plans and latest analysis history, and
   b. then produces auxiliary planning and plan requirements commensurate, i.e., fit, with the most recently, i.e., previously produced such information;
   c. it then offers a time stamped compound of these kinds of information to the derived planner,
   d. whereupon the derived server resumes being the derived server, albeit with updated programmable attributes.

```
type
1118a  dAUX_nm_j
1118b  dREQ_nm_j
1118   dARG_nm_j = (T × ((dAUX_nm_j × dREQ_nm_j) × (PLANS × AHist)))
value
1119   derived_server_nm_j(uid,mereo)(aux,req) ≡
1119a    let plans = ps_pr_ch[ uid ] ?, ahist = ad_s_ch[ uid ] ?,
1119b        daux:dAUX, dreq:dREQ • fit_AuxReq_nm_j((aux,req),(daux,dreq)) in
1119c    s_p_ch[ uid ] ! (clk_ch?,((maux,mreq),(plans,ahist))) ;
1119d    derived_server_nm_j(uid,mereo)(daux,dreq)
1119     end

1119b  fitAuxReq_nm_j: (dAUX_nm_j×dREQ_nm_j)×(dAUX_nm_j×dREQ_nm_j) → Bool
1119b  fitAuxReq_nm_j((aux,req),(daux,dreq)) ≡ ...
```

You may wish to compare formula Items 1118–1119d above with those of formula Items 1099–1100d of Sect. **10.7.7.2** on page 248.

10.7.10 The DERIVED PLANNER$_{nm_i}$, i:[1 : p] TRANSLATOR

We refer to Sect. **10.4.9** for the attributes that play a rôle in determining the derived planner signature.

10.7.10.1 The TRANSLATE_DPdp_{nm_j} Function

This function is an "almost carbon copy" of the TRANSLATE_MPdp_{nm_j} function. Thus Items 1120–1120c are "almost the same" as Items 1101– 1101c on page 249.

1120. The TRANSLATE_DP($_{nm_j}$) results in three text elements:
   a. the **value** keyword,
   b. the *signature* of the derived_planner$_{nm_j}$ definition,
   c. and the *body* of that definition.

The derived_planner$_{nm_j}$ signature of the derived planner contains the *unique identifier*, the *mereology*, cf. Item **10.3.8** on page 223 and the *attributes:* the script, cf. Sect. **10.4.3** on page 230 and Item 1024 on page 230, a set of script pointers, cf. Item 1037 on page 232, a set of analyser names, cf. Item 1038 on page 232, a set of planner identifiers, cf. Item 1039 on page 232, and the channels as implied by the master planner mereology.

```
value
1120   TRANSLATE_DP(dp) ≡
1120a    " value
1120b       derived_planner: dp_ui:DP_UI×DP_Mer×(Script×ANms×DPUIs) → Script_Pts →
1120b            in      s_p_ch[dp_ui], clk_ch, ad_ps_ch[dp_ui]
1120b            out     p_pr_ch[dp_ui]
1120b            in,out p_dpxg_ch[dp_ui]  Unit
1120c       derived_planner(uid_DP(dp),mereo_DP(dp),
1120c           (attr_Script(dp),attr_ANms(dp),attr_DPUIs(dp)))(attr_Script_Ptrs(dp)) ≡ ...   "
```

10.7.10.2 The derived_urban_planning Function

This function is an "almost carbon copy" of the master_urban_planning function. Thus Items 1121–1124c on the following page are "almost the same" as Items 1102– 1105c on page 249.

1121. The core of the derived_planner behaviour is the derived_urban_planning function.
1122. It takes as arguments: the script, a set of analyser names, a set of derived planner identifiers, a set of script pointers, and the time-stamped derived planner argument, cf. Item 1099 on page 248;
1123. and delivers, i.e., yields, a set of "remaining" derived planner identifiers, an updated set of script pointers, and a master result, M_RES, i.e., a master plan, mp:M_PLAN together with the time stamped master argument from which the plan was constructed.
1124. The master urban planning function is not defined by other that a predicate:
   a. the "remaining" derived planner identifiers is a subset of the arguments derived planner identifiers;
   b. the "resulting" master argument is the same as the input master argument, i.e., it is "carried forward";

c. the arguments: the script, the analyser names, the derived planner identifiers, the set of script pointers, the time-stamped master planner argument, and the result plan otherwise satisfies a predicate $\mathcal{P}_{dnm_i}$(script$_{dnm_i}$,anms,dpuis,ptrs,marg$_{dnm_i}$)(dplan$_{dnm_i}$) expressing that the result mplan is an appropriate plan in view of the other arguments.

```
type
1123  D_PLAN_dnm_i
1123  D_RES_dnm_i = D_PLAN_dnm_i × DP_UI-set × D_ARG_dnm_i
value
1122  derived_urban_planning_dnm_i:
1122      Script_dnm_i × ANm-set × DP_UI-set × Script_Ptr-set × D_ARG_dnm_i
1123          → (DP_UI-set × Script_Ptr-set) × D_RES_dnm_i
1121  derived_urban_planning_dnm_i(script,anms,dpuis,ptrs,darg)
1124a     as ((dpuis′,ptrs′),(dplan,ptrs′darg′))
1124a        dpuis′ ⊆ dpuis
1124b     ∧ darg′ = darg
1124c     ∧ P_dnm_i(script,anms,dpuis,ptrs,darg),((dpuis′,ptrs′),(dplan,ptrs′darg′))

1121  P_dnm_i: ((Script_dnm_i×ANM-set×DP_UI-set×Script_Ptr-set×D_ARG_dnm_i)
1121                 ×(DP_UI-set×Script_dnm_i_Ptr-set×D_RES_dnm_i)) → Bool
1121  P_dnm_i((script_dnm_i,anms,dpuis,ptrs,darg_dnm_i),(dp_uis′,ptrs′,dres)) ≡ ...
```

10.7.10.3 The derived_planner$_{nm_j}$ Behaviour

This behaviour is an "almost carbon copy" of the derived_planner$_{nm_j}$ behaviour. Thus Items 1125–1125k are "almost the same" as Items 1106– 1106(f)v on page 250.

1125. The derived_planner behaviour is otherwise as follows:

a. The derived_planner obtains, from the derived server, its time stamped master argument, cf. Item 1099 on page 248;
b. it then invokes the derived urban planning function;
c. the time-stamped result is offered to the plan repository;
d. if the result is OK as a final result,
e. then the behaviour is stopped;
f. otherwise
g. the derived planner inquires the derived planner index generator as for such derived planner identifiers which are not used;
h. the derived planner behaviour is the resumed with the appropriately updated programmable script pointer attribute, in parallel with
i. the distributed parallel composition of the parallel behaviours of the derived servers
j. and the derived planners
k. designated by the derived planner identifiers transcribed into (*nm_dps_ui*) derived server, respectively into (*nm_dp_ui*) derived planner names. For these transcription maps we refer to Sect. **10.2.12** on page 219, Item 981 on page 220.

```
value
1106   derived_planner_dnm_i(uid,mereo,(script_dnm_i,anms,puis))(ptrs) ≡
1106a    let (t,((daux_dnm_i,dreq_dnm_i),(plans,ahist))) = s_p_ch[ uid ] ? in
1106b    let ((dpuis′,ptrs′),dres_dnm_i) = derived_urban_planning_dnm_i(script_dnm_i,anms,dpuis,ptrs) in
1106c    p_pr_ch[ uid ] ! dres_dnm_i ;
1106d    if completed(dres_dnm_i)
```

1106e **then** init_der_serv_planrs(uid,dpuis′) **assert:** ptrs′ = {}
1106f **else**
1106(f)i init_der_serv_plans(uid,dpuis′)
1106(f)ii ‖ derived_planner(uid,mereo,(script$_{dnm_i}$,anms,puis))(ptrs′)
1106 **end end end**

10.8 Initialisation of The Urban Space Analysis & Planning System

Section **10.5** presents a *compiler* from *structures* and *parts* to *behaviours*. This section presents an initialisation of some of the behaviours. First we postulate a global *universe of discourse*, *uod*. Then we summarise the global values of *parts* and *part names*. This is followed by a summaries of *part qualities* – in four subsections: a summary of the global values of unique identifiers; a summary of channel declarations; the system as it is initialised; and the system of derived servers and planners as they evolve.

10.8.1 Summary of Parts and Part Names

value
932 on page 215 *uod* : UoD
933 on page 215 *clk* : CLK = obs_CLK(*uod*)
934 on page 215 *tus* : TUS = obs_TUS(*uod*)
935 on page 215 *ans* : A$_{anm_i}$**-set**, i:[1..n] = { obs_A$_{anm_i}$(aa) | aa∈(obs_AA(*uod*)), i:[1..n]}
936 on page 215 *ad* : AD = obs_AD(obs_AA(*uod*))
937 on page 215 *mps* : MPS = obs_MPS(obs_MPA(*uod*))
938 on page 215 *mp* : MP = obs_MP(obs_MPA(*uod*))
939 on page 215 *dpss* : DPS$_{nm_i}$**-set**, i:[1..p] =
939 on page 215 { obs_DPS$_{nm_i}$(dpc$_{nm_i}$) |
939 on page 215 dpc$_{nm_i}$:DPC$_{nm_i}$•dpc$_{nm_i}$∈obs_DPCS$_{nm_i}$(obs_DPA(*uod*)), i:[1..p] }
940 on page 215 *dps* : DP$_{nm_i}$**-set**, i:[1..p] =
940 on page 215 { obs_DP$_{nm_i}$(dpc$_{nm_i}$) |
940 on page 215 dpc$_{nm_i}$:DPC$_{nm_i}$•dpc$_{nm_i}$∈obs_DPCS$_{nm_i}$(obs_DPA(*uod*)), i:[1..p] }
941 on page 215 *dpxg* : DPXG = obs_DPXG(*uod*)
942 on page 215 *pr* : PR = obs_PR(*uod*)
943 on page 215 *spsps* : (DPS$_{nm_i}$×DP$_{nm_i}$)**-set**, i:[1..p] =
943 on page 215 { (obs_DPS$_{nm_i}$(dpc$_{nm_i}$),obs_DP$_{nm_i}$(dpc$_{nm_i}$)) |
943 on page 215 dpc$_{nm_i}$:DPC$_{nm_i}$•dpc$_{nm_i}$∈ obs_DPCS$_{nm_i}$(obs_DPA(*uod*)), i:[1..p] }

10.8.2 Summary of of Unique Identifiers

value
958 on page 218 clk_{ui} : CLK_UI = uid_CLK(*uod*)
959 on page 218 tus_{ui} : TUS_UI = uid_TUS(*uod*)
960 on page 218 $a_{ui}s$: A_UI**-set** = {uid_A(a)|a:A•a ∈ *ans*}
961 on page 218 ad_{ui} : AD_UI = uid_AD(*ad*)

962 on page 218 mps_{ui} : MPS_UI = uid_MPS(mps)
963 on page 218 mp_{ui} : MP_UI = uid_MP(mp)
964 on page 218 $dps_{ui}s$: DPS_UI-**set** = {uid_DPS(dps)|dps:DPS•dps ∈ $dpss$}
965 on page 218 $dp_{ui}s$: DP_UI-**set** = {uid_DP(dp)|dp:DP•dp ∈ dps}
966 on page 218 $dpxg_{ui}$: DPXG_UI = uid_DPXG($dpxg$)
967 on page 218 pr_{ui} : PR_UI = uid_PR(pr)

967 on page 218 $s_{ui}s$: (MPS_UI|DPS_UI)-**set** = {mps_{ui}} ∪ $dps_{ui}s$
969 on page 218 $p_{ui}s$: (MP_UI|DP_UI)-**set** = {mp_{ui}} ∪ $dp_{ui}s$
970 on page 218 $sips$: (DPS_UI×DP_UI)-**set** = {(uid_DPS(dps),uid_DP(dp))|(dps,dp):(DPS×DP)•(dps,dp)∈sps}
971 on page 219 si_pi_m : DPS_UI $\xrightarrow[m]{}$ DP_UI = [uid_DPS(dps)↦uid_DP(dp)|(dps,dp):(DPS×DP)•(dps,dp)∈sps]
972 on page 219 pi_si_m : DP_UI $\xrightarrow[m]{}$ DPS_UI = [uid_DP(dp)↦uid_DPS(dps)|(dps,dp):(DPS×DP)•(dps,dp)∈sps]

10.8.3 Summary of Channels

channel
1053 on page 236 clk_ch:CLK_MSGch-clk-010
1054 on page 237 {tus_a_ch[a_ui]:TUS_MSG|a_ui:A_UI•a_ui ∈ $a_{ui}s$} ch-tus-a-000
1056 on page 237 tus_mps_ch:TUS_MSGch-tus-mps-000
1057 on page 237 {a_ad_ch[a_ui]:A_MSG|a_ui:A_UI•a_ui ∈ $a_{ui}s$}ch-a-ad-000
1059 on page 238 {ad_s_ch[s_ui]|s_ui:(MPS_UI|DPS_UI)•s_ui ∈{mps_{ui}}∪$dps_{ui}s$}:AD_MSGch-ad-dps-000
1061 on page 238 mps_mp_ch:MPS_MSGch-mps-mp-000
1063 on page 239 {p_pr_ch[p_ui]:PLAN_MSG|p_ui:(MP_UI|DP_UI)•p_ui ∈ $p_{ui}s$} ch-mp-pr-000
1065 on page 239 {p_dpxg_ch[ui]:DPXG_MSG|ui:(MP_UI|DP_UI)•ui ∈ $p_{ui}s$} ch-mp-ix-000
1067 on page 239 {pr_s_ch[ui]:PR_MSGd|ui:(MPS_UI|DPS_UI)•ui ∈ $s_{ui}s$}ch-pr-dps-000
1069 on page 240 {dps_dp_ch[ui]:DPS_MSG$_{nm_j}$|ui:DPS_UI•ui ∈ $dps_{ui}s$}ch-dps-dp-000

10.8.4 The Initial System

1078c on page 241 urb_spa(uid_TUS(tus),mereo_TUS(tus))(attr_Pts(tus))
‖
1071c on page 240 clock(uid_CLK(clk),mereo_CLK(clk))(attr_T(clk))
‖
1085 on page 243 ‖ {analyser$_{ui_i}$(uid_A(a$_{ui_i}$),mereo_A(a$_{ui_i}$))(ana$_{anm_i}$) | ui_i:A_UID • ui_i ∈ $a_{ui}s$}
‖
1071c on page 240 ana_dep(ui_A(ad),mereo_A(ad))(attr_AHist(ad))
‖
1096c on page 246 plan_rep(plans)(attr_AllDPUIs(pr),attr_UsedDPUIs(pr))
‖
1094c on page 245 dpxg_beh(uid_DPXG($dpxg$),mereo_DPXG($dpxg$))(all_dpuis,used_dpuis)
‖
1098c on page 247 master_server(uid_MPS(mps),mereo_MPS(mps))(attr_mAUX(mps),attr_mREQ(mps))
‖
1101c on page 249 master_planner(uid_MP(mp),mereo_MP(mp),
1101c on page 249 (attr_Script(mp),attr_ANms(mp),attr_DPUIs(mp)))(attr_Script_Ptrs(mp))

10.8.5 The Derived Planner System

1117c on page 251 { derived_server$_{dps_{nm_j}}$
1117c on page 251 (uid_DPS(dps_{nm_j}),mereo_DPS(dps_{nm_j}))(attr_dAUX(dps_{nm_j}),attr_dREQ(dps_{nm_j}))
‖
1120c on page 253 derived_planner(uid_DP($dpnm_j$),mereo_DP($dpnm_j$),
1120c on page 253 (attr_Script(dp_{nm_j}),attr_ANms(dp_{nm_j}),attr_DPUIs(dp_{nm_j})))
1120c on page 253 | j:[1..p] }

10.9 Further Work

10.9.1 Reasoning About Deadlock, Starvation, Live-lock and Liveness

The current author is quite unhappy about the way in which he has defined the urban planning, oracle and repository behaviours. Such issues as which invariants are maintained across behaviours are not addressed. In fact, it seems to be good practice, following Dijkstra, Lamport and others, to formulate appropriate such invariants and only then "derive" behaviour definitions accordingly. In a rewrite of this research note, if ever, into a proper paper, the current author hopes to follow proper practices. He hopes to find younger talent to co-author this effort.

10.9.2 Document Handling

I may appear odd to the reader that I now turn to document handling. One central aspect of urban planning, strange, perhaps, to the reader, is that of handling the "zillions upon zillions" of documents that enter into and accrue from urban planning. If handling of these documents is not done properly a true nightmare will occur. So we shall briefly examine the urban planning document situation ! From that we conclude that we must first try understand:

- **What do we mean by a document?**

10.9.2.1 Urban Planning Documents

The urban planning functions and the urban planning behaviours, including both the base and the n derived variants, rely on documents. These documents are **created**, **edited**, **read**, **copied**, and, eventually, **shredded** by urban-planners. Editing documents result in new versions of "the same" document. While a document is being **edited** or **read** we think of it as not being **accessible** to other urban-planners. If urban-planners need to read a latest version of a document while that version is subject to editing by another urban planner, copies must first be made, before editing, one for each "needy" reader. Once, editing has and readings have finished, the "reader" copies need, or can, be shredded.

10.9.2.2 A Document Handling System

In Chapter 9 we sketch[ed] a document handling system domain.[82] That is, not a document handling software system, not even requirements for a document handling software system, but just a description which, in essence, models documents and urban planners' actions on documents. (The urban planners are referred to as document handlers.) The description further models two 'aggregate' notions: one of 'handler management', and one of 'document archive'. Both seem necessary in order to "sort out" the granting of document access rights (that is, permissions to perform operations on documents), and the creation and shredding of documents, and in order to avoid dead-locks in access to and handling of documents.

10.9.3 Validation and Verification (V&V)

By **validation** of a document we shall mean: the primarily informal and social process of checking that the document description meets customer expectations.

Validation serves to get the right product.

By **verification** of a document we shall mean: the primarily formal, i.e., mathematical process of checking, testing and formal proof that the model, which the document description entails, satisfies a number of properties.

Verification serves to get the product right.

By **validation of the urban planning model** of this document we shall understand the social process of explaining the model to urban planning stakeholders, to obtain their reaction, and to possibly change the model according to stakeholder objections.

By **verification of the urban planning model** of this document we shall understand the formal process, based on formalisations of the argument and result types of the description, of testing, model checking and formally proving properties of the model.

MORE TO COME

10.9.4 Urban Planning Project Management

In this research note we have focused on the urban planning project behaviours, their interactions, and their information "passing". Usually publications about urban planning: research papers, technical papers, survey papers, etcetera, focus on specific "functions". In this research note we do not. We focus instead on what we can say about the domain of urban planning: the fact, or the possibility, that an initial, a core, here referred to as a base, urban planning effort (i.e., project, hence behaviour) can "spew off", generate, a number of (derived, i.e., in some sense subsidiary), more specialised, urban planning projects.

10.9.4.1 Urban Planning Projects

We think of a comprehensive urban planning project as carried out by urban planners. As is evident from the model the project consists of one base urban planning project and up to *n* derived urban planning projects. The urban planners involved in these projects are professionals in the areas of planning:

[82] I had, over the years, since mid 1990s, reflected upon the idea of *"what is a document ?"*. A most recent version, as I saw it in 2017, was "documented" in Chapter 7 [58]. But, preparing for my work, at TongJi University, Shanghai, September 2017, I reworked my earlier notes [58] into what is now Chapter 9.

- master urban planning issues:
  - geodesy,
  - geotechniques,
  - meteorology,
- master urban plans:
  - cartography,
  - cadestral matters,
  - zoning;
- derived urban planning issues:
  - industries,
  - residential and shopping,
  - apartment buildings,
  - villas,
  - recreational,
  - etcetera;
- technological infrastructures:
  - transport,
  - electricity,
  - telecommunications,
  - gas,
  - water,
  - waste,
  - etcetera;
- societal infrastructures:
  - health care,
  - schools,
  - police,
  - fire brigades,
  - etcetera;
- etcetera, etcetera, etcetera !

To anyone with any experience in getting such diverse groups and individuals of highly skilled professionals to work together it is obvious that some form of management is required. The term 'comprehensive' was mentioned above. It is meant to express that the comprehensive urban planning project is the only one "dealing" with a given geographic area, and that no other urban planning projects "infringe" upon it, that is, "deal" with sub-areas of that given geographic area.

10.9.4.2 Strategic, Tactical and Operational Management

We can distinguish between

- strategic,
- tactical and
- operational

management.

10.9.4.2.1 Project Resources

But first we need take a look at the **resources** that management is charged with:

- the urban planners, i.e., humans,
- time,
- finances,
- office space,
- support technologies: computing etc.,
- etcetera.

10.9.4.2.2 Strategic Management

By **strategic management** we shall understand the analysis and decisions of, and concerning, scarce resources: people (skills), time, monies: their deployment and trade-offs.

10.9.4.2.3 Tactical Management

By **tactical management** we shall understand the analysis and decisions with respect to budget and time plans, and the monitoring and control of serially reusable resources: office space, computing.

10.9.4.2.4 Operational Management

By **operational management** we shall understand the monitoring and control of the enactment, progress and completion of individual deliverables, i.e., documents, the quality (adherence to

"standards", fulfillment of expectations, etc.) of these documents, and the day-to-day human relations.

10.9.4.3 Urban Planning Management

The above (*strategic, tactical & operational management*) translates, in the context of *urban planning*, into:

TO BE WRITTEN

Chapter 11
Swarms of Drones [November–December 2017]

Contents

11.1 An Informal Introduction 263
11.1.1 Describable Entities 263
11.1.1.1 The Endurants: Parts 263
11.1.1.2 The Perdurants 264
11.1.2 The Contribution of [48] 264
11.1.3 The Contribution of This Report 264
11.2 Entities, Endurants 264
11.2.1 Parts, Atomic and Composite, Sorts, Abstract and Concrete Types 265
11.2.1.1 Universe of Discourse 265
11.2.1.2 The Enterprise 266
11.2.1.3 From Abstract Sorts to Concrete Types 266
11.2.1.3.1 The Auxiliary Function xtr_Ds: 266
11.2.1.3.2 Command Center 267
11.2.1.3.3 Command Center Decomposition 267
11.2.2 Unique Identifiers 267
11.2.2.1 The Enterprise, the Aggregates of Drones and the Geography 267
11.2.2.2 Unique Command Center Identifiers 268
11.2.2.3 Unique Drone Identifiers 268
11.2.2.3.1 Auxiliary Function: xtr_dis: 268
11.2.2.3.2 Auxiliary Function: xtr_D: 269
11.2.3 Mereologies 269
11.2.3.1 Definition 269
11.2.3.2 Origin of the Concept of Mereology as Treated Here 269
11.2.3.3 Basic Mereology Principle 269
11.2.3.4 Engineering versus Methodical Mereology 270
11.2.3.5 Planner Mereology 270
11.2.3.6 Monitor Mereology 271
11.2.3.7 Actuator Mereology 271
11.2.3.8 Enterprise Drone Mereology 272
11.2.3.9 'Other' Drone Mereology 272
11.2.3.10 Geography Mereology 273
11.2.4 Attributes 273
11.2.4.1 The Time Sort 273
11.2.4.2 Positions 274
11.2.4.2.1 A Neighbourhood Concept 274
11.2.4.3 Flight Plans 274
11.2.4.4 Enterprise Drone Attributes 275
11.2.4.4.1 Constituent Types 275
11.2.4.4.2 Attributes 276
11.2.4.4.3 Enterprise Drone Attribute Categories: 276
11.2.4.5 'Other' Drones Attributes 276
11.2.4.5.1 Constituent Types 276
11.2.4.5.2 Attributes 276
11.2.4.6 Drone Dynamics 277

11.2.4.7 Drone Positions 277
11.2.4.8 Monitor Attributes 277
11.2.4.9 Planner Attributes 278
11.2.4.9.1 Swarms and Businesses: 278
11.2.4.9.2 Planner Directories: 278
11.2.4.10 Actuator Attributes 279
11.2.4.11 Geography Attributes 280
11.2.4.11.1 Constituent Types: 280
11.2.4.11.2 Attributes 280
11.3 Operations on Universe of Discourse States 280
11.3.1 The Notion of a State 281
11.3.2 Constants 281
11.3.3 Operations 281
11.3.3.1 A Drone Transfer 281
11.3.3.2 An Enterprise Drone Changing Course 282
11.3.3.3 A Swarm Splitting into Two Swarms 282
11.3.3.4 Two Swarms Joining to form One Swarm 282
11.3.3.5 Etcetera 282
11.4 Perdurants 283
11.4.1 System Compilation 283
11.4.1.1 The Compile Functions 283
11.4.1.2 Some CSP Expression Simplifications 285
11.4.1.3 The Simplified Compilation 285
11.4.2 An Early Narrative on Behaviours 286
11.4.2.1 Either Endurants or Perdurants, Not Both ! 286
11.4.2.2 Focus on Some Behaviours, Not All ! 286
11.4.2.3 The Behaviours – a First Narrative 287
11.4.3 Channels 287
11.4.3.1 The Part Channels 288
11.4.3.1.1 General Remarks: 288
11.4.3.1.2 Part Channel Specifics 288
11.4.3.2 Attribute Channels, General Principles 290
11.4.3.3 The Case Study Attribute Channels 290
11.4.3.3.1 'Other' Drones: 290
11.4.3.3.2 Enterprise Drones: 290
11.4.3.3.3 Geography: 291
11.4.4 The Atomic Behaviours 291
11.4.4.1 Monitor Behaviour 291
11.4.4.2 Planner Behaviour 292
11.4.4.2.1 The Auxiliary transfer Function 292
11.4.4.2.2 The Auxiliary flight_planning Function 293
11.4.4.3 Actuator Behaviour 294
11.4.4.4 'Other' Drone Behaviour 295
11.4.4.5 Enterprise Drone Behaviour 296
11.4.4.6 Geography Behaviour 299
11.5 Conclusion 300

We speculate[83] on a domain of *swarms* and *drones monitored* and *controlled* by a *command center* in some *geography*. Awareness of swarms is registered only in an enterprise command center. We think of these swarms of drones as an enterprise of either package deliverers, crop-dusters, insect sprayers, search & rescuers, traffic monitors, or wildfire fighters – or several of these, united in a notion of *an enterprise* possibly consisting of of "disjoint" *businesses*. We analyse & describe the properties of these phenomena as endurants and as perdurants: parts one can observe and behaviours that one can study. We do not yet examine the problem of drone air traffic management[84]. The analysis & description of this postulated domain follows the principles, techniques and tools laid down in [48].

---

[83] A young researcher colleague, Dr. Yang ShaoFa, of the Software Institute of the Chinese Academy of Sciences in Beijing, at our meeting in Beijing, early November 2017, told me that he was then about to get involved in algorithms for drone maneuvering. So, true to me thinking, that, in order to reflect on such algorithms, one ought try understand the domain. So I sketched the model of this chapter in the week, attending the ICFEM'2017 conference in Xi'An, and presented the model to Dr. Yang upon my return to Beijing.

[84] www.nasa.gov/feature/ames/first-steps-toward-drone-traffic-management, www.sciencedirect.com/science/article/pii/S2046043016300260

11.1 An Informal Introduction

11.1.1 Describable Entities

11.1.1.1 The Endurants: Parts

In the universe of discourse we observe *endurants,* here in the form of parts, and *perdurants,* here in the form of behaviours.

The parts are *discrete endurants,* that is, can be seen or touched by humans, or that can be conceived as an abstraction of a discrete part.

We refer to Fig. 11.1.

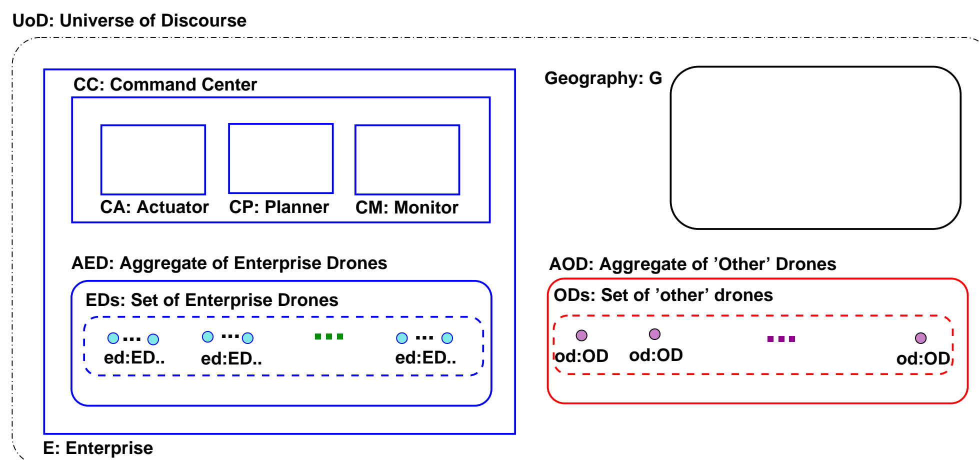


Fig. 11.1 Universe of Discourse

There is a *universe of discourse*, uod:UoD. The universe of discourse embodies: an *enterprise*, e:E. The enterprise consists of an *aggregate of enterprise drones*, aed:AED (which consists of a set, eds:EDs, of enterprise drones). and a *command center*, cc:CC; The universe of discourse also embodies a *geography*, g:G. The universe of discourse finally embodies an *aggregate of 'other' drones*, aod:AOD (which consists of a set, ods:ODs, of these 'other' drones). A *drone* is an *unmanned aerial vehicle.*[85] We distinguish between *enterprise drones*, ed:ED, and *'other' drones*, od:OD. The pragmatics of the enterprise swarms is that of providing enterprise drones for one or more of the following kinds of *businesses:*[86] delivering parcels (mail, packages, etc.)[87], crop dusting[88], aerial spraying[89], wildfire fighting[90], traffic control[91], search and rescue[92], etcetera. A notion of *swarm* is introduced. A swarm is a concept. As a concept a swarm is a set of drones. We

[85] Drones are also referred to as UAVs.
[86] http://www.latimes.com/business/la-fi-drone-traffic-20170501-htmlstory.html
[87] https://www.amazon.com/Amazon-Prime-Air/b?node=8037720011 and https://www.digitaltrends.com/cool-tech/amazon-prime-air-delivery-drones-history-progress/
[88] http://www.uavcropdustersprayers.com/, http://sprayingdrone.com/
[89] https://abjdrones.com/commercial-drone-services/industry-specific-solutions/agriculture/
[90] https://www.smithsonianmag.com/videos/category/innovation/drones-are-now-being-used-to-battle-wildfires/
[91] https://business.esa.int/sites/default/files/Presentation%20on%20UAV%20Road%20Surface%20Monitoring%20and%20Traffic%20Information_0.pdf
[92] http://sardrones.org/

associate swarms with businesses. A business has access to one or more swarms. The enterprise *command center*, cc:CC, can be seen as embodying three kinds of functions: a *monitoring* service, cm:CM, whose function it is to know the locations and dynamics of all drones, whether enterprise drones or 'other' drones; a *planning* service, cp:CP, whose function it is to plan the next moves of all that enterprise's drones; and an *actuator* service, ca:CA, whose functions it is to guide that enterprise's drones as to their next moves. The swarm concept "resides" in the command planner.

11.1.1.2 The Perdurants

The perdurants are entities for which only a fragment exists if we look at or touch them at any given snapshot in time, that is, were we to freeze time we would only see or touch a fragment of the perdurant.

The major ***

MORE TO COME

11.1.2 The Contribution of [48]

The major contributions of [48] are these: a methodology[93] for analysing & describing manifest domains[94], where the methodology builds on an *ontological principle* of viewing the domains as consisting of *endurants* and *perdurants*. Endurants possess properties such as *unique identifiers, mereologies,* and *attributes*. Perdurants are then analysed & described as either *actions, events,* or *behaviours*. The *techniques* to go with the ***

MORE TO COME

The *tools* are ***

MORE TO COME

11.1.3 The Contribution of This Report

TO BE WRITTEN

We relate our work to that of [124].

• • •

The main part of this report is contained in the next three sections: endurants; states, constants, and operations on states; and perdurants.

11.2 Entities, Endurants

By an *entity* we shall understand *a phenomenon, i.e., something that can be observe d, i.e., be seen or touched by humans, or that can be conceived as an abstraction of an entity. We further demand that an entity can be objectively described.*

[93] By a *methodology* we shall understand a set of *principles* for `selecting` and `applying` a number of *techniques*, using *tools*, to – in this case – analyse & describe a domain.

[94] A manifest domain is a human- and artifact-assisted arrangement of endurant, that is spatially "stable", and perdurant, that is temporally "fleeting" entities. Endurant entities are either parts or components or materials. Perdurant entities are either actions or events or behaviours.

By an *endurant* we shall understand *an entity that can be observed or conceived and described as a "complete thing" at no matter which given snapshot of time. Were we to "freeze" time we would still be able to observe the entire endurant.*

11.2.1 Parts, Atomic and Composite, Sorts, Abstract and Concrete Types

By a *discrete endurant* we shall understand *an endurant which is separate, individual or distinct in form or concept.*

By a *part* we shall understand *a discrete endurant which the domain engineer chooses to endow with internal qualities such as unique identification, mereology, and one or more attributes.* We shall define the concepts of unique identifier, mereology and attribute later in this report.

Atomic parts are *those which, in a given context, are deemed to not consist of meaningful, separately observable proper sub-parts.*

Sub-parts are parts.

Composite parts are *those which, in a given context, are deemed to indeed consist of meaningful, separately observable proper sub-parts.*

By a *sort* we shall understand *an abstract type.*

By a *type* we shall here understand *a set of values "of the same kind"* – where we do not further define what we mean by *the same kind"*.

By an *abstract type* we shall understand *a type about whose values we make no assumption* [as to their atomicity or composition.

By a *concrete type* we shall understand *a type about whose values we are making certain assumptions as to their atomicity or composition, and, if composed then how and from which other types they are composed.*

11.2.1.1 Universe of Discourse

By a *universe of discourse*e shall understand *that which we can talk about, refer to and whose entities we can name.* Included in that universe is the *geography*. By *geography* we shall understand a section of the globe, an area of land, its geodesy, its meteorology, etc.

1126. In the **U**niverse **o**f **D**iscourse we can observe the following parts:
   a. an atomic **G**eography,
   b. a composite **E**nterprise,
   c. and an aggregate of '**O**ther'[95] **D**rones.

```
type
1126   UoD, G, E, AOD
value
1126a  obs_G: UoD → G
1126b  obs_E: UoD → E
1126c  obs_AOD: UoD → AOD
```

[95] We apologize for our using the term 'other' drones. These 'other' drones are not necessarily adversary or enemy drones. They are just there – coexisting with the enterprise drones.

11.2.1.2 The Enterprise

1127. From an enterprise one can observe:
    a. a(n enterprise) command center. and
    b. an aggregate of enterprise drones.

```
type
1127a  CC
1127a  AED
value
1127a  obs_CC: E → CC
1127b  obs_AED: E → AED
```

11.2.1.3 From Abstract Sorts to Concrete Types

1128. From an *aggregate of enterprise drones*, AED, we can observe a possibly empty set of drones, EDs
1129. From an *aggregate of 'other' drones*, AOD, we can observe a possibly empty set, ODs, of *'other' drones*.

```
type
1128 ED
1128 EDs = ED-set
1129 OD
1129 ODs = OD-set
value
1128 obs_EDs: AED → EDs
1129 obs_ODs: AOD → ODs
```

Drones, whether 'other' or 'enterprise', are considered atomic.

11.2.1.3.1 The Auxiliary Function xtr_Ds:

We define an auxiliary function, xtr_Ds.

1130. From the universe of discourse we can extract all its drones;
1131. similarly from its enterprise;
1132. similarly from the aggregate of enterprise drones; and
1133. from an aggregate of 'other' drones.

```
1130  xtr_Ds: UoD → (ED|OD)-set
1130  xtr_Ds(uod) ≡
1130     ∪{xtr_Ds(obs_AED(obs_E(uod)))} ∪ xtr_Ds(obs_AOD(uod))
1131  xtr_Ds: E → ED-set
1131  xtr_Ds(e) ≡ xtr_Ds(obs_AED(e))
1132  xtr_Ds: AED → ED-set
1132  xtr_Ds(aed) ≡ obs_EDs(obs_EDs(aed))
1133  xtr_Ds: AOD → OD-set
1133  xtr_Ds(aod) ≡ obs_ODs(aod)
```

1134. In the universe of discourse a drone cannot be both among the enterprise drones and among the 'other' drones.

axiom
```
1134  ∀ uod:UoD,e:E,aed:ES,aod:AOD •
1134     e=obs_E(uod)∧aed=obs_AED(e)∧aod:obs_AOD(uod)
1134        ⇒ xtr_Ds(aed) ∩ xtr_Ds(aod) ={}
```

The functions are partial as the supplied swarm identifier may not be one of the universe of discourse, etc.

11.2.1.3.2 Command Center

A Simple Narrative Figure 11.1 on page 263 shows a graphic rendition of a space of interest. The command center, CC, a composite part, is shown to include three atomic parts: An atomic part, the monitor, CM. It monitors the location and dynamics of all drones. An atomic part, the planner, CP. It plans the next, "friendly", drone movements. The command center also has yet an atomic part, the actuator, CA. It informs "friendly" drones of their next movements. The planner is where "resides" the notion of a enterprise consisting of one or more businesses, where each business has access to zero, one or more swarms, where a swarm is a set of enterprise drone identifiers.
The purpose of the control center is to monitor the whereabouts and dynamics of all drones (done by CM); to plan possible next actions by enterprise drones (done by CP); and to instruct enterprise drones of possible next actions (done by CA).

11.2.1.3.3 Command Center Decomposition

From the composite command center we can observe

1135. the center monitor, CM;
1136. the center planner, CP; and
1137. the center actuator, CA .

type
```
1135  CM
1136  CP
1137  CA
```

value
```
1135  obs_CM: CC → CM
1136  obs_CP: CC → CP
1137  obs_CA: CC → CA
```

11.2.2 Unique Identifiers

Parts are distinguishable through their unique identifiers. A *unique identifier* is a further undefined quantity which we associate with parts such that no two parts of a universe of discourse are identical.

11.2.2.1 The Enterprise, the Aggregates of Drones and the Geography

1138. Although we may not need it for subsequent descriptions we do, for completeness of description, introduce unique identifiers for parts and sub-parts of the universe of discourse:
    a. Geographies, g:G, have unique identification.

b. Enterprises, e:E, have unique identification.
c. Aggregates of enterprise drones, aed:AED, have unique identification.
d. Aggregates of 'other' drones, aod:AOD, have unique identification.
e. Command centers, cc:CC, have unique identification.

```
type
1138    GI, EI, AEDI, AODI, CCI
value
1138a   uid_G: G → GI
1138b   uid_E: E → EI
1138c   uid_AED: AED → AEDI
1138d   uid_OD: AOD → AODI
1138e   uid_CC: CC → CCI
```

11.2.2.2 Unique Command Center Identifiers

1139. The monitor has a unique identifier.
1140. The planner has a unique identifier.
1141. The actuator has a unique identifier.

```
type
1139   CMI
1140   CPI
1141   CAI
value
1139   uid_CM: CM → CMI
1140   uid_CP: CP → CPI
1141   uid_CA: CA → CAI
```

11.2.2.3 Unique Drone Identifiers

1142. Drones have unique identifiers.
   a. whether enterprise or
   b. 'other' drones

```
type
1142   DI = EDI | ODI
value
1142a   uid_ED: ED → EDI
1142b   uid_OD: OD → ODI
```

11.2.2.3.1 Auxiliary Function: xtr_dis:

1143. From the aggregate of enterprise drones;
1144. From the aggregate of 'other' drones;
1145. and from the two parts of a universe of discourse: the enterprise and the 'other' drones.

```
value
1143   xtr_dis: AED → DI-set
1143   xtr_dis(aed) ≡ {uid_ED(ed)|ed:ED•ed ∈ obs_EDs(aed)}
1144   xtr_dis: AOD → DI-set
1144   xtr_dis(aod) ≡ {uid_D(od)|od:OD•od ∈ obs_ODs(aod)}
```

1145 xtr_dis: UoD → DI-**set**
1145 xtr_dis(uod) ≡ xtr_dis(obs_AED(uod)) ∪ xtr_dis(obs_AOD(uod))

11.2.2.3.2 Auxiliary Function: xtr_D:

1146. From the universe of discourse, given a drone identifier of that space, we can extract the identified drone;
1147. similarly from the enterprise;
1148. its aggregate of enterprise drones; and
1149. and from its aggregate of 'other' drones;

1146 xtr_D: UoD → DI $\xrightarrow{\sim}$ D
1146 xtr_D(uod)(di) ≡ **let** d:D • d ∈ xtr_Ds(uod)∧uid_D(d)=di **in** d **end**
1146 **pre**: di ∈ xtr_dis(soi)
1147 xtr_D: E → DI $\xrightarrow{\sim}$ D
1147 xtr_D(e)(di) ≡ **let** d:D • d ∈ xtr_Ds(obs_ES(e))∧uid_D(d)=di **in** d **end**
1147 **pre**: di ∈ xtr_dis(e)
1148 xtr_D: AED → DI $\xrightarrow{\sim}$ D
1148 xtr_D(aed)(di) ≡ ≡ **let** d:D • d ∈ xtr_Ds(aed)∧uid_D(d)=di **in** d **end**
1148 **pre**: di ∈ xtr_dis(es)
1149 xtr_D: AOD → DI $\xrightarrow{\sim}$ D
1149 xtr_D(aod)(di) ≡ **let** d:D • d ∈ xtr_Ds(aod)∧uid_D(d)=di **in** d **end**
1149 **pre**: di ∈ xtr_dis(ds)

11.2.3 Mereologies

11.2.3.1 Definition

Mereology is the study and knowledge of parts and their relations (to other parts and to the "whole") [75].

11.2.3.2 Origin of the Concept of Mereology as Treated Here

We shall [thus] deploy the concept of mereology as advanced by the Polish mathematician, logician and philosopher Stanisław Léschniewski. Douglas T. ("Doug") Ross[96] also contributed along the lines of our approach [142] – hence [51] is dedicated to Doug.

11.2.3.3 Basic Mereology Principle

The basic principle in modelling the mereology of a any universe of discourse is as follows: Let p' be a part with unique identifier p'_{id}. Let p be a sub-part of p' with unique identifier p_{id}. Let the immediate sub-parts of p be $p_1, p_2, \ldots, p_n$ with unique identifiers $p_{1_{id}}, p_{2_{id}}, \ldots, p_{n_{id}}$. That p has

[96] Doug Ross is the originator of the term CAD for *computer aided design*, of APT for *Automatically Programmed Tools*, a language to drive numerically controlled manufacturing, and also SADT for *Structure Analysis and Design Techniques*

mereology $(p'_{id}, \{p_{1_{id}}, p_{2_{id}}, \ldots, p_{n_{id}}\})$. The parts p_j, for $1 \leq j \leq n$ for $n \geq 2$, if atomic, have mereologies $(p_{id}, \{p_{1_{id}}, p_{2_{id}}, \ldots, p_{j-1_{id}}, p_{j+1_{id}}, \ldots, p_{n_{id}}\})$ – where we refer to the second term in that pair by m; and if composite, have mereologies $(p_{id}, (m, m'))$, where the m' term is the set of unique identifiers of the sub-parts of p_j.

11.2.3.4 Engineering versus Methodical Mereology

We shall restrict ourselves to an engineering treatment of the mereology of our universe of discourse. That is in contrast to a strict, methodical treatment. In a methodical description of the mereologies of the various parts of the universe of discourse one assigns a mereology to every part: to the enterprise, the aggregate of 'other' drones and the geography; to the command center of the enterprise and its aggregate of drones; to the monitor, the planner and the actuator of the command center; to the drones of the aggregate of enterprise drones, and to the drones of the aggregate of 'other' drones. We shall "shortcut" most of these mereologies. The reason is this: The *pragmatics* of our attempt to model *drones*, is rooted in our interest in the interactions between the command center's monitor and actuator and the enterprise and 'other' drones. For "completeness" we also include interactions between the geography's meteorology and the above command center and drones. The mereologies of the enterprise, E, the enterprise aggregate of drones AED, and the set of (enterprise) drones, EDs, do not involve drone identifiers. The only "thing" that the monitor and actuator are interested in are the drone identifiers. So we shall thus model the mereologies of our universe of discourse by omitting mereologies for the enterprise, the aggregates of drones, the sets of these aggregates, and the geography, and only describe the mereologies of the monitor, planner and actuator, the enterprise drones and the 'other' drones.

11.2.3.5 Planner Mereology

1150. The planner mereology reflects the center planners awareness[97] of the monitor, the actuator,, and the geography of the universe of discourse.
1151. The plannner mereology further reflects that a *eureka*[98] is provided by, or from, an outside source reflected in the autonomous attribute Cmdl. The value of this attribute changes at its own volition and ranges over commands that directs the planner to perform either of a number of operations.

Eureka examples are: calculate and effect a new flight plan for one or more designated swarms of a designated business; effect the transfer of an enterprise drone from a designated swarm of a business to another, distinctly designated swarm of the same business; etcetera.

```
type
1150  CPM = (CAI × CMI × GI) × Eureka
1151  Eureka == mkNewFP(BI×SI-set×Plan)
1151        | mkChgDB(fsi:SI×tsi:SI×di×DI)
1151        | ...
value
1150  mereo_CP: CP → CPM
1151  Plan = ...
```

[97] That "awareness" includes, amongst others, the planner obtaining information from the monitor of the whereabouts of all drones and providing the actuator with directives for the enterprise drones — all in the context of the *land* and "its" *meteorology*.

[98] "Eureka" comes from the Ancient Greek word $\epsilon\mu\rho\eta\kappa\alpha$ heúrēka, meaning "I have found (it)", which is the first person singular perfect indicative active of the verb $\epsilon\upsilon\rho\eta\kappa\omega$ heuriskō "I find".[1] It is closely related to heuristic, which refers to experience-based techniques for problem solving, learning, and discovery.

We omit expressing a suitable axiom concerning center planner mereologies. Our behavioural analysis & description of monitoring & control of operations on the space of drones will show that command center mereologies may change.

11.2.3.6 Monitor Mereology

The monitor's mereology reflects its awareness of the drones whose position and dynamics it is expected to monitor.

1152. The mereology of the center monitor is a pair: the set of unique identifiers of the drones of the universe of discourse, and the unique identifier of the center planner.

```
type
1152  CMM = DI-set × CPI
value
1152  mereo_CM: CM → CMM
```

1153. For the universe of discourse it is the case that
   a. the drone identifiers of the mereology of a monitor must be exactly those of the drones of the universe of discourse, and
   b. the planner identifier of the mereology of a monitor must be exactly that of the planner of the universe of discourse.

```
axiom
1153   ∀ uod:UoD,e:E,cc:CC,cp:CP,cm:CM,g:G •
1153      e=obs_E(uod)∧cc=obs_CC(e)∧cp=obs_CP(cc)∧cm=obs_CM(cc) ⇒
1153         let (dis,cpi) = mereo_CM(cm) in
1153a          dis = xtr_dis(uod)
1153b       ∧ cpi = uid_CP(cp) end
```

11.2.3.7 Actuator Mereology

The center actuator's mereology reflects its awareness of the enterprise drones whose position and dynamics it is expected to control.

1154. The mereology of the center actuator is a pair: the set of unique identifiers of the business drones of the universe of discourse, and the unique identifier of the center planner.

```
type
1154  CAM = EDI-set × CPI
value
1154  mereo_CA: CA → CAM
```

1155. For all universes of discourse
   a. the drone identifiers of the mereology of a center actuator must be exactly those of the enterprise drones of the space of interest (of the monitor), and
   b. the center planner identifier of the mereology of a center actuator must be exactly that of the center planner of the command center of the space of interest (of the monitor)

```
axiom
1155  ∀ uod:UoD,e:E,cc:CC,cp:CP,ca:CA •
1155     e=obs_E(uod)∧cc=obs_CC(e)∧cp=obs_CP(cc)∧ca=obs_CA(cc) ⇒
1155        let (dis,cpi) = mereo_CA(ca) in
1155a       dis =  tr_dis(e)
1155b     ∧ cpi = uid_CP(cp) end
```

11.2.3.8 Enterprise Drone Mereology

1156. The mereology of an enterprise drone is the triple of the command center monitor, the command center actuator[99], and the geography.

```
type
1156  EDM = CMI × CAI × GI
value
1156  mereo_ED: ED → EDM
```

1157. For all universes of discourse the enterprise drone mereology satisfies:
   a. the unique identifier of the first element of the drone mereology is that of the enterprise's command monitor,
   b. the unique identifier of the second element of the drone mereology is that of the enterprise's command actuator, and
   c. the unique identifier of the third element of the drone mereology is that of the universe of discourse's geography.

```
axiom
1157  ∀ uod:UoD,e:E,cm:CM,ca:CA,ed:ED,g:G •
1157     e=obs_E(uod)∧cm=obs_CM(obs_CC(e))∧ca=obs_CA(obs_CC(e))
1157   ∧ ed ∈ xtr_Ds(e)∧g=obs_G(uod) ⇒
1157        let (cmi,cai,gi) = mereo_D(ed) in
1157a       cmi = uid_CMM(ccm)
1157b     ∧ cai = uid_CAI(cai)
1157c     ∧ gi = uid_G(g) end
```

11.2.3.9 'Other' Drone Mereology

1158. The mereology of an 'other' drone is a pair: the unique identifier of the monitor and the unique identifier of the geography.

```
type
1158  ODM = CMI × GI
value
1158  mereo_OD: OD → ODM
```

We leave it to the reader to formulate a suitable axiom, cf. axiom 1157.

[99] The command center monitor and the command center actuator and their unique identifiers will be defined in Items 1135, 1137 on page 267, 1139 and 1141 on page 268.

11.2.3.10 Geography Mereology

1159. The geography mereology is a pair[100] of the unique of the unique identifiers of the planner and the set of all drones.

type
1159 GM = CPI × CMI × DI-**set**
value
1159 mereo_G: G → GM

We leave it to the reader to formulate a suitable axiom, cf. axiom 1157 on the facing page.

11.2.4 Attributes

We analyse & describe attributes for the following parts: *enterprise drones* and *'other' drones*, *monitor, planner* and *actuator*, and the *geography*. The attributes, that we shall arrive at, are usually concrete in the sense that they comprise values of, as we shall call them, *constituent* types. We shall therefore first analyse & describe these constituent types. Then we introduce the part attributes as expressed in terms of the constituent types. But first we introduce three notions core notions: time, Sect. **11.2.4.1**, positions, Sect. **11.2.4.2**, and flight plans, Sect. **11.2.4.3**.

11.2.4.1 The Time Sort

1160. Let the special sort identifier $\mathbb{T}$ denote times
1161. and the special sort identifier $\mathbb{TI}$ denote time intervals.
1162. Let identifier time designate a "magic" function whose invocations yield times.

type
1160 $\mathbb{T}$
1160 $\mathbb{TI}$
value
1160 time: **Unit** → $\mathbb{T}$

1163. Two times can not be added, multiplied or divided, but subtracting one time from another yields a time interval.
1164. Two times can be compared: smaller than, smaller than or equal, equal, not equal, etc.
1165. Two time intervals can be compared: smaller than, smaller than or equal, equal, not equal, etc.
1166. A time interval can be multiplied by a real number.

Etcetera.

value
1163 $\ominus$: $\mathbb{T} \times \mathbb{T} \to \mathbb{TI}$
1164 $<,\leq,=,\neq,\geq,>$: $\mathbb{T} \times \mathbb{T} \to$ **Bool**
1165 $<,\leq,=,\neq,\geq,>$: $\mathbb{TI} \times \mathbb{TI} \to$ **Bool**
1166 $\otimes$: $\mathbb{TI} \times$ **Real** $\to \mathbb{TI}$

[100] 30.11.2017: I think !

11.2.4.2 Positions

Positions (of drones) play a pivotal rôle.

1167. Each *position* being designated by
1168. *longitude, latitude* and *altitude*.

```
type
1168  LO, LA, AL
1167  P = LO × LA × AL
```

11.2.4.2.1 A Neighbourhood Concept

1169. Two positions are said to be *neighbours* if the *distance* between them is small enough for a drone to fly from one to the other in one to three minutes' time – for drones flying at a speed below `Mach 1`.

```
value
1169   neighbours: P × P → Bool
```

We leave the neighbourhood proposition further undefined.

11.2.4.3 Flight Plans

A crucial notion of our universe of discourse is that of flight plans.

1170. A *flight plan element* is a pair of a time and a position.
1171. A *flight plan* is a sequence of flight plan elements.

```
type
1170   FPE = 𝕋 × P
1171   FP = FLE*
```

1172. such that adjacent entries in flight plans
    a. record increasing times and
    b. neighbouring positions.

```
axiom
1172   ∀ fp:FP,i:Nat • {i,i+1}⊆indsfp ⇒
1172      let (t,p)=fp[i], (t′,p′)=fp[i+1] in
1172a        t ≤ t′
1172b   ∧ neighbours(p,p′)
1172      end
```

11.2.4.4 Enterprise Drone Attributes

11.2.4.4.1 Constituent Types

1173. Enterprise drones have *positions* expressed, for example, in terms of *longitude, latitude* and *altitude*. [101]
1174. Enterprise drones have *velocity* which is a vector of *speed* and three-dimensional, i.e., spatial, *direction*.
1175. Enterprise drones have *acceleration* which is a vector of *increase/decrease of speed per time unit* and *direction*.
1176. Enterprise drones have orientation which is expressed in terms of three quantities: *yaw, pitch* and *roll*.[102]

We leave *speed, direction* and *increase/decrease per time unit* unspecified.

type
1173 POS = P
1174 VEL = SPEED × DIRECTION
1175 ACC = IncrDecrSPEEDperTimeUnit × DIRECTION
1176 ORI = YAW × PITCH × ROLL
1174 SPEED = ...
1174 DIRECTION = ...
1175 IncrDecrSPEEDperTimeUnit = ...

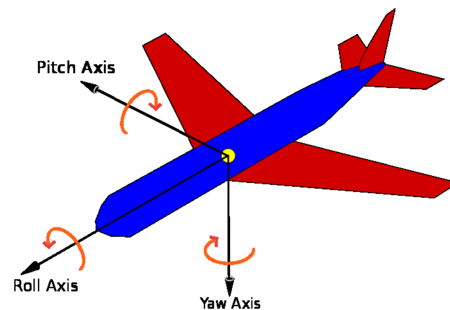


Fig. 11.2 Aircraft Orientation

[101] *Longitude* is a geographic coordinate that specifies the east-west position of a point on the Earth's surface. It is an angular measurement, usually expressed in degrees and denoted by the Greek letter lambda. Meridians (lines running from the North Pole to the South Pole) connect points with the same longitude. *Latitude* is a geographic coordinate that specifies the northsouth position of a point on the Earth's surface. Latitude is an angle (defined below) which ranges from 0° at the Equator to 90° (North or South) at the poles. Lines of constant latitude, or parallels, run eastwest as circles parallel to the equator. *Altitude* or height (sometimes known as depth) is defined based on the context in which it is used (aviation, geometry, geographical survey, sport, and many more). As a general definition, altitude is a distance measurement, usually in the vertical or "up" direction, between a reference datum and a point or object. The reference datum also often varies according to the context.

[102] *Yaw, pitch* and *roll* are seen as symmetry axes of a drone: normal axis, lateral (or transverse) axis and longitudinal (or roll) axis. See Fig. 11.2.

11.2.4.4.2 Attributes

1177. One of the enterprise properties is that of its *dynamics* which is seen as a quadruple of *velocity, acceleration, orientation* and *position*. It is recorded as a reactive attribute.
1178. Enterprise drones follow a flight course, as prescribed in and recorded as a programmable attribute, referred to a the *future flight plan*, FFP.
1179. Enterprise drones have followed a course recorded, also a programmable attribute, as a *past flight plan list*, PFPL.
1180. Finally enterprise drones "remember", in the form of a programmable attribute, the *geography* (i.e., the *area*, the *land* and the *weather*) it is flying over and in !

type
1180 ImG = A×L×W
1177 DYN = s_vel:VEL × s_acc:ACC × s_ori:ORI × s_pos:POS
1178 FPL = FP
1179 PFPL = FP*
value
1177 attr_DYN: ED → DYN
1178 attr_FPL: ED → FPL
1179 attr_PFPL: ED → PFPL
1180 attr_ImG: ED → ImG

Enterprise, as well as 'other' drone, positions must fall within the *Euclidian Point Space* of the geography of the universe of discourse. We leave that as an axiom to be defined – or we could decide that if a drone leaves that space then it is lost, and if drones suddenly "appear, out of the blue", then they are either "brand new", or "reappear".

11.2.4.4.3 Enterprise Drone Attribute Categories:

The position, velocity, acceleration, position and past position list attributes belong to the **reactive** category. The future position list attribute belong to the **programmable** category. Drones have a "zillion" more attributes – which may be introduced in due course.

11.2.4.5 'Other' Drones Attributes

11.2.4.5.1 Constituent Types

The constituent types of 'other' drones are similar to those of some of the enterprise drones.

11.2.4.5.2 Attributes

1181. 'Other' drones have *dynamics*, dyn:DYN.
1182. 'Other' drones "remember", in the form of a programmable attribute, the *immediate geography*, ImG (i.e., the *area*, the *land* and the *weather*) it is flying over and in !

type
1182 A, L, W
1182 ImG = A×L×W
value
1181 attr_DYN: OD → DYN

1182 attr_ImG: OD → ImG

11.2.4.6 Drone Dynamics

1183. By a timed drone dynamics, TiDYN, we understand a quadruplet of *time, position, dynamics* and *immediate geography*.
1184. By a *current drone dynamics* we shall understand a drone identifier-indexed set of timed drone dynamics.
1185. By a *record of [traces of] timed drone dynamics* we shall understand a drone identifier-indexed set of sequences of timed drone dynamics.

type
1183 TiDYN = $\mathbb{T}$ × POS × DYN × ImG
1184 CuDD = (EDI $\overrightarrow{m}$ TiDYN) ∪ (ODI $\overrightarrow{m}$ TiDYN)
1185 RoDD = (EDI $\overrightarrow{m}$ TiDYN*) ∪ (ODI $\overrightarrow{m}$ TiDYN*)

We shall use the notion of *current drone dynamics* as the means whereby the *monitor* ascertains (obtains, by interacting with drones) the dynamics of drones, and the notion of a *record of [traces of] drone dynamics* in the *monitor*.

11.2.4.7 Drone Positions

1186. For all drones whether enterprise or 'other', their positions must lie within the geography of their universe of discourse.

axiom
1186 ∀ uod:UoD,e:E,g:G,d:(ED|OD) •
1186 e = obs_E(uod) ∧ g = obs_G(uod) ∧ d ∈ xtr_Ds(uod) ⇒
1186 **let** eps = attr_EPS(g), (_,_,p) = attr_DYN(d) **in** p ∈ eps **end**

11.2.4.8 Monitor Attributes

The *monitor* "sits between" the *drones* whose dynamics it monitors and the *planner* which it provides with records of drone dynamics. Therefore we introduce the following.

1187. The monitor has just one, a programmable attribute: a trace of the most recent and all past time-stamped recordings of the dynamics of all drones, that is, an element rodd:RoDD, cf. Item 1185.

type
1187 MRoDD = RoDD
value
1187 attr_MRoDD: CM → MRoDD

The monitor "obtains" current drone dynamics, cudd:CuDD (cf. Item 1184) from the *drones* and offers records of [traces of] drone dynamics,(cf. Item 1185) rodd:RoDD, to the *planner*.

11.2.4.9 Planner Attributes

11.2.4.9.1 Swarms and Businesses:

The *planner* is where all decisions are made with respect to where enterprise drones should be flying; which enterprise drones fly together, which no longer – (with this notion of "flying together" leading us to the concept of *swarms*); which swarms of enterprise drones do which kinds of work – (with this notion of work specialisation leading us to the concept of businesses.)

1188. The is a notion of a *business identifier*, BI.

type
1188 BI

11.2.4.9.2 Planner Directories:

Planners have three directories. These are attributes, BDIR (businesses), SDIR (swarms) and DDIR (drones).

1189. BDIR records which swarms are resources of which businesses;
1190. SDIR records which drones "belong" to which swarms.
1191. DDIR "keeps track" of past and present enterprise drone positions, as per enterprise drone identifier.
1192. We shall refer to this triplet of directories by TDIR

type
1189 BDIR = BI $\overrightarrow{m}$ SI-**set**
1190 SDIR = SI $\overrightarrow{m}$ DI-**set**
1191 DDIR = DI $\overrightarrow{m}$ RoDD
1192 TDIR = BDIR × SDIR × DDIR
value
1189 attr_BDIR: CP → BDIR
1190 attr_SDIR: CP → SDIR
1191 attr_DDIR: CP → DPL

All three directories are *programmable attributes*.

The business swarm concept can be visualized by grouping together drones of the same swarm in the visualization of the aggregate set of enterprise drones. Figure 11.3 on the next page attempts this visualization.

1193. For the planners of all universes of discourse the following must be the case.
    a. The swarm directory must
        i. have entries for exactly the swarms of the business directory,
        ii. define disjoint sets of enterprise drone identifiers, and
        iii. these sets must together cover all enterprise drones.
    b. The drone directory must record the present position, the past positions, a list, dpl:DPL, and, besides satisfying axioms 1186, satisfy some further constraints:
        i. they must list exactly the drone identifiers of the aggregate of enterprise drones, and the sum total of its enterprise drone identifiers must be exactly those of the enterprise drones aggregate of enterprise swarms, and
        ii. the head of a drone's present and past position list must similarly be within reasonable distance of that drone's current position.

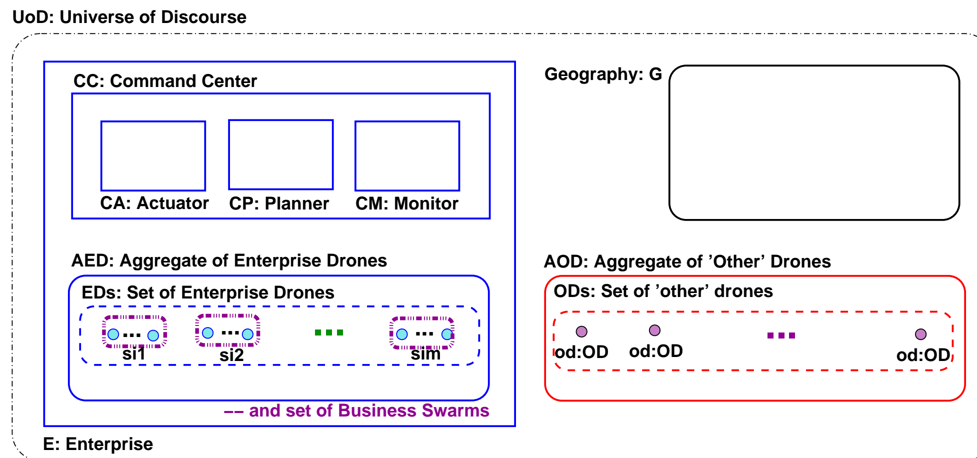


Fig. 11.3 Conceptual Swarms of the Universe of Discourse

axiom
1193 ∀ uod:UpD,e:E,cp:CP,g:G •
1193 e=obs_E(uod)∧cp=obs_CP(obs_CC(e)) ⇒
1193a **let** (bdir,sdir,ddir) = (attr_BDIR,attr_SDIR,attr_DDIR)(cp) **in**
1193(a)i ∪ **rng** bdir = **dom** sdir
1193(a)ii ∧ ∀ si,si′SI•{si,si′}⊆**dom** sdir∧si≠si′ ⇒
1193(a)ii sdir(s) ∩ sdir(s′) = {}
1193(a)iii ∧ ∪ **rng** sdir = xtr_dis(e)
1193(b)i ∧ **dom** ddir = xtr_dis(e)
1193(b)ii ∧ ∀ di:DI•di ∈ **dom** ddir
1193(b)ii **let** (d,dpl) = (attr_DDIR(cp))(di) **in**
1193(b)ii dpl ≠ ⟨⟩
1193(b)ii ⇒ neighbours(f,**hd**(dpl))
1193(b)ii ∧ neighbours(**hd**(dpl),
1193(b)ii attr_EDPOS(xtr_D(obs_Ss(e))(di)))
1193 **end end**

11.2.4.10 Actuator Attributes

The actuator receives, from the planner, flight directives as to which enterprise drones should be redirected. The actuator maintains a record of most recent and all past such flight directives. Finally, the actuator, effects the directives by informing designated enterprise drones as to their next flight plans.

1194. Actuators have one programmable attribute: a flight directive directory. It lists, for each enterprise drone, by identifier, a pair: its current flight plan and a list of past flight plans.

type
1194 FDDIR = EDI $\overrightarrow{m}$ (FP × FP*)
value
1194 attr_FDDIR: CA → FDDIR

11.2.4.11 Geography Attributes

11.2.4.11.1 Constituent Types:

The constituent types of *longitude, latitude* and *altitude* and *positions*, of a *geography*, were introduced in Items 1128.

1195. A further concept of geography is that of *area*.
1196. An area, a:A, is a subset of positions within the geography.

type
1195 A = P-**infset**
axiom
1196 $\forall$ uod:UoD,g:G,a:A• g=obs_G(uod) $\Rightarrow$ a $\subseteq$ attr_EPS(g)

11.2.4.11.2 Attributes

1197. Geographies have, as one of their attributes, a *Euclidian Point Space*, in this case, a *compact*[103] infinite set of three-dimensional positions.

type
1197 EPS = P-**infset**
value
1197 attr_EPS: G $\rightarrow$ EPS

Further geography attributes reflect the “lay of the land and the weather right now !”.

1198. The “lay of the land”, L is a “conglomerate” further undefined geodetics and cadestra[104]
1199. The “weather”W is another “conglomerate” of temperature, humidity, precipitation, air pressure, etc.

type
1198 L
1199 W
value
1198 attr_L: G $\rightarrow$ L
1199 attr_W: G $\rightarrow$ W

11.3 Operations on Universe of Discourse States

Before we analyse & describe perdurants let us take a careful look at the actions that drone and swarm behaviours may take. We refer to this preparatory analysis & description as one of analysing & describing the state operations. From this analysis & description we move on to the analysis & description of behaviours, events and actions. The idea is to be able to prove some relations between the two analyses & descriptions: the state operation and the behaviour analyses & descriptions. We refer to [49, Sects. 2.3 and 2.5].

---

[103] In mathematics, and more specifically in general topology, compactness is a property that generalizes the notion of a subset of Euclidean space being closed (that is, containing all its limit points) and bounded (that is, having all its points lie within some fixed distance of each other). Examples include a closed interval, a rectangle, or a finite set of points.

[104] land surface altitude, streets, buildings (tall or not so tall), power lines, etc.

11.3.1 The Notion of a State

A state is any subset of parts each of which contains one or more dynamic attributes. Following are examples of states of the present case study: a space of interest, an aggregate of 'business' swarms, an aggregate of 'other' swarms, a pair of the aggregates just mentioned, a swarm, or a drone.

11.3.2 Constants

Some quantities of a given universe of discourse are constants. Examples are the unique identifiers of the:

1200. enterprise, e_i;
1201. aggregate of 'other' drones, oi;
1202. geography, g_i;
1203. command center, cc_i;
1204. monitor, cm_i;
1205. planner, cp_i;
1206. actuator, ca_i;
1207. set of 'other' drones, od_{is};
1208. set of enterprise drones, ed_{is};
1209. and the set of all drones, ad_{is}.

value
1200 aed_i:EI = uid_AED(obs_AED(uod))
1201 aod_i:OI = uid_AOD(obs_AOD(uod))
1202 g_i:GI = uid_G(obs_G(uod))
1203 cc_i:CCI = uid_CC(obs_CC(obs_AED(uod)))
1204 cm_i:CMI = uid_CM(obs_CM(obs_CC(obs_AED(uod))))
1205 cp_i:CPI = uid_CP(obs_CP(obs_CC(obs_AED(uod))))
1206 ca_i:CAI = uid_CA(obs_CA(obs_CC(obs_AED(uod))))
1207 od_{is}:ODIs = xtr_dis(obs_AOD(uod))
1208 ed_{is}:EDIs = xtr_dis(obs_AED(uod))
1209 ad_{is}:DI-**set** = $od_{is} \cup ed_{is}$

11.3.3 Operations

An operation is a function from states to states. Following are examples of operations of the present case study: a drone *transfer:* leaving a swarm to join another swarm, a drone *changing course:* an enterprise drone changing course, a swarm *split:* a swarm splitting into two swarms, and swarm *join:* two swarms joining to form one swarm.

11.3.3.1 A Drone Transfer

1210. The *transfer* operator specifies two distinct and unique identifiers, si, si′, of two enterprise swarms, and the unique identifier, di, of an enterprise drone – all of the same universe of discourse. The *transfer* operation further takes a universe of discourse and yields a universe of discourse as follows:
1211. The input argument 'from' and 'to' swarm identifiers are different.
1212. The initial and the final state aggregates of enterprise drones, 'other' drones and geographies are unchanged.

1213. The initial and final state monitors and actuators are unchanged.
1214. The business and the drone directors of the initial and final planner are unchanged.
1215. The 'from' and 'to' input argument swarm identifiers are in the swarm directory and the input argument drone identifiers is in the initial swarm directory entry for the 'from' swarm identifier.
1216. The input argument drone identifier is in final the swarm directory entry for the 'to' swarm identifier.
1217. And the final swarm directory is updated ...

```
value
1210 transfer: DI × SI × SI → UoD ~→ UoD
1210 transfer(di,fsi,tsi)(uod) as uod′
1211    fsi ≠ tsi ∧
1210    let aed = obs_AED(uod), aed′ = obs_AED(uod′), g = obs_G(uod), g′ = obs_G(uod′) in
1210    let cc = obs_CC(aed), cc′ = obs_CC(aed′), aod = obs_AOD(uod), aod′ = obs_AOD(uod′) in
1210    let cm = obs_CM(cc), cm′ = obs_CM(cc′), cp = obs_CP(cc), cp′ = obs_CP(cc′) in
1210    let ca = obs_CA(cc), ca′ = obs_CA(cc′) in
1210    let bdir = attr_BDIR(cc), bdir′ = attr_BDIR(cc′),
1210        sdir = attr_SDIR(cc), sdir′ = attr_SDIR(cc′),
1210        ddir = attr_DDIR(cc), ddir′ = attr_DDIR(cc′) in
1212    post: aed = aed′ ∧ aod = aod′ ∧ g = g′ ∧
1213          cm = cm′ ∧ ca = ca′ ∧
1214          bdir = bdir′ ∧ ddir = ddir′
1215    pre {fsi,tsi} ⊆ dom sdir ∧ di ∈ sdir(fsi)
1216    post di ∉ sdir(fsi′) ∧ di ∈ sdir(tsi′) ∧
1217         sdir′ = sdir † [fsi↦sdir(fsi)∪ di] † [tsi↦sdir(tsi)\di]
1210    end end end end end
```

11.3.3.2 An Enterprise Drone Changing Course

TO BE WRITTEN

11.3.3.3 A Swarm Splitting into Two Swarms

TO BE WRITTEN

11.3.3.4 Two Swarms Joining to form One Swarm

TO BE WRITTEN

11.3.3.5 Etcetera

TO BE WRITTEN

11.4 Perdurants

We observe that the term *train* can have the following "meanings": the *train*, as an *endurant*, parked at the railway station platform, i.e., as a *composite part;* the *train*, as a *perdurant*, as it "speeds" down the railway track, i.e., as a *behaviour;* the *train*, as an *attribute*. This observation motivates that we "magically", as it were, introduce a $\textbf{Behaviour_Signature}_r$ function, cf. [48, Sect. 4]

11.4.1 System Compilation

The $\textbf{Behaviour_Signature}_r$ function "worms" its way, so-to-speak, "down" the "hierarchy" of parts, from the universe of discourse, via its immediate sup-parts, and from these to their sub-parts, and so on, until the $\textbf{Behaviour_Signature}_r$ reaches atomic parts. We shall henceforth do likewise.

11.4.1.1 The Compile Functions

1218. Compilation of a *universe of discourse* results in
   a. the RSL-**Text** of the *core* of the universe of discourse behaviour (which we set to **skip** – allowing us to ignore *core* arguments),
   b. followed by the RSL-**Text** of the parallel composition of the compilation of the enterprise,
   c. followed by the RSL-**Text** of the parallel composition of the compilation of the geography,
   d. followed by the RSL-**Text** of the parallel composition of the compilation of the aggregate of 'other' drones.

1218 $\textbf{Translate}_{UoD}$(uod) ≡
1218a $\mathcal{M}_{\text{uid_UoD(uod)}}$(mereo_UoD(uod),sta(uod))(pro(uod))
1218b ‖ $\textbf{Translate}_{AED}$(obs_AED(uod))
1218c ‖ $\textbf{Translate}_{G}$(obs_G(uod))
1218d ‖ $\textbf{Translate}_{AOD}$(obs_AOD(uod))

1219. Compilation of an *enterprise* results in
   a. the RSL-**Text** of the *core* of the enterprise behaviour (which we set to **skip** – allowing us to ignore *core* arguments),
   b. followed by the RSL-**Text** of the parallel composition of the compilation of the enterprise aggregate of enterprise drones,
   c. followed by the RSL-**Text** of the parallel composition of the compilation of the enterprise command center.

1219 $\textbf{Translate}_{AED}$(e) ≡
1219a $\mathcal{M}_{\text{uid_AED(e)}}$(mereo_E(e),sta(e))(pro(e))
1219b ‖ $\textbf{Translate}_{EDs}$(obs_EDs(e))
1219c ‖ $\textbf{Translate}_{CC}$(obs_CC(e))

1220. Compilation of an *enterprise aggregate of enterprise drones* results in
   a. the RSL-**Text** of the *core* of the aggregate behaviour (which we set to **skip** – allowing us to ignore *core* arguments),
   b. followed by the RSL-**Text** of the parallel composition of the distributed compilation of the enterprise aggregate's set of enterprise drones.

1220 **Translate**$_{EDs}$(es) ≡
1220a $\mathcal{M}_{\text{uid_EDs(es)}}$(mereo_EDS(es),sta(es))(pro(es))
1220b ∥ {**Translate**$_{ED}$(ed)|ed:ED•ed ∈ obs_EDs(s)}

1221. Compilation of an *enterprise drone* results in

a. the RSL-**Text** of the *core* of the enterprise drone behaviour – which is what we really wish to express – and since enterprise drones are here considered atomic, that is where the compilation of enterprise ends.

1221 **Translate**$_{ED}$(ed) ≡
1221a $\mathcal{M}_{\text{uid_ED(ed)}}$(mereo_ED(ed),sta(ed))(pro(ed))

1222. Compilation of an *aggregate of 'other' drones* results in

a. the RSL-**Text** of the *core* of the aggregate 'other' drones behaviour (which we set to **skip** – allowing us to ignore *core* arguments) –
b. followed by the RSL-**Text** of the parallel composition of the distributed compilation of the 'other' drones in the 'other' drones' aggregate set of 'other' drones.

1222 **Translate**$_{AOD}$(aod) ≡
1222a $\mathcal{M}_{\text{uid_OD(od)}}$(mereo_S(ods),sta(ods))(pro(ods))
1222b ∥ {**Translate**$_{OD}$(od)|od:OD•od ∈ obs_ODs(ods)}

1223. Compilation of a(n) *'other' drone* results in

a. the RSL-**Text** of the *core* of the 'other' drone behaviour – which is what we really wish to express – and since 'other' drones are here considered atomic, that is where the compilation of the 'other' drones aggregate

1223a **Translate**_{OD}(ed) ≡
1223a $\mathcal{M}_{\text{uid_OD(od)}}$(mereo_OD(od),sta(od))(pro(od))

1224. Compilation of an atomic *geography* results in

a. the RSL-**Text** of the *core* of the geography behaviour.

1224 **Translate**$_{G}$(g) ≡
1224a $\mathcal{M}_{\text{uid_G(g)}}$(mereo_G(g),sta(g))(pro(g))

1225. Compilation of a composite *command center* results in

a. the RSL-**Text** of the *core* of the command center behaviour (which we set to **skip** – allowing us to ignore *core* arguments)
b. followed by the RSL-**Text** of the parallel composition of the compilation of the command monitor,
c. followed by the RSL-**Text** of the parallel composition of the compilation of the command planner,
d. followed by the RSL-**Text** of the parallel composition of the compilation of the command actuator.

```
1225  TRANSLATE_M(cc) ≡
1225a     M_uid_CC(cc)(mereo_CC(cc),sta(cc))(pro(cc))
1225b   || TRANSLATE_CC(obs_CM(cc))
1225c   || TRANSLATE_CP(obs_CP(cc))
1225d   || TRANSLATE_CA(obs_CA(cc))
```

1226. Compilation of an atomic *command monitor* results in
   a. the RSL-**Text** of the *core* of the monitor behaviour.

```
1226  TRANSLATE_CM(cm) ≡
1226a     M_uid_CM(cm)(mereo_CM(cm),sta(cm))(pro(cm))
```

1227. Compilation of an atomic *command planner* results in
   a. the RSL-**Text** of the *core* of the planner behaviour.

```
1227  TRANSLATE_CP(cp) ≡
1227a     M_uid_CP(cp)(mereo_CP(cp),sta(cp))(pro(cp))
```

1228. Compilation of an atomic *command actuator* results in
   a. the RSL-**Text** of the *core* of the actuator behaviour.

```
1228  TRANSLATE_CA(ca) ≡
1228a     M_uid_CA(ca)(mereo_CA(ca),sta(ca))(pro(ca))
```

11.4.1.2 Some CSP Expression Simplifications

We can justify the following CSP simplifications [98, 102, 141, 144]:

1229. **skip** in parallel with any CSP expression *csp* is *csp*.
1230. The distributed parallel composition of the distributed parallel composition of CSP expressions, csp(i,j), i indexed over I, i.e., i:I, and j:J respectively, is the distributed parallel composition over CSP expressions, csp(i,j), i.e., indexed over (i, j):$I \times J$ – where the index sets iset and jset are assumed.

```
axiom
1230  skip || csp ≡ csp
1230  ||{||{csp(i,j)|i:I•i∈iset}|j:J•j∈jset} ≡ ||{csp(i.j)|i:I,j:J•i∈I-set∧j∈J-set}
```

11.4.1.3 The Simplified Compilation

1231. The simplified compilation results in:

```
1231  TRANSLATE(uod) ≡
1221a       { M_uid_ED(ed)(mereo_ED(ed),sta(ed))(pro(ed))
1221a        | ed:ED • ed ∈ xtr_Ds(obs_AED(uod)) }
1223a     || { M_uid_OD(od)(mereo_OD(od),sta(od))(pro(od))
```

```
1223a     | od:OD • od ∈ xtr_ODs(obs_AOD(uod)) }
1224a   || M_uid_G(g)(mereo_G(g),sta(g))(pro(g))
1224a         where g ≡ obs_G(uod)
1226a   || M_uid_CM(cm)(mereo_CM(cm),sta(cm))(pro(cm))
1226a          where cm ≡ obs_CM(obs_CC(obs_E(uod)))
1227a   || M_uid_CP(cp)(mereo_CP(cp),sta(cp))(pro(cp))
1227a          where cp ≡ obs_CP(obs_CC(obs_E(uod)))
1228a   || M_uid_CA(ca)(mereo_CA(ca),sta(ca))(pro(ca))
1228a          where ca ≡ obs_CA(obs_CC(obs_E(uod)))
```

1232. In Item 1231's Items 1221a, 1223a, 1224a, 1226a, 1227a, and 1228a we replace the "anonymous" behaviour names $\mathcal{M}$ by more meaningful names.

```
1232  TRANSLATE(uod) ≡
1221a     { enterprise_drone_uid_ED(ed)(mereo_ED(ed),sta(ed))(pro(ed))
1221a      | ed:ED • ed ∈ xtr_Ds(obs_AED(uod)) }
1223a   || { other_drone_uid_OD(od)(mereo_OD(od),sta(od))(pro(od))
1223a      | od:OD • od ∈ xtr_ODs(obs_AOD(uod)) }
1224a   || geography_uid_G(g)(mereo_G(g),sta(g))(pro(g))
1224a         where g ≡ obs_G(uod)
1226a   || monitor_uid_CM(cm)(mereo_CM(cm),sta(cm))(pro(cm))
1226a          where cm ≡ obs_CM(obs_CC(obs_E(uod)))
1227a   || planner_uid_CP(cp)(mereo_CP(cp),sta(cp))(pro(cp))
1227a          where cp ≡ obs_CP(obs_CC(obs_E(uod)))
1228a   || actuator_uid_CA(ca)(mereo_CA(ca),sta(ca))(pro(ca))
1228a          where ca ≡ obs_CA(obs_CC(obs_E(uod)))
```

11.4.2 An Early Narrative on Behaviours

11.4.2.1 Either Endurants or Perdurants, Not Both !

First the reader should observe that the manifest parts, in some sense, do no longer "exist"! They have all been replaced by their corresponding behaviours. These behaviours embody all the qualities of their "origin": the unique identifiers, the mereology, and all the attributes – the latter in one form or another: the static attributes as constants (`referred to in the bodies of the behaviour definitions`); the programmable attributes as arguments (`‘‘carried over’’ from one invocation to the next`); and the remaining dynamic attributes as "inputs" (`whose varying values are ‘‘accessed’’ through [dynamic attribute] channels`).

11.4.2.2 Focus on Some Behaviours, Not All !

Secondly we focus, in this case study, only on the behaviour of the *planner*. The other behaviours, the *'other' drones*, *enterprise drones*, *monitor*, *actuator*, and the *geography*, are, in this case study of less interest to us. That is, other case studies could focus on the behaviours of *drones*, or *geographies*, or *monitor*, or *actuator*.

11.4.2.3 The Behaviours – a First Narrative

Drones "continuously" offer their identified dynamics (location, velocity, and possibly more) **to** the *monitor*. *Enterprise drones* "continuously", and in addition, offers to accept flight guidance **from** the *actuator*. The *monitor* "continuously sweeps" the air space and collects the identities of all recognizable drones and their dynamics, and offers this **to** the *planner*. The *planner* does all the interesting work ! It effects the *allocation/reallocation* of drones to/from business swarms; it *calculates enterprise drone flights* and instructs the *actuator* to offer such flight plans to relevant drones; etcetera ! Finally the *actuator*, as instructed by the *planner*, offers flight guidance, as per instructions **from** the *planner*, **to** all or some *enterprise drones*.

11.4.3 Channels

Channels is a concept of CSP [98, 100, 102].

CSP channels are a means for synchronising behaviours and for communicating values between synchronised behaviours, as well as, as a technicality, conveying values of most kinds of dynamic attributes of parts (i.e., endurants) to "their" behavioural counterparts.

There are thus two starting point for the analysis & description of channels: the mereologies and the dynamic attributes of parts. Here we shall single out the following parts and behaviours: the command *monitor, planner* and *actuator*, the *enterprise drones* and the *'other' drones*, and the *geography*. We refer to Fig. 11.4, a slight "refinement" of Fig. 11.1 on page 263.

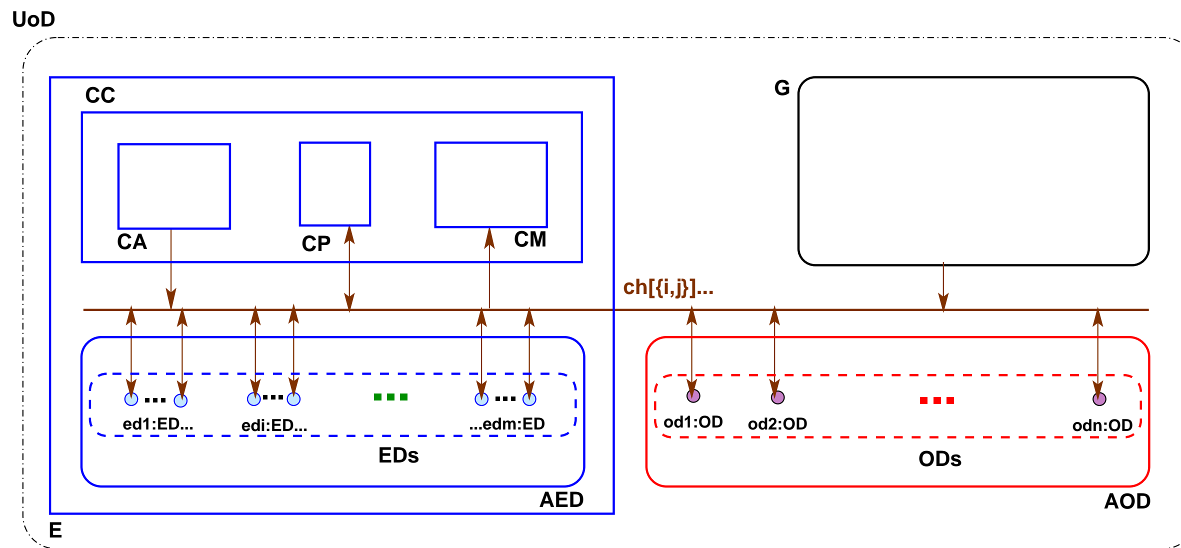


Fig. 11.4 Universe of Discourse with General Channel: **ch[{i,j}] ...**

11.4.3.1 The Part Channels

11.4.3.1.1 General Remarks:

Let there be given a *universe of discourse*. Let us analyse the *unique identifiers* and the *mereologies* of the *planner* cp: (cpi,cpm), *monitor* cm: (cmi,cmm) and *geography* g: (mi,mm), where cpm = (cai,cmi,gi), cmm = ($\{di_1,di_2,\ldots,di_n\}$,cpi) and gm = (cpi,$\{di_1,di_2,\ldots,di_n\}$).

We now interpret these facts. When the *planner mereology* specifies the unique identifiers of the *actuator*, the *monitor*, and the *geography*, then that shall mean there there is a way of communicating messages between the actuator, and the geography, and one side, and the plannner on the other side.

1233. We shall therefore, in a first step of specification development, think of a "grand" array channel over which all communication between behaviours take place. See Fig. 11.4 on the preceding page.
1234. Example indexes into this array channel are shown in the formulas just below.

type
1233 MSG
channel
1233 {ch[fui,tui]|fui,tui:PI • ...}:MSG
value
1234 ch[cpi,cai]!msg *output from* planner *to* actuator.
1234 ch[cpi,cai]? *input from* planner *to* actuator.
1234 etc.

We presently leave the type of messages, MSG, that can be communicated over this "grand" channel further unspecified. We also leave unspecified the pair of distinct unique identifiers that index the channel array. We emphasize that the uniqueness of all part identifiers allow us to use pairs of such as indices. Expression ch[fui,tui]!,sg thus expresses *output from* behaviour indexed by fuit *to* behaviour indexed by tui, whereas expression ch[tui,fui]? thus expresses *input from* behaviour indexed by tui *to* behaviour indexed by fui. Not all combinations of unique identifiers are needed. The channel array is "sparse" ! That property allows us to refine the "grand" channel into the channels illustrated on Fig. 11.5 on the next page. Some channels are array channels: The channels to the drones whether all drones, or just the enterprise drones. Other channels are "single" channels: these are the channels which are anchored in parts with a priori known, i.e., constant unique identifiers.

11.4.3.1.2 Part Channel Specifics

1235. There is an array channel, d_cm_ch[di,cm_i]:D_CM_MSG, from any *drone* ([di]) behaviour to the *monitor* behaviour (whose unique identifier is cm_i). The channel, as an array, forwards the current drone dynamics D_CM_MSG = CuDD.

type
1235 D_CM_MSG = CuDD
channel
1235 {d_cm_ch[di,cm_i]|di:(EDI|ODI)•di ∈ dis}:D_CM_MSG

1236. There is a channel, cm_cp_ch[cm_i,cp_i, from the *monitor* behaviour (cm_i) to the *planner* behaviour (cp_i). It forwards the monitor's records of drone dynamics CM_CP_MSG = MRoDD.

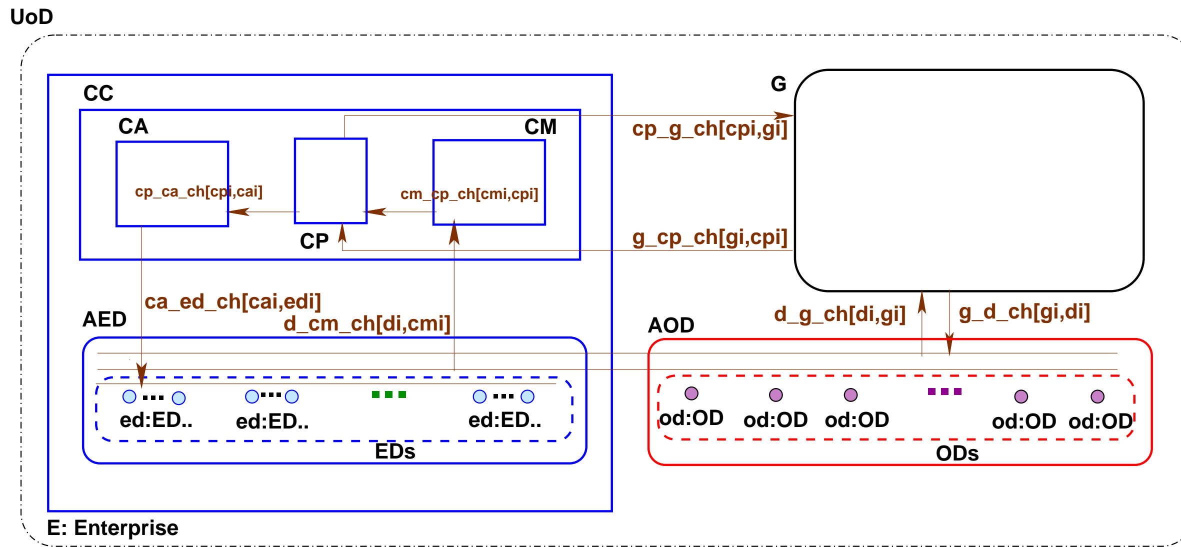


Fig. 11.5 Universe of Discourse with Specific Channels

type
1236 CM_CP_MSG = MRoDD

1236 **channel** m_cp_ch[cm_i,cp_i]:CM_CP_MSG

1237. There is a channel, cp_ca_ch[cp_i,ca_i]:CP_CA_MSG, from the *plannner* behaviour (cp_i) to the *actuator* behaviour (ca_i). It forwards flight plans CP_CA_MSG = FP.

type
1237 CP_CA_MSG = EID $\overrightarrow{m}$ FP
channel
1237 cp_ca_ch[cp_i,ca_i]:CM_CP_MSG

1238. There is an array channel, ca_ed_ch[cai,edi], from the *actuator behaviour* (ca_i) to the *enterprise drone* behaviours (edi for suitable edis). It forwards flight plans, CA_ED_MSG = FP, to enterprise drones in a designated set.

type
1238 CA_ED_MSG = EID × FP
channel
1238 {ca_ed_ch[cai,edi]|edi:EDI•edi ∈ $edis$}:CA_ED_MSG

1239. There is an array channel, g_d_ch[di,g_i]:D_G_MSG, from all the *drone* behaviours (di) to the *geography* behaviour The channels convey, requests for an *immediate geography* for and around a *point*: D_G_MSG = P.

type
1239 D_G_MSG = P
channel
1239 {d_g_ch[di,g_i]|di:(EDI|ODI)•di ∈ dis}:D_H_MSG

1240. There is an array channel, g_d_ch[g_i,di]:G_D_MSG, from the *geography* behaviour to all the *drone* behaviours. The channels convey, for a requested *point*, the immediate geography for that area: G_D_MSG = ImG.

type
1240 G_D_MSG = ImG
channel
1240 {g_d_ch[g_i,di]|di:(EDI|ODI)•di ∈ *dis*}:G_D_MSG

11.4.3.2 Attribute Channels, General Principles

Some of the drone attributes are *reactive*. Being reactive means that their values change surreptitiously. In the physical world of parts that means that these vales must be measured, or somehow ascertained, whenever needed, i.e., "on the fly". Now "our world" is that of a domain description. When dealing with endurants, the value of an attribute, a:A, of part p:P, is expressed as attr_A(p). When dealing with perdurants, that same value is to be expressed as attr_A_ch[uid_P(p)] ?.

1241. This means that we must declare a channel for each part with one or more *dynamic*, however not including *programmable*, attributes A1, A2, ..., An.

channel
1241 attr_A1_ch[p_i]:A1, attr_A2_ch[p_i]:A2, ..., attr_An_ch[p_i]:An

1242. If there are several parts, p1,p2,. . . ,pm:P then an array channel over indices $p1_i$,$p2_i$,. . . ,pm_i is declared for each applicable attribute.

channel
1242 {attr_A1_ch[pj_i]|pj_i:PI•pj_i∈{$p1_i$,$p2_i$,...,pm_i}}:A1,
1242 {attr_A2_ch[pj_i]|pj_i:PI•pj_i∈{$p1_i$,$p2_i$,...,pm_i}}:A2,
1242 ...
1242 {attr_An_ch[pj_i]|pj_i:PI•pj_i∈{$p1_i$,$p2_i$,...,pm_i}}:An

11.4.3.3 The Case Study Attribute Channels

11.4.3.3.1 'Other' Drones:

'Other' drones have the following not biddable or programmable dynamic channels:

1243. dynamics, including velocity, acceleration, orientation and position, {attr_DYN_ch[odi]:DYN|odi:ODI•odi∈*odis*}.

channel
1243 {attr_DYN_ch[odi]:DYN|odi:ODI•odi ∈ *odis*}

11.4.3.3.2 Enterprise Drones:

Enterprise drones have the following not biddable or programmable dynamic channels:

1244. dynamics, including velocity, acceleration, orientation and position, {attr_DYN_ch[edi]:DYN|edi:EDI•edi∈*odis*}.

```
channel
1244   {attr_DYN_ch[ odi ]:DYN|odi:ODI•odi ∈ odis}
```

11.4.3.3.3 Geography:

The geography has the following not biddable or programmable dynamic channels:

1245. land, attr_L_ch[g_i]:L, and
1246. weather, attr_W_ch[g_i]:W.

```
channel
1245   attr_L_ch[ gi ]:L
1246   attr_W_ch[ gi ]:W
```

We do not show any graphics for the attribute channels.

11.4.4 The Atomic Behaviours

TO BE WRITTEN

11.4.4.1 Monitor Behaviour

1247. The signature of the monitor behaviour
   a. lists the monitor's unique identifier, carries the monitor's mereology, has no static arguments (... maybe ...), has the programmable time-stamped recordings, dtp, of all drone positions (present and past) and
   b. further designates the **in**put channel d_cm_ch[*.*] from all drones and the channel **out**put cm_cp_ch[cmi,cpi] to the planner.
1248. The monitor [otherwise] behaves as follows:
   a. All drones provide as input, d_cm_ch[di,cmi]?, their time-stamped positions, rec.
   b. The programmable mrodd attribute is updated, mrodd′, to reflect the latest time stamped dynamics per drone identifier.
   c. The updated attribute is is provided to the planner.
   d. Then the monitor resumes being the monitor, forwarding, as the progammable attribute, the time-stamped drone position recording.

```
value
1247a   monitor: cmi:CMI×cmm:(dis:DI-set×cpi:CPI) → MRoDD →
1247b      in {d_cm_ch[ di,cmi ]|di:DI•di∈dis} out cm_cp_ch  Unit
1248      monitor(mi,(dis,cpi))(mrodd) ≡
1248a      let rec = {[ di ↦ d_cm_ch[ di,cmi ]?|di:DI•di∈dis ]} in
1248b      let mrodd′ = mrodd † [ di↦⟨rec(di)⟩⁀mrodd(di)|di:DI•di∈dis ] in
1248c      cm_cp_ch[ cmi,cpi ] ! mrodd′;
1248d      monitor(cmi,(dis,cpi))(mrodd′)
1248         end end
1248   axiom cmi=cmi∧cpi=cpi
```

We have decided to let the monitor maintain the present and past time-stamped drone positions. It is the monitor which records these positions. Not the planner. But the monitor provides these traces, again-and-again, to the planner.

11.4.4.2 Planner Behaviour

1249. The signature of the planner behaviour
   a. lists the planner's unique identifier, carries the planner's mereology, has, perhaps ..., some static arguments, has the programmable planner directories, and
   b. further designates the single **in**put channel cm_cp_ch and the single **out**put channel cp_ca_ch.
1250. The planner [otherwise] behaves as follows:
   a. the planner [internal] non-deterministically ("coin-flipping") decides whether to transfer a drone between business swarms, or to calculate flight plans, or . . . other.
   b. Depending on the [outcome of the "coin-flipping"] the planner
   c. either effects a transfer,
      i. by delegating to an auxiliary function, transfer, the necessary modifications of the swarm directory –
      ii. whereupon the planner behaviour resumes;
   d. or effects a [re-]calculation on drone flights,
      i. by, again, delegating to an auxiliary function, flight_planning, the necessary calculations –
      ii. which are communicated to the *actuator*,
      iii. whereupon the planner behaviour resumes;
   e. or . . . other !

```
value
1250        planner: cpi:CPI × (cai>CAI×cmi:CMI×gi:GI) × TDIR →
1250            in cm_cp_ch[cmi,cpi], g_cp_ch[gi,cpi] out cp_ca_ch[cpi,cai] Unit
1249        planner(cpi,(cai,cmi,gi),...)(bdir,sdir,ddir) ≡
1250a           let cmd = "transfer" ⊓ "flight_plan" ⊓ ... in
1250b           cases cmd of
1250c               "transfer" →
1250(c)i                let sdir′ = transfer(tdir) in
1250(c)ii               planner(cpi,(cai,cmi,gi),...)(bdir,sdir′,ddir) end
1250d               "flight_plan" →
1250(d)i                let ddir′ = flight_planning(tdir) in
1250(d)ii               planner(cpi,(cai,cmi,gi),...)(bdir,sdir,ddir′) end
1250e               ...
1249            end
1249        axiom cpi=cp_i∧cai=ca_i∧cmi=cm_i∧gi=g_i
```

11.4.4.2.1 The Auxiliary transfer Function

1251. The *transfer* function has a simpler signature than the planner behaviour in that it need not communicate with other behaviours.
   a. The transfer function *internal non-deterministically chooses* a business designator, bi;
   b. from among that business' swarm designators it *internal non-deterministically chooses* two distinct swarm designators, fsi,tsi;

c. and from the fsi entry in sdir (which is set of enterprise drone identifiers), it *internal non-deterministically chooses* an enterprise drone identifier, di.
d. Given the swarm and drone identifiers *the resulting swarm directory* can now be made to reflect the transfer: reference to di is *removed* from the fsi entry in sdir and that reference instead *inserted* into the tsi entry.

```
value
1251   transfer: TDIR → SDIR
1251   transfer(bdir,sdir,ddir) ≡
1251a     let bi:BI•bi ∈ dom bdir in
1251b     let fsi,tsi:SI•{fsi,tsi}⊆bdir(bi)∧fsi≠tsi in
1251c     let di:DI•di ∈ sdir(fsi) in
1251d     sdir † [ fsi↦sdir(fsi)\{di} ] † [ tsi↦sdir(tsi)∪{di} ]
1251      end end end
```

11.4.4.2.2 The Auxiliary flight_planning Function

1252. The signature of the flight_planning behaviour needs two elements: the triplet of business, swarm and drone directories, and the planner-to-actuator channel.
    a. The flight_planning behaviour offers to accept the time-stamped recordings of the most recent drone positions and dynamics as well as all the past such recordings.
    b. The flight_planning behaviour selects, *internal, non-deterministically* a business, designated by bi,
    c. one of whose swarms, designated by si, it has thus decided to perform a flight [re-]calculation for.
    d. An objective for the new flight plan is chosen.
    e. The flight_plan is calculated.
    f. That flight plan is communicated to the *actuator*.
    g. And the flight plan, appended to the drone directory's (past) flight plans.

```
value
1252   flight_planning: TDIR → in cm_cp_ch[cm_i,cp_i], out cp_ca_ch[cp_i,ca_i]  DTP
1252   flight_planning(bdir,sdir,ddir) ≡
1252a     let dtp = cm_cp_ch[cp_i,ca_i] ? ,
1252b         bi:BI • bi ∈ dom bdir
1252c     let si:SI • si ∈ bdir(bi) in
1252d     let fp_obj:fp_objective(bi,si) in
1252e     let flight_plan = calculate_flight_plan(dtp,sdir(si),fp_obj,tdir) in
1252f     cp_ca_ch[cp_i,ca_i] ! flight_plan ;
1252g     ⟨flight_pla⟩⁀ddir
1252      end end end end
type
1252d   FP_OBJ
value
1252d   fp_objective: BI × SI → FP_OBJ
1252d   fp_objective(bi,si) ≡ ...
```

1253. The calculate_flight_plan function is the absolute focal point of the *planner*.

```
1253   calculate_flight_plan: DTP × DI-set × FP−OBJ × TDIR → FP
1253   calculate_flight_plan(dtp,sdir(si),fp_obj,tdir) ≡ ...
```

There are many ways of calculating flight plans.

[124, Mehmood et al., Stony Brook, 2018: *Declarative vs Rule-based Control for Flocking Dynamics*] is one such:

TO BE WRITTEN

In [138, 139, 140, Craig Reynolds: OpenSteer, *Steering Behaviours for Autonomous Characters*]

TO BE WRITTEN

In [126, Reza Olfati-Saber: Flocking for Multi-agent Dynamic Systems: Algorithms and Theory, 2006]

TO BE WRITTEN

The calculate_flight_plan function, Item 1253 on the preceding page, is deliberately provided with all such information that can be gathered and hence can be the only 'external'[105] data that can be provided to such calculation functions,[106] and is therefore left further unspecified; future work[107] will show whether this assumption holds. If it does, then, OK, and we can proceed. If it does not, we shall revise the present model.

11.4.4.3 Actuator Behaviour

1254. The actuator accepts a current flight plan, cfp:CFP, i.e., a number of enterprise drone identifier-indexed flight plans, from the planner.
1255. The signature of the actuator behaviour lists the actuator's unique identifier, carries the actuator's mereology, has, perhaps ..., some static arguments, has the programmable flight directory, and further designates the **in**put channel cp_ca_ch[cpi,cai] and the **out**put channel ca_ed_ch[cai,*].
1256. The actuator further behaves as follows:
    a. It offers to accept a current flight plan from the planner.
    b. It then proceeds to offer those enterprise drones which are designated in the flight plan their flight plan.
    c. Whereupon the actuator resumes being the actuator, now with its programmable flight plan directory updated with the latest such !

```
type
1254  CFP = EDI →m FP
value
1255  actuator: cai:CAI × (cpi:CPI×edis:EDI-set) → FDDIR →
1255      in cp_ca_ch[ cpi,cai ] out {ca_ed_ch[ cai,edi ]|edi:EDI•edi ∈ edis}  Unit
1256  actuator(cai,(cpi,edis),...)(pfp,pfpl) ≡
1256a    let cfp = ca_cp_ch[ cai,cpi ] ? in comment: fp:EDI →m FP
1256b    || {ca_ed_ch[ cai,edi ]!cfp(edi)|edi:EDI•edi ∈ dom cfp} ;
1256c    actuator(cai,(cpi,edis),...)(cfp,⟨pfp⟩⌢pfpl)
1254      end

1255  axiom cai=ca_i∧cpi=cp_i
```

[105] Flight plan *objectives* are here referred to as 'internal'.

[106] Well – better check this!

[107] – for you ShaoFa !

11.4.4.4 'Other' Drone Behaviour

1257. The signature of the *'other' drone* behaviour
   a. lists the 'other' drone's unique identifier, the 'other' drone's mereology, has, perhaps ..., some static arguments; then the programmable attribute of the geography (i.e., the area, the land and the weather) it is moving over and in;
   b. then, as **in**put channels, the *inert, active, autonomous* and *biddable* attributes: velocity, acceleration, orientation and position, and, finally
   c. further designates the array **in**put channel g_d_ch[*] from the *geography* and the array **out**put channel d_cm_ch[*] to the *monitor*.
1258. The 'other' drone otherwise behaves as follows:
1259. internal, non-deterministically the 'other' drone chooses to either ..., or "pro"viding to the monitors request for drone "dyn"amics, or ... .
1260. If the choice is ... ,
1261. If the choice is "provide dynamics" the behaviour drone_monitor is invoked, with arguments similar to that of other_drone, but "marked" with an additional, "frontal" argument: "other", and with "tail", programmable arguments (⟨⟩,⟨⟩).
1262. If the choice is ... .

```
value
1257   other_drone: odi:ODI × (cmi:CMI×gi:GI) × ... → (DYN×ImG) →
1257b      in attr_DYN_ch[odi],g_d_ch[gi,odi] out d_cm_ch[odi,cmi] Unit
1258   other_drone(odi,(cmi,gi),...)(dyn:(v,a,o,p),img) ≡
1259      let mode = "..." ⊓ "pro_dyn" ⊓ "..." in
1259      case mode of
1260         "..." → ... ,
1261         "pro_dyn" → drone_moni(odi,(cmi,gi),...)(dyn:(v,a,o,p),img)
1262         "..." → ...
1259      end
1257      end
```

1263. If the choice is "provide dynamics"
   a. then the drone-monitor behaviour ascertains its dynamics (velocity, acceleration, orientation and position),
   b. informs the monitor 'thereof', and
   c. resumes being the 'other' drone with that updated, programmable dynamics.

```
value
1263    drone_moni: odi:ODI × (cmi:CMI×gi:GI) × ... → (DYN×ImG) →
1263       in attr_DYN_ch[odi],g_d_ch[gi,odi] out d_cm_ch[odi,cmi] Unit
1262    drone_moni(odi,(cmi,gi),...)(dyn:(v,a,o,p),img) ≡
1263a      let (ti,dyn′,img′) =
1263a            (time(),
1263a               (let (v′,a′,o′,p′) = attr_DYN[odi]? in
1263a               (v′,a′,o′,p′),
1263a                  d_g_ch[odi,gi]!p′ ; g_d_ch[gi,odi]? end)) in
1263b      d_cm_ch[odi,cmi] ! (ti,dyn′) ;
1263c      other_drone(cai,(cpi,edis),...)(dyn′,img′)
1263a      end
```

11.4.4.5 Enterprise Drone Behaviour

1264. The enterprise donor lists its enterprise drone's unique identifier, carries it's mereology, has, perhaps ..., some static arguments, the programmable enterprise drone attributes: a pair of the present flight plan, and the past flight plans, and a pair of the most recently observed dynamics and immediate geography, and further designates the single **in**put channel and the **out**put channel array .

Enterprise drones otherwise behave as follows:

1265. internal, non-deterministically an enterprise drone chooses to either "rec"ording the "geo"graphy, i.e., the area, land and weather it is situated in, or "pro"viding to the monitors request for drone "dyn"amics, or "acc"epting the actuators offer of a new "f"light "p"lan, or "move" "on" (i.e., continue to fly), either "follow"ing the "flight plan" most recently received from the actuator, or, "ignor"ing this directive, "just plondering on"!
1266. If the choice is "rec_geo" then the enterprise_geo behaviour is invoked,
1267. If the choice is "pro_dyn" (provide dynamics to the *monitor*) then the enterprise_moni behaviour is invoked,
1268. If the choice is "acc_fp" then the enterprise_accept_flight_plan behaviour is invoked,
1269. If the choice is "move_on" then the enterprise drone decides either to "ignore" the flight plan, or to "follow" it.
    a. If it "ignore"s the flight plan then the enterprise_ignore behaviour is invoked,
    b. If the choice is "follow" then the enterprise_follow behaviour is invoked.

```
1264     enterprise_drone: edi:EDI×(cmi:CMI×cai:CAI×gi:GI) →
1264        ((FPL×PFPL)×(DDYN×ImG)) →
1264        in attr_DYN_ch[ edi ],g_d_ch[ gi,edi ],ca_ed_ch[ cai,edi ]
1264        out d_cm_ch[ edi,cmi ],d_g_ch[ edi,gi ]  Unit
1264     enterprise_drone(edi,(cmi,cai,gi),...)(fpl,pfpl,(ddyn,img)) ≡
1265        let mode = ″rec_geo″ ⊓ ″pro_dyn″ ⊓ ″acc_fp″ ⊓ ″move_on″ in
1265        case mode of
1266           ″rec_geo″ → enterprise_geo(edi,(cmi,cai,gi),...)(fpl,pfpl,(ddyn,img))
1267           ″pro_dyn″ → enterprise_moni(edi,(cmi,cai,gi),...)(fpl,pfpl,(ddyn,img))
1268           ″acc_fp″ → enterprise_acc_fl_pl(edi,(cmi,cai,gi),...)(fpl,pfpl,(ddyn,img))
1269           ″move_on″ →
1269              let m_o_mode = ″ignore″ ⊓ ″follow″ in
1269              case m_o_mode of
1269a                ″ignore″ → enterprise_ignore(edi,(cmi,cai,gi),...)(fpl,pfpl,(ddyn,img))
1269b                ″follow″ → enterprise_follow(edi,(cmi,cai,gi),...)(fpl,pfpl,(ddyn,img))
1275              end
1275              end
1265        end
1265        end
1264     axiom  cmi=cm_i∧cai=ca_i∧gi=g_i
```

1270. If the choice is "rec_geo"
    a. then dynamics is ascertained so as to obtain a positions;
    b. that position is used in order to obtain a "fresh" immediate geography;
    c. with which to resume the enterprise drone behaviour.

```
1264     enterprise_geography: edi:EDI×(cmi:CMI×cai:CAI×gi:GI) →
1264        ((FPL×PFPL)×(DDYN×ImG)) →
```

```
1264      in attr_DYN_ch[ edi ],g_d_ch[ gi,edi ],ca_ed_ch[ cai,edi ]
1264      out d_cm_ch[ edi,cmi ],d_g_ch[ edi,gi ]  Unit
1264    enterprise_geography(edi,(cmi,cai,gi),...)((fpl,pfpl),(ddyn,img)) ≡
1270a     let (v,a,o,p) = attr_DYN_ch[ edi ]? in
1270b     let img′ = d_g_ch[ edi,gi ]!p;g_d_ch[ gi,edi ]? in
1270c     enterprise_drone(edi,(cmi,cai,gi),...)((fpl,pfpl),((v,a,o,p),img′))
1270a     end end
```

1271. If the choice is "pro_dyn" (provide dynamics to the *monitor*)
   a. then a triplet is obtained as follows:
   b. the current time,
   c. the dynamics (v,a,o,p), and
   d. the immediate geography of position p,
   e. such that the *monitor* can be given the current dynamics,
   f. and the enterprise drone behaviour is resumed with updated dynamics and immediate geography.

```
1264    enterprise_monitor: edi:EDI×(cmi:CMI×cai:CAI×gi:GI) →
1264      ((FPL×PFPL)×(DDYN×ImG)) →
1264      in attr_DYN_ch[ edi ],g_d_ch[ gi,edi ],
1264      out d_cm_ch[ edi,cmi ],d_g_ch[ edi,gi ]  Unit
1264    enterprise_monitor(edi,(cmi,cai,gi),...)((fpl,pfpl),(ddyn,img)) ≡
1271a       let (ti,ddyn′,img′) =
1271b             (time(),
1271c                 (let (v,a,o,p) = attr_DYN[ edi ]? in
1271c                 (v,a,o,p),
1271d                     d_g_ch[ edi,gi ]!p;g_d_ch[ gi,edi ]? end)) in
1271e        d_cm_ch[ edi,cmi ] ! (ti,ddyn′) ;
1271f       enterprise_drone(edi,(cmi,cai,gi),...)((fpl,pfpl),(ddyn′,img′))
1271a        end
```

1272. If the choice is "acc_fp"
   a. the enterprise drone offers to accept a new flight plan from the *actuator*
   b. and the enterprise drone behaviour is resumed with that flight plan now becoming the next current flight plan and whatever is left of the hitherto current flight plan appended to the past flight plan list.

```
1264    enterprise_acc_fl_pl: edi:EDI×(cmi:CMI×cai:CAI×gi:GI) →
1264      ((FPL×PFPL)×(DDYN×ImG)) → in ca_ed_ch[ cai,edi ]  Unit
1264    enterprise_axx_fl_pl(edi,(cmi,cai,gi),...)((fpl,pfpl),(ddyn,img)) ≡
1272a       let fpl′ = ca_ed_ch[ cmi,edi ] ? in
1272b       enterprise_drone(edi,(cmi,cai,gi),...)(fp′,⟨fpl⟩⁀pfpl,(ddyn,img))
1272a       end
```

1273. If the choice is "move_on" and the enterprise drone decides to "ignore" the flight plan,
   a. then it ascertains where it might be moving with the current dynamics
   b. and then it just keeps moving on till it reaches that dynamics
   c. from about where it resumes the enterprise drone behaviour.

```
1264    enterprise_ignore: edi:EDI×(cmi:CMI×cai:CAI×gi:GI) →
1264      ((FPL×PFPL)×(DDYN×ImG)) →
1264      in attr_DYN_ch[ edi ] out d_cm_ch[ edi,cmi ],d_g_ch[ edi,gi ]  Unit
1264    enterprise_ignore(edi,(cmi,cai,gi),...)((fpl,pfpl),(ddyn,img)) ≡
1273a     let (v′,a′,o′,p′) = increment(dyn,img) in
1273b     while let (v′′,a′′,o′′,p′′) = attr_DYN_ch[ odi ]? in
1273b             ∼close(p′,p′′) end do manoeuvre(dyn,img) ; wait δ t end ;
1273c     enterprise_drone(cai,(cpi,edis),...)(fpl,pfpl,(attr_DYN_ch[ odi ]?,img))
1273a     end
```

1274. The manoeuvre behaviour is further unspecified. For a fixed wing aircraft it controls the *yaw*, the *roll* and the *pitch* of the aircraft, hence its flight path, by operating the *elevator, aileron, ruddr* and the *thrust* of the aircraft based on its current dynamics, weight (including aircraft fuel), meteorological conditions (winds etc.).

```
value
1274    manoeuvre: DYN × ImG → Unit
1274    manoeuvre(dyn,img) ≡ ...
```

The **wait** δ t is some drone constant.

1275. If the choice is "`move_on`" and the enterprise drone decides to "`follow`" the flight plan,
   a. then, if the current flight plan has been exhausted, i.e., "used-up" it aborts (**chaos**[108])
   b. otherwise it ascertains where it might be moving, i.e., a next dynamics from with the current dynamics.
   c. So it then "moves along" until it has reached that dynamics –
   d. from about where it resumes the enterprise drone behaviour.

```
value
1264    enterprise_follow: edi:EDI×(cmi:CMI×cai:CAI×gi:GI) →
1264      ((FPL×PFPL)×(DDYN×ImG)) →
1264      in attr_DYN_ch[ edi ] out d_cm_ch[ edi,cmi ],d_g_ch[ edi,gi ]  Unit
1264    enterprise_follow(edi,(cmi,cai,gi),...)((fpl,pfpl),(ddyn,img)) ≡
1275a   if fpl = ⟨⟩ then chaos else
1275b   let (v′,a′,o′,p′) = increment(dyn,img,hd fpl) in
1275c   while let (v′′,a′′,o′′,p′′) = attr_DYN_ch[ odi ]? in
1275c           ∼close(p′,p′′) end do manoeuvre(hd fpl,dyn,img) ; wait δ t end ;
1275d   enterprise_drone(edi,(cmi,cai,gi),...)((tlfpl,pfpl),(attr_DYN_ch[ odi ]?,img))
1275a   end end
```

1276. The (overloaded) manoeuvre behaviour is further unspecified. For a fixed wing aircraft it controls the *yaw*, the *roll* and the *pitch* of the aircraft, hence its flight path, by operating the *elevator, aileron, ruddr* and the *thrust* of the aircraft based on its current dynamics, weight (including aircraft fuel), meteorological conditions (winds etc.).

```
value
1276    manoeuvre: FPE × DYN × ImG → Unit
1276    manoeuvre(fpe,dyn,img) ≡ ...
```

The **wait** δ t is some drone constant.

[108] **chaos** means that we simply decide not to describe what then happens !

11.4.4.6 Geography Behaviour

1277. The *geography* behaviour definition
   a. lists the geography behaviour's unique identifier, carries the its mereology, has the static argument of its Euclidean point space, and
   b. further designates the single **in**put channels cp_g_ch[cpi,gi] from the *planner* and d_g_ch[*,gi] from the drones and the **out**put channels g_cp_ch[gi,cpi] to the *planner* and g_d_ch[gi,*] to the *drones*.
1278. The *geography* otherwise behaves as follows:
   a. Internal, non-deterministically the geography chooses to either "resp"ond to a request from the "plan"ner.
   b. If the choice is
   c. "resp_plan"
      i. then the *geography* offers to accept a request from the *planner* for the *immediate geography* of an *area* "around" a *point* and
      ii. then the *geography* offers that information to the *planner*,
      iii. whereupon the geography resumes being that;

      else if the choice is
   d. "resp_dron"
      i. then then the *geography* offers to accept a request from the *planner* for the *immediate geography* of an *area* "around" a *point* and
      ii. then the *geography* offers that information to the *planner*,
      iii. whereupon the geography resumes being that.
1279. The area function takes a pair of a point and a pair of *land* and *weather* and yields an *immediate geography*.

```
value
1277      geography: gi:GI × gm:(cpi:CPI×cmi:CMI×dis:DI-set) × EPS →
1277a         in cp_g_ch[cpi,gi], d_g_ch[*,gi]
1277b         out g_cp_ch[gi,cpi], g_d_ch[gi,*] Unit
1277      geography(gi,(cpi,cmi,dis),eps) ≡
1278a         let mode = "resp_plan" ⊓ "resp_dron" ⊓ ... in
1278b         case mode of
1278c            "resp_plan" →
1278(c)i              let p = cp_g_ch[cpi,gi] ? in
1278(c)ii             g_cp_ch[gi,cpi] ! area(p,(attr_L_ch[gi]?,attr_W_ch[gi]?)) end
1278(c)iii            geography(gi,(cpi,cmi,dis),eps)
1278d            "resp_dron" →
1278(d)i              let (p,di) = ⌈⌉⌊{(d_g_ch[di,gi]?,di)|di:DI•di ∈ dis} in
1278(d)ii             g_cp_ch[di,cpi] ! area(p,(attr_L_ch[gi]?,attr_W_ch[gi]?)) end
1278(d)iii            geography(gi,(cpi,cmi,dis),eps)
1277          end end
axiom
1277      gi=g_i∧cpi=cp_i∧smi=cm_idis=dis
value
1279      area: P × (L × W) → ImG
1279      area(p,(l,w)) ≡ ...
```

11.5 Conclusion

TO BE WRITTEN

Chapter 12

Container Terminals [November 2017]

Contents

12.1 Introduction 304
12.1.1 Reference Literature on Container-related Matters 304
12.2 Some Pictures 304
12.2.1 Terminal Port Container Stowage Area 304
12.2.2 Container Stowage Area and Quay Cranes 305
12.2.3 Container Vessel Routes 305
12.2.4 Containers 306
12.2.4.1 40 and 20 Feet Containers 306
12.2.4.2 Container Markings 306
12.2.5 Container Vessels 306
12.2.6 Container Stowage Area: Bays Rows, Stacks and Tier 307
12.2.7 Stowage Software 308
12.2.8 Quay Cranes 308
12.2.9 Container Stowage Area and Stack Cranes 308
12.2.10 Container Stowage Area 309
12.2.11 Quay Trucks 309
12.2.12 Map of Shanghai and YangShan 309
12.3 SECT 310
12.4 Main Behaviours 311
12.4.1 A Diagram 312
12.4.2 Terminology - a Caveat 312
12.4.3 Assumptions 313
12.5 Endurants 313
12.5.1 Parts 314
12.5.1.1 Terminal Ports 314
12.5.1.2 Quays 315
12.5.1.3 Container Stowage Areas: Bays, Rows and Stacks 315
12.5.1.4 Vessels 316
12.5.1.5 Functions Concerning Container Stowage Areas 316
12.5.1.6 Axioms Concerning Container Stowage Areas 316
12.5.1.7 Stacks 317
12.5.2 Terminal Port Command Centers 317
12.5.2.1 Discussion 317
12.5.2.2 Justification 318
12.5.3 Unique Identifications 318
12.5.3.1 Unique Identifiers: Distinctness of Parts 319
12.5.3.2 Unique Identifiers: Two Useful Abbreviations 319
12.5.3.3 Unique Identifiers: Some Useful Index Set Selection Functions 319
12.5.3.4 Unique Identifiers: Ordering of Bays, Rows and Stacks 319
12.5.4 States, Global Values and Constraints 320
12.5.4.1 States 320
12.5.4.2 Unique Identifiers 320
12.5.4.3 Some Axioms on Uniqueness 321
12.5.5 Mereology 321

12.5.5.1 Physical versus Conceptual Mereology . . . 321
12.5.5.2 Vessels . . . 322
12.5.5.2.1 Physical Mereology: . . . 322
12.5.5.2.2 Conceptual Mereology: . . . 322
12.5.5.3 Quay Cranes . . . 322
12.5.5.3.1 Physical Mereology: . . . 322
12.5.5.3.2 Conceptual Mereology: . . . 323
12.5.5.4 Quay Trucks . . . 323
12.5.5.4.1 Physical Mereology: . . . 323
12.5.5.4.2 Conceptual Mereology: . . . 323
12.5.5.5 Stack Cranes . . . 324
12.5.5.5.1 Physical Mereology: . . . 324
12.5.5.5.2 Conceptual Mereology: . . . 324
12.5.5.6 Container Stowage Areas . . . 324
12.5.5.6.1 Bays, Rows and Stacks: . . . 324
12.5.5.7 Bay Mereology . . . 325
12.5.5.7.1 Physical Vessel Bay Mereology: . . . 325
12.5.5.7.2 Conceptual Vessel Bay Mereology: . . . 325
12.5.5.7.3 Physical Terminal Port Bay (cum Stack) Mereology: 325
12.5.5.7.4 Conceptual Terminal Port Bay (cum Stack) Mereology: . . . 325
12.5.5.8 Land Trucks . . . 326
12.5.5.8.1 Physical Mereology: . . . 326
12.5.5.8.2 Conceptual Mereology: . . . 326
12.5.5.9 Command Center . . . 326
12.5.5.10 Conceptual Mereology of Containers . . . 327
12.5.6 Attributes . . . 327
12.5.6.1 States . . . 327
12.5.6.2 Actions . . . 327
12.5.6.3 Attributes: Quays . . . 327
12.5.6.4 Attributes: Vessels . . . 328
12.5.6.5 Attributes: Quay Cranes . . . 328
12.5.6.6 Attributes: Quay Trucks . . . 329
12.5.6.7 Attributes: Terminal Stack Cranes . . . 329
12.5.6.8 Attributes: Container Stowage Areas . . . 329
12.5.6.9 Attributes: Land Trucks . . . 330
12.5.6.10 Attributes: Command Center . . . 330
12.5.6.11 Attributes: Containers . . . 331
12.6 Perdurants . . . 332
12.6.1 A Modelling Decision . . . 332
12.6.2 Virtual Container Storage Areas . . . 332
12.6.3 Changes to The Parts Model . . . 333
12.6.4 Basic Model Parts . . . 334
12.6.5 Actions, Events, Channels and Behaviours . . . 334
12.6.6 Actions . . . 334
12.6.6.1 Command Center Actions . . . 335
12.6.6.1.1 Motivating the Command Center Concept: . . . 335
12.6.6.1.2 Calculate Next Transaction: . . . 335
12.6.6.1.3 Command Center Action [A]: update_mcc_from_vessel: . . . 336
12.6.6.1.4 Command Center Action [B]: calc_ves_pos: . . . 337
12.6.6.1.5 Command Center Action [C-D-E]: calc_ves_qc . . . 337
12.6.6.1.6 Command Center Action [F-G-H]: calc_qc_qt . . . 337
12.6.6.1.7 Command Center Action [I-J-K]: calc_qt_sc . . . 337
12.6.6.1.8 Command Center Action [L-M-N]: calc_sc_stack . . . 338
12.6.6.1.9 Command Center Action [N-M-L]: calc_stack_sc . . . 338
12.6.6.1.10 Command Center Action [O-P-Q]: calc_sc_lt . . . 338
12.6.6.1.11 Command Center Action [Q-P-O]: calc_lt_sc . . . 339
12.6.6.1.12 Command Center: Further Observations . . . 339
12.6.6.2 Container Storage Area Actions . . . 339
12.6.6.2.1 The Load Pre-/Post-Condtions . . . 339
12.6.6.2.2 The Unload Pre-/Post-Conditions . . . 340
12.6.6.3 Vessel Actions . . . 341

12.6.6.3.1 Action [A]: calc_next_port: 341
12.6.6.3.2 Vessel Action [B]: calc_ves_msg: 341
12.6.6.4 Land Truck Actions 342
12.6.6.4.1 Land Truck Action [R]: calc_truck_delivery: 342
12.6.6.4.2 Land Truck Action [S]: calc_truck_avail: 342
12.6.7 Events 342
12.6.7.1 Active Part Initiation Events 342
12.6.7.2 Active Part Completion Events: 344
12.6.8 Channels 344
12.6.8.1 Channel Declarations 344
12.6.8.2 Channel Messages 344
12.6.8.2.1 A,B,X,Y,C′: Vessel Messages 345
12.6.8.2.2 C,D,E,E′: Vessel/Container/Quay Crane Messages 345
12.6.8.2.3 F,G,H,H′: Quay Crane/Container/Quay Truck Messages 346
12.6.8.2.4 I,J,K,K′: Quay Truck/Container/Stack Crane Messages 346
12.6.8.2.5 L,M,N,N′: Stack Crane/Container/Stack Messages 346
12.6.8.2.6 O,P,Q,Q′: Land Truck/Container/Stack Crane Messages 347
12.6.8.2.7 R,S,T,U,Q,V: Land Truck Messages 347
12.6.9 Behaviours 348
12.6.9.1 Terminal Command Center 348
12.6.9.1.1 The Command Center Behaviour: 348
12.6.9.1.2 The Command Center Monitor Behaviours: 349
12.6.9.1.3 The Command Center Control Behaviours: 349
12.6.9.2 Vessels 351
12.6.9.2.1 Port Approach 351
12.6.9.2.2 Port Arrival 351
12.6.9.2.3 Unloading of Containers 352
12.6.9.2.4 Loading of Containers 352
12.6.9.2.5 Port Departure 353
12.6.9.3 Quay Cranes 353
12.6.9.4 Quay Trucks 353
12.6.9.5 Stack Crane 354
12.6.9.6 Stacks 354
12.6.9.7 Land Trucks 355
12.6.9.8 Containers 356
12.6.10 Initial System 356
12.6.10.1 The Distributed System 356
12.6.10.2 Initial Vessels 356
12.6.10.3 Initial Land Trucks 357
12.6.10.4 Initial Containers 357
12.6.10.5 Initial Terminal Ports 357
12.6.10.6 Initial Quay Cranes 357
12.6.10.7 Initial Quay Trucks 358
12.6.10.8 Initial Stack Cranes 358
12.6.10.9 Initial Stacks 358
12.7 Conclusion 358
12.7.1 An Interpreation of the Behavioural Description 358
12.7.2 What Has Been Done 358
12.7.3 What To Do Next 359
12.7.4 Acknowledgements 359

We present a recording of stages and steps of a development of a domain analysis & description of an answer to he question: *what, mathematically, is a container terminal ?*
This is a report on an experiment. At any stage of development, and the present draft stage is judged 2/3 “completed” it reflects how I view an answer to the question *what is a container terminal port ?* mathematically speaking.

12.1 Introduction

TO BE WRITTEN

12.1.1 Reference Literature on Container-related Matters

We refer to: [28, A Container Line Industry Domain, 2007], [82, A-Z Dictionary of Export, Trade and Shipping Terms], [128, Portworker Development Programme: PDP Units], [132, An interactive simulation model for the logistics planning of container operations in seaports,1996], [5, Stowage planning for container ships to reduce the number of shifts, 1998], [153, Container stowage planning: a methodology for generating computerised solutions, 2000], [4, Container ship stowage problem: complexity and connection to the coloring of circle graphs, 2000], [154, Container stowage pre-planning: using search to generate solutions, a case study, 2001], [79, A genetic algorithm with a compact solution encoding for the container ship stowage problem, 2002], [104, Multi-objective ... stowage and load planning for a container ship with container rehandle ..., 2004], [150, Container terminal operation and operations research - a classification and literature review, 2004], [74, Online rules for container stacking, 2010],

12.2 Some Pictures

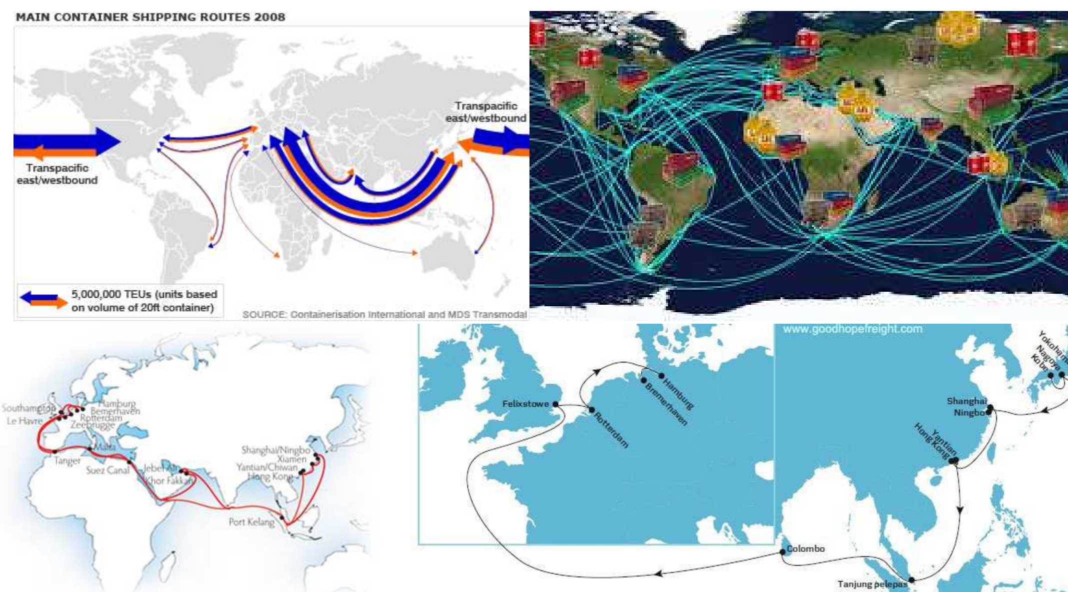
12.2.1 Terminal Port Container Stowage Area

Analysis of the above picture:

- The picture shows a *terminal*.
- At bottom we are hinted (through shadows) at *quay cranes* serving (unshown) *vessels*.
- Most of the picture shows a *container stowage area*, here organized as a series of columns, from one side of the picture to the other side, e.g., left-to-right, sequences (top-to-bottom) of [blue] *bays* with *rows* of *stacks* of *containers*.
- Almost all columns show just one *bay*.
- Three "rightmost" columns show many [non-blue] *bays*.
- Most of the column "tops" and "bottoms" show *stack cranes*.
- The four leftmost columns show *stack cranes* at *bays* "somewhere in the middle" of a column.

12.2.2 Container Stowage Area and Quay Cranes

12.2.3 Container Vessel Routes



12.2.4 Containers

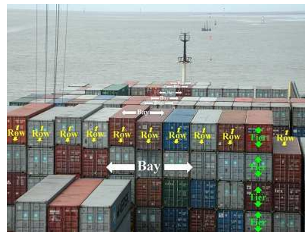
12.2.4.1 40 and 20 Feet Containers

12.2.4.2 Container Markings

12.2.5 Container Vessels

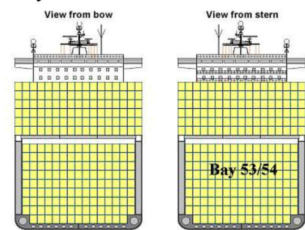
Quay cranes and *vessel* showing *row* of *aft (rear) bay*.

12.2.6 Container Stowage Area: Bays Rows, Stacks and Tier



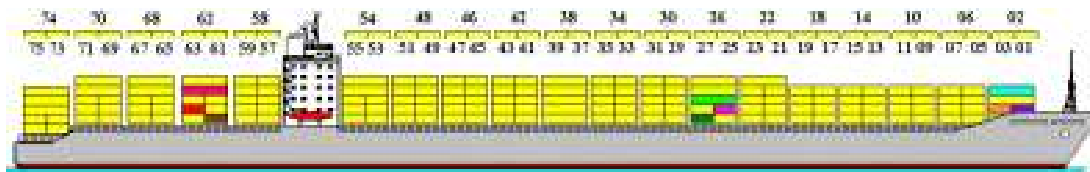
Bay, Row, Tier Numbers.

Row Numbers



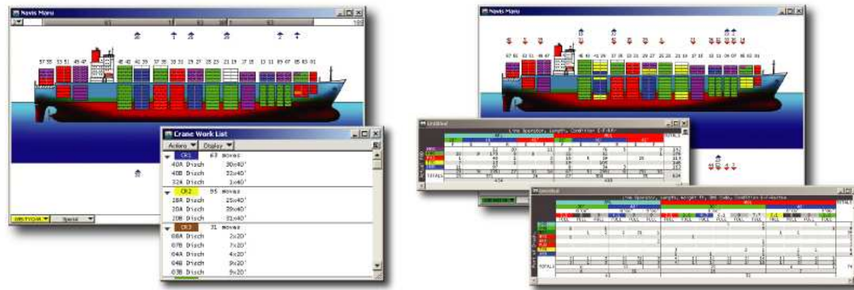
Cross section of a Bay.

Tier Numbers.



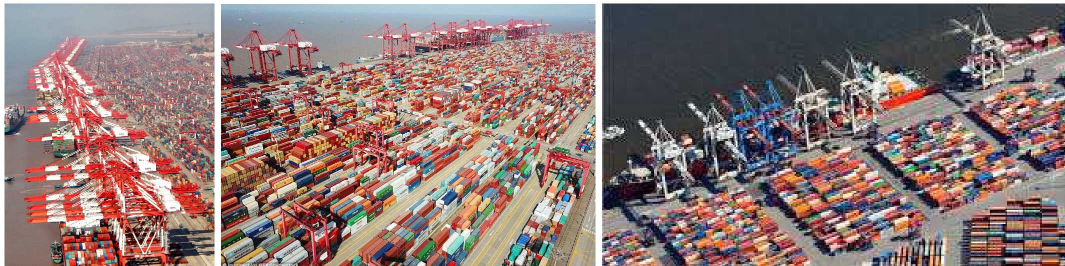
Bay Numbering

12.2.7 Stowage Software



12.2.8 Quay Cranes

12.2.9 Container Stowage Area and Stack Cranes



12.2.10 Container Stowage Area

12.2.11 Quay Trucks

12.2.12 Map of Shanghai and YangShan

12.3 SECT

- *Shanghai East Container Terminal*
  - is the joint venture terminal of
  - *APM Terminals* and
  - *Shanghai International Port Group*
  - in *Wai Gao Qiao* port area of *Shanghai*.
- No.1 Gangjian Road, Pudong New District, Shanghai, China

12.4 Main Behaviours

- From consumer/origin to consumer/final destination:
  - **container loads** onto **land truck**;
  - **land truck travels** to **terminal stack**;
  - **container unloads** by means of **terminal stack crane** from **land truck** onto **terminal stack**.
  - **Container moves** from **stack** to **vessel**:
    - **terminal stack crane moves container** from **terminal stack** to **quay truck**,
    - **quay truck moves container** from **terminal stack** to **quay**,
    - **quay crane moves container** to **top of a vessel stack**;

- **Container moves** on **vessel** from **terminal** to **terminal**:
  - Either container is unloaded at a next terminal port to a stack and from there to a container truck
  - or: container is unloaded at a next terminal port to a stack and from there to a next container vessel.

12.4.1 A Diagram

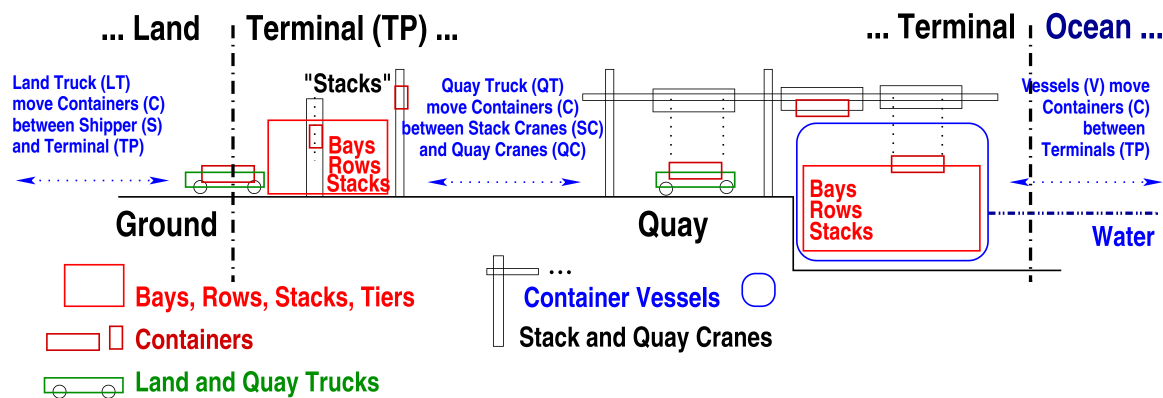


Fig. 1: Container Terminal Ports, I
A "from the side" snapshot of terminal port activities

12.4.2 Terminology - a Caveat

Bay [109]: contains indexed set of *rows* (of stacks of containers).
Container : smallest unit of central (i.e., huge) concern !
Container Stowage Area : An area of a vessel or a terminal where containers are stored, during voyage, respectively awaiting to be either brought out to shippers or onto vessels.
Crane :

Stack Crane : moves *containers* between *land* or *terminal trucks* and *terminal stacks*.
Quay Crane : moves *containers* between [*land* or] *terminal trucks* and *vessels*.

Land : ... as you know it ...
Ocean : ... as you know it ...
Shipper : arranges shipment of containers with container lines
Quay : area of terminal next to vessels (hence water).
Row : contains indexed set of *stacks* (of containers).
Stack : contains indexed set of *containers*.
We shall also, perhaps confusingly, use the term stack referring to the land-based bays of a terminal.
Terminal : area of land and water between land and ocean equipped with container stowage area, and stack and quay cranes, etc.

[109] The terms introduced in this section are mine. They are most likely not the correct technical terms of the container shipping and stowage trade. I expect to revise this section, etc.

Truck :

- **Land Truck** : privately operated truck transport *containers* between *shippers* and *stack cranes*.
- **Quay Truck** : terminal operated special truck transport *containers* between *stack cranes* and *quay cranes*.

Tier : index of *container* in *stack*.
Vessel : contains a *container stowage area*.

12.4.3 Assumptions

Without loss of generality we can assume that there is exactly one stack crane per land-basederminal stack; quay cranes each serve exactly one bay on a vessel; there are enough quay cranes to serve all bays of any berthed vessel; quay trucks may serve any (quay and stack) crane; land trucks may serve more than one terminal; et cetera.

12.5 Endurants

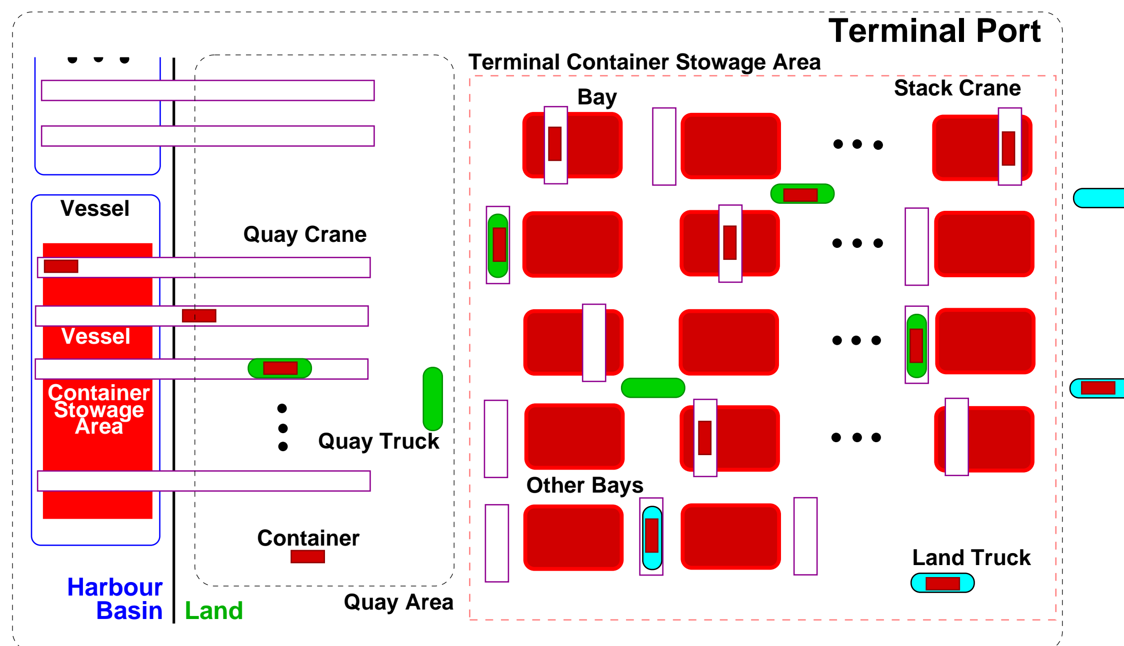


Fig. 2: Container Terminal Ports, II
A "from above" snapshot of terminal port activities

We refer to [53, Sects. 3., 4., and 5.].

Our model focuses initially on parts, that is, manifest, observable phenomena. Our choice of these is expected to be subject to serious revision once we ... MORE TO COME ...

12.5.1 Parts

We refer to [53, Sect. 3.3].

Our model has, perhaps arbitrarily, focused on just some of the manifest, i.e., observable parts of a domain of container terminal ports. We shall invariable refer to container terminal ports as either container terminals, or terminal ports, tp:TP, or just terminals. We expect revisions to the decomposition as shown as we learn more from professional stakeholders, e.g., *APM Terminals/SECT*, Shanghai.

1. In the container line industry, CLI, we can observe
2. a structure, TPS, of all terminal ports, and from each such structure, an indexed set, TPs, of two or more container *terminal* ports, TP;
3. a structure, VS, of all container vessels, and from each such structure, an indexed set, Vs, of one or more container *vessels*, V; and
4. a structure, LTS, of all land trucks, and from each such structure, a non-empty, indexed set, LTs of land trucks, LT;

```
type
1   CLI
2   STPs,  TPs = TP-set, TP
3   SVs,    Vs = V-set, V
4   SLTs,   LTs = LT-set, LT
value
2   obs_STPs: CLI → STPs, obs_TPs: STPs → TPs
3   obs_SVs:  CLI → SVs,   obs_Vs: SVs → Vs
4   obs_SLTs:  CLI → SLTs, obs_LTs: SLTs → LTs
axiom
2   ∀ cli:CLI•card obs_TPs(obs_STPs(cli))≥2
3      ∧ card obs_Vs(obs_SVs(cli))≥1
4      ∧ card obs_LTs(obs_SLTs(cli))≥1
```

12.5.1.1 Terminal Ports

In a terminal port, tp:TP, one can observe

5. a [composite] container stowage area, csa:CSA;
6. a structure, sqc:SQC, of quay cranes, and from that, a non-empty, indexed set, qcs:QCs, of one or more quay cranes, qc:QC;
7. structure, sqt:SQT, of quay trucks, and from that a non-empty, indexed set, qts:QTs, of quay trucks, qt:QT;
8. a structure, Scs:SCS, of stack cranes, and from that a non-empty, indexed set, scs:SCs, of one or more stack cranes, sc:SC;
9. a[n atomic] quay[110], q:Q[111]; and
10. a[n atomic] terminal port monitoring and control center, mcc:MCC.

[110] We can, without loss of generality, describe a terminal as having exactly one quay (!) – just as we, again without any loss of generality, describe it as having exactly one container stowage area.

[111] *Quay:* a long structure, usually built of stone, where boats can be tied up to take on and off their goods.
Pronunciation: key.
Thesaurus: berth, jetty, key, landing, levy, slip, wharf

```
type
5  CSA
6  SQC, QCs = QC-set, QC
7  SQT, QTs = QT-set, QT
8  SCS, SCs = SC-set, SC
9  Q
10 MCC
value
5  obs_CSA: TP → CSA
6  obs_SQC: TP → SQC, obs_QCs: SQC → QCs
7  obs_SQT: TP → SQT, obs_QTs: SQT → QTs
8  obs_SCS: TP → SCS, obs_SCs: SCS → SCs
9  obs_Q: TP → Q
10 obs_MCC: TP → MCC
axiom
6  ∀ sqc:SQC•card obs_QCs(sqc)≥1
7  ∀ sqt:SQT•card obs_QTs(sqt)≥1
8  ∀ scs:SCS•card obs_SCs(scs)≥1
```

12.5.1.2 Quays

Although container terminal port quays can be modelled as composite parts we have chosen to describe them as atomic. We shall subsequently endow the single terminal port quay with such attributes as quay segments, quay positions and berthing[112].

12.5.1.3 Container Stowage Areas: Bays, Rows and Stacks

11. From a container stowage area one can observe a non-empty indexed set of bays,
12. From a bay we can observe a non-empty indexed set of rows.
13. From a row we can observe a non-empty indexed set of stacks.
14. From a stack we can observe a possibly empty indexed set of containers.

```
type
11 BAYS, BAYs = BAY-set, BAY
12 ROWS, ROWs = ROW-set, ROW
13 STKS, STKs = STK-set, STK
14 CONS, CONs = CON-set, CON
value
11 obs_BAYS: CSA → BAYS, obs_BAYs: BAYS → BAYs
12 obs_ROWS: BAY → ROWS, obs_ROWs: ROWS → ROWs
13 obs_STKS: ROW → STKS, obs_STKs: STKS → STKs
14 obs_CONS: STK → CONS, obs_CONs: CONS → CONs
axiom
11 ∀ bays:BAYs • card bays > 0
12 ∀ rows:ROWs • card rows > 0
13 ∀ stks:STKs • card stks > 0
```

[112] Berth: Sufficient space for a vessel to maneuver; a space for a vessel to dock or anchor; (whether occupied by vessels or not). Berthing: To bring (a vessel) to a berth; to provide with a berth.

12.5.1.4 Vessels

From (or in) a vessel one can observe

15. [5] a container stowage area
16. and some other parts.

```
type
5  CSA
16  ...
value
5  obs_CSA: V → CSA
16  ...
```

12.5.1.5 Functions Concerning Container Stowage Areas

17. One can calculate
18. the set of all container storage areas:
19. of all terminal ports together with those
20. of all container lines.

```
value
17  cont_stow_areas: CLI → CSA-set
18  cont_stow_areas(cli) ≡
19     {obs_CSA(tp)|tp:TP•tp ∈ obs_TPs(obs_TPS(cli))}
20   ∪ {obs_CSA(cl)|cl:CL•cl ∈ obs_CLs(obs_CLS(cli))}
```

One can calculate the containers of

21. a stack,
22. a row,
23. a bay, and
24. a container stowage area.

```
value
21  extr_cons_stack: STK → CONs
21  extr_cons_stack(stk)≡obs_CONs(obs_CONS(stk))
22  extr_cons_row: ROW → CONs
22  extr_cons_row(row) ≡
22    {obs_CONs(obs_CONS(stk))|stk:STK•stk∈obs_STKs(obs_STKS(stk))}
23  extr_cons_bay: BAY → CONs
23  extr_cons_bay(bay) ≡
23    {obs_CONs(obs_CONS(row))|row:ROW•row∈obs_ROWs(obs_ROWS(bay))}
24  extr_cons_csa: CSA → CONs
24  extr_cons_csa(csa) ≡
24    {obs_CONs(obs_CONS(bay))|bay:BAY•bay∈obs_BAYs(obs_BAYS(csa))}
```

12.5.1.6 Axioms Concerning Container Stowage Areas

25. All rows contain different, i.e. distinct containers.

26. All bays contain different, i.e. distinct containers.
27. All container stowage areas contain different, i.e. distinct containers.

```
value
25  ∀ cli:CLI •
25    ∀ csa,csa′:CSA•{csa,csa′}⊆cont_stow_areas(cli) •
25      ∀ row,row′:ROW •
25        {row,row′}⊆obs_ROWs(obs_ROWS(csa))∪obs_ROWs(obs_ROWS(csa′))⇒
25        extr_cons_row(row) ∩ extr_cons_row(row′) = {} ∧
26      ∀ bay,bay′:BAY •
26        {bay,bay′}⊆obs_ROWs(obs_ROWS(csa))∪obs_ROWs(obs_ROWS(csa′))⇒
26        extr_cons_bay(bay) ∩ extr_cons_bay(bay′) = {} ∧
27        extr_cons_csa(csa) ∩ extr_cons_csa(csa′) = {}
```

12.5.1.7 Stacks

An aside: We shall use the term 'stack' in two senses: (i) as a component of container storage area bays; and (ii) to refer to the collection of stacks in a bay of a terminal container storage area.

28. Stacks are created empty, and hence stacks can be *empty*.
29. One can *push* a *container* onto a *stack* and obtain a *non-empty stack*.
30. One can *pop* a *container* from a *non-epmpty stack* and obtain a pair of a *container* and a possibly empty *stack*.

```
value
28  empty: () → STK, is_empty: STK → Bool
29  push: CON × STK → STK
30  pop: STK →~ (CON × STK)
axiom
28  is_empty(empty()), ~is_empty(push(c,stk))
29  pop(push(c,stk)) = (c,stk)
30    pre pop(stk),pop(push(c.stk)): ~is_empty(stk)
30  pop(empty()) = chaos
```

12.5.2 Terminal Port Command Centers

12.5.2.1 Discussion

We consider terminal port monitoring & control command centers to be atomic parts. The purpose of a terminal port command center is to monitor and control the allocation and servicing (berthing) of any visiting vessel to quay positions and by quay cranes, the allocation and servicing of vessels by quay cranes, the allocation and servicing of quay cranes by quay trucks, the allocation and servicing of quay trucks to quay cranes, containers and terminal stacks, the allocation and servicing of land trucks to containers and terminal stacks, This implies that there are means for communication between a terminal command center and vessels, quay cranes, stack cranes, quay trucks, land trucks, terminal stacks and containers.

12.5.2.2 Justification

We shall justify the concept of terminal monitoring & control, i.e., command centers. First, using the *domain analysis & description* approach of [53], we know that we are going, through a transcendental deduction, to model certain parts as behaviours. These parts, we decide, after some analysis that we forego, to be vessels, quay cranes, quay trucks, stack cranes stacks, land trucks, and containers. Behaviours are usually like actors:they can instigate actions. But we decide, in our analysis, that some of these behaviours, quay cranes, quay trucks, stack cranes and stacks, are *"passive" actors:* are behaviourally not endowed with being able to initiate "own" actions. Instead, therefore, of all these behaviours, being able to communicate directly, pairwise, as loosely indicated by the figures of Pages 312 and 313, we model them to communicate *via* their terminal command centers.

This is how we justify the introduction of the concept of terminal command centers. They are an abstraction. In *"ye olde days"* you could observe, not one, but, perhaps, a hierarchy of terminal port offices, staffed by people, [each office, each group of staff] with its set of duties: communicating (by radio-phone) with approaching [and departing] vessels; scheduling quay positions, quay cranes and quay trucks; managing the operation of cranes and trucks; and, on a large scale, calculating stowage: on vessels and in terminals. Today, *"an age of ubiquitous computing"*, most of these offices and their staff are replaced by electronics: sensors, actuators, communication and computing, and with massive stowage data processing: where should containers be stowed on board vessels and in terminals so as to near-optimise all operations.

12.5.3 Unique Identifications

We refer to [53, Sect. 5.1].

31. Vessels have unique identifiers.
32. Quay cranes have unique identifiers.
33. Quay trucks have unique identifiers.
34. Stack cranes have unique identifiers.
35. Bays ("Stacks") of terminal container stowage areas have unique identifiers, cf. Item 39.
36. Land trucks have unique identifiers.
37. Terminal port command centers have unique identifiers.
38. Containers have unique identifiers.
39. Bays of container stowage areas have unique identifiers.
40. Rows of a bay have unique identifiers.
41. Stacks of a row have unique identifiers.
42. The part unique identifier types are mutually disjoint.

```
type
31 VI
32 QCI
33 QTI
34 SCI
35 TBI
36 LTI
37 MCCI
38 CI
39 BI
40 RI
41 SI
```

```
value
31 uid_V: V → VI
32 uid_QC: QC → QCI
33 uid_QT: QT → QTI
34 uid_SC: SC → SCI
34 uid_TBI: BAY → TBI
35 uid_LT: LT → LTI
37 uid_MCC: MCC → MCCI
37 uid_CON: CON → CI
34 uid_BAY: BAY → BI
35 uid_ROW: ROW → RI
36 uid_STK: STK → SI
```

```
axiom
42 VI, QCI, QTI, SCI, TBI, LTI, MCCI, CI, RI and SI mutually disjoint
42 TBI ⊂ BI
```

12.5.3.1 Unique Identifiers: Distinctness of Parts

43. If two containers are different then their unique identifiers must be different.

axiom
43 ∀ con,con′:CON • con ≠ con′ ⇒ uid_CON(con) ≠ uid_CON(con′)

The same distinctness criterion applies to stacks, rows, bays, container storage areas, terminal ports, cranes, vessels, etc.

12.5.3.2 Unique Identifiers: Two Useful Abbreviations

Container positions within a container stowage area can be represented in two ways:

44. by a triple of a bay identifier, a row identifier and a stack identifier, and
45. by these three elements and a tier position (i.e., position within a stack).

44 BRS = BI × RI × SI
45 BRSP = BI × RI × SI × **Nat**
axiom
45 ∀ (bu,ri,si,n):BRSP • n>0

12.5.3.3 Unique Identifiers: Some Useful Index Set Selection Functions

46. From a container stowage area once can observe all bay identifiers.
47. From a bay once can observe all row identifiers.
48. From a row once can observe all stack identifiers.
49. From a virtual container storage area, i.e., an icsa:iCSA, one can extract all the unique container identifiers.

value
46 xtr_BIs: CSA → BI-**set**
46 xtr_BIs(csa) ≡ {uid_BAY(bay)|bay:BAY•bay ∈ xtr_BAYs(csa)}

46 xtr_RIs: BAY → RI-**set**
47 xtr_RIs(bay) ≡ {uid_ROW(bay)|row:ROW•row ∈ obs_ROWs(bay)}

46 xtr_SIs: ROW → SI-**set**
48 xtr_SIs(row) ≡ {uid_STK(row)|stk:STK•stk ∈ obs_STKs(row)}

49 xtr_CIs: iCSA → CI-**set**
49 xtr_CIs(icsa) ≡
49 ... [to come] ...

12.5.3.4 Unique Identifiers: Ordering of Bays, Rows and Stacks

The bays of a container stowage area are usually ordered. So are the rows of bays, and stacks of rows. Ordering is here treated as *attributes* of container stowage areas, bays and stacks. We shall treat *attributes* further on.

12.5.4 States, Global Values and Constraints

12.5.4.1 States

50. We postulate a container line industry *cli*:CLI.
    From that we observe, successively, all parts:
51. the set, *cs*:C-**set**, of all containers;
52. the set, *tps*:TPs, of all terminal ports;
53. the set, *vs*:Vs, of all vessels; and
54. the set, *lts*:LTs, of all land trucks.

value
50 *cli*:CLI
51 *cs*:C-**set** = obs_Cs(obs_CS(*cli*))
52 *tps*:TP-**set** = obs_TPs(obs_TPS(*cli*))
53 *vs*:V-**set** = obs_Vs(obs_VS(*cli*))
54 *lts*:LTs = obs_LTs(obs_LTS(*cli*))

We can observe

55. *csas*:CSA-**set**, the set of all terminal port container stowage areas of all terminal ports;
56. *bays*:BAY-**set**, the terminal port bays of all terminals;
57. the set, *qcs*:QC-**set**, of all quay cranes of all terminals;
58. the set, *qts*:QT-**set**, of all quay trucks of all terminal ports; and
59. the set, *scs*:SC-**set**, of all terminal (i.e., stack) cranes of all terminal ports.

value
55 *csas*:CSA-**set** = {obs_CSA(tp)|tp:TP•tp ∈ *tps*}
55 *bays*:BAY-**set** = {obs_BAY(csa)|csa:CSA•csa ∈ *csas*}
57 *qcs*:QC-**set** = {obs_QCs(obs_QCS(tp))|tp:TP•tp ∈ *tps*}
58 *qts*:QT-**set** = {obs_QTs(obs_QTS(tp))|tp:TP•tp ∈ *tps*}
59 *scs*:SC-**set** = {obs_SCs(obs_SCS(tp))|tp:TP•tp ∈ *tps*}

12.5.4.2 Unique Identifiers

Given the generic parts outlined in Sect. **12.5.4.1** we can similarly define generic sets of unique identifiers.

60. There is the set, *c_uis*, of all container identifiers;
61. the set, *tp_uis*, of all terminal port identifiers;
62. the set, *mcc_uis*, of all terminal port command center identifiers;
63. the set, *v_uis*, of all vessel identifiers;
64. the set, *qc_uis*, of quay crane identifiers of all terminal ports;
65. the set, *qt_uis*, of quay truck identifiers of all terminal ports;
66. the set, *sc_uis*, of stack crane identifiers of all terminal ports;
67. the set, *stk_uis*, of stack identifiers of all terminal ports;
68. the set, *lt_uis*, of all land truck identifiers; and
69. the set, *uis*, of all vessel, crane and truck identifiers.

value
60 *c_uis*:CI-**set** = {uid_C(c)|c:C•c∈*cs*}
61 *tp_uis*:TPI-**set** = {uid_TP(tp)|tp:TP•tp∈*tps*}

62 mcc_uis:TPI-**set** = {uid_MCC(obs_MCC(tp))|tp:TP•tp∈tps}
63 v_uis:VI-**set** = {uid_V(v)|v:V•v∈vs}
64 qc_uis:QCI-**set** = {uid_QC(qc)|qc:QC•qc∈qcs}
65 qt_uis:QTI-**set** = {uid_QT(qt)|qt:QT•qt∈qts}
66 sc_uis:SCI-**set** = {uid_SC(sc)|sc:SC•sc∈scs}
67 stk_uis:BI-**set** = {uid_BAY(stk)|stk:BAY•stk∈$stks$}
68 lt_uis:LTI-**set** = {uid_LL(lt)|lt:LT•lt∈lts}
69 uis:(VI|QCI|QTI|SCI|BI|LTI)-**set** = v_uis∪qc_uis∪qt_uis∪sc_uis∪stk_uis∪ lt_uis

70. the map, *tpmcc_idm*, from terminal port identifiers into the identifiers of respective command centers;
71. the map, *mccqc_idsm*, from command center identifiers into the set of quay crane identifiers of respective ports;
72. the map, *mccqt_idsm*, from command center identifiers into the identifiers of quay trucks of respective ports;
73. the map, *mccsc_idsm*, from command center identifiers into the identifiers of quay trucks of respective ports; and
74. the map, *mccbays_idsm*, from command center identifiers into the set of bay identifiers (i.e., "stacks") of respective ports;

value
70 $tpmcc_idm$:(TI $\overrightarrow{m}$ MCCI) = [uid_TP(tp)↦uid_MCC(obs_MCC(tp))|tp:TP•tp ∈ tps]
71 $mccqc_idsm$:(MCCI $\overrightarrow{m}$ QCI-**set**)
71 = [$tpmcc_uim$(uid_TP(tp)) ↦ { uid_QC(qc)
71 | qc:QC • qc ∈ obs_QCs(obs_QCS(tp)) } | tp:TP•tp ∈ tps]
72 $mccqt_idsm$:(MCCI $\overrightarrow{m}$ QTI-**set**) =
72 = [$tpmcc_uim$(uid_TP(tp)) ↦ { uid_QT(qt)
72 | qt:QT • qt ∈ obs_QTs(obs_QTS(tp)) } | tp:TP•tp ∈ tps]
73 $mccsc_idsm$:(MCCI $\overrightarrow{m}$ SCI-**set**)
73 = [$tpmcc_uim$(uid_TP(tp)) ↦ { uid_SC(sc)
73 | sc:SC • sc ∈ obs_SCs(obs_SCS(tp)) } | tp:TP•tp ∈ tps]
74 $mccbays_idsm$:(MCCI $\overrightarrow{m}$ BI-**set**)
74 = [$tpmcc_uim$(uid_TP(tp)) ↦ { uid_B(b)
74 | b:BAY•b ∈ obs_BAYs(obs_BAYS(obs_CSA(tp)))} | tp:TP•tp ∈ tps]

12.5.4.3 Some Axioms on Uniqueness

TO BE WRITTEN

12.5.5 Mereology

We refer to [53, Sect. 5.2].

12.5.5.1 Physical versus Conceptual Mereology

We briefly discuss a distinction that was not made in [53]: whether to base a mereology on *physical connections* or on *functional* or, as we shall call it, *conceptual relations*. We shall, for this domain

model, choose the conceptual view. The physical mereology view can be motivated, i.e. justified, from the figures on pages 312 and 313. The conceptual view is chosen on the basis of the justification of the terminal command centers, cf. Sect. **12.5.2** on page 317. We shall model physical mereology as attributes.[113]

12.5.5.2 Vessels

12.5.5.2.1 Physical Mereology:

75. Vessels are physically "connectable" to quay cranes of any terminal port.

type
75 Phys_V_Mer = QCI-**set**
value
75 attr_Phys_V_Mer: V → Phys_V_mer

12.5.5.2.2 Conceptual Mereology:

76. Container vessels can potentially visit any container terminal port, hence have as [part of] their mereology, a set of terminal port command center identifiers.

type
76 V_Mer = MCCI-**set**
value
76 mereo_V: V → V_Mer
axiom
76 ∀ v:V • v ∈ *vs* ⇒ mereo_V(v) ⊆ *mcc_uis*

12.5.5.3 Quay Cranes

12.5.5.3.1 Physical Mereology:

In modelling the physical mereology, though as an attribute, of quay cranes, we need the notion of quay positions.

77. Quay cranes are, at any time, positioned at one or more adjacent quay positions of an identified segment of such.

type
77 Phys_QC_Mereo = QPSId × QP*
value
77 attr_Phys_QC: QC → Phys_QC_Mereo

78. The quay positions, qcmereo = (qpsid,qpl):QCMereo, must be proper quay positions of the terminal,
79. that is, the segment identifier, qpsid, must be one of the terminal,

[113] Editorial note: Names of physical and of conceptual mereologies have to be "streamlined". As now, they are a "mess" !

80. and the list, qpl, must be contiguously contained within the so identifier segment.

```
axiom ∀ tp:TP,
78 let q = obs_Q(tp), qcs = obs_QCs(obs_QCS(tp)) in
79   ∀ q:Q • q ∈ qcs ⇒
79      let (qpsid,qpl) = obs_Mereo(q), qps = attr_QPSs(q) in
79      qpsid ∈ dom qps
80   ∧ ∃ i,j:Nat • {i,j} ∈ inds qpl ∧ ⟨(qps(qpsi))[k]|i≤k≤j⟩ = qpl
78 end end
```

12.5.5.3.2 Conceptual Mereology:

The conceptual mereology is simpler.

81. Quay cranes are conceptually related to the command center of the terminal in which they are located.

```
type
81   QC_Mer = MCCI
value
81   mereo_QC: QC → QC_Mer
```

12.5.5.4 Quay Trucks

12.5.5.4.1 Physical Mereology:

82. Quay trucks are physically "connectable" to quay and stack cranes.

```
type
82  Phys_QT_Mer = QCI-set × QCI-set
value
82  attr_Phys_QT_Mer: QT → Phys_QT_Mer
```

12.5.5.4.2 Conceptual Mereology:

83. Quay trucks are conceptually connected to the command center of the terminal port of which they are a part.

```
type
83   QT_Mer = MCCI
value
83   mereo_QT: QT → QT_Mer
```

12.5.5.5 Stack Cranes

12.5.5.5.1 Physical Mereology:

84. Terminal stack cranes are positioned to serve one or more terminal area bays, one or more quay trucks and one or more land trucks.
85. The terminal stack crane positions are indeed positions of their terminal
86. and no two of them share bays.

```
type
84  Phys_SCmereo = s_bis:BI-set × s_qtis:QTI-set × s_ltis:LTI-set
axiom
84  ∀ (bis,qtis,ltis):Phys_SCmereo•bis≠{} ∧ qtis≠{} ∧ ltis≠{}
value
84  Phys_SCmereo: SC → Phys_SCmereo
axiom
84  ∀ tp:TP •
84     let csa=obs_CSA(tp), bays=obs_BAYs(obs_BAYS(csa)), scs=obs_SCs(obs_SCS(tp)) in
85     ∀ sc:SC•sc ∈ scs ⇒ Phys_SCmereo(sc) ⊆ xtr_BIs(csa)
86  ∧ ∀ tp′,tp′′:TP•{tc′,tc′′}⊆tcs ∧ tc′≠tc′′
86        ⇒ s_bis(Phys_SCmereo(tc′)) ∩ s_bis(Phys_SCmereo(tc′′))={} end
```

12.5.5.5.2 Conceptual Mereology:

The conceptual stack crane mereology is simple:

87. Each stack is conceptually related to the command center of the terminal at which it is located.

```
type
87  SC_Mer = MCCI
value
87  mereo_SC: SC → SC_Mer
```

12.5.5.6 Container Stowage Areas

12.5.5.6.1 Bays, Rows and Stacks:

The following are some comments related to, but not defining a mereology for container stowage areas.

88. A bay of a container stowage area
    a. has either a predecessor
    b. or a successor,
    c. or both (and then distinct).
    d. No row cannot have neither a predecessor nor a successor.
89. A row of a bay has a predecessor and a successor, the first stack has no predecessor and the last stack has no successor.
90. A stack of a row has a predecessor and a successor, the first stack has no predecessor, and the last stack has no successor.

value
88 BAY_Mer: BAY → ({|'nil'|}|BI) × (BI|{|'nil'|})
89 ROW_Mer: ROW → ({|'nil'|}|RI) × (RI|{|'nil'|})
90 STK_Mer: STK → ({|'nil'|}|SI) × (SI|{|'nil'|})
axiom
88 ∀ csa:CSA • **let** bs = obs_BAYs(obs_BAYS(csa)) **in**
88 ∀ b:BAY • b ∈ bs ⇒
88 **let** (nb,nb′) = mereo_BAY(b) **in**
88 **case** (nb,nb′) **of**
88a ('nil',bi) → bi ∈ xtr_BIs(csa),
88b (bi,'nil') → bi ∈ xtr_BIs(csa),
88d ('nil','nil') → **chaos**,
88c (bi,bi′) → {bi,bi′} ⊆ xtr_BIs(csa) ∧ bi≠bi′
88 **end end end**
89 as for rows
90 as for stacks

12.5.5.7 Bay Mereology

12.5.5.7.1 Physical Vessel Bay Mereology:

91. A vessel bay is topologically related to the vessel on board of which it is placed and to the set of all quay cranes of all terminal ports.

type
91 Phys_VES_BAY_Mer = VI × QCI-**set**

12.5.5.7.2 Conceptual Vessel Bay Mereology:

92. A vessel bay is conceptually related to the set of all command centers of all terminal ports.

type
92 V_BAY_Mer = MCCI-**set**

12.5.5.7.3 Physical Terminal Port Bay (cum Stack) Mereology:

93. A terminal bay (cum stack) is topologically related to the stack cranes of a given terminal port and all land trucks.

type
93 Phys_STK_Mer = SCI-**set** × LTI-**set**

12.5.5.7.4 Conceptual Terminal Port Bay (cum Stack) Mereology:

94. A terminal port bay is conceptually related to the command center of its port.

```
type
94  T_BAY_Mer = MCCI
```

12.5.5.8 Land Trucks

12.5.5.8.1 Physical Mereology:

95. Land trucks are physically “connectable” to stack cranes – of any port.

```
type
95  Phys_LT_Mer = SCI-set
value
95  attr_Phys_LT_Mer: LT → Phys_LT_Mer
```

12.5.5.8.2 Conceptual Mereology:

96. Land trucks are conceptually connected to the command centers of any terminal port.

```
type
96  LT_Mer = MCCI-set
value
96  mereo_LT: LT → LT_Mer
```

12.5.5.9 Command Center

Command centers are basically conceptual quantities. Hence we can expect the physical mereology to be the conceptual mereology.

97. Command centers are physically and conceptually connected toall vessels, all cranes of the terminal port of the command center, all quay trucks of the terminal port of the command center, all stacks (i.e., bays) of the terminal port of the command center, and all land trucks, and all containers.

```
type
97  MCC_Mer = VI-set×QCI-set×QTI-set×SCI-set×BI-set×LTI-set×CI-set
value
97  mereo_MCC: MCC → MCC_Mer
axiom
97  ∀ tp:TP • tp ∈ tps •
97    let qcs:QC-set • qcs = obs_QCs(obs_QCS(tp)),
97        qts:QT-set • qts = obs_QTs(obs_QTS(tp)),
97        scs:SC-set • scs = obs_SCs(obs_SCS(tp)),
97        bs:iBAY-set • bs = obs_Bs(obs_BS(obs_CSA(tp))) in
97    let vis:VI_set • vis = {uid_VI(v)|v:V•v ∈ vs},
97        qcis:QCI_set • qcis = {uid_QCI(qc)|qc:QC•qc ∈ qcs},
97        qtis:QTI_set • qcis = {uid_QTI(qc)|qt:QT•qt ∈ qts},
97        scis:SCI-set • scis = {uid_SCI(sc)|sc:SC•sc ∈ scs},
97        bis:iBAY-set • bis = {uid_BI(b)|b:iBAY•b ∈ bs},
```

```
97         ltis:LTI-set • ltis = {uid_LTI(lt)|lt:LT•lt ∈ lts},
97         cis:SCI-set • cis = {uid_CI(c)|c:C•c ∈ cs} in
97     mereo_MCC(obs_MCC(tp)) = (vis,qcis,scis,sis,bis,ltis,cis) end end
```

12.5.5.10 Conceptual Mereology of Containers

The physical mereology of any container is modelled as a container attribute.

98. The conceptual mereology is modelled by containers being connected to all terminal command centers.

```
type
98  C_Mer = MCCI-set
value
98  mereo_C: C → C_Mer
axiom
98  ∀ c:C • mereo_C(c) = mcc_uis
```

12.5.6 Attributes

We refer to [53, Sect. 5.3].

12.5.6.1 States

By a state we shall mean one or more parts such that these parts have *dynamic* attributes, in our case typically *programmable* attributes.

12.5.6.2 Actions

Actions apply to states and yield possibly updated states and, usually, some result values.

We shall in this section, Sect. **12.5.6**, on attributes, outline a number of *simple* (usually called *primitive*) actions of states. These actions are invoked by some behaviours either at their own volition, or in response to events occurring in other behaviours. The action outcomes are simple enough, but calculations resulting in these outcomes are not. Together the totality of the actions performed by the terminal's monitoring & control of vessels, cranes, trucks and the container stowage area, reflect the complexity of stowage handling.

12.5.6.3 Attributes: Quays

99. Quays are segmented into one or more quay segments, qs:QS, each with a sequence of one or more crane positions, cp:CP.
100. Quay segments and
101. crane positions are further unspecified.

```
type
99  QPOS = QS × CP* axiom ∀ (_,cpl):QPOS•cpl≠⟨⟩
```

```
100  QS
101  CP
```

12.5.6.4 Attributes: Vessels

102. A vessel is
    a. either at sea, at some *programmable* geographical location (longitude and latitude),
    b. or in some *programmable* terminal port – designated by the identifier of its command center and its quay position.
103. We consider the "remainder" of the vessel state as a programmable attribute – which we do not further define. The remainder includes all information about all containers, their bay/row/stack/tier positions, their bill-of-ladings, etc.
104. There may be other vessel attributes.

```
type
102    V_Pos == AtSea | InPort
102a   Longitude, Latitude
102a   AtSea :: Longitude × Latitude
102b   InPort :: MCCI × QPOS
103    VΣ
104    ...
value
102    attr_V_Pos: V → V_Pos
104    attr_VΣ: V → VΣ
104    attr_...: V → ...
axiom
102b   ∀ mkInPort(ti):InPort • ti ∈ tp_uis
```

12.5.6.5 Attributes: Quay Cranes

105. At any one time a quay crane may *programmably* hold a container or may not. We model the container held by a crane by the container identifier.
106. At any one time a quay crane is *programmably* positioned in a quay position within a quay segment.
107. Quay cranes may have other attributes.

```
type
105  QCHold == mkNil('nil') | mkCon(ci:CI)
106  QCPos = QSId × QP
107  ...
value
105  attr_QCHold: QC → QCHold
106  attr_QCPos: QC → QCPos
107  ...
```

12.5.6.6 Attributes: Quay Trucks

108. At any one time a land truck may *programmably* hold a container or may not. We model the container held by a quay truck by the container identifier.
109. Quay trucks may have other attributes.

Note that we do not here model the position of quay trucks.

```
type
108  QTHold == mkNil('nil') | mkCon(ci:CI)
109  ...
value
108  attr_QTHold: QT → QTHold
109  ...
```

12.5.6.7 Attributes: Terminal Stack Cranes

110. At any one time a stack crane may *programmably* hold a container or may not. We model the container held by a crane by the container identifier.
111. Stack cranes are *programmably* positioned at a terminal bay.
112. Stack cranes may have other attributes.

```
type
110  SCHold == mkNil('nil') | mkCon(ci:CI)
111  SCPos = BI
111  ...
value
110  attr_SCHold: SC → SCHold
111  attr_SCPos: SC → SCPos
112  ...
```

12.5.6.8 Attributes: Container Stowage Areas

113. Bays of container storage areas *statically* have total order.
114. Rows of bays *statically* have total order.
115. Stacks of rows *statically* have total order.

We abstract orderings in two ways.

```
type
113  BOm = BI →m Nat, BOl = BI*
114  ROm = RI →m Nat, ROl = RI*
115  SOm = SI →m Nat, SOl = SI*
axiom
113  ∀ bom:BOm•rng bom={1:card dom bom}, ∀ bol:BOl•inds bol={1:len bol}
114  ∀ rom:ROm•rng rom={1:card dom rom}, ∀ rol:ROl•inds rol={1:len rol}
115  ∀ som:SOm•rng som={1:card dom som}, ∀ sol:SOl•inds sol={1:len sol}
value
113  attr_BOm: CSA → BOm,  attr_BOl: CSA → BOl
114  attr_ROm: BAY → ROm,  attr_ROl: BAY → ROl
115  attr_SOm: ROW → SOm,  attr_SOl: ROW → SOl
```

CSAs, BAYs, ROWs and STKs have (presently further) *static d*escriptions[114] and terminal and vessel container stowage areas have definite numbers

116. of bays,
117. and any one such bay a definite number of rows,
118. and any one such row a definite number of stacks,
119. and any one such stack a maximum loading of containers.

```
type
116 CASd
117 BAYd
118 ROWd
119 STKd
value
116 attr_CSAD: CSA → BI →m CSAd
117 attr_BAYD: BAY → RI →m BAYd
118 attr_ROWD: ROW → SI →m ROWd
119 attr_STKD: STK → (Nat × STKd)
```

12.5.6.9 Attributes: Land Trucks

120. At any one time a land truck may *programmably* hold a container or may not. We model the container held by a land truck by the container identifier.
121. Land trucks also possess a further undefined *programmable* land truck state.
122. Land trucks may have other attributes.

Note that we do not here model the position of land trucks.

```
type
120 LTHold == mkNil('nil') | mkCon(ci:CI)
121 LTΣ
122 ...
value
120 attr_LTHold: LT → LTHold
121 attr_LTΣ: LT → LTΣ
122 ...
```

12.5.6.10 Attributes: Command Center

123. The `syntactic description`[115] of the spatial positions of quays, cranes and the container storage area of a terminal, TopLogDescr, is a *static* attribute.
124. The `syntactic description`[116] of the terminal state, i.e., the actual positions and deployment of vessels at quays, quay and stack cranes, quay and land trucks, and the actual container "contents" of these, TermΣDescr, is a *programmable* attribute.

[114] Such descriptions include descriptions of for what kind of containers a container stowage area, a bay, a row and a stack is suitable: flammable, explosives, etc.

[115] A `syntactic description` describes something, i.e., has some `semantics`, from which it is, of course, different.

[116] The `syntactic description` of the terminal state is, of course, not that state, but only its description. The terminal state is the combined states of all cranes, trucks and the container storage area.

```
type
123  TopLogDescr
124  MCCΣDescr
value
123  attr_TopLogDescr: MCC → TopLogDescr
124  attr_TermΣDescr: MCC → TermΣDescr
```

12.5.6.11 Attributes: Containers

125. A Bill-of-Lading[117] is a *static* container attribute. [118]

```
type
125  BoL
value
125  attr_BoL: C → BoL
```

126. At any one time a container is positioned either
    a. in a stack on a vessel: at sea or in a terminal, or
    b. on a quay crane in a terminal port, being either unloaded from or loaded onto a vessel, or
    c. on a quay truck to or from a quay crane, i.e., from or to a stack crane, in a terminal port, or
    d. on a stack crane in a terminal port, being either unloaded from a quay truck onto a terminal stack or loaded from a terminal stack onto a quay truck, or
    e. on a stack in a terminal port, or
    f. on a land truck, or
    g. idle.

    A container position is a *programmable* attribute.
127. There are other container attributes. For convenience we introduce an aggregate attribute: CAttrs for all attributes.

```
type
126   CPos  == onV | onQC | onQT | onSC | onStk | onLT | Idle
126a  onV   :: VI × BRSP × VPos
126a  VPos  == AtSea | InTer
126a  AtSea :: Geo
126a  InTer :: QPSid × QP+
126b  onQC  :: MCCI × QCI
126c  onQT  :: MCCI × QTI
126d  onSC  :: MCCI × SCI
126e  onStk :: MCCI × BRSP
```

[117] https://en.wikipedia.org/wiki/Bill_of_lading: A bill of lading (sometimes abbreviated as B/L or BoL) is a document issued by a carrier (or their agent) to acknowledge receipt of cargo for shipment. In British English, the term relates to ship transport only, and in American English, to any type of transportation of goods. A bill of Lading must be transferable, and serves three main functions: it is a conclusive receipt, i.e. an acknowledgment that the goods have been loaded; and it contains or evidences the terms of the contract of carriage; and it serves as a document of title to the goods, subject to the nemo dat rule. Bills of lading are one of three crucial documents used in international trade to ensure that exporters receive payment and importers receive the merchandise. The other two documents are a policy of insurance and an invoice. Whereas a bill of lading is negotiable, both a policy and an invoice are assignable. In international trade outside of the USA, Bills of lading are distinct from waybills in that they are not negotiable and do not confer title. The **nemo dat rule:** that states that the purchase of a possession from someone who has no ownership right to it also denies the purchaser any ownership title.

[118] For waybills see https://en.wikipedia.org/wiki/Waybill: A waybill (UIC) is a document issued by a carrier giving details and instructions relating to the shipment of a consignment of goods. Typically it will show the names of the consignor and consignee, the point of origin of the consignment, its destination, and route. Most freight forwarders and trucking companies use an in-house waybill called a house bill. These typically contain "conditions of contract of carriage" terms on the back of the form. These terms cover limits to liability and other terms and conditions

```
126f  onLT  :: MCCI × LTI
126g  Idle  :: {|"idle"|}
127   CAttrs
value
126   attr_CPos: C → CPos
127   attr_CAttrs: C → CAttrs
```

12.6 Perdurants

We refer to [53, Sect. 7].

12.6.1 A Modelling Decision

In the *transcendental interpretation* of parts into behaviours we make the following modelling decisions: All atomic and all composite parts become separate behaviours. But there is a twist. Vessels and terminal stacks are now treated as "atomic" behaviours. Containers that up till now were parts of container stowage areas on vessels and in terminal stacks are not behaviours embedded in the behaviours of vessels and terminal stacks, but are "factored" out as separate, atomic behaviours.

This modelling decision entails that container stowage areas, CSAs, of vessels and terminal stacks are modelled by replacing the [physical] containers of these CSAs with *virtual container stowage areas*, vir_CSAs. Where there "before" were containers there are now, instead, descriptions of these: their unique identifiers, their mereology, and their attributes.

12.6.2 Virtual Container Storage Areas

In our transition from endurants to perdurants we shall thus need a notion of container stowage areas which, for want of a better word, we shall call *virtual CSAs*. Instead of stacks embodying containers, they embody

128. container information: their unique identifier, mereology and attributes.

We must secure that no container is referenced more than once across the revised-model;

129. that is, that all ci:CIs are distinct.

```
type
5′   vir_CSA
11′  vir_BAY_s = vir_BAY-set, vir_BAY
12′  vir_ROW_s = vir_ROW-set, vir_ROW = vir_STK-set
13′  vir_STK = vir_STK-set, vir_STK
14′  vir_STK = CInfo*
128  CInfo = CI × CMereo × CAttrs
value
5′   attr_vir_CSA: TP → vir_CSA
11′  attr_vir_BAY_s: vir_CSA → vir_BAY_s, vir_BAY_s = vir_BAY-set, vir_BAY
11′  uid_vir_BAY: vir_BAY → BI
```

12′ attr_vir_ROW_s: vir_BAY → vir_ROW_s
axiom
129 [all CIs of all vir_CSAs are distinct]

12.6.3 Changes to The Parts Model

We revise the parts model of earlier:

type
2 STPs, TPs = TP-**set**, TP
3 SVs, Vs = V-**set**, V
value
2 obs_STPs: CLI → STPs, obs_TPs: STPs → TPs
3 obs_SVs: CLI → SVs, obs_Vs: SVs → Vs

We treat the former CSAs of terminal ports as a composite, concrete part, vir_BAY_m consisting of a set of atomic virtual bays, vir_BAY.

type
11′ vir_BAY_s = vir_BAY-**set**, vir_BAY
value
5 obs_BAY_s: TP → vir_BAY_s
5 uid_BAY: vir_BAY → BI

And we treat the former CSAs of vessels as a programmable attribute of vessels:

attr_vir_CSA: V → vir_CSA

12.6.4 Basic Model Parts

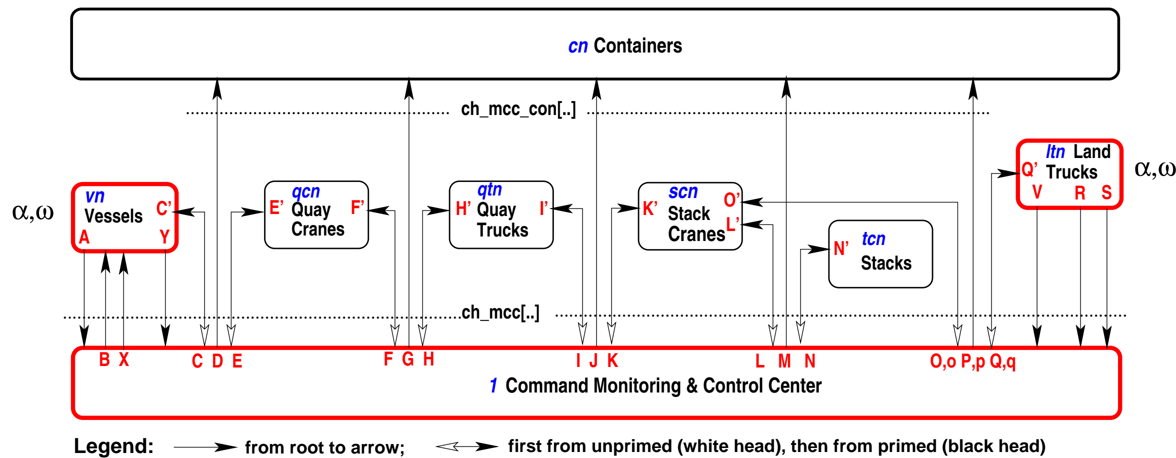


Fig. 3: The Container Terminal Behaviours[119]

There are c_n container behaviours, where c_n is the number of all containers of the system we are modelling. For each terminal port there is 1 controller behaviour, v_n vessel behaviours, where v_n is the number of vessels visiting that terminal port, qc_n quay crane behaviours, where qc_n is the number of quay cranes of that terminal port, qt_n quay truck behaviours, where qt_n is the number of quay trucks of that terminal port, lt_n land truck behaviours, where lt_n is the number of land trucks (of that terminal port), and tb_n terminal stack behaviours, where ts_n is the number of terminal bays of that terminal port.

The vessel, the land truck and the terminal monitoring & control [command] center behaviours are *pro-active:* At their own initiative (volition), they may decide to communicate with other behaviours. The crane, quay truck, stack and container behaviours are *passive:* They respond to interactions with other behaviours.

12.6.5 Actions, Events, Channels and Behaviours

We refer to [53, Sect. 7.1].

In building up to the behavioral analysis & description of the terminal container domain we first analyse the actions and events of that domain. These actions and events are the building blocks of behaviours.

Actions, to remind the reader, are explicitly performed by an actor, i.e., a behaviour, calculates some values and, usually, effect a state change.

Events *"occur to"* actors (behaviours), that is, are not initiated by these, but usually effect state changes.

12.6.6 Actions

We refer to [53, Sects. 7.1.5, 7.3.1].

[119] The labeling **A, B, C, D, ..., X, Y** may seem arbitrary, but isn't!

The unloading of containers from and the loading of container onto container stowage areas are modelled by corresponding actions on virtual container stowage areas. Vessels, land trucks and terminal monitoring & control centers, i.e., command centers, are here modelled as the only entities that can *initiate* actions.

12.6.6.1 Command Center Actions

12.6.6.1.1 Motivating the Command Center Concept:

We refer to the **[A,B,...,U]** labeled arrows of the figure on Page 334.

Imagine a terminal port. It has several vessels berthed along quays. It also has quay space, i.e., positions, for more vessels to berth. Berthed vessels are being serviced by several, perhaps many quay cranes. The totality of quay cranes are being serviced by [many more] quay trucks. The many quay trucks service several terminal bays, i.e., *stacks*. Land trucks are arriving, attending *stacks* and leaving. Quite a "busy scene". So is the case for all container terminal ports.

The concept of a *monitoring & control, i.e., a command* ***center***, is an abstract one; the figure on Page 334 does not show a part with a ***... center*** label. The *actions* of vessels and trucks, and the *events* of cranes, terminal stacks and trucks are either hap-hazard, no-one interferes, they somehow "just happen", or they are somehow co-ordinated.

Whether "free-wheeling" or "more-or-less coordinated" we can think of a *command center* as somehow *monitoring and controlling actions and events*.

Terminal *monitoring & control centers*, also interchangeably referred to as *command centers*, are thus where the logistics of container handling takes place.

You may think of this command center as receiving notices from vessels and land trucks as to their arrival and with information about their containers; thus building up awareness, i.e., a state, of the containers of all incoming and arrived vessels and land trucks, the layout of the terminal and the state of its container stowage area, the current whereabouts of vessels, cranes and trucks. Quite a formidable "state".

We shall therefore model the "comings" and "goings" of vessels, trucks, cranes and stacks as if they were monitored and controlled by a command center, In our modelling we are not assuming any form of efficiency; there is, as yet no notion of optimality, nor of freedom from mistakes and errors. Our modelling – along these lines – is "hidden" in action pre- and post-conditions and thus allows for any degree of internal non-determinism.

12.6.6.1.2 Calculate Next Transaction:

The *core* action of the command center is `calc_nxt_transaction`. We shall define `calc_nxt_transaction` only by its signature and a pair of **pre/post** conditions. In this way we do not have to consider *efficiency, security, safety, etc.,* issues. These, i.e., the efficiency, security, safety, etc., issues can "always" be included in an *requirements engineering* implementation of `calc_nxt_transaction`. Basically the `calc_nxt_transaction` has to consider which of a non-trivially large number of possible actions have to be invoked. They are listed in Items 131 to 137 below. The `calc_nxt_transaction` occurs in time, and occur repeatedly, endlessly, i.e., "ad-infinitum", At any time that `calc_nxt_transaction` is invoked the monitoring and control command center (mcc) is in some state. That state changes as the result of both monitoring actions and control actions. The `calc_nxt_transaction` therefore non-deterministically-internally chooses one among several possible alternatives. If there is no alternative, then a **skip** action is performed.

The command center, mcc, models the following actions and events: **[A]** the `update` of the mcc state, mccσ, in response to the vessel action that inform the mcc of the vessel arrival.

130. The result of a `calc_nxt_transaction` is an transaction designator, MCCTrans and a state change. There are several alternative designators. We mention some:
131. **[B]:** the `calculation` of `vessel` positions for [their] arrivals;
132. **[CDE]:** the `calculation` of `vessel` to `quay` crane container transfers;
133. **[FGH]:** the `calculation` of `quay` crane to `quay` `truck` container transfers;
134. **[IJK]:** the `calculation` of `quay` `truck` to `stack` crane container transfers;
135. **[LMN]:** the `calculation` of `stack` crane to `stack` container transfers;
136. **[OPQ]:** the `calculation` of `land truck` to `stack` crane container transfers;
137. **[X]:** the `calculation` that stowage, for a given vessel, has completed; and
138. the calculation that there is no next transaction that can be commenced.
139. The signature of the `calc_nxt_transaction` involves the unique identifier, mereology, static and programmable attributes, i.e., the state of the command center, and indicates that a command center transaction results and a next state "entered".
140. For this, the perhaps most significant action of the entire container terminal port operation, we "skirt" the definition and leave to a pair pf **pre/post** conditions that of characterising the result and next state.

```
type
130   MCCTrans == QayPos | VSQC_Xfer | QCQT_Xfer | QTSC_Xfer
130              |  SCSTK_Xfer | SCLT_Xfer | LT_Dept | VS_Dept | Skip
131 [B]:    QuayPos     :: VI × QPos
132 [CDE]:  VSQC_Xfer   :: VI × BRS × CI × QCI
133 [FGH]:  QCQT_Xfer   :: QCI × CI × QTI
134 [IJK]:  QTSC_Xfer   :: QTI × CI × SCI
135 [LMN]:  SCSTK_Xfer  :: SCI × CI × BRS
136 [OPQ]:  SCLT_Xfer   :: SCI × CI × LTI
137 [X]:    VS_Dept     :: VI
138         Skip        :: nil
value
139   calc_nxt_transaction: MCCI×mereoMCC×statMCC→MCCΣ→MCCTrans×MCCΣ
139   calc_nxt_transaction(mcci,mccmereo,mmstat)(mccσ) as (mcctrans,mccσ′)
140     pre: 𝒫_calc_nxt_trans((mcci,mccmereo,mccstat)(mccσ))
140     post: 𝒬_calc_nxt_trans((mcci,mccmereo,mccstat)(mccσ))(mcctrans,mccσ′)
```

The above mentioned actions are invoked by the command center in its endeavour to see containers moved from vessels to customers. A similar set of actions affording movement of containers customers to vessels, i.e., in the reverse direction: from land trucks to stack cranes, from stacks to quay trucks, from quay trucks to quay cranes, and from quay cranes to vessels, round off the full picture of all command center actions.

12.6.6.1.3 Command Center Action [A]: `update_mcc_from_vessel:`

141. Command centers
142. upon receiving arrival information, v_info, from arriving vessels, v_i, can update their state "accordingly".
143. We leave undefined the pre- and post-conditions.

```
value
141   update_mcc_from_vessel: VSMCC_MSG × MCC_Σ → MCC_Σ
142   update_mcc_from_vessel((vs_i,vir_csa,vs_info),mcc_σ) as mcc_σ′
143     pre: P_upd_mcc_f_v((vs_i,vir_csa,vs_info),mcc_σ)
143     post: Q_upd_mcc_f_v((vs_i,vir_csa,vs_info),mcc_σ)(mcc_σ′)
```

12.6.6.1.4 Command Center Action [B]: `calc_ves_pos`:

144. Command centers
145. can calculate, q_pos, the quay segment and quay positions for an arriving vessel, v_i.
146. We leave undefined the pre- and post-conditions.

value
144 calc_ves_pos: MCCI×MCC_mereo×TopLog×MCCΣ×VI → (QSId×QP*)×MCCΣ
145 calc_ves_pos(mcc_i,mcc_mereo,toplog,mcc_σ,v_i) **as** (q_pos,mcc_σ′)
146 **pre**: $P_{calc_ves_pos}$(mcc_i,mcc_mereo,toplog,mcc_σ,v_i)
146 **post**: $Q_{calc_ves_pos}$(mcc_i,mcc_mereo,toplog,mcc_σ,v_i)(q_pos,mcc_σ′)

12.6.6.1.5 Command Center Action [C-D-E]: `calc_ves_qc`

147. The command center non-deterministically internally calculates
148. a pair of a triplet: the bay-row-stack coordinates, brs, from which a top container, supposedly ci, is to be removed by quay crane qci, and a next command center state reflecting that calculation (and that the identified quay crane is being so alerted).
149. We leave undefined the relevant pre- and post-conditions

value
147 calc_ves_qc: MCCΣ → (BRS×CI×QCI) × MCCΣ
148 calc_ves_qc(mccσ) **as** ((brs,ci,qci),mccσ')
149 **pre**: $\mathcal{P}_{calc_ves_qc}$(mccσ)
149 **post**: $\mathcal{Q}_{calc_ves_qc}$(mccσ)((brs,ci,qci),mccσ')

12.6.6.1.6 Command Center Action [F-G-H]: `calc_qc_qt`

150. The command center non-deterministically internally
151. calculates a pair of a triplet: the identities of the quay crane from which and the quay truck to which the quay crane is to transfer a container, and an update command center state reflecting that calculation (and that the identified quay crane, container and truck are being so alerted).
152. We leave undefined the relevant pre- and post-conditions

value
150 calc_qc_qt: MCCΣ → (QCI×CI×QTI) × MCCΣ
151 calc_qc_qt(mccσ) **as** ((qci,ci,qti),mccσ')
152 **pre**: $\mathcal{P}_{calc_qc_qt}$(mccσ)
152 **post**: $\mathcal{Q}_{calc_qc_qt}$(mccσ)((qci,ci,qti),mccσ')

12.6.6.1.7 Command Center Action [I-J-K]: `calc_qt_sc`

153. The command center non-deterministically internally
154. calculates a pair of a triplet: the identities of a quay truck, a container, and a stack crane, and an update command center state reflecting that calculation (and that the identified quay truck, container and stack crane are being so alerted).
155. We leave undefined the relevant pre- and post-conditions

value
153 calc_qt_sc: MCCΣ → (QTI×CI>SCI) × MCCΣ
154 calc_qt_sc(mccσ) **as** ((qti,ci,sci),mccσ')
155 **pre**: $\mathcal{P}_{calc_qt_sc}$(mccσ)
155 **post**: $\mathcal{Q}_{calc_qt_sc}$(mccσ)((qti,ci,sci),mccσ')

12.6.6.1.8 Command Center Action [L-M-N]: `calc_sc_stack`

156. The command center non-deterministically internally calculates a pair:
157. a triplet of the identities of a stack crane, a container and a terminal bay/row/stack triplet and a new state that reflects this action.
158. We leave undefined the relevant pre- and post-conditions

value
156 calc_sc_stack: MCCΣ → (SCI×CI×BRS)×MCCΣ
157 calc_sc_stack(mccσ) **as** ((sci,ci,brs),mccσ′)
158 **pre**: $\mathcal{P}_{calc_sc_stack}$(mccσ)
158 **post**: $\mathcal{Q}_{calc_sc_stack}$(mccσ)((sci,ci,brs),mccσ′)

12.6.6.1.9 Command Center Action [N-M-L]: `calc_stack_sc`

159. The command center non-deterministically internally calculates a pair:
160. a triplet of a terminal bay/row/stack triplet and the identities of a container and a stack crane, and a new state that reflects this action.
161. We leave undefined the relevant pre- and post-conditions

value
159 calc_stack_sc: MCCΣ → (BRS×CI×SCI)×MCCΣ
160 calc_stack_sc(mccσ) **as** ((brs,ci,sci),mccσ′)
161 **pre**: $\mathcal{P}_{calc_stack_sc}$(mccσ)
161 **post**: $\mathcal{Q}_{calc_stack_sc}$(mccσ)((brs,ci,sci),mccσ′)

12.6.6.1.10 Command Center Action [O-P-Q]: `calc_sc_lt`

162. The command center non-deterministically internally calculates a pair:
163. a triplet of the identities of a stack crane, a container and a land truck, and a new state that reflects this action.
164. We leave undefined the relevant pre- and post-conditions.

value
162 calc_sc_lt: MCCΣ → (BRS×CI×SCI)×MCCΣ
163 calc_sc_lt(mccσ) **as** ((sci,ci,lti),mccσ′)
164 **pre**: $\mathcal{P}_{calc_sc_lt}$(mccσ)
164 **post**: $\mathcal{Q}_{calc_sc_lt}$(mccσ)((sci,ci,lti),mccσ′)

12.6.6.1.11 Command Center Action [Q-P-O]: `calc_lt_sc`

165. The command center non-deterministically internally calculates a pair:
166. a triplet of the identities of a land truck, a container and a stack crane, and a new state that reflects this action.
167. We leave undefined the relevant pre- and post-conditions.

value
165 calc_lt_sc: MCCΣ → (BRS×CI×SCI)×MCCΣ
166 calc_lt_sc(mccσ) **as** ((lti,ci,sci),mccσ')
167 **pre**: $\mathcal{P}_{calc_lt_sc}$(mccσ)
167 **post**: $\mathcal{Q}_{calc_lt_sc}$(mccσ)((lti,ci,sci),mccσ')

12.6.6.1.12 Command Center: Further Observations

Please observe the following: any terminal command center repeatedly and non-deterministically alternates between any and all of these actions. Observe further that: The intention of the pre- and post-conditions [Items 143, 146, 149, 152, 155, 158, 161, 167, and 164], express requirements to the command center states, mccσ:mccΣ, w.r.t. the information it must handle. Quite a complex state.

12.6.6.2 Container Storage Area Actions

We define two operations on virtual CSAs:

168. one of stacking (loading) a container, referred to by its unique identifier in a virtual CSA,
169. and one of unstacking (unloading) a container;
170. both operations involving bay/row/stack references.

type
170 BRS = BI × RI × SI
value
168 load_CI: vir_CSA × BRS × CI → vir_CSA
168 load_CI(vir_csa,(bi,ri,si),ci) **as** vir_csa′
168 **pre**: $\mathcal{P}_{load}$(vir_csa,(bi,ri,si),ci)
168 **post**: $\mathcal{Q}_{load}$(vir_csa,(bi,ri,si),ci)(vir_csa′)
169 unload_CI: vir_CSA × BRS $\stackrel{\sim}{\to}$ CI × vir_CSA
169 unload_CI(vir_csa,(bi,ri,si)) **as** (ci,vir_csa′)
169 **pre**: $\mathcal{P}_{unload}$(vir_csa,(bi,ri,si))
169 **post**: $\mathcal{Q}_{unload}$(vir_csa,(bi,ri,si))(ci,vir_csa′)

12.6.6.2.1 The Load Pre-/Post-Condtions

171. The virtual vir_CSA, i.e., vir_csa, must be well-formed;
172. the ci must not be embodied in that vir_csa; and
173. the bay/row/stack reference, (bi,ri,si) must be one of the [virtual] container stowage area.

value
168 $\mathcal{P}_{load}$(vir_csa,(bi,ri,si),ci) ≡
171 well_formed(vir_csa) cf. 25– 27 on page 317

```
172   ∧ ci ∉ xtr_CIs(vir_csa)                          cf. 49 on page 319
174   ∧ valid_BRS(bi,ri,si)(vir_csa)

174  valid_BRS: BRS → iCSA → Bool
174  valid_BRS(bi,ri,si)(vir_csa) ≡
174     bi∈ domvir_csa∧ri∈ domvir_csa(bi)∧si∈ dom(vir_csa(bi))(ri)
```

174. The resulting vir_CSA, i.e., vir_csa′, must have the same bay, row and stack identifications, and
175. except for the designated bay, row and stack, must be unchanged.
176. The designated "before", i.e., the stack before loading, must equal the tail of the "after", i.e., the loaded stack, and
177. the top of the "after" stack must equal the "input" argument container identifier.,

```
value
169  Q_load(vir_csa,(bi,ri,si),ci)(vir_csa′) ≡
174    dom vir_csa = dom vir_csa′
174   ∧ ∀ bi′:BI•bi′∈ dom vir_csa(bi′)
174        ⇒ dom vir_csa(bi′)=dom vir_csa′(bi′)
174         ∧ ∀ ri′:RI•bi′∈ dom (vir_csa(bi′))()
174              ⇒ dom (vir_csa(bi′))(ri′)=(dom vir_csa′(bi′))(ri′)
174               ∧ ∀ si′:BI•bi′∈ dom vir_csa(bi′)
174                    ⇒ dom ((vir_csa(bi′))(ri′))(si′)=dom((vir_csa′(bi′))(ri′))(si′)
175                     ∧ ∀ bi′:BI•bi′ ∈ dom vir_csa \ {bi}
175                          ⇒ vir_csa \ {bi}=vir_csa′ \ {bi}
175                           ∧ ∀ ri′:RI•ri′ ∈ dom vir_csa(bi) \ {ri}
175                                ⇒ (vir_csa(bi))(ri′)=(vir_csa′(bi))(ri′)
175                                 ∧ ∀ si′:SI•si′ ∈ dom (vir_csa)(ri′) \ {si}
175                                      ⇒ ((vir_csa)(bi′))(si′)=((vir_csa′)(bi′))(si′)
176                                      ∧ tl((vir_csa′)(bi′))(si′)=((vir_csa′)(bi′))(si′)
177                                      ∧ hd((vir_csa′)(bi′))(si′)=ci
```

12.6.6.2.2 The Unload Pre-/Post-Conditions

178. The virtual vir_csa, i.e., vir_csa,
179. must be wellformed; and
180. the bay/row/stack reference, (bi,ri,si) must be one of the [virtual] container stowage area.

```
value
178  P_unload(vir_csa,(bi,ri,si)) ≡
179    well_formed(vir_csa)
180  ∧ valid_BRS(bi,ri,si)(vir_csa)
```

181.
182.
183.
184.
185.

```
value
169  Q_unload(vir_csa,(bi,ri,si))(ci,vir_csa′) ≡
```

```
181   dom vir_csa = dom vir_csa′
182  ∧ ∀ bi′:BI • bi′ ∈ dom vir_csa \ {bi}
182        ⇒ vir_csa \ {bi} = vir_csa′ \ {bi}
183          ∧ ∀ ri′:RI • ri′ ∈ dom vir_csa(bi) \ {ri}
183               ⇒ (vir_csa(bi))(ri′) = (vir_csa′(bi))(ri′)
184                 ∧ ∀ si′:SI • si′ ∈ dom (vir_csa)(ri′) \ {si}
184                      ⇒ ((vir_csa)(bi′))(si′) = ((vir_csa′)(bi′))(si′)
185                        ∧ ((vir_csa′)(bi′))(si′)=tl((vir_csa′)(bi′))(si′)
185                        ∧ hd((vir_csa)(bi′))(si′)=ci
```

12.6.6.3 Vessel Actions

Vessels (and land trucks) are in a sense, the primary movers in understanding the terminal container domain. Containers are, of course, at the very heart of this domain. But without container vessels (and land trucks) arriving at ports *nothing would happen!* So the actions of vessels are those of actively announcing their arrivals at and departures from ports, and participating, more passively, in the unloading and loading of containers.

12.6.6.3.1 Action [A]: `calc_next_port`:

186. Vessels can calculate, `calc_next_port`, the unique identifier, `mcc_i`, of that ports' monitoring & control center.
187. We do not further define the pre- and post-conditions of the `calc_next_port` action.

```
value
186  calc_next_port: VI×VS_Mereo×VS_Stat → vir_CSA×VSΣ → MCCI×VSΣ
186  calc_next_port(vs_i,vs_mereo,vs_stat)(vir_csa,vσ) ia (mcc_i,vsσ′)
187    pre: P_calc−next−port(vsσ,vs_mereo,vs_stat)
187    post: Q_calc−next−port(vsσ,vs_mereo,vs_stat)(mcc_i,vsσ′)
```

12.6.6.3.2 Vessel Action [B]: `calc_ves_msg`:

188. Vessels can calculate, `calc_ves_info`, the vessel information, vs_info:VS_Info, to be handed to the next ports' command center.
189. This information is combined with the vessel identifier and its virtual CSA,
190. We leave undefined the pre- and post-conditions over vessel states and vessel information.

```
type
188  VS_Info
189  VS_MCC_MSG :: VI×vir_CSA×VS_Info
value
188  calc_ves_msg: VI×VMereo×VStat → VS_Pos×vir_CSA×VSΣ → VS_MCC_MSG×VSΣ
188  calc_ves_msg(vs_i,vs_mereo,vs_stat)(vpos,vir_csa,vsσ) as (vs_mcc_msg,vsσ′)
190    pre: P_calc_ves_mcc_msg(vs_i,vs_mereo,vs_stat)(vpos,vir_csa,vsσ)
190    post: Q_calc_ves_mcc_msg(vs_i,vs_mereo,vs_stat)(vpos,vir_csa,vσ)(vs_mcc_msg,vsσ′)
```

12.6.6.4 Land Truck Actions

Land trucks can initiate the following actions vis-a-vis a targeted terminal port command center: announce, to a terminal command center, its arrival with a container; announce, to a terminal command center, its readiness to haul a container. Land trucks furthermore interacts with stack cranes – as so directed by terminal command centers.

12.6.6.4.1 Land Truck Action [R]: `calc_truck_delivery`:

191. Land trucks, upon approaching, from an outside, terminal ports, calculate
192. the identifier of the next port's command center and a next land truck state.
    We do not define the
193. pre- and
194. post conditions of this calculation.

value
191 calc_truck_delivery: CI × TRUCKΣ → MCCI × LTΣ
192 calc_truck_delivery(ci,ltσ) **as** (mcci,ltσ′)
193 **pre**: $\mathcal{P}_{calc_truck_deliv}$(ci,ltσ)
194 **post**: $Q_{calc_truck_deliv}$(ci,ltσ)(mcci,ltσ′)

12.6.6.4.2 Land Truck Action [S]: `calc_truck_avail`:

195. Land trucks, when free, i.e., available for a next haul, calculate
196. the identifier of a suitable port's command center and a next land truck state.
    We do not define the
197. pre- and
198. post conditions of this calculation.

value
195 calc_truck_avail: LTI × LTΣ → MCCI × LTΣ
196 calc_truck_avail(lti,ltσ) **as** (mcci,ltσ′)
197 **pre**: $\mathcal{P}_{calc_truck_avail}$(lti,ltσ)
198 **post**: $Q_{calc_truck_avail}$(lti,ltσ)(mcci,ltσ′)

12.6.7 Events

We refer to [53, Sect. 7.1.6 and 7.3.2]. Events occur to all entities. For reasons purely of presentation we separate events into active part initiation events and active part completion events. Active part initiation events are those events that signal the initiation of actions. (Let **[Θ]** designate an action, then **[Θ']** designates the completion of that action.) Active part completion events are those events that signal the completion of actions. We do not show the lower case [**d, f, g, h, i, j, k, l, m, n, o**] in Fig. 3.

12.6.7.1 Active Part Initiation Events

Vessels:

199. [α_{vessel}] approaching terminal port;
200. [A] informing the command center, mcc, of a terminal port, of arrival;
201. [B] receiving from an mcc directions as to quay berth positions;
202. [C] receiving from an mcc, for each container to be unloaded or loaded, directions as to these unloads and ladings – and these actual unloads/ladings;
203. [X] receiving from an mcc directions of completion of stowage (no more unloads/loads);
204. [Y] informing the mcc of its departure from terminal port; or
205. [ω_{vessel}] leaving a terminal port.

Land Trucks:

206. [α_{land_truck}] approaching a terminal port;
207. [W] informing its mcc of its arrival;
208. [V] being directed, by an mcc, as to the stack (crane) of destination;
209. [S] the unloading, to a stack crane, of a container;
210. [T] the loading of a container from a stack crane;
211. [R] informing its mcc of its departure; or
212. [ω_{land_truck}] leaving a terminal port.

Containers: the transfers from

213. [D] vessel to quay crane;
214. [d] quay crane to vessel;
215. [G] quay crane to quay truck;
216. [g] quay truck to quay crane;
217. [J] quay truck to stack crane;
218. [j] stack crane to quay truck;
219. [M] stack crane to stack;
220. [m] stack to stack crane;
221. [P] stack crane to land truck; or from
222. [p] land truck to stack crane.

Quay Cranes: being informed, by the command center, mcc, of a container to be

223. [E] picked-up from a vessel;
224. [e] set-down on a vessel;
225. [F] set-down on a quay truck; or
226. [f] picked-up from a quay truck.

Quay Trucks: being informed, by the command center, mcc, of a container to be

227. [H] loaded from a quay crane;
228. [h] picked-up by a quay crane;
229. [I] picked-up by a stack crane; or
230. [i] loaded from a stack crane.

[Terminal] Stack Cranes: being informed, by the command center, mcc, of a container to be

231. [K] picked-up from a quay truck;
232. [k] loaded on to a quay truck;
233. [L] picked-up from a stack;
234. [l] loaded on to a stack;
235. [O] picked-up from a land truck; or
236. [o] loaded on to a land truck.

[Terminal Bay] Stacks: being informed, by the command center, mcc, of a container to be

237. [N] set-down, of a container, from a stack crane; or
238. [n] picked-up, of a container, by a stack crane.

These events, in most cases, prompt interaction with the terminal command center.

12.6.7.2 Active Part Completion Events:

We do not show, in Fig. 3, the **c', e', h', o', q', t'** events.

239. **[C']**
240. **[E']**
241. **[H']**
242. **[O']**
243. **[Q']**
244. **[T']**

12.6.8 Channels

We refer to [53, Sect. 7.2], and we refer to Sect. **12.5.2** and to Fig. 2 on Page 334.

12.6.8.1 Channel Declarations

There are channels between terminal port monitoring & control command center (mcci) and that command centers and that terminal port's

245. all the **containers** (ci), that might visit the terminal port; ch_mcc_con[mcci,ci][120];
246. **vessels** (vi) that might visit that port, ch_mcc[mcci,vi][121];
247. **quay cranes** (qci) of that port, ch_mcc[mcci,qci][122];
248. **quay trucks** (qti) of that port , ch_mcc[mcci,qti][123];
249. **stack cranes** (sci) of that port, ch_mcc[mcci,sci][124];
250. **stacks [bays]** (stki) of that port, ch_mcc[mcci,stki][125]; and
251. **land trucks** (lti) of, in principle, any port, ch_mcc[mcci,lti][126].
252. We shall define the concrete types of messages communicated by these channels subsequently (Sect. **12.6.8.2**).

channel
245 {ch_mcc_con[mcci,ci]|mcci:MCCI,ci:CI•mcci∈*mcc_uis*∧ci∈*c_uis*}:MCC_Con_Cmd
246-251 {ch_mcc[mcci,ui]|mcci:MCCI,ui:(VI|QCI|QTI|SCI|STKI|LTI) • mcci∈*mcc_uis*∧ui∈*uis*}:MCC_Msg
type
252 MCC_Con_Msg, MCC_Msg

12.6.8.2 Channel Messages

We present a careful analysis description, for the channels declared above, of the rather rich variety of messages communicated over channels. All messages "goes to" (a few) or "comes from" (the rest) the command center. Messages from quay cranes, quay trucks, stack cranes, and land trucks

[120] cf. Item 98 on page 327
[121] cf. Item 76 on page 322
[122] cf. Item 81 on page 323
[123] cf. Item 83 on page 323
[124] cf. Item 87 on page 324
[125] cf. Item 94 on page 325
[126] cf. Item 96 on page 326

– directed at the command center – are all in response to the *events* of their being loaded or unloaded.

12.6.8.2.1 A,B,X,Y,C′: Vessel Messages

253. There are a number **command center – vessel** and vice-versa messages:
    a. **A**: Vessels announce their (forthcoming) arrival to the next destination terminal by sending such information, VSArrv, to its monitoring & control (also referred to as command) center, that enables it to handle those vessels' berthing, unloading and loading (of container stowage).[127]
    b. **B**: The terminal command center informs such arriving vessels of their quay segment positions, VSQPos.
    c. **X**: The terminal command center informs vessels of completion of stowage handling, VS-Comp.
    d. **Y**: Vessels inform the terminal of their departure, VesDept.

type
253 MCC_Cmd == VSArrv|VSQPos|VSComp|VSDept|...
253a **A**: VSArrv :: VI × vir_CSA
253b **B**: VSQPos :: VI × (QSId × QP^+)
253c **X**: VSComp :: MCCI × VI
253d **Y**: VSDept :: MCCI × VI

12.6.8.2.2 C,D,E,E′: Vessel/Container/Quay Crane Messages

254. The terminal command center, at a time it so decides, "triggers" the simultaneous transitions, **C,D,E**, of
    a. **C**: unloading (loading) from (to) a vessel stack position of a container (surrogate), VSQC_Xfer, QCVS_Xfer),
    b. **D**: notifying the physical, i.e., the actual container that it is being unloaded (loaded), C_VStoQC (C_QCtoVS), and
    c. **E**: loading (unloading) the container (surrogate) onto (from) a quay crane, VStoQC (QCtoVS).
255. **C′,E′**: The vessel and the quay crane, in response to their being unloaded, respectively loaded with a container "moves" that load, from its top vessel bay/row/stack position to the quay crane and notifies the terminal command center of the completion of that move, VSQC_Compl.

type
253 MCC_Cmd == ... | VSQC_Xfer | QCVS_Xfer | C_VtoQC | C_QCtoV | VQC_Compl
254a VSQC_Xfer, QCVS_Xfer :: VI × (BRS × CI) × QCI
254b C_VStoQC, C_QCtoVS :: VI × CI × QCI
254c VStoQC, QCtoVS :: VI × CI × QCI
255 VSQC_Compl == VS_UnLoad | VS_Load
255 VS_UnLoad, VS_Load :: VI × CI × QCI

[127] What exactly that information is, i.e., any more concrete type model of Ves_Info cannot be given at this early stage in our development of *what a terminal is.*

12.6.8.2.3 F,G,H,H′: Quay Crane/Container/Quay Truck Messages

256. The terminal command center, at a time it so decides "triggers" the simultaneous transitions, **F,G,H**: QCtoQT, of
    a. **F**: the removal of the container from the quay crane,
    b. **G**: the notification of the physical container that it is now being transferred to a quay truck, and
    c. **H**: the loading of that container to a quay truck.
    d. **H′**: The quay truck, in response to it being loaded notifies the terminal command center of the completion of that move.

```
type
253   MCC_Cmd == ... | QCtoQT | ...
256   QCtoQT == UnloadCQC | NowConQT | LoadCQT | QCtoQTCompl
256a  UnloadCQC :: CI × QCI
256b  NowConQT :: CI × QTI
256c  LoadCQT :: CI × QTI
256d  QCtoQTCompl :: ...
```

12.6.8.2.4 I,J,K,K′: Quay Truck/Container/Stack Crane Messages

257. The terminal command center, at a time it so decides "triggers" the simultaneous transitions, **I,J,K**: QTtoSC, of
    a. **I**: the removal of a container from a quay truck,
    b. **J**: the notification of the physical container that it is now being transferred to a stack crane, and
    c. **K**: the loading of that container to a stack crane.
258. **K′**: The stack crane, in response to it being loaded notifies the terminal command center of the completion of that move.

```
type
257   MCC_Cmd = ... | QTtoSC | ...
257   QTtoSC == UnLoadCQT | NowConSC | | QCQTCompl
257a  UnLoadCQT :: CI × QRI
257b  NowConSC :: CI × SCI
257c  LoadCSC :: CI × SCI
258   QCSCCompl :: ...
```

12.6.8.2.5 L,M,N,N′: Stack Crane/Container/Stack Messages

259. The terminal command center, at a time it so decides "triggers" the simultaneous transitions, **L,M,N**: SCtoStack, of
    a. **L**: the unloading of the container from a stack crane;
    b. **M**: the notification of the physical container that it is now being transferred to a stack, and
    c. **N**: the loading of that container to a stack.
260. **N′**: The stack, in response to it being loaded, notifies the terminal command center of the completion of that move.

type
259 MCC_Cmd = ... | SCtoStack | ...
259 SCtoStack == UnLoadCSC | NowConSTK | LoadConSTK | SCStkCompl
259a UnLoadCSC :: CI × SCI
259b NowConSTK :: CI × BRS
259c LoadConSTK :: CI × BRS
260 SCStkCompl :: ...

12.6.8.2.6 O,P,Q,Q′: Land Truck/Container/Stack Crane Messages

261. The terminal command center, at a time it so decides "triggers" the simultaneous transitions, **O,P,Q**: LTtoSC, of
    a. **Q**: the unloading of the container from a land truck to a stack crane;
    b. **P**: the notification of the physical container that it is now being transferred to a stack crane, and
    c. **O**: the loading of that container to a stack crane.
    d. **O′**: The stack crane, in response to it being loaded, notifies the terminal command center of the completion of that move.[128]

type
261 MCC_Cmd = ... | LTtoSC | ...
261 LTtoSC == UnLoadCLT | NowConSC | LoadConSC | LTtoSCCompl
261a UnLoadCLT :: CI × LTI
261b NowConSC :: CI × SCI
261c LoadConSC :: CI × SCI
261d LTtoSCCompl :: ...

12.6.8.2.7 R,S,T,U,Q,V: Land Truck Messages

262. These are the messages that are communicated either from land trucks to command centers or vice versa:
    a. **R**: Land trucks, when approaching a terminal port, informs that port of its offer to deliver an identified container to stowage.
    b. **S**: Land trucks, when approaching a terminal port, informs that port of its offer to accept (load) an identified container from stowage.
    c. **T**: Land trucks, at a terminal, are informed by the terminal of the stack crane at which to deliver (unload) an identified container.
    d. **U**: Land trucks, at a terminal, are informed by the terminal of the stack crane from which to accept an identified container.
    e. **Q**: Land trucks, at a termial, are informed by the terminal of the stack crane at which to unload (deliver) an identified container.
    f. **q**: Land trucks, at a terminal, are informed by the terminal of the stack crane at which to load (accept) an identified container.
    g. **V**: Land trucks, at a terminal, inform the terminal of their departure.

type

[128] The **O′** event is "the same" as the **K′** event.

```
262    MCC_Cmd = ... | LTCmd | ...
262    LTCmd == LTDlvr | LTFtch | LTtoSC | LTfrSC | LTDept
262a   LTDlvr :: LTI × CI
262b   LTFtch :: LTI × CI
262c   LTtoSC :: LTI × CI
262d   LTfrSC :: LTI × CI
262g   LTDept :: LTI
```

12.6.9 Behaviours

We refer to [53, Sects. 7.1.7, 7.3.3-4-5, and 7.4].

To every part of the domain we associate a behaviour. Parts are in space: there are the manifest parts, and there are the notion of their corresponding behaviours. Behaviours are in space and time. We model behaviours as processes defined in RSL^+. We cannot see these processes. We can, however, define their effects.

Parts may move in space: vessels, cranes, trucks and containers certainly do move in space; processes have no notion of spatial location. So we must "fake" the movements of movable parts. We do so as follows: We associate with containers the programmable attribute of location, as outlined in Items 126– 126g on page 331. We omit, for this model, the more explicit modelling of vessels, cranes and trucks but refer to their physical mereologies.

In the model of endurants, cf. Page 315, we modelled vessel and terminal container stowage areas as physically embodying containers, and we could move containers: push and pop them onto, respectively from bay stacks. This model must now, with containers being processes, be changed. The stacks, STACK, of container stowage areas, CAS, now embody unique container identifiers ! We rename these stacks into cistack:CiSTACK

12.6.9.1 Terminal Command Center

The terminal command center is at the core of activities of a terminal port. We refer to the figure on Page 334. "Reading" that figure left-to-right illustrates the movements of containers from **[C-D-E]** vessels to quay cranes, **[F-G-H]** quay cranes to quay trucks, **[I-J-K]** quay trucks to stack cranes, **[L-M-N]** stack cranes to stacks, and from **[O-P-Q]** land truck to stack cranes. A similar "reading" of that figure from right-to-left would illustrate the movements of containers from **[q-p-o]** stack cranes to land trucks; **[n-m-l]** stacks to stack cranes; **[k-j-i]** stack cranes to quay trucks; **[h-g-f]** quay trucks to quay cranes; and from **[e-d-c]** quay cranes to vessels. We have not show the **[c-d-e-f-g-h-i-j-k-l-m-n-o-p-q]** labels, but their points should be obvious (!).

12.6.9.1.1 The Command Center Behaviour:

We distinguish between the command center behaviour offering to *monitor* primarily vessels and land trucks, secondarily cranes, quay cranes and stacks, and offering to *control* vessels, cranes, trucks and containers.

263. The signature of the command center behaviour is a triple of the command center identifier, the conceptual command center mereology and the static command center attributes (i.e., the topological description of the terminal); the programmable command center attributes (i.e., the command center state); and the input/output channels for the command center.
The command center behaviour non-deterministically (externallY) chooses between

264. either monitoring inputs from
265. or controlling (i.e., outputs to)

vessels, cranes, trucks, stacks and containers.

```
value
263 command_center:
263   mcci:MCCI×(vis,qcis,qtis,scis,bis,ltis,cis):MCC_Mer×MCC_Stat
263   → MCCΣ →
263     in,out { ch_mcc[ mcci,ui ]n
263              | mcci:MCCI,ui:(VI|QCI|QTI|SCI|BI|LTI)
263               • ui∈vis∪qcis∪qtis∪scis∪bis∪ltis }
263     out { ch_mcc_con[ mcci,ci ] | ci:CI•ci ∈ cis }  Unit
263 command_center(mcci,(vis,qcis,scis,bis,ltis,cis),mcc_stat)(mccσ) ≡
264    monitoring(mcci,(vis,qcis,scis,bis,ltis,cis),mcc_stat)(mccσ)
263  ⌈⌉
265    control(mcci,(vis,qcis,scis,bis,ltis,cis),mcc_stat)(mccσ)
```

12.6.9.1.2 The Command Center Monitor Behaviours:

The command center monitors the behaviours of vessels, cranes and trucks: **[A,Y′,C′,E′,F′,H′,I′,K′,L′,N′,O′,Q′]**. The input message thus received is typed:

```
type
    VCT_Info = ...
```

That information is used by the command center to update its state:

```
value
    update_MCCΣ:  VCT_Infor → MCCΣ → MCCΣ
```

The definition of monitoring is simple.

266. The signature of the monitoring behaviour is the same as the command center behaviour.
267. The monitor non-deterministically externally (⌈⌉) offers to accept any input, vct_info, message from any vessel, any land truck and from local terminal port quay trucks and cranes.
268. That input, vct_info, enters the update of the command center state, from mccσ to mccσ′.
269. Whereupon the monitoring behaviour resumes being the command center behaviour with an updated state.

```
value
266 monitoring: mcci:MCCI × mis:MCC_Mereo × MCC_Stat
266    → MCCΣ
266    → in,out {chan_mcc[ mcci,i] | i ∈ mis}  Unit
266 monitoring(mcci,mis,mcc_stat)(mccσ) ≡
267    let vct_info = ⌈⌉ { chan_mcc[ mcci,i] ? | i ∈ mis } in
268    let mccσ′ = update_MCCΣ((vct_info,ui))(mccσ) in
269    command_center(mcci,mis,mcc_stat)(mccσ′) end end
```

12.6.9.1.3 The Command Center Control Behaviours:

270. The command center control behaviour has the same signature as the command center behaviour (formula Items 263).

271. In each iteration of the command center behaviour in which it chooses the control alternative it calculates[129] a next [output] transaction. This calculation is at the very core of the overall terminal port. We shall have more to say about this in Sect. **12.7.1** on page 358.
Items, 272a–272j represent 10 alternative transactions.
272. They are "selected" by the **case** clause (Item 272).
So for each of these 10 alternatives there the command center offers a communication. For the **[CDE, FGH, IJK, LMN, OPQ, opq]** cases there is the same triple of concurrently synchronised events. For the **[B,T,X]** clauses there are only a single synchronisation effort. The command center events communicates:
   a. **[B]** the quay positions to arriving vessels,
   the transfer of containers
   b. **[CDE]** from vessel stacks to quay cranes,
   c. **[FGH]** quay cranes to quay trucks,
   d. **[IJK]** quay trucks to stack cranes,
   e. **[LMN]** stack cranes to stacks,
   f. **[OPQ]** stack cranes to land trucks, and
   g. **[opq]** land trucks to stack cranes.
   We also illustrate
   h. **[T]** the bays to which a land truck is to deliver, or fetch a container, and
   i. **[X]** the "signing off" of a vessel by the command center.
   j. For the case that the next transaction cannot be determined [at any given point in time] there is nothing to act upon.
273. After any of these alternatives the command center control behaviour resumes being the command center behaviour with the state updated from the next transaction calculation.

```
value
270 control: mcci:MCCI×(vis,qcis,qtis,scis,bis,ltis,cis):MCC_Mer×MCC_Stat → MCCΣ →
263     in,out {ch_mcc[mcci,ui]|mcci:MCCI,ui:(VI|QCI|QTI|SCI|BI|LTI)•ui∈vis∪qcis∪qtis∪scis∪bis∪ltis}
263     out { ch_mcc_con[mcci,ci] | ci:CI•ci ∈ cis } Unit
270 control(mcci,(vis,qcis,scis,bis,ltis,cis),mcc_stat)(mccσ) ≡
271   let (mcc_trans,mccσ′) = calc_nxt_transaction(mcci,mcc_mereo,mcc_stat)(mccσ) in
272   case mcc_trans of
272a [B]     mkVSQPos(vi,qp) → ch_mcc[mcci,vi] ! mkVSQPos(vi,qp),
272b [CDE]  mkVSQC_Xfer(vi,(brs,ci),qci) →
272b [C]          ch_mcc[mcci,vi] ! mkVes_UnLoad(ci,brs)
272b [D]          || ch_mcc_con[mcci,ci] ! mkNewPos(mcci,qci)
272b [E]          || ch_mcc[mcci,qci] ! mkQC_Load(ci),
272c [FGH]  mkQCQT_Xfer(qci,ci,qti) →
272c [F]          ch_mcc[mcci,qci] ! mkQC_UnLoad(ci)
272c [G]          || ch_mcc_con[mcci,ci] ! mkNewPos(mcci,qti)
272c [H]          || ch_mcc[mcci,qti] ! mkQT_Load(ci),
272d [IJK]  mkQTSC_Xfer(qti,ci,sci) →
272d [I]          ch_mcc[mcci,qci] ! mkQT_UnLoad(ci)
272d [J]          || ch_mcc_con[mcci,ci] ! mkNewPos(mcci,sci)
272d [K]          || ch_mcc[mcci,qti] ! mkSC_Load(ci),
272e [LMN]  mkSCSTK_Xfer(brs,ci,sci,sti) →
272e [L]          ch_mcc[mcci,sci] ! mkSC_UnLoad(ci)
272e [M]          || ch_mcc_con[mcci,ci] ! mkNewPos(mcci,brs)
272e [N]          || ch_mcc[mcci,stki] ! mkSTK_Load(ci,brs),
272f [OPQ]  mkSCLT_Xfer(sci,ci,lti) →
272f [O]          ch_mcc[mcci,sci] ! mkSC_UnLoad(ci)
272f [P]          || ch_mcc_con[mcci,ci] ! mkNewPos(mcci,lti)
272f [Q]          || ch_mcc[mcci,lti] ! mkLT_Load(ci),
272g [opq]  mkLTSC_Xfer(sci,ci,lti) →
272g [o]          ch_mcc[mcci,sci] ! mkSC_Load(ci)
```

[129] For `calc_nxt_transaction` see Items 130 – 140 on page 336

```
272g [p]          || ch_mcc_con[ mcci,ci ] ! mkNewPos(mcci,cti)
272g [q]          || ch_mcc[ mcci,lti ] ! mkLT_UnLoad(ci),
272h [T]     mkLT_Dept(lti) → ch_mcc[ mcci,lti ] ! LTDept(mcci,lti),
272i [Y]     mkVSComp(mcci,vi) → ch_mcc[ mcci,vi ] ! VSComp(mcci,vi),
272i [X]     mkVSDept(mcci,vi) → ch_mcc[ mcci,vi ] ! VSDept(mcci,vi),
272j         _ → skip
272     end ; command_center(mcci,(vis,qcis,scis,bis,ltis,cis),mcc_stat)(mccσ′) end
```

12.6.9.2 Vessels

274. The signature of the vessel behaviour is a triple of the vessel identifier, the conceptual vessel mereology, the static vessel attributes, and the programmable vessel attributes. [We presently leave static attributes unspecified: ...]
     Nondeterministically externally, [], the vessel decides between
275. [A] either approaching a port,
276. [] or [subsequently] arriving at that port,
     or [subsequently] participating in the
277. [] unloading and
278. [] loading of containers of containers,
279. [] or [finally] departing from that port.

```
value
274  vessel: vi:VI×mccis:V_Mereo×V_Sta_Attrs → (V_Pos×vir_CSA×VΣ)
274     → in,out {ch_mcc[ mcci,vi ]|mcci:MCCI•mcci∈mccis}  Unit
274  vessel(vi,mccis,...)(vpos,vir_csa,vσ) ≡
275      port_approach(vi,mccis,...)(vpos,vir_csa,vσ)
276   [] port_arrival(vi,mccis,...)(vpos,vir_csa,vσ)
277   [] unload_container(vi,mccis,...)(vpos,vir_csa,vσ)
278   [] load_container(vi,mccis,...)(vpos,vir_csa,vσ)
279   [] port_departure(vi,mccis,...)(vpos,vir_csa,vσ)
```

12.6.9.2.1 Port Approach

280. The signature of port_approach behaviour is identical to that of vessel behaviour.
281. On approaching any port the vessel calculates the identity of that port's command center.
282. Then, with an updated state, it calculates the information to be handed over to the designated terminal –
283. [A] which is then communicated from the vessel to the command center;
284. whereupon the vessel resumes being a vessel albeit with a doubly updated state.

```
value
280  port_approach: vi:VI×vs_mer:VS_Mereo×VS_Stat→(VS_Pos×vir_CSA×VΣ)
280     → in,out {ch_mcc[ mcci,vi ]|mcci:MCCI•mcci∈mccis}  Unit
280  port_approach(vi,vs_mer,vs_stat)(vpos,vir_csa,vσ) ≡
281     let (mcci,vσ′) = calc_next_port(vi,vs_mer,vs_stat)(vpos,vir_csa,vσ) in
282     let (mkVInfo(vi,vir_csa,vs_info),vσ′′) = calc_ves_msg(vpos,vir_csa,vσ′) in
283     ch_mcc[ mcci,vi ] ! mkVS_Info(vi,vir_csa,vs_info) ;
284     vessel(vi,vs_mer,vs_stat)(vpos,vir_csa,vσ′′) end end
```

12.6.9.2.2 Port Arrival

285. The signature of port_arrival behaviour is identical to that of vessel behaviour.
286. [B] Non-deterministically externally the vessel offers to accept a terminal port quay position from any terminal port's command center.
287. The vessel state is updated accordingly.

288. Whereupon the vessel resumes being a vessel albeit with a state updated with awareness of its quay position.
289. The vessel is ready to receive such quay position from any terminal port.

```
value
285 port_arrival: vi:VI×mccis:V_Mereo×V_Sta_Attrs → (V_Pos×vir_CSA×VΣ)
285    → in,out {ch_mcc[ mcci,vi ]|mcci:MCCI•mcci∈mccis} Unit
285 port_arrival(vi,mccis,...)(vpos,vir_csa,vσ) ≡
286    { let mkVSQPos(vi,(qs,cpl)) = ch_mcc[ mcci,vi ] ? in
287      let vσ′ = upd_ves_state(mcci,(qs,cpl))(vσ) in
288      vessel(vi,mccis,...)(mkInPort(mcci,mkVSQPos(qs,cpl)),vir_csa,vσ′) end end
289     | mcci:MCCI•mcci∈mccis }
```

12.6.9.2.3 Unloading of Containers

290. The signature of port_arrival behaviour is identical to that of vessel behaviour.
291. **[C]** The vessel offers to accept, ch_mcc_v[mcci,vi] ?, a directive from the command center of the terminal port at which it is berthed, to unload, mkUnload((bi,ri,si),ci). a container, identified by ci, at some container stowage area location ((bi,ri,si)).
292. The vessel unloads the container – identified by ci′.
293. If the unloaded container identifier is different from the expected **chaos** erupts !
294. The vessel state, vσ′, is updated accordingly.
295. **[C′]** “Some time has elapsed since the unload directive, modelling” the completion, from the point of view of the vessel, of the unload operation –
296. whereupon the command center is informed of this completion (**[′]**).
297. The vessel resumes being the vessel in a state reflecting the unload.

```
value
290 unload_container: vi:VI × mccis:V_Mereo × V_Sta_Attrs → (V_Pos × iCSA × VΣ) →
290    in,out {ch_mcc[ mcci,vi ]|mcci:MCCI•mcci∈mccis} Unit
290 unload_container(vi,mccis,...)(vpos,vir_csa,vσ) ≡
291    let mkVes_UnLoad(ci,(bi,ri,si)) = ch_mcc[ mcci,vi ] ? in
292    let (ci′,vir_csa′) = unload_CI((bi,ri,si),vir_csa) in
293    if ci′ ≠ ci then chaos end;
294    let vσ′′ = unload_update_VΣ((bi,ri,si),ci)(vir_csa′) in
295    wait sometime ;
296    ch_mcc[ mcci,vi ] ! mkCompl(mkV_UnLoad((bi,ri,si),ci)) ;
297    vessel(vi,mccis,...)(vpos,vir_csa′,vσ′′) end end end
```

12.6.9.2.4 Loading of Containers

298. The signature of load_container behaviour is identical to that of vessel behaviour.
299. **[c]** The vessel offers to accept, ch_mcc_v[mcci,vi] ?, a directive from the command center of the terminal port at which it is berthed, to load, mkLoad((bi,ri,si),ci). a container, identified by ci, at some container stowage area location ((bi,ri,si)).
300. The vessel (in co-operation with a quay crane, see later) then unloads the container – identified by ci.
301. The vessel state, vσ′, is updated accordingly.
302. **[c′]** “Some time has elapsed since the unload directive, modelling” the completion, from the point of view of the vessel, of the unload operation – whereupon the command center is informed of this completion (**[′]**).
303. and the vessels resumes being the vessel in a state reflecting the load.

```
value
298 load_container: vi:VI×mccis:V_Mereo×V_Sta_Attrs → (V_Pos×vir_CSA×VΣ)
298    → in,out {ch_mcc[ mcci,vi ]|mcci:MCCI•mcci∈mccis} Unit
298 load_container(vi,mccis,...)(vpos,vir_csa,vσ) ≡
299    let mkV_Load((bi,ri,si),ci) = ch_mcc[ mcci,vi ] ? in
300    let vir_csa′ = load_CI(vir_csa,(bi,ri,si),ci) in
301    let vσ′ = load_update_VΣ((bi,ri,si),ci) in
302    ch_mcc[ mcci,vi ] ! mkCompl(mkV_Load((bi,ri,si),ci)) ;
303    vessel(vi,mccis,...)(vpos,vir_csa′,vσ′) end end end
```

12.6.9.2.5 Port Departure

304. The signature of port_departure behaviour is identical to that of vessel behaviour.
305. **[Y]** At some time some command center informs a vessel that *stowage*, i.e., the unloading and loading of containers has ended.
306. Vessels update their states accordingly.
307. **[Y']** Vessels respond by informing the command center of their departure.
308. Whereupon vessels resume being vessels.

```
value
304 port_departure: vi:VI×mccis:V_Mereo×V_Sta_Attrs → (V_Pos×vir_CSA×VΣ)
304     → in,out {ch_mcc[ mcci,vi ]|mcci:MCCI•mcci∈mccis}  Unit
304 port_departure(vi,mccis,v_sta)(vpos,vir_csa,vσ) ≡
305     let mkStow_Compl(mcci,vi) ⌈⌉⌊⌋ { ch_mcc[ mcci,vi ] ? | mcci:MCCI•mcci∈mccis } in
306     let vσ′ = update_vessel_state(mkVes_Dept(mcci,vi))(vσ) in
307     ch_mcc[ mcci,vi ] ! mkVes_Dept(mcci,vi) ;
308     vessel(vi,mccis,v_sta)(vpos,vir_csa,vσ′) end end
```

•••

The next three behaviours: quay_crane, quay_truck and stack_crane, are very similar. One substitutes, line-by-line, command center/quay crane, quay crane/quay truck, quay truck/stack crane et cetera !

12.6.9.3 Quay Cranes

309. The signature of the quay_crane behaviour is a triple of the quay crane identifier, the conceptual quay crane mereology, the static quay crane attributes, the programmable quay crane attributes – and the 'command center'/'quay crane' channel.
310. The quay crane offers, non-deterministically externally, to
311. either, **[E]**, accept a directive of a *'container transfer from vessel to quay crane'*.
    a. The quay crane then resumes being a quay crane now holding (a surrogate of) the transferred container.
312. or, **[F]** accept a directive of a transfer *'container from quay crane to quay truck'*.
    a. The quay crane then resumes being a quay crane now holding (a surrogate of) the transferred container.

```
value
309  quay_crane: qci:QCI × mcci:QC_Mer × QC_Sta → (QCHold×QCPos)
309      → ch_mcc[ mcci,qci ]  Unit
309  quay_crane(qci,mcci,qc_sta)(qchold,qcpos) ≡
311      let mkVSQC(ci) = ch_mcc[ mcci,qci ] ? in
311a       quay_crane(qci,mcci,qc_sta)(mkCon(ci),qcpos) end
310    ⌈⌉
312      let mkQCVS(ci) = ch_mcc[ mcci,qci ] ? in
312a       quay_crane(qci,mcci,qc_sta)(mkCon(ci),qcpos) end
```

12.6.9.4 Quay Trucks

313. The signature of the quay_truck behaviour is a triple of the quay truck identifier, the conceptual quay truck mereology, the static quay truck attributes, the programmable quay truck attributes – and the 'command center'/'quay truck' channel.
314. The quay truck offers, non-deterministically externally, to
315. either, **[H]**, accept a directive of a *'container transfer from quay crane to quay truck'*.
    a. The quay truck then resumes being a quay truck now holding (a surrogate of) the transferred container.
316. or, **[I]**, accept a directive of a *'container transfer from quay truck to quay crane'*.
    a. The quay truck then resumes being a quay truck now holding (a surrogate of) the transferred container.

```
value
313   quay_truck: qti:QTI × mcci:QC_Mer × QT_Sta → (QTHold×QTPos)
313     → ch_mcc[ mcci,qci ]  Unit
313   quay_truck(qti,mcci,qt_sta)(qthold,qtpos) ≡
315     let mkQCQT(ci) = ch_mcc[ mcci,qti ] ? in
315a      quay_crane(qti,mcci,qc_sta)(mkCon(ci),qcpos) end
314   []
316     let mkQTQC(ci) = ch_mcc[ mcci,qti ] ? in
316a      quay_crane(qti,mcci,qc_sta)(mkCon(ci),qcpos) end
```

12.6.9.5 Stack Crane

317. The signature of the stack_crane behaviour is a triple of the stack crane stack crane identifier, the conceptual mereology, the static stack crane attributes, the programmable stack crane attributes – and the 'command center'/'stack crane' channel.
318. The stack crane offers, non-deterministically externally, to
319. either, **[K]**, accept a directive of a *'container transfer from quay truck to stack crane'*.
    a. The stack crane then resumes being a stack crane now holding (a surrogate of) the transferred container.
320. or, **[L]**, accept a directive of a *'container transfer from stack crane to quay truck'*.
    a. The stack crane then resumes being a stack crane now holding (a surrogate of) the transferred container.

```
value
317   stack_crane: sci:SCI × mcci:SC_Mer × SC_Sta → (SCHold×SCPos)
317     → ch_mcc[ mcci,sci ]  Unit
317   stack_crane(sci,mcci,sc_sta)(schold,scpos) ≡
319     let mkQTSC(ci) = ch_mcc[ mcci,sci ] ? in
319a      stack_crane(sci,mcci,sc_sta)(mkCon(ci),scpos) end
318   []
320     let mkSCQT(ci) = ch_mcc[ mcci,sci ] ? in
320a      stack_crane(sci,mcci,sc_sta)(mkCon(ci),scpos) end
```

12.6.9.6 Stacks

The stack behaviour is very much like the unload_container container behaviour of the vessel, cf. Items 290 – 294 on page 352.

321. The signature of the stack behaviour is a triple of the stack, i.e. terminal port bay identifier, the conceptual bay mereology, the static bay attributes, the programmable bay attributes and the 'command center'/'stack' channel.
322. The stack offers, **[N]**, to accept directive of a *'container transfer from stack crane to stack'*.
    a. The stack behaviour loads the container, identified by ci′, to the bay/row/stack top, identified by (bi,ri,si).
    b. If the unloaded container identifier is different from the expected **chaos** erupts !
    c. The stack state, bay′, is updated accordingly.
    d. **[N′]** "Some time has elapsed since the load directive, modelling" the completion, from the point of view of the vessel, of the unload operation –
    e. whereupon the command center is informed of this completion (**[′]**).
    f. The stack then resumes being a stack now holding (a surrogate of) the transferred container.

```
value
321   stack: tbi:TBI×mcci:STK_Mer×Stk_Sta_Attrs → (iCSA × Stk_Dir) →
321     in,out {ch_mcc[ mcci,vi ]|mcci:MCCI•mcci∈mccis}  Unit
321   stack(tbi,mcci,stk_sta)(bay,dir) ≡
322     let mkUnload((bi,ri,si),ci) = ch_mcc[ mcci,tbi ] ? in
322a    let (ci′,bay′) = unload_CI((bi,ri,si),bay) in
322b    if ci′ ≠ ci then chaos end ;
322c    let bay′′ = unload_update_BAY((bi,ri,si),ci)(bay′) in
322d    wait sometime ;
322e    ch_mcc[ mcci,tbi ] ! mkCompl(mkUnload((bi,ri,si),ci)) ;
322f   stack(tbi,mcci,stk_sta)(bay′′,dir) end end end
```

12.6.9.7 Land Trucks

323. The signature of the land_truck behaviour is a triple of the land truck identifier, the conceptual land truck mereology and the static land truck attributes, and the programmable land truck attributes.
324. **R**
   a. The land truck calculates the identifier of the next port's command center
   b. and communicates with this center as to its intent to deliver a container identified by ci,
   c. whereupon the land truck resumes being that.
325. **T**
   a. The command center informs the land truck of the bay ('stack'), brs, at which to deliver the container,
   b. whereupon the land truck resumes being that.
326. **Q**
   a. The command center informs the land truck of the delivery of a container from a stack crane,
   b. ...,
   c. whereupon the land truck resumes being that.
327. **V**
   a. The land truck informs the command center of its intent to depart from the terminal port,
   b. whereupon the land truck resumes by leaving the terminal port.

```
value
323  land_truck:
323
323  land_truck(lti,lt_mer,lt_sta)(lt_pos,lt_hold) ≡
324       next_port(lti,lt_mer,lt_sta)(lt_pos,lt_hold)
325    [] stack_location(lti,lt_mer,lt_sta)(lt_pos,lt_hold)
326    [] stack_crane_to_land_truck(lti,lt_mer,lt_sta)(lt_pos,lt_hold)
327    [] land_truck_departure(lti,lt_mer,lt_sta)(lt_pos,lt_hold)
```

```
value
324  next_port(lti,lt_mer,lt_sta)(...,mkHold(ci,cσ)) ≡
324a     let mcci = calc_truck_delivery(ci,cσ) in
324b     ch_mcc[ mcci,lti ] ! mkDlvr(ci,cσ) ;
324c     land_truck(lti,lt_mer,lt_sta)(...,...) end ???
```

```
value
325  stack_location(lti,lt_mer,lt_sta)(...,mkHold(ci,cσ)) ≡
325a     let mkLT_Pos(mcci,brs) = { ch_mcc[ mcci,lti ] ? | mcci:MCCI • mcci ∈ mcc_uis }
325b     land_truck(lti,lt_mer,lt_sta)(...,lt_hold) end ???
```

```
value
326  stack_crane_to_land_truck(lti,lt_mer,lt_sta)(lt_pos,lt_hold) ≡
326a
326b
```

```
value
327  land_truck_departure(lti,lt_mer,lt_sta)(...,...) ???
327a     ch_mcc[ mcci,lti ] ! mkDept(lti) ;
327b     land_truck(lti,lt_mer,lt_sta)(...,...)  ???
```

12.6.9.8 Containers

In RSL, as with all formal specification languages one cannot "move" values. So we model containers of vessels and of terminal port stacks as separate behaviours and replace their "values", C in vessel and terminal port stacks by their unique identifications, CI.

328. The signature of the container behaviour is simple: the container identifier, its mereology, its static values, its position and state[130], and its input channels.
329. **[D,G,J,M,P]** The container is here simplified to just, at any moment, accepting a new position from any terminal ports command center;
330. whereupon the container resumes being that with that new position.

```
value
328  container: ci:CI×mcci_uis:C_Mer×C_Stat → (CPos×CΣ)
328     → in { ch_mcc_con[ mcci,ci ]
328           | mcci:MCCI • mcci∈mcci_uis }  Unit
328  container(ci,mcci_uis,...)(pos,sσ) ≡
329     let mkNewPos(p) = { ch_mcc_con[ mcci,ci ] ?
329                         | mcci:MCCI•mcci∈mcci_uis } in
330     container(ci,mcci_uis,...)(mkNewPos(p),sσ) end
```

12.6.10 Initial System

12.6.10.1 The Distributed System

We remind ourselves that the container line industry includes a set of vessels, a set of land trucks, a set of containers and a set of terminal ports. We rely on the states expounded in Sect. **12.5.4.1**'s Items 50 on page 320 – 54 on page 320.

331. The signature of τ_initial_system is that of a function from an endurant container line industry to its perdurant behaviour, i.e., **Unit**.

This behaviour is expressed as

332. the distributed composition of all vessel behaviours in parallel with
333. the distributed composition of all land truck behaviours in parallel with
334. the distributed composition of all container behaviours in parallel with
335. the distributed composition of all terminal port behaviours.

```
value
332  τ_initial_system: CLI → Unit
332  τ_initial_system(cli) ≡
332     ‖{ τ_vessel(v) | v:V • v ∈ vs }
333  ‖ ‖{ τ_land_truck(lt) | lt:LT • lt ∈ lts }
334  ‖ ‖{ τ_container(c) | c:CON • c ∈ cs }
335  ‖ ‖{ τ_terminal_port(tp) | tp:TP • tp ∈ tps }
```

12.6.10.2 Initial Vessels

336. The signature of the i_vessel `transalation function` is simple: a τ`ranslator` from endurant vessel `parts` v to perdurant vessel `behaviours`, i.e., **Unit**.
337. The transcendental deduction then consists of obtaining the proper arguments for the vessel behaviour –
338. and invoking that behaviour.

[130] As for state: I need to update the container attribute section, Sect. **12.5.6.11** on page 331 to reflect a state (for example: the component contents of a container)

```
value
336  τ_vessel: V → Unit
336  τ_vessel(v) ≡
337    let v_ui = uid_V(v), v_mer = mereo_V(v),
337        v_sta = attr_V_Sta(v), v_pos = attr_V_Pos(v),
337        v_csa = attr_iCSA(v), vσ = attr_VΣ(v) in
338    vessel(v_ui,v_mer,v_sta)(v_pos,v_csa,vσ) end
```

12.6.10.3 Initial Land Trucks

Similarly:

```
τ_land_truck: LT → Unit
τ_land_truck(lt) ≡
        let lt_ui = uid_LT(lt), lt_mer = mereo_LT(lt),
          lt_sta = attr_LT_Sta(lt), lt_pos = attr_LT_Pos(lt),
          lt_hold = attr_LT_Hold(v), ltσ = attr_LTΣ(lt) in
        vessel(lt_ui,lt_mer,lt_sta)(lt_pos,lt_hold,ltσ) end
```

12.6.10.4 Initial Containers

Similarly:

```
τ_container: CON → Unit
τ_container(con) ≡
        let c_ui = uid_CON(con), c_mer = mereo_CON(con),
          c_sta = attr_C_Sta(con), c_pos = attr_C_Pos(con),
          cσ = attr_CONΣ(lt) in
        container(c_ui,c_mer,c_sta)(c_pos,cσ) end
```

12.6.10.5 Initial Terminal Ports

Terminal ports consists of a set of quay cranes, a set of quay trucks a set of stack cranes, and a set of stacks. They translate accordingly:

```
τ_terminal_port: TP → Unit
τ_terminal_port(tp) ≡
        let qcs = obs_QCs(obs_QCS(tp)),
              qts = obs_QTs(obs_QTS(tp)),
              scs = obs_SCs(obs_SCS(tp)),
              stks = obs_STKs(obs_STKS(tp)) in
    || { τ_quay_crane(qc) | qc:QC • qc ∈ qcs } ||
    || { τ_quay_truck(qt) | qt:QT • qt ∈ qts } ||
    || { τ_stack_crane(sc) | sc:SC • sc ∈ scs } ||
    || { τ_stack(stk) | stk:STK • stk ∈ stks } end
```

12.6.10.6 Initial Quay Cranes

```
τ_quay_crane: QC → Unit
τ_(qc) ≡
        let qc_ui = uid_QC(qc), qc_mer = mereo_QC(qc),
          qc_sta = attr_QC_Sta(qc), qc_pos = attr_QC_Pos(qc),
```

```
    qcσ = attr_QCΣ(qc) in
  quay_crane(qc_ui,qc_mer,qc_sta)(qc_pos,qcσ) end
```

12.6.10.7 Initial Quay Trucks

```
τ_quay_truck: QT → Unit
τ_quay_truck(qt) ≡
  let qt_ui = uid_QT(qt), qt_mer = mereo_QT(qt),
    qt_sta = attr_QT_Sta(qt), qt_pos = attr_QT_Pos(qt),
    qtσ = attr_QTΣ(qt) in
  quay_truck(qt_ui,qt_mer,qt_sta)(qt_pos,qtσ) end
```

12.6.10.8 Initial Stack Cranes

```
τ_stack_crane: SC → Unit
τ_stack_crane(sc) ≡
  let sc_ui = uid_SC(sc), sc_mer = mereo_SC(sc),
    sc_sta = attr_SC_Sta(sc), sc_pos = attr_SC_Pos(sc),
    scσ = attr_SCΣ(sc) in
  container(sc_ui,sc_mer,sc_sta)(sc_pos,scσ) end
```

12.6.10.9 Initial Stacks

```
τ_stack: STK → Unit
τ_stack(stk) ≡
  let stk_ui = uid_STK(stk), stk_mer = mereo_STK(stk),
    stk_sta = attr_STK_Sta(stk),
    stkσ = attr_STKΣ(stk) in
  stack(stk_ui,stk_mer,stk_sta)(stkσ) end
```

12.7 Conclusion

TO BE WRITTEN

12.7.1 An Interpreation of the Behavioural Description

TO BE WRITTEN

12.7.2 What Has Been Done

TO BE WRITTEN

12.7.3 What To Do Next

TO BE WRITTEN

12.7.4 Acknowledgements

This report was begun when I was first invited to lecture, for three weeks in November 2018, at ECNU[131], Shanghai, China. For this and for my actual stay at ECNU, I gratefully acknowledge Profs. He JiFeng, Zhu HuiBiao, Wang XiaoLing and Min Zhang. I chose at the time of the invitation to lead the course students through a major, non-trivial example. Since Shanghai is also one of the major container shipping ports of the world, and since the Danish company Maersk, through its subsidiary, APMTerminals, operates a major container terminal port, I decided on the subject fo this experimental report. I gratefully acknowledge the support the ECNU course received from APMTerminals, through its staff, Messrs Henry Bai and Niels Roed.

[131] ECNU: East China Normal University

Chapter 13

Simple Retailer System [January 2021]

Contents

13.1 Two Approaches to Modeling 364
13.1.1 Domain Science & Engineering: DS&E 364
13.1.2 HERAKLIT: http://heraklit.dfki.de/ 364
13.2 The retailer market Case Study 364
13.2.1 Three Rough Sketches 364
13.2.1.1 Identification of "Main Players" 365
13.2.1.2 Main Transaction Sequences 365
13.2.1.3 Detailed Sketch 366
13.2.1.4 Transitions 367
13.3 Endurants: External Qualities 368
13.3.1 Main Decompositions 368
13.3.1.0.1 Narrative 368
13.3.1.0.2 Formalisation 368
13.3.2 Aggregates as Sets 369
13.3.2.0.1 Narrative 369
13.3.2.0.2 Formalisation 369
13.3.3 The Retailer 370
13.3.3.1 The HERAKLIT View 370
13.3.3.1.1 Narrative 370
13.3.3.1.2 Formalisation 370
13.3.3.2 The DS&E View 371
13.3.4 The Market System State 371
13.3.4.0.1 Narrative 371
13.3.4.0.2 Formalisation 371
13.4 Endurants: Internal Qualities 372
13.4.1 Unique Identifiers 372
13.4.1.0.1 Narrative 372
13.4.1.0.2 Formalisation 372
13.4.2 Mereology 373
13.4.2.1 Customer Mereology 373
13.4.2.1.1 Narrative 373
13.4.2.1.2 Formalisation 373
13.4.2.2 Order Management Mereology 374
13.4.2.2.1 Narrative 374
13.4.2.2.2 Formalisation 374
13.4.2.3 Inventory Mereology 374
13.4.2.3.1 Narrative 374
13.4.2.3.2 Formalisation 374
13.4.2.4 Warehouse Mereology 375
13.4.2.4.1 Formalisation 375
13.4.2.5 Supplier Mereology 375
13.4.2.5.1 Narrative 375
13.4.2.5.2 Formalisation 375
13.4.2.6 Courier Service Mereology 375

13.4.2.6.1 Narrative 375
13.4.2.6.2 Formalisation 376
13.4.3 Attributes 376
13.4.3.1 Transactions 376
13.4.3.1.1 Narrative 376
13.4.3.1.2 Formalisation 377
13.4.3.1.3 Narrative 377
13.4.3.1.4 Formalisation 377
13.4.3.2 Customer Attributes 377
13.4.3.2.1 Narrative 378
13.4.3.2.2 Formalisation 378
13.4.3.2.3 Narrative 378
13.4.3.2.4 Formalisation 378
13.4.3.3 Order Management Attributes 379
13.4.3.3.1 Narrative 379
13.4.3.3.2 Formalisation 379
13.4.3.3.3 Narrative 379
13.4.3.3.4 Formalisation 379
13.4.3.4 Inventory Attributes 380
13.4.3.4.1 Narrative 380
13.4.3.4.2 Formalisation 380
13.4.3.4.3 Narrative 380
13.4.3.4.4 Formalisation 380
13.4.3.5 Warehouse Attributes 380
13.4.3.5.1 Narrative 380
13.4.3.5.2 Formalisation 381
13.4.3.5.3 Narrative 381
13.4.3.5.4 Formalisation 381
13.4.3.6 Supplier Attributes 381
13.4.3.6.1 Narrative 381
13.4.3.6.2 Formalisation 381
13.4.3.6.3 Narrative 382
13.4.3.6.4 Formalisation 382
13.4.3.7 Courier Attributes 382
13.4.3.7.1 Narrative 382
13.4.3.7.2 Formalisation 382
13.4.3.7.3 Narrative 382
13.4.3.7.4 Formalisation 382
13.5 Merchandise 383
13.5.1 "Unique Identity" 383
13.5.2 "Mereology" 383
13.5.3 "Attributes" 383
13.5.4 Representation 384
13.6 Perdurants 384
13.6.1 Channels 384
13.6.2 Behaviours 384
13.6.2.1 Customer Behaviour 385
13.6.2.1.1 Narrative 385
13.6.2.1.2 Formalisation 385
13.6.2.2 Order Management Behaviour 386
13.6.2.2.1 Narrative 386
13.6.2.2.2 Formalisation 386
13.6.2.2.3 Narrative 386
13.6.2.2.4 Formalisation 386
13.6.2.2.5 Narrative 387
13.6.2.2.6 Formalisation 387
13.6.2.2.7 Narrative 387
13.6.2.2.8 Formalisation 387
13.6.2.2.9 Narrative 388
13.6.2.2.10 Formalisation 388
13.6.2.2.11 Narrative 388
13.6.2.2.12 Formalisation 389
13.6.2.3 Inventory Behaviour 389

13.6.2.3.1 Narrative 389
13.6.2.3.2 Formalisation 389
13.6.2.3.3 Narrative 389
13.6.2.3.4 Formalisation 389
13.6.2.3.5 Narrative 390
13.6.2.3.6 Formalisation 390
13.6.2.3.7 Narrative 390
13.6.2.3.8 Formalisation 391
13.6.2.3.9 Narrative 391
13.6.2.3.10 Formalisation 391
13.6.2.3.11 Narrative 392
13.6.2.3.12 Formalisation 392
13.6.2.4 Warehouse Behaviour 392
13.6.2.4.1 Narrative 392
13.6.2.4.2 Formalisation 392
13.6.2.4.3 Narrative 393
13.6.2.4.4 Formalisation 393
13.6.2.4.5 Narrative 393
13.6.2.4.6 Formalisation 393
13.6.2.4.7 Narrative 394
13.6.2.4.8 Formalisation 394
13.6.2.4.9 Narrative 395
13.6.2.4.10 Formalisation 395
13.6.2.4.11 Narrative 395
13.6.2.4.12 Formalisation 395
13.6.2.5 Supplier Behaviour 396
13.6.2.5.1 Narrative 396
13.6.2.5.2 Formalisation 396
13.6.2.5.3 Narrative 396
13.6.2.5.4 Formalisation 396
13.6.2.5.5 Narrative 397
13.6.2.5.6 Formalisation 397
13.6.2.6 Courier Service Behaviour 397
13.6.2.6.1 Narrative 397
13.6.2.6.2 Formalisation 397
13.6.2.6.3 Narrative 398
13.6.2.6.4 Formalisation 398
13.6.2.6.5 Formalisation 398
13.6.3 System Initialisation 398
13.7 Conclusion 399
13.7.1 Critique of the 𝔻𝔸&𝔻 Model 399
13.7.2 Proofs about Models 400
13.7.3 Comparison of Models 400
13.7.3.1 “Minor” Discrepancies 400
13.7.3.2 Use of Diagrams 400
13.7.3.3 Interleave versus “True” Concurrency 401
13.7.4 Development Management 401
13.7.5 What Next ? 402

We report an exercise in modeling a retail system such as outlined in both [20, Bjørner, 2002] and [88, Fettke & Reisig, Dec. 21, 2020]. In the present exercise we try follow [88, Fettke & Reisig] – but we do so slavishly following the *domain analysis & description* (𝔻𝔸&𝔻) method of [55, Domain Science & Engineering, Chapters 3–6, Bjørner 2021][132] (𝔻𝕊&𝔼).[133]

---

[132] [55] is scheduled to be published by **Springer** in their *EATCS Monographs in Theoretical Computer Science* series, Winter/Spring of 2021. Till such a time you may fund an electronic copy at `www.imm.dtu.dk/˜dibj/2020/mono/mono.pdf`. That electronic copy may, from time to time, be updated as I “improve” on its text.

[133] Work on this document started December 28, 2020.

13.1 Two Approaches to Modeling

In this report we present a model of a customer/retailer/supplier/... market. We do it in the more-or-less classical style which emanated from the denotational-like formal specification of programming languages and lead to VDM [63, 65] – and from there to RAISE [93]. There are other approaches to modeling discrete systems. One is by means of *symbolic* Petri nets [133, 134, 135, 136, 137].

13.1.1 Domain Science & Engineering: 𝔻𝕊&𝔼

At the center of 𝔻𝕊&𝔼 stands 𝔻𝔸&𝔻: a *domain analysis & description* method. 𝔻𝔸&𝔻 is first outlined in [36, 39, Bjørner, 2010]. 𝔻𝔸&𝔻 found a more final form in [48, 53, Bjørner, 2016-2019]. [53] form the core chapters, Chapters 3–6, of [55]. That forthcoming Springer monograph, [55], covers the 𝔻𝕊&𝔼 concept: *domain science & engineering*.

In this report we shall *slavishly* follow the doctrines of the 𝔻𝔸&𝔻 method. First we consider endurants[134] and "within" our analysis & description of endurants we first focus on their so-called *external qualities* ("form, but not content"), then on *internal qualities: unique identifiers, mereology* and *attributes*. Then, by *transcendental deduction*, we "morph" some endurants into perdurants[135], that is, behaviours. Here we first consider the *channels* and the messages sent over channels between behaviours, before we consider these latter. We do so in the style of [RSL's [92]] CSP [99, 100, 141, 144, 101].

It may seem a long beginning before we get to "process-oriented" modeling.

But a worthwhile thing is worth doing right, hence carefully !

It seems to this author that the ℍ𝔼ℝ𝔸𝕂𝕃𝕀𝕋 approach, in keeping with its name, from the first beginning, considers "all as flowing", that is, as [Petri net-like] processes.

13.1.2 ℍ𝔼ℝ𝔸𝕂𝕃𝕀𝕋: http://heraklit.dfki.de/

Based on Petri net ideas [133, 134, 135, 136, 137] Wolfgang Reisig has conceived and researched ℍ𝔼ℝ𝔸𝕂𝕃𝕀𝕋. In a number of reports and papers, [83, 86, 89, 88, 87, 85, 84], Peter Fettke and Wolfgang Reisig has developed the ℍ𝔼ℝ𝔸𝕂𝕃𝕀𝕋 theory & practice of modeling, what they call *service systems*.

The present report "mimics" [88, ℍ𝔼ℝ𝔸𝕂𝕃𝕀𝕋 *case study: retailer*] in providing a Domain Analysis & Description (𝔻𝔸&𝔻)-oriented model of "the same" domain !

The ℍ𝔼ℝ𝔸𝕂𝕃𝕀𝕋 *retailer case study* [88] straddles three concerns: presenting the ℍ𝔼ℝ𝔸𝕂𝕃𝕀𝕋 methodology, its mathematical foundation and the retailer case study. The 𝔻𝔸&𝔻 case study presented in this report makes use of RSL, the RAISE Specification Language [92] – and can thus concentrate on the case study. The semantics of the RSL-expressed case study is that derived from the semantics of RSL, notably its CSP [99, 100, 141, 144, 101] "subset".

13.2 The retailer market Case Study

The following case study is based on [88]. It does not, in the present version, follow the domain of [88] strictly. But I am quite sure that any discrepancies can be easily incorporated into the present model.

13.2.1 Three Rough Sketches

It is good domain modeling development practice to start a domain modeling project with one or more alternative rough sketch informal descriptions. But they are to be just such rough sketches. No formal meaning is to be attached to these rough sketches. They are meant to get the domain modeling project team "aligned".

We present three, obviously "overlapping", rough sketches.

---

[134] Endurants, colloquially speaking, "end up" as data in the computer.

[135] Perdurants, colloquially speaking, "end up" as processes in the computer.

13.2.1.1 Identification of "Main Players"

We **rough sketch** narrate a description of the domain.

The domain is that of a set of **customers**, a set of **retailers**, a set of **suppliers**, a set of **courier services**. **Retailers** each embody three sub-components: an **order management**, an **inventory**; and a **warehouse**.

See Fig. 13.1

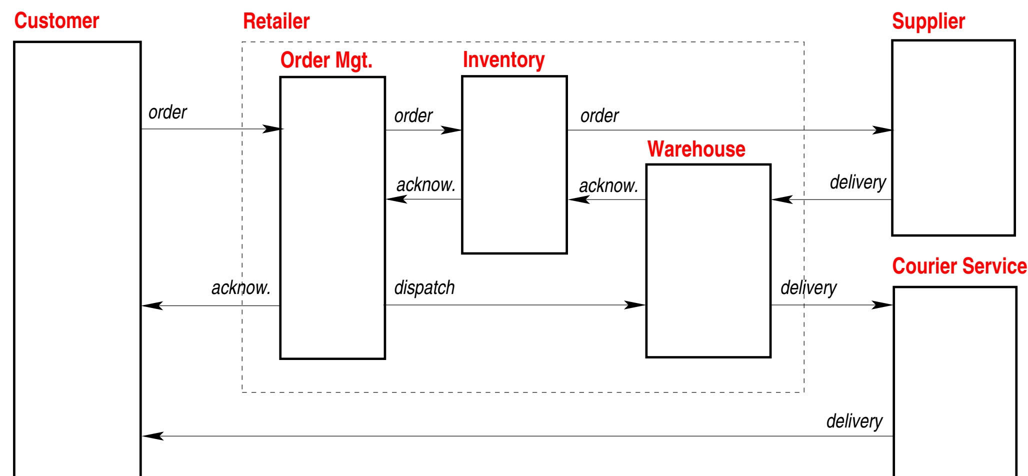


Fig. 13.1 A *Market System*

136

Customers **order** merchandise from retailers' order management. They in tern **order** that merchandise from their inventory [management]. If inventory [management] judges that they have the needed quantity in their warehouse, they **acknowledge** the order management. If inventory [management] judges that they do not have the needed quantity in their warehouse, they proceed to **order** a sufficient quantity of the desired merchandise from a supplier. The supplier eventually **deliver** a quantity to the warehouse of the ordering retailer. That warehouse **acknowledges** receipt to its inventory which eventually **acknowledges** that receipt to its order management. The order management **acknowledges** the customer order and notifies its warehouse of a proper **dispatch**. The warehouse **delivers** the desired merchandise quantity to a courier service which subsequently **delivers** that desired merchandise quantity to the customer.

It is thus we see that there are essentially three four kinds of transactions between market "players": **orders**, **acknowledgments**, **dispatches** and **deliveries**.

In the formalisations to follow we shall refer to customers as c:C, order managements as om:OM, inventories as iv:IV, suppliers as s:S and courier services as cs:CS.[137]

13.2.1.2 Main Transaction Sequences

Customers issue **purchase orders** for **merchandise** from retailers' **order management**; receive **order acknowledgments** from retailers' **order management**; and receive **customer delivered merchandise** (via retailers' warehouses) from **courier services**.

---

[136] None of the figures in this report, Figs. 13.1, 13.2 on the following page, 13.3 on page 368, 13.4 on page 369 and 13.5 on page 370, are formal. That is, they do not add to or detract from the meaning of the formulas otherwise shown in this report. They merely "support", by graphics, the narrative text.

[137] We shall, corresponding, prefix the transaction names: C_OM_Order, OM_I_Order, IV_S_Order, IV_WH_Delivery, WH_CS_Delivery, CS_C_Delivery, WH_IV_Ack, IV_OM_Ack, OM_C_Ack, OM_WH_Dispatch, or some suitable variants thereof.

Retailers' order management inquire with its inventory as to the availability of ordered merchandise; await acknowledgment of availability (of merchandise) from its inventory; informs customer of availability (order acknowledgment); and and dispatch order to warehouse when available.

Retailers' inventory issues acknowledgment of merchandise to order management; issues wholesale orders for supply of merchandise, "when out-of-stock", from suppliers; and receive acknowledgment of supplies from suppliers.

Retailers' warehouse receive merchandise deliveries from suppliers; informs inventory management of merchandise availability; accepts dispatch orders from order management; and forward merchandise for such customer merchandise dispatches to courier services.

Etcetera.

13.2.1.3 Detailed Sketch

We refer to Fig. 13.2.

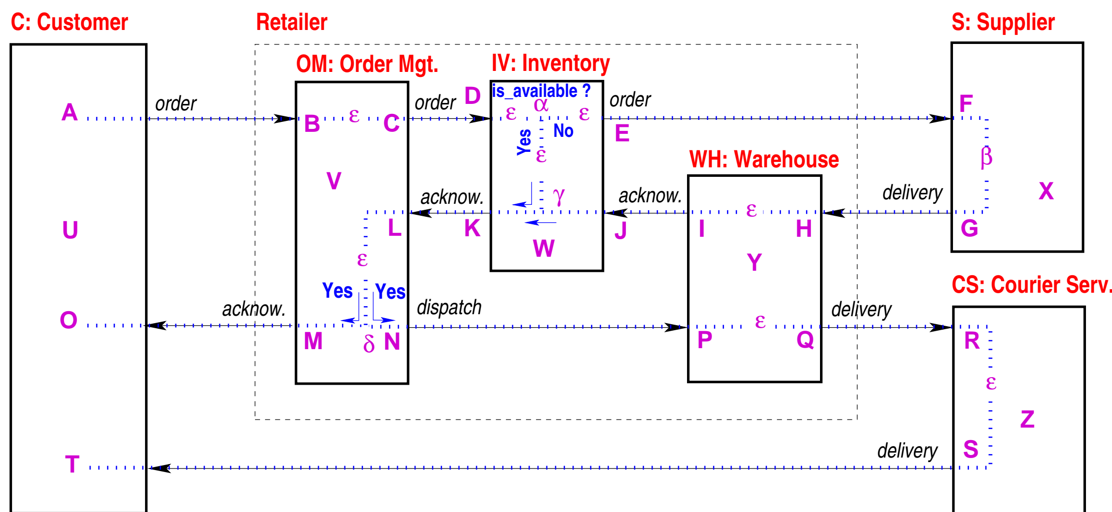


Fig. 13.2 Transaction Sequences

- A It all starts with a customer issuing a purchase order. It is date-time stamped with the customers unique identifier.
  Since no customer can issue more than one such order at a time, such date-time-customer identification is unique and can serve as the unique customer order identification across the market.
  Once the customer has issued the order request it either O awaits replies from some retailer's order management or T some courier service's delivery (of otherwise ordered products) or U resumes other business!
- B The customer order is received by some retailer's order management. That order management makes a note of the incoming order and posits that note in a 'work-to-do' dossier.
- ϵ_B At some time the order management selects an arbitrary "what-to-do-next" note from its dossier. If it is that if an customer order – arising from B – then it
- C issues an inquiry to its inventory as to the availability of the quantity of the named product of the customer order.
- D The inventory receives an inventory inquiry. The inventory makes a note of the incoming order and deposits that note in its 'work-to-do' dossier.

- ϵ_D At some time the inventory selects an arbitrary "what-to-do-next" note from its dossier. If it is that if an inventory inquiry – arising from D – then it
- α examines whether the quantity of the named product of the customer order is "on-hand" (in the retailer's warehouse, as recorded in the inventory).
- E If not the inventory issues a wholesale order to a supplier.
- F The supplier receives a wholesale order. The supplier makes a note of the wholesale order and deposits that note in its 'work-to-do' dossier.
- β It may take same time to respond to the wholesale request. For example, if the supplier first has to manufacture or otherwise get hold of the requested supply.
- G Eventually the supplier transfers the requested quantity of named merchandise to the requesting retailer's warehouse.
- H The warehouse receives this delivery and – eventually - stores it –
- I while notifying its inventory (management) of availability of the [previously] requested merchandise.
- J The inventory receives this notification. It make a note thereof and deposits it in its 'work-to-do' dossier.
- γ Either inquiry α lead to a positive result, or, as now (M) such an inquiry would be positive.
- K Eventually the inventory can inform order management of order availability.
- L Order management receives positive acknowledgment and deposits notes in its 'work-to-do' dossier as to acknowledging the customer of its order and informing the warehouse of its delivery.
- M Eventually order management gets around to service this note:
  - ∞ (D,α,**Yes**) and (D,α,**No**,E,F,G,J,I,J) order management informs the customer of upcoming order delivery
  - ∞ N while also, "at the same time", issuing an order dispatch to its warehouse.
- O The customer receives this information.
- P The warehouse receives this dispatch and makes a note thereof in its 'work-to-do' dossier.
- Q The warehouse eventually issues a delivery order, with ordered merchandise, to a courier service.
- R The courier service receives this delivery and makes a note thereof in its 'work-to-do' dossier.
- S The courier service eventually dispatches the delivery to the customer.
- T The customer, finally, receives the ordered quantity of merchandise.

13.2.1.4 Transitions

In the technical terms of `Petri nets`, the ten (10) horisontal arrows of Fig. 13.2 on the preceding page represent transitions as in **Place-Transition** `nets`. They are labeled by pairs of upper case alphabetic characters: **A–B, C–D, E–F, G–H, I–J, K–L, M–N, O–P, Q–R,** and **S–T**. In the technical terms of `CS`, these ten transitions correspond to pairs of `CSP` input [*ch[...] ?*] and output [*ch[...] ! msg*] clauses. You will find these clauses **highlighted in blue** in Sect. **13.6.2**:

- **A–B**: Items 436 Page 385 and 451 Page 387
- **C–D**: Items 457 Page 387 and 479 Page 390
- **E–F**: Items 500 Page 391 and 546 Page 396
- **G–H**: Items 515 Page 393 and 554 Page 397
- **I–J**: Items 534 Page 395 and 482 Page 390
- **K–L**: Items 506 Page 392 and 460 Page 388
- **M–O**: Items 467 Page 388 and 438 Page 385
- **N–P**: Items 468 Page 388 and 518 Page 394
- **Q–R**: Items 540 Page 396 and 560 Page 398 and
- **S–T**: Items 565 Page 398 and 440 Page 385.

The pairs of formulas listed in each • above represents the **transition**. The formula text from the behaviour definition parameter line up up to the "transition" line defines the **place**. Thus the `RSL/CSP` definition that we shall present, in a sense, corresponds to place-transition nets where each transition has exactly two inputs and two outputs. The other way around: Place-transition nets where transitions have different numbers of inputs, respectively outputs, can be likewise "mimicked" by appropriate `RSL/CSP` definitions.

13.3 Endurants: External Qualities

We now begin the proper, methodical description of the retailer, i.e., the market system. That description is presented in Sects. **13.3–13.6**.

We refer to [55, *Chapter 3*].

13.3.1 Main Decompositions

13.3.1.0.1 Narrative

339. Our market system comprises
340. a customer aggregate,
341. a retailer aggregate,
342. a supplier aggregate and
343. a courier service aggregate.

We consider all these aggregates to be *structures* in the sense of [55, Sect. 4.10].

13.3.1.0.2 Formalisation

```
type
339. MKT
340. CSTa
341. RETa
342. SUPa
343. CSa
value
340. obs_CSTa: MKT → CSTa
341. obs_RETa: MKT → RETa
342. obs_SUPa: MKT → SUPa
343. obs_CSa:  MKT → CSa
```

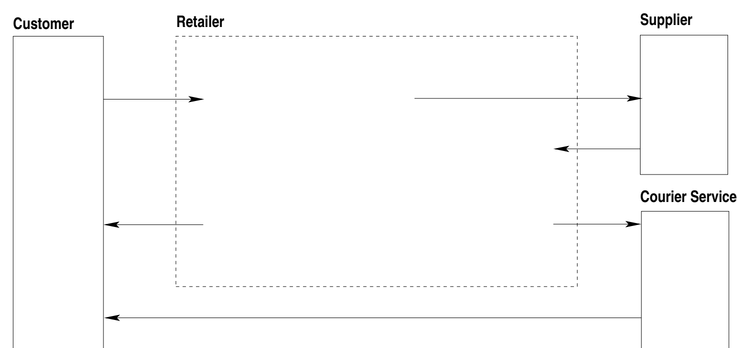


Fig. 13.3 A Simplified Market System

13.3.2 Aggregates as Sets

13.3.2.0.1 Narrative

344. The customer aggregate form a set of one or more customers.
345. The retailer aggregate form a set of one or more retailers.
346. The supplier aggregate form a set of one or more suppliers.
347. The courier service aggregate form a set of one or more courier services.

We consider all these sets to be *structures* and the customers, suppliers and courier services to be *atoms* in the sense of [55, Sects. 4.10 and 4.13].

13.3.2.0.2 Formalisation

type
344. CSTs = C-**set**, **axiom** ∀ csts:CSTs • csts≠{}
345. RETs = R-**set**, **axiom** ∀ rets:CSTs • rets≠{}
346. SUPs = S-**set**, **axiom** ∀ sups:CSTs • sups≠{}
347. CSs = CS-**set**, **axiom** ∀ cts:CSs • trss≠{}
value
344. obs_CSTs: CSTa → CSTs
345. obs_RETs: RETa → RETs
346. obs_SUPs: SUPa → SUPs
347. obs_CSs: CSa → CSs

Fig. 13.4 Aggregates as Sets

13.3.3 The Retailer

13.3.3.1 The HERAKLIT View

We focus on retailers. We treat retailers as *structures*[138,139] of three separately observable parts:

13.3.3.1.1 Narrative

348. an *order management*,
349. an *inventory*[140] and
350. a *warehouse*.

We consider order managements, inventory managements and warehouses to be *atoms* in the sense of [55, Sects. 4.13].

13.3.3.1.2 Formalisation

type
348. OM
349. IV
350. WH
value
348. obs_OM: R → OM
349. obs_IV: R → IV
350. obs_WH: R → WH

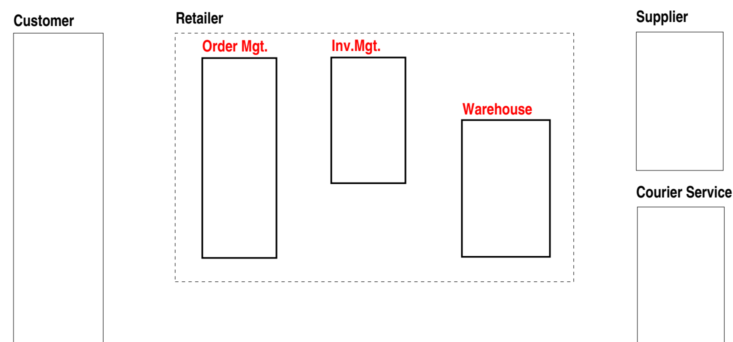


Fig. 13.5 The Retailer

[138] We refer to [88, Sect. 3.10].
[139] We dash the retailer boxes to indicate their "structure"-ness.
[140] We might have modeled a retailer inventory as an attribute of the composite part retailer.

13.3.3.2 The DS&E View

Following the DS&E "approach", i.e., "dogma", retailers might normally have been decomposed into just two components: The order management and the warehouse. Inventory would then become a programmable attribute of order management.

13.3.4 The Market System State

We refer to [55, *Sect. 3.18*]. We postulate some market system mkt. It consists of

13.3.4.0.1 Narrative

351. the market, mkt;
352. all customers *cs*;
353. all retailer order managements *oms*;
354. all retailer inventories *ivs*; and
355. all retailer warehouses *whs*;
356. all suppliers *ss*; and
357. all courier services *css*.

To obtain these we define respective extraction functions.

13.3.4.0.2 Formalisation

value
351. mkt:MKT

352. xtr_Cs: MKT → C-**set**
352. xtr_Cs(mkt) ≡ obs_CSTs(obs_CSTa(mkt))
352. *cs*:C-**set** := xtr_Cs(mkt)

353. xtr_OMs: MKT → OM-**set**t
353. xtr_OMs(mkt) ≡ {om|r:RET,om:OM•r ∈ obs_RETs(obs_RETa(mkt))∧om=obs_OM(r)}
353. *oms*:OM-**set** := xtr_OMs(mkt)

354. xtr_IVS: MKT → IV-**set**
354. xtr_IVS(mkt) ≡ {iv|r:RET,iv:IV•r ∈ obs_RETs(obs_RETa(mkt))∧iv=obs_IV(r)}
354. *ivs*:IV-**set** := xtr_IVS(mkt)

355. xtr_WHs: MKT → WH-**set**
355. xtr_WHs(mkt) ≡ {wh|r:RET,wh:WH•r ∈ obs_RETs(obs_RETa(mkt))∧wh=obs_WH(r)}
355. *whs*:WH-**set** := xtr_WHs(mkt)

356. xtr_Ss: MKT → S-**set**
356. xtr_Ss(mkt) ≡ obs_SUPs(obs_SUPa(mkt))
356. *ss*:S-**set** := xtr_Ss(mkt)

357. xtr_CSs: MKT → CS-**set**

357. xtr_CSs(mkt) ≡ obs_CSs(obs_CSa(mkt))
357. *css*:CS-**set** := xtr_CSs(mkt)

13.4 Endurants: Internal Qualities

13.4.1 Unique Identifiers

We refer to [55, *Sect. 5.2*].

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 "sameness" !

13.4.1.0.1 Narrative

358. Customers, retailer order managements, retailer inventories, retailer warehouses, suppliers and courier services all have distinct unique identifiers.
359. By UI we designate the sort of all unique identifiers.
360. We define auxiliary functions which observe the unique identifiers of all customers, retailers, suppliers and courier services of a market system.
361. *uis* name the set of all unique identifiers.

13.4.1.0.2 Formalisation

type
358. C_UI, OM_UI, IV_UI, WH_UI, S_UI, CS_UI
359. UI = C_UI | OM_UI | IV_UI | WH_UI | S_UI | CS_UI
value
358. uid_C: C → C_UI
358. uid_OM: OM → OM_UI
358. uid_IV: IN → IV_UI
358. uid_WH: WH → WH_UI
358. uid_S: S → S_UI
358. uid_CS: CS → CS_UI
axiom
358. ∀ c,c′:C•{c,c′}⊆*cs* ∧ c≠c′ ⇒ uid_C(c)≠uid_C(c′),
358. ∀ om,om′:OM•{om,om′}⊆*oms* ∧ om≠om′ ⇒ uid_OM(om)≠uid_OM(om′),
358. ∀ iv,iv′:IV•{iv,iv′}⊆*ivs* ∧ iv≠iv′ ⇒ uid_IV(iv)≠uid_IV(iv′),
358. ∀ wh,wh′:WH•{wh,wh′}⊆*whs* ∧ wh≠wh′ ⇒ uid_WH(wh)≠uid_WH(wh′),
358. ∀ s,s′:S•{s,s′}⊆*ss* ∧ s≠s′ ⇒ uid_S(s)≠uid_S(s′),
358. ∀ cs,cs′:CS•{cs,cs′}⊆*css* ∧ cs≠cs′ ⇒ uid_CS(cs)≠uid_CS(cs′).
value
360. xtr_C_UIs: MKT → CI-**set**
360. xtr_C_UIs(mkt) ≡ {uid_C(c)|c:C•c ∈ *cs*}
360. xtr_OM_UIs: MKT → OMI-**set**
360. xtr_OM_UIs(mkt) ≡ {uid_OM(om)|om:OM•om ∈ *oms*}
360. xtr_IV_UIs: MKT → IVI-**set**

360. xtr_IV_UIs(mkt) ≡ {uid_IV(iv)|iv:IV•iv ∈ *ivs*}
360. xtr_WH_UIs: MKT → WHI-**set**
360. xtr_WH_UIs(mkt) ≡ {uid_WH(wh)|wh:WH•wh ∈ *whs*}
360. xtr_S_UIs: MKT → SI-**set**
360. xtr_S_UIs(mkt) ≡ {uid_S(s)|s:S•s ∈ *ss*}
360. xtr_CS_UIs: MKT → CSI-**set**
360. xtr_CS_UIs(mkt) ≡ {uid_CS(cs)|cs:CS•cs ∈ *css*}

360. *cuis*:CUI-**set** = xtr_C_UIs(mkt)
360. *omuis*:OMUI-**set** = xtr_OM_UIs(mkt)
360. *ivuis*:IVUI-**set** = xtr_IV_UIs(mkt)
360. *whuis*:WHUI-**set** = xtr_WH_UIs(mkt)
360. *suis*:SUI-**set** = xtr_S_UIs(mkt)
360. *csuis*:CSUI-**set** = xtr_CS_UIs(mkt)
361. *uis*:UI-**set** = *cuis* ∪ *omuis* ∪ *ivuis* ∪ *whuis* ∪ *suis* ∪ *csuis*
axiom
360. **card** *cuis* + **card** *omuis* + **card** *ivuis* + **card** *whuis* + **card** *suis* + **card** *csuis* = **card** *uis*

13.4.2 Mereology

We refer to [55, *Sect. 5.3*].

Mereology, as a logical/philosophical discipline, can perhaps best be attributed to the Polish mathematician/logician Stanisław Leśniewski [75, 44].

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 materials, or a material 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

13.4.2.1 Customer Mereology

13.4.2.1.1 Narrative

362. The mereology of a customer is a pair:
   - the set of all retail order management identifiers and
   - the set of all courier service identifiers.

13.4.2.1.2 Formalisation

type
362. C_Mer = OM_UI-**set** × CSU_I-**set**
value
362. mereo_C: C → C_Mer

362. mereo_C(c) ≡ (*omuis*,*csuis*)

13.4.2.2 Order Management Mereology

13.4.2.2.1 Narrative

363. The mereology of an order management is the triplet of
    - the set of all customer identifiers,
    - the unique identifier of the retailer's inventory and
    - the unique identifier of the retailer's warehouse.

13.4.2.2.2 Formalisation

```
type
363. OM_Mer = C_UI-set × IV_UI × WH_UI
value
363. mereo_OM: OM → OM_Mer
363. mereo_OM(om) ≡
363.     let r:R • r ∈ obs_RETs(obs_RETa(mkt))∧om=obs_OM(r) in
363.     (cis,uid_IV(obs_IV(r)),uid_WH(obs_WH(r))) end
363.     pre: ∃ r:R • r ∈ obs_RETs(obs_RETa(mkt))∧om=obs_OM(r)
```

13.4.2.3 Inventory Mereology

13.4.2.3.1 Narrative

364. The mereology of an inventory is a triplet of
    - the unique identifier of that inventory's order management,
    - the unique identifier of that inventory's warehouse and
    - the set of all supplier identifiers.

13.4.2.3.2 Formalisation

```
type
364. IV_Mer = OM_UI × WH_UI × S_UI-set
value
364. mereo_IV: IV → IV_Mer
364. mereo_IV(iv) ≡
364.     let r:R • r ∈ obs_RETs(obs_RETa(mkt))∧iv=obs_IV(r) in
364.     (uid_OM(obs_OM(r)),uid_WH(obs_WH(r)),suis) end
364.     pre: ∃ r:R • r ∈ obs_RETs(obs_RETa(mkt))∧iv=obs_IV(r)
```

13.4.2.4 Warehouse Mereology

narrative

365. The mereology of a warehouse is a quadruplet of
    - the warehouse retailer's order management identifier,
    - the warehouse retailer's inventory identifier,
    - the set of all supplier identifiers, and
    - the set of all courier service identifiers,

13.4.2.4.1 Formalisation

```
type
365. WH_Mer = OM_UI × IV_UI × _SUI-set × CS_UI-set
value
365. mereo_WH: WH → WH_Mer
365. mereo_WH(wh) ≡
365.     let r:R • r ∈ obs_RETs(obs_RETa(mkt))∧iv=obs_WH(wh) in
365.     (uid_OM(obs_OM(r)),uid_IV(obs_IV(r)),suis,csuis) end
365.     pre: ∃ r:R • r ∈ obs_RETs(obs_RETa(mkt))∧wh=obs_WH(r)
```

13.4.2.5 Supplier Mereology

13.4.2.5.1 Narrative

366. The mereology of a supplier is a pair:
    - the set of all inventory identifiers and
    - the set of all warehouse identifiers.

13.4.2.5.2 Formalisation

```
type
366. S_Mer = IV_UI-set × WH_UI-set
value
366. mereo_S: S → S_Mer
366. mereo_S(s) ≡ ({uid_IV(iv)|iv:IV • iv ∈ ivuis},{uid_WI(wh)|wh:WH • wh ∈ whs})
```

13.4.2.6 Courier Service Mereology

13.4.2.6.1 Narrative

367. The mereology of a courier service is a pair
    - the set of all warehouse identifiers and
    - the set of all customer identifiers.

13.4.2.6.2 Formalisation

type
367. CS_Mer = WH_UI-**set** × CS_UI-**set**
value
367. mereo_CS: CS → CSMer
367. mereo_CS(t) ≡ ({uid_WI(wh)|wh:WH • wh ∈ *whs*},*cuis*)

13.4.3 Attributes

We refer to [55, *Sects. 5.4–5.5*].

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

Since one can talk about transaction events between the six "players", i.e., the customers, order managements, inventories, warehouses, suppliers and courier services of the 'market' we must, really, consider their transaction histories as [programmable] attributes.

In order to deal with the attributes of these six "players" we really need first consider what they are all focused on: namely the merchandise, i.e., products, they order, store, supply and deliver. For this we refer to Sect. **13.5** on page 383.

13.4.3.1 Transactions

The 'market' is a typical *transaction-oriented* system. By a *transaction* we shall mean an *event* involving two or more "exchanges" of messages between two behaviours. Behaviours will be defined as the result of *transcendental deductions* of part endurants. With part endurants we associate attributes.

Since we can "talk" about events that "occur to parts", that is, as behaviour properties, we shall attribute some of these events to parts. So parts are attributed the transactions in which their behaviours engage (with other behaviours).

Since we can "talk" about "such-and-such" a transaction having been initiated by a behaviour at such-and-such a time ,we shall provide, with each transaction, a prefix of one or more time-stamped unique identifiers of the part/behaviour issuing the transaction.

13.4.3.1.1 Narrative

368. DaTi refers to $\mathbb{TIME}$. We refer to [55, Sect. 2.5]. The expression record_$\mathbb{TIME}$ yields a $\mathbb{TIME}$. You should think of $\mathbb{TIME}$s, for example, as of the form `November 15, 2021: 16:12` and 32 seconds (day, month, year, hour, minute, second).
369. A transaction prefix is either a pair of a customer identifier and a date-time, or is a pair of a pair of order management, inventory, warehouse, supplier or courier service identifier and a date-time, and a transaction prefix.
 The specific details of the pairings of unique identifiers and data-times is given in Items 370–379.

13.4.3.1.2 Formalisation

type
368. DaTi = $\mathbb{TIME}$
369. UI_Pref = ...
axiom
369. [...]

13.4.3.1.3 Narrative

369. More specifically the prefixes are the:
370. purchase order,
371. order inquiry,
372. wholesale order,
373. merchandise delivery,
374. merchandise availability,
375. acknowledge availability,
376. order acknowledgment,
377. dispatch order,
378. forward merchandise and the
379. customer delivery prefixes.

13.4.3.1.4 Formalisation

```
type
369.  UI_Pref = C_OM_Pref | OM_IV_Pref | IV_S_Pref | S_WH_Pref | WH_IV_Pref | IV_OM_Pref
369.            | OM_C_Pref | OM_WH_Pref | WH_CS_Pref | CS_C_Pref
370.  C_OM_Pref = (CUI×DaTi)
371.  OM_IV_Pref = (OMUI×DaTi)×C_OM_Pref
372.  IV_S_Pref = (IVUI×DaTi)×OM_IV_Pref
373.  S_WH_Pref = (SUI×DaTi)×IV_S_Pref
374.  WH_IV_Pref = (WHUI×DaTi)×S_WH_Pref
375.  IV_OM_Pref = (IVUI×DaTi)×(OM_IV_Pref|WH_IV_Pref)
376.  OM_C_Pref = (OMUI×DaTi)×IV_OM_Pref
377.  OM_WH_Pref = (OMUI×DaTi)×OM_C_Pref
378.  WH_CS_Pref = (WHUI×DaTi)×OM_WH_Pref
379.  CS_C_Pref = (CSUI×DaTi)×WH_CS_Pref
```

Two customer to order management to inventory etc. transaction prefixes might then schematically be:

13.4.3.2 Customer Attributes

In order to go about their business of being customers, customers maintain, somehow or other, in their mind, on paper, or otherwise, a number of notes – which we shall refer to as attributes.

To express some of these attributes we need first introduce some auxiliary types.

13.4.3.2.1 Narrative

380. Customers, besides unique identity, have further information: customer names, addresses, telephone nos., e-mail addresses, etc.
381. Customers have bank/credit card, i.e., payment refs.
382. An order comprises a product name, a quantity, the total price, and a payment reference. For simplicity we shall carry this *'order'* information forward in all market transactions.
383. Customers transact with retailer order managements and courier Services:
384. send purchase order to retailers;
385. receive positive acknowledgment on these orders; and
419. accept customer deliveries: a set of merchandise.

Transactions sent by customers are time-stamped with customers identity. Transactions received by customers are time-stamped [with a time-ordered, latest transaction first] grouping of handler identifications (ui:UI) – where order managements, inventories, suppliers, warehouses and courier services are the handlers.

13.4.3.2.2 Formalisation

type
380. CustName, CustAddr, CustPhon, CustEmail, ...
380. CustInfo = CustName × CustAddr × CustPhon × CustEmail × ...
381. PayRef
382. Order = (ProdNm × Quant × Price × PayRef)
383. C−Trans = C_OM_Order | OM_C_Ack | CS_C_Del
384. C_OM_Order :: C_OM_Pref × Order
385. OM_C_Ack :: OM_C_Pref × Order
419. CS_C_Del :: CS_C_Pref × Order× (M-**set**|MI-**set**)

Now the attributes.

13.4.3.2.3 Narrative

386. Customers keep a catalog of merchandise: from whom to order, price, etc. [Simplifying we consider this a static attribute.]
387. Customers keep all the merchandise they have acquired. [A programmable attribute.]
388. Customers can recall [a programmable attribute] the time-stamped transactions it has taken part in wrt. retailer order managements and courier services.

13.4.3.2.4 Formalisation

type
386. C−Catalog = ...
387. C−Merchandise = M-**set**
388. C_TransHist = C_Trans^*
axiom
388. ∀ cth:C_TransHist • [list is time-ordered]
value
386. attr_C_Catalog: C → C_Catalog

387. attr_C_Merchandise: C → C_Merchandise
388. attr_C_TransHist: C → CustTransHist

13.4.3.3 Order Management Attributes

13.4.3.3.1 Narrative

389. Order management partakes in several transactions:
384. accepting customer purchase orders;
390. passing on that order to its inventory;
399. accepting product availability acknowledgment from the inventory;
385. informing customer of product availability; and,
391. when available, directing a dispatch order to its warehouse.
392. Order management makes note of accepted, i.e., incoming messages, (B and L) by keeping a [programmable attribute] 'Work-to-do' "notice board" [a "basket", a "dossier"].

13.4.3.3.2 Formalisation

389. OM_Trans = C_OM_Order | OM_IV_Order | IV_OM_Ack | OM_C_Ack | OM_WH_Dispatch
384. C_OM_Order :: C_OM_Pref × Order
390. OM_IV_Order :: OM_IV_Pref × Order
399. IV_OM_Ack :: IV_OM_Pref × Order
385. OM_C_Ack :: OM_C_Pref × Order
391. OM_WH_Dispatch :: OM_WH_Pref × Order

Now the attributes.

13.4.3.3.3 Narrative

393. An order management 'work-to-do' dossier keeps a set of zero or more notes: customer orders and inventory acknowledgments.
394. Order management records [a static attribute] which suppliers supply which products.
395. Order management also records the programmable order management transaction history OMTransHist attribute records a time-stamped list of all order management transactions, be they vis-a-vis customers, and its retailer's inventory.

13.4.3.3.4 Formalisation

type
393. OM_WorkToDo = (C_OM_Order|IV_OM_Ack)-**set**
394. OM_ProdSupp = ProdNm $\overrightarrow{m}$ SUI-**set**
395. OM_TransHist = OM_Trans*
value
393. attr_OM_WorkToDo: OM → OrdrMgtWorkToDo
394. attr_OM_ProdSupp: OM → ProdSupp
395. attr_OM_TransHist: OM → OM_TransHist

13.4.3.4 Inventory Attributes

13.4.3.4.1 Narrative

396. Inventories partakes in several transactions:
390. accepting merchandise orders from their order management,
397. issuing wholesale order requests to a designated supplier,
398. accepting order acknowledgments from their warehouse, and
399. issuing merchandise availability messages to their order management.

13.4.3.4.2 Formalisation

396. IV_Trans = OM_IV_Order | IV_OM_Ack | IV_S_Order | WH_IV_Ack
397. IV_S_Order :: IV_S_Pref × Order × S_UI
398. WH_IV_Ack :: WH_IV_Pref × Order × WH_UI
399. IV_OM_Ack :: IV_OM_Pref × Order

13.4.3.4.3 Narrative

400. An inventory 'work-to-do' dossier (a programmable attribute) keeps a set of zero or more notes: inventory (merchandise availability) inquiry and merchandise availability.
401. The inventory (a programmable attribute) records, for every product name, its information (as listed in Items 425–430 Page 384), the name of the supplier, and the stock-in-hand.
402. The inventory also records the programmable inventory transaction history IVTransHist attribute records a time-stamped list of all inventory transactions, be they vis-a-vis order management, its retailer's warehouse or a supplier.

13.4.3.4.4 Formalisation

type
400. IV_WorkToDo = (IV_S_Order|IV_OM_Ack)-**set**
401. IV_Inventory = ProdNm $\overrightarrow{m}$ (WhoSalPrice×SugRetPrice×SalPrice×MInfo×SupNm×IV_Stock)
400. IV_Stock = **Nat**
402. IV_TransHist = IVTrans*
value
400. attr_IV_WorkToDo: IV → IV_WorkToDo
401. attr_IV_Inventory: IV → Inventory
402. attr_IV_TransHist: IV → IV_TransHist

13.4.3.5 Warehouse Attributes

13.4.3.5.1 Narrative

403. Warehouses partake in four kinds of transactions:
404. being delivered sets of a product named merchandise from suppliers,
405. informing its inventory of (wholesale) supplier delivery,

406. being ordered by its order management, to dispatch merchandise to customers and
407. delivering merchandise to couriers (for them to deliver to customers).

13.4.3.5.2 Formalisation

403. WH_Trans = S_WH_Del | WH_IV_Ack | OM_WH_Del | WH_CS_Del
404. S_WH_Del = S_WH_Pref × Order × M-**set**
405. WH_IV_Ack = WH_IV_Pref × Order
406. OM_WH_Dispatch = OM_WH_Pref × Order
407. WH_CS_Del = WH_CS_Pref × Order × M-**set**

13.4.3.5.3 Narrative

408.
409. The programmable warehouse Store attribute reflects, for every product name the zero, one or more merchandise of that name.
410. The programmable warehouse WHTransHist attribute records a time-stamped list of all warehouse transactions, be they vis-a-vis suppliers, its retailer's inventory, its retailer's order management, and customers.

13.4.3.5.4 Formalisation

type
408. WH_WorkToDo = (S_WH_Del|OM_WH_Dispatch)-**set**
409. WH_Store = ProdName $\overrightarrow{m}$ M-**set**
410. WH_TransHist = WH_Trans*
value
408. attr_WH_WorkToDo: WH → WH_WH_WorkToDo
409. attr_WH_Store: WH → WH_Store
410. attr_WH_TransHist: WH → WH_TransHist

13.4.3.6 Supplier Attributes

13.4.3.6.1 Narrative

411. Suppliers, in this model, partake in two transactions:
412. accepting wholesale orders for merchandise from retailers' inventories, and
413. delivering such merchandise orders to retailers' warehouses.

13.4.3.6.2 Formalisation

411. S_Trans = IV_S_Order | S_WH_Del
412. IV_S_Order = IV_S_Pref × Order
413. S_WH_Del = S_WH_Pref × Order × M-**set**

13.4.3.6.3 Narrative

414. The programmable supplier attribute S_WorkToDo temporarily contains 'replicas' of "incoming" IV_S_Orders.
415. The programmable supplier attribute S_Products reflects, for every product name a sufficient[141] number of merchandise of that name.
416. The programmable supplier attribute S_TransHist records a time-stamped list of all supplier transactions, be they vis-a-vis retailers' inventory, and retailers' warehouses.

13.4.3.6.4 Formalisation

type
414. S_WorkToDo = IV_S_Order-**set**
415. S_Products = ProdNm $\overrightarrow{m}$ M-**set**
416. S_TransHist = S_Trans*
value
414. attr_S_WorkToDo: S → S_WorkToDo
415. attr_S_Products: S → S_Products
416. attr_S_TransHist: S → S_TransHist

13.4.3.7 Courier Attributes

13.4.3.7.1 Narrative

417. Courier services, in this model, partake in two transactions:
418. accepting merchandise delivery orders to customers from retailers' order management, and
419. delivering merchandise to customers

13.4.3.7.2 Formalisation

type
417. CS_Trans = WH_CS_Del | CS_C_Del
418. WH_CS_Del = WH_CS_Pref × Order × M-**set**
419. CS_C_Del = CS_C_Pref × Order× M-**set**

13.4.3.7.3 Narrative

420. The programmable courier service attribute CS_WorkToDo reflects current, "live" deliveries, and
421. the programmable attribute CS_TransHist the time stamped history of transactions.

13.4.3.7.4 Formalisation

type

141

420. CS_WorkToDo = CS_C_Del-**set**
421. CS_TransHist = CS_Trans^*
value
420. attr_CS_WorkToDo: CS → CS_WtD
421. attr_CS_TransHist: CS → CS_TransHist

13.5 Merchandise

Merchandise (in [88]: Goods) are, using DS&E, modeled as parts. In [88] they are not considered beyond being somehow identified. It is not clear.

422. We shall model merchandise as atomic parts.

type
422. M

13.5.1 "Unique Identity"

423. As parts merchandise have unique identity.
424. Although we shall treat merchandise as behaviours we shall assume that merchandise identities are distinct from any other unique identities of the market.

type
423. MI
value
423. uid_M: M → MI
axiom
424. ∀ m:M • uid_M(m) $\notin$ *cuis* ∪ *omuis* ∪ *ivuis* ∪ *whuis* ∪ *suis* ∪ *tuis*

13.5.2 "Mereology"

Although merchandise, throughout its lifetime, can be related to suppliers, warehouses, courier services and customers we shall omit modeling the mereology of merchandise.

13.5.3 "Attributes"

We suggest the following merchandise attributes:

425. product name;
426. wholesale price;
427. suggested retail price;
428. sales price;
429. actual price;

430. further product information: goods category, weight, packaging measures, volume, manufacturer (with place-of-origin), manufacturing date, sale-by-date, an "how-to-use" guide, guarantee, etc., etc.

type
425. ProdNm
426. WhoSalPrice
427. SugRetPrice
428. SalPrice
429. ActPrice
430. ProdInfo

13.5.4 Representation

We shall not be concerned with the representation of attributes.

13.6 Perdurants

We refer to [55, *Chapters 6–7*].

By transcendental deduction we now "morph" endurants into perdurants. Parts "morph" into behaviours, here modeled in the style of CSP. Their mereology determine the *channels* between part processes.

13.6.1 Channels

We refer to [55, *Sect. 7.5*].

In this report we shall postulate a channel array indexed by pairs (expressed as two-element sets) of unique identifiers. These identifiers are prescribed in the mereology of the relevant parts.

431. So there is a channel whose index sets allow the expression of communication between customers, order management, inventories, warehouses, suppliers and courier services.
432. The type of the messages communicated is the union type of the customer, order management, inventory [management], warehouse, supplier and courier service transactions.

channel
431. {ch[{ui,ui′}]|ui,ui′:UI•ui≠ui′∧{ui,ui′}⊆*uis*}:Channel_Trans
type
432. Channel_Trans = C_Trans|OM_Trans|IV_Trans|WH_Trans|S_Trans|CS_Trans

13.6.2 Behaviours

We refer to [55, *Sects. 7.6–7.8*].

There now follows a sequence of informal narrative and formal specification texts. The formal texts, in a sense, are a culmination of all the previous formal definitions. The formulas involve

rather may identifiers. Some are defined locally, some as behaviour function definition parameters, others in previous formal definitions.

13.6.2.1 Customer Behaviour

13.6.2.1.1 Narrative

433. Customers alternate between retailer shopping and otherwise going about their daily life. Shopping manifests itself in three related events:
   - A the customer issuing a purchase order;
   - O the receiving of acknowledgment of upcoming delivery;
   - T the final acceptance of delivery.

   Daily life is "modeled" by T. Customers alternate, internal don-deterministically, ⊓, between these four events.
434. A When internal non-deterministically choosing to order merchandise, the customer must decide on which retailer, product, how many and at what cost.
435. The customer then assembles a purchase order
436. which it sends to some retailer's order management.
   We refer to [55, Sect. 2.5.3] for understanding the rôle of record_TIME.
   We presently omit defining date.
437. Whereupon the customer resumes being a customer, however with updated transaction history.
438. O At some time the customer receives an acknowledgment from a retailer's order management as to the [positive] acceptance of an order which was purchased some while ago (omui,dati).
439. The customer records this in its transaction history while resuming being a customer
440. T At some time the customer receives the delivery of previously ordered merchandise.
441. The customer records the identities (as well as the merchandise) and
442. resumes being a customer.
443. U Et cetera.[142]

13.6.2.1.2 Formalisation

```
value
434. C: c_ui:CUI×c_mer:(omuis,csuis):C_Mer×C_Catalog → (C_Merchandise×C_TransHist)
434.          in out { ch[{c_ui,om_ui}] | om_ui:OMUI • om_ui ∈ omuis }
434.          in { ch[{c_ui,cs_ui}] | cs_ui:CSUI • cs_ui ∈ csuis } Unit
434. C_Beh(c_ui,c_mer:(omuis,csuis),c_ctlg)(c_merch,c_hist) ≡
434.    A let (om_ui,order) = decide_on_purchase((custinfo,mertbl),c_hist) in
436.        A–B ch[{c_ui,om_ui}] ! ordr:C_OM_Order(((c_ui,record_TIME())),order) ;
437.        C(c_ui,c_mer,c_ctlg)(c_merch,⟨ordr⟩⁀c_hist) end
438. ⊓ O let M–O ack:OM_C_Ack(prefix,order) = ch[{om_ui,c_ui}] ? in
439.      C(cui,cmer,c_ctlg)(merch,⟨ack⟩⁀c_hist) end
440. ⊓ T let S–T del:CS_C_Del(prefix,order,ms) = ch[{cs_ui,c_ui}] ? in
441.      let ms_uis = {uid_MI(m)|m:M•m ∈ ms} in
442.      C(c_ui,c_mer,c_ctlg)(merch ∪ ms,⟨CS_C_Del(prefix,order,ms_uis)⟩⁀c_hist) end end
```

[142] We leave it to the reader to be more specific. The "etcetera" could, for example, describe possible updates to the catalog and merchandise repository.

443. ⊓ U ... C(cui,cmer,ctlg′)(merch′,c_hist)

13.6.2.2 Order Management Behaviour

13.6.2.2.1 Narrative

444. Being order management, **OM**, manifests itself in six events:
445. B accepting customer order,
446. C offering inventory order,
447. L accepting inventory acknowledgment,
448. M offering **OM** acknowledgment acknowledgment to customer, and
     N offering dispatch order to warehouse, and
449. V doing other **OM** business.
450. The **OM** behaviour internal non-deterministically (445., 446., 447., 448. and 449.) alternates between **B, C, L, M, N** and **V**:

13.6.2.2.2 Formalisation

```
444. OM: om_ui:OM_UI × (ommer:(cuis,ivui,whui)):OM_Mer × OM_ProdSupp →
444.        (OM_WorkToDo × OM_TransHist)
444.           in out { ch[{c_ui,om_ui}] | c_ui:CUI•c_ui ∈ cuis }
444.           in out ch[{om_ui,iv_ui}] out ch[{om_ui,wh_ui}]  Unit
444. OM(om_ui,om_mer:(cuis,iv_ui,wh_ui),om_prodsupp)(om_wtd,om_hist) ≡
445.        B OM.C_OM_Order(om_ui,om_mer,om_prodsupp)(om_wtd,om_hist)
446.     ⊓ C OM.OM_IV_Order(om_ui,om_mer,_om_prodsupp)(om_wtd,om_hist)
447.     ⊓ L OM.IV_OM_Ack(om_ui,om_mer,om_prodsupp)(om_wtd,om_hist)
448.     ⊓ M,N OM.Handle_Input(om_ui,om_mer,om_prodsupp)(om_wtd,om_hist)
449.     ⊓ V ... OM(om_ui,om_mer,om_prodsupp)(om_wtd,om_hist)
```

13.6.2.2.3 Narrative

451. B **OM.C_OM_Order** external non-deterministically offers to accept purchase orders from customers.
452. In response, **OM.C_OM_Order** makes a note of this request in its work-to-do dossier, that is, of eventually issuing an inventory order.
453. Thereupon **OM.C_OM_Order** resumes being 'order management' with an appropriately updated work-to-do state.

13.6.2.2.4 Formalisation

```
value
445. COM._OM_Order: om_ui:OM_UI × (om_mer:(cuis,iv_ui,wh_ui)):OM_Mer × OM_ProdSupp →
445.        (OM_WorkToDo × OM_TransHist)
```

```
445.        in out { ch[{c_ui,om_ui}] | c_ui:CUI•c_ui ∈ cuis }
445.        in out ch[{om_ui,iv_ui}] out ch[{om_ui,wh_ui}] Unit
445. COM._OM_Order(omui,om_mer:(cuis,ivui,whui),om_prodsupp)(om_wtd,om_hist) ≡
451.   B ⌈⌉⌊ { let A–B ordr:C_OM_Ordr(((c_ui,dati)),order)=ch[{c_ui,om_ui}] ? in
452.         let om_wtd′ = om_wtd ∪ {OM_IV_Ordr(((c_ui,dati)),order,_)} in
453.         OM(om_ui,om_mer,om_prodsupp)(om_wtd′,⟨ordr⟩⌢omhist)
445.        | c_ui:C_UI • c_ui ∈ cuis end end }
```

13.6.2.2.5 Narrative

454. C **OM.OM_IV_Order** inquires as to whether order_management has a 'work-to-do' note on ordering a quantity of a named product.
455. If so, it selects that note.
456. It then selects a suitable product supplier and a sufficient quantity of the named product.
457. Finally it offers an inquiry to the inventory.
458. Whereupon it resumes being **OM**.
459. If **OM.OM_IV_Order** finds no such note it resumes being **OM**.

13.6.2.2.6 Formalisation

```
value
446. OM.OM_IV_Order: om_ui:OMUI × (om_mer:(cuis,ivui,whui)):OM_Mer × OM_ProdSupp →
446.      (OM_WorkToDo × OM_TransHist)
446.        in out { ch[{c_ui,om_ui}] | c_ui:C_UI•c_ui ∈ cuis }
446.        in out ch[{om_ui,iv_ui}] out ch[{om_ui,wh_ui}] Unit
446. OM.OM_IV_Order(om_ui,om_mer:(cuis,iv_ui,wh_ui),om_prodsupp)(om_wtd,om_hist) ≡
454.   C if OM_IV_Ordr(((c_ui,dati)),order,_) ∈ wtd
455.       then let ordr:OM_IV_Ordr(((c_ui,dati)),order,_) • ordr ∈ wtd in
456.            let s_ui:S_UI • find_supplier(order)(om_prodsupp), dati′ = record_TIME() in
457.            C–D ch[{om_ui,iv_ui}] ! ordr′:OM_IV_Ordr(((om_ui,dati′),(c_ui,dati)),order) ;
458.            OM(om_ui,om_mer,om_prodsupp)(om_wtd \ {ordr},⟨ordr⟩⌢om_hist) end end
459.       else OM(om_ui,om_mer,om_prodsupp)(om_wtd,om_hist) end

456.  find_supplier: Order × OM_ProdSupp → S_UI, find_supplier(order)(om_prodsupp) ≡ ...
```

13.6.2.2.7 Narrative

460. L **OM.IV_OM_Ack** offers to accept an order acknowledgment from the retailer inventory.
461. It places this acknowledgment in the **OM**'s 'work-to-do' "basket" as a "matching" pair of customer acknowledgment and warehouse order dispatch notes.
462. And resumes being **OM**.

13.6.2.2.8 Formalisation

```
value
447. OM.IV_OM_Ack: OM_UI × OM_Mer × OM_ProdSupp →
```

```
447.         (OM_WorkToDo × OM_TransHist)
447.             in out { ch[ {c_ui,om_ui} ] | c_ui:C_UI•c_ui ∈ cuis }
447.             in out ch[ {om_ui,iv_ui} ] out ch[ {om_ui,wh_ui} ]  Unit
447.  OM.IV_OM_Ack(om_ui,om_mer:(cuis,iv_ui,wh_ui),m_prodsupp)(om_wtd,om_hist) ≡
460.    [L] let [K–L] iv_om_ack:IV_OM_Ack(pref,ordr) = ch[ {om_ui,iv_ui} ] ? in
461.        let om_wtd′ = om_wtd ∪ {OM_C_Ack(((om_ui,_),pref),ordr),
461.                                    OM_WH_Dispatch(((om_ui,_),pref),ordr)} in
462.        OM(om_ui,om_mer,om_prodsupp)(om_wtd′,⟨iv_om_ack⟩⁀om_hist) end end
```

13.6.2.2.9 Narrative

463. [M,N] If a suitable, i.e., "matching", pair of customer acknowledgment and warehouse order dispatch notes, can be found in the 'work-to-do' dossier,
464. then time is recorded,
465. the pair of to-do notes identified and
466. that pair removed from the work-to-do basket, whereupon
467. the customer is notified of the acknowledgment, and
468. the warehouse is notified of the order dispatch;
469. an updated order management transaction history is prepared, and
470. the **OM.OM_C_Ack_OM_WH_Desp** resumes being **OM_Beh**;
471. else **OM.OM_C_Ack_OM_WH_Desp** resumes being **OM**.

13.6.2.2.10 Formalisation

```
value
448.  OM.Handle_Input: OM_UI×OM_Mer×OM_ProdSupp→(OM_WorkToDo×OM_TransHist) Unit
448.         (OM_WorkToDo × OM_TransHist)
448.             in out { ch[ {c_ui,om_ui} ] | c_ui:CUI•c_ui ∈ cuis }
448.             in out ch[ {om_ui,iv_ui} ] out ch[ {om_ui,wh_ui} ]  Unit
448.  OM.Handle_Input(om_ui,om_mer:(cuis,iv_ui,wh_ui),om_prodsupp)(om_wtd,om_hist) ≡
463.    [M] if ∃ two:{IV_OM_Ack(((om_ui,_),pref),ordr),OM_WH_Dispatch(((om_ui,_),pref),ordr)} • two ⊆ om_wtd
464.          then let dati′ = record_TIME(),
465.                   iv_om_ack = OM_C_Ack(((om_ui,_),pref),ordr) • iv_om_ack ∈ om_wtd,
465.                   om_wh_dis = OM_WH_Dispatch(((om_ui,_),pref),ordr) • om_wh_dis ∈ wtd,
466.                   om_wtd′ = om_wtd \ {iv_om_ack,om_wh_dis} in
467.               { [M–O] ch[ {om_ui,c_ui} ] ! om_c_ack′:OM_C_Ack(((om_ui,dati′),pref),ordr) ||
468.                 [N–P] ch[ {om_ui,wh_ui} ] ! om_wh_dis′:OM_WH_Dispatch(((om_ui,dati′),pref),ordr) } ;
469.               let om_hist′ = ⟨om_c_ack′,om_wh_dis′⟩⁀om_hist in
470.               OM(om_ui,om_mer,om_prodsupp)(om_wtd′,om_hist′) end end
471.          else  OM(om_ui,om_mer,om_prodsupp)(om_wtd,om_hist) end
```

13.6.2.2.11 Narrative

472. [V] We leave this behaviour further undefined.

13.6.2.2.12 Formalisation

472. V ...

13.6.2.3 Inventory Behaviour

13.6.2.3.1 Narrative

473. The **IV** (inventory) behaviour communicates with the order management and the warehouse of the retailer to which it belongs, and with a variety of suppliers.
The **IV** behaviour alters between External non-deterministically offering to accept
474. **D** order input communications from its order management;
475. **J** order acknowledgment input communications from its warehouse; and
476. α while internal non-deterministically handling incoming orders;
477. **E** internal non-deterministically offering order output communications to a designated supplier; or
478. **K** internal non-deterministically offering acknowledgment communications to its order management.

13.6.2.3.2 Formalisation

```
value
473. IV: IV_UI × (om_ui,wh_ui,suis):IV_Mer → (IV_WtD×IV_Inventory×IV_TransHist) Unit
473.        in out ch[{om_ui,iv_ui}] in ch[{wh_ui,iv_ui}]
473.        out { ch[{s_ui,iv_ui}] | s_ui:S_UI•s_ui ∈ suis } Unit
473. IV(iv_ui,iv_mer:(om_ui,wh_ui,suis))(iv_wtd,iv_inv,iv_hist) ≡
474.   D    IV.OM_IV_Order(iv_ui,iv_mer:(om_ui,wh_ui,suis))(iv_wtd,iv_inv,iv_hist)
475.   J  ⌈⌉ IV.WH_IV_Ack(iv_ui,iv_mer:(om_ui,wh_ui,suis))(iv_wtd,iv_inv,iv_hist)
476.   α  ⌈⌉ IV.Handle_Input(iv_ui,iv_mer:(om_ui,wh_ui,suis))(iv_wtd,iv_inv,iv_hist)
477.   E  ⌈⌉ IV.IV_S_Order(iv_ui,iv_mer:(om_ui,wh_ui,suis))(iv_wtd,iv_inv,iv_hist)
478.   K  ⌈⌉ IV.IV_OM_Ack(iv_ui,iv_mer:(om_ui,wh_ui,suis))(iv_wtd,iv_inv,iv_hist)
473.   pre: iv_ui ∈ ivuis ∧ om_ui ∈ omuis ∧ wh_ui ∈ whuis
473.     ∧ ∃ r:R • r ∈ obs_RETs(obs_RETa(mkt))∧iv_ui=uid_IV(r)∧om_ui=uid_OM(r)∧wh_ui=uid_WH(r)
```

13.6.2.3.3 Narrative

479. D The **IV.OM_IV_Order** behaviour offers to accept an order [input] communication from its order management, which, when received, that order is put in the inventory 'work-to-do' basket –
480. to eventually be handled.
481. Whereupon the **IV.OM_IV_Order** resumes being the **IV** behaviours.

13.6.2.3.4 Formalisation

type

```
480. Handle_OM_IV_Order :: Prefix × Order
value
479. D IV.OM_IV_order: IV_UI × IV_Mer → (IV_WorkToDo×IV_Inventory×IV_TransHist) Unit
479.         in out ch[{om_ui,iv_ui}] in ch[{wh_ui,iv_ui}]
479.         out { ch[{s_ui,iv_ui}] | s_ui:S_UI•s_ui ∈ suis } Unit
479. D IV.OM_IV_order(iv_ui,iv_mer:(om_ui,wh_ui,suis))(iv_wtd,iv_inv,iv_hist) ≡
479.     let C–D OM_IV_Order(prefix,order) = ch[{om_ui,iv_ui}] ? in
480.     let iv_wtd′ = {Handle_OM_IV_Order(prefix,order)} ∪ iv_wtd in
481.     IV(iv_ui,iv_mer:(om_ui,wh_ui,suis))(iv_wtd′,iv_inv,iv_hist) end end
```

13.6.2.3.5 Narrative

482. J The **IV.WH_IV_Ack** behaviour offers to accept a supply availability acknowledgment [input] communication from its warehouse.
483. When received that acknowledgment is put in the inventory 'work-to-do' basket.
484. Whereupon the **IV.OM_IV_Order** resumes being the **IV** behaviours.

13.6.2.3.6 Formalisation

```
value
475. J IV.WH_IV_Ack: IV_UI × IV_Mer → (IV_Inventory × IV_TransHist) Unit
475.         in out ch[{om_ui,iv_ui}] in ch[{wh_ui,iv_ui}]
475.         out { ch[{s_ui,iv_ui}] | s_ui:S_UI•s_ui ∈ suis } Unit
475. J IV.WH_IV_Ack(iv_ui,iv_mer:(om_ui,wh_ui,suis))(iv_inv,iv_hist) ≡
482.     let I–J WH_IV_ack(prefix,order) = ch[{wh_ui,iv_ui}] ? in
483.     let iv_wtd′ = {} ∪ iv_wtd in
484.     IV(iv_ui,iv_mer:(om_ui,wh_ui,suis))(iv_wtd′,iv_inv,iv_hist) end end
```

13.6.2.3.7 Narrative

485. α If there exists, in the 'work-to-do' basket, a handle_OM_IV_order, cf. Item 480 on the previous page.,
486. then observe that order's product name, pn, quantity, q, price, p and payment reference, ref, and
487. observe that product's entry (its wholesale price, wp, suggested retail price, srp, sales price, sp, a recommended supplier, s_ui, and the quantity at hand in the warehouse stock) in the inventory [catalog].
488. If the order quantity is lower than the warehouse stock for that product,
489. then choose a suitable re-order quantity, q′,
490. concoct an inventory-to-supplier order,
491. add that to, and remove the handle order from the 'work-to-do' basket, and
492. adjust the stock quantity in the inventory catalog,
493. before resuming being the **inventory** behaviour;
494. else update the 'work-to-do' basket with an inventory-to-order management acknowledgment to, and remove the handle order from the 'work-to-do' basket
495. and resume being the **inventory** behaviour.

496. If there does not exists, in the 'work-to-do' basket, a handle_OM_IV_order, then resume being the **inventory** behaviour.

13.6.2.3.8 Formalisation

```
type
480.  Handle_OM_IV_Order :: Prefix × Order
value
476.  [α] IV.Handle_Input: IV_UI × IV_Mer → (IV_WorkToDo × IV_Inventory × IV_TransHist)
476.        in out ch[{om_ui,iv_ui}] in ch[{wh_ui,iv_ui}]
476.        out { ch[{s_ui,iv_ui}] | s_ui:S_UI•s_ui ∈ suis } Unit
476.  [α] IV.Handle_Input(iv_ui,iv_mer:(om_ui,wh_ui,suis))(iv_wtd,iv_inv,iv_hist) ≡
485.    if ∃ ho:Handle_OM_IV_Order(prefix,order) • ho ∈ iv_wtd
485.      then let ho:Handle_OM_IV_Order(prefix,order) • ho ∈ iv_wtd
486.           let (pn,q,p,ref) = ho in axiom pn ∈ dom iv_inv
487.           let (wp,srp,sp,mi,s_ui,stock) = iv_inv(pn) in axiom [ p ∈ {srp,sp} ]
488.           if q < stock
489.             then let q′:Nat • q′>q ∧ ... in
490.                  let iv_s_order = IV_S_Order(prefix,(pn,q′,wp,iv_ref)) in
491.                  let iv_wtd′ = {iv_s_order} ∪ iv_wtd \ {ho},
492.                      iv_inv′ = iv_inv † [pn ↦(wp,srp,sp,mi,s_ui,stock − q)] in
493.                  IV(iv_ui,iv_mer)(iv_wtd′,iv_inv′,iv_hist) end end end
494.             else let iv_wtd′ = {IV_OM_Ack(prefix,order)} ∪ iv_wtd \ {ho} in
495.                  IV(iv_ui,iv_mer)(iv_wtd′,iv_inv,iv_hist) end end
496.      else IV(iv_ui,iv_mer)(iv_wtd,iv_inv,iv_hist) end end end end
```

13.6.2.3.9 Narrative

497. [E] If there exists, in the 'work-to-do' basket, a handle_OM_IV_order,
498. then retrieve that order
499. and remove it from the 'work-to-do' basket while
500. communicating the order, updated with a date-timed prefix, to a designated supplier,
501. and resuming being the **inventory** behaviour.
502. If no handle_OM_IV_order is in the basket, then resume being "an unchanged" **inventory** behaviour.

13.6.2.3.10 Formalisation

```
477.  [E] IV.IV_S_Order: IV_UI × IV_Mer → (IV_WorkToDo × IV_Inventory × IV_TransHist)
473.       in out ch[{om_ui,iv_ui}] in ch[{wh_ui,iv_ui}]
473.       out { ch[{s_ui,iv_ui}] | s_ui:S_UI•s_ui ∈ suis } Unit
477.  [E] IV.IV_S_Order(iv_ui,iv_mer:(om_ui,wh_ui,suis))(iv_wtd,iv_inv,iv_hist) ≡
497.    if ∃ o:IV_S_Order(prefix,order,s_ui) • o ∈ iv_wtd axiom s_ui ∈ suis
498.      then let o:IV_S_Order(prefix,order,s_ui) • o ∈ iv_wtd in
499.           let iv_wtd′ = iv_wtd \ {o}, dati = record_TIME() in
500.           E–F ch[{iv_ui,s_ui}] ! msg:IV_S_Order(((iv_ui,dati),prefix),order,wh_ui) ;
501.           IV(iv_ui,iv_mer)(iv_wtd′,iv_inv,⟨msg⟩⁀iv_hist)
477.           end end
```

```
501.      else IV(iv_ui,iv_mer)(iv_wtd,iv_inv,iv_hist) end
```

13.6.2.3.11 Narrative

503. [K] If there exists, in the 'work-to-do' basket, an IV_OM_Ack(prefix,order),
504. then retrieve that order
505. and remove it from the 'work-to-do' basket while
506. communicating the order, updated with a date-timed prefix, to a designated supplier,
507. and resuming being the **inventory** behaviour.
508. If no handle_OM_IV_order is in the basket, then resume being "an unchanged" **inventory** behaviour

13.6.2.3.12 Formalisation

```
value
478. IV.IV_OM_Ack: IV_UI × IV_Mer → (IV_Inventory × IV_TransHist)
478.         in out ch[{om_ui,iv_ui}] in ch[{wh_ui,iv_ui}]
478.         out { ch[{s_ui,iv_ui}] | s_ui:S_UI•s_ui ∈ suis } Unit
478.   IV.IV_OM_Ack(iv_ui,iv_mer:(om_ui,wh_ui,suis))(iv_inv,iv_hist)
503.    if ∃ a:IV_OM_Ack(prefix,order) • a ∈ iv_wtd
504.      then let a:IV_OM_Ack(prefix,order) • a ∈ iv_wtd in
505.           let iv_wtd′ = iv_wtd \ {a}, dati = record_TIME() in
506.           [K–L] ch[{iv_ui,om_ui}] ! msg:IV_OM_Ack(((iv_ui,dati),prefix),order) ;
507.           IV(iv_ui,iv_mer)(iv_wtd′,iv_inv,⟨msg⟩⁀iv_hist)
477.           end end
508.      else IV(iv_ui,iv_mer)(iv_wtd,iv_inv,iv_hist) end
```

13.6.2.4 Warehouse Behaviour

13.6.2.4.1 Narrative

509. The **WH** (warehouse) behaviour accepts supplies from any supplier, provides supply acknowledgments to its inventory, accepts dispatch order from its order management and provides merchandise to any courier service.
Internal non-deterministically the **WH** behaviour alternates between
510. [H] external non-deterministically accepting deliveries from suppliers,
511. [P] accepting order dispatches from its order management,
512. [H,P] handling deferred [but accepted] inputs,
513. [I] offering acknowledgments of supplies to its inventory, and
514. [Q] delivering merchandise (orders) to any one of a number of designated courier services.

13.6.2.4.2 Formalisation

```
value
509. WH: wh_ui:WH_UI × (om_ui,iv_ui,suis,csuis):WH_Mer →
```

```
509.          (WH_WorkToDo×WH_Store×WH_TransHist)
509.            in { ch[{s_ui,iv_ui}] | s_ui:S_UI•s_ui ∈ suis }
509.            out ch[{wh_ui,iv_ui}]  in  ch[{wh_ui,om_ui}]
509.            out { ch[{wh_ui,c_ui}] | c_ui:C_UI•c_ui ∈ cuis }  Unit
509.  WH(wh_ui,wh_mer:(wh_ui,om_ui,iv_ui,suis,csuis))(wh_wtd,wh_store,wh_hist) ≡
510.  [H]     WH.S_WH_Del(wh_ui,wh_mer)(wh_wtd,wh_store,wh_hist)
511.  [P]   ⊓ WH.OM_WH_Disp(wh_ui,wh_mer)(wh_wtd,wh_store,wh_hist)
512.  [H,P] ⊓ WH.Handle_Input(wh_ui,wh_mer)(wh_wtd,wh_store,wh_hist)
513.  [I]   ⊓ WH.WH_IV_Ack(wh_ui,wh_mer)(wh_wtd,wh_store,wh_hist)
514.  [Q]   ⊓ WH.WH_CS_Deliv(whui,whmer)(wh_wtd,wh_store,wh_hist)
509.    pre: wh_ui ∈ whuis ∧ om_ui ∈ omuis ∧ iv_ui ∈ ivuis
509.      ∧ ∃ r:R • r ∈ obs_RETs(obs_RETa(mkt))∧wh_ui=uid_WH(r)∧om_ui=uid_OM(r)∧iv_ui=uid_IV(r)
```

13.6.2.4.3 Narrative

515. [H] The **WH.S_WH_Del** behaviour external non-deterministically offers to accept a supplier to warehouse delivery message.
516. When received the **WH.S_WH_Del** behaviour deposits this message in its 'work-to-do' basket.
517. It then resumes being the **WH** behaviour.

13.6.2.4.4 Formalisation

```
value
510.  [H] WH.S_WH_Del: wh_ui:WH_UI × (_,_,suis,_):WH_Mer →
510.          (WH_WorkToDo×WH_Store×WH_TransHist)
510.            in { ch[{s_ui,wh_ui}] | s_ui:S_UI•s_ui ∈ suis }  Unit
510.  [H] WH.S_WH_Del(wh_ui,wh_mer:(_,_,suis,_))(wh_wtd,wh_store,wh_hist) ≡
515.     ⌈ let G–H delivery:S_WH_Del(prefix,order,ms) = ch[{s_ui,wh_ui}] ? in
516.       let wh_wtd′ = wh_wtd ∪ {delivery} in
517.       WH(wh_ui,wh_mer)(wh_wtd′,wh_store,wh_hist)
515.       end end | s_ui:S_UI•s_ui ∈ suis ⌉
510.    pre: wh_ui ∈ whuis ∧ ∃ r:R • r ∈ obs_RETs(obs_RETa(mkt))∧wh_ui=uid_WH(r)
```

13.6.2.4.5 Narrative

518. [P] The **WH.OM_WH_Disp** behaviour offers to accept an order management to warehouse [order] dispatch message.
519. When received the **WH.OM_WH_Disp** behaviour deposits this message in its 'work-to-do' basket.
520. It then resumes being the **WH** behaviour.

13.6.2.4.6 Formalisation

```
value
511.  [P] WH.OM_WH_Disp: wh_ui:WH_UI×(om_ui,_,_,_):WH_Mer →
```

```
511.              (WH_WorkToDo×WH_Store×WH_TransHist)
511.                 in ch[{om_ui,wh_ui}] Unit
511. [P] WH.OM_WH_Disp(wh_ui,wh_mer:(om_ui,_,_,_))(wh_wtd,wh_store,wh_hist) ≡
518.    let [N–P] dispatch:OM_WH_Dis(prefix,order) = ch[{om_ui,wh_ui}] ? in
519.    let wh_wtd′ = wh_wtd ∪ {dispatch} in
520.    WH(wh_ui,wh_mer)(wh_wtd′,wh_store,wh_hist)
518.    end end
509.    pre: wh_ui ∈ whuis ∧ om_ui ∈ omuis ∧ iv_ui ∈ ivuis
509.       ∧ ∃ r:R • r ∈ obs_RETs(obs_RETa(mkt))∧wh_ui=uid_WH(r)∧om_ui=uid_OM(r)∧iv_ui=uid_IV(r)
```

13.6.2.4.7 Narrative

521. [H,P] The rôle of the **WH.Handle_Input** behaviour is to service either of the two kinds of inputs received by the warehouse, from suppliers, **S**, and from its order management, **OM**.
522. There are two possible kinds of "deferred" messages.
523. If there is a supplier-to-warehouse, S_WH_Del(prefix,order,ms), delivery message,
524. then that message is "converted" into a warehouse-to-inventory delivery acknowledgment message in, the 'work-to-do' basket,
525. and the **WH.Handle_Input** behaviour reverts to being the **WH** behaviour;
526. else if there is an order management-to-warehouse, OM_WH_Dis(prefix,order), message,
527. then that message is "converted" into a warehouse-to-courier service delivery message, WH_CS_Del(prefix,order), in the 'work-to-do' basket,
528. and the **WH.Handle_Input** behaviour reverts to being the **WH** behaviour;
529. if there are no messages in the basket then the **WH.Handle_Input** behaviour reverts to being the **WH** behaviour.
530.

13.6.2.4.8 Formalisation

```
value
521. [H,P] WH.Handle_Input: wh_ui:WH_UI × (_,iv_ui,_,_):WH_Mer →
521.                    (WH_WorkToDo×WH_Store×WH_TransHist)
521.                       out ch[{wh_ui,iv_ui}] Unit
521. [H,P] WH.Handle_Input(wh_ui,wh_mer:(_,iv_ui,_,_))(wh_wtd,wh_store,wh_hist) ≡
522.     case wh_wtd of
523.       {S_WH_Del(prefix,order,ms)} ∪ wh_wtd′ →
524.          let wh_wtd″ = wh_wtd′ ∪ {WH_IV_Ack(prefix,order,ms)} in
525.          WH(wh_ui,wh_mer)(wh_wtd″,wh_store,wh_hist) end
526.       {OM_WH_Dis(prefix,order)} ∪ wh_wtd′ →
527.          let wh_wtd″ = wh_wtd′ ∪ {WH_CS_Del(prefix,order)} in
528.          WH(wh_ui,wh_mer)(wh_wtd″,wh_store,wh_hist) end
529.       _ → WH(wh_ui,wh_mer)(wh_wtd,wh_store,wh_hist)
522.     end
522.    pre: wh_ui ∈ whuis ∧ iv_ui ∈ ivuis
522.       ∧ ∃ r:R • r ∈ obs_RETs(obs_RETa(mkt))∧wh_ui=uid_WH(r)∧iv_ui=uid_IV(r)
522.    axiom: [ there can only at most be the two kinds of messages as 'cased' in the wtd basket. ]
```

13.6.2.4.9 Narrative

531. [I] If there exists a warehouse-to-inventory [supplier] delivery acknowledgment message in the 'work-to-do' basket,
532. then select and remove that message from the basket,
533. record the current time, and
534. communicate the acknowledgment message to the inventory,
535. and resume being the appropriately updated **WH** behaviour;
536. else resume being the otherwise unchanged **WH** behaviour.

13.6.2.4.10 Formalisation

```
value
513. [I] WH.WH_IV_Ack: wh_ui:WH_UI × (_,iv_ui,_,_):WH_Mer →
513.                 (WH_WorkToDo×WH_Store×WH_TransHist)
513.                     out ch[{wh_ui,iv_ui}] Unit
513. [I] WH.WH_IV_Ack(wh_ui,wh_mer)(wh_wtd,wh_store,wh_hist) ≡
531.   if ∃ a:WH_IV_Ack(prefix,order,ms) • d ∈ om_wtd
532.     then let a:WH_IV_Ack(prefix,order,ms) • a ∈ om_wtd  in
532.          let wh_wtd' = wh_wtr \ {a},
533.              dati = record_TIME() in
534.          [I–J] ch[{wh_ui,iv_ui}] ! ack:WH_IV_Ack(((wh_ui,dati),prefix),order) ;
535.          WH(wh_ui,wh_mer)(wh_wtd',wh_store,⟨ack⟩⌢wh_hist) end end
536.     else  WH(wh_ui,wh_mer)(wh_wtd,wh_store,wh_hist) end
509.   pre: wh_ui ∈ whuis ∧ iv_ui ∈ ivuis
509.     ∧ ∃ r:R • r ∈ obs_RETs(obs_RETa(mkt))∧wh_ui=uid_WH(r)∧iv_ui=uid_IV(r)
```

13.6.2.4.11 Narrative

537. [Q] If there exists a order management to warehouse [supplier] delivery message in the 'work-to-do' basket,
538. then select and remove that message from the basket,
539. record the current time, and
540. communicate the acknowledgment message to the inventory,
541. and resume being the appropriately updated **WH** behaviour;
542. else resume being the otherwise unchanged **WH** behaviour.

13.6.2.4.12 Formalisation

```
value
514. [Q] WH.WH_CS_Deliv: wh_ui:WH_UI × (_,_,_,csuis):WH_Mer →
514.              (WH_WorkToDo×WH_CS_Dire×WH_Store×WH_TransHist)
514.                  out { ch[{wh_ui,c_ui}] | c_ui:C_UI•c_ui ∈ cuis }  Unit
514. [Q] WH.WH_CS_Deliv(wh_ui,wh_mer:(_,_,_,csuis))(wh_wtd,wh_store,wh_hist) ≡
537.   if ∃ a:OM_WH_Disp(prefix,order,cs_ui) • a ∈ om_wtd
538.     then let d:OM_WH_Disp(prefix,order,cs_ui) • a ∈ om_wtd  in
538.          let wh_wtd' = wh_wtr \ {d},
```

```
539.         dati = record_TIME(),
539.         os:M-set • os⊆wh_store ∧ card os = q in
540.       Q–R  ch[ {wh_ui,cs_ui} ] ! ack:WH_CS_Disp(((wh_ui,dati),prefix),order,os) ;
541.       WH(wh_ui,wh_mer)(wh_wtd′,wh_store \ {os},⟨ack⟩⁀wh_hist) end end
542.    else  WH(wh_ui,wh_mer)(wh_wtd,wh_store,wh_hist) end
509.   pre: wh_ui ∈ whuis ∧ ∃ r:R • r ∈ obs_RETs(obs_RETa(mkt))∧wh_ui=uid_WH(r)
```

13.6.2.5 Supplier Behaviour

13.6.2.5.1 Narrative

543. The **S**upplier behaviour internal non-deterministically "alternates" between
544. F accepting orders from any retailers' inventory, and
545. G delivering such orders to retailers' warehouses.

13.6.2.5.2 Formalisation

```
value
543.  S: S_UI × (ivuis,whuis):S_Mer → (S_WorkToDo×S_Products×S_TransHist)
543.        in { ch[ {iv_ui,s_ui} ] | iv_ui:IV_UI•iv_ui ∈ ivuis }
543.        out {  ch[ {s_ui,wh_ui} ] | wh_ui:WH_UI•wh_ui ∈ whuis } Unit
543.  S(s_ui,s_mer:(ivuis,whuis))(s_wtd,s_products,s_hist) ≡
544.    F    S.IV_S_Order(s_ui,s_mer:(ivuis,whuis))(s_wtd,s_products,s_hist)
545.    G  ⊓ S.S_WH_Deliv(s_ui,s_mer:(ivuis,whuis))(s_wtd,s_products,s_hist)
543.   pre: s_ui ∈ suis ∧ ∃ s:S • r ∈ obs_Ss(obs_Sa(mkt))∧sui=uid_UI(r)
```

Please note that we have omitted the "intermediary" behaviour of the **S**upplier *handling* inputs. We suggest that such handling is taken care of directly by the **S.S_WH_Deliv**ery behaviour. Also note that we do not describe payment aspects.

13.6.2.5.3 Narrative

546. The **S.IV_S_Order** behaviour external non-deterministically offers to accept merchandise orders from any retailer's inventory behaviour.
547. Having received such an order it proceeds to record it in its 'work-to-do' basket –
548. whereupon it resumes being the **S** behaviour (with the updated basket).

13.6.2.5.4 Formalisation

```
value
544.  F  S.IV_S_Order(s_ui,s_mer:(ivuis,_))(s_wtd,s_products,s_hist) ≡
546.     ⫿ { let  E–F  IV_S_Order(prefix,order) = ch[ {iv_ui,s_ui} ] ? in
547.         let s_wtd′ = s_wtd ∪ {IV_S_Order(prefix,order)} in
548.         S(s_ui,s_mer)(s_wtd′,s_products,s_hist)
546.       | iv_ui:IV_UI•iv_ui ∈ ivuis  end end }
```

13.6.2.5.5 Narrative

549. If there exists a IV_S_Order(prefix,order) message in the 'work-to-do' basket,
550. then select such a message,
551. examine the order,
552. select the quantified number of merchandise of the ordered product,
553. ascertain the current time,
554. and deliver the message to the warehouse of the requesting retailer;
555. then resume being the **S**upplier behaviour with appropriately updated programmable attributes.
556. If there does not exists a IV_S_Order(prefix,order) message in the 'work-to-do' basket, then revert to being the **S**upplier behaviour.

13.6.2.5.6 Formalisation

```
value
545. [G] S.S_WH_Deliv(s_ui,s_mer:(ivuis,whuis))(s_wtd,s_products,s_hist) ≡
549.     if ∃ h:IV_S_Order(prefix,order,wh_ui) • h ∈ s_wtd
550.       then let h:IV_S_Order(prefix,order,wh_ui) • h ∈ s_wtd in
551.             let (pn,q,cost,payref) = order in
552.             let ms:M-set • ms ⊆ s_products(pn)∧card ms = q,
553.                 dati = record_TIME() in
554.                 [G–H] ch[{s_ui,wh_ui}] ! d:S_WH_Deliv(((s_ui,dati),pref),order,ms) ;
555.             S(s_ui,s_mer)(s_wtd \ {h},s_products†[pn↦s_products(pn) \ ms],⟨d⟩⁀s_hist)
550.             end end end
556.       else S(s_ui,s_mer)(s_wtd,s_products,s_hist) end
```

13.6.2.6 Courier Service Behaviour

13.6.2.6.1 Narrative

557. The **CS**, courier service, behaviour internal non-deterministically alternates between
558. [R] offering to accept a warehouse to [customer directed] courier service delivery of merchandise and
559. [S] offering a courier service to customer delivery.

13.6.2.6.2 Formalisation

```
value
557. CS: CS_UI × CS_Mer → (CS_WorkToDo × CS_TransHist)  Unit
557. CS(cs_ui,csmer:(whis,cuis))(cs_wtd,cs_hist) ≡
558.   [R]   CS.WH_CS_Deliv(cs_ui,csmer:(whis,cuis))(cs_wtd,cs_hist)
559.   [S] ⊓ CS.CS_C_Deliv(cs_ui,csmer:(whis,cuis))(cs_wtd,cs_hist)
557.   pre: csui ∈ csuis
```

13.6.2.6.3 Narrative

560. R The **CS.WH_CS_Deliv** behaviour external non-deterministically offers to accept a customer directed delivery request from any retailer's warehouse.
561. Receiving such a request it updates its 'work-to-do' basket accordingly,
562. and reverts to being the courier service **CS**.

13.6.2.6.4 Formalisation

```
value
558. R CS.WH_CS_Deliv: CS_UI × CS_Mer → (CS_WorkToDo × CS_TransHist) Unit
558. R CS.WH_CS_Deliv(cs_ui,csmer:(whis,_))(cs_wtd,cs_hist) ≡
560.     ⌷ { let Q–R r:WH_CS_Deliv(((wh_ui,dati),prefix),order,os) = ch[{wh_ui,s_ui}] ? in
561.         let cs_wtd′ = cs_wtd ∪ {r} in
562.         CS(cs_ui,cs_mer)(cs_wtd′,cs_hist)
560.       | wh_ui:WH_UI•wh_ui ∈ whuis end end }
558.   pre: cs_ui ∈ csuis
```

563. S If there exists a WH_CS_Del(prefix,order,ms,c_ui) dispatch in the 'work-to-do' basket of a courier service
564. then retrieve this dispatch
565. pass it on to the designated customer
566. and revert to being the courier service, **CS**, behaviour with appropriately updated arguments.
567. Otherwise continue being the **CS** behaviour.

13.6.2.6.5 Formalisation

```
value
559. S CS.CS_C_Deliv: CS_UI × CS_Mer → (CS_WorkToDo × CS_TransHist) Unit
559. S CS.CS_C_Deliv(cs_ui,cs_mer)(cs_wtd,cs_hist) ≡
563.     if ∃ whd:WH_CS_Del(prefix,order,ms,c_ui) • whd ∈ cs_wtd
564.       then let d:WH_CS_Deliv(prefix,order,ms,c_ui) • whd ∈ cs_wtd in
565.             S–T ch[{cs_ui,c_ui}] ! cd:CS_C_Del(((cs_ui,record_TIME),prefix),order,ms) ;
566.             CS(cs_ui,csmer)(cs_wtd \ {d},⟨cd⟩⁀cs_hist) end
567.       else CS(cs_ui,csmer)(cs_wtd,cs_hist) end
564.   pre: cs_ui ∈ csuis
```

13.6.3 System Initialisation

We refer to [55, *Sect. 7.8*].

568. Given a market, cf., mkt Item 351 on page 371, we can "synthesize" an RSL clause that stands for the total behaviour of this market.
    We refer to the system state as "generated" in Sect. **13.3.4** on page 371.
569. The market behaviour is the parallel composition of

570. the distributed parallel compositions of all customers,
571. the distributed parallel compositions of all order managements,
572. the distributed parallel compositions of all inventories,
573. the distributed parallel compositions of all warehouses,
574. the distributed parallel compositions of all suppliers and
575. the distributed parallel compositions of all courier services.

```
value
568. mkt, cs,oms,ivs,whs,ss and css.
570. ‖ {C(uid_C(c),mereo_C(c),attr_C_Catalog(c))(attr_C_Merhandise(c),⟨⟩)|c:C•c∈cs}
569. ‖
571. ‖ {OM(uid_OM(om),mereo_OM(c),attr_OM_ProdSupp(om))(attr_OM_WorkToDo(om),⟨⟩)|om:OM•om∈coms}
569. ‖
572. ‖ {IV(uid_IV(iv),mereo_IV(iv))(attr_IV_WorkToDo(iv),attr_IV_Inventory(iv),⟨⟩)|iv:IV•iv∈ivs}
569. ‖
573. ‖ {WH(uid_WH(wh),mereo_WH(wh))(attr_WH_WorkToDo(wh),attr_WH_Store(wh),⟨⟩)|wh:WH•wh∈whs}
569. ‖
574. ‖ {S(uid_S(s),mereo_S(s))(attr_S_WorkToDo(s),attr_S_Products(s),⟨⟩)|s:S•s∈ss}
569. ‖
575. ‖ {CS(uid_CS(cs),mereo_CS(cs))(attr_CS_WorkToDo(cs),⟨⟩)|cs:CS•cs∈css}
```

13.7 Conclusion

13.7.1 Critique of the 𝔻𝔸&𝔻 Model

I am, today, November 15, 2021: 16:12 , not quite happy with my description.

- It was developed too quickly. I started on this model on Dec. 28, 2020. I was the only one to develop this model, cf. Sect. **13.7.4** on page 401.
- Along the "road" I did not take time to carefully consider the naming of types, values, functions and behaviours.
- Also: the individual definitions of order management, inventory, warehouse, supplier and courier service behaviours (**OM, II, WH, S, CS**) into their , as of Jan. 21, 2021, is/was uneven.
  - In the **C** (customer) behaviour description (Items 434 on page 385.– 443 on page 385.) "all" is expressed in that one behaviour description, whereas in the **OM, IV, WH, S** and **CS** behaviour descriptions the descriptions are decomposed into separate internal non-deterministic behaviours, but not quite consistently.
  - I have yet to check that the mereologies and attributes of parts are consistently used in respective behaviour definitions.
  - And I have yet to check that the indexing of all defined types, sorts, unique identifiers, mereologies, attributes, channel and behaviours is consistent.
- But, on the whole, the model gives a reasonably adequate picture of how a model in the 𝔻𝕊&𝔼/𝔻𝔸&𝔻 style would express the ℍ𝔼ℝ𝔸𝕂𝕃𝕀𝕋 *retailer* "challenge".
- All I can say, not in any defense, is: *"I am too*[143] *old for this game these days !"*

[143] I was born Oct. 4, 1937

13.7.2 Proofs about Models

Models are developed, carefully, and honed, "perpetually, for several reasons. One is to be able to prove properties of the domain being modeled but where these properties are not explicitly stated. We speculate on a few – with more to come !

- *"The sum total of merchandise, in the market as modeled, is constant: no merchandise "arise out of the blue" (for example at suppliers), no merchandise "disappears mysteriously" (for example in warehouses, courier services or at customers)."*
- *"Product quantity_on_hands in a retailer's inventory (catalog) is always less than or equal to that retailer's corresponding quantities_at_hand in its warehouse."*
- With the ideal assumption that suppliers can always deliver requested numbers of any product: *"Customers are eventually delivered their ordered merchandise."*
- Etcetera !

We do not show any such proofs in this technical report.

13.7.3 Comparison of Models

We compare our model to that of [88].

13.7.3.1 "Minor" Discrepancies

- It seems, but this has to be checked, that orders, in the DA&D model, are for any number of one particular merchandise product, whereas the HERAKLIT model allows the mixing of several products and of different quantities of these.
- It also seems that ...

MORE TO COME

13.7.3.2 Use of Diagrams

- Somewhere, in Footnote 136 on page 365, it is said that none of the figures in this report play any rôle in the formal aspects of the 'retailer market' description.
  - That is intended to be so.
  - But is it really true ?
    - When I first worked as an M.Sc. graduated engineer in designing data communications "gear" and computers for IBM (1962–1965, 1969) we all drew diagrams !
    - When I then studied computer science (1965-1968) diagrams of software systems (except for "trivial" program flowcharts) were frowned upon.
    - `Petri nets` (around 1962–1963) are based almost exclusively on two-dimensional diagrams. They are easy to grasp,
    - The diagrams of HERAKLIT are likewise appealing.
  - Figure 13.2 on page 366 of this report is rather crucial, I found, in keeping track, while I was developing the description, of all the various segments of that description – in particular in making sure that the interfaces between behaviours "fitted".
  - I can imagine that some readers will find, especially Fig. 13.2 on page 366 useful when reading the description.

- So, perhaps, diagrams, of the kind that Fig. 13.2 on page 366 represents, ought be "woven" into the 𝔻𝔸&𝔻 analysis & description, into its *principles, techniques* and *tools*.

13.7.3.3 Interleave versus "True" Concurrency

The reader is assumed to be quite familiar with these two kinds of semantic terms and their meaning.

- As such, the ℍ𝔼ℝ𝔸𝕂𝕃𝕀𝕋-based model, appears to be at an advantage – in that it expresses "true concurrency".
- But, before you get all too excited, the 𝔻𝕊&𝔼/𝔻𝔸&𝔻 model, as its behaviours are defined, "do not lack far behind", if-at-all !
  - On one hand you can read the basically CSP clauses as if actions in separate behaviours do indeed occur "truly concurrent".
  - On the other hand, by "splitting" up, as in the behaviour definitions of **C, OM, IV, WH, S** and **CS**, these into separate actions, such as *input, handling*, etc., an as full "measure" of local "true concurrency" seems to be achieved.

13.7.4 Development Management

How to organise the development of domain descriptions such as presented in this report ? Here is some advice.

- First conduct a "full trial run", such as the work behind this report represents.
- Then do "the same thing again", but now, "in earnest".
  - The "full trial run" shall cover, basically, the full domain.
  - But it is allowed to "waver", as do the report you are holding in your hand, between different styles of analysis & description.
  - The aim of the "full trial run" is for the development team to settle on "exactly" the style to be followed.
  - At the same time management can better ascertain the manpower and time to be used for the various segments of the work – such as itemized next.

The below items now hint at the, "in earnest" work to be done after a "full trial run".

- Assemble a group of, in this case, as also 'judged' from the "full trial run", six (more) software engineers cum computing scientists, professionally educated, that is, knowledgeable of such texts as [25, 26, 27, 55].
  The group reads the "full trial run" report.
  `For the present project I estimate 2 six-seven person-weeks.`[144,145,146]
- Together the whole group, including the project manager, analyses and describes the external qualities – cf. Sects. **13.2** and **13.3**.
  `For the present project I estimate 2 six-seven person-weeks.`

[144] This means: 2 weeks of calendar time for 6–7 persons.

[145] In this initial, "for earnest" project step the group settles on computerised tools, e.g. LATEX, RAISE/RSL, etc.

[146] The Dansk Datamatik Center **Ada** project, January 1981 – September 1984 did just that: based on a "full trial run" of six M.Sc. students' 6 months' master thesis work (February – August 1980), [67]. These M.Sc. students background was essentially that of [25, 26], that is, knowledgeable about applicative, imperative, logic and parallel programming, operational (in those days called 'mechanical'), first-order and denotational semantics, etc.

- Based on the decomposition of the domain endurants, one project member is assigned to distinct parts, as here the *customer, order management, inventory, warehouse, supplier* and *courier service* parts.
  `For the present project I estimate 1 six-seven person-weeks.`
- Each of these staff members now take care, in synchrony with all others, of the *unique identifier* section, then the *mereology* section, and the *attributes* section, one-by-one.
  `For the present project I estimate (1+2+3) six-seven person-weeks.`
- Each member, on a rotating basis, receives the work of a colleague, every morning, reviews it, and all meets, say at 11am, every work day, to discuss and debate each others' models, whereupon they spend the afternoon on continuing their section of work.
- When work on *internal qualities* has ended, they all meet for as longs as it takes, half a day – three days (!) – to settle on channels.
  `For the present project I estimate 1 six-seven person-weeks.`
- Thereupon they work on defining behaviour signatures and, when all such signatures are thought finalised, then the behaviour definitions.
  `For the present project I estimate 2 six-seven person-weeks.`
- And so forth.[147]
- Do not forget the daily late morning meetings !

13.7.5 What Next ?

- It is my sincere hope that Messrs Fettke and Reisig will comment on the present report.
- I need to know where I have misunderstood the [intentions of the] HERAKLIT model [88].
- I need to know where my model fails in modeling what [88] achieves.
- etcetera !

[147] That is: I estimate a total calendar time of 4+14 weeks for the (one person) "full trial run" plus the (six to seven person) "in earnest" projects.

Chapter 14

Shipping [Spring/Summer 2007, February–March 2021]

Contents

14.1 Informal Sketches of the Shipping Domain 404
14.1.1 The Purpose of A Domain Model for Shipping 404
14.1.2 A First Sketch 405
14.1.3 A Second Sketch 405
14.1.3.1 Strands of Interacting Sets of Behaviours 406
14.1.3.2 Freight 406
14.1.3.2.1 Freight as Endurants 406
14.1.3.2.2 Freight as Behaviours 406
14.1.3.3 Freight Forwarder Behaviour 407
14.1.3.4 Shipping Line Behaviour 407
14.1.4 Some Comments 408
14.1.4.1 Caveat Concerning Sketches 408
14.1.4.2 The Insufficiency of Narrative Descriptions 409
14.1.4.3 What Do Formal Descriptions Contribute ? 409
14.1.4.4 Limitations of Domain Models 409
14.1.4.5 Families of Domain Models 409
14.1.4.6 There is No “Standard Model” 409
14.2 Endurants: External Qualities 410
14.2.1 Freight 410
14.2.2 Endurant Sorts & Observers 410
14.2.3 Endurant Values 411
14.3 Endurants: Internal Qualities 412
14.3.1 Unique Identifiers 412
14.3.1.1 Unique Identifier Types and Observers 412
14.3.1.2 Domain Unique Identifiers 413
14.3.1.3 An Axiom 414
14.3.1.4 Retrieve Endurant Values 414
14.3.2 Mereologies 414
14.3.2.1 A Shift in Modeling 414
14.3.2.2 Mereology Types and Observers 414
14.3.2.2.1 Harbour Mereology 415
14.3.2.2.2 Vessel Mereology 415
14.3.2.2.3 Shipping Line Mereology 416
14.3.2.2.4 Freight Forwarder Mereology 416
14.3.2.2.5 Freight Mereology 417
14.3.2.2.6 Passenger Mereology 417
14.3.2.2.7 Waterways Mereology 418
14.3.2.2.8 Landmass Mereology 418
14.3.3 Attributes 419
14.3.3.1 Attribute Types and Observers 419
14.3.3.1.1 Freight Forwarder Attributes 419
14.3.3.1.2 Shipping Line Attributes 419
14.3.3.1.3 Vessel Attributes 420
14.3.3.1.4 Harbour Attributes 420

14.3.3.1.5 Freight Attributes 420
14.3.3.2 Attribute Wellformedness 421
14.4 Perdurants 421
14.4.1 Freight as Endurants and as Behaviours 421
14.4.2 Actions, Events and Behaviours 421
14.4.3 Global Freight Variable 421
14.4.4 Channels 422
14.4.5 Behaviours 422
14.4.5.1 Behaviour Signatures 422
14.4.5.1.1 Freight Forwarder Signature 422
14.4.5.1.2 Shipping Line Signature 422
14.4.5.1.3 Vessel Signature 423
14.4.5.1.4 Harbour Signature 423
14.4.5.1.5 Freight Signature 423
14.4.5.2 Behaviour Definitions 423
14.4.5.2.1 Freight Forwarder Definition 423
14.4.5.2.2 Shipping Line Behaviour Definition 427
14.4.5.2.3 Vessel Behaviour Definition 428
14.4.5.2.4 Harbour Behaviour Definition 428
14.4.5.2.5 Freight Behaviour Definition 428

This document reports on an experiment: that of modeling a domain of shipping lines[148]

The purposes of the experiment are (i) to further test the methodology of domain analysis & description as outlined in [55], and (ii) to add yet an, as we think it, interesting domain model to a growing series of such [56].

The report is currently in the process of being written, that is, the domain is still being studied, analysed and tentatively described. Please expect that later versions of this document may have sections that are removed, renumbered and/or rewritten wrt. the present November 15, 2021: 16:12 version.

The author regrets not having had contact to real professionals of the shipping line industry. This is most obvious in our treatment of freight forwarder and shipping line behaviours. Here we used simple reasoning to come up with plausible behaviours. The behaviours that we define should convince the reader that whichever similarly reasonable, but now actual, real behaviours can be likewise defined.

The author hopes, even at his advanced age, today he is 83 years old, to be able, somehow, to learn from such contacts.

14.1 Informal Sketches of the Shipping Domain

We shall, as a necessary element in the analysis of a domain to be rigorously analysed & described, first, and informally, delineate that domain.

This initial step of a full development is typically iterative. One outlines a first sketch. If one is not quite happy with that one either improves on that sketch or, throws it away and, produces another sketch – until "satisfied".

14.1.1 The Purpose of A Domain Model for Shipping

Any undertaking of modeling a specific domain has a purpose. The purpose of the shipping domain model of this report is to understand some of the properties of shipping, say such as those expected by people who have freight transported. What these properties are will evolve as the domain model evolves.

[148] with the Greenland **Royal Arctic Line**, https://www.royalarcticline.com, as a leading inspiration.

14.1.2 A First Sketch

We structure the sketch in itemized points.

- The **name of the domain** is *Shipping*.
- The **overall context of the domain** is that
  - there is a continuous "stretch" of **navigable waterways**, an **ocean**;
    - on which **vessels** can sail;
  - there is a concept of **landmass**es with coasts onto the waterways;
    - with **harbours** at which
    - the **vessels** can dock
    - to **unload** and **load freight** and/or **passengers**.
  - There are **shipping lines** which operate these vessels;
    - with these shipping lines accepting requests for and actual freight and/or passengers to be transported;
  - and there are **freight forwarders** which
    - either act as go-between those who wishes freight or passenger transport,
    - or are those freight "owners", respectively passengers,
    - and requests and services accepted requests, i.e., order, for transport.
- The **closer details of the domain** [of shipping further] involve that
  - harbours have **management** and **staff** (including **stevedores**) – which is ignored in the present domain model;
  - vessels have **staff** (**captains, mates, engineers** and **seamen**) – which is (are) ignored in the present domain model;
  - freight forwarders and shipping lines have **management** and **staff** – whose education, training, hiring, rostering[149], laying-off and pensioning which is (are) ignored in the present domain model;
  - harbours, freight forwarders and shipping lines need financial capital in order to establish, maintain, renew and operate – which is ignored in the present domain model; and that
  - all of these have to operate in the context of local, state and international rules & regulations (i.e., laws) – which is ignored in the present domain model

We justify the omissions as they are common to many [other] domains and thus do not specifically characterise the chosen domain.

14.1.3 A Second Sketch

We assume a notion of **state**. The programmable attributes, typically, of endurants are bases for states. States [thus] have values.

- **Actions** are *intended* phenomena that potentially *changes state* values *instantaneously*.
- **Events** are un-intended phenomena which [thus *surreptitiously*] may instantaneously change a state.
- **Behaviours** are sets of sequences of actions, events and behaviours. Behaviours change states, *one change after another* and several changes potentially "at the same time", i.e., *concurrently*, in *parallel*.

[149] – assignment, per day, to time-slots and places of work

14.1.3.1 Strands of Interacting Sets of Behaviours

We shall focus primarily on the behaviours of two "main players": those of **freight forwarders** and those of **shipping lines**. We shall consider the behaviours of **freight, vessels** and **harbours** to be subsidiary, i.e., subservient, to the main behaviours.

The two sets of behaviours each "operate, i.e., behave, on their own", concurrently, but **interacting.**

Freight forwarders, in an interleaved fashion, on behalf of many customers, using many shipping lines, **place orders**, **accept offers** (or **refusals**), **deliver freight** and **passengers** to vessel sides ('alongside'), **fetch freight** and **passengers** from vessel sides ('alongside') and and **handle** all related **"paper-work"** ('bill-of-ladings') and **finances**.

Shipping lines, also in an interleaved fashion, serving many freight forwarders, co-sailing, possibly, with other shipping lines, **accept** or **reject orders**, **issue offers**, sees to it that vessels **fetch freight** (and **passengers**) from vessel sides ('alongside'), sees to it that vessels **deliver freight** and **passengers** to vessel sides ('alongside'), and **handle** all related **"paper-work"** ('bill-of-ladings') and **finances**.

14.1.3.2 Freight

> *From* `Middle English` *freight, from* `Middle Dutch` *vracht,* `Middle Low German` *vrecht* ("cost of transport"), ultimately from `Proto-Germanic` **fra-* (intensive prefix) + `Proto-Germanic` **aihtiz* ("possession"), from `Proto-Indo-European` **$h_2$eyḱ* ("to possess"), equivalent to `for-` + `aught`. Cognate with `Old High German frēht` ("earnings"), `Old English` *æht* ("owndom"), and a doublet of *fraught* [`https://en.wiktionary.org/wiki/freight`].

14.1.3.2.1 Freight as Endurants

Freight is both a singular and a plural term. So, by freight we shall understand one or more items of discrete endurants or any amount of and liquid endurant. That is, for example, one or more 20 or 40 feet containers may be considered one freight item, and any amount of bunker oil may be considered one freight item.

14.1.3.2.2 Freight as Behaviours

Freight are also considered behaviours: **create**d as both endurants and as behaviours by the freight forwarder at the instant of a freight forwarder's first booking inquiry, and **dismantle**d by the freight forwarder at the completion of that freight's transport.[150]

• • •

We shall further sketch two behaviours. The servicing of freight transports, seen from a freight forwarder and the servicing of freight transports, seen from a shipping line. Each of these behaviours if expressible as the composition of several actions. Were we here to also sketch a vessel's freight transport, then we would have to introduce such **events** as the vessel being delayed due to unforeseen weather conditions, the vessel being ship-wrecked ()

[150] We shall, without loss of generality, only model that freight undergo one vessel transport.

14.1.3.3 Freight Forwarder Behaviour

The freight forwarder behaviour basically consists of the following **freight forwarder actions**.

(i) **[FC]** *Freight Creation:* The freight is "created" ! We do not [have to] define the circumstances of creation.[151]

(ii) **[FB]** *Freight Being Booked:* There is the action of **booking space and time for freight between two harbours**. It is directed at a **shipping company** by a **freight forwarder**. How that freight forwarder came to book at that shipping company is left undefined. The shipping line either says *no thanks, another time, perhaps!"*, or propose a sailing (i.e., a vessel and times of departure and arrival), costs, etc., i.e., a bill-of-lading. The freight forwarder accepts "refusals", and either accepts the proposed bill-of-lading, or does not accept proposals.

(iii) **[FD]** *Freight Delivery:* In due time, if proposed bill-of-lading is accepted, the freight forwarder receives notification from the shipping line that the vessel has arrived at designated harbour of departure and the freight forwarder therefore delivers the freight at that harbour.

(iv) **[FT]** *Freight Transport:* The freight forwarder can now trace the freight transport.

(v) **[FR]** *Freight Return:* In due time, the freight forwarder receives notification from the shipping line that the vessel has arrived at designated harbour of arrival and the freight forwarder therefore fetches the freight at that harbour.

(vi) **[FE]** *Freight Dismantlement:* At some [short] time after the freight has been collected it ceases to exist as freight !

(vii) **[FM]** *Freight Management:* And all the time the freight forwarder manages "paperwork" and finances.

End of that story !

•••

The above sequence of characteristic freight forwarder actions can therefore be interpreted as actions for one particular item of freight with these actions being interleaved with those for other freight items also being handled by a freight forwarder. To distinguish between different freight handlings freight forwarders naturally uses the unique freight identifier obtained in action **[FC]**.

14.1.3.4 Shipping Line Behaviour

The shipping line behaviour basically consists of the following **shipping line actions**.

(i) **[SQ]** *Booking Inquiry:* The shipping line, at any time, accepts inquiries, from freight forwarders, as to freight transport. The inquiries states freight essentials, whether inflammable/explosive, whether a container or otherwise packaged (dimensions, weight, etc.), from and to harbours, desirable shipping dates, etc. The shipping line decides, upon acceptance, whether to respond immediately, or after some processing time, say minutes or hours, to the inquiry.

(ii) **[SQH]** *Query Handling:* The shipping line responds to inquiries either instantly or after some [other] processing time. Either the line can satisfy, i.e., **accepts**, the request and sends the inquiring freight forwarder a transport proposal, tentatively reserves space and time (i.e., vessel) for the subject freight, and then awaits its acceptance or refusal, or the line cannot satisfy, i.e., must **refuse**, the request and sends the inquiring freight forwarder a polite negative response – and "closes" that inquiry.

(iii) **[SR]** *Booking Reaffirmation:* The shipping line, at any time, accepts acceptance of accepted orders. It does so, for example, by reaffirming, to the freight forwarder, the now mutually accepted order, while initiating a **physical order handling** "process", i.e., changes the order from tentative to definite.

[151] Technically the freight forwarder behaviour "spawns" off a henceforth concurrently operating freight behaviour.

(iv) **[SV1]** *Vessel Co-ordination, 1:* The shipping line, at some time thereafter, informs the vessel of its cargo for specific sailings, while assuring itself of that vessel's availability.

(v) **[SH1]** *Harbour Co-ordination, 1:* The shipping line, at some time thereafter, informs designated harbours of its plans for the vessel in question to indeed arrive at, unload and load freight, and depart from that harbour, while ensuring that the harbour in question is indeed prepared for that. [We omit treatment of no or negative response from harbours.]

(vi) **[SFA]** *Freight Acceptance:* At the appointed date and time the shipping line observes, by communication from the vessel that the freight forwarder delivers the designated freight alongside the vessel.

(vii) **[SV2]** *Vessel Co-ordination, 2:* Eventually the shipping line is informed that the vessel departs freight origin harbour.

(viii) **[SV3]** *Vessel Co-ordination, 3:* Eventually the shipping line is informed that the vessel arrives at freight destination harbour.

(ix) **[SH2]** *Harbour Co-ordination, 2:* The shipping line informs that harbour of the imminent arrival of one of its vessels.

(x) **[SF]** *Freight Forwarder Notification:* The shipping line informs the freight forwarder of the arrival of "its" freight.

(xi) **[SFD]** *Freight Delivery:* And the freight forwarder informs the shipping line of its receipt of freight.

(xii) **[SFM]** *Freight Management:* All the while the shipping line keeps track of all the "paperwork" and financial matters, and other freight related matters.

End of that story !

•••

Shipping lines handle much freight. The above sequence of twelve characteristic shipping line actions can therefore be interpreted as actions for one particular item of freight, sometimes, like **[SV1-2-3,SH1-2,SFM]**, merged with those for "similarly" transported freight. with these actions being interleaved with those for other freight items To distinguish between different freight handlings shipping lines naturally uses the unique freight identifier obtained in action **[SQ]**.

•••

As the reader will have observed: The "workhorse" of the described domain is the shipping line – as one should indeed expect it to be !

•••

The stories narrated above are as yet not in their final form. Language, clarification and other improvements will eventually find their way into the above text.

14.1.4 Some Comments

14.1.4.1 Caveat Concerning Sketches

In the informal language sketching of a domain there is, however, a serious problem. One way of illustrating the problem is as follows: Replace all domain specific nouns and verbs with $\alpha,\beta,\gamma,...$, respectively $x,y,z,....$ Now you see the problem: What does the sketch now "describe" ? By using nouns and verbs of a domain that may be known to the reader, and for which the reader may have some understanding, but for which any two readers may usually have different understandings, the readers are being "lured" into a possible trap ! Only a proper narrative description that is

strongly linked to a formal specification – one where the αs, βs, γs, ..., respectively xs, ys, zs, ... are given mathematical meanings – may be satisfactory – provided, of course, that the mathematics is *consistent and relatively complete*[152].

14.1.4.2 The Insufficiency of Narrative Descriptions

Why is it not sufficient with just narrative, i.e., informal, descriptions ? The answer is simple. For the shipping domain, just sketched, it might seem sufficient. But assume that you, the reader of this sketch, come from somewhere where there is no "nearby" notion of waterways, hence of vessels, etc. The fact that our sketches uses many terms from the shipping domain does not make them understandable, in-and-by-themselves. Take another example: You are from some Pacific island. There are no railways there. And you sketch a railway domain. The Pacific islander, really, have no clue as to the meaning of such terms as a railway track, a railway switch, etc. So the meaning of terms – such as presented in the above sketches – are far from clear. The danger in this is that these terms many be understood to possess properties that were not sketched.

14.1.4.3 What Do Formal Descriptions Contribute ?

Conjoining a narrative, informal text with formal, mathematical text is meant to "fill-the-gap": to allow the user of domain model to asset properties beyond what has been explicitly described and, based on such formalisations, reason that a postulated domain property holds, or does not hold.

14.1.4.4 Limitations of Domain Models

But we cannot possibly, neither informally narrate nor formally specify a "complete domain", that is, "all" domain properties. Our domain descriptions must necessarily focus on some properties while ignoring other properties. That is, every domain model has a purpose.

14.1.4.5 Families of Domain Models

So, for the domain of shipping, we can thus expect a set of domain models. One like the one presented in this report. Another which focus of vessels: their loading and unloading. Yet another which focus on vessel navigation: on the ocean and into and out from harbours. Etetera.

14.1.4.6 There is No "Standard Model"

Just like for physics, there is not standard model. But, as for physics, there is now, with [55], a "standard" way of developing and presenting domain models. With Newton's Classical Mechanics[153] described in terms of differential equations etc. there is a "standard" approach to analysing & describing mechanics (etc.). For every heretofore not described classical mechanics domain problem the physicists and engineers now know how to tack the analysis & description of that domain. Similarly for every human-assisted discrete dynamics and primarily artifact "populated" domain

[152] – where 'consistent and relative complete' are well-defined notion of mathematical logic

[153] Other contributors to the formal description of Classical Mechanics were Gottfried Wilhelm Leibniz, Joseph-Louis Lagrange, Leonard Euler, etc.

the computer scientists and software engineers now know how to tack the analysis & description of that domain.

14.2 Endurants: External Qualities

14.2.1 Freight

Although the notion of 'freight' is, indeed, a core concept of this report, it will not *"play center stage'*.

14.2.2 Endurant Sorts & Observers

576. We shall consider an **aggregate** of **shipping** in the context of
    a. an **aggregate** of *navigable* **waterways**, i.e., an ocean, rivers and canals[154] with identification of harbours;
    b. an **aggregate** of **land masses**, i.e., continents and islands, small and large;
    c. an **aggregate** of thus identified **harbours**;
    d. an **aggregate** of **vessels** that can carry freight and/or passengers;
    e. an **aggregate** of **shipping lines** – which commands (owns or operate) these vessels;
    f. an **aggregate** of **freight forwarders**[155];
    g. an **aggregate** of **freight**; and
    h. an **aggregate** of **passengers**.
577. The aggregate of harbours is here seen as a set of harbours –
578. with harbours to be further defined.
579. The aggregate of vessels is here seen as a set of vessels –
580. with vessels to be further defined.
581. The aggregate of shipping lines is here seen as a set of shipping lines –
582. with shipping lines to be further defined.
583. The aggregate of freight forwarders is here seen as a set of freight forwarders –
584. with freight forwarders to be further defined.
585. The aggregate of freight is here Sean's as a set of freight –
586. with freight to be further defined.
587. The aggregate of passengers is here seen as a set of passengers –
588. with passenger to be further defined.

The waterways and land masses are here further undefined. Harbours, vessels, shipping lines, freight forwarders, freight and passengers will be further defined below.

---

[154] There are but two oceans. [We do not exclude the *Caspian Sea*. Our model covers both that and "the other" ocean, as the only two !] The "other", the larger ocean is, for pragmatic reasons "divided" up into separately named "oceans": the *Atlantics*, North and South, the *Pacific*, the *Indian*, the *Arabian Sea*, the *Barents Sea*, the *Arctic Sea*, the *Anarctic Sea* (*Southern Ocean, Austral Ocean*), etc. Basically two canals provide short-cuts between two otherwise disperse areas of that one ocean: the *Suez* and the *Panama*.

[155] The borderline between freight forwarders and shipping lines is fuzzy. Some shipping lines offer freight forwarding: the logistics of moving freight between end-customer and vessel, etc.

type
576. S
576a. WV
576b. LM
576c. AH
576d. AV
576e. ASL
576f. AFF
576g. AF
576h. AP
577. Hs = H-**set**
578. H
579. Vs = V-**set**
580. V
581. SLs = SC-**set**
582. SL
583. FFs = FF-**set**
584. FF
585. Fs = F-**set**
586. F
587. Ps = P-**set**
588. P
value
576a. obs_WV: S → WV
576b. obs_LM: S → LM
576c. obs_AH: S → AH
576d. obs_AV: S → AV
576e. obs_ASL: S → ASC
576f. obs_AFF: S → AFF
576g. obs_AF: S → AF
576h. obs_AP: SS → AP
577. obs_Hs: AH → Hs
579. obs_Vs: AV → Vs
581. obs_SLs: ASL → SL-**set**
583. obs_FFs: AFF → FF-**set**
585. obs_Fs: AF → F-**set**
587. obs_Ps: AP → P-**set**

The waterways with its harbours define an in[de]finite set of [circular] routes that can be sailed by the vessels. There are vessels other than those owned or commanded by the company. The company is also characterised by a definite set of routes sailed/serviced by its vessels. All this will be clear as we proceed.

14.2.3 Endurant Values

From an **aggregate** of **shipping** one can extract all its subsidiary endurants – starting with that aggregate:

589. the **aggregate** of **shipping**,
   a. its **aggregate** of **waterways**,
   b. its **aggregate** of **land masses**,
   c. its **aggregate** of **harbours**,
   d. its **aggregate** of **vessels**,
   e. its **aggregate** of **shipping lines**,
   f. its **aggregate** of **freight forwarders**,
   g. its **aggregate** of **freight** and
   h. its **aggregate** of **passengers**;

   and their
590. **set** of **harbours**,
591. **set** of **vessels**,
592. **set** of **shipping lines**,
593. **set** of **freight forwarders**,
594. **set** of **freight**,
595. **set** of **passengers**,
596. **harbours**,
597. **vessels**,
598. **shipping lines**,
599. **freight forwarders**,
600. **freight** and
601. **passengers**.

value
589. s_e:SL [ι 576, π 410]
589a. wv_e:WV = obs_WV(s_e) [ι 576a, π 410]
589b. lm_e:LM = obs_LM(s_e) [ι 576b, π 410]
589c. ah_e:AH = obs_AH(s_e) [ι 576c, π 410]
589d. av_e:AV = obs_AV(s_e) [ι 576d, π 410]
589e. asl_e:ASL = obs_ASL(s_e) [ι 576e, π 410]

589f. aff_e:AFF = obs_AFF(s_e) [ι 576f, π 410]
589g. af_e:AF = obs_AF(s_e) [ι 576g, π 410]
589h. ap_e:AP = obs_AP(s_e) [ι 576h, π 410]
590. hs_e:Hs = obs_Hs(ah_e) [ι 577, π 410]
591. vs_e:Vs = obs_Vs(av_e) [ι 579, π 410]
592. sls_e:SLs = obs_SLs(asl_e) [ι 581, π 410]
593. ffs_e:FFs = obs_FFs(afc_e) [ι 583, π 410]
594. fs_e:Fs = obs_Fs(af_e) [ι 585, π 410]
595. ps_e:Ps = obs_Ps(ap_e) [ι 587, π 410]
596. h_es:H_UI-**set** = {h|h:H•h ∈ obs_Hs(ah_e)} [ι 578, π 410]
597. v_es:V_UI-**set** = {v|v:V•v ∈ obs_Vs(av_e)} [ι 580, π 410]
598. sl_es:SC_UI-**set** = {sl|sl:SL•sl ∈ obs_SLs(sls_e)} [ι 582, π 410]
599. ff_es:FF_UI-**set** = {ff|ff:FF•ff ∈ obs_FF(fcs_e)} [ι 584, π 410]
600. f_es:F_UI-**set** = {f|f:F•f ∈ obs_Fs(fs_e)} [ι 586, π 410]
601. p_es:P_UI-**set** = {p|p:P•p ∈ obs_Ps(ps_e)} [ι 589, π 411]

602. We can define the set of all endurants.

value
602. all_ends = $\{s_e\}\cup\{wv_e\}\cup\{lm_e\}\cup\{ah_e\}\cup\{av_e\}\cup\{asl_e\}\cup\{aff_e\}\cup\{ap_e\}$
602. $\cup hs_e \cup vs_e \cup sls_e \cup ffs_e \cup ff_e \cup ps_e \cup h_es \cup v_es \cup sl_es \cup ff_es \cup f_es \cup p_es$

14.3 Endurants: Internal Qualities

14.3.1 Unique Identifiers

14.3.1.1 Unique Identifier Types and Observers

We can associate unique identifiers with:

603. The **aggregate** of **shipping**;
604. the **aggregate** of **waterways**;
605. the **aggregate** of **land masses**;
606. the **aggregate** of **harbours**;
607. the **aggregate** of **vessels**;
608. the **aggregate** of **shipping lines**
609. the **aggregate** of **freight forwarders**
610. the **aggregate** of **freight**
611. the **aggregate** of **passengers**
612. the **set** of **harbours**;
613. each **individual harbour**;
614. the **set** of **vessels**;
615. each **individual vessel**;
616. the **set** of **shipping lines**;
617. each **individual shipping line**;
618. the **set** of **freight forwarders**;
619. each **individual freight forwarder**;
620. the **set** of **freight**;
621. each **individual freight**;
622. the **set** of **passengers**;
623. each **individual passenger**;

type
603. S_UI [ι 576, π 410]
604. WV_UI [ι 576a, π 410]
605. LM_UI [ι 576b, π 410]
606. AH_UI [ι 576c, π 410]
607. AV_UI [ι 576d, π 410]
608. ASC_UI [ι 576e, π 410]
609. AFF_UI [ι 576f, π 410]
610. AF_UI [ι 576g, π 410]
611. AP_UI [ι 576h, π 410]
612. Hs_UI [ι 577, π 410]
613. H_UI [ι 578, π 410]

614. Vs_UI [ι 579, π 410]
615. V_UI [ι 580, π 410]
616. SCs_UI [ι 581, π 410]
617. SC_UI [ι 582, π 410]
618. FFs_UI [ι 583, π 410]
619. FF_UI [ι 584, π 410]
620. Fs_UI [ι 585, π 410]
621. F_UI [ι 586, π 410]
622. Ps_UI [ι 587, π 410]
623. P_UI [ι 589, π 411]

value

603. uid_S: S → S_UI
604. uid_WV: WV → WV_UI
605. uid_LM: LM → LM_UI
606. uid_AH: AH → AH_UI
607. uid_AV: AV → AV_UI
608. uid_ASL: ASC → ASC_UI
609. uid_AFF: AFF→ AFF_UI
610. uid_AF: AF → AF_UI
611. uid_AP: AP: → AP_UI
612. uid_Hs: Hs → Hs_UI
613. uid_H: H → H_UI
614. uid_Vs: Vs → Vs_UI
615. uid_V: V → V_UI
616. uid_SLs: SLs → SLs_UI
617. uid_SL: SL → SL_UI
618. uid_FFs: FFs → FFs_UI
619. uid_FF: FF → FC_UI
620. uid_Fs: Fs → Fs_UI
621. uid_F: F → F_UI
622. uid_Ps: Ps → Ps_UI
623. uid_P: P → P_UI

14.3.1.2 Domain Unique Identifiers

From an **aggregate** of **shipping lines** one can extract all the unique identifiers of its subsidiary endurants – staring with that aggregate:

624. the **aggregate** of **shipping**,
   a. its **aggregate** of **waterways**,
   b. its **aggregate** of **land masses**,
   c. its **aggregate** of **harbours**,
   d. its **aggregate** of **vessels**,
   e. its **aggregate** of **shipping lines**,
   f. its **aggregate** of **freight forwarders**,
   g. its **aggregate** of **freight** and
   h. its **aggregate** of **passengers**;

   and their
625. **set** of **harbours**,
626. **set** of **vessels**,
627. **set** of **shipping lines**,
628. **set** of **freight forwarders**,
629. **set** of **freight**,
630. **set** of **passengers**,
631. **harbours**,
632. **vessels**,
633. **shipping lines**,
634. **freight forwarders**,
635. **freight** and
636. **passengers**.

value

624. s_{ui}:S_UI = uid_S(s_e) [ι 576, π 410]
624a. wv_{ui}:WV_UI = uid_WV(obs_WV(s_e)) [ι 576a, π 410]
624b. lm_{ui}:LM_UI = uid_LM(obs_LM(s_e)) [ι 576b, π 410]
624c. ah_{ui}:AH_UI = uid_AH(obs_AH(s_e)) [ι 576c, π 410]
624d. av_{ui}:AV_UI = uid_AV(obs_AV(s_e)) [ι 576d, π 410]
624e. asl_{ui}:ASL_UI = uid_ASC(obs_ASL(s_e)) [ι 576e, π 410]
624f. aff_{ui}:AFF_UI = uid_AFF(obs_AFF(s_e)) [ι 576f, π 410]
624g. af_{ui}:AF_UI = uid_AF(obs_AF(s_e)) [ι 576g, π 410]
624h. ap_{ui}:AP_UI = uid_AP(obs_AP(s_e)) [ι 576h, π 410]
625. hs_{ui}:Hs_UI = uid_Hs(obs_Hs(ah_e)) [ι 577, π 410]
626. vs_{ui}:Vs_UI = uid_Vs(obs_Vs(av_e)) [ι 579, π 410]
627. sls_{ui}:SLs_UI = uid_SLs(obs_SLs(asl_e)) [ι 581, π 410]
628. ffs_{ui}:FFs_UI = uid_FFs(obs_FFs(afc_e)) [ι 583, π 410]
629. fs_{ui}:Fs_UI = uid_Fs(obs_Fs(af_e)) [ι 585, π 410]
630. ps_{ui}:Ps_UI = uid_Ps(obs_Ps(ap_e)) [ι 587, π 410]

631. $h_{ui}s$:H_UI-**set** = {uid_H(h)|h:H•h ∈ obs_Hs(ah_e)} [ι 578, π 410]
632. $v_{ui}s$:V_UI-**set** = {uid_V(v)|v:V•v ∈ obs_Vs(av_e)} [ι 580, π 410]
633. $sc_{ui}s$:SL_UI-**set** = {uid_SL(sl)|sl:SL•sl ∈ obs_SLs(sls_e)} [ι 582, π 410]
634. $ff_{ui}s$:FF_UI-**set** = {uid_FF(ff)|ff:FF•ff ∈ obs_FF(ffs_e)} [ι 584, π 410]
635. $f_{ui}s$:F_UI-**set** = {uid_F(f)|f:F•f ∈ obs_Fs(fs_e)} [ι 586, π 410]
636. $p_{ui}s$:P_UI-**set** = {uid_P(p)|p:P•p ∈ obs_Ps(ps_e)} [ι 589, π 411]

637. We can define the set of all endurant identifiers.

value
637. all_uids = $\{s_{ui}\}\cup\{wv_{ui}\}\cup\{lm_{ui}\}\cup\{ah_{ui}\}\cup\{av_{ui}\}\cup\{asc_{ui}\}\cup\{aff_{ui}\}\cup\{ap_{ui}\}$
637. $\cup hs_{ui}\cup vs_{ui}\cup scs_{ui}\cup ffs_{ui}\cup fs_{ui}\cup ps_{ui}\cup h_s s\cup v_{ui}s\cup sc_{ui}s\cup ff_{ui}s\cup f_{ui}s\cup p_{ui}s$

14.3.1.3 An Axiom

638. Endurants are uniquely identified.

axiom
638. □ **card** all_ends = **card** all_uids

The **always** operator, □, expresses that **card** all_ends=**card** all_uids holds at any time.

14.3.1.4 Retrieve Endurant Values

639. Given a unique identifier, ui, in all_uids and given the set of all endurants all_ends we can retrieve the endurant, e of identifier ui.

value
639. get_E: UI → E
639. get_E(ui) ≡ **let** e:E • e ∈ all_ends ⇒ uid_E(e)=ui **in** e **end**

14.3.2 Mereologies

14.3.2.1 A Shift in Modeling

Till now we have modeled the shipping line domain considering all its endurants to be non-structures (cf. [55, *Sects. 4.8 and 4.10*]). From now on we shall consider all aggregates and sets of endurants as structures. This means that we can dismiss our modeling of the unique identifiers for all aggregates and set of endurants void and nil. Thus we shall only model the mereology of what we basically treat as atomic endurants: *freight forwarders*, *shipping lines*, *vessels*, *harbours*, *freight* and *passengers*.

14.3.2.2 Mereology Types and Observers

The **mereology** that we shall promote emphasises both **topological** and **conceptual** properties of shipping line systems. They express topological properties when mandating unique identifiers

of spatially close/related endurants, And they express conceptual properties when mandating unique identifiers of endurants with which shipping lines "do business" ! Further topological and conceptual properties of shipping line systems will be expressed in Sect. **14.3.3** where we treat **attributes** of shipping line systems.

14.3.2.2.1 Harbour Mereology

640. Harbour mereologies are
   - the non-empty set of unique identifiers of vessels that may use the harbour,
   - the pair of two possibly empty sets of unique identifiers of freight: one identifying freight to be loaded (todo), the other having been unloaded (done),
   - the unique identifier of the waterways and the
   - the unique identifier of the landmass.

type
640. H_Mer = V_UI-**set** × (todo:F_UI-**set**×done:F_UI-**set**) × WV_UI × LM_UI
value
640. mereo_H: H → H_Mer

641. The well-formedness of a harbour mereology entails
   - that its set of vessel identifiers is non-empty and included in the set of all vessel identifiers,
   - the "to do" and the "done" freight does not "overlap" and are a subset of all freight.
   - that its waterways identifier is that of the known waterway[s], and
   - that its landmass identifier is that of the known landmass.

value
641. wf_H_Mer: H_Mer → Book
641. wf_H_Mer(vuis,(todo,done),wvui,lmui) ≡
641. {}≠ vuis ⊆ $v_{ui}s$ [ι 632, π 413]
641. ∧ todo ∩ done = {} ∧ todo∪done⊆$f_{ui}s$ [ι 635, π 413]
641. ∧ wvui = wv_{ui} [ι 624a, π 413]
641. ∧ lmui = lm_{ui} [ι 624b, π 413]

14.3.2.2.2 Vessel Mereology

642. Vessel mereologies are
   - the non-empty set of unique identifiers of harbours that it may use,
   - the non-empty set of unique identifiers of shipping lines for which it sails, i.e., which share an agreement to operate that vessel, and
   - the unique identifier of the waterways.

type
642. V_Mer = H_UI-**set** × SL_UI-**set** × WV_UI
value
642. mereo_V: V → V_Mer

643. The well-formedness of a vessel mereology entails

- that its set of harbour identifiers is non-empty and included in the set of all harbour identifiers,
- that its set of shipping line identifiers is non-empty and included in the set of all shipping line identifiers,
- and that its waterways identifier is that of the known waterways.

643. wf_V_Mer: V_Mer → **Bool**
643. wf_V_Mer(huis,scuis,wvui) ≡
643. {}≠huis⊆$h_{ui}s$ [ι 631, π 413]
643. ∧ {}≠ scuis=$sc_{ui}s$ [ι 633, π 413]
643. ∧ wvui=wv_{ui} [ι 624a, π 413]

14.3.2.2.3 Shipping Line Mereology

644. Shipping line mereologies are
   - the non-empty set of unique identifiers of vessels that it operates,
   - the non-empty set of unique identifiers of freight forwarders which it services and
   - the non-empty set of identifiers of harbours that it uses,

type
644. SL_Mer = V_UI-**set** × FF_UI-**set** × H_UI-**set**
value
644. mereo_SL: SL → SL_Mer

645. The well-formedness of a shipping line mereology entails
   - that its set of vessel identifiers is non-empty and included in the set of all vessel identifiers,
   - that its set of freight forwarder identifiers is non-empty and included in the set of all freight forwarder identifiers, and that its set of harbour identifiers is non-empty and included in the set of all harbour identifiers.

value
645. wf_SC_Mer: SC_Mer → **Bool**
645. wf_SC_Mer(vuis,fcuis,huis) ≡
645. {}≠vuis⊆$v_{ui}s$
645. ∧ {}≠ffuis⊆$ff_{ui}s$
645. ∧ {}≠huis⊆$h_{ui}s$ [ι 632, π 413, ι 634, π 413]

Two or more shipping lines may **co-sail** one or more vessels[156].

14.3.2.2.4 Freight Forwarder Mereology

646. Freight forwarder mereologies are
   - the non-empty set of unique identifiers of shipping lines that it uses,
   - the non-empty set of unique identifiers of harbours to which it delivers and from which it fetches freight, and the possibly empty set of unique identifiers of freight with which it is involved.

[156] We shall not model the specifics, i.e., details of co-sailing.

type
646. FF_Mer = SL_UI-**set** × H_UI-**set** × F_UI-**set**
value
646. mereo_FF: FF → FF_Mer

647. The well-formedness of a freight forwarder mereology entails

- the non-empty set of unique identifiers of known shipping lines that it uses and
- the non-empty set of unique identifiers of known harbours

value
647. wf_FF_Mer: FF_Mer → **Bool**
647. wf_FF_Mer(sluis,huis,_) ≡
647. {}≠sluis⊆$sl_{ui}s$ [ι 627, π 413]
647. ∧ {}≠huis⊆$h_{ui}s$ [ι 631, π 413]

14.3.2.2.5 Freight Mereology

648. Freight mereologies are

- the unique identifier of the freight forwarder,
- the unique identifier of the shipping line which is intended to ship, or which ships that freight, and
- the pair of unique identifiers of the two harbour involved in the freight transport.

type
648. F_Mer = FF_UI × SC_UI × (H_UI×H_UI)
value
648. mereo_F: F → F_Mer

649. The well-formedness of a freight mereology entails

- that the freight forwarder identifier is known,
- that the shipping line identifier is known and
- that the two known harbours are different.

649. is_wf_F_Mer: F_Mer → Book
649. is_wf_F_Mer(ffui,scui,(fhui,thui)) ≡
649. ffui∈$ff_{ui}s$
649. ∧ scui∈$sc_{ui}s$
649. ∧ fhui≠thui ∧ {fhui,thui}⊆$h_{ui}s$

14.3.2.2.6 Passenger Mereology

650. Passenger mereologies are

- the identifier of the vessels with which they have traveled, are traveling or intend to travel, and
- the unique identifier of the shipping lines with whom they have travel-led, are traveling or intend to travel.

```
type
650. P_Mer = V_UI-set × SC_UI-set
value
650. mereo_P: P → P_Mer
```

651. The well-formedness of a passenger mereology entails
   - that the set of vessel identifiers is known,
   - that the set of shipping line identifiers is known, and
   - that the shipping lines are indeed operating the identified vessels.

```
value
651. wf_P_Mer: P_Mer → Bool
651. wf_P_Mer(vuis,scuis) ≡
651.   vuis⊆v_uis ∧ scuis⊆sc_uis ∧ [ι 632, π 413, ι 633, π 413]
651.   ∀ v_ui:V_UI • v_ui ∈ vuis, ∃ sc_ui:SC_UI • sc_ui ∈ scuis ⇒
651.     let sc = get_part(sc_ui) in let (vuis′,_) = mereo_SC(sc) in v_ui ∈ vuis′ end end
```

14.3.2.2.7 Waterways Mereology

652.
653.
654.
655.

```
type
652.
653.
654.
655.
value
652.
653.
654.
655.
```

14.3.2.2.8 Landmass Mereology

656.
657.
658.
659.

```
type
656.
657.
658.
659.
value
```

656.
657.
658.
659.

14.3.3 Attributes

14.3.3.1 Attribute Types and Observers

We shall illustrate but a very few attributes. Those we choose to illustrate appear to be the ones most relevant for the specific examples of *freight forwarder, shipping line, vessel, harbour* and *freight behaviours*.

14.3.3.1.1 Freight Forwarder Attributes

660. For any one specific freight, the freight forwarder, undergoes a sequence of states. These are sketched in Sect. **14.1.3.3** on page 407. FFHΣ models the set of state names for these.
661. Freight forwarder history is a freight identifier indexed, reverse-ordered chronological sequence of freight state labeled freight information.
662. We leave FFInfo further undefined,

type
660. FFHΣ = ″FC″ | ″FBB″ | ″FB″ | ″FD″ | ″FT″ | ″FR″ | ″FE″ | ″FM″
661. FFHist = F_UI $\overrightarrow{m}$ (𝕋𝕀𝕄𝔼 × FFHΣ × FFInfo)*
661. FFInfo = ...
value
661. attr_FFHist: FF → FFHist

14.3.3.1.2 Shipping Line Attributes

663.
664.
665.
666.

type
663.
664.
665.
666.
value
663.
664.
665.
666.

14.3.3.1.3 Vessel Attributes

667.
668.
669.
670.

type
667.
668.
669.
670.
value
667.
668.
669.
670.

14.3.3.1.4 Harbour Attributes

671.
672.
673.
674.

type
671.
672.
673.
674.
value
671.
672.
673.
674.

14.3.3.1.5 Freight Attributes

675.
676.
677.
678.

type
675.
676.
677.
678.
value

675.
676.
677.
678.

14.3.3.2 Attribute Wellformedness

14.4 Perdurants

By the **transcendental deductions** introduced in [55, *Chapter 6*] we now interpret some endurant parts as behaviours. A behaviour is a set of sequences of actions, events and behaviours. Behaviours interact, here expressed in the style of CSP [99, 100, 101, *C.A.R. Hoare*] as embedded in RSL [92].

14.4.1 Freight as Endurants and as Behaviours

The central entity of the shipping line domain is that of **freight**. Freight have, so far, been considered as atomic endurants. We shall now transcendentally deduce freight into behaviours. There is a dynamically varying number of uniquely identified freight. We suggest to model freight as follows: Freight is created by the freight forwarder. At the moment of such creation the freight "receives" its, i.e., a unique identifier, one that has not been used before, and one that will never be used, in the creation of other freight, again. Once a freight has completed a full transport as directed by the freight forwarder and carried out by a shipping line and one of its vessels, that freight ceases to be a freight, that is, as an endurants and as a behaviour. Its unique identifier will never be the identifier of other freight.

14.4.2 Actions, Events and Behaviours

14.4.3 Global Freight Variable

Freight occurs, appears, and freight disappears. In this model we assume a fixed number of freight forwarders, shipping lines, vessels and harbours[157]. But we must model a varying number of freight. We shall, for simplicity, and without loss of generality, assume that freight becomes so when in the care of freight forwarders, and that freight ceases to be freight, i.e., to exist, one it has been transported.

Although we shall model freight as behaviours we shall introduce, as a technicality,

679. a global variable freight_uids which is initialised to an empty set of unique freight identifiers.

At any time it contains the set of all unique identifiers of freight which have been created as freight, When freight ceases to exist that freight's unique identifier is not deleted from `freight_uids`.

variable
679. freight_uids:F_UI-**set** := {}
value
680. get_F_UI: **Unit** → F_UI

[157] We also assume fixed waterways and land masses.

```
680.  get_F_UI() ≡
680.    let f_ui:F_UI • f_UI ∉ freight_uids in
680.    freight_uids := freight_uids ∪ {f_ui};
680.    f_ui end
```

680. get_F_UI is a value-returning action.
   - It applies to the global state and returns a "new, hitherto unused" unique freight identifier
   - while updating the global state variable freight_uids with that identifier.

14.4.4 Channels

In order for CSP-modeled behaviours to **interact**, they must **communicate**, and they do so over the medium of, as here, **channels**.

We shall name the full ensemble of channels over which any of the *shipping company, freight forwarder, harbour* and *harbour* behaviours communicate

- **channel** ch[{ui_i,ui_j}]: MSG

where indices ui_i and ui_j are unique identifiers of these behaviours – cum endurant parts, and where MSG is the **type** of the communicated value.

14.4.5 Behaviours

14.4.5.1 Behaviour Signatures

14.4.5.1.1 Freight Forwarder Signature

681. We introduce the notion of "the making of a freight behaviour skeleton" NewF:
   - either there is not such skeleton, "nil",
   - or there are the elements that make up a freight endurant: a unique freight identifier, a freight mereology and the static attributes of a freight. What they are is really of no consequence. The programmable attribute only becomes relevant as soon as the freight endurant, and hence the freight behaviour is created.

682.

```
type
681.  NewF = ″nil″ | F_UI × F_Mer × F_Stat
value
682.  ff: ffui:FF_UI × (sluis,vuis,fuis):FF_Mer × ffstat:FF_Stat → ffprgr:FF_Prgr
682.      → { ch[ {ffui,ui} ] | ui:SL_UI|F_UI•slui∈sluis∪vuis∪fuis }  Unit
```

14.4.5.1.2 Shipping Line Signature

683.

value
683. sl: slui:SL_UI × (vuis,ffuis,huis):SL_Mer × slstat:SL_Stat → slprgr:SL_Prgr
683. → { ch[{slui,ui}] | ui:FF_UI|V_UI|H_UI•ui∈$ffuis \cup huis$ } **Unit**

14.4.5.1.3 Vessel Signature

14.4.5.1.4 Harbour Signature

14.4.5.1.5 Freight Signature

14.4.5.2 Behaviour Definitions

14.4.5.2.1 Freight Forwarder Definition

681. We have introduced, cf. Item 681 on the facing page, the notion of "the making of a freight behaviour skeleton" NewF. To repeat:
   - either there is not such skeleton, `"nil"`,
   - or there are the elements that make up a freight endurant: a unique freight identifier, a freight mereology and the static attributes of a freight, What they are is really of no consequence. The programmable attribute only becomes relevant as soon as the freight endurant, and hence the freight behaviour is created.

684. The *freight forwarder* behaviour may
685. [FC] non-deterministically internally, ⌈⌉, choose to [somehow] accept an item of freight, ..., as expressed in the ffc behaviour, and, likewise non-deterministically internally, decide to "convert" the skeleton into a behaviour.

685a.–685d. Non-deterministically internally the freight forwarder behaviour chooses among the former alternative behaviour, ffc, or the following specific freight related alternatives.

   a. [FB] The freight forwarder communicates a booking order to a shipping line. The shipping line either accepts this booking with a proposed bill-of-lading, or declines it. The freight forwarder must accept declined bookings and must either accept or decline a proposed bill-of-lading.
   We assume that the time elapsed between the freight forwarder communicating its booking and the shipping line responding to this booking is such that the booking and its response can be modeled as a single behaviour composed from two CSP output/input actions.
   [ffb stands for '*f*reight *f*orwarder *b*ooking'.]
   b. [FD] The freight forwarder is informed by the shipping line that the designated vessel is ready to accept the freight for transport.
   We assume that the time elapsed between the freight forwarder receiving this alert and the freight forwarder being able to respond is such that the alert and its response can realistically be modeled as a single behaviour composed from two CSP output/input actions. See next.
   [ffd stands for '*f*reight *f*orwarder *d*elivery alert (from shipping line)'.]
   c. [FR] The freight forwarder is informed by the shipping line that the designated vessel is ready to return the freight it has transported.
   We assume that the time elapsed between the freight forwarder receiving this alert and the freight forwarder being able to respond is such that the alert and its response must most realistically be modeled as two behaviours. See next.
   [ffr stands for '*f*reight *f*orwarder freight *r*eturn (message, from shipping line)'.]
   d. [FE] The freight forwarder collects the freight and its saga as 'freight' is over.
   [ffe stands for '*f*reight *f*orwarder freight *e*nding'.]

686. [FM] In-between, before and after these specific freight related actions, the freight forwarder “performs” management actions “of its own” !
[ff stands for ‘*f*reight *f*orwarder *m*anagement’.]

[stands for]

```
type
681.  NewF = "nil" | F_UI × F_Mer × F_Stat
value
682.  ff: fui:FF_UI × (sluis,fuis):FF_Mer × ffstat:FF_Stat → ffhist:FF_Hist
682.      → { ch[{slui,fui}] | slui:SL_UI•slui∈sluis }  Unit
684.  ff(ffui,(sluis,fuis),ffstat)(ffhist) ≡
685.   [FC]        ffc(ffui,(sluis,fuis),ffstat)(ffhist)
685a.  [FB]   ⊓  (⌈⌉ ffb(ffui,(sluis,fuis),ffstat)(ffhist)
685b.  [FD]        ⌈⌉ ffd(ffui,(sluis,fuis),ffstat)(ffhist)
685c.  [FR]        ⌈⌉ ffr(ffui,(sluis,fuis),ffstat)(ffhist)
685d.  [FE]        ⌈⌉ ffe(ffui,(sluis,fuis),ffstat)(ffhist))
686.   [FM]     ⊓ ffm(ffui,(sluis,fuis),ffstat)(ffhist)
```

Freight Creation:

687. Freight forwarders
688. non-deterministically internally, somehow, accept freight. Technically this is modeled by the freight forwarder obtaining a hitherto unused unique identifier,
689. and, from own attribute values and from the freight “customer”, ”...”, creating a freight endurant, mkF(fui,fmer,fstat) –
690. which it transcendentally deduces into a freight behaviour
691. which behaves concurrently, ‖,
692. with a resumed freight forwarder behaviour with an augmented history that reflects the creation of a freight (endurant and behaviour).

```
type
681.  mkF :: F_UI × F_Mer × F_Stat
value
682.  ffc: ffui:FF_UI × (sluis,fuis):FF_Mer × ffstat:FF_Stat → ffprgr:FF_Prgr
682.      → { ch[{slui,fui}] | slui:SL_UI•slui∈sluis }  Unit
687.  ffc(ffui,(sluis,fuis),ffstat)(ffhist) ≡
688.     let f_ui = get_F_UI() in
689.     let mkF(fui,fmer,fstat) = heureka_Freight(f_ui,ffstat,...) in [axiom fui = f_ui]
690.     f(mkF(fui,fmer,fstat))(⟨(record_TIME())⟩) end
691.     ‖
692.     ff(ffui,(sluis,fuis),ffstat)([fui↦⟨(record_TIME(),mkF(fui,fmer,fstat))⟩]∪ffhist) end

689.  heureka_Freight: F_UI × F_Stat × ... → mkF
```

Freight Booking:

693. For the case that the freight forwarder history, for some freight, fui, records a singleton, h, which designates the creation of that freight, the freight forwarder offers the following transactions
    a. with a selected shipping line, slui, and for transport between specific harbours:
    b. I offers, to that shipping line, a booking request containing the description, mkF(...), of the freight, and the from- and to harbours of requested transport.
    c. While awaiting a reply from the shipping line,

d. the freight forwarder records the time, τ', and an element, h′, of the freight forwarder history.
e. Before resuming being the freight forwarder behaviour, ff, the freight forwarder
f. records the time, τ', and an element, h′, of the freight forwarder history.

694. For the case that the freight forwarder history, for some freight, fui, does not, for any freight (fui), record a singleton, h:⟨(τ,mkF(fui,fmer,fstat))⟩]∪fhist, which designates the creation of some freight, the freight forwarder does not engage in this alternative of the freight forwarder, ff, behaviour.

```
type
693b.     mkBooking :: SL_UI × mkF(F_UI,F_Mer,F_Stat) × (H_UI×fd:TIME) × (H_UI×td:TIME)
693b.         axiom ∀ mkb:mkBooking • fd(mkb)<td(mkb)
693c.     Reply == mk_Decline_Booking_Request(SL_UI,t:TIME,F_UI)
693c.                | mk_Accept_Booking_Request(SL_UI,t:TIME,bol:BoL,(H_UI×TIME),(H_UI×TIME))
value
682.  ffb: ffui:FF_UI × (sluis,fuis):FF_Mer × ffstat:FF_Stat → ffhist:FF_Hist
682.      → { ch[ {slui,fui} ] | slui:SL_UI•slui∈sluis }  Unit
693.  ffb(ffui,(sluis,fuis),ffstat)(ffhist:[ fui↦h:⟨(τ,mkF(fui,fmer,fstat))⟩ ]∪ffhist′) ≡
693a.     freight_booking(ffui,(sluis,fuis),ffstat)(mkF(fui,fmer,fstat))(ffhist)

693a.  freight_booking: ffui:FF_UI × (sluis,fuis):FF_Mer × ffstat:FF_Stat → mkf:MkF → ffhist:FF_Hist
682.      → { ch[ {slui,fui} ] | slui:SL_UI•slui∈sluis }  Unit
693a.  freight_booking(ffui,(sluis,fuis),ffstat)(mkf)(ffhist) ≡
693a.     let (slui,(fh,fd),(th,td)) = select_shipping_line_and_time(ffstat,mkf,ffhist) in
693b.     ch[ {ffui,slui} ] ! mkBooking(slui,mkF(fui,fmer,fstat),(fh,fd),(th,td)) ;
693d.     let τ′ = record_TIME(), h″ = ⟨(τ′,mkBooking(slui,mkF(fui,fmer,fstat),(fh,fd),(th,td)))⟩ in
693c.     let reply = ch[ {ffui,slui} ] ? in
693f.     let τ″ = record_TIME(), h‴ = ⟨(τ″,reply)⟩ in
693e.     ff(ffui,(sluis,fuis),ffstat)([ fui↦h‴⁀h″⁀h ]∪fhist)
693.      end end end end

693a.  select_shipping_line_and_time: mkF(F_UI,F_Mer,F_Stat) × MkF × FF_Hist
693a.     → SL_UI × (H_UI×fd:TIME) × (H_UI×td:TIME)
```

Freight Acceptance and Delivery

695. For the case that the freight forwarder history, for some freight, fui, records a first, i.e., a most recent element which designates the booking acceptance, [fui↦⟨(τ,mk_Accept_Booking_Request(slui,t,bol,(fh,fd),(th,td)))⟩⁀h]∪fhist, for a freight, the freight forwarder offers the following transactions:

    a. initially it offers to accept a designated, previously booked freight delivery to harbour of disembarkment;
    b. before delivering this freight
    c. the freight forwarder records the time, τ'', and an element, h″, of the freight forwarder history;
    d. before resuming being the freight forwarder behaviour, ff,
    e. and, concurrently informing the freight of its freight forwarder to harbour transfer,
    f. the freight forwarder records the time, τ''', and an element, h‴, of the freight forwarder history.

```
type
695.  BoL [ Bill-of-Lading ]
695.  mk_Accept_Booking_Request :: TIME × BoL × (H_UI×fd:TIME) × (H_UI×td:TIME)
```

```
695a.  mkPlsDelive :: F_UI × H_UI × 𝕋𝕀𝕄𝔼
695a.  mkDelivery :: F_UI × H_UI × V_UI × 𝕋𝕀𝕄𝔼
value
682.  ffd: ffui:FF_UI × (sluis,fuis):FF_Mer × ffstat:FF_Stat → FF_Hist
682.      → { ch[{slui,fui}] | slui:SL_UI•slui∈sluis }  Unit
695.  ffd(ffui,(sluis,fuis),ffstat)
695.          ([fui↦h:⟨(τ,mk_Accept_Booking_Request(slui,t,bol,(fh,fd),(th,td)))⟩⌢h′]∪fhist) ≡
695a.      let mkPlsDeliver(slui,fui,hui,vui,τ′) = ch[{ffui,slui}] ? in
695c.      let τ″ = record_𝕋𝕀𝕄𝔼(), h″ = ⟨(τ″,mkPlsDeliver(slui,fui,hui,τ″))⟩ in
695b.      ch[{ffui,hui}] ! mkDelivery(ffui,fui,hui,vui) ;
695f.      let τ‴ = record_𝕋𝕀𝕄𝔼(), h‴ = ⟨(τ‴,mkDelivery(slui,fui,hui,τ″))⟩ in
695d.      ff(ffui,(sluis,fuis),ffstat)([fui↦h‴⌢h″⌢h]∪fhist)
695e.   || ch[{ffui,fui}] ! mkXferFFtoH(τ‴,ffui,hui)
695.       end end end
```

Freight Declination and Re-booking:

696. For the case that the freight forwarder history, for some freight, fui, records a first, i.e., a most recent element which designates a booking rejection mk_Decline_Booking_Request(slui,t,fui), the freight forwarder offers the transactions that are similar to those of Items 693a–693e Page 425.

```
value
696.  ffd(ffui,(sluis,fuis),ffstat)
696.          (ffhist:[fui↦h:⟨(τ,mk_Decline_Booking_Request(slui,t,fui,mkF(fui,fmer,fstat)))⟩⌢h′]∪ffhist′) ≡
696.     freight_booking(ffui,(sluis,fuis),ffstat)(mkF(fui,fmer,fstat))(ffhist)
```

Freight Recovery:

697. For the case that the freight forwarder history, for some freight, fui, records a first, i.e., a most recent element which designates the delivery of freight, in its care: mkDelivery(slui,fui,hui,τ), the freight forwarder offers the following transaction:
    a. it offers to accept an alert from the shipping line as to the impending vessel arrival at destination port whereupon it
    b. informs the freight of its harbour to freight forwarder transfer,
    c. resumes being the freight forwarder behaviour now suitably updated with that knowledge!

```
value
682.  ffr: ffui:FF_UI × (sluis,fuis):FF_Mer × ffstat:FF_Stat → ffhist:FF_Prgr
682.      → { ch[{slui,fui}] | slui:SL_UI•slui∈sluis }  Unit
697.  ffr(ffui,(sluis,fuis),ffstat)([fui↦hist:⟨(τ,mkDelivery(slui,fui,hui,τ′))⟩⌢hist′]∪ffhist) ≡
697a.      let mkReturn(slui,fui,hui,vui,dat) = ch[{slui,ffui}] ?
697a.          τ″ = record_𝕋𝕀𝕄𝔼() in
697b.      ch[{ffui,fui}] ! mkXferHtoFF(τ″,ffui,hui)
697c.    || ff(ffui,(sluis,fuis),ffstat)([fui↦⟨(τ″,mkReturn(slui,fui,hui,vui,dat))⟩⌢hist]∪fhist)
697.      end
```

Freight Termination:

698. For the case that the freight forwarder history, for some freight, fui, records a first, i.e., a most recent element which designates the return of freight, in its care: mkReturn(slui,fui,hui,vui,dat), the freight forwarder offers the following transaction:
    a. the freight forwarder inquires with a designated return harbour, hui, as to the designated, returned freight, fui

b. and resumes being the freight forwarder behaviour now suitably updated with that knowledge!
c. while, at the same time as resumption also informing the freight that it no longer has freight status!

```
value
682.   ffe: ffui:FF_UI × (sluis,fuis):FF_Mer × ffstat:FF_Stat → hist:FF_Hist
682.       → { ch[ {slui,fui} ] | slui:SL_UI•slui∈sluis }  Unit
685d.  ffe(ffui,(sluis,fuis),ffstat)([ fui↦hist:⟨(τ''',hist:mkReturn(slui,fui,hui,vui,dat))⟩⁀hist' ]∪ffhist) ≡
698a.      let mkReturnedFreight(fui,...) = ch[ {hui,ffui} ] ? in
698b.      ff(ffui,(sluis,fuis),ffstat)([ fui↦⟨mkReturnedFreight(fui,...)⟩⁀hist ]∪fhist)
698c.   || ch[ {ffui,fui} ] ! mkTerminateFreight(ffui,...)
685d.      end
```

Freight Forwarder Management:

699.
700.
701.

```
value
682.   ffm: ffui:FF_UI × (sluis,fuis):FF_Mer × ffstat:FF_Stat → ffprgr:FF_Prgr
682.       → { ch[ {slui,fui} ] | slui:SL_UI•slui∈sluis }  Unit
686.   ffm(ffui,(sluis,fuis),ffstat)([ fui↦⟨(τ,[ FC ])⟩ ]∪fhist) ≡
699.
700.
701.
```

14.4.5.2.2 Shipping Line Behaviour Definition

702.
703.
704.
705.
706.
707.
708.
709.
710.
711.

```
value
683.   sl: slui:SL_UI × (vuis,ffuis,huis):SL_Mer × slstat:SL_Stat → slprgr:SL_Prgr → newf:NewF
683.       → { ch[ {slui,ui} ] | ui:FF_UI|V_UI|H_UI•ui∈ffuis∪huis }  Unit
703.   sl(slui,(vuis,ffuis,huis),slstat)(slprgr)(newf) ≡
703.
704.
705.
706.
707.
708.
709.
```

710.
711.

14.4.5.2.3 Vessel Behaviour Definition

14.4.5.2.4 Harbour Behaviour Definition

712.
713.
714.
715.
716.
717.
718.
719.
720.
721.

```
value
712.  harbour:
712.  harbour(hui,(ffuis,sluis),hstat)(hhist) ≡
713.
714.
715.
716.
717.
718.
719.
720.
721.
```

14.4.5.2.5 Freight Behaviour Definition

722.
723.
724.
725.
726.
727.
728.
729.
730.
731.

```
value
722.  freight: fui:F_UI × (ffui,(fhui,thui),vui,slui):F_Met × F_Stat → F_Hist →
722.        in { ch[{fhui,ui}] | ui:(FH_UI|V_UI|SL_UI)•ui ∈ {ffui,fhui,thui,vui,slui} }  Unit
722.  freight(fui,(ffui,(fhui,thui),vui,slui),hstat)(fhist) ≡
723.      let i:mkFFtoH(...) = ch[{ffui,fui}] ? in freight(fui,(ffui,(fhui,thui),vui,slui),hstat)(⟨i⟩⁀fhist) end
724.   [] let i:mk(...) =  ch[{vui,fui}] ? in freight(fui,(ffui,(fhui,thui),vui,slui),hstat)(⟨i⟩⁀fhist) end
```

```
725.  [] let i:mk(...) = ch[ {vui,fui} ] ? in freight(fui,(ffui,(fhui,thui),vui,slui),hstat)(⟨i⟩⁀fhist) end
726.  [] let i:mk(...) = ch[ {thui,fui} ] ? in freight(fui,(ffui,(fhui,thui),vui,slui),hstat)(⟨i⟩⁀fhist) end
727.  [] let i:mk(...) = ch[ {ffui,fui} ] ? in freight(fui,(ffui,(fhui,thui),vui,slui),hstat)(⟨i⟩⁀fhist) end
729.  [] let i:mk(...) = ch[ {ffui,fui} ] ? in freight(fui,(ffui,(fhui,thui),vui,slui),hstat)(⟨i⟩⁀fhist) end
730.  [] let i:mk(...) = ch[ {ffui,fui} ] ? in freight(fui,(ffui,(fhui,thui),vui,slui),hstat)(⟨i⟩⁀fhist) end
731.  [] let i:mk(...) = ch[ {ffui,fui} ] ? in skip end
```

Chapter 15
Rivers [March–April 2021]

Contents

15.1 Introduction 431
15.1.1 Waterways 431
15.1.2 Visualisation of Rivers 432
15.1.2.1 Rivers 432
15.1.2.2 Deltas 432
15.1.3 Structure of This Report 432
15.2 External Qualities – The Endurants 433
15.3 Internal Qualities 435
15.3.1 Unique Identifiers 435
15.3.2 Mereologies 435
15.3.3 Routes 436
15.3.4 Attributes 437
15.4 Conclusion 438

Presently this document represents a technical-scientific note. It is technical in that much of the material can be found in other technical notes of mine. It is – perhaps – scientific in that I am searching for a nice, well, beautiful, way of modeling rivers.

15.1 Introduction

15.1.1 Waterways

By waterways we mean rivers, canals, lakes and oceans – such as are navigable by vessels: barges, boats and ships.

Rivers are naturally flowing watercourses, and typically flow until discharging their water into a lake, sea, ocean, or another river, while canals are constructed to connect existing rivers, seas, or lakes. However, occasionally some rivers do not discharge their water into lakes, seas, oceans, or other rivers. Rivers that do not empty into another body of water might flow into the ground or simply dry up before reaching another body of water. Additionally, small rivers can also be referred to as streams, rivulets, creeks, rills, or brooks.

The natural water system of the earth includes 71% ocean with land continents being traversed by brooks, rivers, lakes and river deltas.

Headwaters are streams and rivers (tributaries) that are the source of a stream or river.

A tributary is a river or stream that flows into another stream, river, or lake.

A delta is a large, silty area at the mouth of a river at which the river splits into many different slow-flowing channels that have muddy banks. New land is created at deltas. Deltas are often triangular-shaped, hence the name (the Greek letter 'delta' is shaped like a triangle).

The trunk is the main course of river.

Confluence: In geography, a confluence (also: conflux) occurs where two or more flowing bodies of water join together to form a single flow. A confluence can occur in several configurations: at the point where a tributary joins a larger river (main stem); or where two streams meet to become the source of a river of a new name; or where two separated channels of a river (forming a river island) rejoin at the downstream end.

Towns and Harbours: In this report we model towns. That is, we therefore also model that towns have harbours – allowing river (and canal) vessels to berth (a place for mooring in a harbour) for cargo loading, unloading and resting.

15.1.2 Visualisation of Rivers

15.1.2.1 Rivers

Figures 15.1 and 15.2 on the facing page illustrate a number of rivers.

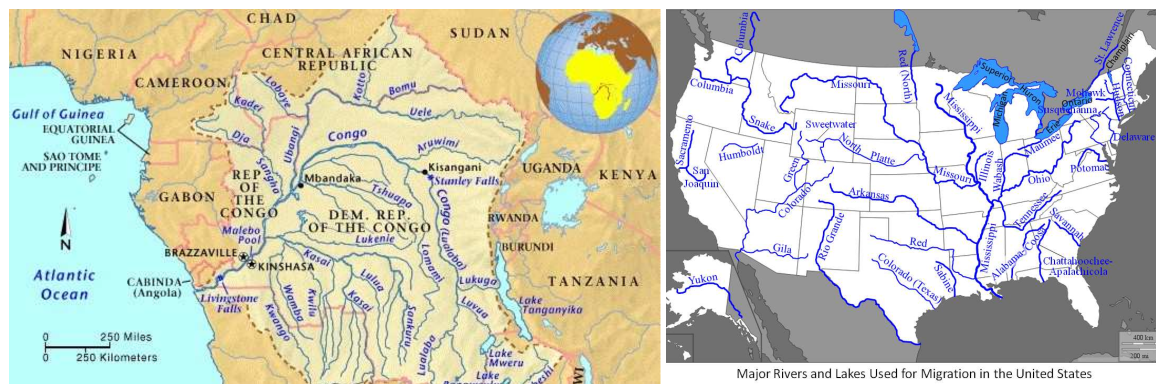


Fig. 15.1 The Congo and the US Rivers

15.1.2.2 Deltas

We illustrate four deltas, Fig. 15.3 on the next page:

15.1.3 Structure of This Report

Rivers are narrated and formalised in Sects.:

- **15.2** [Endurants],
- **15.3.1** [Unique Identifiers],
- **15.3.2** [Mereology], and
- **15.3.4** [Attributes].

We omit from this compendium references to a number of 'River Terminologies'.

Fig. 15.2 The Amazon and The Danube Rivers

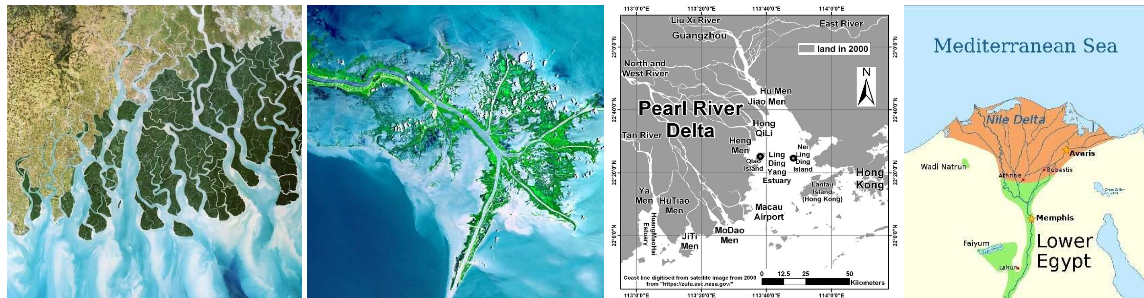


Fig. 15.3 The Ganges, Mississippi, Pearl and the Nile Deltas

15.2 External Qualities – The Endurants

732. A river net is modeled as a graph, more specifically as a tree. The *root* of that river net tree is the mouth (or delta) of the river net. The *leaves* of that river net tree are the sources of respective trees. Paths from leaves to the root define *flows* of water.
733. We can thus, from a river net observe vertices
734. and edges.
735. River vertices model either a *source:* **so:SO**, a *mouth:* **mo:MO**, or possibly some *confluence:* **ko:KO**.
     A river may thus be "punctuated" by zero or more confluences, k:KO.
     A confluence defines the joining a 'main' river with zero[158] or more rivers into that 'main' river. We can talk about the "upstream" and the "downstream" of rivers from their confluence.
736. River edges model *stretches:* st:ST.
     A stretch is a linear sequences of simple, se:SE, or composite ce:CE, river elements.
737. River elements are either simple: (ch) river channels, which we shall call *river channels*: **CH**, or (la) lakes: **LA**, or (lo) locks: **LO**, or (wa) waterfalls (or *rapids*): **WA**, or (da) dams: **DA**, or (to) towns

[158] Normally, though, one would expect, not zero, but one

(cities, villages): **to:TO**[159]; or composite, ce:CE: a dam with a lock, **(da:DA,la:LA)**, a town with a lake, **(to:TO,la:LA)**, etcetera; even a town with a lake and a confluence, **to:TO,la:LA,ko:KO**. Etcetera.

type
732. RiN
733. V
734. E
735. SO, MO, KO
736. ST = (SE|CE)*
737. CH, LA, LO, WA, KO, DA, TO
737. SE = CH | LA | LO | WA | DA | TO
737. DaLo, WaLo, ToLa, ToLaKo, ...
737. CE = DaLo | WaLo | ToLa | ToLaKo | ...
value
735. obs_Vs: RiN → V-**set**
735. **axiom**
735. ∀ g:G,vs:V-**set**•vs ∈ obs_Vs(g) ⇒ vs≠{}
735. ∧ ∀ v:V•v ∈ vs ⇒ is_SO(v) ∨ is_KO(v) ∨ is_MO(v)
736. obs_Es: RiN → E-**set**
736. **axiom**
736. ∀ g:G,es:E-**set**•es ∈ obs_Es(g) ⇒ es≠{}
736. ∧ ∀ e:E•e ∈ es ⇒ is_ST(e)
736. obs_ST: E → ST
732. xtr_In_Degree_0_Vertices: RiN → SO-**set**
732. xtr_Out_Degree_0_Vertex: RiN → MO

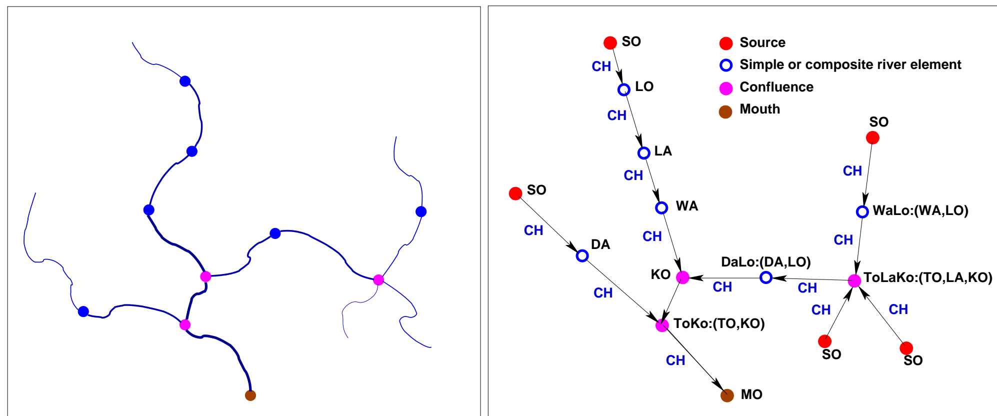


Fig. 15.4 The "Composition" of a River Net: Right Tree is an abstraction of the Left Tree

[159] Towns is here really a synonym for river harbours, places along the river (or a canal) where river vessels can stop (moor) for the loading and unloading of cargo and for resting.

15.3 Internal Qualities

We refer to [55, Chapter 5]

15.3.1 Unique Identifiers

We shall associate unique identifiers both with vertices, edges and verteax and edge elements.

738. River net vertices and edges have unique identifiers.
739. River net sources, confluences and mouths have unique identifiers.
740. River net stretches have unique identifiers.
741. River net channels, lakes, locks, waterfalls, dams and towns as well as combinations of these, that is, simple and composite river entities have unique identifiers.

```
type
738. V_UI, E_UI
739. SO_UI, KO_UI, MO_UI
740. ST_UI
741. CH_UI, LA_UI, LO_UI, WA_UI, DA_UI, TO_UI, DaLo_UI, WaLo_UI, ToLa_UI, ToLaKo_UI, ...
value
738. uid_V: V→V_UI, uid_E: E→ E_U
739. uid_SO: SO→SO_UI, uid_KO: KO→KO_UI, uid_MO: MO→MO_UI,
740. uid_ST: ST→ST_UI
741. uid_CH: CH→CH_UI, uid_LA: LA→LA_UI, uid_LO: LO→LO_UI, uid_WA: WA→WA_UI,
741. uid_DA: DA→DA_UI, uid_TO: TO→TO_UI,
741. uid_DaLo: DaLo→DaLo_UI, uid_WaLo: WaLo→WaLo_UI, uid_ToLa: ToLa→ToLa_UI,
741. uid_ToLaKo: ToLaKo→ToLaKo_UI, ...
```

742. All these identifiers are distinct.

The ⋒ operator takes the pairwise intersection of the types in its argument list and examines them for disjointness.

```
axiom
742. ⋒(V_UI,E_UI,SO_UI,KO_UI,MO_UI,ST_UI,CH_UI,
742.          LA_UI,LO_UI,WA_UI,DA_UI,TO_UI,DaLo_UI,WaLo_UI,ToLa_UI,ToLaKo_UI)
```

743. There are [many] other constraints, please state them !

743. [left as exercise to the reader !]

15.3.2 Mereologies

744. The mereology of a river vertex is a pair: a set of unique identifiers, E_UI, of river edges, i.e., stretches, linear sequences of simple and composite river elements, incident upon the vertex, and a set of unique identifiers, E_UI, of river edges emanating from the vertex. If the vertex is a source then the first element of this pair is empty. If the vertex is a mount then the second element of this pair is empty. For a confluence vertex both elements of the pair are non-empty.

745. The mereology of a river edge, that is, the linear sequence of simple and composite river elements between two adjacent vertices, is a pair: the first element is a unique identifier of a river vertex and so is the second element of the pair.

We present the river net mereology in two forms. The first was with respect to its graph rendition. The second is with respect to its river element rendition.

746. The mereology of a source is just the single unique identifier of the first simple or composite river element of the stretch emanating from the source.
747. The mereology of a confluence is a triplet: the single unique identifier of the last simple or composite river element of the stretch of the main river incident upon the source, a set of unique identifier of the last simple or composite river element of the stretches of the tributary rivers incident upon the source, and the single unique identifier of the first simple or composite river element of the main river stretch emanating from the confluence.
748. The mereology of a mouth is just the single unique identifier of the last simple or composite river element of the stretch incident upon the mouth
749. The mereologies of simple and composite river elements are pairs: of the unique identifier of the river elements, including sources and confluences, upstream adjacent to the river element being "mereologised", and of the unique identifier of the river elements, including confluences and mouths, downstream adjacent to the river element being "mereologised".

744. Mer_V = E_UI-**set** × E_UI-**set**
745. Mer_E = V_UI × V_UI
746. Mer_SO = SE_UI | CE_UI
747. Mer_KO = (SE_UI|CE_UI) × (SE_UI|CE_UI)-**set** × (SE_UI|CE_UI)
748. Mer_MO = SE_UI | CE_UI
749. Mer_RE = (SO_UI|CO_UI|SE_UI|CE_UI) × (SE_UI|CE_UI|CO_UI|MO_UI)

750. The unique vertex and edge identifiers must be identifiers of the vertices and edges of a graph.
751. Similarly, the unique source, confluence and mouth identifiers must be identifiers of respective sources, confluences and mouths of a graph.
752. And likewise for simple and composite element identifiers.
753. No two sources, confluences, mouths, simple and composite elements have identical uinque identifiers.
754. There are other constraints, please state them!

axiom
750. [left as exercise to the reader !]
751. [left as exercise to the reader !]
752. [left as exercise to the reader !]
753. [left as exercise to the reader !]
754. [left as exercise to the reader !]

15.3.3 Routes

755. A vertex-edge-vertex path is a sequence of zero or more edges. We define the edge_paths function – recursively.
756. That is, the empty sequence, $\langle\rangle$, is a vertex-edge-vertex path, [the first basis clause].
757. If e is an edge of g, and if (vi,vj) is in the mereology of e, then the $\langle(vi,ej,vk)\rangle$, where ej is the unique identifier of e is a vertex-edge-vertex path.

758. If p and p' are paths of g such that the last vertex identifier of the last element of p is the same as the first vertex identifier of the first element of p', then the sequence p followed by the sequence p' is a vertex-edge-vertex path of g [the inductive clause].
759. Only such paths which can be constructed by the above rules are edge paths [the extremal clause].

```
type
755. EP = E^ω
755. edge_paths: G → EP-set
755. edge_paths(g) ≡
756.     let ps = {⟨⟩}
757.         ∪ {⟨(vi,uid_E(e),vk)⟩|e:E•e ∈ xtr_Es(g)∧(vi,vk)∈mereo_E(e)}
758.         ∪ {p⌢p′|p,p′:EP•{p,p′}⊆ps∧lVlEP(9)=fVlfEP(p′)} in
755.     ps end
```

15.3.4 Attributes

This author is not "an expert" on neither geograhical matters relating to rivers, lakes, etc., nor on the management of rivers: flood control, river taffic, etc. So, please, do not expect a very illuminating set of river attribute examples. All the attribute specifications are "tuned" to the purpose of the ensuing domain desription: whether for one or another form of river system study or eventual software system realisation.

760. River entities have geodetical positions –
761. all three dimensions: longitude, latitude and altitude[160].
762. River entities cover geodetical areas[161].
763. River entities have normal, low, high and overflow water levels[162].
764. River channels have "extent" in the form, for example of a precise description[163] of its course.[164]
765. Lakes have a precise [three dimensional] description of their form, ...
766. Locks have ... et cetera
767. Waterfalls ...
768. Dams ...
769. Towns ...
770. Sources ...
771. Confluences ...
772. Mouths ...
773. Compositions of these have respective unions of these attributes.

```
type
760. GeoPos = Long × Lat × Alt
761. Long, Lat, Alt
763. Area
763. LoWL = ..., NoWL = ..., HiWL = ..., OfWL = ...
764. Course = ...
```

[160] These are facts: How we represent them is a matter for geographers. Also: What is really mean by the 'position' of a source, or a river channel, etc. ? Also that is left for others to care about !

[161] See Footnote 160.

[162] See Footnote 160.

[163] See Footnote 160. In any **domain description**, yes, a precise description – whether "computable" [i.e., realizable] or not !

[164] – in a subsequent **requirements prescription** the domain description's "precise" form is replaced by, for example, a reasonably detailed [and computable] three dimensional *Bézier curve* specification [en.wikipedia.org/wiki/B%C3%A9zier_curve].

765. LakeForm = ...
767. ...; 768. ...; 769. ...;770. ...; 771. ...; 772. ...; 773. ...
value
760. attr_GeoPos: (SO|KO|MO|SE|CE) → GeoPos
761. attr_Long: GeoPos → ..., attr_lat: GeoPos → ...,attr_Alt: GeoPos → ...
763. attr_Area: (SO|KO|MO|SE|CE) → Area
763. attr_(LoWL|NoWL|HiWL|OfWL): (SO|KO|MO|SE|CE) → LoWL|NoWL|HiWL|OfWL
764. attr_CH: CH → Course
765. attr_LakeForm: LA → LakeForm
766. attr_...: LO → ...; 767. attr_...: WF → ...; 768. attr_...: DA → ...; 769. attr_...: TO → ...;
770. attr_...: SO → ...; 771. attr_...: KO → ...; 772. attr_...: MO → ...; 773. attr_...: ... → ...

We illustrate the issue of river attributes primarily to show you the sheer size and complexity of the task !

774. River entities have positions "within" their areas[165].
775. No two distinct river entities have conflicting (?) areas[166].
776. Two mereologically immediately adjacent river entities have bordering areas[167].
777.
778.

Axiom 775 is rather "sweeping". It implies, of course, that river channels do not cross one another; that two or more non-channel river entities similarly do not "interfere" with one another, i.e., are truly "separate".

15.4 Conclusion

TO BE WRITTEN

[165] See Footnote 160.
[166] For example: their areas do not overlap. See Footnote 160.
[167] See Footnote 160.

Chapter 16
Canals [March–April 2021]

Contents

16.1 Introduction 440
16.2 Visualisation of Canals 440
16.2.1 Canals and Water Systems 440
16.2.2 Locks 441
16.3 The Endurants 442
16.3.1 Some Introductory Remarks 443
16.3.1.1 The Dutch Polder System 443
16.3.1.2 Natural versus Artefactual Domains 443
16.3.1.3 Editorial Remarks 444
16.3.1.4 A Broad Sketch Narrative of Canal System Entities 445
16.3.1.5 A Plan for The Canal System Description 446
16.3.1.6 No Structures 446
16.3.1.7 Sequences of Presentation 447
16.3.1.8 Naming Conventions 447
16.3.2 External Qualities 448
16.3.2.1 Endurant Sorts 448
16.3.2.2 Some Calculations 451
16.3.3 Internal Qualities 453
16.3.3.1 Unique Identifiers 453
16.3.3.1.1 Unique Identifier Sorts 453
16.3.3.1.2 Some Calculations 454
16.3.3.1.3 An Axiom 456
16.3.3.1.4 Another Representation of UI Values 456
16.3.3.1.5 An Extract Function 456
16.3.3.2 Mereologies 457
16.3.3.2.1 Mereology Types 457
16.3.3.2.2 The Mereology Axiom 461
16.3.3.2.3 Well-formed Mereologies 461
16.3.3.3 Routes 467
16.3.3.3.1 Preliminaries 467
16.3.3.3.2 All Routes 468
16.3.3.3.3 Connected Canal Systems 468
16.3.3.3.4 A Canal System Axiom 468
16.3.3.4 Attributes 469
16.3.3.4.1 Spatial and Temporal Attributes 469
16.3.3.4.2 Canal System, Net and Polder Attributes 471
16.3.3.4.3 Canal Hub and Link Attributes 471
16.3.3.5 Well-formedness of Attributes 472
16.3.4 Speculations 472
16.4 Conclusion 473

Presently this document represents a technical-scientific note. It is technical in that much of the material can be found in other technical notes of mine. It is – perhaps – scientific in that I am searching for a nice, well, beautiful, way of modeling canals, such as for example those of the Dutch Rijkswaterstaat[168]. I am fascinated with Holland's tackling of their land/water/river/ocean levels.

16.1 Introduction

Canals are artificial or human-made channels or waterways that are used for navigation, transporting water, crop irrigation, or drainage purposes. Therefore, a canal can be considered an artificial version of a river. Canals are artificial or human-made channels or waterways that are used for navigation, transporting water, crop irrigation, or drainage purposes. Therefore, a canal can be considered an artificial version of a river.

16.2 Visualisation of Canals

16.2.1 Canals and Water Systems

We illustrate just four ship/barge/boat and water level control canal systems, Figs. 16.1, 16.2, 16.3 on the facing page and 16.4 on page 442.

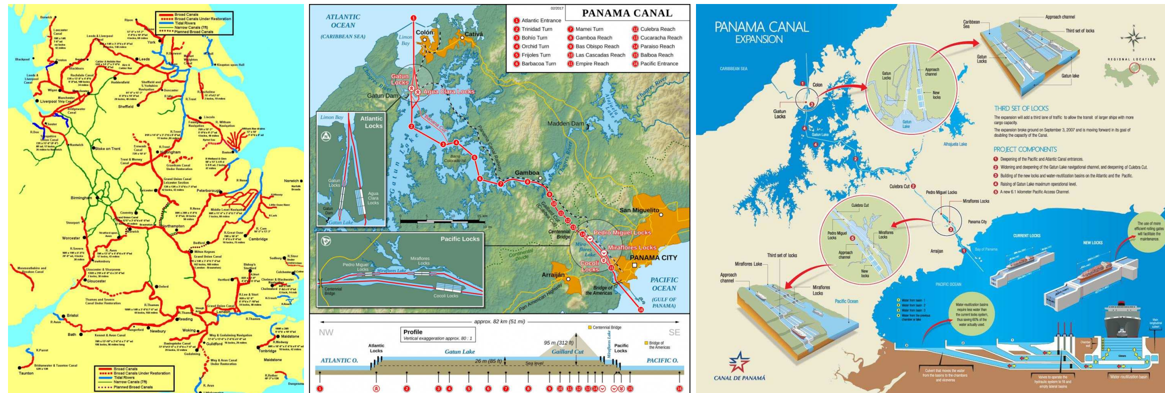


Fig. 16.1 UK Canals and The Panama Canal

The rightmost figure of Fig. 16.4 is from the Dutch *Rijkswaaterstaat:* www.rijkswaterstaat.nl/english/.

168 `https://www.rijkswaterstaat.nl/` The Dutch canal system is first and foremost, it appears, for the control of water levels, secondly for ship/barge/boat navigation.

Fig. 16.2 The Swedish Göta Kanal

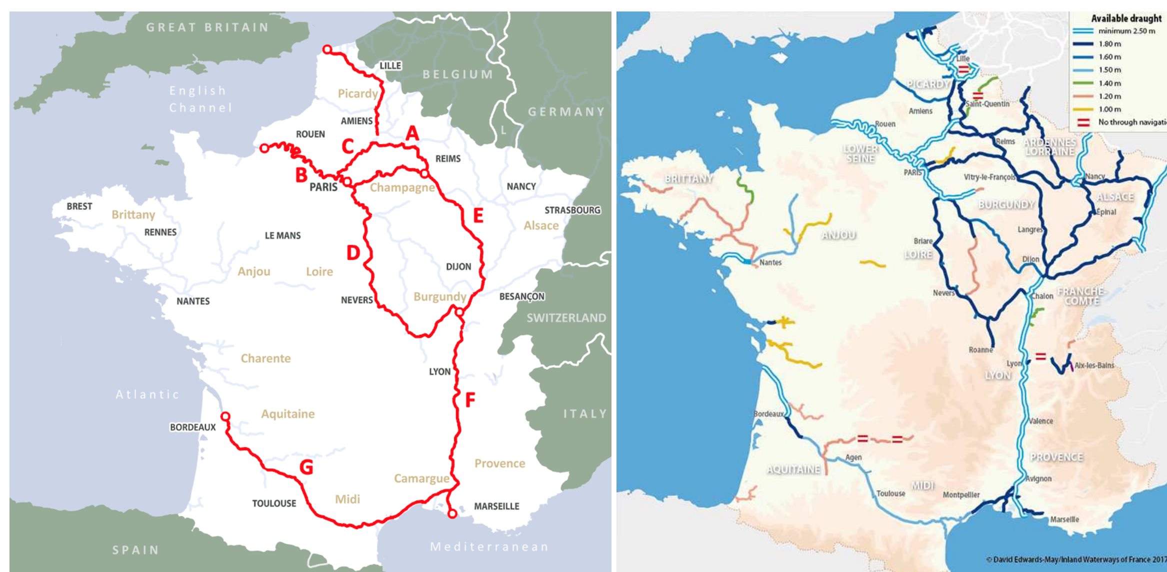


Fig. 16.3 French Rivers and Canals

16.2.2 Locks

A lock is a device used for raising and lowering boats, ships and other watercraft between stretches of water of different levels on river and canal waterways. The distinguishing feature of a lock is a fixed chamber in which the water level can be varied. Locks are used to make a river more easily navigable, or to allow a canal to cross land that is not level. Later canals used more and larger locks to allow a more direct route to be taken.[169]

We illustrate a number of locks: Figs. 16.5 on the following page and 16.6 on page 443.

[169] https://en.wikipedia.org/wiki/Lock_(water_navigation)

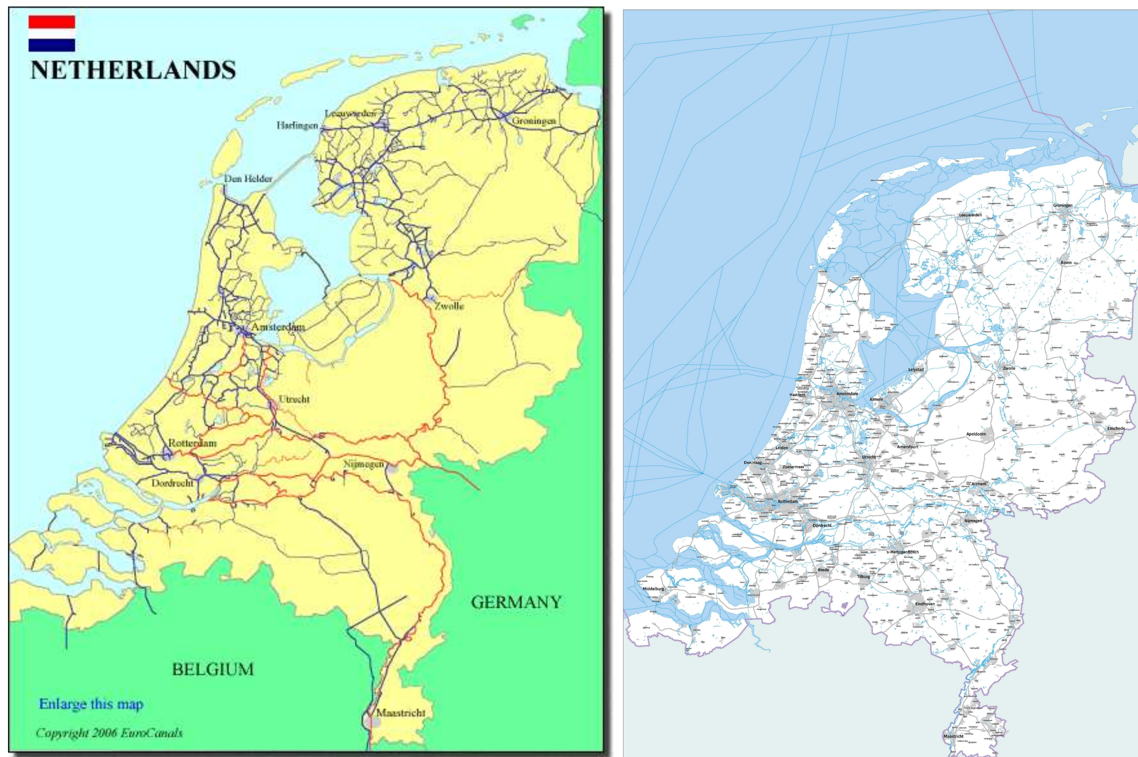


Fig. 16.4 Dutch Rivers and Canals

Fig. 16.5 Inland Canal Locks

16.3 The Endurants

As an example we wish our model to include the Dutch system of *polders, pumps, canals, locks, dikes, flood barriers, lakes, storm barriers* and the *ocean*.[170]

[170] www.fao.org/fileadmin/templates/giahs/PDF/Dutch-Polder-System_2010.pdf

Fig. 16.6 Harbour Canal Locks

16.3.1 Some Introductory Remarks

16.3.1.1 The Dutch Polder System

We refer to Figs. 16.8 to 16.11 on pages 444–445.

Fig. 16.7 The Dutch Polder System

16.3.1.2 Natural versus Artefactual Domains

In contrast to river nets modeled earlier in this compendium, a system of mostly natural endurants, canal systems of *polders, pumps, canals, locks, dikes, flood barriers, lakes, storm barriers* and the *ocean* are, in a sense, dominated by man-made, i.e., artefactual endurants.

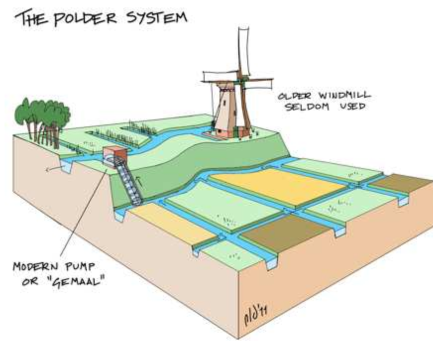


Fig. 16.8 A Polder Schematic and The De Cruquius Pump

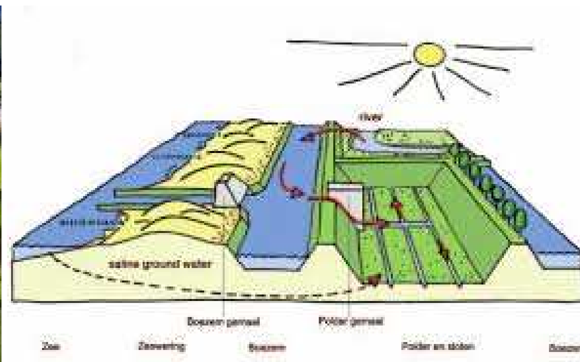


Fig. 16.9 A Polder. Another Polder Schematic

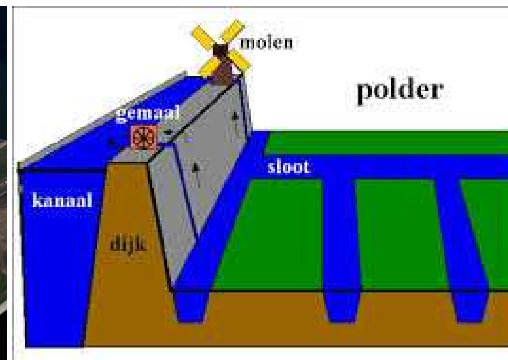


Fig. 16.10 A Barrier. A Final Polder Schematic

16.3.1.3 Editorial Remarks

In order to develop an appropriate domain analysis & description of a reasonably comprehensive and representative canal domain I need answers to the following questions – and may more that can be derived from answer to these questions:

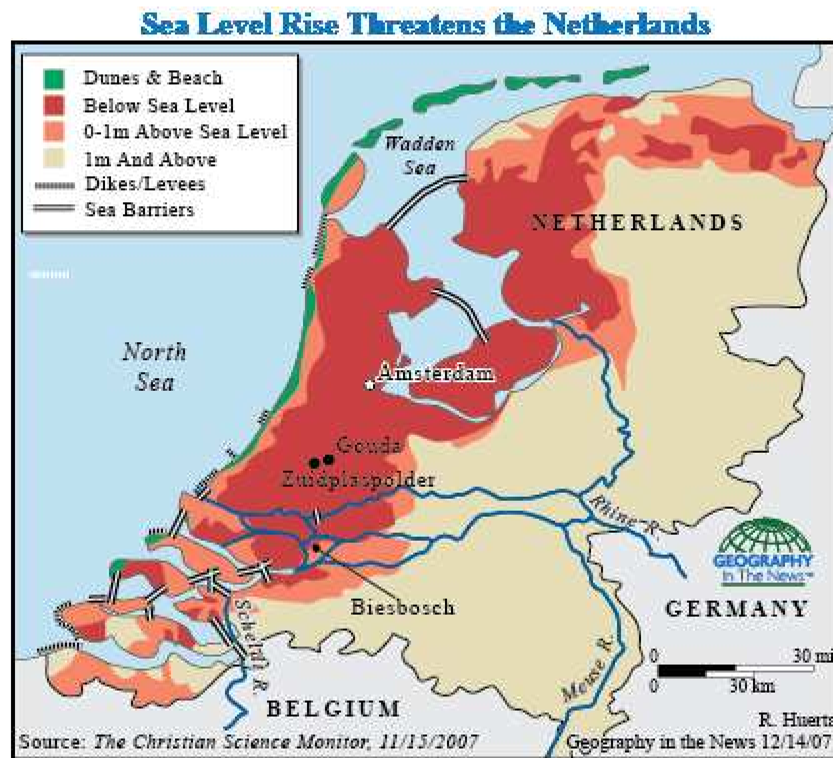


Fig. 16.11 The Land-Water Levels of The Netherlands

- **Canal Flow:** Are canals generally stagnant, or do canal water flow, that is, do canal flow have a preferred direction ?
- **Canal Graphs:** Do a canal form a[n undirected] graph, i.e., can canals be confluent with other canals ?
- **Canals and Rivers:** It is assumed that canals can be confluent with rivers. When canals join a river is it always with a lock of the canal onto the river – and the river flow is basically not interfered with by that canal, or otherwise ?
- **Canal Levels:** Can a canal pass a river overhead ? Or otherwise ? Same for canals and roads. Do canals run through mountains or over valleys ?
- **Canal Locks:** It is assumed that an otherwise "unhindered" stretch of canal can have one or more locks. Yes or no ?
- **Canal Pumps:** It is assumed that there are two kinds of canal pumps: those in connection with locks, and those **not** in connection wit any locks. Yes or no ? I need information about the latter.
- **Polders and Pumps:** Are polders predominantly characterisable in terms of their land area and the pumps that keep these dry ?
- **Pumps and Canals:** Do polder pumps always operate in the context of canals ?

16.3.1.4 A Broad Sketch Narrative of Canal System Entities

- We take our departure point in the polders: So a polder-etc.-canal system contains **polders** and **polder pumps** take the water out of the polders and "puts" it in higher level **canals**.

- **Canals** are modeled as an undirected [general] graph whose vertices are **canal entities** and whose edges are given by the mereology of these entities – as to how they are topologically connected.
- The following are **canal entities:**
  - **canal channels:** like river channels, only artefactual;
  - **canal locks:** the locks as illustrated earlier;
  - **canal pumps:** pumps water into locks – rather than using water from higher level canals;
  - **canal gates:** protects the interior from ocean storm surges.

16.3.1.5 A Plan for The Canal System Description

Our plan is to analyse & describe

- external qualities of canal system endurants, Sect. **16.3.2**.
- internal qualities of canal system, Sect. **16.3.3.1**
- internal qualities of canal system, Sect. **16.3.3.2**
- internal qualities of canal system, Sect. **16.3.3.4**

For each of these categories we analyse & describe

- **sort**s and **type**s of these entities: endurants, unique identifiers, mereologies and attributes;
- **observer functions**, i.e., **obs_**···, **uid_**··· and **attr_**··· for the observance of endurants, their unique identifiers, their mereologies and their attributes;
- **auxiliary functions** and
- **well-formedness predicates is_wf_**···.

The external and internal quality definitions should be so conceived by the domain analyser & describer as to capture an essence, if not "the essence", of endurants. But they can never capture the essence "completely". As for the relation between *context free grammars* and *context sensitive grammars*, we must therefore introduce the notion of **well-formedness axioms**. The axioms constrain the relations between external and the various categories of internal qualities. More specifically:

779. The well-formedness of a canal system, is_wf_CS [779] is the conjunction of the well-formedness of canal system identifiers, is_wf_CS_Identities [841 on page 454], mereologies, is_wf_CS_Mereology [883a on page 461], and attributes, is_wf_CS_Attributes [945 on page 472].

type
779. CS
value
779. is_wf_CS: CS → **Bool**
779. is_wf_CS(cs) ≡ is_wf_CS_Identifiers(cs)∧is_wf_CS_Mereologies(cs)∧ is_wf_CS_Attributes(cs)

16.3.1.6 No Structures

In this (long and detailed) example domain analysis & description I shall not use the pragmatic "device" of *structures* [cf, [55, *Sect.4.10*]]. Everything will be painstakingly analysed and described.

Some clarifying comments are in order:

- Compound endurants are either

  - *Cartesian*[171] or
  - *sets*.
- In analysing Cartesians, say c, into composite endurants, we analyse c into a number of components, $c_i, c_j, ..., c_k$, of respective sorts, $C_i, C_j, ..., C_k$, by means of observers obs_Ci, obs_Cj, ..., obs_Ck.
  - The Cartesians, C, in this report, all have:
    - unique identifiers,
    - mereologies and
    - attributes.
  - So do each of the $C_i, C_j, ..., C_k$.
- In analysing an endurant, E, into sets, say s or sort S, we first analyses E into a separately observable endurant Ss, i.e., obs_Ss, which we then, at the same time define as Ss = S-**set**.
- An Ss endurant thus has all the internal qualities:
  - a unique identifier,
  - a mereology
  - and attributes.

16.3.1.7 Sequences of Presentation

The sequence in which endurant sorts are introduced is "repeated" in the sequences in which unique identifier sorts and mereology types are introduced. Thus the sequences of narrative and formal

- endurant items,
  - sort items, Items 780–794 on pages 449–450,
  - value items, Items 801–820 on pages 451–453 and
  - "alternative" value items, Items v801–v820 on page 453,

are "repeated" in

- unique identifier
  - sort items, Items 821–839 on pages 453–454,
  - value items, Items 841–856 on pages 454–455 and
  - "alternative" value items, Items v794–v808 on page 456,

and in

- mereology
  - type items, Items 865 to 881 on pages 457–460 and
  - well-formedness items, Items 883 to 900 on pages 461–466.

16.3.1.8 Naming Conventions

Some care has been taken in order to name endurants, including sets of and predicates and functions over these; their unique identifiers and typed sets and values of and predicates and functions over these; their mereologies and typed sets and values of and predicates and functions

---

[171] Cartesian is spelled with a large 'C', after René Descartes, the French mathematician (1596–1650) `https://da.wikipedia.org/wiki/René_Descartes`.

over these; and their attributes and typed sets and values of and predicates and functions over these.

16.3.2 External Qualities

16.3.2.1 Endurant Sorts

The narrative(s) that follow serves two purposes:

- a formal purpose: the identification of endurants, and
- an informal purpose: in "casually familiarising" the reader as the the rôle of these endurants.

The former purpose is the only one to formalise. The latter purpose informally "herald" things to come – motivating, in a sense, theses "things", the internal qualities and, if we had included a treatment of canal perdurants, the behaviours of these canal elements seen as behaviours.

All the elements mentioned below consist of both discrete endurants and fluids, i.e., water. In contrast to the treatment of such conjoins in [55, Sect. 4.13.3] we shall, in an informal digression from the principles, techniques and tool of the *analysis & description calculi* of [55, Chapters 4–5], omit "half the story" ! It will be partly "restored" in out treatment of *canal attributes*, Sect. **16.3.3.4**.

In this section we shall narrate all the different endurant sorts, Items 780–800 (Pages 448–449), before we formalise them (Pages 449–450). We beg the readers forebearance in possibly having to thumb between narrative (page)s and formalisations (page)s.

780. **Canal systems**, CS, are given.
781. From a canal system one can observe a **canal net**, CN.
782. From a canal system one can observe a **polder aggregate**, PA.

Observing two endurants of a composite endurant is as if the composite is a Cartesian product of two. Hence the "(...,...)" of Fig. 16.12 on page 450.

783. From canal nets one can observe **canal hub aggregates**, CA_HA, and
784. **canal link aggregates**, CA_LA.
785. From a polder aggregate one can observe a **polder set**, Ps, of polders, P. One observes the set, not its elements.
786. From a canal hub aggregate one can observe a **hub set**, CA_Hs, of hubs, CA_H. One observes the set, not its elements.
787. From a canal link aggregate one can observe a **canal link set**, CA_Ls, of canal links, CA_L. One observes the set, not its elements.
788. Polders are considered atomic. A **polder** is a low-lying tract of land that forms an artificial hydrological entity, enclosed by embankments known as dikes. The three types of polder[172] are:

---

[172] The ground level in drained marshes subsides over time. All polders will eventually be below the surrounding water level some or all of the time. Water enters the low-lying polder through infiltration and water pressure of groundwater, or rainfall, or transport of water by rivers and canals. This usually means that the polder has an excess of water, which is pumped out or drained by opening sluices at low tide. Care must be taken not to set the internal water level too low. Polder land made up of peat (former marshland) will sink in relation to its previous level, because of peat decomposing when exposed to oxygen from the air.

Polders are at risk from flooding at all times, and care must be taken to protect the surrounding dikes. Dikes are typically built with locally available materials, and each material has its own risks: sand is prone to collapse owing to saturation by water; dry peat is lighter than water and potentially unable to retain water in very dry seasons. Some animals dig tunnels in the barrier, allowing water to infiltrate the structure; the muskrat is known for this activity and hunted in certain European countries because of it. Polders are most commonly, though not exclusively, found in river deltas, former fenlands, and coastal areas.

- Land reclaimed from a body of water, such as a lake or the seabed.
- Flood plains separated from the sea or river by a dike.
- Marshes separated from the surrounding water by a dike and subsequently drained; these are also known as koogs.

789. Canal hubs are considered atomic and are:
790. either **canal begin/end**s (that is, where there is no continuation of a canal: where it ends "blind', or where begins "suddenly'[173]), CA_BE,
791. or **canal confluence**s (of three or more canals[174]), CA_CO,
792. or **canal outlet**s, CA_OU (where canals join a *river*, or a *lake*, or an *ocean*). These sorts are all considered atomic.
793. **Canal links** are aggregates.
794. From canal links we choose to observe a set of **canal link element**s, CA_LE[175]. (Canal links are such, through their mereology, see Sect. **16.3.3.2**, that they form two reversible sequences between connecting edges.)
795. Canal link elements are considered atomic and are
796. either **canal channel**s, CA_CH[176],
797. or **canal lock**s, CA_LO[177],
798. or **canal lock pump**s, CA_LO_PU[178],
799. or **canal polder pump**s, CA_PO_PU[179].
800. We do not further describe **canal outlets, rivers, lakes** and **oceans**.

type
780. CS
781. CN
782. PA
783. CA_HA
784. CA_LA
785. Ps = P-**set**, P
786. CA_Hs = CA_H-**set**
787. CA_Ls = CA_L-**set**
788. P
789. CA_H == CA_BE|CA_CO|CA_OU
790. CA_BE :: ...
791. CA_CO :: ..
792. CA_OU :: ..
793. CA_L
794. CA_LEs = CA_LE-**set**
795. CA_LE == CA_CH|CA_LO|CA_LO_PU|CA_PO_PU
796. CA_CH :: ...

[173] A canal "end" is a canal channel which is "connected" only at one end to a canal channel.

[174] Without loss of generality we model only confluences of three canals.

[175] We could have chosen other abstractions, for example, to observe a sequence of elements. More on this later.

[176] A canal channel offers a "straight", un-interrupted "stretch" of water – like does a river channel.

[177] A canal lock) is always connected to two distinct canal link elements. Canal locks still act like a waterway, as does a canal channel.

[178] Canal lock pumps are like canal locks, but with pumps. A *canal lock pump* is connected to a canal lock and the two canal link elements connected by the lock. It takes takes water either from the lower lying canal link element and pumps it up into the lock chamber, or from the lock chamber and pumps it up to the higher level canal link element. Canal locks are without pumps. The canal link elements mentioned here are usually canal channels.

[179] A canal polder pump is a pump that takes water from a polder and deposits it in a canal which is at a higher level than the polder.

797. CA_LO :: ...
799. CA_PO_PU :: ...
value
781. obs_CN: CS → CN
782. obs_PA: CS → PA
783. obs_CA_HA: CS → CA_HA
784. obs_CA_LA: CN → CA_LA
785. obs_Ps: PA → Ps
786. obs_CA_Hs: CA_HA → CA_Hs
787. obs_CA_Ls: CA_LA → CA_Ls
794. obs_CA_LEs: CA_L → CA_LEs

Figure 16.12 shows the ontology of a wide class of canal systems.

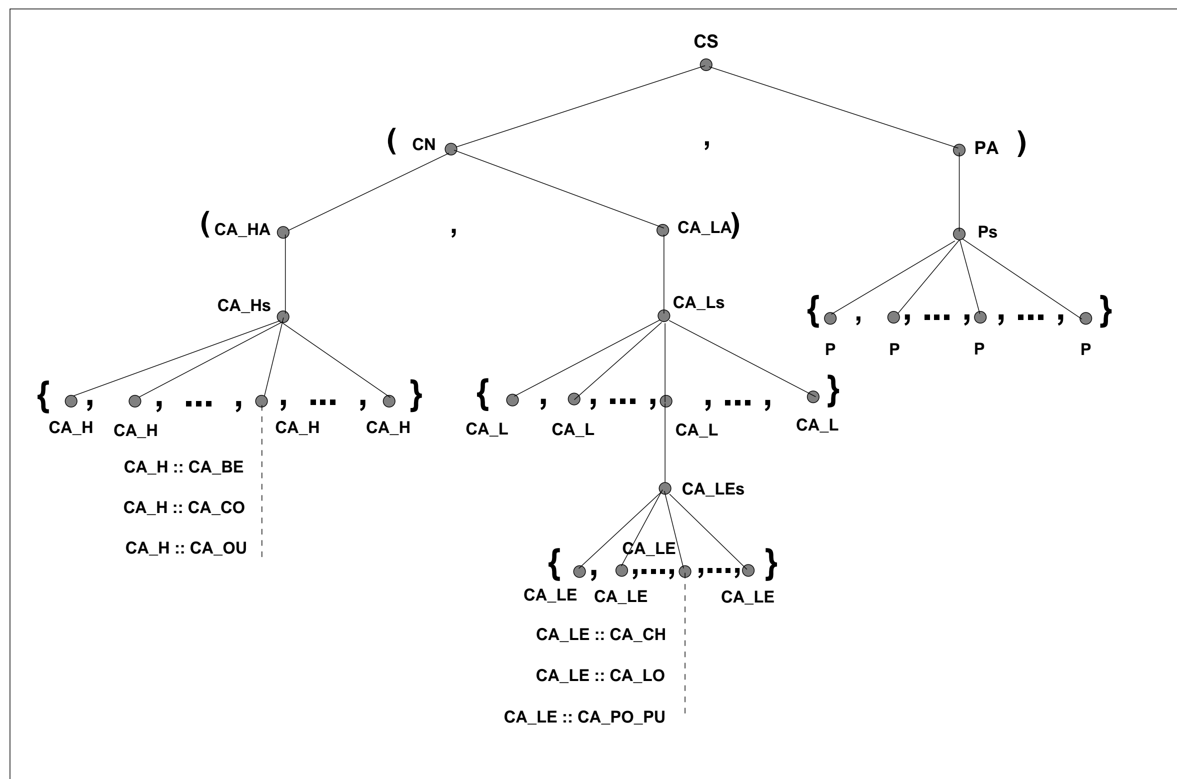


Fig. 16.12 Canal System Ontology

Figure 16.13 on the facing page shows the schematisation of a specific canal system.

Figure 16.14 on the next page shows the individual endurants of a canal system for that shown in Fig. 16.13 on the facing page. Given what we have formalised so far, i.e., formula 780–787, this is really all we can "diagram". The "part" list of Fig. 16.14 on the next page cannot show other than that there are these parts, but not how they are connected – that is first revealed when we ascribe mereologies – and that there are canal channels, not, for example, their length – that is first revealed when we ascribe attributes, such as length.

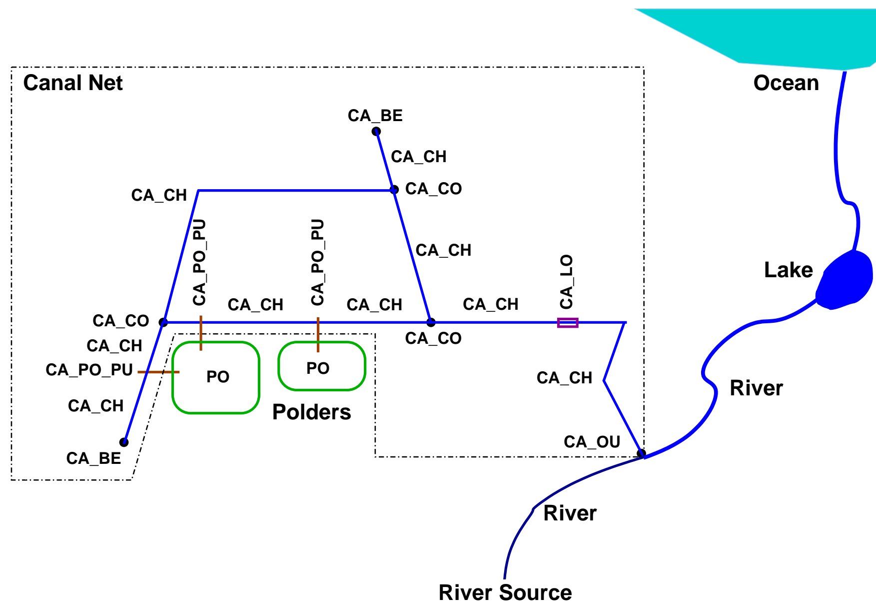


Fig. 16.13 A Schematised Specific Canal System: Canal Net + Polders

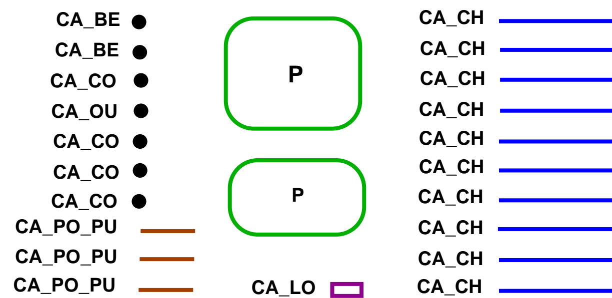


Fig. 16.14 Component Endurants of the Canal System of Fig. 16.13

16.3.2.2 Some Calculations

We refer to Fig. 16.15 on the following page. We shall list the endurant parts – and later on their unique identifierss in the left-to-right order of a breadth-first traversal of the canal ontology.

801. Let *cs* be a "global" canal system.[180]
802. Canal nets and polders can be seen as consisting of the following endurants, modeled as a map, map_ends:
803. the canal system, cs_{end}, [ι 780, π 448]

[180] Introducing *cs* allows us to refer to it and its "derivatives", "all over", and thus "universally prefix quantify" many axioms.

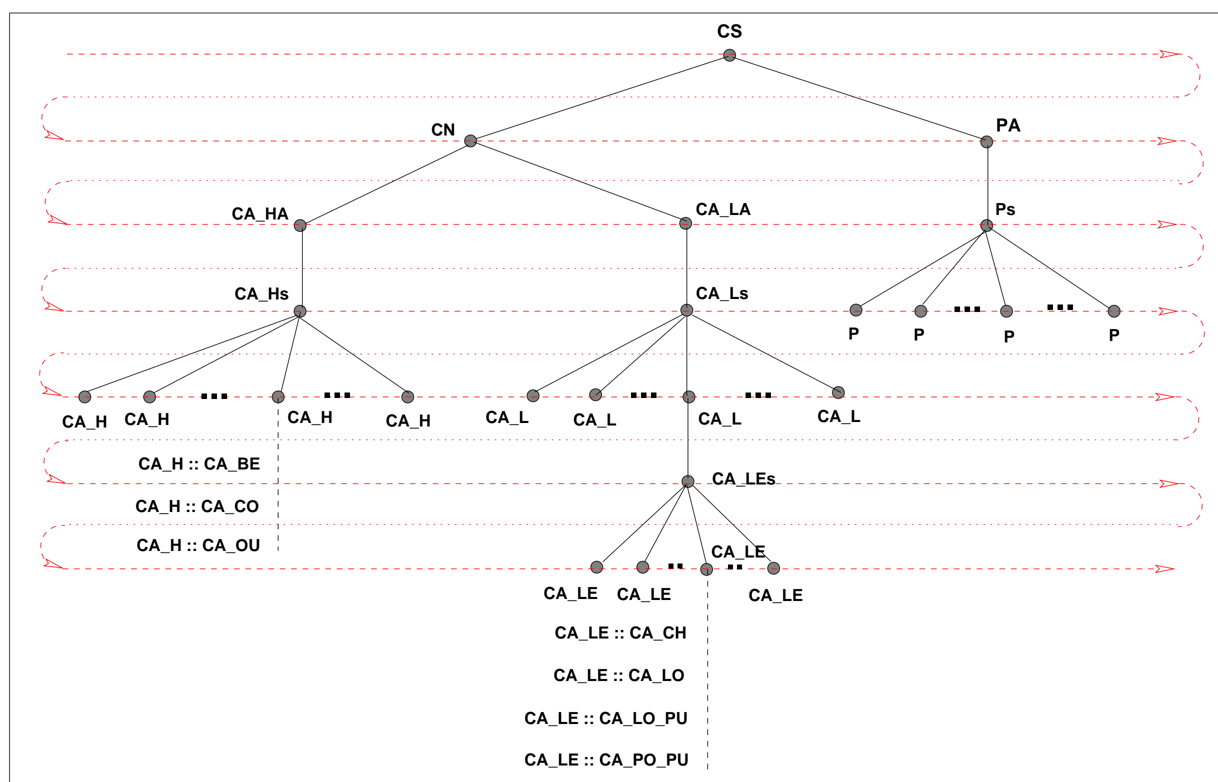


Fig. 16.15 A Breadth-first Left-to-Right [Top-down] Canal Ontology Traversal

804. the canal net, cn_{end}, [ι 781, π 448]
805. the polder aggregate, pa_{end}, [ι 782, π 448]
806. the canal net hub aggregate, ca_ha_{end}, [ι 783, π 448],
807. the canal net link aggregate, ca_la_{end}, [ι 784, π 448]
808. the set of polders, ps_{end}, [ι 785, π 448]
809. the set of hubs, ca_hs_{end}, [ι 786, π 448]
810. the set of canal links, ca_ls_{end}, [ι 787, π 448]
811. the set of polders, pos_{end}, [ι 788, π 448]
812. the set of hubs, ca_hs_{end}, [ι 786, π 448] have the following kinds of hubs: [ι 789, π 449]
813. canal begin/ends, ca_bes_{end}, [ι 790, π 449],
814. canal confluences, ca_cos_{end}, [ι 791, π 449] and
815. canal outlets, ca_ous_{end}, [ι 792, π 449],
816. the set of canal link elements, ca_les_{end} [ι 794, π 449],
817. the canal link elements [ι 795, π 449] are of the following kinds:
818. canal channels, ca_chs_{end}, [ι 796, π 449]
819. canal locks, ca_los_{end}, [ι 797, π 449]
820. canal polder pumps, $ca_po_pus_{end}$ [ι 799, π 449].

value

```
801.  cs:CS
802.  map_ends: MAP_END[181]
802.  map_ends = [
```

803. cs_{end} $\mapsto \{cs\}$, [ι 780, π 448]
804. cn_{end} $\mapsto$ {obs_CN(map_ends(cs_{end}))}, [ι 781, π 448]
805. pa_{end} $\mapsto$ {obs_PA(map_ends(cs))}, [ι 782, π 448]
806. ca_ha_{end} $\mapsto$ {obs_CA_HA(map_ends(cs))}, [ι 783, π 448]
807. ca_la_{end} $\mapsto$ {obs_CA_LA(map_ends(cn_{end}))}, [ι 784, π 448]
808. ps_{end} $\mapsto$ obs_Ps(map_ends(pa_{end}), [ι 785, π 448]
809. ca_hs_{end} $\mapsto$ obs_CA_Hs(map_ends(ca_pa_{end})), [ι 786, π 448]
810. ca_ls_{end} $\mapsto$ obs_CA_Ls(map_ends(ca_ps_{end})), [ι 787, π 448]
811. pos_{end} $\mapsto$ map_ends(ca_ps_{end})), [ι 788, π 448]
813. ca_bes_{end} $\mapsto$ map_ends(ca_hs_{end}) \ CA_BE, [ι 790, π 449]
814. ca_cos_{end} $\mapsto$ map_ends(ca_hs_{end}) \ CA_CO, [ι 791, π 449]
815. ca_ous_{end} $\mapsto$ map_ends(ca_hs_{end}) \ CA_OU, [ι 792, π 449]
816. ca_les_{end} $\mapsto$ obs_CA_Ls(map_ends(ca_ps_{end})), [ι 787, π 448]
818. ca_chs_{end} $\mapsto \cup$ map_ends(ca_les_{end}) \ CA_CH, [ι 796, π 449]
819. ca_los_{end} $\mapsto \cup$ map_ends(ca_les_{end}) \ CA_LO, [ι 797, π 449]
820. $ca_po_pus_{end}$ $\mapsto \cup map_ends(ca_les_ps_{end})$ \ CA_PO_PU][ι 799, π 449]

We, in a name-overloading fashion, define – note the ν prefix of the formula item numbers:

value
ν803. cs_{end} $= cs$, [ι 780, π 448]
ν804. cn_{end} = obs_CN(map_ends(cs_{end})), [ι 781, π 448]
ν805. pa_{end} = obs_PA(map_ends(cs)), [ι 782, π 448]
ν806. ca_ha_{end} = obs_CA_HA(map_ends(cs)), [ι 783, π 448]
ν807. ca_la_{end} = obs_CA_LA(map_ends(cn_{end})), [ι 784, π 448]
ν808. ps_{end} = obs_Ps(map_ends(pa_{end})), [ι 785, π 448]
ν809. ca_hs_{end} = obs_CA_Hs(map_ends(ca_pa_{end})), [ι 786, π 448]
ν810. ca_ls_{end} = obs_CA_Ls(map_ends(ca_la_{end})), [ι 787, π 448]
ν811. pos_{end} = map_ends(ca_ps_{end}), [ι 788, π 448]
ν813. ca_bes_{end} = map_ends(ca_hs_{end}) \ CA_BE, [ι 790, π 449]
ν814. ca_cos_{end} = map_ends(ca_hs_{end}) \ CA_CO, [ι 791, π 449]
ν815. ca_ous_{end} = map_ends(ca_hs_{end}) \ CA_OU, [ι 792, π 449]
ν816. ca_cles_{end} = obs_CA_LEs(map_ends(ca_ls_{end})), [ι 794, π 449]
ν818. ca_chs_{end} $= \cup$ map_ends(ca_les_{end}) \ CA_CH, [ι 796, π 449]
ν819. ca_los_{end} $= \cup$ map_ends(ca_les_{end}) \ CA_LO, [ι 797, π 449]
ν820. $ca_po_pus_{end}$ $= \cup map_ends(ca_les_ps_{end})$ \ CA_PO_PU, [ι 799, π 449]

16.3.3 Internal Qualities

16.3.3.1 Unique Identifiers

16.3.3.1.1 Unique Identifier Sorts

821. Canal systems have unique identifiers [ι 803, π 451].
822. Canal nets have unique identifiers [ι 804, π 452].
823. Polder aggregates have unique identifiers [ι 805, π 452].
824. Canal hub aggregates have unique identifiers [ι 806, π 452].
825. Canal link aggregates have unique identifiers [ι 807, π 452].
826. Polder sets (of polders) have unique identifiers [ι 808, π 452].
827. Canal hub sets have unique identifiers [ι 809, π 452].

[181] We invite the reader to formulate the MAP_END type. As you can see from Items 791–802, it is a map from some sort of names to sets of endurants.

828. Canal link sets have unique identifiers [ι 810, π 452].
829. Polders have unique identifiers.
830. Canal hubs have unique identifiers:
831. canal begin/ends [ι 813, π 452],
832. canal confluences [ι 814, π 452] and
833. canal outlets [ι 815, π 452].
834. Canal links have unique identifiers [ι 793, π 449].
835. Canal link element sets have unique identifiers [ι 816, π 452].
836. Canal link elements have unique identifiers:
837. canal channels [ι 818, π 452],
838. canal locks [ι 819, π 452] and
839. canal polder pumps [ι 820, π 452].

```
type
821. CS_UI [ι 780, π 448]
822. CN_UI [ι 781, π 448]
823. PA_UI [ι 782, π 448]
824. CA_HA_UI [ι 783, π 448]
825. CA_LA_UI [ι 784, π 448]
826. Ps_UI [ι 785, π 448]
827. CA_Hs_UI [ι 786, π 448]
828. CA_Ls_UI [ι 787, π 448]
829. P_UI [ι 788, π 448]
830. CA_H_UI = [ι 789, π 449]
830.     CA_BE_UI|CA_CO_UI|CA_OU
831. CA_BE_UI [ι 790, π 449]
832. CA_CO_UI [ι 791, π 449]
833. CA_OU_UI [ι 792, π 449]
834. CA_L_UI [ι 793, π 449]
835. CA_LEs_UI [ι 794, π 449]
836. CA_LE_UI = CA_CH_UI [ι 795, π 449]
836.     |CA_LO_UI|CA_LO_PU_UI|CA_PO_PU_UI
837. CA_CH_UI [ι 796, π 449]
838. CA_LO_UI [ι 797, π 449]
839. CA_PO_PU_UI [ι 799, π 449]
value
821. uid_CS: CS→ CS_UI
822. uid_CN: CN − > CN_UI
823. uid_PA: PA → PA_UI
824. uid_CA_HA: CA_HA → CA_HA_UI
825. uid_CA_LA: CA_LA → CA_LA_UI
826. uid_Ps: Ps → Ps_UI
827. uid_CA_Hs: CA_Hs → CA_Hs_UI
828. uid_CA_Ls: CA_Ls → CA_Ls_UI
829. uid_P: P → P_UI
831. uid_CA_BE: CA_BE → CA_BE_UI
832. uid_CA_CO: CA_CO → CA_CO_UI
833. uid_CA_OU: CA_OU → CA_OU_UI
834. uid_CA_L: CA_L → CA_L_UI
835. uid_CA_LEs: CA_LEs → CA_LEs_UI
837. uid_CA_CH: CA_CA_CH → CA_CH_UI
838. uid_CA_LO: CA_CA_LO → CA_CA_LO_UI
839. uid_CA_PO_PU: CA_LE → CA_PO_PU_UI
```

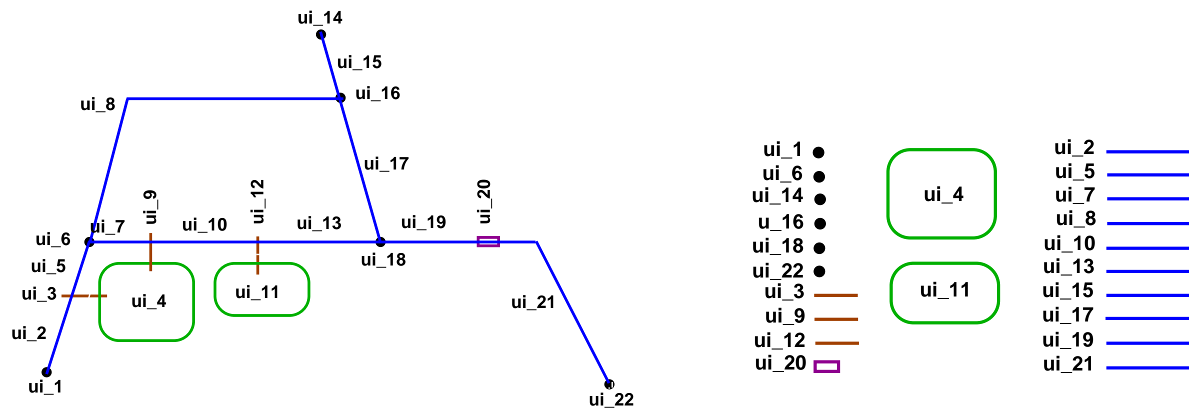


Fig. 16.16 Unique Identifiers of the Canal System of Figs. 16.13 and 16.14

16.3.3.1.2 Some Calculations

840. We can calculate the following sets of unique identifiers, seen as a map from some kind of RSL names to sets of unique identifiers:
841. the canal system singleton set of its unique identifier, [ι 803, π 451],

842. the canal net singleton set of its unique identifier, [ι 804, π 452],
843. the polder aggregate singleton set of its unique identifier, [ι 805, π 452],
844. the canal hub aggregate singleton set of its unique identifier, [ι 806, π 452],
845. the canal link aggregate singleton set of its unique identifier, [ι 807, π 452],
846. the polder set singleton set of its unique identifier, [ι 808, π 452],
847. the hub set singleton set of its unique identifier, [ι 809, π 452],
848. the link set singleton set of its unique identifier, [ι 810, π 452],
849. the set of polder unique identifiers, [ι 808, π 452],
850. the set of canal begin/end unique identifiers, [ι 791, π 449],
851. the set of canal confluence unique identifiers, [ι 791, π 449],
852. the set of canal outlet unique identifiers, [ι 791, π 449],
853. the set of canal link set unique identifiers, [ι 792, π 449],
854. the set of canal link unique identifiers, [ι 793, π 449],
855. the set of canal channel unique identifiers, [ι 796, π 449],
856. the set of canal lock unique identifiers, [ι 797, π 449],
857. the set of canal lock pump unique identifiers, [ι 798, π 449] and
858. the set of canal polder pump unique identifiers, [ι 799, π 449].

To define the next map we make use of the following generic function:

859. It applies to a set of endurants of sort X and yields the set of unique identifiers of the members of that set.

```
type
859. uid_X: X-set → X_UI-set
value
859. uid_X(xs) ≡ {uid_X(x)|x:X•x ∈ xs}
```

```
value
840.  map_uids: MAP_UI[182]
840.  map_uids = [ [ι 780, π 448]
841.    cs_uid          ↦ uid_CS(map_end(cs_end)), [ι 803, π 451]
842.    cn_uid          ↦ uid_CN(map_ends(cn_end)), [ι 804, π 452]
843.    pa_uid          ↦ uid_PA(map_ends(pa_end)), [ι 805, π 452]
844.    ca_ha_uid       ↦ uid_CA_HA(map_ends(ca_ha_end)) , [ι 806, π 452]
845.    ca_la_uid       ↦ uid_CA_LA(map_ends(ca_la_end)) , [ι 807, π 452]
846.    ps_uid          ↦ uid_P(map_ends(ps_end)), [ι 799, π 449]
847.    ca_hs_uid       ↦ uid_CA_Hs(map_ends(ca_hs_end)) , [ι 809, π 452]
848.    ca_ls_uid       ↦ uid_CA_Ls(map_ends(ca_ls_end)) , [ι 810, π 452]
849.    pos_uid         ↦ uid_P(map_ends(ps_end)) , [ι 810, π 452]
850.    ca_bes_uid      ↦ uid_BE(map_ends(ca_bes_end)), [ι 813, π 452]
851.    ca_cos_uid      ↦ uid_CO(map_ends(ca_cos_end)), [ι 814, π 452]
852.    ca_ous_uid      ↦ uid_OU(map_ends(ca_ous_end)), [ι 815, π 452]
853.    ca_les_uid      ↦ uid_CA_LEs(map_ends(ca_ls_end)), [ι 816, π 452]
855.    ca_chs_uid      ↦ uid_CA_CH(map_ends(ca_chs_end)), [ι 818, π 452]
856.    ca_los_uid      ↦ uid_CA_LO(map_ends(ca_los_end)), [ι 819, π 452]
858.    ca_po_pus_uid   ↦ uid_PO_PU(map_ends(ca_po_pus_end)) ] [ι 820, π 452]
```

[182] We invite the reader to formulate the MAP_UI type. As you can see from Items 791–802, it is a map from some sort of names to sets of unique identifiers.

16.3.3.1.3 An Axiom

860. Let end_{parts} stand for the set of all composite and atomic canal system endurants,
861. and end_{uids} the set of all their unique identifiers.
862. The number of endurants parts equals the number of endurant part unique idenifiers, is_wf_CS_Identities(cs).

value
860. end_{parts} = ∪ **rng** proper_map_ends
861. end_{uids} = ∪ **rng** map_uids
axiom
862. is_wf_CS_Identities: CS → **Bool**
862. is_wf_CS_Identities(cs) ≡ **card** end_{parts} = **card** end_{uids}

16.3.3.1.4 Another Representation of UI Values

We, in a somewhat name-overloading fashion, similarly define:

value
ν841. cs_{uid} = uid_CS(cs_{end}), [ι 803, π 451]
ν842. cn_{uid} = uid_CN(cn_{end}), [ι 804, π 452]
ν843. pa_{uid} = uid_PA(pa_{end}), [ι 805, π 452]
ν844. ca_ha_{uid} = uid_CA_HA(ca_ha_{end}) , [ι 806, π 452]
ν845. ca_la_{uid} = uid_CA_LA(ca_la_{end}) , [ι 807, π 452]
ν846. ps_{uid} = uid_Ps(ps_{end}), [ι 808, π 452]
ν847. ca_hs_{uid} = uid_Hs(ca_pa_{end}), [ι 809, π 452]
ν848. ca_ls_{uid} = uid_Ls(ca_ps_{end}), [ι 810, π 452]
ν849. pos_{uid} = uid_P(ps_{end}), [ι 811, π 452]
ν850. ca_bes_{uid} = uid_BE(ca_bes_{end}), [ι 813, π 452]
ν851. ca_cos_{uid} = uid_CO(ca_cos_{end}), [ι 814, π 452]
ν852. ca_ous_{uid} = uid_OU(ca_ous_{end}), [ι 815, π 452]
ν853. ca_cles_{uid} = uid_CA_LEs(ca_les_{end}), [ι 816, π 452]
ν854. ca_chs_{uid} = uid_CA_CH(ca_chs_{end}), [ι 818, π 452]
ν855. ca_los_{uid} = uid_CA_LO(ca_los_{end}), [ι 819, π 452]
ν857. $ca_po_pus_{uid}$ = uid_PO_PU($ca_po_pus_{end}$)[ι 820, π 452]

16.3.3.1.5 An Extract Function

863. Given 860. end_{parts} and 861. end_{uids}, we can, from any known unique identifier obtain its corresponding part:

value
863. get_part: UI → END
863. get_part(ui) ≡ **let** p:P • p ∈ end_{parts} • uid_P(p)=ui **in** p **end**; **pre**: ui ∈ **rng** end_{uids}

16.3.3.2 Mereologies

16.3.3.2.1 Mereology Types

We shall focus only on the **topological mereologies** of canal system endurants. These can be "read off" the ontology tree of Fig. 16.12 on page 450. Had we included the modeling of vessels that ply the waters of canals, then the mereologies of most canal endurants wouldalso include sets of vessel identifiers.

As for the definitions of endurants, cf. Items 780 on page 448 to 799 on page 449, and the unique identifiers, cf. Items 821 on page 453 to 839 on page 454, we define the mereologies for each category of endurants. These mereologies are defined using the unique identifiers of the endurants immediately "above" and "below" them in the ontology "tree" of Fig. 16.12 on page 450.

Common Hub and Link Types: From the unique identifier section we take over types defined in Items 823 and 824 on page 453

864. while introducing a set of their identifiers:

```
type
823.  CA_HE_UI = CA_BE_UI|CA_CO_UI|CA_OU_UI
824.  CA_LE_UI = CA_CH_UI|CA_LO_UI|LO_PU_UI|PO_PU_UI
864. CA_LE_UI_H = (CL_HE_UI|CP_LE_UI)-set
```

Canal Systems:

865. The mereology of a *canal system* is a pair of the unique identifiers of the canal net and of the polder aggregate.

```
type
865. CS_Mer = CN_UI × PA_UI
value
865. mereo_CS: CS → mereo_CS
```

Canal Nets:

866. The mereology of a *canal net* aggregate is a pair of the unique identifier of the canal system, of which it is a part, and a pair of the set of the unique identifiers of the canal hub agregate and the canal link aggregate of the net.

```
type
value
866. CN_Mer = CS_UI × (CA_HA × CA_LA)
value
866. mereo_CN: CN → CN_Mer
```

Polder Aggregates:

867. The mereology of a *polder* aggregate is a pair of the unique identifier of the canal system, of which it is a part, and the unique identifier of the polder set it "spawns".

```
type
867. PA_Mer = CS_UI × Ps_UI
value
867. mereo_PA: PA → = PA_Mer
```

Canal Hub Aggregates:

868. The mereology of a *hub aggregate* is a pair of the unique identifier of the canal net it belongs to and the hub set it "spawns".

type
868. CA_HA_Mer = CN_UI × CA_Hs
value
868. mereo_CA_HA: HA → CA_HA_Mer

Canal Link Aggregates:

869. The mereology of a *link aggregate* is a pair of the unique identifier of the canal net it belongs to and a set of the unique identifiers of the links that it "spawns".

type
869. CA_LA_Mer = CN_UI × CA_Ls
value
869. mereo_CA_LA: LA → CA_LA_Mer

Sets of Polders:

870. The mereology of a *polder set* is a pair of the unique identifier of the polder aggregate it belongs to and a set of the unique identifiers of the polders that it "spawns".

type
870. Ps_Mer = PA_UI × P_UI-**set**
value
870. mereo_Ps: Ps → Pa_Mer

Sets of Hubs:

871. The mereology of a *hub set* is a pair of the unique identifier of the hub aggregate it belongs to and a set of the unique identifiers of the hubs that it "spawns".

type
871. CA_Hs_Mer = CA_HA_UI × CA_H_UI-**set**
value
871. mereo_CA_Hs: CA_Hs → CA_Hs_Mer

Sets of Links:

872. The mereology of a *link set* is a pair of the unique identifier of the link aggregate it belongs to and a set of the unique identifiers of the links that it "spawns".

type
872. CA_Ls_Mer = CS_LA_UI × CA_L_UI-**set**
value
872. mereo_CA_Ls: Ls → CA_Ls_Mer

Polders:

873. The mereology of a*polder* is a pair of the unique identifier of the polder aggregate and a set of unique identifiers of *canal polder pump*s.

type
873. P_Mer = Ps_UI
value
873. mereo_P: P → P_Mer

Hubs:

- Hubs are not individually “recognisable” as such. They are either begin/ends, confluences or outlets; cf. Item 789 on page 449.
- The mereologies of hubs thus “translates” into the mereology of either begin/ends, confluences or outlets.

  ∞ **Begin/End**

  874. The mereology of a *canal begin/end* is a pair: the unique identifier of the canal hub set it belongs to and the singleton set of the unique identifier of the first canal link element for which it is the begin/end.

  type
  874. CA_BE_Mer = CA_Hs_UI × s:CA_LE_UI-**set** **axiom** ∀ (_,s):CA_BE_Mer • **card** s=1
  value
  874. mereo_CA_BE: CA_BE → CA_BE_Mer

  ∞ **Confluence**

  875. The mereology of a *canal confluence* is a pair: the unique identifier of the canal hub set it belongs and set of two or more canal element unique identfiers, one for each canal link incident upon the canal confluence.

  type
  875. CA_CO_Mer = CA_Hs_UI × s:CL_E_UI-**set** **axiom** ∀ (_,s):CA_CO_Mer • **card** s≥2
  value
  875. mereo_CA_CO: CA_CO → mereo_CA_CO

  ∞ **Outlet**

  876. The mereology of an *outlet* is a pair: the unique identifier of the canal hub set it belongs and the singleton set of the unique identifier of the last canal link element for which it is the outlet.

  type
  876. CA_OU_Mer = CA_Hs_UI × s:CL_E_UI-**set** **axiom** ∀ (_,s):CA_OU_Mer • **card** s=1
  value
  876. mereo_CA_OU: CA_OU → CA_OU_Mer

Canal Links:

877. The mereology of a *canal link* are triples: the unique identifier of the canal link set to which it belongs, a two element set of the canal hubs that the link is linking, and a list (i.e., an ordered sequence) of the unique identifiers of the one or more canal link elements of the link.

type
877. CA_L_Mer = CA_Ls_UI × CA_H_UI-**set** × s:CA_LE_UI*
877. **axiom** ∀ (_,s,l):CA_L_Mer • **card** s=2 ∧ **len** l ≥ 1
value
877. mereo_CA_L: CA_L → CA_L_Mer

Sets of Canal Link Elements:

878. The mereology of any *canal link element* includes a pair: the unique identifier of the canal link to which it belongs and a two element set, one element is the unique identifier of either a canal hub or a[another] canal link element, the second element is the unique identifier of either a [next] canal link element or a canal hub – these we call CLE_UI_P.

type
878. CA_LE_Mer_Common = CL_UI × seuis:(CA_H_UI|CA_LE_UI)-**set**
878. **axiom** ∀ (clui,chluis):CA_LE_Mer • **card** chluis = 2

Canal Link Elements:

- Canal link elements are not individually "recognisable" as such. They are either canal channels, canal locks, canal locks with pumps or are canal polder pumps; cf. Item 795 on page 449.

  - **Canal Channels**
  879. The mereology of any *canal channel* is as the mereology included in any canal element mereology, cf. Item 878.

  type
  879. CA_CH_Mer = se:CA_LE_Mer_Common
  value
  879. mereo_CA_CH: CA_CH → CA_CH_Mer

  - **Canal Locks**
  880. The mereology of any *canal lock* is as the mereology included in any canal element mereology, cf. Item 878.

  type
  880. CA_LO_Mer = se:CA_LE_Mer_Common
  value
  880. mereo_CA_LO: CA_LO → CA_LO_Mer

  - **Canal Polder Pumps**
  881. The mereology of any *canal polder pump*, is a pair: in addition to the mereology of any canal link element – which is now first element of the pair, has the second element being the unique identifier of a polder.

  type
  881. CA_PO_PU_Mer = se:CA_LE_Mer_Common × P_UI
  value
  881. mereo_CA_PO_PU: CA_PO_PU → CA_PO_PU_Mer

16.3.3.2.2 The Mereology Axiom

It is You, the domain analysers & describers, who decide on the mereologies of a domain ! You may wish to emphasize topological aspects of a domain; or you may wish to emphasize "co-ordination" relations between topologically "unrelatable" parts; or you may choose a mix of these; it all, also, depends on which aspects You wish to emphasize when transcendentally deducing [certain] parts into behaviours. Therefore the **mereology axiom** to be expressed reflects Your choice. Here we have chosen to emphasize the topological aspects of the canal domain. We use the term *well-formedness* of the mereology of an endurant. But do not be mislead ! It is not a property that we impose on the domain endurant. It is a fact. We cannot escape from that fact. Later, in the requirements engineering of a possible software product for a domain, You may decide to implement data structures to reflect mereologies, in which case you shall undoubtedly need to prove that your choice of data structures, their initialisation and update does indeed satisfy the axioms of the domain model.

882. For a canal system to be mereologically, cum topologically well-formed means that the canal system mereology is well-formed.

axiom
882. is_wf_CS_Mereology(cs_{end})

16.3.3.2.3 Well-formed Mereologies

Canal Systems:

883. Canal system well-formednes, is_wf_CS_Mereology,
   a. besides the appropriateness of its own mereology,
   b. is secured by the well-formedness of the canal net aggregate and polder aggregate, is_wf_-CN_Mereology and is_wf_PA_Mereology.

value
883. is_wf_CS_Mereology: CS → **Bool**
883. is_wf_CS_Mereology(cs_{end}) ≡
883a. **let** (cn_ui,pa_ui) = mereo_CS(cs_{end}) **in** [ι 865, π 457]
883a. cn_ui = cn_{uid} ∧ pa_ui = pa_{uid} **end** ∧ [ι 842 π 455, ι 843 π 455]
883b. is_wf_CN_Mereology(cn_{end})∧is_wf_PA_Mereology(pa_{end})

Canal Nets:

884. Well-formedness of canal nets, is_wf_CN_Mereology,
   a. besides the appropriateness of its own mereology, is_wf_CS_Mereology,
   b. is secured by the well-formedness of link and the hub aggregates, is_wf_CA_LA_Mereology, and all links, is_wf_CA_HA_Mereology.

value
884. is_wf_CN_Mereology: CN → **Bool**
axiom
884. is_wf_CN_Mereology(cn_{end}) ≡
884a. **let** (cn_ui,ca_ha_ui,ca_la_ui) = mereo_CN(cn_{end}) **in** [ι 866, π 457]
884a. cn_ui = cn_{uid} ∧ ca_ha_ui = ca_ha_{uid} ∧ ca_la_ui = ca_la_{uid} **end** ∧[ι 842 π 455, ι 844 π 455, ι 845 π 455]
884b. is_wf_CA_HA_Mereology(ca_ha_{end})∧is_wf_CA_LA_Mereology(ca_la_{end})

Polder Aggregates:

885. Well-formedness of polder aggregates, is_wf_PA_Mereology,
   a. besides the appropriateness of its own mereology,
   b. is secured by the well–formedness of the polder set is_wf_Ps_Mereology.

```
type
value
885.  is_wf_PA_Mereology: PA → Bool
885.  is_wf_PA_Mereology(pa_end) ≡
885a.     let (cs_ui,ps_ui) = mereo_PA(pa_end) in [ι 867, π 457]
885a.     cs_ui = cs_uid ∧ ps_ui = ps_uid end ∧[ι 841 π 455, ι 846 π 455]
885b.     is_wf_Ps_Mereology(ps_end)
```

Canal Hub Aggregates:

886. Well-formedness of canal hub aggregates, is_wf_CA_HA_Mereology,
   a. besides the appropriateness of its own mereology,
   b. is secured by the well-formedness of its set of hubs.

```
value
886.  is_wf_CA_HA_Mereology: CA_HA → Bool
886.  is_wf_CA_HA_Mereology(hub) ≡
886a.     let (cnui,cahsui) = mereo_CA_HA(hub) in [ι 868, π 458]
886a.     cnui = cn_uid ∧ cahsui = ca_hs_uid end ∧[ι 842 π 455, ι 847 π 455]
886b.     is_wf_CA_Hs(ca_hs_end)
```

Canal Link Aggregates:

887. Well-formedness of canal hub aggregates, is_wf_CA_HA_Mereology,
   a. besides the appropriateness of its own mereology,
   b. is secured by the well-formedness of its set of links.

```
value
887.  is_wf_CA_LA_Mereology: CA_HA → Bool
887.  is_wf_CA_LA_Mereology(la) ≡
887a.     let (cnui,clsui) = mereo_CA_LA(la) in [ι 869, π 458]
887a.     cnui = cn_uid ∧ clsui = cls_uid end ∧ [ι 842 π 455, ι 848 π 455]
887b.     is_wf_CA_Ls(cls_end)
```

Sets of Polders:

888. Well-formedness of sets of polders, is_wf_Ps_Mereology,
   a. besides the appropriateness of its own mereology,
   b. is secured by the well-formedness of its individual polders.

```
value
888.  is_wf_Ps_Mereology: Ps → Bool
888.  is_wf_Ps_Mereology(ps) ≡
888a.     let (paui,puis) =  mereo_Ps(ps_end) in [ι 870, π 458]
888a.     paui = ca_pa_uid ∧ puis = ca_pos_uid end ∧ [ι 842 π 455, ι 843 π 455]
888b.     ∀ po:PO • po ∈ pos_end ⇒ is_wf_P(po)
```

Sets of Hubs:

889. Well-formedness of a hub set is_wf_CA_Hs_Mereology,
   a. besides the appropriateness of its own mereology,
   b. is secured by the well-formedness of its individual hubs.

value
889. is_wf_CA_Hs_Mereology: CA_Hs → **Bool**
889. is_wf_CA_Hs_Mereology(ca_hs_{end}) ≡
889. **let** (cahaui,cahuis) = mereo_CA_Hs(ca_bes_{end}∪ca_cos_{end}∪ca_ous_{end}) **in** [ι 871, π 458]
889. cahaui = ca_ha_{uid} ∧ cahuis = ca_hs_{uid} **end** ∧ [ι 844 π 455, ι 847 π 455]
889a. ∀ hub:CA_H • hub ∈ ca_hs_{end} ⇒ is_wf_CA_H(hub)

Sets of Links:

890. Well-formedness of sets of links
   a. besides the appropriateness of its own mereology,
   b. is secured by the well-formedness of all of its individual links.

value
890. is_wf_CA_Ls_Mereology: mereo_CA_Ls → **Bool**
890. is_wf_CA_Ls_Mereology(ca_ls_{end}) ≡
890a. **let** (cs_la_ui,ca_ls_ui) = mereo_CA_Ls(ca_ls_{end}) **in** [ι 872, π 458]
890a. cs_la_ui = ∧ ca_ls_ui = ca_ls_{uid} **end** ∧ [ι 845 π 455, ι 848 π 455]
890b. ∀ link:CA_L • link ∈ ca_ls_{end} ⇒ is_wf_CA_L(link)

Polders:

891. Well-formedness of polders, is_wf_P_Mereology, depends jst on the appropriateness of its own mereology.

value
891. is_wf_Mereology_Polder: mereo_P → **Bool**
891. is_wf_Mereology_Polder(p) ≡
891. **let** ps_ui = mereo_P(p) **in** [ι 873, π 458]
891. ps_ui = ps_{uid} **end** ≡ [ι 846, π 455]

Hubs:

789 Hubs are not individually "recognisable" as such. They are either begin/ends, confluences or outlets; cf. Item 789 on page 449.
892. The well-formedness of hubs thus "translates" into the well-formedness of either begin/ends, confluences or outlets.

type
789. CA_H == CA_BE|CA_CO|CA_OU
value
892. is_wf_Mereology_H: H → **Bool**
892. is_wf_Mereology_H(h) ≡
892. is_CA_BE(h) → is_wf_Mereology_CA_BE(h),
892. is_CA_CO(h) → is_wf_Mereology_CA_CO(h),
892. _ → is_wf_Mereology_CA_OU(h)

- **Begin/End**

893. Well-formedness of the mereology of begin/end hubs, is_wf_Mereology_CA_BE, depends just on the appropriateness of their own mereology.

```
value
893. is_wf_Mereology_CA_BE: CA_BE → Bool
893. is_wf_Mereology_CA_BE(be) ≡ ≡
893.    let (cahsui,cleuis) = mereo_CA_BE(be) in [ι 874, π 459]
893.    cahsui ∈ ca_hs_uid ∧ cleuis ∈ ca_les_uid end [ι 847 π 455, ι 853 π 455]
```

- **Confluence**

894. Well-formedness of the mereology of confluence hubs, is_wf_Mereology_CA_CO, depends just on the appropriateness of their own mereology.

```
value
894. is_wf_Mereology_CA_CO: CA_CO → Bool
894. is_wf_Mereology_CA_CO(co) ≡
894.    let (cahsui,cleuis) = mereo_CA_CO(co) in [ι 875, π 459]
894.    cahsui ∈ ca_hs_uid ∧ cleuis ∈ ca_les_uid end [ι 847 π 455, ι 853 π 455]
```

- **Outlet**

895. Well-formedness of the mereology of outlet hubs, is_wf_Mereology_CA_OU, depends just on the appropriateness of their own mereology.

```
895. is_wf_Mereology_CA_OU: CA_CO → Bool
895. is_wf_Mereology_CA_OU(ou) ≡
895.    let (cahsui,cleuis) = mereo_CA_OU(ou) in [ι 876, π 459]
895.    cahsui ∈ ca_hs_uid ∧ cleuis ∈ ca_les_uid end [ι 847 π 455, ι 853 π 455]
```

Canal Links:

896. The well-formedness of canal links depends on
   a. the appropriateness of its own mereology, that is, that its unique identifier references are indeed to canal system identifiers,
   b. the well-formedness of the set of link elements that can be observed from a canal link, that is, that they form a sequence of canal link elements – connecting two canal hubs, and
   c. the ("remaining") well-formedness of the canal link elements.

```
896.  is_wf_Mereology_CA_L: CA_L → Bool
896.  is_wf_Mereology_CA_L(link) ≡
896a.    let (calsui,cahuis,caleuil) = mereo_CAL_L(link) in [ι 876, π 459]
896a.    calsui = ∧ cahuis = ∧ caleuil = ∧ [ι 847 π 455, ι 853 π 455, ι 853 π 455]
896b.    wf_Link_Es(obs_CA_LEs(link))(cahuis)(caleuil) ∧
896c.    ∀ le:CA_LE•le ∈ obs_CA_LEs(link) ⇒ is_wf_Mereology_CA_LE(le) end
```

Well-formed Sets of Canal Link Elements:
The introduction of the wf_Link_Es predicate represents a slight deviation from the introduction of the usual is_wf_Mereology predicates.

897. The wf_Link_Es predicate applies to a set of link elements, link, and a unique identifier list, uil, of unique link element identifiers. The predicate holds if the set, link: CA_LE-**set**, of link elements not only can be ordered in the sequence indicated by uil.

a. The length of the unique identifier list, uil, must match the cadinality of the set link.
b. Let linkl be the list of link elements prescribed by uil.
   i. The elements of a list "alternate" as follows:
      1. canal locks have either canal hubs or canal channels as immediate neighbours[183];
      2. Canal locks and polder pumps cannot be adjacent.
      3. It is allowed for two or more canal channels to be adjacent.
      4. Thus canal links may have either canal channels of canal locks as first/last elements.
   ii. Now there are the following cases of "neighbour" mereologies to observe:
   iii. For a singleton list, linkl, its only element must connect the two distinct hubs identified in cahuis.
   iv. for a two-element unique identifier list, ⟨luil,ruil⟩ one of their common mereology identifiers are shared, i.e., their elements are connected, and the other common mereology identifiers are those of canal hubs, i.e. end-points.
   v. For lists of length three or more elements
      1. the first and last elements must have end-points,
      2. and for all elements in-between it must be the case that the neighbour identifiers
      3. of the previous and the following link elements
      4. must share identifiers with the quantified element
      5. and share identifier with

```
value
897.      wf_Link_Es: CA_LE-set → CA_H_UI-set × CA_LE_UI* → Bool
897.      wf_Link_Es(link)(euis:{l_ca_h_ui,r_ca_h_ui})(uil) ≡ [ axiom card euis = 2 ]
897a.        card link = len uil ∧
897b.        let linkl = ⟨ le | i:Nat, ce:C_LE•
897b.                        1≤i≤len uil∧le∈link∧uid_CA_LE(le)=uil[ i ]⟩ in
897(b)i.       is_wf_neighbours(linkl) ∧
897(b)ii.      case linkl of
897(b)iii.       ⟨ui⟩ →
897(b)iii.           cahauis = seuis(mereo_LE(get_part(ui))),
897(b)iii.           axiom: let {lui,rui}=seuis(mereo_LE(get_part(ui))) in
897(b)iii.                 wf_end_points(lui,rui)(euis) end
897(b)iv.        ⟨lui,rui⟩ →
897(b)iv.            wf_end_points(lui,rui)(euis),
897(b)v.         ⟨lui⟩⌢link⌢⟨rui⟩ → [ axiom: len linkl ≥ 3, i.e., link≠⟨⟩ ]
897(b)v1.            wf_end_points(lui,rui)(euis) ∧
897(b)v2.            ∀ i:Nat • 1<i<len linkl ⇒
897(b)v3.              let {uim1,uim1}=seuis(mereo_CA_LE(get_part(linkl[ i−1 ]))),
897(b)v3.                  {uip1,uip1}=seuis(mereo_CA_LE(get_part(linkl[ i+1 ]))) in
897(b)v4.              axiom: lui∈{uim1,uim1} ∧ rui∈{uip1,uip1}
897(b)v5.              let uism1={uim1,uim1}\{lui}, uisp1={uip1,uip1}\{rui} in
897(b)v5.              link[ i ] = uism1 = uisp1 end
897.       end end  end

897(b)i.       is_wf_neighbours: CA_LE* → Bool
897(b)i.       is_wf_neighbours(linkl) ≡
897(b)i.         ∀ i:Nat • {i,i+1} ⊆ inds linkl ⇒
897(b)i1.            is_CA_LO_UI(linkl[ i ]) ⇒ ~(is_CA_LO_UI(linkl[ i+1 ])∨is_PO_PU_UI(linkl[ i+1 ]))

897(b)iv.   is_shared: UI × UI-set × UI-set → Book
```

[183] That is, a sequence of locks, such as illustrated in Fig. 16.5 on page 442, is here considered a single lock whose attributes "reveals" its "multiplicity".

```
897(b)iv.  is_shared(ui,luis,ruis) ≡ ui ∈ luis ∩ ruis
897(b)iv.
897(b)iv.  shared: UI-set × UI-set → UI
897(b)iv.  shared(luis,ruis) ≡ luis ∩ ruis
897(b)iv.     pre: ∃ ui:UI • is_shared(ui,luis,ruis)
897(b)iv.
897(b)iv.  wf_end_points: (UI×UI) → CA_H_UI-set → Bool
897(b)iv.  wf_end_points(lui,rui)(euis) ≡ [ axiom: card euis = 2 ]
897(b)iv.        let {llui,lrui} = seuis(mereo_LE(get_part(lui))),
897(b)iv.            {rlui,rrui} = seuis(mereo_LE(get_part(rui))) in
897(b)iv.        if ∃ ui:CA_LE_UI • is_shared(ui,{llui,lrui},{rlui,rrui})
897(b)iv.            then let ui = shared({llui,lrui},{rlui,rrui}) in
897(b)iv.                 {llui,lrui,rlui,rrui}\{ui}=euis ∧
897(b)iv.                 euis⊆ca_bes_uid∪ca_cus_uid∪ca_ous_uid end
897(b)iv.            else false end end,
```

Canal Link Elements:

...

MORE TO COME

- **Canal Channels**

898. is_wf_CA_CH_Mereology,
   a.
   b.
   c.

898.
898.
898a.
898b.
898c.

- **Canal Locks**

899. is_wf_CA_LO_Mereology,
   a.
   b.
   c.

899.
899.
899a.
899b.
899c.

- **Canal Polder Pumps**

900. a.
   b.
   c.

900.
900.
900a.
900b.
900c.

16.3.3.3 Routes

16.3.3.3.1 Preliminaries

901. By an end-identifier we mean the unique identifier of a begin/end or an outlet.
902. By a middle-identifier we mean the unique identifier of a confluence, channel, lock, lock with pump or a polder pump.
903. By a unit identifier we mean either an end-identifier or a middle-identifier.
904. By a canal route we mean a sequence of one or more unique identifiers of atomic canal entities, two if one of the identifiers is that of a begin/end or an outlet unit.

Notice that adjacent canal route identifiers be distinct. But a triplet of adjacent canal route identifiers may have the same first and last elements. It is allowed that a route, so-to-speak, goes forward and backward. There is, in a sense, no preferred directions in canal systems.

```
type
901.  E_UI = CA_BE_UI|CA_OU_UI
902.  M_UI = CA_CO_UI|CA_CH_UI|CA_LO_UI|CA_LO_PU_UI|CA_PO_PU_UI
903.  UI = E_UI|M_UI
904.  CR = UI*
axiom
904.  ∀ cr:CR,i:Nat • {i,i+1}⊆inds cr ⇒ cr[i]≠cr[i+1]
```

905. Let uid_MU be a "common" unique identifier observer of middle units.
906. Let mereo_MU be a "common" mereology observer of middle units other than polder pumps.
907. From middle units, i.e., confluences, channels, locks, lock with pumps and polder pumps we can extract simple, one-, two- or three element canal routes.

```
type
905.  uid_MU = uid_CA_CO|uid_CA_CH|uid_CA_LO|uid_CA_LO_PU|uid_CA_PO_PU
906.  mereo_MU = mereo_CA_CO|mereo_CA_CH|mereo_CA_LO|uid_CA_LO_PU
907.  MU = CA_CO|CA_CH|CA_LO|CA_LO_PU|CA_PO_PU
value
907.  xtr_M_UI_CRs: MU → CR-set
907.  xtr_M_UI_CRs(mu) ≡
907.    let mu_ui = uid_MU(mu),
907.      {ui1,ui2} =
907.        is_CA_PO_PU(mu) →
907.          let (_,cuis,_) = mereo_CA_PO_PU(mu) in cuis end
907.        _ → let (_,cuis) = mereo_MU(mu) in cuis end
907.    {⟨mu_ui⟩,⟨ui1,mu_ui⟩,⟨ui2,mu_ui⟩,⟨mu_ui,ui1⟩,⟨mu_ui,ui2⟩,⟨ui1,mu_ui,ui2⟩,⟨ui2,mu_ui,ui1⟩}
907.    end
```

16.3.3.3.2 All Routes

908. By means of xtr_M_UI_CRs we can extract, xtr_CRs(mus), the infinite set of canal routes from any set, mus, of middle canal elements.
909. First we calculate initial, i.e., simple routes.
910. Then for every two routes, a "left" and a "right" route, in the set of routes beng recursively defined, such that the last element of the left route is identical to the first element of the right route, the route formed by concatenating the left and right routes "around" the shared element is a route.
911. The set of routes of a canal system is the least fix-point soluion the the equation of Item 910.
912. No two adjacent identifiers are the same.

```
type
907.  MU = CA_CO|CA_CH|CA_LO|CA_LO_PU|CA_PO_PU
value
908.  xtr_CRs: MU-set → CR-infset
908.  xtr_CRs(mus) ≡
909.    let icrs = ∪{xtr_M_UI_CRs(mu)|mu:MU•mu ∈ mus} in
910.    let crs = icrs ∪ {lr⌢⟨ui⟩⌢rl|lr,⟨ui⟩,rr:CR•{lr⌢⟨ui⟩,⟨ui⟩⌢rr}∈crs} in
911.    crs
912.    axiom: ∀ cr:CR, i:Nat • cr isn crs ∧ {i,i+1}⊆inds cr ⇒ cr[i]≠cr[i+1]
908.    end end
```

16.3.3.3.3 Connected Canal Systems

913. Canal systems, such as we shall understand them, are connected.
914. That is, there is a route from any canal element to any other other canal element.
915. Let mus be the set of all middle elements of a canal system.
916. Let rs be the infinite set of all routes of mus.
917. Now, for any two unique identifiers of middle elements there must be a route in rs.

```
value
913.  is_connected_CS: CS → Bool
914.  is_connected_CS(cs) ≡ in
915.    let mus = ca_cos_end∪ca_chs_end∪ca_los_end∪ca_lo_pus_end∪ca_po_pus_end in
916.    let rs = xtr_CRs(mus) in
917.    ∀ ui:M_UI • {fui,tui}⊆uid_MU(mus) ⇒ ∃ r:R • r ∈ rs and r[1]=r[len r]
914.    end end
```

16.3.3.3.4 A Canal System Axiom

918. Canal systems are connected.

```
axiom
918.  ∀ cs:CS • is_connected_CS(cs)
```

16.3.3.4 Attributes

We shall treat the issue of canal part attributes, not, as is usual, one-by-one, sort-by-sort, but more-or-less "collectively", across canal hubs and links. And we do so category-by-category of attribute kinds: spatial, temporal and other.

16.3.3.4.1 Spatial and Temporal Attributes

Spatial Attributes:

Natural and artefactual, that is, mane-made endurants reside in space. We have dealt with space, i.e., $\mathbb{SPACE}$, in [55, *Sects. 2.2 and 3.4*]. Subsidiary spatial concepts are those of $\mathbb{VOLUME}$, $\mathbb{AREA}$, $\mathbb{CURVE}$ (or $\mathbb{LINE}$), and $\mathbb{POINT}$. All canal system endurants possess, whether we choose to model them or not, such spatial attributes. We shall not here be bothered by any representation, let alone computational representations, of spatial attributes. They are facts. Any properties that two $\mathbb{AREA}$s, a_i and a_j may have in common – like **bordering**, **overlapping disjoint** or **properly contained** – are facts and should, as such be expressed in terms of **axiom**s. They are not properties that can, hence must, be proven. Once a *domain description*, involving spatial concepts is the base for a *requirements prescription*, then, if these spatial concepts are not *projected* out of the evolving requirements, they must, eventually, be prescribed – or assumed to have – computable representations. In that case axioms concerning spatial quantities are turned into **proof obligations** that must, eventually, be **discharged**.

Let us establish the following spatial attributes, common to all canal parts:

919. Location: A single $\mathbb{POINT}$ in $\mathbb{SPACE}$ characterised by its longitude, latitude and altitude, the latter height above or depth below sea level, including 0. How these are measured is of no concern in this model.
920. Extent: An $\mathbb{AREA}$, i.e., a plane in $\mathbb{SPACE}$, i.e., a dense set of $\mathbb{POINT}$s according to some topology.
921. Volume: A proper subset $\mathbb{SPACE}$, i.e., a three dimensional dense set of $\mathbb{POINT}$s $\mathbb{SPACE}$, according to some topology.
922. The Location of a canal part is always **embedded** in its Extent.
923. The Extent of a canal part is always **embedded** in its Volume

```
type
919. Location
920. Extent
921. Volume
value
919. attr_Location: CS|CN|PA|CA_HA|CA_LA|CA_LE → Location
920. attr_Extent: CS|CN|PA|CA_HA|CA_LA|CA_LE → Extent
921. attr_Volume: CS|CN|PA|CA_HA|CA_LA|CA_LE → Volume
921. is_embedded: Location × Extent → Bool, is_embedded: Extent × Volume → Bool
axiom
922. ∀ e:(CS|CN|PA|CA_HA|CA_LA|CA_LE)•is_embedded(attr_Location(e),attr_Extent(e))
923. ∀ e:(CS|CN|PA|CA_HA|CA_LA|CA_LE)•is_embedded(attr_Extent(e),attr_Volume(e))
```

Let us establish the following ***s, common to some canal parts:

924. Let us addume the sort notions of Latitude, Longitude and Altitude,
925. And let us assume "sea level" Altitude value "0".
926. A projected extent is an extent all of whose **altiude** elements are zero (0), i.e., "at sea level".
927. We assume functions, latitude, logitude, altitude, that extract respective elements of a point.

928. No two distinct hubs and link elements can share neither location, area nor volume – so they are **disjoint**.

Canal channels may share projected extents.

```
type
924. Latitude, Longitude, Altitude
value
925. ″𝟘″: Altitude
926. projected_Extent: Extent → Extent
927. latitude: ℙ𝕆𝕀ℕ𝕋 → Latitude, longitude: ℙ𝕆𝕀ℕ𝕋 → Longitude, altitude: ℙ𝕆𝕀ℕ𝕋 → Altitude
axiom
928. ∀ e,e′:(CS|CN|PA|CA_HA|CA_LA|CA_LE): e≠e′ ⇒ disjoint(attr_Volume(e),attr_Volume(e′))
924. ∀ e,e′:CA_CH •
925.
926.
```

Temporal Attributes:

Natural and artefactual, that is, mane-made endurants reside in time. We have dealt with space, i.e., 𝕋𝕀𝕄𝔼, in [55, *Sects. 2.2 and 3.5*]. Subsidiary spatial concepts are those of 𝕋𝕀𝕄𝔼 and 𝕋𝕀𝕄𝔼 𝕀ℕ𝕋𝔼ℝ𝕍𝔸𝕃s. All canal system endurants possess, whether we choose to model them or not, such temporal attributes. We shall not here be bothered by any representation, let alone computational representations, of temporal attributes. They are facts. Any properties that two 𝕋𝕀𝕄𝔼 𝕀ℕ𝕋𝔼ℝ𝕍𝔸𝕃s, ti_i and ti_j may have in common, like **bordering** or **overlapping**, are facts and should, as such be expressed in terms of **axiom**s[184]. They are not properties that can, hence must, be proven. Once a *domain description*, involving temporal concepts is the base for a *requirements prescription*, then, if these temporal concepts are not *projected* out of the evolving requirements, they must, eventually, be prescribed – or assumed to have – computable representations. In that case axioms concerning temporal quantities are turned into **proof obligations** that must, eventually, be **discharged**.

Event Attributes

Some events can, for example, be talked about, by humans. They, so-to-speak, belong to an event-category: *"von hörensagen"*. Examples are: *"a canal lock opened at time τ "*; *"a polder pump stopped pumping at time τ' "*; and *"a canal vessel passed a certain canal channel point at time τ'' "*. Let refer to the event as e:E. If, for an endurant, p of sort P, they are relevant to an analysis & description of a domain, then they must be noted, for example in the form of an attribute named, say, history_E:

type history_E = 𝕋𝕀𝕄𝔼 $\overrightarrow{m}$ E

Continuous Time Attributes

Mostly one models discrete time phenomena. But often phenomena are continuous time varying. Examples are: *"the canal water level"*, *"the canal water temperature"*, and *"the position of a vessel along a canal"*. If, for an endurant, p of sort P, such a phenomenon, e:E, is relevant to an analysis & description of a domain, then it must be noted, for example in the form of an attribute named, say, history_E:

type history_E = 𝕋𝕀𝕄𝔼 → E

[184] We refer here to the 𝕋𝕀𝕄𝔼 and 𝕋𝕀𝕄𝔼 𝕀ℕ𝕋𝔼ℝ𝕍𝔸𝕃 operators of [55, *Sects. 2.2 and 3.5*]

16.3.3.4.2 Canal System, Net and Polder Attributes

929. Canal systems have location and extent.
930. So do canal nets and
931. polder aggregates.
932. Canal nets and polder aggregates are **bordering**[185].
933. Canal nets and polder aggregates are properly **embedded**[186] in canal systems.
934. Etc.

```
value
929. attr_Location: CS → Location; attr_Extent: CS → Extent
930. attr_Location: CN → Location; attr_Extent: CN → Extent
931. attr_Location: PA → Location; attr_Extent: PA → Extent
axiom
932. ∀ cs:CS • are_bordering(attr_Extent(obs_CN(cs)),attr_Extent(obs_PA(cs)))
933. ∀ cs:CS • is_embedded(attr_Extent(obs_CN(cs)),cs)∧is_embedded(cs,attr_Extent(obs_PA(cs)))
934. ...
```

16.3.3.4.3 Canal Hub and Link Attributes

Two kinds of attributes shared across hubs and links, therefore their elements, stand out: *water level*s and *ambient* and *water temperature*s.

Water Temperatures:

935. Let there be given a notion of water temperature.

Generally, over time, one can associate with any canal hub and link element,

936. high,
937. normal and
938. low water

water temperatures, and specifically, at any time,

939. current water temperatures.

```
type
935. Wa_Temp
936. Hi_Temp = 𝕋𝕀𝕄𝔼 → Wa_Temp
937. No_Temp = 𝕋𝕀𝕄𝔼 → Wa_Temp
938. Lo_Temp = 𝕋𝕀𝕄𝔼 → Wa_Temp
939. Cu_Temp = Wa_Temp
value
936. attr_Hi_Temp: H → Hi_Temp, attr_LE_Temp: LE → Hi_Temp
937. attr_No_Temp: H → No_Temp, attr_LE_Temp: LE → No_Temp
938. attr_Lo_Temp: H → Lo_Temp, attr_LE_Temp: LE → Lo_Temp
939. attr_Cu_Temp: H → Cu_Temp, attr_LE_Temp: LE → Cu_Temp
```

[185] We leave it to a chosen Topology to define the **are_bordering** predicate

[186] We leave it to a chosen Topology to define the **is_embedded** predicate

The Hi_Temp, No_Temp and Lo_Temp attributes are normally continuous functions over time. They are facts. One does not have to "go out" and measure them! We do not have to think of representations for the Hi_Temp, No_Temp and Lo_Temp attributes.

Water Levels:

940. Let there be given a notion of water level.

Generally, over time, one can associate with any canal hub and link element,

941. high,
942. normal and
943. low

water levels, and specifically, at any time,

944. current water level.

type
940. Wa_Lev
941. Hi_WL = $\mathbb{TIME}$ → Wa_Lev
942. No_WL = $\mathbb{TIME}$ → Wa_Lev
943. Lo_WL = $\mathbb{TIME}$ → Wa_Lev
944. Cu_WL = Wa_Lev
value
941. attr_Hi_WL: H → Hi_WL, attr_LE_WL: LE → Hi_WL
942. attr_No_WL: H → No_WL, attr_LE_WL: LE → Hi_WL
943. attr_Lo_WL: H → Lo_WL, attr_LE_WL: LE → Hi_WL
944. attr_Cu_WL: H → Cu_WL, attr_LE_WL: LE → Hi_WL

The Hi_WL, No_WL and Lo_WL attributes are normally continuous functions[187] over time. Remarks on Hi_WL, No_WL and Lo_WL attributes similar to those of the Hi_Temp, No_Temp and Lo_Temp attributes as to continuity and representations apply.

MORE TO COME

16.3.3.5 Well-formedness of Attributes

945. There is a predicate, is_wf_CS_Attributes.
946.
947.
948.
949.

MORE TO COME

16.3.4 Speculations

TO BE WRITTEN

[187] – barring cyclones, tornados and the like!

16.4 Conclusion

TO BE WRITTEN

Part III
Two Postlude “Domain” Examples

Chapter 17
A Stock Exchange [January 2010]

Contents

17.1 Introduction 477
17.2 The Problem 477
17.3 A Domain Description 478
17.3.1 Market and Limit Offers and Bids 478
17.3.2 Order Books 479
17.3.3 Aggregate Offers 479
17.3.4 The TSE Itayose "Algorithm" 481
17.3.5 Match Executions 482
17.3.6 Order Handling 482
17.4 Tetsuo Tamai's IEEE Computer Journal Paper 483

I thank Prof. Tetsuo Tamai, Tokyo University, for commenting on an early version of this chapter: clarifying issues and identifying mistakes and typos.

This chapter was begun on January 24. It is being released, first time, January 28.

17.1 Introduction

This chapter shall try describe: narrate and formalise some facets of the (now "old"[188]) stock trading system of the TSE: Tokyo Stock Exchange (especially the 'matching' aspects).

17.2 The Problem

The reason that I try tackle a description (albeit of the "old" system) is that Prof. Tetsuo Tamai published a delightful paper [152, IEEE Computer Journal, June 2009 (vol. 42 no. 6) pp. 58-65)], Social Impact of Information Systems, in which a rather sad story is unfolded: a human error key

[188] We write "old" since, as of January 4, 2010, that 'old' stock trading system has been replaced by the so-called arrowhead system. We refer to the following documents:

- http://www.tse.or.jp/english/rules/equities/arrowhead/pamphlet.html
- http://www.tse.or.jp/english/rules/equities/arrowhead/pamphlet-e.pdf
- http://www.tse.or.jp/english/rules/equities/arrowhead/pamphlet1e.pdf

- http://www.tse.or.jp/english/rules/equities/arrowhead/pamphlet2e.html

input: an offer for selling stocks, although "ridiculous" in its input data (*"sell 610 thousand stocks, each at one (1) Japanese Yen"*, whereas one stock at 610,000 JPY was meant), and although several immediate — within seconds — attempts to cancel this "order", could not be canceled ! This lead to a loss for the selling broker at around 42 Billion Yen, at today's exchange rate, 26 Jan. 2010, 469 million US$s ![189] Prof. Tetsuo Tamai's paper gives a, to me, chilling account of what I judge as an extremely sloppy and irresponsible design process by TSE and Fujitsu. It also leaves, I think, a strong impression of arrogance on the part of TSE. This arrogance, I claim, is still there in the documents listed in Footnote 188 on the preceding page.

So the problem is a threefold one of

- **Proper Requirements:** How does one (in this case a stock exchange) prescribe (to the software developer) what is required by an appropriate hardware/software system for, as in this case, stock handling: acceptance of buy bids and sell offers, the possible withdrawal (or cancellation) of such submitted offers, and their matching (i.e., the actual trade whereby buy bids are marched in an appropriate, clear and transparent manner).
- **Correctness of Implementation:** How does one make sure that the software/hardware system meets customers' expectations.
- **Proper Explanation to Lay Users:** How does one explain, to the individual and institutional customers of the stock exchange, those offering stocks for sale of bids for buying stocks – how does one explain – in a clear and transparent manner the applicable rules governing stock handling.[190]

I shall only try contribute, in this document, to a solution to the first of these sub-problems.

17.3 A Domain Description

17.3.1 Market and Limit Offers and Bids

1. A market sell offer or buy bid specifies
   a. the unique identification of the stock,
   b. the number of stocks to be sold or bought, and
   c. the unique name of the seller.
2. A limit sell offer or buy bid specifies the same information as a market sell offer or buy bid (i.e., Items 1a–1c), and
   d. the price at which the identified stock is to be sold or bought.
3. A trade order is either a (mkMkt marked) market order or (mkLim marked) a limit order.
4. A trading command is either a sell order or a buy bid.
5. The sell orders are made unique by the mkSell "make" function.
6. The buy orders are made unique by the mkBuy "make" function.

type
  1 Market = Stock_id × Noumber_of_Stocks × Name_of_Customer
  1a   Stock_id

[189] So far three years of law court case hearing etc., has, on Dec. 4, 2009, resulted in complainant being awarded 10.7 billion Yen in damages. See `http://www.ft.com/cms/s/0/e9d89050-e0d7-11de-9f58-00144feab49a.html`.

[190] The rules as explained in the Footnote 188 on the previous page listed documents are far from clear and transparent: they are full of references to fast computers, overlapping processing, etc., etc.: matters with which these buying and selling customers should not be concerned — so, at least, thinks this author !

```
1b   Number_of_Stocks = {|n•n:Nat∧n>1|}
1c   Name_of_Customer
2  Limit = Market × Price
2d   Price = {|n•n:Nat∧n>1|}
3  Trade == mkMkt(m:Market) | mkLim(l:Limit)
4  Trading_Command = Sell_Order | Buy_Bid
5  Sell_Order == mkSell(t:Trade)
6  Buy_Bid == mkBuy(t:Trade)
```

17.3.2 Order Books

7. We introduce a concept of linear, discrete time.
8. For each stock the stock exchange keeps an order book.
9. An order book for stock $s_{id} : SI$ keeps track of limit buy bids and limit sell offers (for the *id*entified stock, s_{id}), as well as the market buy bids and sell offers; that is, for each price
   d. the number stocks, by unique order number, offered for sale at that price, that is, limit sell orders, and
   e. the number of stocks, by unique order number, bid for buying at that price, that is, limitbuy bid orders;
   f. if an offer is a market sell offer, then the number of stocks to be sold is recorded, and if an offer is a market buy bid (also an offer), then the number of stocks to be bought is recorded,
10. Over time the stock exchange displays a series of full order books.
11. A trade unit is a pair of a unique order number and an amount (a number larger than 0) of stocks.
12. An amount designates a number of one or more stocks.

```
type
  7  T
  8  All_Stocks_Order_Book = Stock_Id →m Stock_Order_Book
  9  Stock_Order_Book = (Price →m Orders) × Market_Offers
  9  Orders:: so:Sell_Orders × bo:Buy_Bids
  9d   Sell_Orders = On →m Amount
  9e   Buy_Bids = On →m Amount
  9f  Market_Offers :: mkSell(n:Nat) × mkBuy(n:Nat)
  10  TSE = T →m All_Stocks_Order_Book
  11  TU = On × Amount
  12  Amount = {|n•Nat∧n≥1|}
```

17.3.3 Aggregate Offers

13. We introduce the concepts of aggregate sell and buy orders for a given stock at a given price (and at a given time).
14. The aggregate sell orders for a given stock at a given price is
   g. the stocks being market sell offered and
   h. the number of stocks being limit offered for sale at that price or lower

15. The aggregate bur bids for a given stock at a given price is
    i. including the stocks being market bid offered and
    j. the number of stocks being limit bid for buying at that price or higher

```
value
  14  aggr_sell: All_Stocks_Order_Book × Stock_Id × Price → Nat
  14  aggr_sell(asob,sid,p) ≡
  14    let ((sos,_),(mkSell(ns),_)) = asob(sid) in
  14g     ns +
  14h     all_sell_summation(sos,p) end
  15  aggr_buy: All_Stocks_Order_Book × Stock_Id × Price → Nat
  15  aggr_buy(asob,sid,p) ≡
  15    let ((_,bbs),(_,mkBuy(nb))) = asob(sid) in
  15i     nb +
  15j     nb + all_buy_summation(bbs,p) end

  all_sell_summation: Sell_Orders × Price → Nat
  all_sell_summation(sos,p) ≡
      let ps = {p′|p′:Prices • p′ ∈ dom sos ∧ p′ ≥ p} in accumulate(sos,ps)(0) end

  all_buy_summation: Buy_Bids × Price → Nat
  all_buy_summation(bbs,p) ≡
      let ps = {p′|p′:Prices • p′ ∈ dom bos ∧ p′ ≤ p} in accumulate(bbs,ps)(0) end
```

The auxiliary accumulate function is shared between the all_sell_summation and the all_buy_-summation functions. It sums the amounts of limit stocks in the price range of the accumulate function argument ps. The auxiliary sum function sums the amounts of limit stocks — "pealing off" the their unique order numbers.

```
value
  accumulate: (Price →m Orders) × Price-set → Nat → Nat
  accumulate(pos,ps)(n) ≡
      case ps of {} → n, {p}∪ ps′ → accumulate(pos,ps′)(n+sum(pos(p)){dom pos(p)}) end

  sum: (Sell_Orders|Buy_Bids) → On-set → Nat
  sum(ords)(ns) ≡
      case ns of {} → 0, {n}∪ ns′ → ords(n)+sum(ords)(ns′) end
```

To handle the sub_limit_sells and sub_limit_buys indicated by Item 17c on the facing page of the Itayose "algorithm" we need the corresponding sub_sell_summation and sub_buy_summation functions:

```
value
  sub_sell_summation: Stock_Order_Book × Price → Nat
  sub_sell_summation(((sos,_),(ns,_)),p) ≡ ns +
      let ps = {p′|p′:Prices • p′ ∈ dom sos ∧ p′ > p} in accumulate(sos,ps)(0) end

  sub_buy_summation: Stock_Order_Book × Price → Nat
  sub_buy_summation(((_,bbs),(_,nb)),p) ≡ nb +
      let ps = {p′|p′:Prices • p′ ∈ dom bos ∧ p′ < p} in accumulate(bbs,ps)(0) end
```

17.3.4 The TSE Itayose "Algorithm"

16. The TSE practices the so-called Itayose "algorithm" to decide on opening and closing prices[191]. That is, the Itayose "algorithm" determines a single so-called 'execution' price, one that matches sell and buy orders[192]:
17. The "matching sell and buy orders" rules:
   a. *All market orders must be 'executed'*[193].
   b. *All limit orders to sell/buy at prices lower/higher than the 'execution price'*[194] *must be executed.*
   c. *The following amount of limit orders to sell or buy at the execution prices must be executed: the entire amount of either all sell or all buy orders, and at least one 'trading unit'*[195] *from 'the opposite side of the order book'*[196].

```
value
17 match: All_Stocks_Order_Book × Stock_Id → Price-set
17 match(asob,sid) as ps
17    pre: sid ∈ dom asob
17    post: ∀ p′:Price • p′ ∈ ps ⇒
17′         ∃ os:On-set •
17a′              market_buys(asob(sid))
17b′             + sub_limit_buys(asob(sid))(p′)
17c′             + all_priced_buys(asob(sid))(p′)
17a′           =   market_sells(asob(sid))
17b′             + sub_limit_sells(asob(sid))(p′)
17c′             + some_priced_buys(asob(sid))(p′)(os) ∨
17″         ∃ os:On-set •
17a″              market_buys(asob(sid))
17b″             + sub_limit_buys(asob(sid))(p′)
17c″             + some_priced_buys(asob(sid))(p′)(os)
17a″           =   market_sells(asob(sid))
17b″             + sub_limit_sells(asob(sid))(p′)
17c″             + all_priced_buys(asob(sid))(p′) ∨
```

The match function calculates a set of prices for each of which a match can be made. The set may be empty: there is no price which satisfies the match rules (cf. Items 17a–17c below). The set may be a singleton set: there is a unique price which satisfies match rules Items 17a–17c. The set may contain more than one price: there is not a unique price which satisfies match rules Items 17a–17c. The single (′) and the double (″) quoted (17a–17c) group of lines, in the match formulas above, correspond to the Itayose "algorithm"s Item 17c *'opposite sides of the order book'* description. The existential quantification of a set of order numbers of lines 17′ and 17″ correspond to that "algorithms" (still Item 17c) point of *at least one 'trading unit'*. It may be that the **post** condition predicate is only fulfilled for all trading units – so be it.

```
value
market_buys: Stock_Order_Book → Amount
```

---

[191] [152, pp 59, col. 1, lines 4-3 from bottom, cf. Page 485]

[192] [152, pp 59, col. 2, lines 1–3 and Items 1.–3. after yellow, four line 'insert', cf. Page 485] These items 1.–3. are reproduced as "our" Items 17a–17c.

[193] To execute an order:

[194] Execution price:

[195] Trading unit:

[196] The opposite side of the order book:

```
market_buys((_,(_,mkBuys(nb))),p) ≡ nb

market_sells: Stock_Order_Book → Amount
market_sells((_,(mkSells(ns),_)),p) ≡ ns

sub_limit_buys: Stock_Order_Book → Price → Amount
sub_limit_buys(((,bbs),_))(p) ≡ sub_buy_summation(bbs,p)

sub_limit_sells: Stock_Order_Book → Price → Amount
sub_limit_sells((sos,_))(p) ≡ sub_sell_summation(sos,p)

all_priced_buys: Stock_Order_Book → Price → Amount
all_priced_buys((_,bbs),_)(p) ≡ sum(bbs(p))

all_priced_sells: Stock_Order_Book → Price → Amount
all_priced_sells((sos,_),_)(p) ≡ sum(sos(p))

some_priced_buys: Stock_Order_Book → Price → On-set → Amount
some_priced_buys((_,bbs),_)(p)(os) ≡
    let tbs = bbs(p) in if {}≠os∧os⊆dom tbs then sum(tbs)(os) else 0 end end

some_priced_sells: Stock_Order_Book → Price → On-set → Amount
some_priced_sells((sos,_),_)(p)(os) ≡
    let tss = sos(p) in if {}≠os∧os⊆dom tss then sum(tss)(os) else 0 end end
```

The formalisation of the Itayose "algorithm", as well as that "algorithm" [itself], does not guarantee a match where a match "ought" be possible. The "stumbling block" seems to be the Itayose "algorithm"s Item 17c. There it says: *'at least one trading unit'*. We suggest that a match could be made in which some of the stocks of a candidate trading unit be matched with the remaining stocks also being traded, but now with the stock exchange being the buyer and with the stock exchange immediately "turning around" and posting those remaining stocks as a TSE marked trading unit for sale.

Much more to come: essentially I have only modeled column 2, rightmost column, Page 59 of [152, Tetsuo Tamai, "TSE"]. Next to be modeled is column 1, leftmost column, Page 60 of [152]. See these same page numbers of the present document !

17.3.5 Match Executions

TO BE WRITTEN

17.3.6 Order Handling

TO BE WRITTEN

17.4 Tetsuo Tamai's IEEE Computer Journal Paper

For private, limited circulation only, I take the liberty of enclosing Tetsuo Tamai's IEEE Computer Journal paper.

COVER FEATURE

SOCIAL IMPACT OF INFORMATION SYSTEM FAILURES

Tetsuo Tamai, *University of Tokyo*

The social impact of information systems becomes visible when serious system failures occur. A case of mistyping in entering a stock order by Mizuho Securities and the following lawsuit between Mizuho and the Tokyo Stock Exchange sheds light on the critical role of software in society.

Almost daily, we hear news of system failures that have had a serious impact on society. The ACM Risks Forum moderated by Peter Neumann is an informative source that compiles various reported instances of computer-related risks (http://catless.ncl.ac.uk/risks).

One of journalism's shortcomings is that it makes a loud outcry when trouble occurs with a computer-based system, but it remains silent when nothing goes wrong. This gives the general public the wrong impression that computer systems are highly unreliable. Indeed, as software is invisible and not easy for ordinary people to understand, they generally perceive software to be something unfathomable and undependable.

Another problem is that when a system failure occurs, news sources offer no technical details. Reporters usually don't have the knowledge about software and information systems needed to report technically significant facts, and the stakeholders are generally reluctant to disclose details. The London Ambulance Service failure case is often cited in software engineering literature because its detailed inquiry report is open to the public, which only emphasizes how rare such cases are (www.cs.ucl.ac.uk/staff/a.finkelstein/las/lascase0.9.pdf).

MIZUHO SECURITIES VERSUS THE TOKYO STOCK EXCHANGE

The case of Mizuho Securities versus the Tokyo Stock Exchange (TSE) is archived in the 12 December 2005 issue of the *Risks Digest* (http://catless.ncl.ac.uk/risks/24.12.html), and additional information can be obtained from sources such as the *Times* (www.timesonline.co.uk/tol/news/world/asia/article755598.ece) and the *New York Times* (www.nytimes.com/2005/12/13/business/worldbusiness/13glitch.html?_r=1), among others.

The incident started with the mistyping of an order to sell a share of J-Com, a start-up recruiting company, on the day its shares were first offered to the public. An employee at Mizuho Securities, intending to sell one share at 610,000 yen, mistakenly typed an order to sell 610,000 shares at 1 yen.

Published by the IEEE Computer Society
0018-9162/09/$25.00 © 2009 IEEE

What happened after that was beyond imagination. The order went through and was accepted by the Tokyo Stock Exchange Order System. Mizuho noticed the blunder and tried to withdraw the order, but the cancel command failed repeatedly. Thus, it was obliged to start buying back the shares itself to cut the loss. In the end, Mizuho's total loss amounted to 40 billion yen ($225 million). Four days later, TSE called a news conference and admitted that the cancel command issued by Mizuho failed because of a program error in the TSE system. Mizuho demanded compensation for the loss, but TSE refused. Then, Mizuho sued TSE for damages.

When such a case goes to court, we can gain access to documents presented as evidence, which provides a rare opportunity to obtain information about the technical details behind system failures. Still, requesting and acquiring documents from the court requires considerable effort by the third party. As it happened, Mizuho contacted me to give an expert opinion, thus I had access to all materials presented to the court. Admittedly, there is always the possibility of bias, but as a scientist, I have endeavored to report this case as impartially as possible.

Another reason for examining this case is that it involved several typical and interesting software engineering issues including human interface design, fail-safety issues, design anomalies, error injection by fixing code, ambiguous requirements specification, insufficient regression testing, subcontracting, product liability, and corporate governance.

WHAT HAPPENED

J-Com was initially offered on the Tokyo Stock Exchange Mother Index on 8 December 2005. On that day, a Mizuho employee got a call from a client telling him to sell a single share of J-Com at 610,000 yen. At 9:27 a.m., the employee entered an order to sell 610,000 shares at 1 yen through a Fidessa (Mizuho's securities ordering system) terminal. Although a "Beyond price limit" warning appeared on the screen, he ignored it (pushing the Enter key twice meant "ignore warning" by the specification), and the order was sent to the TSE Stock Order System. J-Com's outstanding shares totaled 14,500, which means the erroneous order was to sell 42 times the total number of shares.

At 9:28 a.m., this order was displayed on the TSE system board, and the initial price was set at 672,000 yen.

Price determination mechanism

TSE stock prices are determined by two methods: *Itayose* (matching on the board) and *Zaraba* (regular market). The Itayose method is mainly used to decide opening and closing prices; the Zaraba method is used during continuous auction trading for the rest of the trading session. In the J-Com case, the Itayose method was used as it was the first day of determining the J-Com stock price.

There are two order types for selling or buying stocks: *market orders* and *limit orders*. Market orders do not specify the price to buy or sell and accept the price the market determines, while limit orders specify the price. When sell and buy orders are matched to execute trading, market orders of both sell and buy are always given the first priority.

Market participants generally want to buy low and sell high. But when the Itayose method is applied, there is no current market price to refer to, and thus there can be a variety of sell/buy orders, resulting in a wide range of

An employee at Mizuho Securities, intending to sell one share at 610,000 yen, mistakenly typed an order to sell 610,000 shares at 1 yen.

prices. With the Itayose method, a single execution price is determined that matches sell and buy orders by satisfying the following rules:

1. All market orders must be executed.
2. All limit orders to sell/buy at prices lower/higher than the execution price must be executed.
3. The following amount of limit orders to sell or buy at the execution price must be executed: the entire amount of either all sell or all buy orders, and at least one trading unit from the opposite side of the order book.

The third rule is complicated but functions as a tie-breaker when the first two rules do not determine a unique price. Looking at an example helps to understand how the rules work.

Table 1 represents an instance of the order book. The center column gives the prices. The left center column shows the volume of sell offers at the corresponding price, while the right center column shows the volume of the buy bids. The volume of the market sell orders and the market buy orders is displayed at the bottom line and at the top line, respectively. The leftmost column shows the aggregate volume of sell offers (working from the bottom to the top in the order of priority), and the rightmost column gives the aggregate volume of buy bids (working from the top to the bottom in the order of priority).

We start by focusing on rules (1) and (2) to determine the opening price. First, the price level is searched where the amounts of the aggregated sell and the aggregated buy cross over. In this case, the line is between 500 yen and 499

Table 1. Order book example illustrating Itayose method.

Sell offer			Buy bid	
Aggregate sell orders	Shares offered at bid	Price (yen)	Buy offers at bid	Aggregate buy orders
		Market	4,000	
48,000	8,000	502	0	4,000
40,000	20,000	501	2,000	6,000
20,000	5,000	500	3,000	9,000
15,000	6,000	499	15,000	24,000
9,000	3,000	498	8,000	32,000
6,000	0	497	20,000	52,000
	6,000	Market		

yen. These two prices satisfy conditions (1) and (2), so they are the opening price candidates. Then, applying rule (3), the price is finally determined as 499 yen.

Of course, this algorithm does not always determine the price. For example, if the orders are all buy and no sell, there is no solution that satisfies all three rules. An additional mechanism that holds back transactions even if the matching price is found by the Itayose method is a measure to prevent sudden price leaps or drops. On the TSE, an immediate execution only takes place if the next execution price is within a certain range from the previous execution price. The price level determines the range. For example, if the most recently executed price was 500 yen, the next execution price must be within the range of 490-510 yen. In other words, it can only fluctuate up to 10 yen in either direction.

Suppose the matched price is beyond this range—for example, 550 yen when the previous price was 500 yen. Then, execution does not take place; instead a special bid quote of 510 yen is indicated to call for offers at this price. If no offers at this price are received, the special bid quote will be raised to 520 yen after 5 minutes, and so on until equilibrium is achieved. This mechanism is intended to make a smooth transition between widely divergent prices.

But on the morning of 8 December, J-Com had no previous price. In such cases, the publicly assessed value is used in place of the previous price, which was 610,000 yen. Because the matched price was much higher, a special bid quote of 610,000 yen was shown at 9:00 a.m., then raised to 641,000 yen at 9:10 a.m., which means the range was ± 31,000 yen, and raised again to 672,000 yen at 9:20 a.m. Table 2 shows the order book at that moment, when the 1-yen sell offer came in.

Initial price determination

The term "reverse special quote" denotes this particularly rare event. It means that when a special buy bid quote is displayed, a sell order of low price with a significant amount that reverses the situation to a special sell offer quote comes in (or conversely a special sell offer quote is reversed to a special buy bid quote). TSE has another rule that applies to such a case. This rule stipulates that the previous special quote is fixed as the execution price, and the transaction proceeds. Thus, the initial price of J-Com was now determined to be 672,000 yen. In addition to the step price range set for reducing sudden price change, there is also a price limit range for a day. The upper and lower limits of the price for each stock are defined based on the initial price of the day. In the J-Com case, the limits were defined at the moment when the initial price was determined: The upper limit was 772,000 yen, and the lower limit was 572,000 yen.

In regular trading, the price limits are fixed at the start of the market day, and orders with prices exceeding the limit (either upper or lower) are rejected. But when the initial price is determined during the market time, as in the J-Com case, orders received before the price limits are set are not ignored. Instead, the price of an order exceeding the upper limit is adjusted to the upper price limit, and an order under the lower limit is adjusted to the lower price limit. Thus, the 1-yen order by Mizuho was adjusted to 572,000 yen.

Noticing the mistake, Mizuho entered a cancel command through a Fidessa terminal at 9:29:21, but it failed. Between 9:33:17 and 9:35:40, Mizuho tried to cancel the order several times through TSE system terminals that are installed at the Mizuho site, but the cancellations failed. Mizuho called TSE asking for a cancellation on the TSE side, but the answer was no.

At 9:35:33, Mizuho started to buy back J-Com shares. In the end, it could only buy back 510,000 shares; nearly 100,000 shares were bought by others and never restored.

Aftermath

On 12 December, four days after the incident, TSE president Takuo Tsurushima held a press conference and admitted that the order cancellation by Mizuho failed because of a defect in the TSE Stock Order System.

Table 2. Order book for J-Com stock at 9:20 a.m.

Sell offer			Buy bid	
Aggregate	Amount	Price (yen)	Amount	Aggregate
		Market	253	
1,432	695	OVR*	1,479	1,732
737		6,750	4	1,736
737		6,740	6	1,742
737		6,730	6	1,748
737	3	6,720**	28	1,776
734		6,710	2	1,778
734		6,700	3	1,781
734	1	6,690	1	1,782
733		6,680	1	1,783
733	114	UDR***	120	1,903
	619	Market		

* More than 675,000 yen ** Special buy quote *** Less than 668,000 yen

Mizuho could not buy back 96,236 shares, and it was impossible for Mizuho to deliver real shares to those who had bought them. An exceptional measure was taken to settle trading by paying 912,000 yen per share in cash. The result was a 30-billion-yen loss to Mizuho. Mizuho had already suffered a loss of 10 billion yen by buying back 510,000 shares, thus the total loss amounted to 40 billion yen.

Mizuho and TSE started negotiations on compensation for damages in March 2006, but they failed to reach an agreement. Mizuho sent a formal letter to TSE in August 2006 requesting compensation, which TSE declined by sending a letter of refusal.

Mizuho filed a suit against TSE in the Tokyo District Court on 27 October 2006, demanding compensation of 41.5 billion yen. The first oral pleadings took place on 15 December 2006, and trials were held 13 times in two years, the last on 19 December 2008. The court's decision in that trial was scheduled to be given on 27 February 2009, but the court decided to postpone the decision.

In the contract between TSE and each user of the TSE Stock Order System, including Mizuho, there is a clause on exemption from responsibility on the TSE side except when a serious mistake is attributed to TSE. The crucial issue was whether the damage caused by the system defect was due to a serious mistake beyond the range of exemption. TSE also argued that as the incident started with a mistake on the Mizuho side, the mistakes and the resulting damages should be canceled out.

PROBABLE CAUSE

The TSE system unduly rejected the Mizuho order cancellation because the module for processing order cancellation erroneously judged that the J-Com target sell order had been completely executed, thus leaving no transactions to be canceled. This bug had been hiding for five years.

Fujitsu developed the system under contract with TSE and released it for use in May 2000. An evidence document submitted to the court reported that a similar error was found during integration testing in February 2000 and that the current fault occurred as a result of fixing that error.

But there are several mysteries surrounding this apparently simple failure case. Initially, TSE maintained that the target cancellation order could not be found because its price had been changed from 1 yen to the adjusted price of 572,000 yen, whereas the designated cancel price command was the original 1 yen. This explanation is bizarre as it implies that the order data is searched in the database using price as a key when it is obvious that price cannot be a key because there can be multiple orders with the same price. In addition, as this case shows, the price of the same order can be modified during the transaction. This explanation turned out to be wrong, but it came from the fact that there was indeed a logic in the procedure that partly used price to search order data. TSE also maintained that if buy orders did not flow in continuously and thus the target sell orders were not always being matched to buy orders, the order cancel module would not have been invoked within the order matching module but instead invoked in the order entry module, and then the cancellation would have succeeded. However, this explanation implies that different cancel modules are called or the same module behaves differently according to when it is invoked.

The third question, and probably the most crucial one with respect to the direct cause of the error, is how data handling identifies orders causing a reverse special quote. That information is written into a database containing

COVER FEATURE

the order book data, but once the information is used in determining the execution price, it is immediately cleared. The rationale behind this design decision is mysterious. The programmer who was charged with fixing the February 2000 bug intended to use this data to judge the type of order to be canceled but he did not know that the data no longer existed.

TSE and Fujitsu claimed that this incident occurred in a highly exceptional situation when the following seven conditions held at the same time:

1. The daily price limits have not been determined.
2. The special quote is displayed.
3. The reverse special quote occurs.
4. The price of the order that has caused the reverse special quote is out of the newly defined daily price limits.
5. The target order of cancellation caused the reverse special quote.
6. The target cancellation order is in the process of sell and buy matching, which forces the cancellation process to wait.
7. The target order is continually being matched.

The order cancellation module appears to have insufficient cohesion as different functions are overloaded.

A general procedure for the order cancellation module would be as follows:

1. Find the order to be canceled.
2. Determine if the order satisfies conditions for cancellation.
3. Execute cancellation if the conditions are met.

Because each order has a few simple attributes—stock name, sell or buy, remaining number of shares to be processed (if 0, the order is completed), and price—the condition that an order can be canceled is straightforward: "the remaining number of shares to be processed is greater than zero." There is only one other condition that cannot be determined by the order attribute data but can be determined by its execution state: If the target order is in the process of matching, the cancel process must wait.

A remarkable point to note is that factors such as undefined limit price, display of special quote, reversing special quote, price adjustment to the limit, and so forth have no influence on the cancellation judgment. Thinking in this way, it seems that the system design artificially introduced the seven complicated conditions listed by TSE and Fujitsu.

DESIGN ANOMALY

Figure 1 shows a flowchart of the module that handles order cancellation. Because order cancellation and order change are processed in the same way, the two functions are overloaded in this same module, but for the sake of simplicity I only deal with order cancellation.

The flowchart is not shown to provide details but to illustrate the kind of documents presented to the court. It is extracted and modified from a document submitted as evidence by the defendant, which was an analysis of the error reported by a TSE system engineer. The plaintiff required the defendant to provide the entire design specification and source code, but the defendant refused and the judge did not force the issue, being reluctant to go into technical details in court.

Part A of the flowchart deals with the logic of price adjustment to limit if necessary. The decision logic is as follows:

- if the order to cancel is sell and the price is lower than the lower limit, it is adjusted to the lower limit; and
- if the order to cancel is buy and the price is higher than the upper limit, it is adjusted to the upper limit.

Part B of the flowchart is the logic inserted in February 2000 when an error was found during testing and caused a failure in December 2005. Its logic is as follows: If called in the order matching process; and limit prices are already set; and the order to cancel is a buy over the limit price or a sell under the limit price and is not a reverse special quote order; then a cancellation is infeasible because all shares are already executed. Although this logic is unduly complicated, it is sound only if all the if-conditions are correctly judged. Unfortunately, the judgment on "if not a reverse special quote" gave a wrong answer of "true" in this Mizuho case, and the decision erroneously judged that the cancellation was infeasible.

Insufficient information is available to allow capturing details of the system design, but from what is available we can infer the following design flaws.

Problems in database design

Three databases are related to the problem in this case: Order DB, Sell/Buy Price DB, and Stock Brand DB. The Order DB stores data of all entered orders. This database should include the current attributes of each order, including those necessary for judging whether the designated order can be canceled. For example, because there is a record field for the executed shares in this database, determining if all the shares of the order have been executed or not should be a trivial process. However, due to the time gap between usage and update of the data, the process is much more complicated. If the principle of database integrity is respected, the logic would be much clearer, but performance seems to be given higher priority than integrity.

Part A of the flowchart in Figure 1 calculates price adjustment within the cancellation handling module, which implies that the price data in the Order DB does not reflect the current status.

The Sell/Buy Price DB sorts sell/buy orders by price for each stock brand. This is by nature a secondary database constructed from the Order DB. The secondary index is price, but identifying an order uniquely in the database requires the order ID. The explanation that price is used to search the database must refer to search in this database, and the price adjustment logic embedded in the order cancel module should be related to it. The data handling over the Price DB and the Order DB appears to be unduly complicated.

The Stock Brand DB corresponds to a physical order book for each stock, but its substantial data is stored in the Sell/Buy Price DB and only some specific data for each stock brand is kept here. However, to implement a rule that an order that has caused a reverse special quote has an exceptional priority in matching—lower than the regular case—the customer ID and order ID of such a stock is written in this database, and they are cleared as soon as the matching is done. This kind of temporary usage of a database goes against the general principle that a database should save persistent data accessed by multiple modules.

Problems in module design.

The part of the system that handles order cancellation appears to have low modularity. The logic in part B of the flowchart made a wrong judgment because the information telling it that the target order had induced the reverse special quote had been temporarily written on the Stock Brand DB by the order matching module and had already been cleared. This implies an accidental module coupling between the order matching and order cancelling modules.

The order cancellation module appears to have insufficient cohesion as different functions are overloaded. It is not clear how the tasks of searching the target order to be canceled, determining cancellability, executing cancellation, and updating the database are this module's responsibility.

LESSONS LEARNED

In addition to the insights into the associated software design problems, this case provides lessons learned with regard to software engineering technologies, processes, and social aspects.

Safety and human interface

If the order entry system on either the Mizuho or TSE side had been equipped with more elaborate safety measures, the accident could have been avoided. It was not the first time that the mistyping of a stock order resulted in a big loss. For instance, in December 2001, a trader at UBS Warburg, the Swiss investment bank, lost more than 10 billion yen while trying to sell 16 shares of the Japanese advertising company Dentsu at 610,000 yen each. He sold 610,000 shares at six yen each. (The similarity between these two cases, including the common figure of 610,000, is remarkable.)

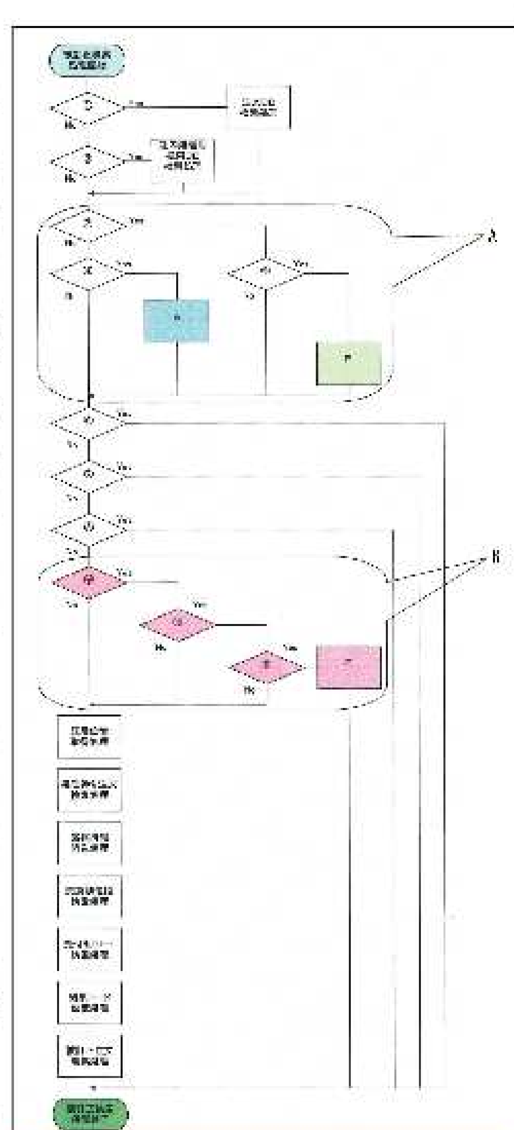


Figure 1. Flowchart of the order cancellation module.

COVER FEATURE

Table 3. Associations between the Software Engineering Code of Ethics and Professional Practice and the TSE-Mizuho case.

Engineering issue	Applicable ACM/IEEE-CS Principle	Ethics clause
Design anomaly	3.01	Strive for high quality, acceptable cost and a reasonable schedule, ensuring significant tradeoffs are clear to and accepted by the employer and the client, and are available for consideration by the user and the public.
	3.14	Maintain the integrity of data, being sensitive to outdated or flawed occurrences.
Safety and human interface	1.03	Approve software only if they have a well-founded belief that it is safe, meets specifications, passes appropriate tests, and does not diminish quality of life, diminish privacy or harm the environment. The ultimate effect of the work should be to the public good.
Requirements specification	3.07	Strive to fully understand the specifications for software on which they work.
	3.08	Ensure that specifications for software on which they work have been well documented, satisfy the users' requirements and have the appropriate approvals.
Verification and validation	3.10	Ensure adequate testing, debugging, and review of software and related documents on which they work.
Role of user and developer	4.02	Only endorse documents either prepared under their supervision or within their areas of competence and with which they are in agreement.
	5.01	Ensure good management for any project on which they work, including effective procedures for promotion of quality and reduction of risk.
Chain of subcontracting	2.01	Provide service in their areas of competence, being honest and forthright about any limitations of their experience and education.
	3.04	Ensure that they are qualified for any project on which they work or propose to work by an appropriate combination of education and training, and experience.

The habit of ignoring warning messages is common, but it was a critical factor in these cases. It raises the question of how to design a safe—but not clumsy—human interface.

Requirements specification

Development of the current TSE Stock Order System started with the request for proposal (RFP) that TSE presented to the software industry in January 1998. Two companies submitted proposals, and TSE selected Fujitsu as the vendor with which to contract. After several discussions between TSE and Fujitsu, Fujitsu wrote the requirements specification, which TSE approved.

With respect to the order cancellation requirement, it is only mentioned as a function to "Cancel order" in the RFP, and no further details are given there. In the requirements specification, six conditions are listed when cancel (or change) orders are not allowed, but none of them fit the Mizuho case. The document also states that "in all the other cases, change/cancel condition checking should be the same as the current system." Here, "current system" refers to the prior version of the TSE Stock Order System, also developed by Fujitsu, which had been in use until May 2000.

The phrase "the same as the current system" frequently appears in this requirements specification, which was criticized by software experts after the Mizuho incident was publicized. The phrase may be acceptable if there is a consensus between the user and the developer on what it means in each context, but when things go wrong, the question arises whether the specification descriptions were adequate.

Verification and validation

The fact that an error was injected while fixing a bug found in testing is so typical that every textbook on testing warns about this possibility. It is obvious that regression testing was not properly done. It is perhaps too easy to criticize this oversight, but it would be worthwhile to study why it happened in this particular case. So far, not many details have been disclosed.

Role of user and developer

It is conceivable that communication between the user and the developer was inadequate during the TSE system development. The user, TSE, basically did not participate in the process of design and implementation. More involvement of the user during the entire development process would have promoted deeper understanding of the requirements by the developer, and the defect injected during testing might have been avoided.

Subcontracting chain

As in many large-scale information system development projects, the TSE system project was organized in a hierarchical subcontracting structure. The engineer who was in charge of fixing the code in question had a low position in the subcontracting chain. This organizational structure was the likely cause of the misunderstanding about database usage. Such a subcontract structure has often been studied from the industry and labor problem point of view, but it is also important to examine it from the engineering point of view.

Product liability

The extent to which software is regarded as a product amenable to product liability laws may depend on legal and cultural boundaries, but there is a general worldwide trend demanding stricter liability for software. More lawsuits are being filed, and thus software engineers must be more knowledgeable about software product liability issues.

ETHICAL ISSUES

This case raises several questions about professional ethics. However, we should be careful in relating ethical issues and legal matters. Illegal conduct and unethical conduct are of course not equivalent. Moreover, the Mizuho incident is a civil case, not a criminal case.

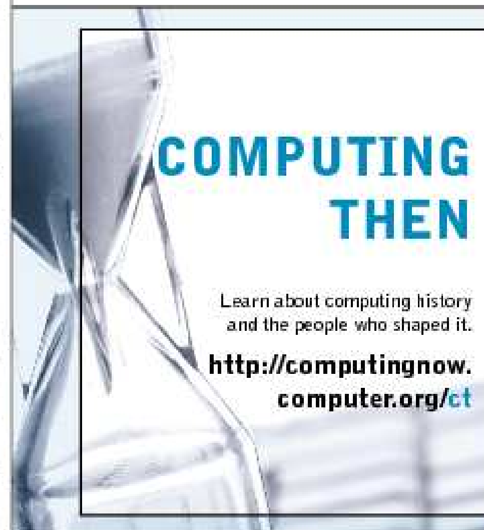
The Software Engineering Code of Ethics and Professional Practice developed by an ACM and IEEE Computer Society joint task force provides a good framework for discussing ethical issues. The Code comprises eight principles, and each clause is numbered by its principle category and the sequence within the principle. The principles are numbered as 1: Public, 2: Client and Employer, 3: Product, 4: Judgment, 5: Management, 6: Profession, 7: Colleagues, and 8: Self.

As Table 3 shows, some clauses in the Code have relatively strong associations with various aspects of the TSE-Mizuho case. However, this discussion is by no means intended to blame the software engineers who participated in planning, soliciting requirements, designing, implementing, testing, or maintaining the TSE system or other related activities, or to suggest negligence of ethical obligations. First, the Code was not intended to be used in this fashion. Second, the collected facts and disclosed materials are insufficient to precisely judge what kind of specific conduct caused the unfortunate result. However, linking the problems in this case with plausibly related ethical obligation clauses as shown in Table 3 can provide a basis for considering the ethical aspects of this incident and other similar cases.

In addition to individual ethical conduct, the Mizuho-TSE case raises issues pertaining to corporate governance. Why did such an erroneous order by a trader go through unnoticed at Mizuho? Did the TSE staff respond appropriately when they were consulted about the order cancellation? How did Fujitsu manage subcontractors? Corporate governance is another domain where software engineering must deal with social and ethical issues.

If we can learn valuable lessons from this unfortunate incident, it would be beneficial. We should also encourage people who have access to information about similar system failures having significant social impact to analyze and report those cases.

Tetsuo Tamai is a professor in the Graduate School of Arts and Sciences at the University of Tokyo. His research interests include requirements engineering and formal and informal approaches to domain modeling. He received a DrS in mathematical engineering from the University of Tokyo. He is a member of the IEEE Computer Society, the ACM, the Information Processing Society of Japan, and the Japan Society for Software Science and Engineering. Contact him at tamai@graco.c.u-tokyo.ac.jp.



Chapter 18

An “Extensible” Virtual Shared Memory [May–July 2010]

Contents

18.1 Introduction 494
18.1.1 On Targets of Formal Specification 494
18.1.2 Why Specify Software Concepts Formally 495
18.1.3 An XVSM Type System 495
18.1.3.1 Type Systems 496
18.1.3.2 Static and Dynamic Type Systems 496
18.1.3.3 Why Type Systems 496
18.1.4 Words of Caution 496
18.2 XVSM Trees 497
18.2.1 XTree Rules 497
18.2.2 XTree Types 497
18.2.3 XTree Type Designator Wellformedness 497
18.2.4 XTree Type Functions 498
18.2.5 XTree Wellformedness 498
18.2.6 XTree Subtypes 499
18.3 XTree Operations 500
18.3.1 XTree Multiset Union 500
18.3.2 Commensurate Multiset Arguments 500
18.3.3 Type “Prediction” 501
18.3.4 A Theorem: Correctness of Type “Prediction” 501
18.3.5 XTree Multiset Equality 501
18.3.6 XTree Multiset Subset 502
18.3.7 Property Multiset Membership 502
18.3.8 XTree Multiset Membership 502
18.3.9 XTree Multiset Cardinality 502
18.3.10 Arbitrary Selection of XTrees or Properties from Multisets 503
18.3.11 XTree Multiset Difference 503
18.3.12 XTree List Concatenation 503
18.3.13 XTree List Equality 504
18.3.14 XTree List Property Membership 504
18.3.15 XTree List XTree Membership 504
18.3.16 XTree List Length 504
18.3.17 XTree List Head 505
18.3.18 XTree List Tail 505
18.3.19 XTree List Nth Element 505
18.4 Indexing 505
18.4.1 Paths and Indexes 505
18.4.2 Proper Index 506
18.4.3 Index Selecting 506
18.4.4 Path Indexing 506
18.5 Queries 507
18.5.1 Generally on Semantics 507
18.5.2 Syntax: Simple XVSM Queries 508
18.5.2.1 Syntax: Predefined Selector Queries 508

18.5.2.2 Semantics: Predefined Selector Queries 508
18.5.2.2.1 Count 508
18.5.2.2.2 Sort Up 509

This document presents work in progress. The document constitutes a technical note. It reports on an attempt to formalise XVSM: the Extensible Virtual Shared Memory as reported in the Dipl.Ing. thesis by Stefan Craß: *A Formal Model of the Extensible Virtual Shared Memory (XVSM) and its Implementation in Haskell – Design and Specification. Technische Universität Wien, 05.02.2010* [76].

18.1 Introduction

XVSM, the Extensible Virtual Shared Memory concept, has been described in a number of conference proceeding publications: [113, 114, 77, 13]. The MSc Thesis [76] claims to present a formal model, but what is presented is not a proper formal model. To be a proper formal model there must be an abstract presentation in some formal, that is, mathematically well defined specification language and there must be a formal proof system for that language. Usually a formal semantics is also an abstract specification. Haskell, although a commendable programming language, is not suited for the specification of a semantics of XVSM, and [76] presents a Haskell implementation of XVSM and not an abstraction. A reasonably precise, even readable (and also executable), definition of XVSM could have been done in Haskell. Such a definition would carefully build up definitions, in Haskell, of the syntax of XVSM XTrees, of XVSM queries, etc. We shall present a formal definition of XVSM in RSL, the Raise Specification Language [92, 93].

18.1.1 On Targets of Formal Specification

Formalisation of software concepts started in the 1960s. The most notable example was that of the formal (operational semantics) description of the PL/I programming language [8, 9, 10]. The ULD notation emerged (ULD I, ULD II, ULD III -- ULD for Universal Language Description). This name of notation was later renamed into VDL (Vienna Language Description) by J.A.N. Lee [115]. Peter Lucas (sometimes with Kurt Walk) reviewed the [123, 118, 117, 119, 120, 121, 122] semantics descriptions of notably ULD III and the background for VDM (the Vienna Development Method).

As a result of the VDL (research and experimental development) work the IBM Vienna Laboratory undertook, in 1973, to develop, for the IBM market a new PL/I compiler for a new IBM computer (code named FSM: Future Systems machine). The US and European IBM laboratories' development of this computer was, eventually, curtailed, in February 1974. Nevertheless, the IBM Vienna Laboratory, was able to complete the work on a formal (denotational semantics-like) description of PL/I [7]. This work led to VDM [64, 109, 65, 110, 111, 90] – which later led to RAISE [92, 93] (1990). All the other now available formal specification languages came after VDM: Alloy [106] (2000), B, Event B [2] (1990, 2000) and Z [155] (1980).

First with VDM and now, as here, with RSL, formal specification has been used – other than for the semantics description of programming languages – first for formalising software designs, then for formalising requirements for general software, and for formalising (their) domain descriptions.

In this technical note we apply, not for the first time, formal specification to what the proposers of XVSM refers to as *middleware:* computer software that connects software components or applications. The software consists of a set of services that allows multiple processes running on one or more machines to interact (including sharing data). *Middleware* technology evolved to provide for interoperability in support of the move to coherent distributed architectures, which are most often used to support and simplify complex distributed applications. It includes web

servers, application servers, and similar tools that support application development and delivery. *Middleware* is especially integral to modern information technology based on `XML, SOAP, Web` services, and service-oriented architecture.

18.1.2 Why Specify Software Concepts Formally

A number of independent reasons can be given for why one might wish to formally specify a software concept[197]. We itemize some of these:

- **As a design aid:** In researching and experimentally developing the design of a software concept, experiments with formal models of the software concept, or just some of its sub-concepts, have shown to help clarify and simplify many design issues[198].
- **As a communication document:** A suitably narrated and formalised specification, such as the present technical note lays a ground for (but is not yet), can be used as a, or the, 'semantics' specification for `XVSM`. It can serve as a standards document.
- **As a basis for implementation:** A suitably narrated and formalised specification, such as the present technical note, can serve as a basis for (thus provably) correct implementations of proper `XVSM` middleware.
- **As a basis for teaching & training:** An `XVSM` communication document can serve as the basis for instruction in the use (i.e., 'programming') of `XVSM`-dependent applications.
- **As a basis for proving properties of `XVSM`:** The formal specfication of `XVSM`, such as attempted, or at least begun, with the present technical note, can be referred to in formal proofs of properties of `XVSM` and its applications.

18.1.3 An `XVSM` Type System

One of the great contributions of computing science to mathematics has been the studies made of type systems. And one of the great advances of software engineering from the middle 1950s till today has been the use of suitable, usually static, type systems.

The current author has (therefore) been quite surprised when discovering, that a language such as the `XVSM` query language and the *Core Application Programming Interface Languages*[199] (such as `CAPI-1`, `CAPI-2`, and `CAPI-3`) has not been endowed by a type system. Instead of erroneous query and transaction results (here modelled by **chaos**) an `XVSM` programme could use these type testing facilties to secure provably correct uses of `XVSM`.

We shall, here and there, 'divert' from a straight line reformulation of [76], and present components of an `XVSM` Type System (XVSM/TS).

---

[197] By a software concept we mean such concepts as (the semantics of) programming languages, database models or database management systems, operating systems, specific application systems [such as for air traffic, banking, manufacturing, transportation, or other], their requirements, their underlying domains, etc.

[198] The current author offers the following observation (i) and advice (ii): (i) it seems that formalisation was not used in the conceptualisation of `XVSM`; and (ii) further extensions of `XVSM` should preferably be based on the present – or similar, reworked – formalisation and should itself use formal modelling. In reading publications about `XVSM` an experienced reader of precise descriptions too easily resolves that there are simply far too many ambiguous, incinsistent and incomplete description points: they may not be so, but the current egnlish texts leaves such an experienced reader of precise descriptions to resolve so.

[199] An application programming interface (API) is an interface implemented by a software program which enables it to interact with other software.

18.1.3.1 Type Systems

Many kinds of type systems can be proposed for XVSM. Defining a type system may imply that only correctly typed data, i.e., XTrees, and only arguments to operations: queries and actions, that, in some weak or strong sense, satisfy the signature (that is, the type) of the operation are allowed. (We then speak of a weakly, respectively strongly type language.) We shall, through the judicious use of concepts of sub, commensurate- and super-types, suggest one (of several possible) XVSM type systems. It is important to emphasize this: that either one of several XVSM type systems are possible. The one presented ehre may not be the best for a number of contemplated applications of XVSM, but it is probably a sensible one! Recommendable monographs cum textbooks on type systems and programming languages are [143, 129]. Further foundational studies of type systems are provided in the monographs [94, 1].

18.1.3.2 Static and Dynamic Type Systems

A programming language is said to use static typing when type checking can be performed during compile-time as opposed to run-time.

A programming language is said to be dynamically typed when the majority of its type checking can only be performed at run-time as opposed to at compile-time. In dynamic typing, values have types but variables do not (necessarily); that is, a variable can refer to a value of any type. Whether one can speak of XVSM variables is not known.

We shall anyway think of the type system that we shall put forward for XVSM as being a dynamic type system.

18.1.3.3 Why Type Systems

Reasons for endowing XVSM with a type system can be itemized:

- **Safety:** Checking, before execution, that an operation, with the types of its argument and the types of the space-based data, that is, XTrees, satisfy the type rules helps avoid otherwise meaningless operations.
- **Optimisation:** Static type-checking may provide useful compile-time information. Dynamic type-checking may provide useful run-time information.
- **Documentation:** In expressive type systems, types can serve as a form of documentation, since they can illustrate the intent of the programmer.
- **Abstraction (Modularity):** Types allow programmers to think about programs at a higher level than the bit or byte, not bothering with low-level implementation.

Any one chosen type system will have been devised so as to satisfy at least one of the above reasons.

18.1.4 Words of Caution

The type system proposed here for XVSM is just an example. I am not quite sure that my particular design choices are the right ones for a system like XVSM. A perhaps more proper XVSM type system should evolve as the result of close, concentrated discussions and work, in Vienna, not over the Internet, between the leading authors of [113, 114, 77, 13, 76] and Dines Bjørner. But what I am rather sure of is that for XVSM to be considered a serious contender for so-called space-based

computing XVSM must be endowed with a type system and with a suitable set of type system (run-time) operations.

18.2 XVSM Trees

18.2.1 XTree Rules

18. There are labels and labels are further unspecified quantities.
19. Properties are pairs of labels and XTrees, that is, a property is such a pair.
20. An XTree is either an XTree value or an XTree multiset or an XTree sequence (an XTree list).
    a. An XTree value is either some XTree text or is some XTree integer.
    b. An XTree multiset consists of a multiset of properties.
    c. An XTree sequence consists of a list of properties.

type
18 L
19 P = L × XT
20 XT = XV | XL | XS
20a XV == $\mu\alpha\kappa$ST(sel_txt:**Text**) | $\mu\alpha\kappa$IN(sel_i:**Int**)
20b XS == $\mu\alpha\kappa$XS(sel_xs:(P $\overrightarrow{m}$ **Nat**))
20c XL == $\mu\alpha\kappa$XL(sel_xl:P*)

18.2.2 XTree Types

21. An XTree type is either
    a. an integer type, or
    b. a text type, or
    c. a multiset type which maps its entry labels into corresponding XTree type, or
    d. a sequence type which is a sequence of labelled XTree types.

type
21 XTTy = IntTy | TxtTy | MulTy | SeqTy
21a IntTy == mkITy
21b TxtTy == mkTTy
21c MulTy == $\mu\alpha\kappa$MTy(m:(L $\overrightarrow{m}$ XTTy))
21d SeqTy == $\mu\alpha\kappa$STy(m:(L × XTTy)*)

XTTy are type designators.

18.2.3 XTree Type Designator Wellformedness

22. A type designator, i.e., any XTTy is wellformed if it satisfies the following conditions:
    a. Integer and text type designators are wellformed.
    b. Multiset type designators are wellformed if the type designators for any label are wellformed.

c. Sequence type designators are wellformed if all labelled type designators are wellformed and if the type designators for identifically labelled entries are the same type.[200]

value
22. wf_XTTy: XTTy → **Bool**
22. wf_XTTy(t) ≡
22. **case** t **of**
22a. mkITy → **true**,
22a. mkTTy → **true**,
22b. $\mu\alpha\kappa$MTy(tym) → ∀ t′:XXTy•t′ ∈ **rng** tym ⇒ wf_XTTy(t′)
22c. $\mu\alpha\kappa$STy(tyl) →
22c. ∀ (l′,t′):(L×XTTy)•(l′,t′) ∈ **elems** tyl ⇒ wf_XTTy(t′) ∧
22c. ∀ (l″,t″):(L×XTTy)•(l″,t″)∈ **elems** tyl ⇒ xtr_type(t′) = xtr_type(t″) **end**

18.2.4 XTree Type Functions

23. Given an `XTree` one can “extract” its type:

a. The type of an integer value is `mkITy`.
b. The type of a text value is `mkTTy`.
c. The type of an `XTree` multiset, ms, is $\mu\alpha\kappa$`MTy(tym)` where `tym` is a mapping from the labels of ms to the `XTree` type of the corresponding values.[201]
d. The type of an `XTree` sequence, sq, is $\mu\alpha\kappa$`STy(tys)` where `tys` is a sequence of labelled `XTree` types of the indexed (and labelled) `XTree` values of the sequence.

value
23. xtr_type: XT $\tilde{\rightarrow}$ XTTy
23. xtr_type(xt) ≡
23. **case** xt **of**
23a. $\mu\alpha\kappa$IN(intg) → mkITy,
23b. $\mu\alpha\kappa$ST(text) → mkTTy,
23c. $\mu\alpha\kappa$XS(xs) → $\mu\alpha\kappa$MTy([l↦xtr_type(xt′)|l:L•l ∈ **dom** xs ∧ xt′ ∈ xs(l)]),
23d. $\mu\alpha\kappa$XL(xl) → $\mu\alpha\kappa$STy(⟨(l,xtr_type(xt′))|i:**Nat**•i ∈ **inds** xl ∧ xl(i)=(l,xt′)⟩)
23. **end**
23. **pre**: type_conform(xt)

18.2.5 XTree Wellformedness

24. An `XTree` is type_conformant if

a. it is an integer, or
b. it is a text, or
c. it is a multiset all of whose `XTree`s are type_conformant and all identically labelled `XTree`s have the same type, or

[200] **Note:** This constraints is in line with the constraint of Item 21c on the previous page.

[201] **Note:** Thus we constrain two or more properties with the same label to be of the same type – or, as we shall see, subtypes of such a type. This is a consequence of Item 21c on the preceding page.

d. it is a sequence all of whose XTrees are type_conformant and all of whose identically labelled XTrees have the same type.

value
24. type_conform: XT → **Bool**
24a. type_conform(xt) ≡
24. **case** xt **of**
24a. μακIN(intg) → **true**,
24b. μακST(text) → **true**,
24c. μακXS(xs) →
24c. ∀ (l′,xt′),(l″,xt″):(L×XT)•{(l′,xt′),(l″,xt″)}⊆**dom** xs ∧
24c. type_conform(xt′) ∧
24c.a l′=l″ ⇒ xtr_type(xt′) = xtr_type(xt″),
24d. μακXL(xl) →
24d. ∀ (l′,xt′),(l″,xt″):(L×XT)•{(l′,xt′),(l″,xt″)}⊆**elems** xl ∧
24d. type_conform(xt′) ∧
24d.a l′=l″ ⇒ xtr_type(xt′)=xtr_type(xt″)
24. **end**

Discussion: Whether, in formula lines 24c.a and 24d.a, to insist on equality of types or to allow one type to be a subtype of the other (whichever way) is a question to be considered.

18.2.6 XTree Subtypes

25. We define a subtype relation as a relation between a pair of type designators:
   a. The XTree integer type is (i.e., designates) a subtype of itself.
   b. The XTree text type is (i.e., designates) a subtype of itself.
   c. Let two multiset type designators be μακMTy(tym′) and μακMTy(tym″). μακMTy(tym′) is (i.e., designates) a subtype of μακMTy(tym″)
      i. if the definition set of labels of μακMTy(tym′) is a subset of the definition set of labels of μακMTy(tym″),
      ii. and, if for identical labels, ℓ, in μακMTy(tym′) and μακMTy(tym′(ℓ)) is (i.e., designates) a subtype of μακMTy(tym″(ℓ)).
   d. Let two sequence type designators be μακSTy(tyl′) and μακSTy(tyl″). μακSTy(tyl′) is (i.e., designates) a subtype of μακSTy(tyl″)
      i. if the length of tyl′ is less than or equal to the length of tyl′,[202]
      ii. if for index positions, i, of tyl′ the labels of the indexed properties tyl′(i) (= (l′,t′)) and tyl″(i) (= (l″,t″)) are the same (l′=l″) and
      iii. type designator t' is (i.e., designates) a subtype of t″.
   e. Only such pairs of types as implied by the above can possibly enjoy a subtype relation.

value
25. is_subtype: XXTy × XXTy → **Bool**
25. is_subtype(ta,tb) ≡
25. **case** (ta,tb) **of**
25a. (mkITy,mkITy) → **true**,
25b. (mkTTy,mkTTy) → **true**,
25c. (μακMTy(tym′),μακMTy(tym′)) →

[202] We could, instead of this "prefix" subtype property, have defined an "embedded" subtype property: that tyl′ is a subtype of a properly embedded sequence of tyl″

```
25(c)i.      dom tym′ ⊆ tym″ ∧
25(c)ii.      ∀ l:L•l ∈ dom tym′ ⇒ is_subtype(tym′(l),tym″(l)),
25d.     (μακSTy(tyl′),μακSTy(tyl′)) →
25(d)i.      len tyl′ ≤  tyl″ ∧
25(d)ii.      ∀ i:Nat • 1≤i≤len tyl′ ⇒
25(d)ii.        let ((l′,t′),(l″,t″))=(tyl′(i),tyl″(i)) in
25(d)ii.        l′=l″ ∧
25(d)iii.        is_subtype(t′,t″) end,
25e.      _ → false
25.    end
```

Please note that if td′ and td″ are type designators, then either td′ denotes a subtype of td″ or td″ denotes a subtype of td′ or neither denotes a subtype of the other.

18.3 XTree Operations

18.3.1 XTree Multiset Union

26. By the union of two multisets we understand their bag (i.e., multiset) union.
    a. For any property which is common to both multisets the multiset union maps the property into the sum of its number of occurrences in the two argument multisets.
    b. For any property which is only in one of the multisets the multiset union contains that property with the number of occurrences designated by that multiset.
    c. Shared label values must be of comparable_types.

```
value
26   XSunion: XS × XS → XS
26a  XSunion(μακXS(xs1),μακXS(xs2)) ≡
26a     μακXS([p↦xs1(p)+xs2(p)|p ∈ dom xs1 ∩ dom xs2]
26b           ∪ xs1\dom xs2 ∪ xs2\dom xs1)
26c   pre: comparable_types(xtr_type(μακXS(xs1)),xtr_type(μακXS(xs2)))
```

18.3.2 Commensurate Multiset Arguments

27. Two multiset values (types) are comparable
28. if for identical (i.e., shared) labels have identical types (are equal);
29. or maybe we should just ask for an appropriate subtype relation.

```
value
27.   comparable_values: XS × XS → Bool
27.   comparable_values(μακXS(lm′),μακXS(lm″)) ≡
28.     ∀ l:L • l ∈ dom lm′ ∩ lm″ ⇒
28.      (xtr_type(lm′(l)) = xtr_type(lm″(l)) ∨
29.       is_subtype(xtr_type(lm′(l)),xtr_type(lm″(l))) ∨
29.       is_subtype(xtr_type(lm″(l)),xtr_type(lm′(l))))
```

value
27. comparable_types: XTTy × XTTy → **Bool**
27. comparable_types($\mu\alpha\kappa$XT(lmt′),$\mu\alpha\kappa$XT(lmt″)) ≡
28. ∀ l:L • l ∈ **dom** lmt′ ∩ lmt″ ⇒
28. (lmt′(l) = lmt″(l) ∨
29. is_subtype(lmt′(l),lmt″(l)) ∨ is_subtype(lmt″(l),lmt′(l)))

18.3.3 Type “Prediction”

30. One can calculate the type of the result of a multiset union from its two arguments:

a.
b.
c.
d.

30.
30a.
30b.
30c.
30d.

18.3.4 A Theorem: Correctness of Type “Prediction”

31. One can prove the following theorem:

a.
b.
c.
d.
e.

31.
31a.
31b.
31c.
31d.
31e.

18.3.5 XTree Multiset Equality

32. Multiset equality is bag equality of the multisets.

value
32 XSequal: XS × XS → **Bool**
32 XSequal($\mu\alpha\kappa$XS(xs1),$\mu\alpha\kappa$XS(xs2)) ≡ xs1 = xs2

18.3.6 XTree Multiset Subset

33. One multiset is a subset of another multiset
    a. if the first has a subset of the properties of the latter and
    b. and, for each property of the first its number of occurrences in the former is equal to or smaller than its number of occurrences in the latter.

value
33 XSsubset: XS × XS → **Bool**
33 XSsubset($\mu\alpha\kappa$XS(xs1),$\mu\alpha\kappa$XS(xs2)) ≡
33a **dom** xs1 ⊆ **dom** xs2 ∧
33b ∀ p:P • p ∈ **dom** xs1 ⇒ xs1(p)≤xs2(p)

18.3.7 Property Multiset Membership

34. A property, p=(l,xt), is in a multiset if it occurs in the multiset with a cardinality higher than 0.

value
34 XSmember: P × XS → **Bool**
34 XSmember(p,$\mu\alpha\kappa$XS(xs)) ≡ p ∈ **dom** xs ∧ xs(p)>0

18.3.8 XTree Multiset Membership

35. An XTree, xt, is a member of a multiset, xs, if there exists a label, ℓ such that the property (ℓ,xt) is a member of xs.

value
35 XSmember: XT × XS → **Bool**
35 XSmember(xt,$\mu\alpha\kappa$XS(xs)) ≡ ∃ l:L • XSmember((l,xt),$\mu\alpha\kappa$XS(xs))

18.3.9 XTree Multiset Cardinality

36. The cardinality of a multiset is the sum total of all the XTrees of distinct properties of that multiset.

value
36 XScard($\mu\alpha\kappa$XS(xs)) ≡
36 **if** xs = [ ] **then** 0
36 **else**
36   **let** (l,xt):P•(l,xt)∈ **dom** xs **in**
36   xs(l,xt) + XScard($\mu\alpha\kappa$XS(xs\{(l,xt)})) **end end**

18.3.10 Arbitrary Selection of XTrees or Properties from Multisets

37. To select an XTree of a multiset
    a. is undefined if the multiset is empty.
    b. If it is not empty then an arbitrary property is chosen from the (definition set of the) multiset and the XTree of that property is yielded.
    c. To select a property of a multiset basically follows the above description.

```
value
37  XSselectXT: XS ~→ XT
37  XSselectXT(μακXS(xs)) ≡
37a   if xs=[]
37a     then chaos
37b     else let (l,xt):P•(l,xt) ∈ dom xs in xt end
37b   end

37  XSselectP: XS ~→ P
37  XSselectP(μακXS(xs)) ≡
37a   if xs=[]
37a     then chaos
37c     else let p:P•p ∈ dom xs in p end
37c   end
```

18.3.11 XTree Multiset Difference

38. The multiset difference of two multisets, xs1 and xs2,
    a. is the multiset where properties that are in both xs1 and xs2 occur in the result with their number of occurrences being their difference, if larger than 0,
    b. to which is joined the multiset of xs1 whose properties are not in xs2.

```
value
38  XTreeDiff: XS × XS → XS
38  XTreeDiff(μακXS(xs1),μακXS(xs2)) ≡
38a    mkXS(rm0([p↦xs1(p)−xs2(p)|p:P•p ∈ dom xs1 ∩ dom xs2])
38b    ∪ xs1\dom xs2)

   rm0: (P →m Int) → (P →m Nat)
   rm0(pmn) ≡ [p↦pmn(p)|p:P•p ∈ dom pmn ∧ pmn(p)>0]
```

18.3.12 XTree List Concatenation

39. The concatenation of two XTree lists is the usual concatenation of lists.
40. Labels, ℓ, common to the two XTree lists must designate XTree, xt1 and xt2 (i.e., properties (ℓ,xt1) and (ℓ,xt2)) where one is a subtype of the pther (i.e., including "vice versa").

value
39 XTreeListConc: XL × XL → XL
39 XTreeListConc($\mu\alpha\kappa$XL(xl1),$\mu\alpha\kappa$XL(xl2)) ≡ $\mu\alpha\kappa$XL(xl1⌢xl2)
40 **pre** ∀ (l1,xtt),(l2,xt2):P•(l1,xt1) ∈ **elems** xl1∧(l2,xt2) ∈ **elems** xl2 ∧ l1=l2 ⇒
40 subtype(xt1,xt2)∨subtype(xt2,xt1)

18.3.13 XTree List Equality

41. The equality of two `XTree` lists is the usual equality of lists.

value
41 XTreeListEqual: XL × XL → **Bool**
41 XTreeListEqual($\mu\alpha\kappa$XL(xl1),$\mu\alpha\kappa$XL(xl2)) ≡ xl1=xl2

18.3.14 XTree List Property Membership

42. A property is a member of an `XTree` list
43. if there is an index into the list which identifies that property.

value
42 XMbrTreeList: P × XL → **Bool**
43 XMbrTreeList(p,$\mu\alpha\kappa$XL(xl)) ≡ ∃ i:**Nat** • i ∈ **inds** xl ∧ p=xl(i)

18.3.15 XTree List `XTree` Membership

44. An `XTree` is a member of an `XTree` list
45. if there is an index into the list which identifies that `XTree`.

value
44 XMbrTreeList: XT × XL → **Bool**
45 XMbrTreeList(xt,$\mu\alpha\kappa$XL(xl)) ≡ ∃ i:**Nat**,l:Label • i ∈ **inds** xl ∧ (l,xt)=xl(i)

18.3.16 XTree List Length

46. The length of an `XTree` list
    a. is the length of the list it contains.

value
46 XTreeListLength: XL → **Nat**
46a XTreeListLength($\mu\alpha\kappa$XL(xl)) ≡ **len** xl

18.3.17 XTree List Head

47. The head, or first, element of an XTree list
    a. is the head property of the list it contains.

value
47 XTreeListHead: XL → P
47a XTreeListHead($\mu\alpha\kappa$XL(xl)) ≡ **if** xl=⟨⟩ **then chaos else hd** xl **end**

18.3.18 XTree List Tail

48. The tail, or rest, of an XTree list
    a. is the tail of the list it contains.

value
48 XTreeListTail: XL → XL
48a XTreeListTail($\mu\alpha\kappa$XL(xl)) ≡ **if** xl=⟨⟩ **then chaos else** $\mu\alpha\kappa$XL(**tl** xl) **end**

18.3.19 XTree List Nth Element

49. The nth element of a list
    a. if n is an index of the list then it is the property indexed by n else it is undefined.

value
49 NthXTreeListElem: **Nat** × XL $\xrightarrow{\sim}$ P
49a NthXTreeListElem(n,$\mu\alpha\kappa$XL(xl)) ≡ **if** 0<n≤**len** xl **then** xl(n) **else chaos end**

18.4 Indexing

18.4.1 Paths and Indexes

50. An index is either a label or a wildcard or a ???
51. non-zero natural number.
52. A path is a finite sequence of zero, one or more indexes.

type
50 Index == $\mu\alpha\kappa$L(l:L) | $\mu\alpha\kappa$WldCrd | $\mu\alpha\kappa$Nat(i:Nat1)
51 Nat1 = {|n:**Nat**•n>0|}
52 Path = Index*

18.4.2 Proper Index

53. We define an is_Index predicate over indexes and Xtrees.
    a. If there is a property, (ℓ,xt), which is in a multiset $\mu\alpha\kappa$XS(xs) then ℓ is an index of that $\mu\alpha\kappa$XS(xs).
    b. If there is an index, j, into the list, xl, of an `XTree` list, $\mu\alpha\kappa$XL(xl), then j is an index of that mkXL(xl);
    c. if, furthermore, there is the property, (ℓ,xt) at list xl position j, then ℓ is an index into mkXL(xl); and
    d. $\mu\alpha\kappa$WldCrd is (always) an index.

value

```
        is_Index: Index × XT → Bool
        is_Index(i,xt) ≡
            case (i,xt) of
53a          (μακL(l),μακXS(xs)) → ≡ ∃ xt′:XT•(l,xt′) ∈ dom xs,
53b          (μακNat(j),μακXL(xl)) → j ∈ inds xl,
53c          (μακL(l),μακXL(xl)) → ∃ j:Nat1,xt′:XT•j ∈ inds xl∧xl(j)=(l,xt′),
53d          (μακWldCrd,_) → true,
              _ → false
            end
```

18.4.3 Index Selecting

54. Given an index it thus may or may not identify an `XTree`, xt′, or a property, p:P, of an argument `XTree`, xt. The definition follows those of Items 53a–53c.

value

```
54   Identify: Index × XT ~→ (XT|P)
54   Identify(i,xt) ≡
54     case (i,xt) of
53a      (μακL(l),μακXS(xs)) → let xt′:XT•(l,xt′)∈ dom xs in xt′ end,
53b      (μακNat1(i),μακXL(xl)) → xl(i),
53c      (μακL(l),μακXL(xl)) → let i:Nat1,xt′:XT•i ∈ inds xl∧xl(i)=(l,xt′) in xt′ end,
53d      (μακWldCrd,μακXS(xs)) → let p:P•p ∈ dom xs in p end,
53d      (μακWldCrd,μακXL(xl)) → hd xl
54     end
54   pre is_Index(i,xt)
```

18.4.4 Path Indexing

55. Given an XTree, xt, a path, pth, may or may not identify an `XTree`, xtr′, of xt. The selection function, Select is defined recursively:
    a. If the path is empty then the argument XTree, xt, is yielded.
    b. If the head of the path is an index of the `XTree`, xt, then the so indexed `XTree`, xt_x, is selected.
    c. Otherwise the path, pth, is ill-defined.

```
value
55   Select: XT × Path ~→ XT | P
55a  Select(xtop,⟨⟩) ≡ xtop
55b  Select(xt,⟨i⟩⌢pth) ≡
55b    if is_Index(i,xt)
55b      then
55b        let e = Identify(i,xt) in
55b        if e:P ∧ pth≠⟨⟩ then chaos end
55b        Select(e,pth) end
55c      else chaos end
```

18.5 Queries

56. An XVSM query is a [piped] sequence of simple XVSM queries.

```
type
56  Q = SQ*
```

18.5.1 Generally on Semantics

57. The idea is the following:

   a. The meaning of a simple XVSM query, sq:SQ, as applied to an XTree, xt:XT, is expressed as MSQ(sq)(xt), and is to be an XTree multiset or an XTree list. **Not an XTree value ?**
   b. The meaning of an XVSM query, q:Q, as applied to an XTree, xt:XT, is expressed as MQ(q)(xt), and is to be an XTree multiset or an XTree list.
   c. The meaning function, MQ, when applied to an empty query, ⟨⟩, is MQ(⟨⟩)(xt), that is, xt.
   d. The meaning function, MQ, when applied to a non-empty query, ⟨sq⟩⌢q, is MQ(q)(MSQ(sq)(xt)).
   e. Both MSQ and MQ may be undefined for some combinations of queries and Xtrees.

```
value
57a  MSQ: SQ → XT ~→ XT
57b  MQ: Q → XT ~→ XT
57b  MSQ(sq) as xt
57c  MQ(⟨⟩)(xt) ≡ xt
57d  MQ(⟨sq⟩⌢q)(xt) ≡ MQ(q)(MSQ(sq)(xt))

57e  MQ(⟨sq⟩⌢q)(xt) ≡
57e    if IS_UNDEFINED(MSQ(sq)(xt))
57e      then IS_UNDEFINED(MQ(⟨sq⟩⌢q)(xt))
57e      else ... to be defined ...
57e    end
```

18.5.2 Syntax: Simple XVSM Queries

58. A simple XVSM query is either a selector query or a matchmaker query.
59. A [simple] selector [XVSM] query is either a predefined selector quiry or ...

type
58 SQ = SelQ | MatchQ
59 SelQ = PreSelQ | ...

18.5.2.1 Syntax: Predefined Selector Queries

60. A predefined selector query is either a count, a sort_up, a sort_down, a reverse, an identity, or a unique (slector) query.
    a. A count query states a non-zero natural number.
    b. A sort up query states a path.
    c. A sort down query states a path.
    d. A reverse query does not present an argument.
    e. An identity query does not present an argument.
    f. A unique (selector) query states a path.

60 PreSelQ = Cnt | SrtUp | SrtDo | Rev | Id | Uniq | ...
60a Cnt == $\mu\alpha\kappa$Cnt(sel_n:**Nat**)
60b SrtUp == $\mu\alpha\kappa$SrtUp(sel_p:Path)
60c SrtDo == $\mu\alpha\kappa$SrtDo(sel_p:Path)
60d Rev == mk_Rev
60e Id == mk_Id
60f Uniq == $\mu\alpha\kappa$Uniq(sel_p:Path)

18.5.2.2 Semantics: Predefined Selector Queries

18.5.2.2.1 Count

61. The $\mu\alpha\kappa$Cnt(n) selector query applies to an XTree, xt, and,
    a. if it is an XTree list and if the list is of length n or more, yields the XTree list $\mu\alpha\kappa$XL(xl′) of the first n properties of xt = $\mu\alpha\kappa$XL(xl), else it yields **chaos**; or
    b. if it is an XTree multiset and if the multiset has at least n properties, yields an XTree multiset, $\mu\alpha\kappa$XS(xs′), of n arbitrarily chosen properties of xt = $\mu\alpha\kappa$XS(xs), else it yields **chaos**.

61 MPreSelQ: PreSelQ → XT $\tilde{\rightarrow}$ XT
61 MPreSelQ($\mu\alpha\kappa$Cnt(n))(xt) ≡
61 **case** xt **of**
61a $\mu\alpha\kappa$XL(xl) →
61a **if len** xl≥n **then** $\mu\alpha\kappa$XL(⟨xl(i)|i:**Nat**•i:[1..n]⟩) **else chaos end**,
61b $\mu\alpha\kappa$XS(xs) →
61b **if card dom** xs≥n
61b **then let** ps:P-**set**•**card** ps=n ∧ ps⊆**dom** xs **in**
61b $\mu\alpha\kappa$XS([p↦xs(p)|p:P•p ∈ ps]) **end**
61b **else chaos**

```
61b        end,
61b      _ → chaos
61b   end
```

18.5.2.2.2 Sort Up

62. The $\mu\alpha\kappa$SrtUp selector query applies to a (relative) path, pth⁀ℓ, and an `XTree`, xt.
    a. First we Select from xt the `XTree`, xt′′, identified by the path pth.
    b. The selected `XTree`, xt′′, is either a list or a multiset.
    c. The result of MPreSelQ($\mu\alpha\kappa$SrtUp(pth⁀ℓ))(xt) is the `XTree` list xt′ which has all the entries that xt has except that these are now ordered with respect to the ordering of the ℓ values of xt′′.

```
value
62   MPreSelQ: SrtUp → XT → XL
62   MPreSelQ(μακSrtUp(pth⁀ℓ))(xt) ≡
62a    let xt′′ = Select(xt)(pth) in
62b    let vl =


62c    end end
```

MUCH MORE TO COME

Part IV
Bibliography

Chapter 19
Bibliography

Contents

19.1 Bibliographical Notes 513
19.2 References 513

19.1 Bibliographical Notes

TO BE WRITTEN

19.2 References

1. Martin Abadi and Luca Cardelli. A Theory of Objects. Monographs in Computer Science. Springer–Verlag, New York, NY, USA, August 1996.
2. Jean-Raymond Abrial. The B Book: Assigning Programs to Meanings and Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge, England, 1996 and 2009.
3. Rober Audi. The Cambridge Dictionary of Philosophy. Cambridge University Press, The Pitt Building, Trumpington Street, Cambridge CB2 1RP, England, 1995.
4. Mordecai Avriel, Michal Penn, and Naomi Shpirer. Container ship stowage problem: complexity and connection to the coloring of circle graphs. Discrete Applied Mathematics, 103(1–3):271–279, 15 July 2000. Faculty of Industrial Engineering and Management, Technion, Israel Institute of Technology, Haifa 3200, Israel.
5. Mordecai Avriel, Michal Penn, Naomi Shpirer, and Smadar Witteboon. Stowage planning for container ships to reduce the number of shifts. Annals of Operations Research, 76(9):55–71, January 1998.
6. David Basin and Se'an Matthews. A conservative extension of first-order logic and its applications to theorem proving. In FSTTCS 1993: Foundations of Software Technology and Theoretical Computer Science, volume 761 of Lecture Notes in Computer Science, pages 151–160. Springer, 2005.
7. H. Bekič, D. Bjørner, W. Henhapl, C. B. Jones, and P. Lucas. A Formal Definition of a PL/I Subset. Technical Report 25.139, IBM Laboratory, Vienna, December 1974.
8. Hans Bekič, Peter Lucas, Kurt Walk, and Many Others. Formal Definition of PL/I, ULD Version I. Technical report, IBM Laboratory, Vienna, 1966.
9. Hans Bekič, Peter Lucas, Kurt Walk, and Many Others. Formal Definition of PL/I, ULD Version II. Technical report, IBM Laboratory, Vienna, 1968.
10. Hans Bekič, Peter Lucas, Kurt Walk, and Many Others. Formal Definition of PL/I, ULD Version III. IBM Laboratory, Vienna, 1969.
11. Claude Berge. Théorie des Graphes et ses Applications. Collection Universitaire de Mathematiques. Dunod, Paris, 1958. See [12].
12. Claude Berge. Graphs, volume 6 of Mathematical Library. North-Holland Publ. Co., second revised edition of part 1 of the 1973 english version edition, 1985. See [11].

13. Sandford Bessler, Eva Kühn, Richard Mordinyi, and Slobodanka Tomic. Using tuple-spaces to manage the storage and dissemination of spatial-temporal content. Journal of Computer and System Sciences, page 10, February 2010. Link: http://dx.doi.org/10.1016/j.jcss.2010.01.010.
14. D. Bjørner. Stepwise Transformation of Software Architectures. In [65], chapter 11, pages 353–378. Prentice-Hall, 1982.
15. Dines Bjørner. Software Development Graphs — A Unifying Concept for Software Development? In K.V. Nori, editor, Vol. 241 of Lecture Notes in Computer Science: Foundations of Software Technology and Theoretical Computer Science, pages 1–9. Springer–Verlag, Dec. 1986.
16. Dines Bjørner. The Stepwise Development of Software Development Graphs: Meta-Programming VDM Developments. In See [66], volume 252 of LNCS, pages 77–96. Springer-Verlag, Heidelberg, Germany, March 1987.
17. Dines Bjørner. Specification and Transformation: Methodology Aspects of the Vienna Development Method. In TAPSOFT'89, volume 352 of Lab. Note, pages 1–35. Springer-Verlag, Heidelberg, Germany, 1989.
18. Dines Bjørner. Software Systems Engineering — From Domain Analysis to Requirements Capture: An Air Traffic Control Example. In 2nd Asia-Pacific Software Engineering Conference (APSEC '95). IEEE Computer Society, 6–9 December 1995. Brisbane, Queensland, Australia.
19. 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.
20. 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.
21. 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.
22. Dines Bjørner. New Results and Trends in Formal Techniques for the Development of Software for Transportation Systems. In FORMS2003: Symposium on Formal Methods for Railway Operation and Control Systems. Institut für Verkehrssicherheit und Automatisierungstechnik, Techn.Univ. of Braunschweig, Germany, 15–16 May 2003. Conf. held at Techn.Univ. of Budapest, Hungary. Editors: G. Tarnai and E. Schnieder, Germany. www2.imm.dtu.dk/~dibj/dines-amore.pdf.
23. Dines Bjørner. The Grand Challenge – FAQs of the R&D of a Railway Domain Theory. In IFIP World Computer Congress, Topical Days: TRain: The Railway Domain, IFIP, Amsterdam, The Netherlands, 2004. Kluwer Academic Press.
24. Dines Bjørner. Towards a Formal Model of CyberRail. In Building the Information Society, IFIP 18th World Computer Congress, Tpical Sessions, 22–27 August, 2004, Toulouse, France — Ed. Renéne Jacquart, pages 657–664. Kluwer Academic Publishers, August 2004.
25. Dines Bjørner. Software Engineering, Vol. 1: Abstraction and Modelling. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. See [29, 32].
26. 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 [30, 33].
27. Dines Bjørner. Software Engineering, Vol. 3: Domains, Requirements and Software Design. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. See [31, 34].
28. 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.
29. Dines Bjørner. Software Engineering, Vol. 1: Abstraction and Modelling. Qinghua University Press, 2008.
30. Dines Bjørner. Software Engineering, Vol. 2: Specification of Systems and Languages. Qinghua University Press, 2008.
31. Dines Bjørner. Software Engineering, Vol. 3: Domains, Requirements and Software Design. Qinghua University Press, 2008.
32. Dines Bjørner. **Chinese:** Software Engineering, Vol. 1: Abstraction and Modelling. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.
33. Dines Bjørner. **Chinese:** Software Engineering, Vol. 2: Specification of Systems and Languages. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.
34. Dines Bjørner. **Chinese:** Software Engineering, Vol. 3: Domains, Requirements and Software Design. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.
35. 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.
36. 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.

37. 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.
38. 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.
39. 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.
40. Dines Bjørner. Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions. In Rainbow of Computer Science, Festschrift for Hermann Maurer on the Occasion of His 70th Anniversary., Festschrift (eds. C. Calude, G. Rozenberg and A. Saloma), pages 167–183. Springer, Heidelberg, Germany, January 2011. www.imm.dtu.dk/˜dibj/maurer-bjorner.pdf.
41. Dines Bjørner. Documents – a Domain Description. Experimental Research Report 2013-3, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013.
42. 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.
43. Dines Bjørner. Road Transportation – a Domain Description www.imm.dtu.dk/˜dibj/road-p.pdf. Experimental Research Report 2013-4, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013.
44. 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.
45. 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.
46. Dines Bjørner. Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions. Extensive revision of [40]. www.imm.dtu.dk/ dibj/2016/demos/faoc-demo.pdf. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, 2016.
47. 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.
48. 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.
49. Dines Bjørner. Domain analysis & description - the implicit and explicit semantics problem www.imm.dtu.dk/ dibj/2017/bjorner-impex.pdf. In Régine Laleau, Dominique Méry, Shin Nakajima, and Elena Troubitsyna, editors, Proceedings Joint Workshop on Handling IMPlicit and EXplicit knowledge in formal system development (IMPEX) and Formal and Model-Driven Techniques for Developing Trustworthy Systems (FM&MDD), Xi'An, China, 16th November 2017, volume 271 of Electronic Proceedings in Theoretical Computer Science, pages 1–23. Open Publishing Association, 2018.
50. Dines Bjørner. Domain Facets: Analysis & Description. Extensive revision of [35]. www.imm.dtu.dk/ dibj/2016/facets/faoc-facets.pdf. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, May 2018.
51. Dines Bjørner. To Every Manifest Domain a CSP Expression www.imm.dtu.dk/ dibj/2016/mereo/mereo.pdf. Journal of Logical and Algebraic Methods in Programming, 1(94):91–108, January 2018.
52. 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.
53. Dines Bjørner. Domain Analysis & Description – Principles, Techniques and Modelling Languages. www.imm.dtu.dk/˜dibj/2018/tosem/Bjorner-TOSEM.pdf. ACM Trans. on Software Engineering and Methodology, 28(2), April 2019. 68 pages.
54. Dines Bjørner. Domain Analysis & Description: Sorts, Types, Intents. www.imm.dtu.dk/ dibj/2019/ty+so/HavelundFestschriftOctober2020.pdf. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, November 2019. Paper for Klaus Havelund Festschrift, October 2020.
55. Dines Bjørner. Domain Science & Engineering – A Foundation for Software Development. EATCS Monographs in Theoretical Computer Science. Springer, 2021.
56. Dines Bjørner. *Rigorous Domain Descriptions*. A compendium of draft domain description sketches carried out over the years 1995–2021. Chapters cover: *Graphs*, *Railways*, *Road Transport The "7 Seas"*, *The "Blue Skies"*, *Credit Cards Weather Information*, *Documents*, *Urban Planning*, *Swarms of Drones*, *Container Terminals*, *A Retailer Market*, *Shipping*, *Rivers*, *Canals*, *Stock Exchangew*, and *Web Transactions*. This document is currently being edited. Own: www.imm.dtu.dk/ dibj/2021/dd/dd.pdf, Fredsvej 11, DK-2840 Holte, Denmark, Fall 2021.
57. Dines Bjørner. [59] Chap. 10: Towards a Family of Script Languages – – Licenses and Contracts – Incomplete Sketch, pages 283–328. JAIST Press, March 2009.
58. Dines Bjørner. [59] Chap. 7: Documents – A Rough Sketch Domain Analysis, pages 179–200. JAIST Press, March 2009.

59. Dines Bjørner. Domain Engineering: Technology Management, Research and Engineering. A JAIST Press Research Monograph # 4, 536 pages, March 2009.
60. Dines Bjørner. Domain Case Studies:

    - 2021: *Shipping*, April 2021. www.imm.dtu.dk/˜dibj/2021/ral/ral.pdf
    - 2021: *Rivers and Canals – Endurants – A Technical Note*, March 2021. www.imm.dtu.dk/˜dibj/2021/-Graphs/Rivers-and-Canals.pdf
    - 2021: *A Retailer Market*, January 2021. www.imm.dtu.dk/˜dibj/2021/Retailer/BjornerHeraklit27-January2021.pdf
    - 2019: *Container Terminals*, ECNU, Shanghai, China www.imm.dtu.dk/˜dibj/2018/yangshan/-maersk-pa.pdf
    - 2018: *Documents*, TongJi Univ., Shanghai, China www.imm.dtu.dk/˜dibj/2017/docs/docs.pdf
    - 2017: *Urban Planning*, TongJi Univ., Shanghai, China www.imm.dtu.dk/˜dibj/2018/BjornerUrbanPlanning24Jan2018.pdf
    - 2017: *Swarms of Drones*, Inst. of Softw., Chinese Acad. of Sci., Peking, China www.imm.dtu.-dk/˜dibj/2017/swarms/swarm-paper.pdf
    - 2013: *Road Transport*, Techn. Univ. of Denmark www.imm.dtu.dk/˜dibj/road-p.pdf
    - 2012: *Credit Cards*, Uppsala, Sweden www.imm.dtu.dk/˜dibj/2016/credit/accs.pdf
    - 2012: *Weather Information*, Bergen, Norway www.imm.dtu.dk/˜dibj/2016/wis/wis-p.pdf
    - 2010: *Web-based Transaction Processing*, Techn. Univ. of Vienna, Austria www.imm.dtu.dk/˜dibj/-wfdftp.pdf
    - 2010: *The Tokyo Stock Exchange*, Tokyo Univ., Japan www.imm.dtu.dk/˜db/todai/tse-1.pdf, www.imm.dtu.dk/˜db/todai/tse-2.pdf
    - 2009: *Pipelines*, Techn. Univ. of Graz, Austria www.imm.dtu.dk/˜dibj/pipe-p.pdf
    - 2007: *A Container Line Industry Domain*, Techn. Univ. of Denmark www.imm.dtu.dk/˜dibj/container--paper.pdf
    - 2002: *The Market*, Techn. Univ. of Denmark www.imm.dtu.dk/˜dibj/themarket.pdf
    - 1995–2004: *Railways*, Techn. Univ. of Denmark - a compendium www.imm.dtu.dk/˜dibj/train--book.pdf

    Experimental research reports carried out to “discover”, try-out and refine method principles, techniques and tools, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark.
61. 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.
62. Dines Bjørner, Christian Gram, Ole N. Oest, and Leif Rystrøm. Dansk Datamatik Center. In 3rd IFIP WG 9.7 Working Conference on History of Nordic Computing, IFIP Advances in Information and Communication Technology, pages 2–34. Springer, 2010.
63. Dines Bjørner and Cliff B. Jones, editors. The Vienna Development Method: The Meta-Language, volume 61 of LNCS. Springer, 1978.
64. Dines Bjørner and Cliff B. Jones, editors. The Vienna Development Method: The Meta-Language, volume 61 of LNCS. Springer, 1978. This was the first monograph on Meta-IV.
65. Dines Bjørner and Cliff B. Jones, editors. Formal Specification and Software Development. Prentice-Hall, 1982.
66. Dines Bjørner, Cliff B. Jones, Micheal Mac an Airchinnigh, and Erich J. Neuhold, editors. VDM – A Formal Method at Work. Proc. VDM-Europe Symposium 1987, Brussels, Belgium, Springer, Lecture Notes in Computer Science, Vol. 252, March 1987.
67. Dines Bjørner and Ole N. Oest, editors. Towards a Formal Description of Ada, volume 98 of LNCS. Springer, 1980.
68. Dines Bjørner, Søren Prehn, and Chris W. George. Formal Models of Railway Systems: Domains. Technical report, Dept. of IT, Technical University of Denmark, Bldg. 344, DK–2800 Lyngby, Denmark, September 23 1999. Presented at the FME Rail Workshop on Formal Methods in Railway Systems, FM'99 World Congress on Formal Methods, Toulouse, France. Avaliable on CD ROM.
69. Dines Bjørner, Søren Prehn, and Chris W. George. Formal Models of Railway Systems: Requirements. Technical report, Dept. of IT, Technical University of Denmark, Bldg. 344, DK–2800 Lyngby, Denmark, September 23 1999. Presented at the FME Rail Workshop on Formal Methods in Railway Systems, FM'99 World Congress on Formal Methods, Toulouse, France. Avaliable on CD ROM.
70. 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.

71. 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.
72. Andrzej Blikle and Mikkel Thorup. On conservative extensions of syntax in the process of system development. In Proceedings of VDM'90, VDM and Z—Formal Methods in Software Development, volume 428 of Lecture notes in computer science, page 22. Springer, 1990.
73. J.A. Bondy and U.S.R. Murty. Graph Theory with Applications. American Elsevier, N.Y. and MacMillan, London, 1976.
74. Bram Borgman, Eelco van Asperen, and Rommert Dekker. Online rules for container stacking. OR Spectrum, 32:687–716, 19 March 2010.
75. Roberto Casati and Achille C. Varzi. Parts and Places: the structures of spatial representation. MIT Press, 1999.
76. Stefan Craß. A Formal Model of the Extensible Virtual Shared Memory (XVSM) and its Implementation in Haskell – Design and Specification. M.sc., Technische Universität Wien, A-1040 Wien, Karlsplatz 13, Austria, Febrauary 5 2010.
77. Steran Craß, Eva Kühn, and Gernot Salzer. Algebraic Foundation of a Data Model for an Extensible Space-based Collaboration Protocol. In Bipin C. Desai, editor, IDEAS 2009, pages 301–306, Cetraro, Calabria, Italy, September 16–18 2009.
78. Dieter Klaua. Über einen ansatz zur mehrwertigen mengenlehre. Monatsbreicht, 7:859867, 1965.
79. Opher Dubrovsky, Gregory Levitin, and Michal Penn. A genetic algorithm with a compact solution encoding for the container ship stowage problem. Journal of Heuristics, 8(6):585–599, November 2002.
80. Ali Enayat. Conservative extensions of models of set theory and generalizations. The Journal of Symbolic Logic, 51(4):1005–1021, December 1986.
81. S. Even. Graph Algorithms. Computer Science Press, Md., USA, 1979.
82. Bureau Export. A-Z Dictionary of Export, Trade and Shipping Terms. www.exportbureau.com/trade_shipping_terms/dictionary.html, 2007.
83. Peter Fettke and Wolfgang Reisig. Modelling service-oriented systems and cloud services with HERAKLIT. CoRR, abs/2009.14040, 2020.
84. Peter Fettke and Wolfgang Reisig. HERAKLIT – epistemologically motivated modeling of computer-integrated systems. HERAKLIT working paper, v1, December 15, 2020, http://www.heraklit.org, 2020.
85. Peter Fettke and Wolfgang Reisig. HERAKLIT case study: 8-second hell. HERAKLIT working paper, v1, December 12, 2020, http://www.heraklit.org, 2020.
86. Peter Fettke and Wolfgang Reisig. HERAKLIT case study: adder. HERAKLIT working paper, v1, December 5, 2020, http://www.heraklit.org, 2020.
87. Peter Fettke and Wolfgang Reisig. HERAKLIT case study: parallel adder. HERAKLIT working paper, v1, December 5, 2020, http://www.heraklit.org, 2020.
88. Peter Fettke and Wolfgang Reisig. HERAKLIT case study: retailer. HERAKLIT working paper, v1, December 21, 2020, http://www.heraklit.org, 2020.
89. Peter Fettke and Wolfgang Reisig. HERAKLIT case study: service system. HERAKLIT working paper, v1, November 20, 2020, http://www.heraklit.org, 2020.
90. 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.
91. Arve Gengelbach and Tjark Weber. Model-theoretic conservative extension for definitional theories. Electronic Notes Theoretical Computer Science, (338):133–145, 2017.
92. 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.
93. 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.
94. Jean-Yves Girard, Y. Lafont, and P. Taylor. Proofs and Types, volume 7. Cambridge Univ. Press, Cambridge, UK, Cambridge Tracts in Theoretical Computer Science edition, 1989.
95. David Harel. Statecharts: A visual formalism for complex systems. Science of Computer Programming, 8(3):231–274, 1987.
96. David Harel and Rami Marelly. Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine. Springer-Verlag, 2003.
97. Frank Harrary. Graph Theory. Addison Wesley Publishing Co., 1972.
98. Charles Anthony Richard Hoare. Communicating Sequential Processes. Communications of the ACM, 21(8), Aug. 1978.
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.

101. 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).
102. Charles Anthony Richard Hoare. Communicating Sequential Processes. Published electronically: usingcsp.com/cspbook.pdf, 2004. Second edition of [100]. See also usingcsp.com/.
103. Lloyd Humberstone. On a conservative extension argument of dana scott. Logic Journal of the IGPL, 19(1):241–288, February 2011.
104. Akio Imai, Kazuya Sasaki, Etsuko Nishimura, and Stratos Papadimitriou. Multi-objective simultaneous stowage and load planning for a container ship with container rehandle in yard stacks. European Journal of Operational Research, 171:373–389, 2006.
105. ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992, 1996, 1999.
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. James J. Buckley and Esfanidar Eslami. An Introduction to Fuzzy Logic and Fuzzy Sets. Springer, 2002.
109. C. B. Jones. Software Development: A Rigorous Approach. Prentice-Hall, 1980.
110. C. B. Jones. Systematic Software Development — Using VDM. Prentice-Hall, 1986.
111. C. B. Jones. Systematic Software Development — Using VDM, 2nd Edition. Prentice-Hall, 1989.
112. Dieter Klaus. The Logic of Fuzzy Set Theory: A Historical Approach, page 22 pages. Springer, Heidelberg Germany, 2014. www.researchgate.net/publication/266736510_The_Logic_of_Fuzzy_Set_Theory_A_Historical_Approach.
113. Eva Kühn, Richard Mordinyi, László Keszthelyi, and Christian Schreiber. Introducing the Concept of Customizable Structued Space for Agent Coordination in the Production of Automation Domain. In Sierra Decker, Sichman and Castelfranchi, editors, 8th Intl. Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS 2009), volume 625–632 of Proceedings of Autonomous Agents and Multi-Agent Systems, Budapest, Hungary, May 10–15 2009. 8.
114. Eva Kühn, Richard Mordinyi, László Keszthelyi, Christian Schreiber, Sandford Bessler, and Slobodanka Tomic. Aspect-oriented Space Containers for Efficient Publish/Subscribe Scenarios in Intelligent Transportation Systems. In T. Dillon and P. HereroR. Meersmann, editors, OTM 2009, Part I, volume 5870 of LNCS, pages 432–448. Springer, 2009.
115. J.A.N. Lee and W. Delmore. The Vienna Definition Language, a generalization of instruction definitions. In SIGPLAN Symp. on Programming Language Definitions, San Francisco, Aug. 1969.
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. P. Lucas. Formal Definition of Programming Languages and Systems. In Proc. IFIP'71. IFIP World Congress Proceedings, Springer, 1971.
118. P. Lucas. On the Semantics of Programming Languages and Software Devices. In Rustin, editor, Formal Semantics of Programming Languages. Prentice-Hall, 1972.
119. P. Lucas. On the formalization of programming languages: Early history and main approaches. In D. Bjørner and C. B. Jones, editors, [64]. Springer, 1978.
120. P. Lucas. Formal Semantics of Programming Languages: VDL. IBM Journal of Devt. and Res., 25(5):549–561, 1981.
121. P. Lucas. Main approaches to formal specification. In [14], chapter 1, pages 3–24. Prentice-Hall, 1982.
122. P. Lucas. Origins, hopes, and achievements. In [66], pages 1–18. Springer, 1987.
123. P. Lucas and K. Walk. On the Formal Description of PL/I. Annual Review Automatic Programming Part 3, 6(3), 1969.
124. Usama Mehmood, Radu Grosu, Ashish Tiwari, Nicola Paoletti, Shan Lin, Yang JunXing, Dung Phan, Scott D. Stoller, and Scott A. Smolka. *Declarative vs Rule-based Control for Flocking Dynamics*. In Proceedings of ACM/SIGAPP Symposium on Applied Computing (SACC 2018). ACM Press, April 9–13, 2018. 8 pages.
125. Lev Nachmanson. Microsofts Automated Layout Tool. Technical report, MS Research, 2021. https://github.com/microsoft/automated-graph-layout.
126. Reza Olfati-Saber. Flocking for Multi-agent Dynamic Systems: Algorithms and Theory. IEEE Transactions on Automatic Control, 51(3):401–420, 13 March 2006. http://ieeexplore.ieee.org/document/1605401/; DOI: 10.1109/TAC.2005.864190; Thayer School of Engineering, Dartmouth College, Hanover, NH, USA.
127. Oystein Ore. Graphs and their Uses . The Mathematical Association of America, 1963.
128. International Labour Organisation. Portworker Development Programme: PDP Units. Enumerate PDP units. , April 2002.
129. Benjamin Pierce. Types and Programming Languages. The MIT Press, 2002.
130. Martin Pěnička and Dines Bjørner. From Railway Resource Planning to Train Operation — a Brief Survey of Complementary Formalisations. In Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions, 22–27 August, 2004, Toulouse, France — Ed. Renéne Jacquart, pages 629–636. Kluwer Academic Publishers, August 2004.

131. 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.
132. K.V. Ramani. An interactive simulation model for the logistics planning of container operations in seaports. SIMULATION, 66(5):291–300, 1996.
133. Wolfgang Reisig. Petri Nets: An Introduction, volume 4 of EATCS Monographs in Theoretical Computer Science. Springer Verlag, May 1985.
134. Wolfgang Reisig. A Primer in Petri Net Design. Springer Verlag, March 1992. 120 pages.
135. Wolfgang Reisig. The Expressive Power of Abstract State Machines. Computing and Informatics, 22(1–2), 2003.
136. Wolfgang Reisig. Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien. Leitfäden der Informatik. Vieweg+Teubner, 1st edition, 15 June 2010. 248 pages; ISBN 978-3-8348-1290-2.
137. Wolfgang Reisig. Understanding Petri Nets Modeling Techniques, Analysis Methods, Case Studies. Springer, 2013. 230+XXVII pages, 145 illus.
138. Craig Reynolds. *Flocks, Herds and Schools: A Distributed Behavioral Model*. SIGGRAPH Computer Graphics, 21(4), August 1987. https://doi.org/10.1145/37402.37406.
139. Craig Reynolds. *Steering Behaviors for Autonomous Characters*. In Proceedings of Game Developers Conference, pages 763–782, 1999.
140. Craig Reynolds. OpenSteer, *Steering Behaviours for Autonomous Characters*, 2004. http://opensteer.sourceforge.net.
141. 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.
142. Douglas T. Ross. Toward foundations for the understanding of type. In Proceedings of the 1976 conference on Data: Abstraction, definition and structure, pages 63–65, New York, NY, USA, 1976. ACM. http://doi.acm.org/10.1145/800237.807120.
143. David A. Schmidt. The Structure of Typed Programming Languages. MIT Press, 1994. ISBN 0262193493.
144. 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.
145. Joseph R. Schoenfeld. Mathematical Logic. Addison-Wesley Publishing Company, 1967. (On Conservative Extensions, pp 55-56).
146. Kai Sørlander. Det Uomgængelige – Filosofiske Deduktioner [The Inevitable – Philosophical Deductions, with a foreword by Georg Henrik von Wright]. Munksgaard · Rosinante, 1994. 168 pages.
147. Kai Sørlander. Under Evighedens Synsvinkel [Under the viewpoint of eternity]. Munksgaard · Rosinante, 1997. 200 pages.
148. Kai Sørlander. Den Endegyldige Sandhed [The Final Truth]. Rosinante, 2002. 187 pages.
149. Kai Sørlander. Indføring i Filosofien [Introduction to The Philosophy]. Informations Forlag, 2016. 233 pages.
150. Dirk Steenken, Stefan Voß, and Robert Stahlbock. Container terminal operation and operations research - a classification and literature review. OR Spectrum, 26(1):3–49, January 2004.
151. 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.
152. Tetsuo Tamai. Social Impact of Information System Failures. Computer, IEEE Computer Society Journal, 42(6):58–65, June 2009.
153. I.D. Wilson and P.A. Roach. Container stowage planning: a methodology for generating computerised solutions. Journal of the Operational Research Society, 51(11):1248–1255, 1 November 2000. Palgrave Macmillan. University of Glamorgan, UK.
154. I.D. Wilson, P.A. Roach, and J. A. Ware. Container stowage pre-planning: using search to generate solutions, a case study. Knowledge-Based Systems, 14(3–4):137–145, June 2001.
155. James Charles Paul Woodcock and James Davies. Using Z: Specification, Proof and Refinement. Prentice Hall International Series in Computer Science, 1996.
156. Lotfi A. Zadeh. Fuzzy sets. Information and Control, 8(3):338–353, 1965.

Part V
Appendix

Appendix A
Domain Analysis & Description: A Primer

Contents

A.1 Domains ... 524
A.2 Endurants ... 524
A.2.1 External Qualities ... 525
A.2.1.1 A Classification Calculus ... 525
A.2.1.1.1 Classification Predicates ... 525
A.2.1.1.2 A Classification Calculus ... 526
A.2.1.2 An Observer Calculus ... 526
A.2.1.2.1 Classifiers versus Observers ... 527
A.2.1.3 A Description Calculus ... 527
A.2.1.3.1 Endurant Describers ... 527
A.2.1.4 A State of Endurants ... 529
A.2.2 Internal Qualities ... 530
A.2.2.1 Unique Identification ... 530
A.2.2.1.1 The Unique Identifier Observer ... 530
A.2.2.1.2 The Unique Identifier Describer ... 530
A.2.2.2 Mereology ... 531
A.2.2.2.1 The Mereology Observer ... 531
A.2.2.2.2 The Mereology Describer ... 531
A.2.2.3 Attributes ... 532
A.2.2.3.1 The Attribute Observers ... 532
A.2.2.3.2 The Attribute Categories ... 533
A.2.2.3.3 Attribute Pragmatics ... 533
A.2.2.3.4 The Attributes Describer ... 533
A.2.2.4 Intentional "Pull" ... 535
A.2.3 Narratives ... 535
A.3 Space, State and Time ... 535
A.3.1 Space ... 536
A.3.1.1 Spatial Types ... 536
A.3.1.2 Spatial Attributes ... 536
A.3.2 State ... 536
A.3.3 Time ... 537
A.3.3.1 Time and Time Interval Sorts and Operators ... 537
A.3.3.2 The Time Observer ... 537
A.4 Perdurants ... 537
A.4.1 Actions, Events and Behaviours ... 538
A.4.2 Determinacy and Non-determinacy ... 538
A.4.3 Co-operating Domains ... 539
A.4.4 Transcendental Deduction ... 539
A.4.5 Channels ... 540
A.4.6 Behaviours ... 540
A.4.6.1 Behaviour Signatures ... 540
A.4.6.2 Behaviour Definition 'Bodies' ... 541
A.4.6.2.1 Semiformal Examples of Behaviour Interactions ... 541
A.4.7 Compilation of Domain Descriptions ... 543
A.4.8 Initialising Domains ... 544

A.1 Domains

By a **domain** we shall understand a **rationally describable** area 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"] parts, and its ***living species entities***: ***plants*** and ***animals*** including, notably, ***humans*** [55, Sect. 4.2, Defn. 27] ■

A phenomenon, ϕ, is an ***entity***, `is_entity(`ϕ`)`, if it 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] ■

There are an indefinite number of entities in any domain. This follows from philosophic-analytic reasoning outlined by the philosopher Kai Sørlander [146, 147, 148, 149]. We refer to [55, Sect. 2.2.3] for a summary.

By an *endurant*, `is_endurant(`e`)`, we shall understand an entity, e, 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.

By a *perdurant*, `is_perdurant(`e`)`, we shall understand an entity, e, 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] ■

External qualities of endurants of a manifest domain are, in a simplifying sense, those we, for example with our eyes blinded, can touch, hence manifestedly "observe", and hence speak about abstractly. We shall say more about external qualities in Sect. **A.2.1**.

Internal qualities of endurants of a manifest domain are those we, with our eyes open and with instruments, can measure ■ We shall say more about internal qualities in Sect. **A.2.2**.

A.2 Endurants

We shall now present the analysis & description calculi, that we, as humans, i.e., scientists and engineers, deploy when observing a domain. As we shall see those calculi relate to the upper ontology of domain descriptions such as shown in Fig. **A**.1.

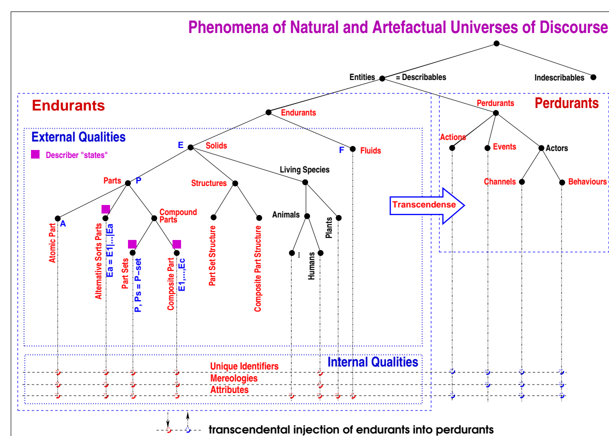


Fig. A.1 An Upper Ontology

A.2.1 External Qualities

Our treatment of endurants "follow" the upper ontology of Fig. **A**.1 in a left-to-right, depth-first traversal of the endurant "tree" (of Fig. **A**.1).

A.2.1.1 A Classification Calculus

A.2.1.1.1 Classification Predicates

Endurants, e [is_endurant(e)], are either **solid** [is_solid(e)]; or **fluid** [is_fluid(e)] (such as liquids, gases and plasmas). Solid endurants appears to be the "work-horse" of the domains we shall be concerned with.

A **solid**, e, is either a **part** [is_part(e)]; or a **structure** [is_structure(e)]; or a **living species** [is_living_species(e)].

A **part**, p, is either an **atomic part** [is_atomic_part(p)]; or an **alternative sorts part** [is_alternative_sorts_part(p)]; or a **compound part** [is_compound_part(p)].

An **atomic part**, by definition, has no proper sub-parts. It is the domain analyser cum describer who decides which parts are atomic and which not. Atomic parts are further characterised by their internal qualities.

An **alternative sorts part** is a part which is of either of two or more distinct sorts; alternative sorts parts may be atomic – so analysis leading, eventually to a decision on atomicity is preceded by one of inquiring as to alternate "sorting" !

A **compound part** is either a **part set** [is_part_set(p)], or a **composite part** [is_composite(p)];.

A **part set** is a set of parts of some sort.

A **composite** part consists of two or more parts (and could be modeled as a Cartesian of these).

A **structure** is either a part set structure [is_part_set_structure]; or a composite structure [is_composite_structure].

A **part set structure** is a set of parts.

A **composite structure** consists of two or more parts (and could be modeled as a Cartesian of these).

A **living species** is either an **animal** [is_animal(e)]; or a **plant** [is_plant(e)].

An **animal** is either a **human** [is_human(e)]; or other.

Fluids are presently further un-analysed.

A Note on Living Species 'Parts': In this primer we could, but do not, consider living species as parts in the sense of later, in this primer, considering their transcendental "morphing" into behaviours. The reader is invited to fill in the necessary details !

A Note on Structures: The distinction between non-structure and structure compounds is one of pragmatics. For non-structure compounds we shall ascribe internal qualities, i.e., unique identifiers, mereologies and attributes, to their parts, for structure compounds we shall not. Eventually non-structure parts will also be represented as behaviours, structure parts not. In Fig. **A**.1 this latter is indicated by their being no dashed vertical lines from structure sets and composites connecting to horisontal internal quality lines. Section **A.2.1.4** should clarify this issue.

■ .... **Begin Example 0: Some Examples**

An automobile is a solid and a part. The fuel contents of an automobile is a fluid. The amalgam of a road net and the amalgam of a set of automobiles is here seen as a structure. Water, gasoline and beer are examples of liquids. The air in a room is an example of a gas. Ionized gases are plasmas, such as that of a neon tube. From the point of view of an owner, an automobile is an

atomic part; from the point of view of an auto manufacturer and an auto maintenance & repair shop, an automobile is a composite part.
.... **End of Example 0** ■

A.2.1.1.2 A Classification Calculus

We summarise the predicates:

- is_entity,
- is_endurant,
- is_perdurant,
- is_solid,
- is_fluid,
- is_part,
- is_structure,
- is_part_set_structure,
- is_composite_structure,
- is_living_species,
- is_atomic,
- is_alternative_sort ■,
- is_compound,
- is_set ■,
- is_composite ■,
- is_animal,
- is_human,
- is_..., and
- is_plant.

The classifier predicates are *mental tools* to be used by domain analysers & describers. They are, so to speak, part of their *"mental luggage"*. Domain analysers & describers analyse the domain. They identify entities; they focus first on endurants (and, later, perdurants); and they apply the above-listed predicates, one-by-one, in order to classify what is being analysed. In Fig. **A**.1 **magenta**-coloured squares, ■s, are affixed to a number of ["leaf"] nodes. Once endurants are classified into either of these **magenta**–coloured classes we can start *observing* and *describing* these endurants.

• An Aside: Types and Sorts

By a **type** we shall understand a set of (further characerised) values. By a **concrete type** the "further characterisation" may amount to typing the values as being for example **integer**s, or **Boolean**s, or **sets** or **Cartesians** of otherwise typed values, or total or partial **functions** or **maps** from otherwise typed definition set values to typed range values, etc. By **sort** we shall understand an **abstract type**, i.e., a type which is not concrete, one whose "inner constellation" is not revealed.

• A Last Aside: Natural and Artefactual Endurants

Our delineation of the concept of 'domain', such as we study it, critically embodied the terms *'human assisted'* and *'artefactual'*. The distinction between these two is not explicitly reflected in the set of classification predicates.

A.2.1.2 An Observer Calculus

We suggest some observers.

observe_alternative_sorts applies to parts p of sort P (for which is_alternative_sort(p) is claimed to hold) and yields a part, p' of sort P' different from P. Observing, in a repeated fashion, distinct alternative-sort parts, p_a, p_b, ..., or p_c – exhaustively[1] – shall yield p_{a_i}, p_{b_j}, ..., or p_{c_k}, all of respective, corresponding sorts P_a, P_b, ..., P_c. These are the alternative sorts.

[1] – till no other parts p_x are observed of sorts different from sort P_a, P_b, ..., P_c

`observe_part_set` applies to parts p of sort P (for which `is_part_set(p)` is claimed to hold) and yields a set of sub-parts, $\{p_a, p_b, ..., p_c\}$ of the same sort P'. Observing, in a repeated fashion, distinct single-sort set parts, p, p', ..., p'', all [claimed to be of the] of the same sort, P, shall yield p_{a_i}, p_{bj}, ..., p_{ck}, p'_{a_i}, p'_{b_j}, ..., p'_{c_k}, ..., respectively p''_{a_i}, p''_{b_j}, ..., p''_{c_k}, all of same sort P'.

`observe_composite_part` applies to parts p of sort P (for which `is_composite_part(p)` holds) and yields a compound of sub-parts, $p_i, p_j, ..., p_k$ of respective, different sorts: $P_i, P_j, ..., P_k$. Observing, in a repeated fashion, distinct composite parts, $p, p', ..., p''$, all [claimed to be of the] of the same sort, P, shall yield $p_i, p_j, ..., p_k$, and $p'_i, p'_j, ..., p'_k, ...$, respectively $p''_i, p''_j, ..., p''_k$, all of respective, corresponding sorts $P_i, P_j, ..., P_k$.

We do not formalise the language in which the above observer function "definitions" , cf. ≡, are expressed.

A.2.1.2.1 Classifiers versus Observers

The classification predicates are "applied", by the domain analysers & describers, to phenomena of a "real" world, i.e., the domain, and yields, in the mind of the domain analyser & describer, truth values, *true* or *false*. The observer functions, Sect. **A.2.1.2**, are "applied", by the domain analysers & describers, to phenomena of a "real" world, i.e., the domain, and yields, in the mind of the domain analyser & describer, a segment, i.e., endurants, of that domain.

A.2.1.3 A Description Calculus

One thing is the domain with its endurants and perdurants. Another thing is its analysis, observation and description. We can analyse, observe and describe the domain. Our analyses, observations and descriptions, of course, does not change the domain. Domain analysis yields truth values in the mind of the analyser. Domain observation focuses attention on a subset of the domain. Domain descriptions yield, as we shall see, RSL$^+$TEXT. That is RSL texts augmented with an indefinite number of endurant (**obs**_P), this section), unique identifier (see Sect. **A.2.2.1**, **uid**_P), mereology (see Sect. **A.2.2.2** **mereo**_P), and attribute (see Sect. **A.2.2.3** **attr**_A) observer function symbols. The **obs**_P observer function applies to parts of sort P' and yield sub-parts of sort P. For **uid**_P, **mereo**_P and **attr**_A see respective Sects. **A.2.2.1**, **A.2.2.2** and **A.2.2.3**. We shall in the sequel omit the "bold-facing" of **obs_**, **uid_**, **mereo_** and **attr_**. We shall not formalise the RSL$^+$TEXT description language.

A.2.1.3.1 Endurant Describers

Each of the describer functions make use of the observer functions defined in Sect. **A.2.1.2**. The describer functions apply to domain endurants and yield RSL$^+$TEXT. The texts, `txt`, between ❝ and ❞, i.e., ❝ `txt` ❞, is the RSL$^+$TEXT being "generated" !

• Describe Alternative Sort Parts

```
describe_alternative_sorts_part(p) ≡
    let ((p1,❝ E1 ❞),...,(pn,❝ En ❞))
                = observe_alternative_sorts_part(p) in
❝ Narration:
    [s]  ... narrative on alternative sorts ...
    [o]  ... narrative on sort observers ...
    [p]  ... narrative 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
axiom
[p] [ disjointness of alternative sorts ] E_1, ..., E_n
end
pre: is_alternative_sorts_set(p) ❞
```

• **Describe Part Sets**

```
describe_part_sets(p) ≡
    let (_,❝ P ❞) = observe_part_sets(p) in
❝ Narration:
    [s] ... narrative on sort ...
    [o] ... narrative on sort observer ...
    [p] ... narrative on proof obligation ...
Formalisation:
    type
    [s] P
    [s] Ps = P-set
    value
    [o] obs_Ps: E → Ps
    end
    pre: is_single_sort_set(p) ❞
```

• **Describe Composite Parts**

```
describe_composite_part(p) ≡
    let (_²,(❝ E1,...,En ❞)) = observe_composite_part(p) in
❝ Narration:
    [s] ... narrative on sorts ...
    [o] ... narrative on sort observers ...
    [p] ... narrative 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
    pre: is_composite(p) ❞
```

• **Describe Parts** In summary:

```
value
    describe_parts(p) ≡
        is_alternative_sorts_part(p) →
            → describe_alternative_sorts_part(p),
        is_part_sets(p)
```

[1] 1 The use of the underscore, _ , shall inform the reader that there is no need, here, for naming a value.

```
            → describe_part_sets(p),
      is_composite_part(p)
            → describe_composite_part(p)
```

A.2.1.4 A State of Endurants

By means of the endurant observers, **obs_**P, one can, for a given domain and instant of time, speak of the domain endurant state, say σ:Σ. It consists of all the observed parts, whether atomic and proper, i.e., non-structure parts: part sets (and their parts, etc.), and composite parts (and its parts, etc.). Informally:

```
value
   gen_state: P-set → P → P-set
   gen_state(ps)(p) ≡
         is_atomic(p) → {p},
         is_part_set(p) → {p} ∪ { gen_state(p′) | p′ ∈ obs_Ps(p) },
         is_composite(p) → {p} ∪
               gen_state(obs_P1(p)) ∪ gen_state(obs_P2(p)) ∪ ... ∪ gen_state(obs_Pn(p)),
         is_structure(p) →
               is_part_set_structure(p) → { gen_state(p′) | p′ ∈ obs_Ps(p) },
               is_composite_structure(p) →
                     gen_state(obs_P1(p)) ∪ gen_state(obs_P2(p)) ∪ ... ∪ gen_state(obs_Pn(p))
```

Note that the structure part p is not added to 'structure' results in accordance with being a structure. Now the state of a domain universe of discourse, uod, is gen_state(uod). We return to the notion of state in Sect. **A.3.2**.

■ .... **Begin Example 1: External Qualities**

63. The universe of discourse is a road transport system.
64. The road transport system is a structure, a composite of three part sets: road links, road intersections and automobiles.
65. Links, hus and automobolies are considered atomic.

```
type
63. RTS
64. RLS, RHS, AS
64. RLs = L-set, RHs = H-set, As = A-set
65. L, H, A
value
64. obs_RLS: RTS→RLS, obs_RHS: RHS→RLS, obs_AS: RTS→AS
64. obs_RLs: RLS→RLs, obs_RHs: RHS→RHs, obs_As: AS→As
```

66. Let *ls* stand for all links of a domain.
67. Let *hs* stand for all hubs of a domain.
68. Let *as* stand for all automobiles of a domain.
69. The domain state consists of all road hubs, road links and automobiles.
70. If there is a link then there is a hub, otherwise the link, hub and automobile sets may be empty.

```
value
66. ls = obs_RLs(obs_RLS(rts))
67. hs = obs_RHs(obs_RHS(rts))
```

68. as = obs_As(obs_AS(rts))
69. domain_state: RTS → (H|L|A)-**set**
69. domain_state(rts) ≡ $ls \cup hs \cup as$
axiom
70. $ls \neq \{\} \Rightarrow hs \neq \{\}$

. . . . **End of Example 1** ■

A.2.2 Internal Qualities

Internal qualities of endurants of a manifest domain are, in a 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.

We refer to [55, *Sects. 2.2.3–4, 3.8, and 5.2–5.3*] for a fuller discussion of the concepts and unique identification and mereology.

A.2.2.1 Unique Identification

Parts are uniquely identified. This follows from philosophic-analytic reasoning outlined by Kai Sørlander [146, 147, 148, 149]. We refer to [55, *Chapter 2 and Sect. 5.2.1*] for summaries.

A.2.2.1.1 The Unique Identifier Observer

With each part sort P we associate a further undefined unique identifier sort Π and a similarly further undefined unique identifier observer function **uid**_P such that for all parts $p,p',...,p''$ of sort P **uid**_P(p), **uid**_P(p'), ..., **uid**_P(p'') yield distinct unique identifiers ($\pi,\pi',...,\pi''$ respectively).

A.2.2.1.2 The Unique Identifier Describer

unique_identifier_observer(p) ≡
" **Narration:**
[s] ... narrative on unique identifier sort UI ...
[u] ... narrative on unique identifier observer ...
[a] ... axiom on uniqueness of unique identifiers ...
Formalisation:
type
[s] UI
value
[u] **uid**_P: P → UI
axiom
[a] [disjointness of UI wrt. all sorts] "

■ . . . . **Begin Example 2: Internal Qualities: Unique Identifiers**

71. Each link, hub and automobile has a unique identifier.

72. Let *lis*,*his*,*ais* stand for all link, hub and automobile identfiers of a domain.
73. No two or more links, hubs or automobiles have identifical identifiers.
74. Given a link identifier of a domain one can *extract* the identified link.
75. Given a hub identifier of a domain one can *extract* the identified hub.

```
type
71. LI, HI, AI
value
71. uid_L: L→LI, uid_H: H→HI, uid_A: A→AI
71. lis = {uid_LI(l)|l:L•l ∈ ls}
71. his = {uid_HI(h)|h:H•h ∈ hs}
71. ais = {uid_AI(a)|a:A•a ∈ as}
axiom
73. card lis + card his + card ais = card ls + card hs + card as
value
74. xtr_L: LI → L
74. xtr_L(li) ≡ let l:L • l ∈ ls ∧ uid_L(l)=li in l end
75. xtr_H: HI → H
75. xtr_H(hi) ≡ let h:H • h ∈ hs ∧ uid_H(h)=hi in h end
```

. . . . **End of Example 2** ■

A.2.2.2 Mereology

"Mereology is a theory of part-hood **relations:** of the relations of part to whole and the relations of part to part within a whole"[2].

We shall deploy mereology practically. That is, we are not studying mereology. We are using the ideas of mereology for experimental research and engineering purposes.

For natural endurants a typical relation is that of *"next-to"*. For artefactual endurants typical relations make explicit how the designers of these artefacts intended their logical, not necessarily geographical relationship, to be: *"next-to"*, *"to-be-part-of"*, *"as-an-element-of-a-set"*, etcetera.

A.2.2.2.1 The Mereology Observer

The mereology relations are here expressed in terms of the unique part identifiers. Let $p : P$ (of sort P) be a part with unique identifiers π. Let $\{p_1 : P_1, p_2 : P_2, ..., p_m : P_m\}$ be the set of parts (or respective sorts) to which p is [mereologically] related. We can express this my stating that **obs_mereo**_P(p) = $\{\pi_1 : \Pi_1, \pi_2 : \Pi_2, ..., p\pi_m : \Pi_m\}$ or **value obs_mereo**_P: P →UI-**set** – i.e., as a set of unique identifiers. **mereo**_P is the mereology observer. One could also express the mereology of a part, p, as a triplet ($iuis$, $iouis$, $ouis$) of sets of unique identifiers: those, $iuis$, of parts from whose transcendentally deduced (see Sect. **A.4.4**) corresponding behaviours p only receives "input", those, $iouis$, of parts from and to whose behaviours p receives "input" and "output", and those, $ouis$, of parts to whose behaviours p only delivers "output", In general: **value obs_mereo**_P: P →MT where MT is a type expression over unique identifier types.

A.2.2.2.2 The Mereology Describer

```
mereology_observer(p) ≡
```

[2] Achille Varzi: Mereology, http://plato.stanford.edu/entries/mereology/ 2009 and [75].

❝ **Narration:**

[t] ... narrative on mereology type ...
[m] ... narrative on mereology observer ...
[a] ... narrative on mereology type constraints ...

Formalisation:

type
[t] MT = $\mathcal{M}$(UI$_i$,UI$_j$,...,UI$_k$)
value
[m] **obs_mereo**_P: P → MT
axiom [Well–formedness of Domain Mereologies]
[a] $\mathcal{A}$: $\mathcal{A}$(MT): Well–formedness **of** Mereologies ❞

■ .... **Begin Example 3: Internal Qualities: Mereology**

76. The mereology of links is a set of two distinct hubs identifiers [of the hubs of the domain to which the link is connected] and a set of automobile identifiers [of the automobiles allowed to enter the link].
77. The mereology of hubs is a set of one or more link identifiers [of the links of the domain to which the hub is connected] and a set of automobile identifiers [of the automobiles allowed to enter the hub].
78. The mereology of an automobile is a set of link and hub identifiers [of the links and hubs of the domain that the automobile may enter].

```
type
76. LM = HI-set×AI-set
77. HM = LI-set×AI-set
78. AM = (LI|HI)-set
value
76.–78. mereo_L: L→LM, mereo_H: H→HM, mereo_A: A→AM
axiom
76. ∀ l:L•l ∈ ls ⇒ let (his,ais) = mereo_L(l) in
76.         his ⊆ his ∧ ais ⊆ ais end
77. ∀ h:H•h ∈ hs ⇒ let (lis,ais) = mereo_H(h) in
77.         lis ⊆ lis ∧ ais ⊆ ais end
78. ∀ a:A•a ∈ as ⇒ let ris = mereo_A(a) in
78.         ris ⊆ lis ∪ his end
```

.... **End of Example 3** ■

A.2.2.3 Attributes

Whereas unique identification and mereology are both of abstract, existential, logic nature, attributes are of concrete nature: physical, biological or historical nature. Attributes have values and attribute values are of types. *Two or more endurants that all have sets of attribute values of the same type, as well as the same unique identifier type and mereology types, are of the same sort.* This is an endurant sort-determining mantra.

A.2.2.3.1 The Attribute Observers

From any part, *p*:*P*, we can thus identify a set of attribute type names, $A_{p_1}, A_{p_2}, ...A_{p_p}$, informally:

- **attributes_P**: $P \to$ ❛❛ $A_{p_1},A_{p_2},...,A_{p_p}$ ❜❜.

Given a $p{:}P$, **attr**_A obtains the value of attribute A. The **attr**_A_{p_i} s are attribute observers of $p_i{:}P_i$.

A.2.2.3.2 The Attribute Categories

Michael A. Jackson [107] has suggested a hierarchy of attribute categories. *Static attributes:* values do not change. *Dynamic attributes:* values can change. Within the dynamic attribute category there are either: *inert attributes:* values are not determined by the endurant, but by "an outside" (e.g., other endurants); or *reactive attributes:* values which, if they change, change in response to external stimuli; or *active attributes:* values which change of the "own volition" of the part; within the active attribute category there are: *autonomous attributes:* values which change only on the "own volition" of the part; *biddable attributes:* values, values that may be prescribed[3] but may fail to attain the prescribed value; and *programmable attributes:* values which are prescribed etc. For our purposes we "reduce" these six categories to four, CAT = STA|INR|MON|PRO:

- **static** [STA],
- **inert** [INR],
- **monitorable** [MON], and
- **programmable** [PRO].

A.2.2.3.3 Attribute Pragmatics

There are many kinds of attributes. There are the obvious attributes of **physical nature:** spatial attributes: length (e.g., m), area (e.g., m^2), volume (e.g., m^3); temporal attributes: time stamps, and time intervals (e.g., s); spatio-temporal attributes: velocity (e.g., m/s), acceleration (e.g., m/s^2), etc.; and other physics attributes: mass (e.g., kg), electric current (e.g., A), thermodynamic temperature (e.g., K), amount of substance (*mole*), and luminous intensity (*lumen*). Based on these a multitude of further, additional kinds of physical attributes can be expressed; we refer to [55, Table 5.2, Sect. 5.4.5.1.]

There are the attributes of **chemical nature**, related to substance, i.e., matter (*mole*), as typified by the *periodic table of elements* (hydrogen, lithium, beryllium, etc.).

And then there are the attributes of **historical nature**. By this we mean, attributes which record *events* related to parts. Often such (historical, i.e., time–stamped) events relate two parts: *"part p 'interacted' with part q at such-and-such a time"*.

Finally there are a kind of **concept of appearance** attributes *'colour'*, *'newness/usedness/aged'*, *'roughness'/'smoothness'*, etcetera, of a part. We shall often find that these attributes are of a *fuzzy* nature [78, 112, 156, 108].

A.2.2.3.4 The Attributes Describer

```
describe_attributes(p) ≡
  let {❛❛ A1, ..., Am ❜❜} = attributes_P(p) in
  ❛❛ Narration:
        [t]  ... narrative on attribute sorts ...
        [o]  ... narrative on attribute sort observers ...
        [p]  ... narrative on attribute sort proof obligations ...
  Formalisation:
        type
        [t]  A1, ..., Am,
```

[3] – by the transcendent part behaviour

```
    value
    [o]  attr_A1: P→A1 [CAT],
    [o]  attr_A1: P→A2 [CAT],
    [o]  ...,
    [o]  attr_A1: P→Am [CAT].
    proof obligation [Disjointness of Attribute Types]
    [p]  PO: let P be any part sort in
    [p]        let a:(A1|A2|...|Am) in
    [p]        is_Ai(a) ≠ is_Aj(a) [i≠i, i,j:[1..m]] end end ❞
end
```

■ .... **Begin Example 4: Internal Qualities: Attributes**

79. Link attributes:
    a. Link have lengths.
    b. A link history records the time-ordered discrete times that any automobile has appeared on the link.

80. Hub attributes:
    a. A hub history records the time-ordered discrete times that any automobile has appeared at the hub.
    b. A hub state, $h\sigma$:$H\Sigma$, models the red/yellow/green signal setting. It expresses which pairs of links have green light from *in* to *out*.
    c. A hub state space, $h\omega$:$H\Omega$, models the set of all road intersection, i.e., hub signal settings. A hub with no signal has a single hub state hub state space with that state having green light in all directions !
    d. Any current hub state must be in the hub state space.

81. Automobile attributes:
    a. An automobile history records the discrete times that the automobile has been on a link and at a hub.
    b. An automobile has a road position: either at a hub, or along a link, a fraction of the distanve from one hub to the next.

```
type
79a. LEN
79b. LHist = AI →m TIME*
80a. HHist = AI →m TIME*
80b. HΣ = (LI×LI)-set
80c. HΩ = HΣ-set
81a. AHist = RI →m TIME* , RI = LI|HI
81b. APos == atH | onL
81b. atH :: HI
81b. onL :: LI × HI × F
81b. F = Real axiom ∀ f:F • 0≤f≤1
value
79a. attr_LEN: L → LEN
79b. attr_LHist: L → LHist
80a. attr_HHist: H → HHist
80b. attr_HΣ: H → HΣ
80c. attr_HΩ: H → HΩ
```

axiom
79b.–80a. ∀ lhist:LHist, hhist:HHist •
79b.–80a. ∀ **tl**:$\mathbb{TIME}^*$•**tl** ∈ **rng** lhist ∪ **rng** lhist⇒is_ordered(**tl**)
80d. ∀ h:H • h ∈ *hs* ⇒ attr_HΣ(h) ∈ attr_HΩ(h)
value
79b.–80a. is_ordered: $\mathbb{TIME}^*$ → **Bool**
79b.–80a. is_ordered(**tl**) ≡ ∀ i,j:**Nat**•{i,j}∈ **inds tl**∧i<j⇒**tl**[i]<**tl**[j]
axiom
81b. ∀ ath:atH • s_HI:ath ∈ *his*
81b. ∀ onl:onL • s_LI(onl)∈*lis*∧s_HI(onl)∈*his*
81b. ∧s_HI(onl)∈mereo_L(xtr_L(s_LI(onl)))

. . . . **End of Example 4** ∎

A.2.2.4 Intentional "Pull"

The concept of *intentional "pull"* is [also] a new concept[4]. For artefacts one can claim that certain parts p:P are created in order to serve other parts q:Q, and vice versa. *"roads serve to convey transport, automobiles serve to transport goods"*. Historical events record interactions between such parts p and q. So a historical attribute of p records its interaction with q, and a historical attribute of q records its interaction with p, and *"one cannot have one without the other"*, and this is what we mean by **intentional "pull"**! So introducing historical attributes for a sort P usually entails also introducing historical attributes for another sort Q, etcetera. And this consequentially implies that the domain analyser cum describer must express a necessary **intentional "pull" axiom** that expresses that *"one cannot have one without the other"*.

∎ . . . . **Begin Example 5: Internal Qualities: Intentional Pull**

We leave it to the reader to formally narrate and formalise the following example of intentional pull. The example continues that of the attributes of links, hubs and automobiles above.

If an automobile is recorded, in the history of a link, to have been on that link, at time τ, then that link records the automobile to have been there at that time. And vice-versa: if a link history records an automobile, at some time, then that automobile's history records that link, at that time.

. . . . **End of Example 5** ∎

A.2.3 Narratives

How much, how little ?̇

Narratives on external qualities should be very short. Perhaps it is admissible to just mention sort/part names and their composition, relying on the readers' goodwill.

The real narrative description of endurants comes with their internal qualities: mereology and attributes.

A.3 Space, State and Time

Three notions: *space, state* and *time*, are given by analytic-philosophic reasoning [146, 147, 148, 149]. That is, they are neither endurants nor perdurants; nor, more "down-to-earth", attributes.

[4] It "parallels" the *gravitational pull* of physics; hence the 'naming' !

A.3.1 Space

*"The two relations **asymmetric** and **symmetric**, by a transcendental deduction, can be given an interpretation: the relation (spatial) **direction** is asymmetric; and the relation (spatial) **distance** is symmetric. Direction and distance can be understood as spatial relations. From these relations are derived the relation in-between. **Hence** we must conclude that **[...] entities exist in space**. Space is therefore an unavoidable characteristic of any possible world"* [149, pp 154] ■

A.3.1.1 Spatial Types

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

type
82 $\mathbb{SPACE}$
83 $\mathbb{POINT}$

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

A.3.1.2 Spatial Attributes

84. A point observer, observe_$\mathbb{POINT}$, is a function which applies to endurants, e, and yield an indefinite point, $\ell : \mathbb{POINT}$, of, or within, that endurant !

value
84 observe_$\mathbb{POINT}$: E $\rightarrow$ $\mathbb{POINT}$

A.3.2 State

We first touched upon the notion of 'state' in Sect. **A.2.1.4**. In [that] section a state was delineated as any set of domain endurants. We can now offer a more refined delineation of state: a state is [now] any set of endurants each of which has at least one dynamic attribute.

Internal qualities of endurants allow us to express predicates over endurant values. It is thus the internal qualities of endurants that gives "flesh" to the notion of manifest endurants.

Analytic-philosophically we can argue: *"Entities may be ascribed predicates which it is not logically necessary that they are ascribed. How can that be possible ? Only if we accept that entities may be ascribed predicates which are in-commensurable with predicates that they are actually ascribed. That is possible, we must conclude, if entities can **exist** in distinct **states**"* [149, 158–159] ■

A.3.3 Time

"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 ***[...] entities exist in time****. So every possible world must exist in time"* [149, pp 159] ■

A.3.3.1 Time and Time Interval Sorts and Operators

85. There is an abstract type $\mathbb{T}$*ime*,
86. and an abstract type $\mathbb{TI}$: $\mathbb{T}$*ime*$\mathbb{I}$*nterval*.
87. There is no $\mathbb{T}$*ime* origin, but there is a "zero" $\mathbb{TI}$me interval.
88. One can add (subtract) a time interval to (from) a time and obtain a time; and 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.
89. One can subtract a time from another time obtaining a time interval respecting that the subtrahend is smaller than or equal to the minuend.
90. One can multiply a time interval with a real and obtain a time interval.
91. One can compare two times and two time intervals.

```
type
85  𝕋
86  𝕋𝕀
value
87  0:𝕋𝕀
88  +,-: (𝕋 × 𝕋𝕀 → 𝕋|𝕋𝕀 × 𝕋𝕀 ~→ 𝕋𝕀)
89  -: 𝕋 × 𝕋 → 𝕋𝕀
90  *: 𝕋𝕀 × Real → 𝕋𝕀
91  <,≤,=,≠,≥,>: (𝕋 × 𝕋|𝕋𝕀 × 𝕋𝕀) → Bool
axiom
88 ∀ t:𝕋 • t+0 = t
```

A.3.3.2 The Time Observer

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

```
type
92  𝕋
value
92  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.

A.4 Perdurants

We introduce this section with remarks on *actions, events and behaviours*, on *determinacy and non-determinacy*, on *co-operating domains*, and on *transcendental deduction*. The real "meat" of this section are sections **A.4.5**, **A.4.6**, and **A.4.8**.

A.4.1 Actions, Events and Behaviours

By a **action** we shall understand something that occurs in time, lasting, however, no time, or, at least, we ignore time – considering actions as indivisible, taking place as the result of a "willed" [other] action, and usually changing the state.

By an **event** we shall understand something that occurs in time, lasting, however, no time, taking place as spontaneously, not as the result of a "willed" action, but possibly as the result of another event, and usually changing the state.

By a **behaviour** we shall understand a set of sequences of actions, events and [other, sub-] behaviours, some of which relate to, i.e., interact with one another, cf. Sect. **A.4.3**.[5]

• • •

The purpose of this overall section on perdurants is to indicate how part descriptions lead to behaviour descriptions such that domains with two or more parts lead to domain behaviours consisting of two or more concurrent behaviours.

A.4.2 Determinacy and Non-determinacy

The remarks of this section are informal. They relate to the meaning, the semantics, of descriptions of behaviours. We assume acceptance of the **state** concept as outline in Sects. **A.2.1.4** and **A.3.2**.

Simple Behaviour Descriptions: A behaviour description is a simple one if it consists only of a sequence of action and event descriptions.

Serial Executions: Now let us consider the meaning, operationally, in terms of executions of simple behaviour description. We shall call such executions serial executions. A serial execution consists of the interpretation of action and event descriptions, one-by-one, one-after-the-other, step-by-step. Execution of actions and the occurrence of events result, usually, in state changes.

Determinacy: If the state change, as the result of an action execution, is predictable from the action description and the state just before action execution, then we say that the execution is determinate, i.e., represents determinacy.

Non-determinacy: If the state change, as the result of an action execution, is not predictable, i.e. represent *non-determinacy*, then we say that the execution is non-determinate. Occurence of events represents non-determinacy.

Single Behaviour Internal Non-determinism: Single behaviour descriptions may offer a next action or event in the form of a *choice* between two or more alternatives. When that choice is left to the behaviour itself we refer to it as an internal choice, leading to internal non-determinism.

Concurrent Behaviour Descriptions: A *general behaviour description* consists of simple and general behaviour descriptions. A general behaviour description describes *concurrency*. These may or may not be describing interaction between the intended behaviours. Henceforth we assume that the behaviour descriptions do describe interaction.

Concurrent Executions: Thus a general behaviour description describes *concurrent executions*.

Internal Non-determinacy: We now assume a general behaviour description and its denoted executions. When a behaviour description offers the choice between two or more alternative next actions or events where that choice is left to the behaviour itself we refer to it as an internal choice, leading to internal non-determinism.

Multiple Behaviour and Internal and External Non-determinacy: We now assume a general behaviour description and its denoted executions.

[5] This 'relation' is, in CSP [99, 100, 141, 144, 101], expressed in terms of CSP input/outputs: ch[index] ?, respectively ch[index] ! value where values are communicated over indexed channels.

External Non-determinacy: When a behaviour description offers the choice between two or more alternative next actions or events where that choice is determined by other behaviours the we refer to it as an external choice, leading to external non-determinism.

Determinacy: When (segments of) a behaviour description does not imply non-determinacy, then it implies determinacy, i.e., next-state predictability.

A.4.3 Co-operating Domains

A domain usually consists of two or more parts. And hence, as we shall see, domains consists of a corresponding number of behaviours. Usually these behaviours co-operate, i.e., interact. Interaction may take many forms. A common form is that of two behaviours *synchronising* and *communicating*. They synchronise at a point in their behaviours, i.e., both being at specific points, when they communicate. They communicate, for example, by one behaviour offering a value of some kind to the other of the two behaviours. There could be other forms of behaviour interaction. The one we have chosen corresponds to Tony Hoare's concept of **communicating sequential processes** [101] One could have chosen other descriptional models: `Petri nets` [137], or `Message Sequence Charts, MSC` [105], or `Statecharts` [95], or `Live Sequence Charts` [96], or other.

A.4.4 Transcendental Deduction

A Philosophical Principle: *"A **transcendental argument** is an argument which elucidates the conditions for the possibility of some fundamental phenomenon whose existence is unchallenged or uncontroversial in the philosophical context in which the argument is propounded"* [3, Anthony Brueckner, page 808].

"Such an argument proceeds deductively, from a premise of asserting the existence of some basic phenomenon (such as a meaningful discourse, conceptualisation of objective states of affairs, or the practice of making promises), to a conclusion asserting the existence of some interesting, substantive enabling conditions for that phenomenon" [3, Anthony Brueckner, page 808].

From Parts to Behaviours: The Possibility: So from the existence of endurants we shall assert the existence of behaviours. That is, from the existence of parts, we shall elucidate the conditions for the possibility of behaviours; with the existence of behaviours being unchallenged or uncontroversial in the philosophical context in which the argument is propounded.

■ .... **Begin Example 6: From Parts to Behaviours**

A railway train, in the vernacular, i.e., common parlance, may refer to either a train, as a[n **endurant**] part, waiting on a platform at a railway station, or a train, as a [**perdurant**] behaviour, "speeding" down the tracks, or a train [departures and arrivals], as an element of a time table [**attribute**].

.... **End of Example 6** ■

From Parts to Behaviours: The Deduction: So, for each, usually atomic, part we analyse and describe (i) the actions, (ii) the events, and (iii) the behaviour of that part, with that behaviour being a sequence of these actions (i), events (ii), and [suitable] sub-behaviours. The behaviour is then composed from these actions, events and sub-behaviours. The composition follows the part analysis wrt. actions, event and behaviours.

•••

As mentioned above, we do not consider the transcendental "morphing" of living species into behaviours, but could !

A.4.5 Channels

The mereology-analysis of parts and their relations transcendentally leads to the CSP notion of channels [99, 100, 141, 144, 101]. The interaction between behaviours is afforded by means of channels. We model "our" channels in the form of CSP channels. For any given domain there is thus a channel array:

- **channel** { ch[{ui_j,ui_k}] | ... }: MSG

where (...) ui_j,ui_k is any distinct pair of unique domain part identifiers, and where MSG is the type of messages communicated over these channels.

A.4.6 Behaviours

We first deal with behaviour signatures. They are strongly related to the internal qualities of parts. Then we deal with the definitions of behaviour 'bodies'. Here we have little, as concerns a systematic treatment, really, to say ! Finally we deal with the compilation of all distinct part sort domain behaviours.

A.4.6.1 Behaviour Signatures

Part behaviours are modeled as never-ending CSP processes [99, 100, 141, 144, 101]. The definition of the signature of these processes entail:

- naming the behaviour: the part sort name, P,

and the parameters:

- a unique part identifier type, UI,
- a mereology type expression, Mereo,
- a set of static attribute, i.e., constant valued variable, types, Stat_Attrs,
- a set of programmable attribute types, Prgr_Attrs,
- a set of channel expressions, {ch[...]|...} – to model inert and monitorable attributes,[6] and
- the **Unit** literal.

value
P: ui:UI × Mereo × Stat_Attrs
→ Prgr_Attrs → { ch[...] | ... } **Unit**

The **Unit** literal expresses that the behaviour is cyclic (expressed by tail-recursion of the behaviour 'body' description) and that the behaviour [potentially] changes the state — with that state now being represented by the full ensemble of all domain part behaviours.

[6] We refer to [55, Sect.?.7.3] for details !

A.4.6.2 Behaviour Definition 'Bodies'

Parts, whether they are atomic, composite or sets, are simply translated[7]:

```
TRANSLATEPart(e) ≡
  ❝ value
      M_P(ui,me,sv)(pv) ≡
          let (me′,pv′) = F_P(ui,me,sv)(pv) in
          M_P(ui,me′,sv)(pv′) end

      F_P: UI × Mereo × Stat_Attrs
              → Prgr_Attrs → { ch[ ... ] | ... }
                           → (Mereo×Prgr_Attrs)
      F_P(ui,me,sv)(pv) ≡ ... ❞
```

The interesting aspect here is the function $\mathcal{F}_P$. As already mentioned above, we shall not present a rigorous analysis of how part behaviours interact. But we shall somewhat informally reason about interaction possibilities. From what we shall present next we hope one day to present as rigorous a perdurant analysis & descriptions as we claim to have presented an endurant analysis & description.

A.4.6.2.1 Semiformal Examples of Behaviour Interactions

Let P and Q each stand for a set of behaviours. Both defined by a single definitions, hinted at in formula lines 93– 101 respectively 102–103 below. That is, these P and Q "body" definitions hint at $\mathcal{F}_P$, respectively $\mathcal{F}_Q$ definitions.

93. A behaviour P is defined.
94. *n* clauses are to be deterministically executed: first clause_1, then clause_1, etc., finally clause_n.
95. An internal non-deterministic choice is to be made between the elaboration of either clause_1 or clause_2 or ... or clause_n.
96. Behaviour P(pui)(...,...)(...) deterministically offers behaviour Q(quj)(...,...)(...) a message –
97. which may be accepted by that behaviour.
98. Behaviour P(pui)(...,...)(...) externally non-deterministically offers either of behaviours Q(quj)(...,...)(...) a message –
99. which may be accepted by one of these behaviours.
100. Behaviour P(pui)(...,...)(...) internally non-deterministically offers either of behaviours Q(quj)(...,...)(...) a message –
99 which may be accepted by one of these behaviours.
101. Behaviour P(pui)(...,...)(...) is resumed.

102. Behaviour Q(quj)(...,...)(...) is defined.
97 Behaviour Q(quj)(...,...)(...) determinisitically offers to accept communication with a specific behaviour P(pui)(...,...)(...).
99 Behaviour Q(quj)(...,...)(...) external non-deterministically offers to accept communication with either of behaviours P(pui)(...,...)(...) where pui ranges of a set.
103. Behaviour Q(quj)(...,...)(...) internal non-deterministically chooses to accept a message from any of the behaviours P(pui)(...,...)(...), if offered.
104. Behaviour Q(quj)(...,...)(...) is resumed.

[7] Note that $\mathcal{F}_P$ "returns" also a possibly updated part mereology: hubs and links may be inserted into or removed from a road net.

```
value
93.   P(pui)(...,...)(...) ≡
94.      ... clause_1 ; clause_2 ; ... ; clause_n ...
95.      ... clause_1 ⌈⌉ clause_2 ⌈⌉ ... ⌈⌉ clause_n ...
96.      ... ch[ {pui,quj} ] ! mkMsg(...) ...
98.      ... ⌈⌉⌊⌋ { ch[ {pui,quj} ] ! mkMsg(...) | quj:QUJ • ... } ...
100.     ... ⌈⌉ { ch[ {pui,quj} ] ! mkMsg(...) | quj:QUJ • ... } ...
101.     ... P(pui)(...,...)(...)

102.  Q(quj)(...,...)(...) ≡
97.      ... let mkMsg(...) = ch[ {pui,quj} ] ? in ... end ...
99.      ... let mkMsg(...) = ⌈⌉⌊⌋ { ch[ {pui,quj} ] ? | pui:PUI • ... } in ... end ...
103.     ... let mkMsg(...) = ⌈⌉ { ch[ {pui,quj} ] ? | pui:PUI • ... } in ... end ...
104.     ... Q(quj)(...,...)(...)
```

■ .... **Begin Example 7: Automobile and Link Behaviours**

105. We simplify automobile behaviour on a Link.
   a. Provisionally, "updating" the automobile history.
   b. Internally non-deterministically, either
      i. the automobile remains, "idling", i.e., not moving, on the link and with update history,
      ii. however, first informing the link of its position,
   c. 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 with updated history, or
      ii. **else**,
         1. while obtaining a "next link" from the mereology of the hub (where that next link could very well be the same as the link the vehicle is about to leave),
         2. the vehicle informs the hub that it is now at that hub, identified by th_ui,
         3. and updates its history,
         4. whereupon the vehicle resumes the vehicle behaviour positioned at that hub with updated history;
   d. or
   e. the vehicle "disappears — off the radar" !

```
value
105 auto: ai:AI × ris:AM → onL(fhi,li,f,thi):APos × AHist
105            → {ch[ {ai,ri} ]|ri:RI•ri ∈ ris} Unit
105 auto(ai,ris)(onL(fhi,li,f,thi),ahi) ≡
105a       let ahi′ = ahi†[ li ↦ ⟨record_TIME()⟩⁀ahi(li) ] in
105(b)ii   (ch[ {li,ai} ] ! onL(fhi,li,f,thi) ;
105(b)i    auto(ai,ris)(vp,ahi′))
105b       ⌈⌉
105(c)i    if f<1
105(c)i      then
105(c)i1       let onl = onL(fhi,li,incr(f),thi) in
105(c)i2       ch[ {li,ai} ] ! onl ;
105(c)i3       auto(ai,ris)(onl,ahi′) end
```

```
105(c)ii      else
105(c)ii1        let nli:L_UI • nli ∈ mereo_H(get_H(thi)) in
105(c)ii2        ch[ {thi,ai} ] ! atH(thi) ;
105(c)ii3        let ahi′′ = ahi†[ thi ↦ ⟨record_TIME()⟩⁀ahi(thi) ] in
105(c)ii4        auto(ai,ris)(atH(li,thi,nxt_lui),ahi′′)
105(c)ii3        end end
105(c)i       end
105d        ⊓
105e       stop end
105(c)i1    incr: Fract → Fract ...
```

106. We abstract link behaviours.

   a. Externally non-deterministically a link behaviour is here simplified to
   b. offer to accept communication messages, m, cf. Items 105(b)ii and 105(c)i2 above,
   c. from any automobile in its mereology,
   d. the link behaviour history is updated, and
   e. the link behaviour resumes with that history.

```
value
106   link: LI × (his,ais):LM × LEN
106           → LHist → { ch[ {li,ai} ] | ai ∈ ais } Unit
106   link(li,(his,ais),le)(lhi) ≡
106     ⨅
106a        { let m = ch[ li,ai ] ? in
106d          let lhi′ = lhi † [ li ↦ ⟨record_TIME()⟩⁀lhi(li) ] in
106e          link(li,m,le)(lhi′)
106c        | ai:AI • ai ∈ ais end end }
```

. . . . **End of Example 7** ■

A.4.7 Compilation of Domain Descriptions

Every part, p, of sort P (P = type_name(p)), of a domain is compiled, **TRANSLATE**P(p), once. The compilation applies to a universe of discourse and yields an RSL$^+$TEXT. Informally that compilation proceeds as follows: ps stand for all domain parts, whether atomic or composite. p_s_ns stand for the type names of all these parts – if there are several parts of the same sort, then its sort name is represented once in p_s_ns. u_s_ps is now a subset of ps, with exactly one representative for each part sort. The ⁀ clause: ⟨ ... | p ∈ u_s_ps ⟩ concatenates the RSL$^+$TEXT sequences of the individual part descriptions.

```
value
   compile_system: UoD → RSL+TEXT
   compile_system(uod) ≡
      let ps = calc_parts({uod})({}) in
      let p_s_ns = {typ_nam(p)|p ∈ ps}  in
      let u_s_ps = {p|p ∈ ps, typ_nam(p) ∈ p_s_ns} in
      ⁀ ⟨describe_parts(p),
         unique_identifier_observer(d),
         mereology_observer(p),
         describe_attributes(p),
```

```
      TRANSLATE P(p)|p ∈ u_s_ps⟩
    end end end
```

A.4.8 Initialising Domains

For each part sort there is exactly one description. But for each part sort there may be many instances of that part in the domain.

107. initialising a domain system starts with a domain, uod, and yields a behaviour, **Unit**.
108. It is based on all parts, ps.
109. The domain behaviour is the parallel, ‖, composition
110. of the behaviours of
111. all parts – whose
112. unique identifier, mereology and
113. static and programmable attribute values,
114. as well as the mereology-determined channels – have been "extracted" from the endurant part,
115. where also the behaviour signature have been stated.

```
value
107. initialise_domain_behaviour: UoD → Unit
107. initialise_domain_behaviour(uod) ≡
108.   let ps = calc_parts({uod})({}) in
109.   ‖ {
112.     let ui = uid_P(p), me = mereo_P(p),
112.         sv = stat_attrs(p), pv = progr_attrs(p),
112.         ioc = in_o_channels(me) in
115.      𝓜P → ui:UI × me:MT × sv:Stat_Attrs
115.          → pv:Prgr_Attrs → ioc  Unit
110.      𝓜P(ui,me,sv)(pv)
111.    | p:P • p ∈ ps end }
108.   end
```

Here index P and suffix _P has the P be type_name(p) where type_name is defined from the part observer functions.

Appendix B
An RSL Primer

Contents

B.1 Types **547**
- **B.1.1** Type Expressions 547
  - **B.1.1.1** Atomic Types 547
    - **B.1.1.1.1** Basic Types: 547
  - **B.1.1.2** Composite Types 547
    - **B.1.1.2.1** Composite Type Expressions: 548
- **B.1.2** Type Definitions 548
  - **B.1.2.1** Concrete Types 548
    - **B.1.2.1.1** Type Definition: 549
    - **B.1.2.1.2** Variety of Type Definitions: 549
    - **B.1.2.1.3** Record Types: 549
  - **B.1.2.2** Subtypes 549
    - **B.1.2.2.1** Subtypes: 549
  - **B.1.2.3** Sorts — Abstract Types 550
    - **B.1.2.3.1** Sorts: 550

B.2 The RSL Predicate Calculus **550**
- **B.2.1** Propositional Expressions 550
    - **B.2.1.0.1** Propositional Expressions: 550
- **B.2.2** Simple Predicate Expressions 550
    - **B.2.2.0.1** Simple Predicate Expressions: 550

B.3 Concrete RSL Types: Values and Operations **551**
- **B.3.1** Arithmetic 551
    - **B.3.1.0.1** Arithmetic: 551
- **B.3.2** Set Expressions 551
  - **B.3.2.1** Set Enumerations 551
    - **B.3.2.1.1** Set Enumerations: 551
  - **B.3.2.2** Set Comprehension 551
    - **B.3.2.2.1** Set Comprehension: 551
- **B.3.3** Cartesian Expressions 552
  - **B.3.3.1** Cartesian Enumerations 552
    - **B.3.3.1.1** Cartesian Enumerations: 552
- **B.3.4** List Expressions 552
  - **B.3.4.1** List Enumerations 552
    - **B.3.4.1.1** List Enumerations: 552
  - **B.3.4.2** List Comprehension 552
    - **B.3.4.2.1** List Comprehension: 552
- **B.3.5** Map Expressions 553
  - **B.3.5.1** Map Enumerations 553
    - **B.3.5.1.1** Map Enumerations: 553
  - **B.3.5.2** Map Comprehension 553
    - **B.3.5.2.1** Map Comprehension: 553
- **B.3.6** Set Operations 553
  - **B.3.6.1** Set Operator Signatures 553
    - **B.3.6.1.1** Set Operations: 553

B.3.6.2 Set Examples 554
B.3.6.2.1 Set Examples: 554
B.3.6.3 Informal Explication 554
B.3.6.4 Set Operator Definitions 555
B.3.6.4.1 Set Operation Definitions: 555
B.3.7 Cartesian Operations 555
B.3.7.0.1 Cartesian Operations: 555
B.3.8 List Operations 555
B.3.8.1 List Operator Signatures 555
B.3.8.1.1 List Operations: 555
B.3.8.2 List Operation Examples 556
B.3.8.2.1 List Examples: 556
B.3.8.3 Informal Explication 556
B.3.8.4 List Operator Definitions 556
B.3.8.4.1 List Operator Definitions: 556
B.3.9 Map Operations 557
B.3.9.1 Map Operator Signatures and Map Operation Examples 557
B.3.9.2 Map Operation Explication 558
B.3.9.3 Map Operation Redefinitions 558
B.3.9.3.1 Map Operation Redefinitions: 558
B.4 λ-Calculus + Functions 559
B.4.1 The λ-Calculus Syntax 559
B.4.1.0.1 λ-Calculus Syntax: 559
B.4.2 Free and Bound Variables 559
B.4.2.0.1 Free and Bound Variables: 559
B.4.3 Substitution 559
B.4.3.0.1 Substitution: 559
B.4.4 α-Renaming and β-Reduction 560
B.4.4.0.1 α and β Conversions: 560
B.4.5 Function Signatures 560
B.4.5.0.1 Sorts and Function Signatures: 560
B.4.6 Function Definitions 560
B.4.6.0.1 Explicit Function Definitions: 560
B.4.6.0.2 Implicit Function Definitions: 561
B.5 Other Applicative Expressions 561
B.5.1 Simple let Expressions 561
B.5.1.0.1 Let Expressions: 561
B.5.2 Recursive let Expressions 561
B.5.2.0.1 Recursive **let** Expressions: 561
B.5.3 Predicative let Expressions 562
B.5.3.0.1 Predicative **let** Expressions: 562
B.5.4 Pattern and “Wild Card” let Expressions 562
B.5.4.0.1 Patterns: 562
B.5.5 Conditionals 562
B.5.5.0.1 Conditionals: 563
B.5.6 Operator/Operand Expressions 563
B.5.6.0.1 Operator/Operand Expressions: 563
B.6 Imperative Constructs 563
B.6.1 Statements and State Changes 563
B.6.1.0.1 Statements and State Change: 564
B.6.2 Variables and Assignment 564
B.6.2.0.1 Variables and Assignment: 564
B.6.3 Statement Sequences and skip 564
B.6.3.0.1 Statement Sequences and **skip**: 564
B.6.4 Imperative Conditionals 564
B.6.4.0.1 Imperative Conditionals: 564
B.6.5 Iterative Conditionals 565
B.6.5.0.1 Iterative Conditionals: 565
B.6.6 Iterative Sequencing 565
B.6.6.0.1 Iterative Sequencing: 565
B.7 Process Constructs 565
B.7.1 Process Channels 565
B.7.1.0.1 Process Channels: 565

B.7.2 Process Composition 565
B.7.2.0.1 Process Composition: 565
B.7.3 Input/Output Events 566
B.7.3.0.1 Input/Output Events: 566
B.7.4 Process Definitions 566
B.7.4.0.1 Process Definitions: 566
B.8 Simple RSL Specifications 566
B.8.0.0.1 Simple RSL Specifications: 566
B.9 RSL Module Specifications 567
B.9.1 Modules 567
B.9.2 Schemes 567
B.9.3 Module Extension 568

This is an ultra-short introduction to the RAISE Specification Language, RSL.

B.1 Types

The reader is kindly asked to study first the decomposition of this section into its sub-parts and sub-sub-parts.

B.1.1 Type Expressions

Type expressions are expressions whose value are type, that is, possibly infinite sets of values (of "that" type).

B.1.1.1 Atomic Types

Atomic types have (atomic) values. That is, values which we consider to have no proper constituent (sub-)values, i.e., cannot, to us, be meaningfully "taken apart".

RSL has a number of built-in atomic types. There are the Booleans, integers, natural numbers, reals, characters, and texts.

B.1.1.1.1 Basic Types:

type
[1] **Bool**
[2] **Int**
[3] **Nat**
[4] **Real**
[5] **Char**
[6] **Text**

B.1.1.2 Composite Types

Composite types have composite values. That is, values which we consider to have proper constituent (sub-)values, i.e., can, to us, be meaningfully "taken apart".

From these one can form type expressions: finite sets, infinite sets, Cartesian products, lists, maps, etc.

Let A, B and C be any type names or type expressions, then:

B.1.1.2.1 Composite Type Expressions:

[7] A-**set**
[8] A-**infset**
[9] A × B × ... × C
[10] A^*
[11] A^ω
[12] A $\overrightarrow{m}$ B
[13] A → B
[14] A $\stackrel{\sim}{\to}$ 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:

1. The Boolean type of truth values **false** and **true**.
2. The integer type on integers ..., –2, –1, 0, 1, 2, ...
3. The natural number type of positive integer values 0, 1, 2, ...
4. The real number type of real values, i.e., values whose numerals can be written as an integer, followed by a period ("."), followed by a natural number (the fraction).
5. The character type of character values ″a″, ″bb″, ...
6. The text type of character string values ″aa″, ″aaa″, ..., ″abc″, ...
7. The set type of finite cardinality set values.
8. The set type of infinite and finite cardinality set values.
9. The Cartesian type of Cartesian values.
10. The list type of finite length list values.
11. The list type of infinite and finite length list values.
12. The map type of finite definition set map values.
13. The function type of total function values.
14. The function type of partial function values.
15. 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.

B.1.2 Type Definitions

B.1.2.1 Concrete Types

Types can be concrete in which case the structure of the type is specified by type expressions:

B.1.2.1.1 Type Definition:

```
type
    A = Type_expr
```

Some schematic type definitions are:

B.1.2.1.2 Variety of Type Definitions:

```
[1] Type_name = Type_expr /* without |s or subtypes */
[2] Type_name = Type_expr_1 | Type_expr_2 | ... | Type_expr_n
[3] Type_name ==
        mk_id_1(s_a1:Type_name_a1,...,s_ai:Type_name_ai) |
        ... |
        mk_id_n(s_z1:Type_name_z1,...,s_zk:Type_name_zk)
[4] Type_name :: sel_a:Type_name_a ... sel_z:Type_name_z
[5] Type_name = {| v:Type_name′ • 𝒫(v) |}
```

where a form of [2–3] is provided by combining the types:

B.1.2.1.3 Record Types:

```
    Type_name = A | B | ... | Z
    A == mk_id_1(s_a1:A_1,...,s_ai:A_i)
    B == mk_id_2(s_b1:B_1,...,s_bj:B_j)
    ...
    Z == mk_id_n(s_z1:Z_1,...,s_zk:Z_k)
```

Types A, B, ..., Z are disjoint, i.e., shares no values, provided all mk_id_k are distinct and due to the use of the disjoint record type constructor ==.

```
axiom
    ∀ a1:A_1, a2:A_2, ..., ai:Ai •
        s_a1(mk_id_1(a1,a2,...,ai))=a1 ∧ s_a2(mk_id_1(a1,a2,...,ai))=a2 ∧
        ... ∧ s_ai(mk_id_1(a1,a2,...,ai))=ai ∧
    ∀ a:A • let mk_id_1(a1′,a2′,...,ai′) = a in
        a1′ = s_a1(a) ∧ a2′ = s_a2(a) ∧ ... ∧ ai′ = s_ai(a) end
```

Note: Values of type A, where that type is defined by A::B×C×D, can be expressed A(b,c,d) for b:B, c:D, d:D.

B.1.2.2 Subtypes

In RSL, each type represents a set of values. Such a set can be delimited by means of predicates. The set of values b which have type B and which satisfy the predicate 𝒫, constitute the subtype A:

B.1.2.2.1 Subtypes:

```
type
    A = {| b:B • 𝒫(b) |}
```

B.1.2.3 Sorts — Abstract Types

Types can be (abstract) sorts in which case their structure is not specified:

B.1.2.3.1 Sorts:

```
type
  A, B, ..., C
```

B.2 The RSL Predicate Calculus

B.2.1 Propositional Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values (**true** or **false** [or **chaos**]). Then:

B.2.1.0.1 Propositional Expressions:

```
false, true
a, b, ..., c ~a, a∧b, a∨b, a⇒b, a=b, a≠b
```

are propositional expressions having Boolean values. $\sim, \wedge, \vee, \Rightarrow, =, \neq$ and $\square$ are Boolean connectives (i.e., operators). They can be read as: not, and, or, if then (or implies), equal and not equal.

B.2.2 Simple Predicate Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values, let x, y, ..., z (or term expressions) designate non-Boolean values and let i, j, ..., k designate number values, then:

B.2.2.0.1 Simple Predicate Expressions:

$\forall$ x:X • $\mathcal{P}(x)$
$\exists$ y:Y • $\mathcal{Q}(y)$
$\exists$! z:Z • $\mathcal{R}(z)$

are quantified expressions — also being predicate expressions.

They are "read" as: For all x (values in type X) the predicate $\mathcal{P}(x)$ holds; there exists (at least) one y (value in type Y) such that the predicate $\mathcal{Q}(y)$ holds; and there exists a unique z (value in type Z) such that the predicate $\mathcal{R}(z)$ holds.

B.3 Concrete RSL Types: Values and Operations

B.3.1 Arithmetic

B.3.1.0.1 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)
```

B.3.2 Set Expressions

B.3.2.1 Set Enumerations

Let the below a's denote values of type A, then the below designate simple set enumerations:

B.3.2.1.1 Set Enumerations:

$\{\{\}, \{a\}, \{e_1,e_2,...,e_n\}, ...\} \in$ A-**set**
$\{\{\}, \{a\}, \{e_1,e_2,...,e_n\}, ..., \{e_1,e_2,...\}\} \in$ A-**infset**

B.3.2.2 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.

B.3.2.2.1 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)}
```

B.3.3 Cartesian Expressions

B.3.3.1 Cartesian Enumerations

Let e range over values of Cartesian types involving $A, B, \ldots, C$, then the below expressions are simple Cartesian enumerations:

B.3.3.1.1 Cartesian Enumerations:

```
type
    A, B, ..., C
    A × B × ... × C
value
    (e1,e2,...,en)
```

B.3.4 List Expressions

B.3.4.1 List Enumerations

Let a range over values of type A, then the below expressions are simple list enumerations:

B.3.4.1.1 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.

B.3.4.2 List Comprehension

The last line below expresses list comprehension.

B.3.4.2.1 List Comprehension:

```
type
    A, B, P = A → Bool, Q = A ~→ B
value
    comprehend: Aω × P × Q ~→ Bω
    comprehend(l,P,Q) ≡
        ⟨ Q(l(i)) | i in ⟨1..len l⟩ • P(l(i))⟩
```

B.3.5 Map Expressions

B.3.5.1 Map Enumerations

Let (possibly indexed) u and v range over values of type $T1$ and $T2$, respectively, then the below expressions are simple map enumerations:

B.3.5.1.1 Map Enumerations:

```
type
   T1, T2
   M = T1 →m T2
value
   u,u1,u2,...,un:T1, v,v1,v2,...,vn:T2
   [], [u↦v], ..., [u1↦v1,u2↦v2,...,un↦vn] ∀ ∈ M
```

B.3.5.2 Map Comprehension

The last line below expresses map comprehension:

B.3.5.2.1 Map Comprehension:

```
type
   U, V, X, Y
   M = U →m V
   F = U →~ X
   G = V →~ Y
   P = U → Bool
value
   comprehend: M×F×G×P → (X →m Y)
   comprehend(m,F,G,P) ≡
      [ F(u) ↦ G(m(u)) | u:U • u ∈ dom m ∧ P(u) ]
```

B.3.6 Set Operations

B.3.6.1 Set Operator Signatures

B.3.6.1.1 Set Operations:

```
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 $\cap$: (A-**infset**)-**infset** $\to$ A-**infset**
24 $\backslash$: A-**infset** $\times$ A-**infset** $\to$ A-**infset**
25 $\subset$: A-**infset** $\times$ A-**infset** $\to$ **Bool**
26 $\subseteq$: A-**infset** $\times$ A-**infset** $\to$ **Bool**
27 =: A-**infset** $\times$ A-**infset** $\to$ **Bool**
28 $\neq$: A-**infset** $\times$ A-**infset** $\to$ **Bool**
29 **card**: A-**infset** $\stackrel{\sim}{\to}$ **Nat**

B.3.6.2 Set Examples

B.3.6.2.1 Set Examples:

examples
a ∈ {a,b,c}
a ∉ {}, a ∉ {b,c}
{a,b,c} ∪ {a,b,d,e} = {a,b,c,d,e}
∪{{a},{a,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

B.3.6.3 Informal Explication

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.

B.3.6.4 Set Operator Definitions

The operations can be defined as follows (≡ is the definition symbol):

B.3.6.4.1 Set Operation Definitions:

```
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 ∗/
```

B.3.7 Cartesian Operations

B.3.7.0.1 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
```

B.3.8 List Operations

B.3.8.1 List Operator Signatures

B.3.8.1.1 List Operations:

```
value
    hd: A^ω ~→ A
    tl: A^ω ~→ A^ω
    len: A^ω ~→ Nat
    inds: A^ω → Nat-infset
```

elems: $\mathsf{A}^\omega \to \mathsf{A}$-**infset**
.(.): $\mathsf{A}^\omega \times$ **Nat** $\stackrel{\sim}{\to}$ A
$\widehat{}$: $\mathsf{A}^* \times \mathsf{A}^\omega \to \mathsf{A}^\omega$
=: $\mathsf{A}^\omega \times \mathsf{A}^\omega \to$ **Bool**
≠: $\mathsf{A}^\omega \times \mathsf{A}^\omega \to$ **Bool**

B.3.8.2 List Operation Examples

B.3.8.2.1 List 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⟩$\widehat{}$⟨a,b,d⟩ = ⟨a,b,c,a,b,d⟩
⟨a,b,c⟩=⟨a,b,c⟩
⟨a,b,c⟩ ≠ ⟨a,b,d⟩

B.3.8.3 Informal Explication

- **hd**: Head gives the first element in a nonempty list.
- **tl**: Tail gives the remaining list of a nonempty list when Head is removed.
- **len**: Length gives the number of elements in a finite list.
- **inds**: Indices give the set of indices from 1 to the length of a nonempty list. For empty lists, this set is the empty set as well.
- **elems**: Elements gives the possibly infinite set of all distinct elements in a list.
- $\ell(i)$: Indexing with a natural number, i larger than 0, into a list ℓ having a number of elements larger than or equal to i, gives the ith element of the list.
- $\widehat{}$: 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:

B.3.8.4 List Operator Definitions

B.3.8.4.1 List Operator Definitions:

value
is_finite_list: $\mathsf{A}^\omega \to$ **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″)
```

B.3.9 Map Operations

B.3.9.1 Map Operator Signatures and Map Operation Examples

```
value
    m(a): M → A ⇸ B, m(a) = b

    dom: M → A-infset [domain of map]
        dom [a1↦b1,a2↦b2,...,an↦bn] = {a1,a2,...,an}

    rng: M → B-infset [range of map]
        rng [a1↦b1,a2↦b2,...,an↦bn] = {b1,b2,...,bn}

    †: M × M → M [override extension]
        [a↦b,a′↦b′,a″↦b″] † [a′↦b″,a″↦b′] = [a↦b,a′↦b″,a″↦b′]

    ∪: M × M → M [merge ∪]
        [a↦b,a′↦b′,a″↦b″] ∪ [a‴↦b‴] = [a↦b,a′↦b′,a″↦b″,a‴↦b‴]

    \: M × A-infset → M [restriction by]
        [a↦b,a′↦b′,a″↦b″]\{a} = [a′↦b′,a″↦b″]
```

```
/: M × A-infset → M [restriction to]
    [a↦b,a′↦b′,a″↦b″]/{a′,a″} = [a′↦b′,a″↦b″]

=,≠: M × M → Bool

°: (A →m B) × (B →m C) → (A →m C) [composition]
    [a↦b,a′↦b′] ° [b↦c,b′↦c′,b″↦c″] = [a↦c,a′↦c′]
```

B.3.9.2 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.

B.3.9.3 Map Operation Redefinitions

The map operations can also be defined as follows:

B.3.9.3.1 Map Operation Redefinitions:

```
value
  rng m ≡ { m(a) | a:A • a ∈ dom m }

  m1 † m2 ≡
    [ a↦b | a:A,b:B •
      a ∈ dom m1 \ dom m2 ∧ 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
```

B.4 λ-Calculus + Functions

B.4.1 The λ-Calculus Syntax

B.4.1.0.1 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), ...
```

B.4.2 Free and Bound Variables

B.4.2.0.1 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).

B.4.3 Substitution

In RSL, the following rules for substitution apply:

B.4.3.0.1 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)).

B.4.4 α-Renaming and β-Reduction

B.4.4.0.1 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)

B.4.5 Function Signatures

For sorts we may want to postulate some functions:

B.4.5.0.1 Sorts and Function Signatures:

```
type
    A, B, C
value
    obs_B: A → B,
    obs_C: A → C,
    gen_A: B×C → A
```

B.4.6 Function Definitions

Functions can be defined explicitly:

B.4.6.0.1 Explicit Function Definitions:

```
value
    f: Arguments → Result
    f(args) ≡ DValueExpr
```

g: Arguments $\stackrel{\sim}{\to}$ Result
g(args) ≡ ValueAndStateChangeClause
pre P(args)

Or functions can be defined implicitly:

B.4.6.0.2 Implicit Function Definitions:

value
f: Arguments → Result
f(args) **as** result
post P1(args,result)

g: Arguments $\stackrel{\sim}{\to}$ Result
g(args) **as** result
pre P2(args)
post P3(args,result)

The symbol $\stackrel{\sim}{\to}$ 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.

B.5 Other Applicative Expressions

B.5.1 Simple let Expressions

Simple (i.e., nonrecursive) **let** expressions:

B.5.1.0.1 Let Expressions:

let a = $\mathcal{E}_d$ **in** $\mathcal{E}_b$(a) **end**

is an "expanded" form of:

(λa.$\mathcal{E}_b$(a))($\mathcal{E}_d$)

B.5.2 Recursive let Expressions

Recursive **let** expressions are written as:

B.5.2.0.1 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)

B.5.3 Predicative let Expressions

Predicative **let** expressions:

B.5.3.0.1 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).

B.5.4 Pattern and "Wild Card" let Expressions

Patterns and wild cards can be used:

B.5.4.0.1 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**

B.5.5 Conditionals

Various kinds of conditional expressions are offered by RSL:

B.5.5.0.1 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
```

B.5.6 Operator/Operand Expressions

B.5.6.0.1 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⟩ ::= !
```

B.6 Imperative Constructs

B.6.1 Statements and State Changes

Often, following the RAISE method, software development starts with highly abstract-applicative constructs which, through stages of refinements, are turned into concrete and imperative constructs. Imperative constructs are thus inevitable in RSL.

B.6.1.0.1 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**.

B.6.2 Variables and Assignment

B.6.2.0.1 Variables and Assignment:

```
0. variable v:Type := expression
1. v := expr
```

B.6.3 Statement Sequences and skip

Sequencing is expressed using the ';' operator. **skip** is the empty statement having no value or side-effect.

B.6.3.0.1 Statement Sequences and **skip**:

```
2. skip
3. stm_1;stm_2;...;stm_n
```

B.6.4 Imperative Conditionals

B.6.4.0.1 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
```

B.6.5 Iterative Conditionals

B.6.5.0.1 Iterative Conditionals:

6. **while** expr **do** stm **end**
7. **do** stmt **until** expr **end**

B.6.6 Iterative Sequencing

B.6.6.0.1 Iterative Sequencing:

8. **for** e **in** list_expr • P(b) **do** S(b) **end**

B.7 Process Constructs

B.7.1 Process Channels

Let A and B stand for two types of (channel) messages and i:KIdx for channel array indexes, then:

B.7.1.0.1 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).

B.7.2 Process Composition

Let P and Q stand for names of process functions, i.e., of functions which express willingness to engage in input and/or output events, thereby communicating over declared channels. Let P() and Q stand for process expressions, then:

B.7.2.0.1 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.

B.7.3 Input/Output Events

Let c, k[i] and e designate channels of type A and B, then:

B.7.3.0.1 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.

B.7.4 Process Definitions

The below signatures are just examples. They emphasise that process functions must somehow express, in their signature, via which channels they wish to engage in input and output events.

B.7.4.0.1 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.

B.8 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:

B.8.0.0.1 Simple RSL Specifications:

type
...

variable
...
channel
...
value
...
axiom
...

B.9 RSL Module Specifications

B.9.1 Modules

Modules are clusters of one or more declarations:

```
Id =
  class
    declaration_1
    declaration_2
    ...
    declaration_n
  end
```

where declarations are either

- types
- values
- axioms
- variables
- channels
- modules

By a **class** we mean a possibly infinite set of one or more mathematical entities satisfying the declarations.

B.9.2 Schemes

```
scheme Id =
  class
    declaration_1
    declaration_2
    ...
    declaration_n
  end
```

By a **scheme** we mean a named possibly infinite set of one or more mathematical entities satisfying the declarations.

B.9.3 Module Extension

```
Id = extend Id_1,Id_2,...,Id_m with
   class
      declaration_1
      declaration_2
      ...
      declaration_n
   end
```

Usually we make sure that the extensions are conservative [145, 80, 72, 6, 103, 91].

Etcetera !