Domain Science & Engineering

A Prerequisite for Requirements Engineering

Dines Bjørner[1]
Fredsvej 11, DK-2840 Holte, Denmark
DTU, DK-2800 Kgs. Lyngby, Denmark
E–Mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~dibj

May 23, 2015: 15:31
Lecture Notes for a MAP-i PhD Course, Portugal, 25–28 May 2015
A Compendium

[1]This document is being compiled in two "parallel", commensurate versions: a "paper" and a "slide" version, www.imm.dtu.dk/~dibj/domains/daad-s.pdf. The slide version is that of Chap. 1, Appendix A and Chap. 4 of the paper version. Margin numbers of the paper version refer to slide numbers.

It's life that matters, nothing but life –
the process of discovering, the everlasting and perpetual process,
not the discovery itself, at all.

Fyodor Dostoyevsky, The Idiot, 1868, Part 3, Sect. V

A Prologue

This document has been put together in April–May 2015 from a number of (one report and otherwise from) published papers. Chapters 1, 6 and 7 are the basis for 11 lectures and 6 workshop (i.e., exercise) sessions, These are read May 25–26 and May 28, 2015 at the University of Minho. They constitute a MAP-i PhD course; MAP: Minho, Aveiro and Porto. The lectures are supported by almost 500 slides. Their page numbers are referred to in the margins of Chapters 1, 6 and 7.

Summary of PhD Course

This document takes the view that software specifications and programs are best understood as mathematical objects. This is in contrast to other views, notably such which are dominant in the USA, that the development of software is best understood as sociological and psychological
2 objects.

In this PhD course we cover two aspects of **software engineering: domain engineering** (Lectures 1–6) and **requirements engineering** (Lectures 7–10). We also cover some aspects of **domain science.**

The lectures are supported by extensive material: A comprehensive set of **lecture notes:**

www.imm.dtu.dk/~dibj/portugal/Braga-MAP-i.pdf,

and each lectures by **lecture slides:**

3 **www.imm.dtu.dk/~dibj/portugal/BL0.pdf--BL11.pdf**.

We will be together Monday, Tuesday and Thursday 10:00–17:30 **'Formal Lectures'** alternate with **'Workshop Sessions'.** In workshop sessions we shall try, You and I, to describe a domain. We will select this domain right after lunch today and start describing it. You are supposed to think about this domain mornings, before wee meet and late afternoons, after we have "left". Wednesday I will give a Faculty Seminar: **A New Foundation for Computing Science** 14:00–14:45, Room DI-A2

Monday 25 May, 2015

- **L0**: **Opening Lecture**
  Monday, 25 May 2015: 10:00–10:20
- **L1**: **An Overview of Domain Description** [Sect. 1.1 Pages 15–73]
  Monday, 25 May 2015: 10:30–11:15
- **L2**: **Parts** [Sects. 1.2.1–1.2.6 Pages 23–33]
  Monday, 25 May 2015: 11:30–12:15
- **1. Workshop:** An Example Domain
  Monday, 25 May 2015: 12:30–13:00
- **Lunch:** 13:00–14:30
- **L3**: **Unique Identifiers, Mereologies and Attributes** [Sects. 1.2.7–1.2.9 Pages 33–43]
  Monday, 25 May 2015: 14:30–15:15
- **2. Workshop:** An Example Domain
  Monday, 25 May 2015: 15:30–16:15
- **L4**: **Components, Materials – and Discussion of Endurants** [Sects. 1.2.10–1.2.13 Pages 44–52]
  Monday, 25 May 2015: 16:45–17:30

4

Tuesday 26 May, 2015

- **L5**: **Perdurants** [Sect. 1.3 Pages 52–63]
  Tuesday, 26 May 2015: 10:00–10:45
- **3. Workshop:** An Example Domain
  Tuesday, 26 May 2015: 11:00–11:45
- **L6**: **A Summary Domain Description** [Chapter 6, Pages 140–149]
  Tuesday, 26 May 2015: 12:00–13:00
- **Lunch:** 13:00–14:30
- **4. Workshop:** An Example Domain
  Tuesday, 26 May 2015: 14:30–15:15
- **L7**: **Requirements – An Overview, and Projection** [Sects. 7 and 7.2–7.2.1 Pages 151–156]
  Tuesday, 26 May 2015: 15:30–16:15
- **L8**: **Domain Requirements: Instantiation and Determination** [Sects. 7.2.2–7.2.3 Pages 156–161]
  Tuesday, 26 May 2015: 16:45–17:30

5

Wednesday 27 May:

- 14:00–14:45 **Faculty Seminar:** Room DI-A2

  Title: **A New Foundation for Computing Science** **Paper**, **Slides**
  Abstract: We argue that computing systems requirements must be based on precisely described domain models — and we argue that domain science & engineering offers a new dimension in computing. We review our work in this area and we outline a research and experimental engineering programme for the triptych of domain enginering, requirements engineering and software design.

6

Thursday 28 May, 2015

- **L9**: **Domain Requirements: Extension and Fitting** [Sects. 7.2.4–7.2.5 Pages 161–168]
  Thursday, 28 May 2015: 10:00–11:15
- **5. Workshop:** Example Domain
  Thursday, 28 May 2015: 11:30–12:00
- **L10**: **Interface Requirements** [Sect. 7.3 Pages 168–172]
  Thursday, 28 May 2015: 12:15–13:00
- **Lunch:** 13:00–14:30
- **6. Workshop:** Example Domain
  Thursday, 28 May 2015: 14:30–15:15
- **L11**: **Conclusion** [Sects. 1.4.1–1.4.2, 1.4.5–1.4.6, etc., Pages 63–64 and 72–73]
  Thursday, 28 May 2015: 15:30–16:30
- **L12**: **Discussion of Research Topics** [Sects. 1.4.4 etc., Pages 70–72 etc.]
  Thursday, 28 May 2015: 16:45–17:30

What is in this compendium ?

This compendium has been put together in preparation for the MAP-i May 24–27, 2015 PhD course at Univ. of Minho, Braga. That course covers Chapters 1, 6 and 7. Several chapters are revisions of the papers cited in the display line.

Chapter 1: Domain Analysis & Description : Pages 15–73 [42]

This chapter is the most important chapter of this compendium. It introduces the calculi of domain analysis and description prompts.

Chapter 2: Domain Facets : Pages 74–89 [30]

This chapter "extends" the domain description methodology by principles, techniques and tools for analysing and describing domain facets such as: support technologies, rules & regulations, script languages, management & organisation, and human behaviour.

Chapter 3: Prompt Semantics : Pages 90–105 [40, 44]

The calculi of domain analysis and description prompts introduced in Chapter 1 are given an operational semantics. The chapter is incomplete.

Chapter 4: Domains: Their Simulation, Monitoring and Control : Pages 106–114 [34]

This chapter, in a sense, presupposes Chapter 7. The chapter shows how domain descriptions (as well as requirements prescriptions) can be the basis for the design of simulation (demo), monitoring and control software.

Chapter 5: A Rôle for Mereology in Domain Science and Engineering : Pages 115–139 [39]

The Polish mathematician/philosopher Stanisław Leśhniewski viewed mereology [113, 57] as concerned with the understanding of parts and relations between parts (and a "whole"). In this chapter we show a model for his mereology as well as a relation between any mereology and CSP [97].

Chapter 6: A Domain Description : Pages 140–149

The next chapter, Chapter 7, assumes an existing domain description — so here it is. It is of a road net transportation system.

Chapter 7: Requirements : Pages 151–179 [26]

This chapter shows how one can systematically "derive" major aspects of requirements from domain descriptions. With this methodology we can now claim domain descriptions serve a rôle in software development.

Chapter 8: Closing : Pages 181–185

This chapter is primarily perfunctory.

Chapter 9: Bibliography : Pages 186–197

Appendices

Appendix A: RSL : Pages 199–215

The formal specification language used in this compendium for descriptions and prescriptions is RSL [85], part of the RAISE method [86]. The notation and its informal meaning is summarised in this appendix.

Appendix B: Indexes : Starting Page 216.

The indexes covers mainly Chapter 1.

Contents

I Domain Science & Engineering **14**

1 Domain Analysis & Description **15**
1.1 Introduction . . . 15
1.1.1 The TripTych Approach to Software Engineering . . . 15
1.1.2 Method and Methodology . . . 16
Method . . . 16
Discussion . . . 16
Methodology . . . 17
1.1.3 Computer and Computing Science . . . 17
1.1.4 What Is a Manifest Domain ? . . . 17
1.1.5 What Is a Domain Description ? . . . 18
1.1.6 Towards a Methodology of Domain Analysis & Description . . . 19
Practicalities of Domain Analysis & Description . . . 19
The Four Domain Analysis & Description "Players" . . . 20
An Interactive Domain Analysis & Description Dialogue . . . 20
Prompts . . . 20
A Domain Analysis & Description Language . . . 20
The Domain Description Language . . . 21
Domain Descriptions: Narration & Formalisation . . . 21
1.1.7 One Domain – Many Models ? . . . 21
1.1.8 Formal Concept Analysis . . . 21
A Formalisation . . . 21
Types Are Formal Concepts . . . 22
Practicalities . . . 22
Formal Concepts: A Wider Implication . . . 23
1.2 Endurant Entities . . . 23
1.2.1 General . . . 23
a: Analysis Prompt: is-entity . . . 23
1.2.2 Endurants and Perdurants . . . 23
b: Analysis Prompt: is-endurant . . . 24
c: Analysis Prompt: is-perdurant . . . 24
1.2.3 Discrete and Continuous Endurants . . . 24
d: Analysis Prompt: is discrete . . . 24
e: Analysis Prompt: is continuous . . . 24
1.2.4 Parts, Components and Materials . . . 25
General . . . 25
Part, Component and Material Prompts . . . 25
f: Analysis Prompt: is part . . . 25
g: Analysis Prompt: is component . . . 25
h: Analysis Prompt: is material . . . 26
1.2.5 Atomic and Composite Parts . . . 26
i: Analysis Prompt: is-atomic . . . 26
j: Analysis Prompt: is-composite . . . 26
1.2.6 On Observing Part Sorts . . . 27
Types and Sorts . . . 27
On Discovering Part Sorts . . . 27
k: Analysis Prompt: observe-parts . . . 27
Part Sort Observer Functions . . . 27
1: Description Prompt: observe-part-sorts . . . 28
On Discovering Concrete Part Types . . . 29

l: Analysis Prompt: has-concrete-type . . . 29
2: Description Prompt: observe-part-type . . . 29
Forms of Part Types . . . 30
Part Sort and Type Derivation Chains . . . 30
No Recursive Derivations . . . 30
Names of Part Sorts and Types . . . 30
More On Part Sorts and Types . . . 31
Derivation Lattices . . . 32
External and Internal Qualities of Parts . . . 32
Three Categories of Internal Qualities . . . 33
1.2.7 Unique Part Identifiers . . . 33
3: Description Prompt: observe-unique-identifier . . . 33
1.2.8 Mereology . . . 34
Part Relations . . . 34
Part Mereology: Types and Functions . . . 34
m: Analysis Prompt: has-mereology . . . 34
4: Description Prompt: observe-mereology . . . 35
Update of Mereologies . . . 36
Formulation of Mereologies . . . 37
1.2.9 Part Attributes . . . 37
Inseparability of Attributes from Endurants . . . 37
Attribute Quality and Attribute Value . . . 38
Endurant Attributes: Types and Functions . . . 38
n: Analysis Prompt: attribute-names . . . 38
The Attribute Value Observer . . . 38
5: Description Prompt: observe-attributes . . . 39
Attribute Categories . . . 40
Access to Attribute Values . . . 42
Shared Attributes . . . 42
1.2.10 Components . . . 44
o: Analysis Prompt: has-components . . . 44
6: Description Prompt: observe-component-sorts . . . 44
1.2.11 Materials . . . 45
p: Analysis Prompt: has-materials . . . 45
7: Description Prompt: observe-material-sorts . . . 45
Materials-related Part Attributes . . . 46
Laws of Material Flows and Leaks . . . 47
1.2.12 "No Junk, No Confusion" . . . 48
Pipe Routes . . . 49
Well-formed Routes . . . 50
Well-formed Pipeline Systems . . . 50
Embedded Routes . . . 51
A Theorem . . . 51
1.2.13 Discussion of Endurants . . . 52
1.3 Perdurant Entities . . . 52
1.3.1 States . . . 52
1.3.2 Actions, Events and Behaviours . . . 52
Time Considerations . . . 53
Actors . . . 53
Parts, Attributes and Behaviours . . . 53
1.3.3 Discrete Actions . . . 53
1.3.4 Discrete Events . . . 54
1.3.5 Discrete Behaviours . . . 54
Channels and Communication . . . 54
Relations Between Attribute Sharing and Channels . . . 54
1.3.6 Continuous Behaviours . . . 56
1.3.7 Attribute Value Access . . . 56
Access to Static Attribute Values . . . 56
Access to External Attribute Values . . . 56
Access to Programmable Attribute Values . . . 57
1.3.8 Perdurant Signatures and Definitions . . . 57
1.3.9 Action Signatures and Definitions . . . 57
1.3.10 Event Signatures and Definitions . . . 58
1.3.11 Discrete Behaviour Signatures and Definitions . . . 59

Process Schema I: Abstract is_composite(p) . . . 60
Process Schema II: Concrete is_composite(p) . . . 60
Process Schema III: is_atomic(p) . . . 61
Process Schema IV: Core Process (I) . . . 62
Process Schema V: Core Process (II) . . . 62
1.3.12 Concurrency: Communication and Synchronisation . . . 63
1.3.13 Summary and Discussion of Perdurants . . . 63
Summary . . . 63
Discussion . . . 63
1.4 Closing . . . 63
1.4.1 Analysis & Description Calculi for Other Domains . . . 63
1.4.2 On Domain Description Languages . . . 64
1.4.3 Comparison to Other Work . . . 64
Background: The TripTych Domain Ontology . . . 64
General . . . 64
1: Ontology Science & Engineering: . . . 64
2: Knowledge Engineering: . . . 66
Specific . . . 66
3: Database Analysis: . . . 67
4: Domain Analysis: . . . 67
5: Domain Specific Languages: . . . 67
6: Feature-oriented Domain Analysis (FODA): . . . 67
7: Software Product Line Engineering: . . . 67
8: Problem Frames: . . . 68
9: Domain Specific Software Architectures (DSSA): . . . 68
10: Domain Driven Design (DDD): . . . 68
11: Unified Modeling Language (UML): . . . 69
12: Requirements Engineering: . . . 69
Summary of Comparisons . . . 69
1.4.4 Open Problems . . . 70
1: Ontology Relations: . . . 70
2: Analysis of Perdurants: . . . 70
3: Commensurate Discrete and Continuous Models: . . . 70
4: Interplay between Parts and Materials: . . . 70
5: Dynamics: . . . 70
6: Precise Descriptions of Manifest Domains: . . . 71
7: Towards Mathematical Models of Domain Analysis & Description: . . . 71
8: Laws of Descriptions: A Calculus of Prompts: . . . 71
9: Domains and Galois Connections: . . . 72
10: Laws of Domain Description Prompts: . . . 72
11: Domain Theories:: . . . 72
1.4.5 Tony Hoare's Summary on 'Domain Modeling' . . . 72
1.4.6 Beauty Is Our Business . . . 73

2 Domain Facets 74
2.1 Stake-holders . . . 74
2.2 Domain Facets . . . 74
2.2.1 Intrinsics . . . 75
Conceptual Versus Actual Intrinsics . . . 77
On Modelling Intrinsics . . . 77
2.2.2 Support Technologies . . . 77
On Modelling Support Technologies . . . 79
2.2.3 Management and Organisation . . . 79
Conceptual Analysis, First Part . . . 80
Methodological Consequences . . . 80
Conceptual Analysis, Second Part . . . 81
On Modelling Management and Organisation . . . 82
2.2.4 Rules and Regulations . . . 82
A Meta-characterisation of Rules and Regulations . . . 83
On Modelling Rules and Regulations . . . 84
2.2.5 Scripts and Licensing Languages . . . 84
Licensing Languages . . . 86
On Modelling Scripts . . . 86
2.2.6 Human Behaviour . . . 86

A Meta-characterisation of Human Behaviour . . . 87
On Modelling Human Behaviour . . . 88
2.2.7 Completion . . . 88
2.2.8 Integrating Formal Descriptions . . . 88
2.3 Closing Discussion . . . 89

3 Prompt Semantics 90
3.1 A Model of The Analysis & Description Process . . . 90
3.1.1 A Summary of Prompts . . . 90
3.1.2 Preliminaries . . . 91
3.1.3 Initialising the Domain Analysis & Description Process . . . 91
3.1.4 A Domain Analysis & Description State . . . 91
3.1.5 Analysis & Description of Endurants . . . 91
Analysis & Description of Part Sorts . . . 93
Analysis & Description of Part Materials . . . 94
Analysis & Description of Material Parts . . . 94
Analysis & Description of Composite Endurants . . . 94
Analysis & Description of Concrete Sort Types . . . 95
Analysis & Description of Abstract Sorts . . . 96
Analysis & Description of Unique Identifiers . . . 96
Analysis & Description of Mereologies . . . 97
Analysis & Description of Part Attributes . . . 97
3.1.6 Discussion of The Model . . . 97
Termination . . . 97
Axioms and Proof Obligations . . . 98
Order of Analysis & Description: A Meaning of '⊕' . . . 98
3.2 A Model of The Analysis & Description Prompts . . . 98
3.2.1 On the Domain Analyser's Image of Domains . . . 98
3.2.2 An Abstract Syntax of Domains . . . 98
Domain Nodes . . . 98
The Root Domain Node . . . 100
Domain Description Trees . . . 100
Syntax . . . 100
Generating Description Tree Paths . . . 100
Well-formedness of Domain Nodes . . . 101
Well-formed Composite and Material Nodes . . . 101
No Recursively Defined Sorts . . . 102
No Duplicate Definitions . . . 102
Defined Duplicate Sort Names . . . 102
3.2.3 Node Selection . . . 103
3.2.4 Index of Prompts . . . 103
Analysis Prompts . . . 104
Description Prompts . . . 104
3.2.5 A Formal Description of a Meaning of Prompts . . . 104
The Iterative Nature of The Description Process . . . 104
How Are We Modelling the Prompts . . . 104
The Model . . . 105
3.2.6 Discussion . . . 105
3.3 Discussion of the Models . . . 105

4 Domains: Their Simulation, Monitoring and Control 106
4.1 Introduction . . . 106
4.2 Domain Descriptions . . . 107
4.3 Interpretations . . . 107
4.3.1 What Is a Domain-based Demo? . . . 107
Examples . . . 107
Towards a Theory of Visualisation and Acoustic Manifestation . . . 108
4.3.2 Simulations . . . 108
Explication of Figure 4.1 . . . 108
Script-based Simulation . . . 109
The Development Arrow . . . 110
4.3.3 Monitoring & Control . . . 110
Monitoring . . . 111
Control . . . 111

4.3.4 Machine Development . . . 111
Machines . . . 111
Requirements Development . . . 111
4.3.5 Verifiable Software Development . . . 112
An Example Set of Conjectures . . . 112
Chains of Verifiable Developments . . . 113
4.4 Conclusion . . . 113
4.4.1 Discussion . . . 114
What Have We Achieved . . . 114
What Have We Not Achieved — Some Conjectures . . . 114
What Should We Do Next . . . 114

5 A Rôle for Mereology in Domain Science and Engineering 115
5.1 Introduction . . . 115
5.1.1 Computing Science Mereology . . . 115
5.1.2 From Domains via Requirements to Software . . . 116
5.1.3 Domains: Science and Engineering . . . 117
5.1.4 Contributions of This Contribution . . . 117
5.1.5 Structure of This Contribution . . . 117
5.2 Our Concept of Mereology . . . 117
5.2.1 Informal Characterisation . . . 117
5.2.2 Six Examples . . . 118
Air Traffic . . . 118
Buildings . . . 119
Financial Service Industry . . . 120
Machine Assemblies . . . 120
Oil Industry . . . 121
"The" Overall Assembly . . . 121
A Concretised Composite parts . . . 122
Railway Nets . . . 122
Discussion . . . 122
5.3 An Abstract, Syntactic Model of Mereologies . . . 123
5.3.1 Parts and Subparts . . . 123
The Model . . . 123
5.3.2 'Within' and 'Adjacency' Relations . . . 124
'Within' . . . 124
'Transitive Within' . . . 125
'Adjacency' . . . 125
Transitive 'Adjacency' . . . 125
5.3.3 Unique Identifications . . . 126
5.3.4 Attributes . . . 127
5.3.5 Connections . . . 127
Connector Wellformedness . . . 128
Connector and Attribute Sharing Axioms . . . 128
Sharing . . . 129
5.3.6 Uniqueness of Parts . . . 129
Uniqueness of Embedded and Adjacent Parts . . . 129
5.4 An Axiom System . . . 129
5.4.1 Parts and Attributes . . . 129
$\mathscr{P}$ The Part Sort . . . 129
$\mathscr{A}$ The Attribute Sort . . . 129
5.4.2 The Axioms . . . 130
$\mathbb{P}$ Part-hood . . . 130
$\mathbb{PP}$ Proper Part-hood . . . 130
$\mathbb{O}$ Overlap . . . 130
$\mathbb{U}$ Underlap . . . 130
$\mathbb{OX}$ Over-cross . . . 130
$\mathbb{UX}$ Under-cross . . . 130
$\mathbb{PO}$ Proper Overlap . . . 131
5.4.3 Satisfaction . . . 131
A Proof Sketch . . . 131
5.5 An Analysis of Properties of Parts . . . 131
5.5.1 Mereological Properties . . . 132
An Example . . . 132

Unique Identifier and Mereology Types . . . 132
5.5.2 Properties . . . 133
5.5.3 Attributes . . . 134
5.5.4 Discussion . . . 136
5.6 A Semantic CSP Model of Mereology . . . 136
5.6.1 A Semantic Model of a Class of Mereologies . . . 136
Parts ≃ Processes . . . 136
Connectors ≃ Channels . . . 136
Process Definitions . . . 137
5.6.2 Discussion . . . 138
General . . . 138
Partial Evaluation . . . 138
5.7 Concluding Remarks . . . 138
5.7.1 Relation to Other Work . . . 138
5.7.2 What Has Been Achieved ? . . . 139
5.7.3 Future Work . . . 139

6 A Domain Description 140
6.1 Endurants . . . 140
6.1.1 Domain, Net, Fleet and Monitor . . . 140
6.1.2 Hubs and Links . . . 141
6.1.3 Unique Identfiers . . . 141
6.1.4 Mereology . . . 142
6.1.5 Attributes, I . . . 143
6.1.6 Attributes, II . . . 143
6.1.7 Routes . . . 146
6.2 Perdurants . . . 146
6.2.1 Vehicle to Monitor Channel . . . 146
6.2.2 Link Disappearance Event . . . 146
6.2.3 Road Traffic . . . 147

II Requirements Engineering 150

7 Requirements 151
7.1 Introduction . . . 151
7.1.1 General Considerations . . . 151
7.1.2 Four Stages of Requirements Development . . . 153
Problem and/or Objective Sketch . . . 153
Systems Requirements . . . 153
User and External Equipment Requirements . . . 153
Functional Requirements . . . 154
7.2 Domain Requirements . . . 154
7.2.1 Domain Projection . . . 154
Domain Projection — Narrative . . . 154
Domain Projection — Formalisation . . . 155
A Projection Operator . . . 156
7.2.2 Domain Instantiation . . . 156
Domain Instantiation — Narrative . . . 157
Domain Instantiation — Formalisation . . . 157
Domain Instantiation — Formalisation: Well-formedness . . . 158
Summary Well-formedness Predicate . . . 159
Domain Instantiation — Abstraction . . . 159
An Instantiation Operator . . . 160
7.2.3 Domain Determination . . . 160
Domain Determination: Example . . . 160
All Toll-road Links are One-way Links . . . 160
All Toll-road Hubs are Free-flow . . . 160
A Domain Determination Operator . . . 161
7.2.4 Domain Extension . . . 161
The Core Requirements Example: Domain Extension . . . 161
A Domain Extension Operator . . . 167
7.2.5 Requirements Fitting . . . 167
Some Definitions . . . 167
Requirements Fitting Procedure — A Sketch . . . 168

Requirements Fitting – An Example . . . 168
7.2.6 Domain Requirements Consolidation . . . 168
7.3 Interface Requirements . . . 168
7.3.1 Shared Phenomena . . . 168
7.3.2 Shared Endurants . . . 169
Data Initialisation . . . 169
Data Refreshment . . . 172
7.3.3 Shared Actions, Events and Behaviours . . . 172
7.4 Machine Requirements . . . 172
7.4.1 Delineation of Machine Requirements . . . 172
On Machine Requirements . . . 172
Machine Requirements Facets . . . 172
7.4.2 Performance Requirements . . . 172
7.4.3 Dependability Requirements . . . 173
Failures, Errors and Faults . . . 173
Accessibility . . . 174
Availability . . . 174
Integrity . . . 175
Safety . . . 175
Security . . . 176
Robustness . . . 176
7.4.4 Maintenance Requirements . . . 176
Delineation and Facets of Maintenance Requirements . . . 176
Adaptive Maintenance . . . 176
Corrective Maintenance . . . 177
Perfective Maintenance . . . 177
Preventive Maintenance . . . 177
Extensional Maintenance . . . 177
7.4.5 Platform Requirements . . . 178
Delineation and Facets of Platform Requirements . . . 178
Development Platform . . . 178
Execution Platform . . . 178
Maintenance Platform . . . 178
Demonstration Platform . . . 178
7.4.6 Documentation Requirements . . . 179
7.4.7 Discussion . . . 179

III Conclusion 180

8 Discussion of Research Topics 181
8.1 Domain Science & Engineering Topics . . . 181
8.1.1 Analysis & Description Calculi for Other Domains . . . 181
8.1.2 On Domain Description Languages . . . 181
8.1.3 Ontology Relations . . . 182
8.1.4 Analysis of Perdurants . . . 182
8.1.5 Commensurate Discrete and Continuous Models . . . 182
8.1.6 Interplay between Parts, Materials and Components . . . 182
8.1.7 Dynamics . . . 182
8.1.8 Precise Descriptions of Manifest Domains . . . 183
8.1.9 Towards Mathematical Models of Domain Analysis & Description . . . 183
8.1.10 Laws of Descriptions: A Calculus of Prompts . . . 183
8.1.11 Domains and Galois Connections . . . 184
8.1.12 Laws of Domain Description Prompts . . . 184
8.1.13 Domain Theories: . . . 184
8.1.14 External Attributes . . . 185
8.2 Requirements Topics . . . 185
8.2.1 Domain Requirements Methodology . . . 185
8.2.2 Domain Requirements Operator Theory . . . 185
8.2.3 Methodology for Interface Requirements . . . 185

9 Bibliography **186**
9.1 Bibliographical Notes . . . 186
9.1.1 Published Papers . . . 186
9.1.2 Reports . . . 186
9.2 References . . . 187

IV Appendix **198**

A RSL **199**
A.1 RSL: The Raise Specification Language . . . 199
A.1.1 Type Expressions . . . 199
Atomic Types . . . 199
Composite Types . . . 199
Concrete Composite Types . . . 199
Sorts and Observer Functions . . . 201
A.1.2 Type Definitions . . . 201
Concrete Types . . . 201
Subtypes . . . 202
Sorts — Abstract Types . . . 202
A.1.3 The RSL Predicate Calculus . . . 202
Propositional Expressions . . . 202
Simple Predicate Expressions . . . 202
Quantified Expressions . . . 202
A.1.4 Concrete RSL Types: Values and Operations . . . 203
Arithmetic . . . 203
Set Expressions . . . 203
Set Enumerations . . . 203
Set Comprehension . . . 203
Cartesian Expressions . . . 203
Cartesian Enumerations . . . 203
List Expressions . . . 203
List Enumerations . . . 203
List Comprehension . . . 204
Map Expressions . . . 204
Map Enumerations . . . 204
Map Comprehension . . . 204
Set Operations . . . 204
Set Operator Signatures . . . 204
Set Examples . . . 205
Informal Explication . . . 205
Set Operator Definitions . . . 206
Cartesian Operations . . . 206
List Operations . . . 206
List Operator Signatures . . . 206
List Operation Examples . . . 206
Informal Explication . . . 207
List Operator Definitions . . . 207
Map Operations . . . 208
Map Operator Signatures and Map Operation Examples . . . 208
Map Operation Explication . . . 208
Map Operation Redefinitions . . . 209
A.1.5 λ-Calculus + Functions . . . 209
The λ-Calculus Syntax . . . 209
Free and Bound Variables . . . 210
Substitution . . . 210
α-Renaming and β-Reduction . . . 210
Function Signatures . . . 211
Function Definitions . . . 211
A.1.6 Other Applicative Expressions . . . 211
Simple let Expressions . . . 211
Recursive let Expressions . . . 212
Predicative let Expressions . . . 212
Pattern and "Wild Card" let Expressions . . . 212

Conditionals . . . 212
Operator/Operand Expressions . . . 213
A.1.7 Imperative Constructs . . . 213
Statements and State Changes . . . 213
Variables and Assignment . . . 214
Statement Sequences and skip . . . 214
Imperative Conditionals . . . 214
Iterative Conditionals . . . 214
Iterative Sequencing . . . 214
A.1.8 Process Constructs . . . 214
Process Channels . . . 214
Process Composition . . . 214
Input/Output Events . . . 215
Process Definitions . . . 215
A.1.9 Simple RSL Specifications . . . 215

B Indexes **216**
B.1 Index of Endurant Analysis Prompts . . . 216
B.2 Description Language Observers and "Built-in" Functions . . . 216
B.3 Domain Description Prompts and Their Schemas . . . 216
B.4 Attribute Analysis Prompts . . . 217
B.5 [Well–formedness] Axioms . . . 217
B.6 [Disjoint Sort] Proof Obligations . . . 217
B.7 Definitions . . . 217
B.8 Examples . . . 222
B.9 Concepts . . . 225
B.10 RSL Language Constructs . . . 233

Part I

Domain Science & Engineering

This part reflects of a number domain issues: [1] how to analyse & describe manifest domains, [2] domain facets, [3] an operational semantics of the domain prompts of Chapter 1, [4] the simulation, monitoring and control of domains [5] a relation between Leśniewski's Mereology and our way of descrining, hence modelling manifest domains, and [6] an example domain description.

Chapter 1

Domain Analysis & Description

Abstract

We show that manifest domains, an understanding of which are a prerequisite for software requirements prescriptions, can be precisely described: narrated and formalised. We show that manifest domains can be understood as a collection of endurant, that is, basically spatial entities: parts, components and materials, and perdurant, that is, basically temporal entities: actions, events and behaviours. We show that parts can be modeled in terms of external qualities whether: atomic or composite parts, having internal qualities: unique identifications, mereologies, which model relations between parts, and attributes. We show the manifest domain analysis endeavour can be supported by a calculus of manifest domain analysis prompts: `is_entity`, `is_endurant`, `is_perdurant`, `is_part`, `is_component`, `is_material`, `is_atomic`, `is_composite`, `has_components`, `has_materials`, `has_concrete_type`, `attribute_names`, `is_stationary`, etcetera.We show how the manifest domain description endeavour can be supported by a calculus of manifest domain description prompts: `observe_part_sorts`, `observe_part_type`, `observe_components`, `observe_materials`, `observe_unique_identifier`, `observe_mereology`, `observe_attributes`, `observe_location` and `observe_position`. We show how to model essential aspects of perdurants in terms of their signatures based on the concepts of endurants.And we show how one can "compile" descriptions of endurant parts into descriptions of perdurant behaviours. We do not show prompt calculi for perdurants. The above contributions express a method with principles, technique and tools for constructing domain descriptions.

1.1 Introduction

The broader subject of this chapter is that of software development. The narrower subject is that of manifest domain engineering. We see software development in the context of the `TripTych` approach, see next section, just below. The contribution of this chapter is twofold: the propagation of manifest domain engineering as a first phase of the development of a large class of software — and a set of principles, techniques and tools for the engineering of the analysis & descriptions of manifest domains. These principles, techniques and tools are embodied in a set of analysis and description prompts. We claim that this embodiment in the form of prompts is novel, that the (yet to be investigated) "calculus" is a first such "method calculus".

1.1.1 The `TripTych` Approach to Software Engineering

We suggest a `TripTych` view of software engineering: *before software can be designed and coded we must have a reasonable grasp of "its" requirements; before requirements can be prescribed*

we must have a reasonable grasp of "the underlying" domain. To us, therefore, software engineering contains the three sub-disciplines:

- domain engineering,
- requirements engineering and
- software design.

This paper contributes, we claim, to a methodology for domain analysis &[1] domain description. References [26, 31] show how to "refine" domain descriptions into requirements prescriptions, and reference [34] indicates more general relations between domain descriptions and domain demos, domain simulators and more general domain specific software.

In branches of engineering based on natural sciences professional engineers are educated in these sciences. Telecommunications engineers know Maxwell's Laws. Maybe they cannot themselves "discover" such laws, but they can "refine" them into designs, for example, for mobile telephony radio transmission towers. Aeronautical engineers know laws of fluid mechanics. Maybe they cannot themselves "discover" such laws, but they can "refine" them into designs, for example, for the design of airplane wings. And so forth for other engineering branches.

Our point is here the following: software engineers must domain specialise. This is already done, to a degree, for designers of compilers, operating systems, database systems, Internet/Web systems, etcetera. But is it done for software engineering banking systems, traffic systems, health care, insurance, etc. ? We do not think so, but we claim it should be done.

1.1.2 Method and Methodology

Method

By a **method** we shall understand a "somehow structured" set of `principles` for `select`ing and `apply`ing a number of `techniques` and `tools` for `analysing` problems and `synthesizing` solutions for a given domain ■[2]

The 'somehow structuring' amounts, in this treatise on domain analysis & description, to the techniques and tools being related to a set of domain analysis & description "prompts", "issued by the method", prompting the domain engineer, hence carried out by the domain analyser & describer[3] — conditional upon the result of other prompts.

Discussion

There may be other 'definitions' of the term 'method'. The above is the one that will be adhered to in this paper. The main idea is that there is a clear understanding of what we mean by, as here, a software development method, in particular a *domain analysis & description method.*

The **main principles** of the `TripTych` domain analysis and description approach are those of abstraction and both narrative and formal modeling. This means that evolving domain descriptions necessarily limit themselves to a subset of the domain focusing on what is considered relevant, that is, abstract "away" some domain phenomena.

The **main techniques** of the `TripTych` domain analysis and description approach are besides those techniques which are in general associated with formal descriptions, focus on the techniques that relate to the deployment of of the individual prompts.

And the **main tools** of the `TripTych` domain analysis and description approach are the analysis and description prompts and the description language, here the `Raise Specification Language RSL` [85].

[1] When, as here, we write *A* & *B* we mean *A* & *B* to be one subject.

[2] Definitions and examples are delimited by ■ respectively ■ symbols.

[3] We shall thus use the term domain engineer to cover both the analyser & the describer.

A main contribution of this paper is therefore that of "painstakingly" elucidating the principles, techniques and tools of the domain analysis & description method.

Methodology 30

By **methodology** we shall understand the study and knowledge about one or more methods[4] ■

1.1.3 Computer and Computing Science 31

By **computer science** we shall understand the study and knowledge of the conceptual phenomena that "exists" inside computers and, in a wider context than just computers and computing, of the theories "behind" their formal description languages ■ Computer science is often also referred to as theoretical computer science. 32

By **computing science** we shall understand the study and knowledge of how to construct and describe those phenomena ■ Another term for computing science is programming methodology. 33

This paper is a computing science paper. It is concerned with the construction of domain descriptions. It puts forward a calculus for analysing and describing domains. It does not theorize about this calculus. There are no theorems about this calculus and hence no proofs. We leave that to another study and paper.

1.1.4 What Is a Manifest Domain ? 34

We offer a number of complementary delineations of what we mean by a manifest domain. But first some examples, "by name" !

Example 1 . **Manifest Domain Names**: Examples of suggestive names of manifest domains are: *air traffic, banks, container lines, documents, hospitals, manufacturing, pipelines, railways* and *road nets* ■ 35

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 ■ 36

Example 2 . **Manifest Domain Endurants**: Examples of (names of) endurants are ***Air traffic:*** *aircraft, airport, air lane.* ***Banks****: client, passbook.* ***Container lines:*** *container, container vessel, container terminal port.* ***Documents:*** *document, document collection.* ***Hospitals:*** *patient, medical staff, ward, bed, patient medical journal.* ***Pipelines:*** *well, pump, pipe, valve, sink, oil.* ***Railways:*** *simple rail unit, point, crossover, line, track, station.* ***Road nets:*** *link (street segment), hub (street intersection)* ■ 37

Example 3 . **Manifest Domain Perdurants**: Examples of (names of) perdurants are ***Air traffic:*** *start (ascend) an aircraft, change aircraft course.* ***Banks:*** *open, deposit into, withdraw from, close (an account).* ***Container lines:*** *move container off or on board a vessel.* ***Documents:*** *open, edit, copy, shred.* ***Hospitals:*** *admit, diagnose, treat (patients).* ***Pipelines:*** *start pump, stop pump, open valve, close valve.* ***Railways:*** *switch rail point, start train.* ***Road nets:*** *set a hub signal, sense a vehicle* ■ 38

A **manifest domain** is further seen as a mapping from *entities* to *qualities*, that is, a mapping from manifest phenomena to usually non-manifest qualities ■ 39

Example 4 . **Endurant Entity Qualities**: Examples of (names of) endurant qualities: ***Pipeline:*** *unique identity of a pipeline unit, mereology (connectedness) of a pipeline unit, length of a pipe, (pumping) height of a pump, open/close status of a valve.* ***Road net:*** *unique identity of a road unit (hub or link), road unit mereology: identity of neighbouring hubs of a link, identity of links emanating from a hub, and state of hub (traversal) signal* ■

Example 5 . Perdurant Entity Qualities: Examples of (names of) perdurant qualities: ***Pipeline:*** *the signature of an open (or close) valve action, the signature of a start (or stop) pump action, etc.* ***Road net:*** *the signature of an insert (or remove) link action, the signature of an insert (or remove) hub action, the signature of a vehicle behaviour, etc.* ■

Our definitions of what a manifest domain is are, to our own taste, not fully adequate; they ought be so sharp that one can unequivocally distinguish such domains that are not manifest domains from those which are (!). Examples of the former are: the Internet, language compilers, operating systems, data bases, etcetera. As we progress we shall sharpen our definition of 'manifest domain'.

We shall in the rest of this chapter just write 'domain' instead of 'manifest domain'.

1.1.5 What Is a Domain Description ?

By a **domain description** we understand a collection of pairs of narrative and commensurate formal texts, where each pair describes either aspects of an endurant entity or aspects of a perdurant entity ■

What does it mean that some text describes a domain entity ?

For a text to be a **description text** it must be possible from that text to either, if it is a narrative, to reason, informally, that the *designated* entity is described to have some properties that the reader of the text can observe that the described entities also have; or, if it is a formalisation to prove, mathematically, that the formal text *denotes* the postulated properties ■

Example 6 . Narrative Description of Bank System Endurants:

1 A banking system consists of a bank and collections of clients and of passbooks.

2 A bank attribute is that of a general ledger.

3 A collection of clients is a set of uniquely identified clients.

4 A collection of passbooks is a set of uniquely identified passbooks.

5 A client "possess" zero, one or more passbook identifiers.

6 Two or more clients may share the same passbook.

7 The general ledger records, for each passbook identifier, amongst others, the set of one or more client identifiers sharing that passbook, etc.

Etcetera ■

Example 7 . Formal Description of Bank System Endurants:

type
1. B, CC, CPB
value
1. **obs_part**_CC: B → CC,
1. **obs_part**_CPB: B → CPB
type
2. GL
value
2. **attr**_GL: B → GL
type
3. C, CI, CC = C-**set**,
4. PB, PBI, CPB = PB-**set**
value
5. **attr**_C: C → PBI-**set**
type
7. GL = PBI $\xrightarrow[m]{}$ SH × ...
7. SH = PBI-**set**

[4]Please note our distinction between method and methodology. We often find the two, to us, separate terms used interchangeably.

Etcetera ■

46

Example 8 . Narrative Description of Bank System Perdurants:

8 Clients and the bank possess cash (i.e., monies).

9 Clients can open a bank account and receive in return a passbook.

10 Clients may deposit monies into an account in response to which the passbook and the general ledger are updated.

11 Clients may withdraw monies from an account: if the balance of monies in the designated account is not less than the requested amount the client is given the (natural number) designated monies and the passbook and the general ledger are updated.

Etcetera ■

47

Example 9 . Formal Description of Bank System Perdurants:

type
8. M

value
8. **attr**_M: (B|C) → M
9. open: B → B × PB
10. deposit: PB → M → B → B × PB
11. withdraw: PB → B → **Nat** $\xrightarrow{\sim}$ B × PB × M

Etcetera ■

48

By a **domain description** we shall thus understand a text which describes the entities of the domain: whether endurant or perdurant, and when endurant whether discrete or continuous, atomic or composite; or when perdurant whether actions, events or behaviours. as well as the qualities of these entities.

49

So the task of the domain analyser cum describer is clear: There is a domain: right in front of our very eyes, and it is expected that that domain be described.

Section 9.1.2 lists 10 draft reports (accessible on the Internet). They give examples of domain descriptions. These descriptions were carried out in order to research and develop the domain analysis and description concepts now summarised in the present chapter. These reports ought now be revised, some slightly, others less so, so as to follow all of the prescriptions of the current chapter.

1.1.6 Towards a Methodology of Domain Analysis & Description

50

Practicalities of Domain Analysis & Description How does one go about analysing & describing a domain ? Well, for the first, one has to designate one or more domain analysers cum domain describers, i.e., trained domain scientists cum domain engineers. How does one get hold of a domain engineer ? One takes a software engineer and *educates* and *trains* that person in domain science & domain engineering. A derivative purpose of this paper is to unveil aspects of domain science & domain engineering. The education and training consists in bringing forth a number of scientific and engineering issues of domain analysis and of domain description. Among the engineering issues are such as: *what do I do when confronted with the task of domain analysis ?* and *with the task of description ?* and *when, where and how do I* `select` *and* `apply` *which* `techniques` *and which* `tools` ? Finally, there is the issue of *how do I, as a domain describer, choose appropriate abstractions and models ?*

51

52

The Four Domain Analysis & Description "Players" We can say that there are four 'players' at work here. the domain, the domain analyser & describer, the domain analysis & description method, and the evolving domain analysis & description. (i) The *domain* is there. The domain analyser & describer cannot change the domain. Analysing & describing the domain does not change it[5]. In a meta-physical sense it is inert. In the physical sense the domain will usually contain entities that are static (i.e., constant), and entities that are dynamic (i.e., variable). (ii) The *domain analyser & domain describer* is a human, preferably a scientist/engineer[6], well-educated and trained in domain science & engineering. The domain analyser & describer observes the domain, analyses it according to a method and thereby produces a domain description. (iii) As a concept the *method* is here considered "fixed". By 'fixed' we mean that its principles, techniques and tools do not change during a domain analysis & description. The domain analyser & describer may very well apply these principles, techniques and tools more-or-less haphazardly during domain analysis & description, flaunting the method, but the method remains invariant. The method, however, may vary from one domain analysis & description (project) to another domain analysis & description (project). Domain analysers & describers, may, for example, have become wiser from a project to the next. (iv) Finally there is the evolving *domain analysis & description*. That description is a text, usually both informal and formal. Applying a *domain description prompt* to the domain yields an *additional domain description text* which is added to the thus evolving *domain description*. One may speculate of the rôle of the "input" domain description. Does it change ? Does it help determine the additional domain description text ? Etcetera. Without loss of generality we can assume that the "input" domain description is changed[7] and that it helps determine the added text.

Of course, analysis & description is a trial-and-error, iterative process. During a sequence of analyses, that is, analysis prompts, the analyser "discovers" either more pleasing abstractions or that earlier analyses or descriptions were wrong. So they are corrected.

An Interactive Domain Analysis & Description Dialogue We see domain analysis & description as a process involving the above-mentioned four 'players', that is, as a dialogue between the domain analyser & describer and the domain, where the dialogue is guided by the method and the result is the description. We see the method as a 'player' which issues prompts: alternating between: *"analyse this"* (analysis prompts) and *"describe that"* (synthesis or, rather, description prompts).

Prompts In this paper we shall suggest a number of *domain analysis prompts* and a number of *domain description prompts*. The **domain analysis prompt**s, (schematically: `analyse_named_condition`(e)) directs the analyser to inquire as to the truth of whatever the prompt "names" at wherever part (component or material), e, in the domain the prompt so designates. Based on the truth value of an analysed entity the domain analyser may then be prompted to describe that part (or material). The **domain description prompt**s, (schematically: `describe_type_or_quality`(e)) directs the (analyser cum) describer to formulate both an informal and a formal description of the type or qualities of the entity designated by the prompt.

The prompts form languages, and there are thus two languages at play here.

A Domain Analysis & Description Language The 'Domain Analysis & Description Language' thus consists of a number of meta-functions, the prompts. The meta-functions have names (say `is_endurant`) and types, but have no formal definition. They are not computable. They are "performed" by the domain analysers & describers. These meta-functions are systematically introduced and informally explained in Sect. 1.2.

[5] Observing domains, such as we are trying to encircle the concept of domain, is not like observing the physical world at the level of subatomic particles. The experimental physicists' instruments of observation changes what is being observed.

[6] At the present time domain analysis appears to be partly an art, partly a scientific endeavour. Until such a time when domain analysis & description principles, techniques and tools have matured it will remain so.

[7] for example being "stylistically" revised.

The Domain Description Language The 'Domain Description Language' is RSL [85], the RAISE Specification Language [86]. With suitable, simple adjustments it could also be either of Alloy [100], Event B [1], VDM–SL [48, 49, 77] or Z [157]. We have chosen RSL because of its simple provision for defining sorts, expressing axioms, and postulating observers over sorts. 64

Domain Descriptions: Narration & Formalisation Descriptions *must* be readable and *should* be mathematically precise.[8] For that reason we decompose domain description fragments into clearly identified[9] "pairs" of narrative texts and formal texts.

1.1.7 One Domain – Many Models ? 65

Will two or more domain engineers cum scientists arrive at "the same domain description" ? No, almost certainly not ! What do we mean by "the same domain description" ? To each proper description we can associate a mathematical meaning, its semantics. Not only is it very unlikely that the syntactic form of the domain descriptions are the same or even "marginally similar". But it is also very unlikely that the two (or more) semantics are the same; that is, that all properties that can be proved for one domain model can be proved also for the other, and vice versa. Why will different domain models emerge ? Two different domain 66 describers will, undoubtedly, when analysing and describing independently, focus on different aspects of the domain. One describer may focus attention on certain phenomena, different from those chosen by another describer. One describer may choose some abstractions where another may choose more concrete presentations. Etcetera. We can thus expect that a set of domain description developments lead to a 67 set of distinct models. As these domain descriptions are communicated amongst domain engineers cum scientists we can expect that iterated domain description developments within this group of developers will lead to fewer and more similar models. Just like physicists, over the centuries of research, have arrived at a few models of nature, we can expect there to develop some consensus model of "standard" domains. We expect, that sometime in future, software engineers, when commencing software development for a 68 "standard domain", that is, one for which there exists one or more "standard models", will start with the development of a domain description based on "one of the standard models" — just like control engineers of automatic control "repeat" an essence of a domain model for a control problem. We follow up on this modeling issue in Sect. 1.4.4, Item 8, Page 71: *"Laws of Descriptions: A Calculus of Prompts"*.

1.1.8 Formal Concept Analysis 69

Domain analysis involves that of concept analysis. As soon as we have identified an entity for analysis we have identified a concept. The entity is a spatio-temporal, i.e., a physical thing. Once we speak of it, it becomes a concept. Instead of examining just one entity the domain analyser shall examine many entities. Instead of describing one entity the domain describer shall describe a class of entities. Ganter & Wille's [83] addresses this issue.

A Formalisation 70

This section is a transcription of Ganter & Wille's [83] *Formal Concept Analysis, Mathematical Foundations*, the 1999 edition, Pages 17–18.

Some Notation: By $\mathscr{E}$ we shall understand the type of entities; by $\mathbb{E}$ we shall understand an entity of type $\mathscr{E}$; by $\mathscr{Q}$ we shall understand the type of qualities; by $\mathbb{Q}$ we shall understand a quality of type $\mathscr{Q}$; by $\mathscr{E}$**-set** we shall understand the type of sets of entities; by $\mathbb{ES}$ we shall understand a set of entities of type $\mathscr{E}$**-set**; by $\mathscr{Q}$**-set** we shall understand the type of sets of qualities; and by $\mathbb{QS}$ we shall understand a a set of qualities of type $\mathscr{Q}$**-set**. 71

[8] One must insist on formalised domain descriptions in order to be able to verify that domain descriptions satisfy a number of properties not explicitly formulated as well as in order to verify that requirements prescriptions satisfy domain descriptions.

[9] The "clear identification" is here achieved by narrative text item and corresponding formula line numbers.

Definition: 1 ***Formal Context:*** A **formal context** $\mathbb{K} := (\mathbb{ES}, \mathbb{I}, \mathbb{QS})$ consists of two sets; $\mathbb{ES}$ of entities and $\mathbb{QS}$ of qualities, and a relation $\mathbb{I}$ between $\mathbb{E}$ and $\mathbb{Q}$. ■

To express that $\mathbb{E}$ is in relation $\mathbb{I}$ to a Quality $\mathbb{Q}$ we write $\mathbb{E} \cdot \mathbb{I} \cdot \mathbb{Q}$, which we read as *"entity* $\mathbb{E}$ ***has*** *quality* $\mathbb{Q}$*"*.

Example endurant entities are a specific vehicle, another specific vehicle, etcetera; a specific street segment (link), another street segment, etcetera; a specific road intersection (hub), another specific road intersection, etcetera, a monitor. Example endurant entity qualities are (a vehicle) has mobility, (a vehicle) has velocity ($\geq$0), (a vehicle) has acceleration, etcetera; (a link) has length ($>$0), (a link)has location, (a link)has traffic state, etcetera.

Definition: 2 ***Qualities Common to a Set of Entities:*** For any subset, $s\mathbb{ES} \subseteq \mathbb{ES}$, of entities we can define $\mathscr{DQ}$ for "derive[d] set of qualities".

$\mathscr{DQ} : \mathscr{E}$**-set** $\to$ ($\mathscr{E}$**-set** $\times$ $\mathscr{I}$ $\times$ $\mathscr{Q}$**-set**) $\to$ $\mathscr{Q}$**-set**
$\mathscr{DQ}(s\mathbb{ES})(\mathbb{ES}, \mathbb{I}, \mathbb{QS}) \equiv \{\mathbb{Q} \mid \mathbb{Q}{:}\mathscr{Q}, \mathbb{E}{:}\mathscr{E} \cdot \mathbb{E} \in s\mathbb{ES} \wedge \mathbb{E} \cdot \mathbb{I} \cdot \mathbb{Q}\}$
pre: $s\mathbb{ES} \subseteq \mathbb{ES}$

The above expresses: *"the set of qualities common to entities in* $s\mathbb{ES}$*"*. ■

Definition: 3 ***Entities Common to a Set of Qualities:*** For any subset, $s\mathbb{QS} \subseteq \mathbb{QS}$, of qualities we can define $\mathscr{DE}$ for "derive[d] set of entities".

$\mathscr{DE}$: $\mathscr{Q}$**-set** $\to$ ($\mathscr{E}$**-set** $\times$ $\mathscr{I}$ $\times$ $\mathscr{Q}$**-set**) $\to$ $\mathscr{E}$**-set**
$\mathscr{DE}(s\mathbb{QS})(\mathbb{ES}, \mathbb{I}, \mathbb{QS}) \equiv \{\mathbb{E} \mid \mathbb{E}{:}\mathscr{E}, \mathbb{Q}{:}\mathscr{Q} \cdot \mathbb{Q} \in s\mathbb{Q} \wedge \mathbb{E} \cdot \mathbb{I} \cdot \mathbb{Q}\}$,
pre: $s\mathbb{QS} \subseteq \mathbb{QS}$

The above expresses: *"the set of entities which have all qualities in* $s\mathbb{QS}$*"*. ■

Definition: 4 ***Formal Concept:*** A **formal concept** of a context $\mathbb{K}$ is a pair:

- $(s\mathbb{Q}, s\mathbb{E})$ where
  - $\mathscr{DQ}(s\mathbb{E})(\mathbb{E}, \mathbb{I}, \mathbb{Q}) = s\mathbb{Q}$ and
  - $\mathscr{DE}(s\mathbb{Q})(\mathbb{E}, \mathbb{I}, \mathbb{Q}) = s\mathbb{E}$;
- $s\mathbb{Q}$ is called the **intent** of $\mathbb{K}$ and $s\mathbb{E}$ is called the **extent** of $\mathbb{K}$. ■

Types Are Formal Concepts

76

Now comes the "crunch": *In the* `TripTych` *domain analysis we strive to find formal concepts and, when we think we have found one, we assign a type (or a sort) and qualities to it !*

Practicalities

77

There is a little problem. To search for all those entities of a domain which each have the same sets of qualities is not feasible. So we do a combination of two things: (i) we identify a small set of entities all having the same qualities and tentatively associate them with a type, and (ii) we identify certain nouns of our national language and if such a noun does indeed designate a set of entities all having the same set of qualities then we tentatively associate the noun with a type. Having thus, tentatively, identified a type we conjecture that type and search for counterexamples, that is, entities which refutes the conjecture. This "process" of conjectures and refutations is iterated until some satisfaction is arrived at that the postulated type constitutes a reasonable conjecture.

Formal Concepts: A Wider Implication 79

The formal concepts of a domain form Galois Connections [83]. We gladly admit that this fact is one of the reasons why we emphasise formal concept analysis. At the same time we must admit that this paper does not do justice to this fact. We have experimented with the analysis & description of a number of domains and have noticed such Galois connections but it is, for us, too early to report on this. Thus we invite the reader to study this aspect of domain analysis.

1.2 Endurant Entities 80

In the rest of this chapter we shall consider entities in the context of their being manifest (i.e., spatio-temporal).

1.2.1 General

Definition 1 . ***Entity:*** *By an* ***entity*** *we shall understand a* ***phenomenon****, i.e., something that can be* ***observed****, i.e., be seen or touched by humans, or that can be* ***conceived*** *as an* ***abstraction*** *of an entity. We further demand that an entity can be objectively described* ■[10]

81

Analysis Prompt 1 . ***is_entity:*** *The domain analyser analyses "things" (θ) into either entities or non-entities. The method can thus be said to provide the* ***domain analysis prompt****:*

- `is_entity` — *where* `is_entity` *(θ) holds if θ is an entity* ■[11]

`is_entity` is said to be a **prerequisite prompt** for all other prompts. 82

Whither Entities: The "demands" that entities be observable and objectively describable raises some philosophical questions. Are sentiments, like feelings, emotions or "hunches" observable? This author thinks not. And, if so, can they be other than artistically described? It seems that psychologically and aesthetically "phenomena" appears to lie beyond objective description. We shall leave these speculations for later.

1.2.2 Endurants and Perdurants 83

Definition 2 . ***Endurant:*** *By an* ***endurant*** *we shall understand an entity that can be observed or conceived and described as a "complete thing" at no matter which given snapshot of time. Were we to "freeze" time we would still be able to observe the entire endurant* ■

That is, endurants "reside" in space. Endurants are, in the words of Whitehead [154], continuants. 84

Example 10 . **Traffic System Endurants**: Examples of traffic system endurants are: traffic system, road nets, fleets of vehicles, sets of hubs (i.e., street intersections), sets of links (i.e., street segments [between hubs]), and individual hubs, links and vehicles ■

85

Definition 3 . ***Perdurant:*** *By a* ***perdurant*** *we shall understand an entity for which only a fragment exists if we look at or touch them at any given snapshot in time, that is, where we to freeze time we would only see or touch a fragment of the perdurant* ■

That is, perdurants "reside" in space and time. Perdurants are, in the words of Whitehead [154], occurrents. 86

[10] **Definitions** and **examples** are delimited by ■ respectively ■

[11] **Analysis** prompt definitions and **description** prompt definitions and schemes are delimited by ■ respectively ■.

Example 11 . Traffic System Perdurants: Examples of road net perdurants are: insertion and removal of hubs or links (actions), disappearance of links (events), vehicles entering or leaving the road net (actions), vehicles crashing (events) and road traffic (behaviour) ■

87

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

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

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

Analysis Prompt 3 . `is_perdurant:` *The domain analyser analyses an entity ϕ into perdurants as prompted by the* ***domain analysis prompt****:*

- `is_perdurant` — *ϕ is a perdurant if* `is_perdurant(`ϕ`)` *holds.*

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

88

In the words of Whitehead [154] — as communicated by Sowa [145, Page 70] — an endurant has stable qualities that enable its various appearances at different times to be recognised as the same individual; a perdurant is in a state of flux that prevents it from being recognised by a stable set of qualities.

89

Necessity and Possibility: It is indeed possible to make the endurant/perdurant distinction. But is it necessary ? We shall argue that it is 'by necessity' that we make this distinction. Space and time are fundamental notions. They cannot be dispensed with. So, to describe manifest domains without resort to space and time is not reasonable.

1.2.3 Discrete and Continuous Endurants

90

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

91

Example 12 . Discrete Endurants: Examples of discrete endurants are a road net, a link, a hub, a vehicle, a traffic signal, etcetera ■

92

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

93

Example 13 . Continuous Endurants: Examples of continuous endurants are water, oil, gas, sand, grain, etcetera ■

94

Analysis Prompt 4 . `is_discrete:` *The domain analyser analyse endurants e into discrete entities as prompted by the* ***domain analysis prompt****:*

- `is_discrete` — *e is discrete if* `is_discrete(e)` *holds* ■

Analysis Prompt 5 . `is_continuous:` *The domain analyser analyse endurants e into continuous entities as prompted by the* ***domain analysis prompt****:*

- `is_continuous` — *e is continuous if* `is_continuous(e)` *holds* ■

1.2.4 Parts, Components and Materials

95

General

Definition 6 . ***Part:*** *By a **part** we shall understand a discrete endurant which the domain engineer chooses to endow with **internal qualities** such as unique identification, mereology, and one or more attributes* ■

96 We shall define the terms 'unique identification', 'mereology', and 'attributes' shortly.

Example 14 . **Parts**: Example 10 on Page 23 illustrated, and examples 18 on the next page and 19 on the following page illustrate parts ■

97

Definition 7 . ***Component:*** *By a **component** we shall understand a discrete endurant which we, the domain analyser cum describer chooses to **not** endow with **internal qualities*** ■

98

Example 15 . **Components**: Examples of components are: chairs, tables, sofas and book cases in a living room, letters, newspapers, and small packages in a mail box, machine assembly units on a conveyor belt, boxes in containers of a container vessel, etcetera ■

99

"At the Discretion of the Domain Engineer": We emphasise the following analysis and description aspects: (a) The domain is full of observable phenomena. It is the decision of the domain analyser cum describer whether to analyse and describe some such phenomena, that is, whether to include them in a domain model. (b) The borderline between an endurant being (considered) discrete or being (considered) continuous is fuzzy. It is the decision of the domain analyser cum describer whether to model an endurant as discrete or continuous. (c) The borderline between a discrete endurant being (considered) a part or being 100
(considered) a component is fuzzy. It is the decision of the domain analyser cum describer whether to model a discrete endurant as a part or as a component. (d) In Sect. 1.3.11 we shall show how to "compile" parts into processes. A factor, therefore, in determining whether to model a discrete endurant as a part or as a component is whether we may consider a discrete endurant as also representing a process. 101

Definition 8 . ***Material:*** *By a **material** we shall understand a continuous endurant* ■

Example 16 . **Materials**: Examples of material endurants are: air of an air conditioning system, grain of a silo, gravel of a barge, oil (or gas) of a pipeline, sewage of a waste disposal system, and water of a hydro-electric power plant. ■

102

Example 17 . **Parts Containing Materials**: Pipeline units are here considered discrete, i.e., parts. Pipeline units serve to convey material ■

Part, Component and Material Prompts

103

Analysis Prompt 6 . ***is_part:*** *The domain analyser analyse endurants e into part entities as prompted by the **domain analysis prompt**:*

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

We remind the reader that the outcome of `is_part(e)` is very much dependent on the domain engineer's intention with the domain description, cf. Sect. 1.2.4. 104

Analysis Prompt 7 . ***is_component:*** *The domain analyser analyse endurants e into component entities as prompted by the **domain analysis prompt**:*

- `is_component` — *e is a component if* `is_component(e)` *holds* ■

We remind the reader that the outcome of `is_component(e)` is very much dependent on the domain engineer's intention with the domain description, cf. Sect. 1.2.4 on the preceding page.

Analysis Prompt 8 . ***`is_material:`*** *The domain analyser analyse endurants e into material entities as prompted by the* ***domain analysis prompt****:*

- *`is_material` — e is a material if `is_material(e)` holds* ■

We remind the reader that the outcome of `is_material(e)` is very much dependent on the domain engineer's intention with the domain description, cf. Sect. 1.2.4 on the previous page.

•••

Sections 1.2.5–1.2.9 (Pages 26–43) focus on the external and internal qualities of parts. In contrast, Sects. 1.2.10–1.2.11 (Pages 44–48) focus on components and materials.

1.2.5 Atomic and Composite Parts

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

Definition 9 . ***Atomic Part:*** ***Atomic part****s are those which, in a given context, are deemed to not consist of meaningful, separately observable proper* sub-part*s* ■

A **sub-part** is a part ■

Example 18 . **Atomic Parts**: Examples of atomic parts of the above mentioned domains are: aircraft (of air traffic), demand/deposit accounts (of banks), containers (of container lines), documents (of document systems), hubs, links and vehicles (of road traffic), patients, medical staff and beds (of hospitals), pipes, valves and pumps (of pipeline systems), and rail units and locomotives (of railway systems) ■

Definition 10 . ***Composite Part:*** ***Composite part****s are those which, in a given context, are deemed to* indeed *consist of meaningful, separately observable proper* sub-part*s* ■

Example 19 . **Composite Parts**: Examples of atomic parts of the above mentioned domains are: airports and air lanes (of air traffic), banks (of a financial service industry), container vessels (of container lines), dossiers of documents (of document systems), routes (of road nets), medical wards (of hospitals), pipelines (of pipeline systems), and trains, rail lines and train stations (of railway systems). ■

Analysis Prompt 9 . ***`is_atomic:`*** *The domain analyser analyses a discrete endurant, i.e., a part p into an atomic endurant:*

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

Analysis Prompt 10 . ***`is_composite:`*** *The domain analyser analyses a discrete endurant, i.e., a part p into a composite endurant:*

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

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

Whither Atomic or Composite: If we are analysing & describing vehicles in the context of a road net, cf. Example 10 on Page 23, then we have chosen to abstract vehicles as atomic; if, on the other hand, we are analysing & describing vehicles in the context of an automobile maintenance garage then we might very well choose to abstract vehicles as composite — the sub-parts being the object of diagnosis by the auto mechanics.

1.2.6 On Observing Part Sorts 114

Types and Sorts

We use the term 'sort' when we wish to speak of an abstract type [140], that is, a type for which we do not wish to express a model[12]. We shall use the term 'type' to cover both abstract types and concrete types.

On Discovering Part Sorts 115

Recall from Sect. 1.1.8 on Page 22 that we "equate" a formal concept with a type (i.e., a sort). Thus, to us, a part sort is a set of all those entities which all have exactly the same qualities. Our aim now is to present the basic principles that let the domain analyser decide on part sorts. We observe parts one-by-one. 116

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

Thus there is, in this analysis, a "eureka", a step where we shift focus from the concrete to the abstract, from observing specific part instances to postulating a sort: from one to the many. 117

Analysis Prompt 11 . ***`observe_parts:`*** *The* ***domain analysis prompt****:*

- *`observe_parts(p)`*

directs the domain analyser to observe the sub-parts of p ■

Let us say the sub-parts of p are: $\{p_1,p_2,\ldots,p_m\}$

(β) The analyser analyses, for each of these parts, p_{i_k}, which formal concept, i.e., sort, it belongs to; let us say that it is of sort P_k; thus the sub-parts of p are of sorts $\{P_1,P_2,\ldots,P_m\}$. Some P_k may be atomic sorts, some may be composite sorts. 118

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

(γ) It is therefore concluded, that is, decided, that $\{p_i,p_j,p_\ell,\ldots,p_n\}$ are all of the same part sort P with observable part sub-sorts $\{P_1,P_2,\ldots,P_m\}$. 119

Above we have *type-font-highlighted* three sentences: (α,β,γ). When you analyse what they "prescribe" you will see that they entail a "depth-first search" for part sorts. The β sentence says it rather directly: *"The analyser analyses, for each of these parts, p_k, which formal concept, i.e., part sort it belongs to."* To do this analysis in a proper way, the analyser must ("recursively") analyse the parts "down" to their atomicity, and from the atomic parts decide on their part sort, and work ("recurse") their way "back", through possibly intermediate composite parts, to the p_ks.

Part Sort Observer Functions 120

The above analysis amounts to the analyser first "applying" the domain analysis prompt `is_composite`(p) to a discrete endurant, where we now assume that the obtained truth value is **true**. Let us assume that parts p:P consists of sub-parts of sorts $\{P_1,P_2,\ldots,P_m\}$. Since we cannot automatically guarantee that our domain descriptions secure that P and each P_i ($[1\leq i\leq m]$) denotes disjoint sets of entities we must prove it. 121

[12] for example, in terms of the concrete types: sets, Cartesians, lists, maps, or other.

Domain Description Prompt 1 . ***observe_part_sorts:*** *If* `is_composite(p)` *holds, then the analyser "applies" the description language observer prompt*

- `observe_part_sorts(p)`

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

1. `observe_part_sorts` schema

Narration:
[s] ... narrative text on sorts ...
[o] ... narrative text on sort observers ...
[i] ... narrative text on sort recognisers ...
[p] ... narrative text on proof obligations ...
Formalisation:
type
[s] P,
[s] P_i [$1 \leq i \leq m$] **comment:** P_i [$1 \leq i \leq m$] abbreviates P_1, P_2, ..., P_m
value
[o] **obs_part_**P_i: P $\rightarrow$ P_i [$1 \leq i \leq m$]
[i] **is_**P_i: $P_i \rightarrow$ **Bool** [$1 \leq i \leq m$]

proof obligation [Disjointness of part sorts]
[p] $\forall\ p{:}(P_1|P_2|...|P_m)$ •
[p] $\bigwedge\ \{$**is_**P_i(p) $\equiv \bigvee \sim \{$**is_**P_j(p) | j $\in$ {1..m} \ {i}$\}$ | i $\in$ {1..m}$\}$

`is_composite` *is a* ***prerequisite prompt*** *of* `observe_part_sorts` ■

We do not here state guidelines for discharging these kinds of proof obligations. But we will very informally sketch such discharges, see below.

Example 20 . **Composite and Atomic Part Sorts of Transportation**: The following example illustrates the multiple use of the `observe_part_sorts` function: first to δ, a specific transport domain, Item 12, then to an $n : N$, the net of that domain, Item 13, and then to an $f : F$, the fleet of that domain, Item 14.

12 A transportation domain is viewed as composed from a net (of hubs and links), a fleet (of vehicles) and a monitor.

13 A transportation net is here seen as composed from a collection of hubs and a collection of links.

14 A fleet is here seen as a collection of vehicles.

The monitor is considered an atomic part.

type
12. N, F, M
value
12. **obs_part_**N:$\Delta \rightarrow$N, **obs_part_**F:$\Delta \rightarrow$F, **obs_part_**M:$\Delta \rightarrow$M
type
13. HC, LC
value
13. **obs_part_**HC:N$\rightarrow$HC, **obs_part_**LC:N$\rightarrow$LC

type
14. VC
value
14. **obs_part**_VC:F→VC

A proof obligation has to be discharged, one that shows disjointedness of sorts N, F and M. An informal sketch is: entities of sort N are composite and consists of two parts: aggregations of hubs, HS, and aggregations of links, LS. Entities of sort F consists of an aggregation, VS, of vehicles. So already that makes N and F disjoint. M is an atomic entity — where N and F are both composite. Hence the three sorts N, F and M are disjoint ■

On Discovering Concrete Part Types

Analysis Prompt 12 . ***has_concrete_type:*** *The domain analyser may decide that it is expedient, i.e., pragmatically sound, to render a part sort, P, whether atomic or composite, as a concrete type, T. That decision is prompted by the holding of the* ***domain analysis prompt****:*

- `has_concrete_type(p)`.

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

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

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

- `observe_part_type(p)`[13]

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

2. `observe_part_type` schema

Narration:
[t1] ... narrative text on sorts and types S_i ...
[t2] ... narrative text on types T ...
[o] ... narrative text on type observers ...
Formalisation:
type
[t1] $S_1, S_2, ..., S_m, ..., S_n$,
[t2] $T = \mathscr{E}(S_1,S_2,...,S_n)$
value
[o] **obs_part**_T: P → T

where $S_1,S_2,...,S_m,...,S_n$ may be any types, including part sorts, where $0 \leq m \leq n \geq 1$, where m is the number of new (atomic or composite) sorts, and where $n-m$ is the number of concrete types (like **Bool, Int, Nat***) or sorts already analysed & described. and $\mathscr{E}(S_1,S_2,...,S_n)$ is a type expression* ■

The type names, T, of the concrete type, as well as those of the auxiliary types, $S_1,S_2,...,S_m$, are chosen by the domain describer: they may have already been chosen for other sort–to–type descriptions, or they may be new.

[13] `has_concrete_type` is a **prerequisite prompt** of `observe_part_type`.

Example 21 . **Concrete Part Types of Transportation**: We continue Example 20 on Page 28:

15 A collection of hubs is here seen as a set of hubs and a collection of links is here seen as a set of links.

16 Hubs and links are, until further analysis, part sorts.

17 A collection of vehicles is here seen as a set of vehicles.

18 Vehicles are, until further analysis, part sorts.

type
15. Hs = H-**set**, Ls = L-**set**
16. H, L
17. Vs = V-**set**
18. V

value
15. **obs_part**_Hs:HC→Hs, **obs_part**_Ls:LC→Ls
17. **obs_part**_Vs:VC→Vs ■

Forms of Part Types

131

Usually it is wise to restrict the part type definitions, $T_i = \mathscr{E}_i(Q,R,...,S)$, to simple type expressions. T=A-**set** or T=A* or T=ID $\xrightarrow[m]{}$ A or T=A$_t$|B$_t$|...|C$_t$ where ID is a sort of unique identifiers, T=A$_t$|B$_t$|...|C$_t$ defines the disjoint types A$_t$==mkA$_s$(s:A$_s$), B$_t$==mkB$_s$(s:B$_s$), ..., C$_t$==mkC$_s$(s:C$_s$), and where A, A$_s$, B$_s$, ..., C$_s$ are sorts. Instead of A$_t$==mkA(a:A$_s$), etc., we may write A$_t$::A$_s$ etc.

Part Sort and Type Derivation Chains

132

Let P be a composite sort. Let P$_1$, P$_2$, ..., P$_m$ be the part sorts "discovered" by means of `observe_part_sorts`(p) where p:P. We say that P$_1$, P$_2$, ..., P$_m$ are (immediately) **derived** from P. If P$_k$ is
133 derived from P$_j$ and P$_j$ is derived from P$_i$, then, by transitivity, P$_k$ is **derived** from P$_i$.

No Recursive Derivations We "mandate" that if P$_k$ is derived from P$_j$ then there can be no P derived from P$_j$ such that P is P$_j$, that is, P$_j$ cannot be derived from P$_j$.

That is, we do not allow recursive domain sorts.

It is not a question, actually of allowing recursive domain sorts. It is, we claim to have observed, in very many domain modeling experiments, that there are no recursive domain sorts !

Names of Part Sorts and Types

134

The domain analysis and domain description text prompts `observe_part_sorts`, `observe_material_sorts` and `observe_part_type` — as well as the `attribute_names`, `observe_material_sorts`, `observe_unique_identifier`, `observe_mereology` and `observe_attributes` prompts introduced below — "yield" type names. That is, it is as if there is a reservoir of an indefinite-size
135 set of such names from which these names are "pulled", and once obtained are never "pulled" again. There may be domains for which two distinct part sorts may be composed from identical part sorts. In this case the domain analyser indicates so by prescribing a part sort already introduced.

Example 22 . **Container Line Sorts**: Our example is that of a container line with container vessels and
136 container terminal ports.

19 A container line contains a number of container vessels and a number of container terminal ports, as well as other components.

20 A container vessel contains a container stowage area, etc.

21 A container terminal port contains a container stowage area, etc.

22 A container stowage ares contains a set of uniquely identified container bays.

23 A container bay contains a set of uniquely identified container rows.

24 A container row contains a set of uniquely identified container stacks.

25 A container stack contains a stack, i.e., a first-in, last-out sequence of containers.

26 Containers are further undefined.

After a some slight editing we get: 137

```
type
  CL
  VS, VI, V, Vs = VI →m V,
  PS, PI, P, Ps = PI →m P
value
  obs_part_VS: CL → VS
  obs_part_Vs: VS → Vs
  obs_part_PS: CL → PS
  obs_part_Ps: CTPS → CTPs
type
  CSA
value
  obs_part_CSA: V → CSA
  obs_part_CSA: P → CSA
```

```
type
  BAYS, BI, BAY, Bays=BI →m BAY
  ROWS, RI, ROW, Rows=RI →m ROW
  STKS, SI, STK, Stks=SI →m STK
  C
value
  obs_part_BAYS: CSA → BAYS,
  obs_part_Bays: BAYS → Bays
  obs_part_ROWS: BAY → ROWS,
  obs_part_Rows: ROWS → Rows
  obs_part_STKS: ROW → STKS,
  obs_part_Stks: STKS → Stks
  obs_part_Stk: STK → C*
```

Note that observe_part_sorts(v:V) and observe_part_sorts(p:P) both yield CSA ■

More On Part Sorts and Types 138

The above "experimental example" motivates the below. We can always assume that composite parts p:P abstractly consists of a definite number of sub-parts.

Example 23. We comment on Example 20 on Page 28: parts of type Δ and N are composed from three, respectively two abstract sub-parts of distinct types ■

Some of the parts, say p_{i_z} of $\{p_{i_1},p_{i_2},\ldots,p_{i_m}\}$, of p:P, may themselves be composite.

Example 24. We comment on Example 20 on Page 28: parts of type N, F, HC, LC and VC are all composite ■ 139

There are, pragmatically speaking, two cases for such compositionality. Either the part, p_{i_z}, of type t_{i_z}, is is composed from a definite number of abstract or concrete sub-parts of distinct types.

Example 25. We comment on Example 20 on Page 28: parts of type N are composed from three sub-parts ■

Or it is composed from an indefinite number of sub-parts of the same sort.

Example 26. We comment on Example 20 on Page 28: parts of type HC, LC and VC are composed from an indefinite numbers of hubs, links and vehicles, respectively ■ 140

Example 27 . **Pipeline Parts**:

27 A pipeline consists of an indefinite number of pipeline units.

28 A pipeline units is either a well, or a pipe, or a pump, or a valve, or a fork, or a join, or a sink.

29 All these unit sorts are atomic and disjoint.

type
27. PL, U, We, Pi, Pu, Va, Fo, Jo, Si
27. Well, Pipe, Pump, Valv, Fork, Join, Sink
value
27. **obs_part**_Us: PL → U-**set**
type
28. U == We | Pi | Pu | Va | Fo | Jo | Si
29. We::Well, Pi::Pipe, Pu::Pump, Va::Valv, Fo:Fork, Jo::Join, Si::Sink

141 The experimental research report [38] covers pipelines in some detail ■

Derivation Lattices Derivation chains start with the domain name, say Δ, and (definitively) end with the name of an atomic sort. Sets of derivation chains form join lattices [12].

Example 28 . Derivation Chains: Figure 1.1 illustrates two part sort and type derivation chains. based
142 on Examples 20 on Page 28 and 22 on Page 30, respectively. The "–>" of Fig. 1.1 stands for $\xrightarrow{m}$ ■

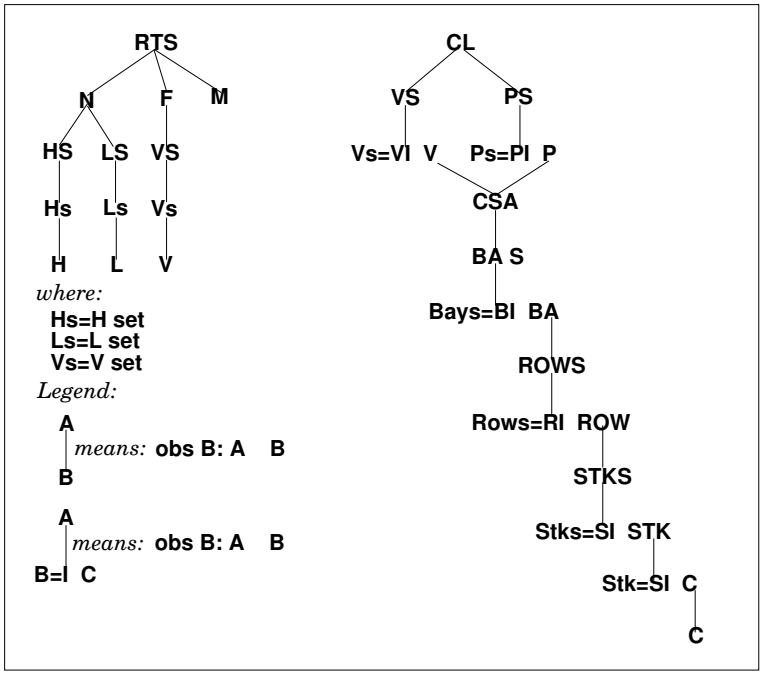

Figure 1.1: Two Domain Lattices: Examples 20 on Page 28 and 22 on Page 30

External and Internal Qualities of Parts 143

By an **external part quality** we shall understand the `is_atomic`, `is_composite`, `is_discrete` and `is_continuous` qualities. By an **internal part quality** we shall understand the part qualities to be outlined in the next sections: `unique identification`, `mereology` and `attributes`. By **part qualities** we mean the sum total of `external endurant` and `internal endurant` qualities.

Three Categories of Internal Qualities 144

We suggest that the internal qualities of parts be analysed into three categories: (i) a category of unique part identifiers, (ii) a category of mereological quantities and (iii) a category of general attributes. Part mereologies are about sharing qualities between parts. Some such sharing expresses spatio-topological properties of how parts are organised. Other part sharing aspects express relations (like equality) of part attributes. We base our modeling of mereologies on the notion of unique part identifiers. Hence we cover **internal qualities** in the order (i–ii–iii). 145

1.2.7 Unique Part Identifiers 146

Two parts are either identical or a distinct, i.e., unique. Two parts are identical if all their respective qualities have the same values. That is, their location in space/time are one and the same. Two parts are distinct even if all the attribute qualities of the two parts, that we have chosen to consider have the same values, if, in that case, their space/time locations are distinct. 147

We can assume, without any loss of generality, (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 sort PI of P), (iii) such that distinct part sorts, P_i and P_j, have distinctly named unique identifier sorts, say PI_i and PI_j, (iv) that all π_i:PI_i and π_j:PI_j are distinct, and (v) that the observer function **uid**_P applied to p yields the unique identifier, say π:PI, of p. 148

Representation of Unique Identifiers: Unique identifiers are abstractions. When we endow two parts (say of the same sort) with distinct unique identifiers then we are simply saying that these two parts are distinct. We are not assuming anything about how these identifiers otherwise come about. 149

Domain Description Prompt 3 . ***observe_unique_identifier:*** *We can therefore apply the* ***domain description prompt****:*

- `observe_unique_identifier`

to parts p:P resulting in the analyser writing down the unique identifier type and observer domain description text according to the following schema: 150

3. `observe_unique_identifier` schema

Narration:
[s] ... narrative text on unique identifier sort ...
[u] ... narrative text on unique identifier observer ...
[a] ... axiom on uniqueness of unique identifiers ...
Formalisation:
type
[s] PI
value
[u] **uid**_P: P → PI
axiom
[a] $\mathscr{U}$

$\mathscr{U}$ *is a predicate over part sorts and unique part identifier sorts. The unique part identifier sort,* PI*, is unique, as are all part sort names,* P*.*

151

Example 29 . **Unique Transportation Net Part Identifiers**: We continue Example 20 on Page 28.

30 Links and hubs have unique identifiers

31 and unique identifier observers.

type
30. LI, HI
value
31. **uid**_LI: L → LI
31. **uid**_HI: H → HI
axiom [Well−formedness of Links, L, and Hubs, H]
30. ∀ l,l′:L • l≠l′⇒**uid**_LI(l)≠**uid**_LI(l′),
30. ∀ h,h′:H • h≠h′⇒**uid**_HI(h)≠**uid**_HI(h′) ■

1.2.8 Mereology

152

Mereology is the study and knowledge of parts and part relations. Mereology as a logical/philosophical discipline can perhaps best be attributed to the Polish mathematician/logician Stanisław Leśniewski [57, 39].

Part Relations

153

Which are the relations that can be relevant for part-hood ? We give some examples. Two otherwise distinct parts may share attribute values. [14]

Example 30 . **Shared Attribute Mereology**: (i) two or more distinct public transport busses may run according to the same, thus "shared", bus time table; (ii) all vehicles in a traffic participate in that traffic, each with their "share", that is, position on links or at hubs – as observed by the (thus postulated, and shared) traffic observer. etcetera ■

154

Two otherwise distinct parts may be said to, for example, be topologically "adjacent" or one "embedded" within the other.

Example 31 . **Topological Connectedness Mereology**:(i) two rail units may be connected (i.e., adjacent), (ii) a road link may be connected to two road hubs; (iii) a road hub may be connected to zero or more road links; etcetera. ■

The above examples are in no way indicative of the "space" of part relations that may be relevant for part-hood. The domain analyser is expected to do a bit of experimental research in order to discover necessary, sufficient and pleasing "mereology-hoods" !

Part Mereology: Types and Functions

155

Analysis Prompt 13 . ***has_mereology:*** *To discover necessary, sufficient and pleasing "mereology-hoods" the analyser can be said to endow a truth value* **true** *to the* ***domain analysis prompt****:*

- *has_mereology*

156

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

We can define a **mereology type** as a type $\mathscr{E}$xpression over unique [part] identifier types. We generalise to unique [part] identifiers over a definite collection of part sorts, P1, P2, ..., Pn, where the parts p1:P1, p2:P2, ..., pn:Pn are not necessarily (immediate) sub-parts of some part p:P.

type
PI1, PI2, ..., PIn
MT = $\mathscr{E}$(PI1, PI2, ..., PIn),

157

Domain Description Prompt 4 . ***observe_mereology:*** *If* `has_mereology(p)` *holds for parts p of type P, then the analyser can apply the* ***domain description prompt****:*

- `observe_mereology`

to parts of that type and write down the mereology types and observers *domain description text according to the following schema:* 158

4. `observe_mereology` schema

Narration:
[t] ... narrative text on mereology type ...
[m] ... narrative text on mereology observer ...
[a] ... narrative text on mereology type constraints ...
Formalisation:
type
[t] MT[15] = $\mathscr{E}$(PI1,PI2,...,PIm)
value
[m] **obs_mereo**_P: P → MT
axiom [Well−formedness of Domain Mereologies]
[a] $\mathscr{A}$(MT)

159

Here $\mathscr{E}$(PI1,PI2,...,PIm) is a type expression over possibly all unique identifier types of the domain description, and $\mathscr{A}$(MT) is a predicate over possibly all unique identifier types of the domain description. To write down the concrete type definition for MT requires a bit of analysis and thinking. `has_mereology` *is a* ***prerequisite prompt*** *for* `observe_mereology` ■

160

Example 32 . **Road Net Part Mereologies**: We continue Example 20 on Page 28 and Example 29 on Page 33.

32 Links are connected to exactly two distinct hubs.

33 Hubs are connected to zero or more links.

34 For a given net the link and hub identifiers of the mereology of hubs and links must be those of links and hubs, respectively, of the net.

161

type
32. LM′ = HI-**set**, LM = {|his:HI-**set** • **card**(his)=2|}
33. HM = LI-**set**
value
32. **obs_mereo**_L: L → LM
33. **obs_mereo**_H: H → HM
axiom [Well−formedness of Road Nets, N]
34. ∀ n:N,l:L,h:H• l ∈ **obs_part**_Ls(**obs_part**_LC(n))∧h ∈ **obs_part**_Hs(**obs_part**_GC(n))
34. **let** his=mereology_H(l), lis=mereology_H(h) **in**
34. his⊆∪{**uid**_H(h) | h ∈ **obs_part**_Hs(**obs_part**_HC(n))}
34. ∧ lis⊆∪{**uid**_H(l) | l ∈ **obs_part**_Ls(**obs_part**_LC(n))} **end** ■

162

Example 33 . **Pipeline Parts Mereology**: We continue Example 27 on Page 31. Pipeline units serve to conduct fluid or gaseous material. The flow of these occur in only one direction: from so-called input to so-called output.

35 Wells have exactly one connection to an output unit.

36 Pipes, pumps and valves have exactly one connection from an input unit and one connection to an output unit.

37 Forks have exactly one connection from an input unit and exactly two connections to distinct output units.

38 Joins have exactly one two connection from distinct input units and one connection to an output unit.

39 Sinks have exactly one connection from an input unit.

40 Thus we model the mereology of a pipeline unit as a pair of disjoint sets of unique pipeline unit identifiers.

163

```
type
40.  UM′=(UI-set×UI-set)
40.  UM={|(iuis,ouis):UI-set×UI-set•iuis ∩ ouis={}|}
value
40.  obs_mereo_U: UM
axiom [Well−formedness of Pipeline Systems, PLS (0)]
    ∀ pl:PL,u:U • u ∈ obs_part_Us(pl) ⇒
        let (iuis,ouis)=obs_mereo_U(u) in
        case (card iuis,card ouis) of
35.       (0,1) → is_We(u),
36.       (1,1) → is_Pi(u)∨is_Pu(u)∨is_Va(u),
37.       (1,2) → is_Fo(u),
38.       (2,1) → is_Jo(u),
39.       (1,0) → is_Si(u)
        end end
```

Example 50 on Page 47 (axiom Page 47), Example 51 on Page 48 (axiom Page 48) and Example 52 on Page 49 (axiom Page 50) illustrates the need to constrain the sets of endurant entities denoted by definitions of part sort, unique identifier and mereology attribute definitions. ■

Update of Mereologies

164

We normally consider a part's mereology to be constant. There may, however, be cases where the mereology of a part changes. In order to update mereology values the description language offers the "built-in" operator:

Mereology Update Function

- **upd_mereology**: P → M → P

165 for all relevant M and P. The meaning of **upd_mereology** is, informally:

[14] For the concept of attribute value see Sect. 1.2.9 on Page 38.
[15] MT will be used several times in Sect. 1.3.11.

```
type
    P, M
value
    upd_mereology: P → M → P
    upd_mereology(p)(m) as p′
        post: obs_mereo_H(p′) = m
```

166

The above is a simplification. It lacks explaining that all other aspects of the part p:P are left unchanged. It also omits mentioning some proof obligations. The updated mereology must, for example, only specify such unique identifiers of parts that are indeed existing parts. A proper formal explication requires that we set up a formal model of the domain/method/analyser/description quadrangle. 167

Example 34 . **Mereology Update**: The example is that of updating the mereology of a hub. Cf. Example 32 on Page 35.

41 Inserting a link, l:L, between two hubs, ha:H,hb:H require the update of the mereologies of these two existing hubs.

42 The unique identifier of the inserted link, l:L, is li, li=**uid**_L(l) and h is either ha or hb;

43 li is joined to the mereology of both ha or hb; and respective hubs are updated accordingly.

```
    value
41. update_hub_mereology: H → LI → H
42. update_hub_mereology(h)(li) ≡
43.     let m = {li} ∪ obs_mereo_H(h) in upd_mereology(h)(m) end ■
```

Formulation of Mereologies 168

The `observe_mereology` domain descriptor, Page 35, may give the impression that the mereo type MT can be described "at the point of issue" of the `observe_mereology` prompt. Since the MT type expression may, in general, depend on any part sort the mereo type MT can, for some domains, "first" be described when all part sorts have been dealt with. In [40] we we present a model of one form of evaluation of the `TripTych` analysis and description prompts.

1.2.9 Part Attributes 169

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

Inseparability of Attributes from Endurants

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[16], or seen[17], but can be objectively measured[18]. Thus, in our quest for describing domains where humans play an active

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

[17] One cannot see electric current, and one may touch an electric wire, but only if it conducts reasonably high voltage can one feel it.

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

rôle, we rule out subjective "attributes": feelings, sentiments, moods. Thus we shall abstain, in our domain science also from matters of aesthetics. We learned from Sect. 1.1.8 that 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) !

Attribute Quality and Attribute Value

170

We distinguish between an attribute, as a logical proposition and an attribute value as a value in some not necessarily Boolean value space.

Example 35 . **Attribute Propositions and Other Values**: A particular street segment (i.e., a link), say ℓ, satisfies the proposition (attribute) has_length, and may then have value length 90 meter for that attribute. Another link satisfies the same proposition but has another value; and yet another link satisfies the same proposition and may have the same value. That is: all links satisfies has_length and has some value for that attribute. A particular road transport domain, δ, has three immediate sub-parts: net, n, fleet, f, and monitor m; typically nets `has_net_name` and `has_net_owner` proposition attributes with, for example, `US Interstate Highway System` respectively `US Department of Transportation` as values for those attributes There may be other components of the net value n ■

Endurant Attributes: Types and Functions

171

Let us recall that attributes cover qualities other than unique identifiers and mereology. Let us then consider that parts have one or more attributes. These attributes are qualities which help characterise "what it means" to be a part.

172

Example 36 . **Atomic Part Attributes**: Examples of attributes of atomic parts such as a human are: *name, gender, birth-date, birth-place, nationality, height, weight, eye colour, hair colour,* etc. Examples of attributes of transport net links are: *length, location, 1 or 2-way link, link condition*, etc. ■

173

Example 37 . **Composite Part Attributes**: Examples of attributes of composite parts such as a road net are: *owner, public or private net, free-way or toll road, a map of the net,* etc. Examples of attributes of a group of people could be: *statistic distributions of gender, age, income, education, nationality, religion,* etc. ■

174

We now assume that all parts have attributes. The question is now, in general, how many and, particularly, which.

Analysis Prompt 14 . ***attribute_names:*** *The* ***domain analysis prompt*** `attribute_names` *when applied to a part p yields the set of names of its attribute types:*

- `attribute_names(p)`: $\{\eta A_1, \eta A_2, ..., \eta A_n\}$.

η *is a type operator. Applied to a type A it yields is name*[19] ■

175

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

176

The Attribute Value Observer

The "built-in" description language operator

- **attr_**A

177

applies to parts, p:P, where ηA∈`attribute_names`(p). It yields the value of attribute A of p.

[19] Normally, in non-formula texts, type A is referred to by ηA. In formulas A denote a type, that is, a set of entities. Hence, when we wish to emphasize that we speak of the name of that type we use ηA. But often we omit the distinction

Domain Description Prompt 5 . ***observe_attributes:*** *The domain analyser experiments, thinks and reflects about part attributes. That process is initated by the* ***domain description prompt****:*

- `observe_attributes`.

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

178

5. `observe_attributes` schema

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

Formalisation:
type
[t] A_i $[1 \leq i \leq n]$
value
[o] **attr_**A_i:P→A_i $[1 \leq i \leq n]$
[i] **is_**A_i:A_i→**Bool** $[1 \leq i \leq n]$
proof obligation [Disjointness of Attribute Types]
[p] ∀ δ:Δ
[p] **let** P be any part sort **in** [the Δ domain description]
[p] **let** a:(A_1|A_2|...|A_n) **in** **is_**A_i(a) ≠ **is_**A_j(a) **end end** $[i \neq j, 1 \leq i,j \leq n]$

179

The **type** *(or rather sort) definitions: A_1, A_2, ..., A_n inform us that the domain analyser has decided to focus on the distinctly named A_1, A_2, ..., A_n attributes.*[20] *And the* **value** *clauses* ***attr_***A_1*:P→*A_1, ***attr_***A_2*:P→*A_2, ..., ***attr_***A_n*:P→*A_n *are then "automatically" given: if a part (type P) has an attribute A_i then there is postulated, "by definition" [eureka] an attribute observer function* ***attr_***A_i*:P→*A_i *etcetera* ■

180

The fact that, for example, A_1, A_2, ..., A_n are attributes of p:P, means that the propositions

- `has_attribute_`A_1(p), `has_attribute_`A_2(p), ..., and `has_attribute_`A_n(p)

holds. Thus the observer functions **attr_**A_1, **attr_**A_2, ..., **attr_**A_n can be applied to p in P and yield attribute values a_1:A_1, a_2:A_2, ..., a_n:A_n respectively. 181

Example 38 . **Road Hub Attributes**: After some analysis a domain analyser may arrive at some interesting hub attributes:

44 hub state: from which links (by reference) can one reach which links (by reference),

45 hub state space: the set of all potential hub states that a hub may attain,

46 such that

   a the links referred to in the state are links of the hub mereology

   b and the state is in the state space.

47 Etcetera — i.e., there are other attributes not mentioned here.

182

[20] The attribute type names are not like type names of, for example, a programming language. Instead they are chosen by the domain analyser to reflect on domain phenomena. Cf. Example 36 on the preceding page and Example 37.

type
44. HΣ = (LI×LI)-**set**
45. HΩ = HΣ-**set**
value
44. **attr**_HΣ:H→HΣ
45. **attr**_HΩ:H→HΩ
axiom [Well−formedness of Hub States, HΣ]
46. ∀ h:H • **let** lis = **obs_mereo**_H(h) **in**
46. **let** hσ = **attr**_HΣ(h) **in**
46a. {li,li′|li,li′:LI•(li,li′)∈ hσ}⊆lis
46b. ∧ hσ ∈ **attr**_HΩ(h)
46. **end end**
type
47. ..., ...
value
47. **attr**_..., ... ■

Attribute Categories

183

One can suggest a hierarchy of part attribute categories: static or dynamic values — and within the dynamic value category: inert values or reactive values or active values — and within the dynamic active value category: autonomous values or biddable values or programmable values. We now review these attribute value types. (The review is based on [101, M.A. Jackson].)

Part attributes are either constant or varying, i.e., **static** or **dynamic** attributes. By a **static attribute**, `is_static_attribute`, we shall understand an attribute whose values are constants, i.e., cannot change. By a **dynamic attribute**, `is_dynamic_attribute`, we shall understand an attribute whose values are variable, i.e., can change.

Dynamic attributes are either inert, reactive or active attributes. By an **inert attribute**, `is_inert_attribute`, we shall understand a dynamic attribute whose values only change as the result of external stimuli where these stimuli prescribe properties of these new values. By a **reactive attribute**, `is_reactive_attribute`, we shall understand a dynamic attribute whose values, if they vary, change value in response to the change of other attribute values. By an **active attribute**, `is_active_attribute`, we shall understand a dynamic attribute whose values change (also) of its own volition.

Example 39 . **Inert and Reactive Attributes**: Buses (i.e., vehicles) have a *timetable* attribute which is dynamic, i.e., can change, namely when the operator of the bus decides so, thus the bus timetable attribute is inert. Pipeline valve units include the two attributes of *valve opening* (`open, close`) and *internal flow* (measured, say `gallons` per `second`). The valve opening attribute is of the programmable attribute category. The flow attribute is reactive (flow changes with valve opening/closing) ■

Active attributes are either autonomous, biddable or programmable attributes. By an **autonomous attribute**, `is_autonomous_attribute`, we shall understand a dynamic active attribute whose values change value only "on their own volition". The values of an autonomous attributes are a "law onto themselves and their surroundings". By a **biddable attribute**, `is_biddable_attribute`, (of a part) we shall understand a dynamic active attribute whose values may be subject to a contract as to which values it is expected to exhibit. By a **programmable attribute**, `is_programmable_attribute`, we shall understand a dynamic active attribute whose values can be accurately prescribed.

Example 40 . **Static, Programmable and Inert Link Attributes**:

48 Some link attributes

a (link) length,

b (link) name, e.g., *Fifth Ave. between 50th and 51st Streets*),

can be considered static,

49 whereas other link attributes

a (link) state (*one-way: say from 51st to 50th*),

b (link) state space (one single state, one-way: say from 51st to 50th)

can be considered programmable,

50 Finally link attributes

a link state–of–repair,

b date last maintained,

can be considered inert.

189

```
type
48a. LEN
value
48a. obs_part_LEN: L → LEN
type
48b. Name
value
48b. obs_part_Name: L → Name
type
49a. LΣ′=(HI×HI)-set
49a. LΣ={|lσ:LΣ • card lσ ≤ 2|}
value
49a. obs_part_LΣ: L → LΣ
type
49b. LΩ′=LΣ-set
49b. LΩ={|lω:LΩ • card lω = 1|}
value
49b. obs_part_LΩ: L → LΩ
type
50a. LSoR
50b. DLM
value
50a. obs_part_LSoR: L → LSoR
50b. obs_part_DLM: L → DLM ■
```

190

Example 41 . Autonomous and Programmable Hub Attributes: We continue Example **??**. Time progresses autonomously, Hub states are programmed (*traffic signals*): changing from red to green via yellow, in one pair of (co-linear) directions, while changing, in the same time interval, from green via yellow to red in the "perpendicular" directions ■

191

External Attributes: By an **external attribute** we shall understand

either a inert, or a reactive, or an autonomous, or a biddable

attribute ■ Thus we can define the domain analysis prompt: `is_external_attribute`, as:

```
value
  is_external_attribute: P → Bool
  is_external_attribute(p) ≡
      is_dynamic_attribute(p) ∧ ∼is_programmable_attribute(p)
  pre: is_endurant(p) ∧ is_discrete(p)
```

192

Figure 1.2 on the following page captures the attribute value ontology.

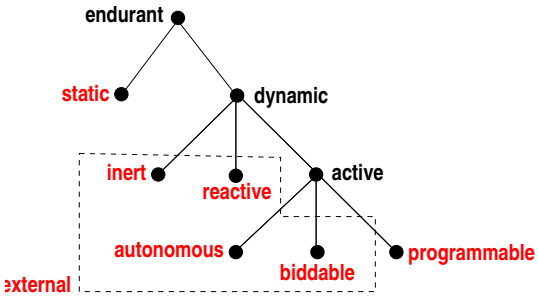

Figure 1.2: Attribute Value Ontology

Access to Attribute Values

193

In an action, event or a behaviour description static values of parts, p, (say of type A) can be "copied", **attr**_A(p), and still retain their (static) value. But, for action, event or behaviour descriptions, dynamic values of parts, p, cannot be "copied", but **attr**_A(p) must be "performed" every time they are needed. That 194
is: static values require at most one domain access, whereas dynamic values require repeated domain accesses.

We shall return to the issue of attribute value access in Sect. 1.3.7.

Shared Attributes

195

Normally part attributes of different part sorts are distinctly named. If, however, `observe_attributes`(p_{ik}:P_i) and `observe_attributes`($p_{j\ell}$:P_j), for any two distinct part sorts, P_i and P_j, of a domain, "discovers" identically named attributes, say A, then we say that parts p_i:P_i and p_j:P_j **share** attribute A. that is, that a:**attr**_A(p_i) (and a′:**attr**_A(p_j)) is a **shared attribute** (with a=a′ always (□)
196 holding).

Attribute Naming: Thus the domain describer has to exert great care when naming attribute types. If P_i and P_j are two distinct types of a domain then if and only if an attribute of P_i is to be shared with an
197 attribute of P_j must that attribute be identically named in the description of P_i and P_j.

Example 42 ***Shared Attributes.*** Examples of shared attributes: (i) Bus timetable attributes have the same value as the regional transport system timetable attribute. (ii) Bus clock attributes have the same value as the regional transport system clock attribute. (iii) Bus owner attributes have the same value as the regional transport system owner attribute. (iv) Bank customer passbooks record bank transactions on, for example, demand/deposit accounts share values with the bank general ledger passbook entries. (v) A link incident upon or emanating from a hub shares the connection between that link and the hub as an attribute. (vi) Two pipeline units[21], p_i, p_j, that are connected, such that an outlet π_j of p_i "feeds into" an inlet π_i of p_j, are said to share the connection (modeled by, e.g., $\{(\pi_i, \pi_j)\}$. ■

198

Example 43 . **Shared Timetables**: The fleet and vehicles of Example 20 on Page 28 and Example 21 on Page 30 is that of a bus company.

51 From the fleet and from the vehicles we observe unique identifiers.

52 Every bus mereology records the same one unique fleet identifier.

53 The fleet mereology records the set of all unique bus identifiers.

54 A bus timetable is a share fleet and bus attribute.

199

type
51. FI, VI, BT
value
51. **uid_**F: F → FI
51. **uid_**V: V → VI
52. **obs_mereo_**F: F → VI-**set** [cf. Sect. 1.2.8 on Page 34]
53. **obs_mereo_**V: V → FI
54. **attr_**BT: (F|V) → BT
axiom
□ ∀ f:F ⇒
∀ v:V • v ∈ **obs_part_**Vs(**obs_part_**VC(f)) • **attr_**BT(f) = **attr_**BT(v)
[which is the same as]
□ ∀ f:F ⇒
{**attr_**BT(f)}={**attr_**BT(v):v:V•v ∈ **obs_part_**Vs(**obs_part_**VC(f))} ■

200

Part attributes of one sort, P_i, may be simple type expressions such as A-**set**, where A may be an attribute of some other part sort, P_j, in which case we say that part attributes A-**set** and A are shared. 201

Example 44 . **Shared Passbooks**:

55 A banking system contains
- an administration and
- a set of customers.

56 The administration contains a general ledger.

57 An attribute of a general ledger is a set of passbooks.

58 An attribute of a customer is that of a passbook.

59 Passbooks are uniquely identified by unique customer identifiers.

202

type
55. [parts] BS, AD, GL, CS, Cs = C-**set**
58. [attributes] PB
value
55. **obs_part_**AD: BS → AD
56. **obs_part_**GL: AD → GL
57. **attr_**PBs: GL → PB-**set**
55. **obs_part_**CS: BS → CS
55. **obs_part_**Cs: BS → Cs
58. **attr_**PB: C → PB
59. **uid_**PB: PB → PBI
axiom
□ ∀ bs:BS •
attr_PBs(**attr_**GL(**obs_part_**AD(bs)))
= {**attr_**PB(c)|c:C•c ∈ **obs_part_**Cs(**obs_part_**CS(bs))} ■

[21] See upcoming Example 33 on Page 36

1.2.10 Components

203

We refer to Sect. 1.2.4 on Page 25 for a first coverage of the concept of components: definition and examples.

Components are discrete endurants which are not considered parts.

- is_component(k) ≡ is_endurant(k)∧∼is_part(k)

Example 45 . **Parts and Components**: We observe components as associated with atomic parts: The contents, that is, the collection of zero, one or more boxes, of a container is the components of the container part. Conveyor belts transport machine assembly units and are thus considered the components of the conveyor belt.

204

We now complement the observe_part_sorts (of Sect. 1.2.6). We assume, without loss of generality, that only atomic parts may contain components. Let p:P be some atomic part.

Analysis Prompt 15 . *has_components:* *The* ***domain analysis prompt****:*

- *has_components(p)*

yields **true** *if atomic part p potentially contains components otherwise false* ■

205

Let us assume that parts p:P embodies components of sorts $\{K_1,K_2,\ldots,K_n\}$. Since we cannot automatically guarantee that our domain descriptions secure that each K_i ($[1\leq i\leq n]$) denotes disjoint sets of entities we must prove it.

Domain Description Prompt 6 . *observe_component_sorts:* *The* ***domain description prompt****:*

- *observe_component_sorts(e)*

yields the component sorts and component sort observers *domain description text according to the following schema:*

206

6. observe_component_sorts schema

Narration:
[s] ... narrative text on component sorts ...
[o] ... narrative text on component sort observers ...
[i] ... narrative text on component sort recognisers ...
[p] ... narrative text on component sort proof obligations ...
Formalisation:
type
[s] K1, K2, ..., Kn
[s] KS = (K1|K2|...|Kn)**-set**
value
[o] **components**: P → KS
[i] **is**_K_i: K → **Bool** $[1\leq i\leq n]$
Proof Obligation:
[Disjointness of Component Sorts]
[p] ∀ m_i:(K_1|K_2|...|K_n) •
[p] ⋀ {**is**_K_i(m_i) ≡ ⋁∼{**is**_K_j(m_i)|j ∈ {1..m}\{i}}|i ∈ {1..m}}

The K_i are all distinct ■

207

Example 46 . **Container Components**: We continue Example 22 on Page 30.

60 When we apply `obs_component_sorts_C` to any container c:C we obtain

a a type clause stating the sorts of the various components of a container,

b a union type clause over these component sorts, and

c the component observer function signature.

type
60a K1, K2, ..., Kn
60b KS = (K1|K2|...|Kn)**-set**
value
60c **obs_comp**_KS: C → KS ■

208

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

1.2.11 Materials

209

We refer to Sect. 1.2.4 on Page 25 for a first coverage of the concept of materials.

Continuous endurants (i.e., **material**s) are entities, m, which satisfy:

- `is_material`(m) ≡ `is_endurant`(m)∧`is_continuous`(m)

Example 47 . **Parts and Materials**: We observe materials as associated with atomic parts: Thus liquid or gaseous materials are observed in pipeline units ■

We shall in this paper not cover the case of parts being immersed in materials[22]. 210

We assume, without loss of generality, that only atomic parts may contain materials. Let p:P be some atomic part.

Analysis Prompt 16 . ***has_materials:*** *The* ***domain analysis prompt****:*

- *has_materials(p)*

yields **true** *if the atomic part p:P potentially contains materials otherwise false* ■

211

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

Domain Description Prompt 7 . ***observe_material_sorts:*** *The* ***domain description prompt****:*

- *observe_material_sorts(e)*

yields the material sorts and material sort observers *domain description text according to the following schema:* 212

7. `observe_material_sorts` schema

Narration:

[22] Most such cases have the material play a minor, an abstract rôle with respect to the immersed parts. That is, we presently leave it to hydro- and aerodynamics to domain analyse those cases.

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

Formalisation:

type
[s] M_i $[1 \leq i \leq n]$
[s] MS = M1 M2 ... Mn
value
[o] **obs_mat**_M_i: P → M_i $[1 \leq i \leq n]$
[o] **materials**: P → MS
[i] **is**_M_i: M → **Bool** $[1 \leq i \leq n]$
proof obligation [Disjointness of Material Sorts]
[p] ∀ m_i:($M_1|M_2|...|M_n$) •
[p] ∧ {**is**_$M_i(m_i)$ ≡ ⋁∼{**is**_$M_j(m_i)$|j ∈ {1..m}\{i}}|i ∈ {1..m}}

The M_i are all distinct ■

213

Example 48 . **Pipeline Material**: We continue Example 27 on Page 31 and Example 33 on Page 36.

61 When we apply `obs_material_sorts_U` to any unit u:U we obtain

a a type clause stating the material sort LoG for some further undefined liquid or gaseous material, and

b a material observer function signature.

type
61a LoG
value
61b **obs_mat**_LoG: U → LoG ■

Materials-related Part Attributes

214

It seems that the "interplay" between parts and materials is an area where domain analysis in the sense of
215 this paper is relevant.

Example 49 . **Pipeline Material Flow**: We continue Examples 27, 33 and 48. Let us postulate a[n attribute] sort Flow. We now wish to examine the flow of liquid (or gaseous) material in pipeline units. We use two types

62 F for "productive" flow, and L for wasteful leak.

Flow and leak is measured, for example, in terms of volume of material per second. We then postulate the
216 following unit attributes "measured" at the point of in- or out-flow or in the interior of a unit.

63 current flow of material into a unit input connector,

64 maximum flow of material into a unit input connector while maintaining laminar flow,

65 current flow of material out of a unit output connector,

66 maximum flow of material out of a unit output connector while maintaining laminar flow,

67 current leak of material at a unit input connector,

68 maximum guaranteed leak of material at a unit input connector,

69 current leak of material at a unit input connector,

70 maximum guaranteed leak of material at a unit input connector,

71 current leak of material from "within" a unit, and

72 maximum guaranteed leak of material from "within" a unit.

```
type
62. F, L
value
63. attr_cur_iF: U → UI → F
64. attr_max_iF: U → UI → F
65. attr_cur_oF: U → UI → F
66. attr_max_oF: U → UI → F
67. attr_cur_iL: U → UI → L
68. attr_max_iL: U → UI → L
69. attr_cur_oL: U → UI → L
70. attr_max_oL: U → UI → L
71. attr_cur_L: U → L
72. attr_max_L: U → L
```

The maximum flow attributes are static attributes and are typically provided by the manufacturer as indicators of flows below which laminar flow can be expected. The current flow attributes are dynamic attributes ■

Laws of Material Flows and Leaks 218

It may be difficult or costly, or both, to ascertain flows and leaks in materials-based domains. But one can certainly speak of these concepts. This casts new light on domain modeling. That is in contrast to incorporating such notions of flows and leaks in requirements modeling where one has to show implementability.

Modeling flows and leaks is important to the modeling of materials-based domains.

Example 50 . Pipelines: Intra Unit Flow and Leak Law:

73 For every unit of a pipeline system, except the well and the sink units, the following law apply.

74 The flows into a unit equal

- a the leak at the inputs
- b plus the leak within the unit
- c plus the flows out of the unit
- d plus the leaks at the outputs.

```
axiom [Well−formedness of Pipeline Systems, PLS (1)]
73. ∀ pls:PLS,b:B\We\Si,u:U •
73.   b ∈ obs_part_Bs(pls)∧u=obs_part_U(b)⇒
73.   let (iuis,ouis) = obs_mereo_U(u) in
74.   sum_cur_iF(iuis)(u) =
74a.    sum_cur_iL(iuis)(u)                    217
74b.  ⊕ attr_cur_L(u)                          219
74c.  ⊕ sum_cur_oF(ouis)(u)                    220
74d.  ⊕ sum_cur_oL(ouis)(u)
73.   end
```

221

75 The sum_cur_iF (cf. Item 74) sums current input flows over all input connectors.

76 The sum_cur_iL (cf. Item 74a) sums current input leaks over all input connectors.

77 The sum_cur_oF (cf. Item 74c) sums current output flows over all output connectors.

78 The sum_cur_oL (cf. Item 74d) sums current output leaks over all output connectors.

75. sum_cur_iF: UI-**set** → U → F
75. sum_cur_iF(iuis)(u) ≡ ⊕ {**attr**_cur_iF(ui)(u)|ui:UI•ui ∈ iuis}
76. sum_cur_iL: UI-**set** → U → L
76. sum_cur_iL(iuis)(u) ≡ ⊕ {**attr**_cur_iL(ui)(u)|ui:UI•ui ∈ iuis}
77. sum_cur_oF: UI-**set** → U → F
77. sum_cur_oF(ouis)(u) ≡ ⊕ {**attr**_cur_iF(ui)(u)|ui:UI•ui ∈ ouis}
78. sum_cur_oL: UI-**set** → U → L
78. sum_cur_oL(ouis)(u) ≡ ⊕ {**attr**_cur_iL(ui)(u)|ui:UI•ui ∈ ouis}
    ⊕: (F|L) × (F|L) → F

222 where ⊕ is both an infix and a distributed-fix function which adds flows and or leaks ■

Example 51 . **Pipelines: Inter Unit Flow and Leak Law**:

79 For every pair of connected units of a pipeline system the following law apply:

a the flow out of a unit directed at another unit minus the leak at that output connector

b equals the flow into that other unit at the connector from the given unit plus the leak at that connector.

axiom [Well−formedness of Pipeline Systems, PLS (2)]
79. ∀ pls:PLS,b,b′:B,u,u′:U•
79. {b,b′}⊆**obs_part**_Bs(pls)∧b≠b′∧u′=**obs_part**_U(b′)
79. ∧ **let** (iuis,ouis)=**obs_mereo**_U(u),(iuis′,ouis′)=**obs_mereo**_U(u′),
79. ui=**uid**_U(u),ui′=**uid**_U(u′) **in**
79. ui ∈ iuis ∧ ui′ ∈ ouis′ ⇒
79a. **attr**_cur_oF(u′)(ui′) − **attr**_leak_oF(u′)(ui′)
79b. = **attr**_cur_iF(u)(ui) + **attr**_leak_iF(u)(ui)
79. **end**
79. **comment:** b′ precedes b ■

223 From the above two laws one can prove the **theorem:** what is pumped from the wells equals what is leaked from the systems plus what is output to the sinks. We need formalising the flow and leak summation functions.

1.2.12 "No Junk, No Confusion"

224

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

- Page 34 for *Well-formedness of Links, L, and Hubs, H*,
- Page 35 for *Well-formedness of Domain Mereologies*,
- Page 35 for *Well-formedness of Road Nets, N*,
- Page 36 for *Well-formedness of Pipeline Systems, PLS (0)*,
- Page 40 for *Well-formedness of Hub States, HΣ*,
- Page 47 for *Well-formedness of Pipeline Systems, PLS (1)*,
- Page 48 for *Well-formedness of Pipeline Systems, PLS (2)*,
- Page 49 for *Well-formedness of Pipeline Route Descriptors* and
- Page 50 for *Well-formedness of Pipeline Systems, PLS (3)*.

227

To avoid confusion we have stated a number of proof obligations:

- Page 28 for *Disjointness of Part Sorts*,
- Page 39 for *Disjointness of Attribute Types* and
- Page 46 for *Disjointness of Material Sorts*.

228

Example 52 . **No Pipeline Junk**: We continue Example 27 on Page 31 and Example 33 on Page 36.

80 We define a proper pipeline route to be a sequence of pipeline units.

a such that the i^{th} and $i{+}1^{st}$ units in sequences longer than 1 are (forward) adjacent, in the sense defined below, and

b such that the route is acyclic, in the sense also defined below.

To formalise the above we describe some auxiliary notions. 229

Pipe Routes

81 A route descriptor is the sequence of unit identifiers of the units of a route (of a pipeline system).

```
    type
80.   R′ = U^ω
80.   R = {| r:Route′•wf_Route(r) |}
81.   RD = UI^ω
    axiom [Well−formedness of Pipeline Route Descriptors, RD]
81.   ∀ rd:RD • ∃ r:R•rd=descriptor(r)
    value
81.   descriptor: R → RD
81.   descriptor(r) ≡ ⟨uid_UI(r[i])|i:Nat•1≤i≤len r⟩
```

230

82 Two units are (forward) adjacent if the output unit identifiers of one shares a unique unit identifier with the input identifiers of the other.

```
value
82.  adjacent: U × U → Bool
82.  adjacent(u,u′) ≡
82.    let (,ouis)=obs_mereo_U(u),
82.        (iuis,)=obs_mereo_U(u′) in
82.    ouis ∩ iuis ≠ {} end
```

231

83 Given a pipeline system, *pls*, one can identify the (possibly infinite) set of (possibly infinite) routes of that pipeline system.

- a The empty sequence, $\langle\rangle$, is a route of *pls*.
- b Let u be a unit of *pls*, then $\langle u\rangle$ is a route of *pls*.
- c Let u, u' be adjacent units of *pls* then $\langle u, u'\rangle$ is a route of *pls*.
- d If r and r' are routes of *pls* such that the last element of r is the same as the first element of r', then $r^\frown$**tl** r' is a route of *pls*.
- e No sequence of units is a route unless it follows from a finite number of applications of the basis and induction clauses of Items 83a–83d.

```
value
83.   Routes: PLS → R-infset
83.   Routes(pls) ≡
83a.    let rs = ⟨⟩
83b.        ∪ {⟨u⟩|u:U•u ∈ obs_part_Us(pls)}
83c.        ∪ {⟨u,u′⟩|u,u′:U•{u,u′}⊆obs_part_Us(pls) ∧ adjacent(u,u′)}
83d.        ∪ {r^tl r′|r,r′:R•{r,r′}⊆rs∧r[len r]=hd r′}
83e.    in rs end
```

232

Well-formed Routes

84 A route is acyclic if no two route positions reveal the same unique unit identifier.

```
value
84.  acyclic_Route: R → Bool
84.  acyclic_Route(r) ≡ ~∃ i,j:Nat•{i,j}⊆inds r ∧ i≠j ∧ r[i]=r[j]
```

233

Well-formed Pipeline Systems

85 A pipeline system is well-formed if

- a none of its routes are circular and
- b all of its routes are embedded in well-to-sink routes.

```
axiom [Well−formedness of Pipeline Systems, PLS (3)]
85.  ∀ pls:PLS •
85a.    non_circular(pls)
85b.  ∧ are_embedded_in_well_to_sink_Routes(pls)
value
85.  non_circular_PLS: PLS → Bool
85.  non_circular_PLS(pls) ≡
85.    ∀ r:R•r ∈ routes(p)∧acyclic_Route(r)
```

234

86 We define well-formedness in terms of well-to-sink routes, i.e., routes which start with a well unit and end with a sink unit.

```
value
86. well_to_sink_Routes: PLS → R-set
86. well_to_sink_Routes(pls) ≡
86.   let rs = Routes(pls) in
86.   {r|r:R•r ∈ rs ∧ is_We(r[1]) ∧ is_Si(r[len r])} end
```

235

87 A pipeline system is well-formed if all of its routes are embedded in well-to-sink routes.

```
87. are_embedded_in_well_to_sink_Routes: PLS → Bool
87. are_embedded_in_well_to_sink_Routes(pls) ≡
87.   let wsrs = well_to_sink_Routes(pls) in
87.   ∀ r:R • r ∈ Routes(pls) ⇒
87.     ∃ r′:R,i,j:Nat •
87.       r′ ∈ wsrs
87.       ∧ {i,j}⊆inds r′∧i≤j
87.       ∧ r = ⟨r′[k]|k:Nat•i≤k≤j⟩ end
```

236

Embedded Routes

88 For every route we can define the set of all its embedded routes.

```
value
88. embedded_Routes: R → R-set
88. embedded_Routes(r) ≡
88.   {⟨r[k]|k:Nat•i≤k≤j⟩ | i,j:Nat• i {i,j}⊆inds(r) ∧ i≤j}
```

237

A Theorem

89 The following theorem is conjectured:

a the set of all routes (of the pipeline system)

b is the set of all well-to-sink routes (of a pipeline system) and

c all their embedded routes

```
theorem:
89.  ∀ pls:PLS •
89.  let rs = Routes(pls),
89.    wsrs = well_to_sink_Routes(pls) in
89a.  rs =
89b.    wsrs ∪
89c.    ∪ {{r′|r′:R • r′ ∈ embedded_Routes(r″)} | r″:R • r″ ∈ wsrs}
88.  end ■
```

238

The above example, besides illustrating one way of coping with "junk", also illustrated the need for introducing a number of auxiliary notions: types, functions, axioms and theorems.

1.2.13 Discussion of Endurants

In Sect. 1.2.6 on Page 27 a "depth-first" search for part sorts was hinted at. It essentially expressed that we discover domains epistemologically[23] but understand them ontologically.[24] The Danish philosopher Søren Kirkegaard (1813–1855) expressed it this way: *Life is lived forwards, but is understood backwards.* The presentation of the of the ***domain analysis prompt***s and the ***domain description prompt***s results in domain descriptions which are ontological. The "depth-first" search recognizes the epistemological nature of bringing about understanding. This "depth-first" search that ends with the analysis of atomic part sorts can be guided, i.e., hastened (shortened), by postulating composite sorts that "correspond" to vernacular nouns: everyday nouns that stand for classes of endurants.

We could have chosen our ***domain analysis prompt***s and ***domain description prompt***s to reflect a "bottom-up" epistemology, one that reflected how we composed composite understandings from initially atomic parts. We leave such a collection of ***domain analysis prompt***s and ***domain description prompt***s to the reader.

1.3 Perdurant Entities

We shall give only a cursory overview of perdurants. That is, we shall not present a set of ***domain analysis prompt***s and a set of ***domain description prompt***s leading to description language, i.e., RSL texts describing perdurant entities.

The reason for giving this albeit cursory overview of perdurants is that, through this cursory overview, we can justify our detailed study of endurants, their part and subparts, their unique identifiers, mereology and attributes. This justification is manifested (i) in expressing the types of signatures, (ii) in basing behaviours on parts, (iii) in basing the for need for CSP-oriented inter-behaviour communications on shared part attributes, (iv) in indexing behaviours as are parts, i.e., on unique identifiers, and (v) in directing inter-behaviour communications across channel arrays indexed as per the mereology of the part behaviours. These are all notions related to endurants and are now justified by their use in describing perdurants.

Perdurants can perhaps best be explained in terms of a notion of state and a notion of time. We shall, in this paper, not detail notions of time, but refer to [96, 74, 52, 150].

1.3.1 States

Definition 11 . ***State:*** *By a* ***state*** *we shall understand any collection of* ***parts*** *each of which has at least one* ***dynamic attribute*** *or* ***has_components*** *or* ***has_materials*** ■

Example 53 . **States**: Some examples of states are: A road hub can be a state, cf. Hub State, HΣ, Example 38 on Page 39. A road net can be a state – since its hubs can be. Container stowage areas, CSA, Example 22 on Page 30, of container vessels and container terminal ports can be states as containers can be removed from and put on top of container stacks. Pipeline pipes can be states as they potentially carry material. Conveyor belts can be states as they potentially carry components ■

1.3.2 Actions, Events and Behaviours

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

On Action, Event and Behaviour Distinctions: The distinction into action, event and behaviour perdurants is pragmatic.

[23] **Epistemology**: the theory of knowledge, especially with regard to its methods, validity, and scope. Epistemology is the investigation of what distinguishes justified belief from opinion.

[24] **Ontology**: the branch of metaphysics dealing with the nature of being.

Time Considerations

We shall, without loss of generality, assume that actions and events are atomic and that behaviours are composite. Atomic perdurants may "occur" during some time interval, but we omit consideration of and concern for what actually goes on during such an interval. Composite perdurants can be analysed into "constituent" actions, events and "sub-behaviours". We shall also omit consideration of temporal properties of behaviours. Instead we shall refer to two seminal monographs: Specifying Systems [110, Leslie Lamport] and Duration Calculus: A Formal Approach to Real-Time Systems [160, Zhou ChaoChen and Michael Reichhardt Hansen]. For a seminal book on "time in computing" we refer to the eclectic [79, 2012]. And for seminal book on time at the epistemology level we refer to [150, 1991].

Actors

Definition 12 . ***Actor:*** *By an **actor** we shall understand something that is capable of initiating and/or carrying out actions, events or behaviours* ■

We shall, in principle, associate an actor with each part. These actors will be described as behaviours. These behaviours evolve around a state. The state is the set of qualities, in particular the dynamic attributes, of the associated parts and/or any possible components or materials of the parts.

Example 54 . **Actors**: We refer to the road transport and the pipeline systems examples of earlier. The fleet, each vehicle and the road management of the *Transportation System* of Examples 20 on Page 28 and 43 on Page 42 can be considered actors; so can the net and its links and hubs. The pipeline monitor and each pipeline unit of the *Pipeline System*, Example 27 on Page 31 and Examples 27 on Page 31 and 33 on Page 36 will be considered actors. The bank general ledger and each bank customer of the *Shared Passbooks* example, Example 44 on Page 43, will be considered actors ■

Parts, Attributes and Behaviours

Example 54 focused on what shall soon become a major relation within domains: that of parts being also considered actors, or more specifically, being also considered to be behaviours.

Example 55 . **Parts, Attributes and Behaviours**: Consider the term 'train'[25]. It has several possible "meanings". (i) the train as a part, viz., as standing on a train station platform; (ii) the train as listed in a timetable (an attribute of a transport system part), (iii) the train as a behaviour: speeding down the rail track ■

1.3.3 Discrete Actions

Definition 13 . ***Discrete Action:*** *By a **discrete action** [155] we shall understand a foreseeable thing which deliberately potentially changes a well-formed state, in one step, usually into another, still well-formed state, and for which an actor can be made responsible* ■

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

Example 56 . **Road Net Actions**: Examples of *Road Net* actions initiated by the net actor are: insertion of hubs, insertion of links, removal of hubs, removal of links, setting of hub states. Examples of *Traffic System* actions initiated by vehicle actors are: moving a vehicle along a link, stopping a vehicle, starting a vehicle, moving a vehicle from a link to a hub and moving a vehicle from a hub to a link ■

[25] This example is due to Paul Lindgreen, a Danish computer scientist. It dates from the late 1970s.

1.3.4 Discrete Events

256

Definition 14 . ***Event:*** *By an **event** we shall understand some unforeseen thing, that is, some 'not-planned-for' "action", one which surreptitiously, non-deterministically changes a well-formed state into another, but usually not a well-formed state, and for which no particular domain actor can be made responsible* ■

257

Events can be characterised by a pair of (before and after) states, a predicate over these and, optionally, a time or time interval. The notion of event continues to puzzle philosophers [70, 130, 117, 66, 91, 9, 107,
258 59, 123, 58]. We note, in particular, [66, 9, 107].

Example 57 . **Road Net and Road Traffic Events**: Some road net events are: "disappearance" of a hub or a link, failure of a hub state to change properly when so requested, and occurrence of a hub state leading traffic into "wrong-way" links. Some road traffic events are: the crashing of one or more vehicles (whatever 'crashing' means), a car moving in the wrong direction of a one-way link, and the clogging of a hub with too many vehicles ■

1.3.5 Discrete Behaviours

259

Definition 15 . ***Discrete Behaviour:*** *By a **discrete behaviour** we shall understand a set of sequences of potentially interacting sets of discrete actions, events and behaviours* ■

260

Example 58 . **Behaviours**: Examples of behaviours: Road Nets: A sequence of hub and link insertions and removals, link disappearances, etc. Road Traffic: A sequence of movements of vehicles along links, entering, circling and leaving hubs, crashing of vehicles, etc. Pipelines: A sequence of pipeline pump and valve openings and closings, and failures to do so (events), etc. Container Vessels and Ports: Concurrent sequences of movements (by cranes) of containers from vessel to port (unloading), with sequences of movements (by cranes) from port to vessel (loading), with dropping of containers by cranes, etcetera ■

Channels and Communication

261

Behaviours sometimes synchronise and usually communicate. Using the CSP [97] notation (adopted by RSL) we introduce and model behaviour communication. Communication is abstracted as the sending (ch ! m) and receipt (ch ?) of messages, m:M, over channels, ch.

type M
channel ch M

262

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

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

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

Relations Between Attribute Sharing and Channels

263

We shall now interpret the syntactic notion of attribute sharing with the semantic notion of channels. This is in line with the above-hinted interpretation of parts with behaviours, and, as we shall soon see part
264 attributes, part components and part materials with behaviour states.

Thus, for every pair of parts, p_{ik}:P_i and $p_{j\ell}$:P_j, of distinct sorts, P_i and P_j which share attribute values in A we are going to associate a channel. If there is only one pair of parts, p_{ik}:P_i and $p_{j\ell}$:P_j, of these sorts, then just a simple channel, say ch_{P_i,P_j}.

channel ch_{P_i,P_j}:A.

If there is only one part, p_i:P_i, but a definite set of parts p_{jk}:P_j, with shared attributes, then a *vector* of channels. Let $\{p_{j1}, p_{j2}, ..., p_{jn}\}$ be all the part of the domain of sort P_j. Then $uids : \{\pi_{p_{j1}}, \pi_{p_{j2}}, ..., \pi_{p_{jn}}\}$ is the set of their unique identifiers. Now a schematic channel array declaration can be suggested:

channel $\{\mathsf{ch}[\{\pi_i,\pi_j\}]|\pi_i$=**uid**_$\mathsf{P}_i(\mathsf{p}_i)\wedge\pi_j \in \mathsf{uids}\}$:A.

The above can be extended from channel matrices to channel tensors, etc., hence the term channel 'array'.

Example 59 . **Bus System Channels**: We extend Examples 20 on Page 28 and 43 on Page 42. We consider the fleet and the vehicles to be behaviours.

90 We assume some transportation system, δ. From that system we observe

91 the fleet and

92 the vehicles.

93 The fleet to vehicle channel array is indexed by the 2-element sets of the unique fleet identifier and the unique vehicle identifiers. We consider bus timetables to be the only message communicated between the fleet and the vehicle behaviours.

value
90. δ:Δ,
91. f:F = **obs_part**_F(δ),
92. vs:V-**set** = **obs_part**_Vs(**obs_part**_VC((**obs_part**_F(δ))))

channel
93. {fch[{**uid**_F(f),**uid**_V(v)}]|v:V•v ∈ vs}:BT ■

Example 60 . **Bank System Channels**: We extend Example 44 on Page 43. We consider the general ledger and the customers to be behaviours.

94 We assume some bank system. From the bank system

95 we observe the general ledger.

96 and the set of customers.

97 We consider passbooks to be the only message communicated between the general ledger and the customer behaviours.

value
94. bs:BS
95. gl=**obs_part**_GL(**obs_part**_AD(bs)):GL
96. cs=**obs_part**_Cs(**obs_part**_CS(bs)):C-**set**

channel
97. {bsch[{**uid**_GL(gl),**uid**_C(c)}]|c:C•c ∈ cs}:PB ■

1.3.6 Continuous Behaviours

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

Example 61 . **Flow in Pipelines**: We refer to Examples 33, 48, 49, 50 and 51. Let us assume that oil is the (only) material of the pipeline units. Let us assume that there is a sufficient volume of oil in the pipeline units leading up to a pump. Let us assume that the pipeline units leading from the pump (especially valves and pumps) are all open for oil flow. Whether or not that oil is flowing, if the pump is pumping (with a sufficient head[26]) then there will be oil flowing from the pump outlet into adjacent pipeline units ■

To describe the flow of material (say in pipelines) requires knowledge about a number of material attributes — not all of which have been covered in the above-mentioned examples. To express flows one resorts to the mathematics of fluid-dynamics using such second order differential equations as first derived by Bernoulli (1700–1782) and Navier–Stokes (1785–1836 and 1819–1903).

1.3.7 Attribute Value Access

We refer to paragraph "Access to Attribute Values" in Section 1.2.9 Page 42. We can distinguish between three kinds of attributes: the constant attributes which are those whose values are static; the programmable attributes which are those dynamic values are exclusively set by part processes; and the remaining dynamic attributes are here seen as individual behaviours.

Access to Static Attribute Values

The **constant attributes** can be "copied" **attr_**A(p) (and retain their values).

Access to External Attribute Values

By the **external behaviour attributes** we shall thus understand the inert, reactive, autonomous and the biddable attributes ■

98 Let ξA be the set of names, ηA, of all external behaviour attributes.

99 Let $\Pi_{\xi A}$ be the set of indexes into the external attribute channel, say attr_A_ch, one for each distinct attribute name, A, in ξA.

100 Each external behaviour attribute is seen as an individual behaviour, each "accessible" by means of a channel, attr_A_ch.

101 External attribute values are then accessed by the input, from channel attr_A_ch[π]-accessible external attribute behaviours.

102 The **type** of attr_A_ch[π] is considered to be **Unit**$\xrightarrow{\sim}$A.

```
98.  value
98.    ξA: {ηA|A is any external attribute name}
99.    Π_ξA: Π-set
100. channel
100.   {attr_A_ch[π]|π ∈ Π_ξA}
101. value
101.   attr_A_ch[π] ?
101. type
101.   attr_A_ch[π]: Unit~→A [abbrv.:𝕌A]
```

[26] The **pump head** is the linear vertical measurement of the maximum height a specific pump can deliver a liquid to the pump outlet.

We shall omit the η prefix in actual descriptions. The choice of representing external behaviour attributes as behaviours is a technical one. See Items 345c and 345a Page 162 for a use of the concept of external behaviour attribute channels. 275

Access to Programmable Attribute Values 276

The **programmable attributes** are treated as function arguments. This is a technical choice. It is motivated as follows. We find that programmable attribute values are set (i.e., updated) by part processes. That is, to each part, whether atomic or composite, we associate a behaviour. That behaviour is (to be) described as we describe functions. These functions (normally) *"go on forever"*. Therefore these functions are described basically by a "tail" recursive definition:

value f: Arg → Arg; f(a) ≡ (... **let** a′ = F(...)(a) **in** f(a′) **end**)

where $\mathscr{F}$ is some expression based on values defined within the function definition body of f and on a's "input" argument a, and where a can be seen as a programmable attribute.

1.3.8 Perdurant Signatures and Definitions 277

We shall treat perdurants as functions. In our cursory overview of perdurants we shall focus on one perdurant quality: function signatures. 278

Definition 16 . ***Function Signature:*** *By a* ***function signature*** *we shall understand a function name and a function type expression* ■

Definition 17 . ***Function Type Expression:*** *By a* ***function type expression*** *we shall understand a pair of type expressions. separated by a function type constructor either → (total function) or $\xrightarrow{\sim}$ (partial function)* ■

The type expressions are usually part sort or type, material sort or attribute type names, but may, occasionally be expressions over respective type names involving **-set**, ×, *, $\xrightarrow[m]{}$ and | type constructors.

1.3.9 Action Signatures and Definitions 279

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

action: VAL → Σ $\xrightarrow{\sim}$ Σ
action(v)(σ) **as** σ′
  pre: $\mathscr{P}$(v,σ)
  post: $\mathscr{Q}$(v,σ,σ′)

expresses that a selection of the domain as provided by the Σ type expression is acted upon and possibly changed. The partial function type operator $\xrightarrow{\sim}$ shall indicate that action(v)(σ) may not be defined for the argument, i.e., initial state σ and/or the argument v:VAL, hence the precondition $\mathscr{P}$(v,σ). The post condition $\mathscr{Q}$(v,σ,σ′) characterises the "after" state, σ′:Σ, with respect to the "before" state, σ:Σ, and possible arguments (v:VAL). 280 281

Example 62 . **Insert Hub Action Formalisation**: We formalise aspects of the above-mentioned hub and link actions:

103 Insertion of a hub requires

104 that no hub exists in the net with the unique identifier of the inserted hub,

105 and then results in an updated net with that hub.

```
value
103. insert_H: H → N ~→ N
103. insert_H(h)(n) as n′
104.    pre: ∼∃ h′:H•h′ ∈ obs_part_Hs(obs_part_HS(n))•uid_H(h)=uid_H(h′)
105.    post: obs_part_Hs(obs_part_HS(n′))=obs_part_Hs(obs_part_HS(n))∪{h} ■
```

Which could be the argument values, V:VAL, of actions ? Well, there can basically be only two kinds of argument values: parts, components and materials, respectively unique part identifiers, mereologies and attribute values. It basically has to be so since there are no other kinds of values in domains. There can be exceptions to the above (Booleans, natural numbers), but they are rare !

Perdurant (action) analysis thus proceeds as follows: identifying relevant actions, assigning names to these, delineating the "smallest" relevant state[27], ascribing signatures to action functions, and determining action pre-conditions and action post-conditions. Of these, ascribing signatures is, perhaps, the most crucial: In the process of determining the action signature one oftentimes discovers that part or material attributes have been left "undiscovered".

Example 63 shows examples of signatures whose arguments are either parts, or parts and unique identifiers, or parts and unique identifiers and attributes.

Example 63 . Some Function Signatures: Inserting a link between two identified hubs in a net:

```
value insert_L: L × (HI × HI) → N ~→ N
```

Removing a hub and removing a link:

```
value remove_H: HI → N ~→ N
      remove_L: LI → N ~→ N
```

Changing a hub state.

```
value change_HΣ: HI × HΣ → N ~→ N ■
```

1.3.10 Event Signatures and Definitions

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

```
value
  event: Σ × Σ → Bool
  event(σ,σ′) as true⌈⌉false
      pre:  P(σ)
      post: Q(σ,σ′)
```

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

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

Example 64 . Link Disappearence Formalisation: We formalise aspects of the above-mentioned link disappearance event:

106 The result net is not well-formed.

107 For a link to disappear there must be at least one link in the net;

108 and such a link may disappear such that

109 it together with the resulting net makes up for the "original" net.

```
value
106.  link_diss_event: N × N′ × Bool
106.  link_diss_event(n,n′) as tf
107.    pre: obs_part_Ls(obs_part_LS(n))≠{}
108.    post: ∃ l:L•l ∈ obs_part_Ls(obs_part_LS(n)) ⇒
109.          l ∉ obs_part_Ls(obs_part_LS(n′))
109.          ∧ n′ ∪ {l} = obs_part_Ls(obs_part_LS(n)) ■
```

1.3.11 Discrete Behaviour Signatures and Definitions 289

We shall only cover behaviour signatures when expressed in RSL/CSP [85]. The behaviour functions are now called processes. That a behaviour function is a never-ending function, i.e., a process, is "revealed" in the function signature by the "trailing" Unit:

behaviour: ... → ... **Unit**

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

behaviour: **Unit** → ...

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

behaviour: ... → **in** ch ...

290

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

behaviour: ... → **out** ch ...

That a process accepts other arguments is "revealed" in the function signature as follows:

behaviour: ARG → ...

where ARG can be any type expression:

T, T→T, T→T→T, etcetera

291

As shown in [39] we can, without loss of generality, associate with each part a behaviour; parts which share attributes and are therefore referred to in some parts' mereology, can communicate (their "sharing") via channels. The process evolves around a state: its unique identity, $\pi : \Pi$,, its possibly changing mereology, mt:MT[28], the possible components and materials of the part[29], and the constant, the external and the programmable attributes of the part. A behaviour signature is therefore: 292

behaviour: π:Π × me:MT × sa:SA × ea:EA → pa:PA → **out** ochs **in** ichns **Unit**

where (i) π:Π is the unique identifier of part P, i.e., π=**uid**_P(p), (ii) me:ME is the mereology of part P, me = **obs_mereo**_P(p), (iii) sa:SA lists the static attribute values of the part behaviour, (iv) ea:EA lists the external attribute channels of the part behaviour, (v) ps:PA lists the programmable attribute values of the part behaviour, and where (vi) ochs and ichns refer to the shared attributes of the behaviours.

We focus, for a little while, on the expression of sa:SA, ea:EA and pa:PA, that is, on the concrete types of SA, EA and PA.

$\mathscr{S}_{\mathscr{A}}$: SA simply lists the static value types: $svT_1, svT_2, ..., svT_s$ where s is the number of static attributes of parts p:P.

$\mathscr{E}_{\mathscr{A}}$ EA simply lists the channel indexes to the external attribute values: $((eA_1, \pi_{eA_1}), (eA_2, \pi_{eA_2}), ..., (eA_x, \pi_{eA_x}))$[30] where x is the number, 0 or more, of external attributes of parts p:P.

$\mathscr{P}_{\mathscr{A}}$ PA simply lists appropriate programmable value expression type: $(pvT_1, pvT_2, ..., pvT_q)$ where q is the number of programmable attributes of parts p:P

Let P be a composite sort defined in terms of sub-sorts PA, PB, ..., PC. The process compiled from cp:P, is composed from a process, $\mathscr{M}_{cP_{\text{CORE}}}$, relying on and handling the unique identifier, mereology and attributes of process p as defined by P operating in parallel with processes $p_a, p_b, \ldots, p_c$ where p_a is "derived" from PA, p_b is "derived" from PB, ..., and p_c is "derived" from PC. The domain description "compilation" schematic below "formalises" the above.

Process Schema I: Abstract `is_composite(p)`

```
value
    compile_process: P → RSL-Text
    compile_process(p) ≡
            M_cP_CORE(uid_P(p),obs_mereo_P(p),S_A(p),E_A(p))(P_A(p))
          ‖ compile_process(obs_part_PA(p))
          ‖ compile_process(obs_part_PB(p))
          ‖ ...
          ‖ compile_process(obs_part_PC(p))
```

The text macros: $\mathscr{S}_{\mathscr{A}}$, $\mathscr{E}_{\mathscr{A}}$ and $\mathscr{P}_{\mathscr{A}}$ were informally explained above. Part sorts PA, PB, ..., PC are obtained from the `observe_part_sorts` prompt, Page 28.

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

Process Schema II: Concrete `is_composite(p)`

```
type
    Qs = Q-set
value
    qs:Q-set = obs_part_Qs(p)
    compile_process: P → RSL-Text
    compile_process(p) ≡
            M_cP_CORE(uid_P(p),obs_mereo_P(p),S_A(p),E_A(p))(P_A(p))
        ‖ ‖{compile_process(q)|q:Q•q ∈ qs}
```

[28] For MT see footnote 15 on Page 36.

[29] — we shall neither treat components nor materials further in this document

[30] See paragraph *Access to External Attribute Values* on Page 56.

Process Schema III: `is_atomic(p)`

```
value
    compile_process: P → RSL-Text
    compile_process(p) ≡
        𝓜aP_CORE(uid_P(p),obs_mereo_P(p),𝓢𝒜(p),ℰ𝒜(p))(𝒫𝒜(p))
```

298

Example 65 . Bus Timetable Coordination: We refer to Examples 20 on Page 28, 21 on Page 30, 43 on Page 42 and 59 on Page 55.

110 δ is the transportation system; f is the fleet part of that system; vs is the set of vehicles of the fleet; bt is the shared bus timetable of the fleet and the vehicles.

111 The fleet process is compiled as per Process Schema II (Page 60)

299

```
    type
        Δ, F, VC       [Example 20 on Page 28]
        V, Vs=V-set    [Example 21 on Page 30]
        FI, VI, BT     [Example 43 on Page 42]
    channel
        {fch...}       [Example 59 on Page 55]
    value
110.    δ:Δ,
110.    f:F = obs_part_F(δ),
110.    vs:V-set = obs_part_Vs(obs_part_VC(f)),
110.    bt:BT = attr_BT(f)
    axiom
110.    ∀ v:V•v ∈ vs ⇒ bt = attr_BT(v) [Example 43 on Page 42]
    value
111.    fleet: fi:FI×BT → in,out {fch[{fi,uid_V(v)}]|v:V•v ∈ vs} process
111.    fleet(fi,bt) ≡
111.      𝓜F(fi,bt)
111.      ‖ ‖ {vehicle(uid_V(v),fi:FI,bt)|v:V•v ∈ vs}
111.    vehicle: vi:VI×fi:FI×bt:BT → in,out fch[{fi,vi}] process
111.    vehicle(vi,fi,bt) ≡ 𝓜V(vi,fi,bt)
```

300

Fleet and vehicle processes $\mathscr{M}_F$ and $\mathscr{M}_V$ are both "never-ending" processes:

```
    value
        𝓜F: fi:FI×bt:BT → in,out {fch[{fi,uid_V(v)}]|v:V•v ∈ vs} process
        𝓜F(fi,bt) ≡ let bt′ = ℱ(fi,bt) in 𝓜F(fi,bt′) end

        𝓜V: vi:VI×fi:FI×bt:BT → in,out fch[{fi,vi}] process
        𝓜V(vi,fi,bt) ≡ let bt′ = 𝒱(vi,bt) in 𝓜V(vi,fi,bt′) end
```

The "core" processes, $\mathscr{F}$ and $\mathscr{V}$, are simple actions. In this example we simplify them to change only bus timetables. The expression of actual synchronisation and communication between the fleet and the vehicle processes are contained in $\mathscr{F}$ and $\mathscr{V}$. 301

value
$\mathscr{F}$: fi:FI×bt:BT → **in**,**out** {fch[{fi,**uid**_V(v)|v:V•v ∈ vs}]} BT
$\mathscr{F}$(fi,bt) ≡ ...

$\mathscr{V}$: vi:VI×fi:FI×bt:BT → **in**,**out** fch[{fi,vi}] BT
$\mathscr{V}$(vi,fi,bt) ≡ ...

What the synchronisation and communication between the fleet and the vehicle processes consists of we leave to the reader ! ■

Process Schema IV: Core Process (I)

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

$\mathscr{M}_{cP_{\text{CORE}}}$: π:Π×me:MT×sa:SA×ea:EA→pa:PA→**in** inchs **out** ochs **Unit**
$\mathscr{M}_{cP_{\text{CORE}}}$(π,me,sa,ea)(pa) ≡
  let (me′,pa′) = $\mathscr{F}$(π,me,sa,ea)(pa) **in**
  $\mathscr{M}_{cP_{\text{CORE}}}$(π,me′,sa,ea)(pa′) **end**

$\mathscr{F}$: π:Π×me:MT×sa:SA×ea:EA→PA→**in** inchs **out** ochs → MT×PA

$\mathscr{F}$ potentially communicates with all those part processes (of the whole domain) with which it shares attributes, that is, has connectors. $\mathscr{F}$ is expected to contain input/output clauses referencing the channels of the in ... out ... part of their signatures. These clauses enable the sharing of attributes. $\mathscr{F}$ also contains expressions, attr_ch[(A,π)] ?, to external attributes. An example of the update of programmable attributes is shown in the vehicle definitions in Sect. 6.2.3, Pages 148 and 149.

The $\mathscr{F}$ action non-deterministically internal choice chooses between

- either [1,2,3,4]
  - [1] accepting input from
  - [4] another part process,
  - [2] then optionally offering a reply to that other process, and
  - [3] finally delivering an updated state;
- or [5,6,7,8] offering
  - [5] an output,
  - [6] val,
  - [8] to another part process,
  - [7] and then delivering an updated state;
- or [9] doing own work resulting in an updated state.

Process Schema V: Core Process (II)

value
  $\mathscr{F}$: π:Π → me:MT → sa:SA × ea:EA → pa:PA → **in**,**out** $\mathscr{E}$(π,me) MT × PA
  $\mathscr{F}$(π,me,sa,ea)(pa) ≡
[1] [] { **let** val = ch[π′] ? **in**

```
[2]          ch[π'] ! in_reply(sa,ea,pa)(val) ;
[3]          in_update(me,sa,ea,pa)(π',sa,ea,pa) end
[4]        | π' ∈ 𝒪(π,me)}
[5]   ∏ [] { let (π',val) = await_reply(me,sa,ea,pa) in
[6]          ch[π'] ! out_reply(val,sa,ea,pa) ;
[7]          out_update(me,sa,ea,pa) end
[8]        | π' ∈ 𝒪(π,me)}
[9]   ∏      (me,own_work(sa,ea,pa))

        in_reply: SA×EA×PA × VAL → VAL
        in_update: (MT×SA×EA×PA) → (MT×PA)
        await_reply: (MT×SA×EA×PA) → Π×VAL
        out_reply: (SA×EA×PA×VAL) → VAL
        out_update: (MT×SA×EA×PA) → (MT×PA)
        own_work: SA×EA×PA → (MT×PA)
```

We leave these auxiliary functions and VAL undefined.

1.3.12 Concurrency: Communication and Synchronisation 306

Process Schemas I, II and IV (Pages 60, 60 and 62), reveal that two or more parts, which temporally coexist (i.e., at the same time), imply a notion of concurrency. Process Schema IV, through the RSL/CSP language expressions ch ! v and ch ?, indicates the notions of communication and synchronisation. Other than this we shall not cover these crucial notion related to parallelism.

1.3.13 Summary and Discussion of Perdurants 307

The most significant contribution of Sect. 1.3 has been to show that for every domain description there exists a normal form behaviour — here expressed in terms of a CSP process expression.

Summary 308

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

Discussion 309

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

1.4 Closing

1.4.1 Analysis & Description Calculi for Other Domains

The analysis and description calculus of this paper appears suitable for manifest domains. For other domains other calculi appears necessary. There is the introvert, composite domain of systems software: operating systems, compilers, database management systems, Internet-related software, etcetera. The classical

computer science and software engineering disciplines related to these components of systems software appears to have provided the necessary analysis and description "calculi." There is the domain of financial systems software accounting & bookkeeping, banking systems, insurance, financial instruments handling (stocks, etc.), etcetera. We refer to Sect. 9.1.2 on Page 186 [Item 8]. Etcetera. For each domain characterisable by a distinct set of analysis & description calculus prompts such calculi must be identified.

It seems straightforward: to base a method for analysing & describing a category of domains on the idea of prompts like those developed in this paper.

1.4.2 On Domain Description Languages

We have in this paper expressed the domain descriptions in the RAISE [86] specification language RSL [85]. With what is thought of as basically inessential, editorial changes, one can reformulate these domain description texts in either of Alloy [100] or The B-Method [1] or VDM [48, 49, 77] or Z [157]. One could also express domain descriptions algebraically, for example in CafeOBJ [81, 68, 80, 56]. The analysis and the description prompts remain the same. The description prompts now lead to CafeOBJ texts.

We did not go into much detail with respect to perdurants, let alone behaviours. For all the very many domain descriptions, covered elsewhere, RSL (with its CSP sub-language) suffices. But there are cases where we have conjoined our RSL domain descriptions with descriptions in Petri Nets [132] or MSC [99] (Message Sequence Charts) or StateCharts [92]. Since this paper only focused on endurants there was no need, it appears, to get involved in temporal issues. When that becomes necessary, in a study or description of perdurants, then we either deploy DC: The Duration Calculus [160] or TLA+: Temporal Logic of Actions [110].

1.4.3 Comparison to Other Work

Background: The TripTych Domain Ontology

Sections 1.2–1.3 outlined the TripTych modeling approach to domain entities. We shall now compare that approach to a number of techniques and tools (12 in all) that are somehow related — if only by the term 'domain' ! Common to all the 12 "other" approaches is that none of them present a prompt calculus that help the domain analyser elicit a, or the, domain description. But before these comparisons let us put it in the context of the ontology thinking of philosophers. The seminal reference here is [145, John Sowa; Chap. 2: Ontology]. The crucial concept here is that of ontological commitments. From Aristotle via Kant, Hegel, Peirce, Husserl, Whitehead and Heidegger to Quine, there has been an abundance of proposals for the structuring of the ontological commitments into categories. These proposals all have in common that they are concerned with *"all there exists in the world"*. We are only interested in the manifest, that is, physical world and in what can be described in that world. Figure 1.3 on the facing page shows the tree-like structuring of what modern day AI researchers cum ontologists would call *an upper ontology*.

General

Two approaches to structuring domain understand will be reviewed.

1: Ontology Science & Engineering: Ontologies are *"formal representations of a set of concepts within a domain and the relationships between those concepts"* — expressed usually in some logic. Ontology engineering [10] construct ontologies. Ontology science appears to mainly study structures of ontologies, especially so-called upper ontology structures, and these studies "waver" between philosophy and information science[31]. Internet published ontologies usually consists of thousands of

[31] We take the liberty of regarding information science as part of computer science, cf. Sect. 1.1.3 on Page 17.

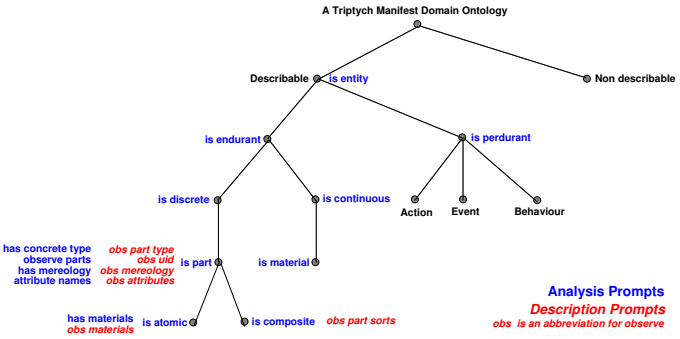

Figure 1.3: The Upper Ontology of TripTych Manifest Domains

logical expressions. These are represented in some, for example, low-level mechanisable form so that they can be interchanged between ontology research groups and processed by various tools. There does not seem to be a concern for "deriving" such ontologies into requirements for software. Usually ontology presentations either start with the presentation of, or makes reference to its reliance on, an upper ontology. The term 'ontology' has been much used in connection with automating the design of various aspects WWW applications [153]. Description Logic ([6]) has been proposed as a language for the Semantic Web [7].

The interplay between endurants and perdurants is studied in [13, *Endurants and perdurants in directly depicting ontologies*]. That study investigates axiom systems for two ontologies. One for endurants (SPAN), another for perdurants (SNAP). No examples of descriptions of specific domains are, however, given, and thus no specific techniques nor tools are given, method components which could help the engineer in constructing specific domain descriptions. This paper is therefore only relevant to the current paper insofar as it justifies our emphasis on endurant versus perdurant entities. The [13, Endurants and perdurants in directly depicting ontologies] paper is an information (i.e., domain) cum computer science paper (cf. Sect. 1.1.3 on Page 17). For more on SPAN and SNAP see [14].

The interplay between endurant and perdurant entities and their qualities is studied in [105, *Qualities, Quantities, and the Endurant-Perdurant Distinction in Top-Level Ontologies*]. In our study the term quality is made specific and covers the ideas of external and internal qualities, cf. Sect. 1.2.6 on Page 32. External qualities focus on whether endurant or perdurant, whether part, component or material, whether action, event or behaviour, whether atomic or composite part, etcetera. Internal qualities focus on unique identifiers (of parts), the mereology (of parts), and the attributes (of parts, components and materials), that is, of endurants. In [105, Johansson] the relationship between universals (types), particulars (values of types) and qualities is not "restricted" as in the TripTych domain analysis, but is axiomatically interwoven in an almost "recursive" manner. Values [of types ('quantities' [of 'qualities'])] are, for example, seen as subordinated types; this is an ontological distinction that we do not make. The concern of [105, Johansson] is also the relations between qualities and both endurant and perdurant entities, where we have yet to focus on "qualities", other than signatures, of perdurants. [105, Johansson] investigates the quality/quantity issue wrt. endurance/perdurance and poses the questions: [b] are non-persisting quality instances enduring, perduring or neither ? and [c] are persisting quality instances enduring, perduring or neither ? and arrives, after some analysis of the endurance/perdurance concepts, at the answers: [b′] non-persisting quality instances are neither enduring nor perduring particulars (i.e., entities), and [c′] persisting quality instances are enduring particulars. Answer [b′] justifies our separating enduring and perduring entities into two disjoint, but jointly "exhaustive" ontologies. The more general study of [105, Johansson] is therefore really not relevant to our prompt calculi, in which we do not speculate on more abstract, conceptual qualities, but settle on external endurant qualities, on the unique identifier, mereology and attribute qualities of endurants, and

the simple relations between endurants and perdurants outlined in Sect. 1.3, specifically in the relations between signatures of actions, events and behaviours and the endurant sorts (Sects. 1.3.8–1.3.11), and especially the relation between parts and behaviours as outlined in Sect. 1.3.11. That is, the TripTych approach to ontology, i.e., its domain concept, is not only model-theoretic, but, we risk to say, radically different.

The concerns of TripTych domain science & engineering is based on that of algorithmic engineering. The domains to which we are applying our analysis & description tools and techniques are spatio-temporal, that is, can be observed, physically; this is in contrast to such conceptual domains as various branches of mathematics, physics, biology, etcetera. Domain science & engineering is not aimed at letting the computer solve problems based on the knowledge it may have stored. Instead it builds models based on knowledge of, but not "in" the domain. The TripTych form of domain science & engineering differs from conventional ontological engineering in the following, essential ways: The TripTych domain descriptions rely essentially on a "built-in" upper ontology: types, abstract as well as model-oriented (i.e., concrete) and actions, events and behaviours. Domain science & engineering is not, to a first degree, concerned with modalities, and hence do not focus on the modeling of knowledge and belief, necessity and possibility, i.e., alethic modalities, epistemic modality (certainty), promise and obligation (deontic modalities), etcetera.

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

The next section overlaps with the present section.

2: Knowledge Engineering: The concept of knowledge has occupied philosophers since Plato. No common agreement on what 'knowledge' is has been reached. From [115, 5, 119, 147] we may learn that *knowledge is a familiarity with someone or something; it can include facts, information, descriptions, or skills acquired through experience or education; it can refer to the theoretical or practical understanding of a subject; knowledge is produced by socio-cognitive aggregates (mainly humans) and is structured according to our understanding of how human reasoning and logic works.* The seminal reference here is [72]. The aim of knowledge engineering was formulated, in 1983, by an originator of the concept, Edward A. Feigenbaum [75]: knowledge engineering is an engineering discipline that involves integrating knowledge into computer systems in order to solve complex problems normally requiring a high level of human expertise. Knowledge engineering focus on continually building up (acquire) large, shared data bases (i.e., knowledge bases), their continued maintenance, testing the validity of the stored 'knowledge', continued experiments with respect to knowledge representation, etcetera. Knowledge engineering can, perhaps, best be understood in contrast to algorithmic engineering: In the latter we seek more-or-less conventional, usually imperative programming language expressions of algorithms whose algorithmic structure embodies the knowledge required to solve the problem being solved by the algorithm. The former seeks to solve problems based on an interpreter inferring possible solutions from logical data. This logical data has three parts: a collection that "mimics" the semantics of, say, the imperative programming language, a collection that formulates the problem, and a collection that constitutes the knowledge particular to the problem. We refer to [50].

Domain science & engineering is not aimed at letting the computer solve problems based on the knowledge it may have stored. Instead it builds models based on knowledge of the domain.

Finally, the domains to which we are applying 'our form of' domain analysis are domains which focus on spatio-temporal phenomena. That is, domains which have concrete renditions: air traffic, banks, container lines, manufacturing, pipelines, railways, road transport, stock exchanges, etcetera. In contrast one may claim that the domains described in classical ontologies and knowledge representations are mostly conceptual: mathematics, physics, biology, etcetera.

Specific

3: Database Analysis: There are different, however related "schools of database analysis". DSD: the Bachman (or data structure) diagram model [8]; RDM: the relational data model [62]; and ER: entity set relationshp model [60] "schools". DSD and ER aim at graphically specifying database structures. Codd's RDM simplifies the data models of DSD and ER while offering two kinds of languages with which to operate on RDM databases: SQL and Relational Algebra. All three "schools" are focused more on data modeling for databases than on domain modeling both endurant and perdurant entities.

4: Domain Analysis: Domain analysis, or product line analysis (see below), as it was then conceived in the early 1980s by James Neighbors [121], is the analysis of related software systems in a domain to find their common and variable parts. It is a model of wider business context for the system. This form of domain analysis turns matters "upside-down": it is the set of software "systems" (or packages) that is subject to some form of inquiry, albeit having some domain in mind, in order to find common features of the software that can be said to represent a named domain.

In this section we shall mainly be comparing the TripTych approach to domain analysis to that of Reubén Prieto-Dĩaz's approach [126, 127, 128]. Firstly, our understanding of domain analysis basically coincides with Prieto-Dĩaz's. Secondly, in, for example, [126], Prieto-Dĩaz's domain analysis is focused on the very important stages that precede the kind of domain modeling that we have described: major concerns are selection of what appears to be similar, but specific entities, identification of common features, abstraction of entities and classification. Selection and identification is assumed in our approach, but we suggest to follow the ideas of Prieto-Dĩaz. Abstraction (from values to types and signatures) and classification into parts, materials, actions, events and behaviours is what we have focused on. All-in-all we find Prieto-Dĩaz's work very relevant to our work: relating to it by providing guidance to pre-modeling steps, thereby emphasising issues that are necessarily informal, yet difficult to get started on by most software engineers. Where we might differ is on the following: although Prieto-Dĩaz does mention a need for domain specific languages, he does not show examples of domain descriptions in such DSLs. We, of course, basically use mathematics as the DSL. In our approach we do not consider requirements, let alone software components, as do Prieto-Dĩaz, but we find that that is not an important issue.

5: Domain Specific Languages: Martin Fowler[32] defines a *Domain-specific language* (DSL) *as a computer programming language of limited expressiveness focused on a particular domain* [78]. Other references are [118, 146]. Common to [146, 118, 78] is that they define a domain in terms of classes of software packages; that they never really "derive" the DSL from a description of the domain; and that they certainly do not describe the domain in terms of that DSL, for example, by formalising the DSL.

6: Feature-oriented Domain Analysis (FODA): Feature oriented domain analysis (FODA) is a domain analysis method which introduced feature modeling to domain engineering. FODA was developed in 1990 following several U.S. Government research projects. Its concepts have been regarded as critically advancing software engineering and software reuse. The US Government supported report [106] states: "*FODA is a necessary first step*" for software reuse. To the extent that TripTych domain engineering with its subsequent requirements engineering indeed encourages reuse at all levels: domain descriptions and requirements prescription, we can only agree. Another source on FODA is [64]. Since FODA "leans" quite heavily on 'Software Product Line Engineering' our remarks in that section, next, apply equally well here.

7: Software Product Line Engineering: Software product line engineering, earlier known as domain engineering, is the entire process of reusing domain knowledge in the production of new software systems. Key concerns of software product line engineering are reuse, the building of repositories of reusable software components, and domain specific languages with which to more-or-less automatically build software based on reusable software components. These are not the primary concerns of TripTych domain science & engineering. But they do become concerns as we move from domain

[32] http://martinfowler.com/dsl.h

descriptions to requirements prescriptions. But it strongly seems that software product line engineering is not really focused on the concerns of domain description — such as is TripTych domain engineering. It seems that software product line engineering is primarily based, as is, for example, FODA: Feature-oriented Domain Analysis, on analysing features of software systems. Our [34] puts the ideas of software product lines and model-oriented software development in the context of the TripTych approach.

8: Problem Frames: The concept of problem frames is covered in [102]. Jackson's prescription for software development focus on the "triple development" of descriptions of the problem world, the requirements and the machine (i.e., the hardware and software) to be built. Here domain analysis means the same as for us: the problem world analysis. In the problem frame approach the software developer plays three, that is, all the TripTych rôles: domain engineer, requirements engineer and software engineer, "all at the same time", iterating between these rôles repeatedly. So, perhaps belabouring the point, domain engineering is done only to the extent needed by the prescription of requirements and the design of software. These, really are minor points. But in "restricting" oneself to consider only those aspects of the domain which are mandated by the requirements prescription and software design one is considering a potentially smaller fragment [103] of the domain than is suggested by the TripTych approach. At the same time one is, however, sure to consider aspects of the domain that might have been overlooked when pursuing domain description development in the "more general" TripTych approach.

9: Domain Specific Software Architectures (DSSA): It seems that the concept of DSSA was formulated by a group of ARPA[33] project "seekers" who also performed a year long study (from around early-mid 1990s); key members of the DSSA project were Will Tracz, Bob Balzer, Rick Hayes-Roth and Richard Platek [149]. The [149] definition of domain engineering is *"the process of creating a DSSA: domain analysis and domain modeling followed by creating a software architecture and populating it with software components."* This definition is basically followed also by [120, 142, 116]. Defined and pursued this way, DSSA appears, notably in these latter references, to start with the analysis of software components, "per domain", to identify commonalities within application software, and to then base the idea of software architecture on these findings. Thus DSSA turns matter "upside-down" with respect to TripTych requirements development by starting with software components, assuming that these satisfy some requirements, and then suggesting domain specific software built using these components. This is not what we are doing: we suggest, [26], that requirements can be "derived" systematically from, and formally related back to domain descriptionss without, in principle, considering software components, whether already existing, or being subsequently developed. Of course, given a domain description it is obvious that one can develop, from it, any number of requirements prescriptions and that these may strongly hint at shared, (to be) implemented software components; but it may also, as well, be the case that two or more requirements prescriptions "derived" from the same domain description may share no software components whatsoever ! It seems to this author that had the DSSA promoters based their studies and practice on also using formal specifications, at all levels of their study and practice, then some very interesting insights might have arisen.

10: Domain Driven Design (DDD): Domain-driven design (DDD)[34] *"is an approach to developing software for complex needs by deeply connecting the implementation to an evolving model of the core business concepts; the premise of domain-driven design is the following: placing the project's primary focus on the core domain and domain logic; basing complex designs on a model; initiating a creative collaboration between technical and domain experts to iteratively cut ever closer to the conceptual heart of the problem."*[35] We have studied some of the DDD literature, mostly only accessible on the Internet,

[33] ARPA: The US DoD Advanced Research Projects Agency
[34] Eric Evans: http://www.domaindrivendesign.org/
[35] http://en.wikipedia.org/wiki/Domain-driven_design

but see also [95], and find that it really does not contribute to new insight into domains such as wee see them: it is just "plain, good old software engineering cooked up with a new jargon.

11: Unified Modeling Language (UML): Three books representative of UML are [53, 138, 104]. The term domain analysis appears numerous times in these books, yet there is no clear, definitive understanding of whether it, the domain, stands for entities in the domain such as we understand it, or whether it is wrought up, as in several of the 'approaches' treated in this section, to wit, in items [3–5, 7–9] with either software design (as it most often is), or requirements prescription. Certainly, in UML, in [53, 138, 104] as well as in most published papers claiming "adherence" to UML, that domain analysis usuallyis manifested in some UML text which "models" some requirements facet. Nothing is necessarily wrong with that, but it is therefore not really the TripTych form of domain analysis with its concepts of abstract representations of endurant and perdurants, with its distinctions between domain and requirements, and with its possibility of "deriving" requirements prescriptions from domain descriptions. The UML notion of class diagrams is worth relating to our structuring of the domain. Class diagrams appear to be inspired by [8, Bachman, 1969] and [60, Chen, 1976]. It seems that (i) each part sort — as well as other than part (or material) sorts — deserves a class diagram (box); and (ii) that (assignable) attributes — as well as other non-part (or material) types — are written into the diagram box. Class diagram boxes are line-connected with annotations where some annotations are as per the mereology of the part type and the connected part types and others are not part related. The class diagrams are said to be object-oriented but it is not clear how objects relate to parts as many are rather implementation-oriented quantities. All this needs looking into a bit more, for those who care.

12: Requirements Engineering: There are in-numerous books and published papers on requirements engineering. A seminal one is [152]. I, myself, find [112] full of very useful, non-trivial insight. [69] is seminal in that it brings a number or early contributions and views on requirements engineering. Conventional text books, notably [122, 125, 144] all have their "mandatory", yet conventional coverage of requirements engineering. None of them "derive" requirements from domain descriptions, yes, OK, from domains, but since their description is not mandated it is unclear what "the domain" is. Most of them repeatedly refer to domain analysis but since a written record of that domain analysis is not mandated it is unclear what "domain analysis" really amounts to. Axel van Laamsweerde's book [152] is remarkable. Although also it does not mandate descriptions of domains it is quite precise as to the relationships between domains and requirements. Besides, it has a fine treatment of the distinction between goals and requirements, also formally. Most of the advices given in [112] can beneficially be followed also in TripTych requirements development. Neither [152] nor [112] preempts TripTych requirements development.

Summary of Comparisons

We find that there are two kinds of relevant comparisons: the concept of ontology, its science more than its engineering, and the *Problem Frame* work of Michael A. Jackson. The ontology work, as commented upon in Item **[1]** (Pages 64–66), is partly relevant to our work: There are at least two issues: Different classes of domains may need distinct upper ontologies. Section 1.4.1 suggests that there may be different upper ontologies for non-manifest domains such as *financial systems*, etcetera. This seems to warrant at least a comparative study. We have assumed, cf. Sect. 1.2.9, that attributes cannot be separated from parts. [105, Johansson 2005] develops the notion that *persisting quality instances are enduring particulars.* The issue need further clarification.

Of all the other "comparison" items (**[2]**–**[12]**) basically only Jackson's *problem frames* (item **[8]**) really take the same view of domains and, in essence, basically maintain similar relations between requirements prescription and domain description. So potential sources of, we should claim, mutual inspiration ought be found in one-another's work — with, for example, [88, 103], and the present document, being a good starting point.

But none of the referenced works make the distinction between discrete endurants (parts) and their qualities, with their further distinctions between unique identifiers, mereology and attributes. And none of them makes the distinction between parts and materials. Therefore our contribution can include the mapping of parts into behaviours interacting as per the part mereologies as highlighted in the process schemas of Sect. 1.3.11 Pages 60–63.

1.4.4 Open Problems

The TripTych approach to software development, based, as it is, on an initial, serious phase of domain engineering — a new phase of software engineering for which we claim to now have laid a solid foundation we claim, for domain engineering — opens up for a variety of issues that need further study. The entries in this section are not ordered according to any specific principle.

1: Ontology Relations: As was evident in our coverage, Item **[1]** Pages 64–66, a more exact understanding of the relations between the "classical" AI/information science/ontology view of domains, cf. [13, 14, 105], and the algorithmic view of domains, as presented in the current paper, seems required. The almost disparate jargon of the two "camps" seems, however, to be a hindrance.

2: Analysis of Perdurants: A study of perdurants, Sect. 1.3, as detailed as that of our study of endurants, ought be carried out. One difficulty, as we see it, is the choice of formalisms: whereas the basic formalisms for the expression of endurants and their qualities was type theory and simple functions and predicates, there is no such simple set of formal constructs that can "carry" the expression of behaviours.Besides the textual CSP, [97], there is graphic notations of Petri Nets, [132], Message Sequence Charts, [99], State-charts, [92], and others.

3: Commensurate Discrete and Continuous Models: Section 1.3.6 on Page 56 hinted at co-extensive descriptions of discrete and continuous behaviours, the former in, for example, RSL, the latter in, typically, the calculus mathematics of partial different equations (PDEs). The problem that arises in this situation is the following: there will be, say variable identifiers, e.g., x, y, ..., z which in the RSL formalisation has one set of meanings, but which in the PDE "formalisation" has another set of meanings. Current formal specification languages[36] do not cope with continuity. Some research is going on. But to substantially cover, for example, the proper description of laminar and turbulent flows in networks (e.g., pipelines, Example 61 on Page 56) requires more substantial results.

4: Interplay between Parts and Materials: Examples 49 on Page 46, 50 on Page 47, 51 on Page 48 and 61 on Page 56 revealed but a small fraction of the problems that may arise in connection with modeling the interplay between parts and materials. Subject to proper formal specification language and, for example PDE specification we may expect more interesting laws, as for example those of Examples 50 on Page 47, 51 on Page 48, and even proof of these as if they were theorems. Formal specifications have focused on verifying properties of requirements and software designs. With co-extensive (i.e., commensurate) formal specifications of both discrete and continuous behaviours we may expect formal specifications to also serve as bases for predictions.

5: Dynamics: There is a serious limitation in what can be modeled with the present approach. Although we can model the dynamic introduction of new atomic or removal of existing parts, when members of a composite set of such parts, we cannot model the dynamic introduction or removal of the processes corresponding to such parts. Also we have not shown how to model global time. And, although we can model spatial positions, we have not shown how to model spatial locations. These deliberate omissions are due to the facts that the description language, RSL, cannot model continuity and that it cannot provide for arbitrary models of time [150]. Here is an area worth studying.

[36] Alloy [100], Event B [1], RSL [85], VDM-SL [48, 49, 77], Z, etc.

6: Precise Descriptions of Manifest Domains: The focus on the principles, techniques and tools of domain analysis & description has been such domains in which humans play an active rôle. Formal descriptions of domains may serve to prove properties of domains, in other words, to understand better these domains, and to validate requirements derived from such domain descriptions, and thereby to ensure that software derived from such requirements is not only correct, but also meet users expectations. Improved understanding of man-made domains — without necessarily leading to new software — may serve to improve the "business processes" of these domains, make them more palatable for the human actors, make them more efficient wrt. resource-usage. Descriptions of domains are descriptions of the syntax and semantics of the technical languages used in speaking about and in the domain. The domain analysis required for the design of programming languages is based on computability: mathematical logic and recursive function theory. The domain analysis required for "real-world" domains is not based on computability: that "world" is not computable. Requirements engineering based on domain descriptions is based on deriving computable subsets of refined domain descriptions. The classical theory and practice of programming language semantics and compiler development [15] and [23, Part VII (Chapters 16–19)] can now be further developed into a theory and practice for deriving general software from formal domain descriptions [26].

Descriptions of domains are descriptions of the syntax and semantics of the technical languages used in speaking about and in the domain. The domain analysis required for the design of programming languages is based on computability: mathematical logic and recursive function theory. The domain analysis required for "real-world" domains is not based on computability: that "world" is not computable. Requirements engineering based on domain descriptions is based on deriving computable subsets of refined domain descriptions. The classical theory and practice of programming language semantics and compiler development [15] and [23, Part VII (Chapters 16–19)] can now be further developed into a theory and practice for deriving general software from formal domain descriptions [26].

Physicists study 'Mother Nature', the world without us. Domain scientists study man-made part and material based universes with which we interact — the world within and without us. Classical engineering builds on laws of physics to design and construct buildings, chemical compounds, machines and electrical and electronic products. So far software engineers have not expressed software requirements on any precise description of the basis domain. This paper strongly suggests such a possibility. Regardless: it is interesting to also formally describe domains; and, as shown, it can be done.

7: Towards Mathematical Models of Domain Analysis & Description: There are two aspects to a precise description of the ***domain analysis prompt***s and ***domain description prompt***s. There is that of describing the individual prompts as if they were "machine instructions" for an albeit strange machine; and there is that of describing the interplay between prompts: the sequencing of ***domain description prompt***s as determined by the outcome of the ***domain analysis prompt***s. We have described and formalised the latter in [43, Processes]; and we are in the midst of describing and formalising the former in [35, Prompts].

8: Laws of Descriptions: A Calculus of Prompts: Laws of descriptions deal with the order and results of applying the domain analysis and description prompts. Some laws are covered in [33]. It is expected that establishing formal models of the prompts, for example as outlined in [35, 43], will help identify such laws. The various description prompts apply to parts (etc.) of specified sorts (etc.) and to a "hidden state". The "hidden state" has two major elements: the domain and the evolving description texts. An "execution" of a prompt potentially changes that "hidden state". Let P, PA and PB be composite part sorts where PA and PB are derived from P. Let $\mathfrak{R}_i$, $\mathfrak{R}_j$, etc., be suitable functions which rename sort, type and attribute names. In a proper prompt calculus we would expect `observe_part_sorts_PA;observe_part_sorts_PB`, when "executed" by one and the same domain engineer, to yield the same "hidden state" as `observe_part_sorts_PB;`$\mathfrak{R}_i$`;observe_part_sorts_PA;`$\mathfrak{R}_j$. Also one would expect `observe_part_sorts_PA;`$\mathfrak{R}_i$`;observe_part_sorts_PA;`$\mathfrak{R}_j$. to yield the same state as just `observe_part_sorts_PA` given suitable renaming functions.

Well ? or does one really ?

There are some assumptions that are made here. One pair of assumptions is that the domain is fixed and to one observer. yields the same analysis and description results no matter in which order prompts are "executed". Another assumption is that the domain engineer does not get wiser as analysis and description progresses. If, as one can very well expect, the domain engineer does get wiser, then former results may be discarded and either replaced by newer analysis and descriptions or prompts repeated. In such cases these laws do not hold.

9: Domains and Galois Connections: Section 1.1.8 on Page 23 very briefly mentioned that formal concepts form Galois Connections. In the seminal [83] a careful study is made of this fact and beautiful examples show the implications for domains. It seems that our examples have all been too simple. They do not easily lead on to the "discovery" of "new" domain concepts from appropriate concept lattices. We refer to [47, Section 9]. Further study need be done.

10: Laws of Domain Description Prompts: Typically `observe_part_sorts` applies to a composite part, p:P, and yield descriptions of one or more part sorts: $p_1{:}P_1$, $p_2{:}P_2$, ..., $p_m{:}P_m$. Let $p_i{:}P_i$, $p_j{:}P_j$, ..., $p_k{:}P_k$ (of these) be composite. Now `observe_part_sorts`(p_i) and `observe_part_sorts`(p_j), etc., can be applied and yield texts $text_i$, respectively $text_j$. A law of domain description prompts now expresses that the order in which the two or more observers is applied is immaterial, that is, they commute. In [33] we made an early exploration of such laws of domain description prompts. More work, see also below, need be done.

11: Domain Theories:: An ultimate goal of domain science & engineering is to prove properties of domains. Well, maybe not properties of domains, but then at least properties of domain descriptions. If one can be convinced that a posited domain description indeed is a faithful description of a domain, then proofs of properties of the domain description are proofs of properties of that domain. Ultimately domain science & engineering must embrace such studies of *laws of domains*. Here is a fertile ground for zillions of Master and PhD theses !

Example 66 . A Law of Train Traffic:Let a transport net, n:N, be that of a railroad system. Hubs are train stations. Links are rail lines between stations. Let a train timetable record train arrivals and train departures from stations. And let such a timetable be modulo some time interval, say typically 24 hours. Now let us (idealistically) assume that actual trains arrive at and depart from train stations according the train timetable and that the train traffic includes all and only such trains as are listed in the train timetable. Now a law of train traffic expresses *"Over the modulo time interval of a train timetable it is the case that the number of trains arriving at a station minus the number of trains ending their journey at that station plus the number of trains starting their journey at that station equals number of trains departing from that station."* ■

1.4.5 Tony Hoare's Summary on 'Domain Modeling'

In a 2006 e-mail, in response, undoubtedly to my steadfast, perhaps conceived as stubborn insistence, on domain engineering, Tony Hoare summed up his reaction to domain engineering as follows, and I quote[37]:

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

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

2 They make explicit the preconditions about the real world that have to be made in any embedded software design, especially one that is going to be formally proved.

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

3 They describe the whole range of possible designs for the software, and the whole range of technologies available for its realisation.

4 They provide a framework for a full analysis of requirements, which is wholly independent of the technology of implementation.

5 They enumerate and analyse the decisions that must be taken earlier or later in any design project, and identify those that are independent and those that conflict. Late discovery of feature interactions can be avoided."

All of these issues are dealt with in [24, Part IV].

Tony Hoare's list pertains to a wider range that just the Manifest Domains treated in this chapter.

1.4.6 Beauty Is Our Business

This paper started with a quote from Dostovevsky's `The Idiot`.

It's life that matters, nothing but life –
the process of discovering, the everlasting and perpetual process,
not the discovery itself, at all.[38]

I find that quote appropriate in the following, albeit rather mundane, sense: It is the process of analysing and describing a domain that exhilarates me: that causes me to feel very happy and excited. There is beauty [76, E.W. Dijkstra Festschrift] not only in the result but also in the process.

[38] `Fyodor Dostoyevsky, The Idiot, 1868, Part 3, Sect. V`

Chapter 2

Domain Facets

Chapter Status

This chapter is to be revised.

The domain analysis & description method outlined in Chapter 1 was idealised ! It did not take into account the views that anyone particular stake-holder might have on that stake-holder's understanding of the domain. The techniques and tools of the domain analysis & description approach appear "technical" in the sense that they could be applied without considering such state-holder views. This chapter attempts to remedy that problem.

2.1 Stake-holders

The key concept above was that of stake-holder. Let us define and examine that concept.

Definition 18 . ***State-holder:*** *By a* ***domain stake-holder*** *we shall understand a person, or a group of persons, "united" somehow in their common interest in, or dependency on the domain; or an institution, an enterprise, or a group of such, (again) characterised (and, again, loosely) by their common interest in, or dependency on the domain* ■

Example 67 . **Some Stake-holders**:For the domain of banking we can list at least the following distinct, i.e., different stake-holder groups: clients, i.e., customers how have demand/deposit accounts, etc., bank teller, "back office"bank clerks, bank managers (at various levels). bank owners, suppliers (of banking equipment), banking regulators[1], politicians, etcetera. They are distinct in that no two of these groups appears to have exactly and only the same concerns. ■

What separates one group of stake-holders from other groups are that they each put different emphasis on the inclusion or understanding of a number for domain facets (to be defined immediately below). In this chapter we shall now concern ourselves with the concept of facets.

2.2 Domain Facets

Definition 19 . ***Facet:*** *By a* ***domain facet*** *we shall understand one amongst a finite set of generic ways of analysing a domain: a view of the domain, such that the different facets cover conceptually different views, and such that these views together cover the domain* ■

[1] In the US: the Federal Deposit Insurance Corporation, the Federal Reserve Board, and the Office of the Comptroller of the Currency; in Great Britain: Financial Services Authority.

The hedge here is "finite set of generic ways". Thus there is an assumption, a conjecture to be possibly refuted. Namely the postulate that there is a finite number of facets. We shall offer the following facets: intrinsics, support technology, management and organisation, rules and regulations (and scripts), and human behaviour.

2.2.1 Intrinsics

Definition 20 . ***Intrinsics:*** *By **domain intrinsics** we shall understand those phenomena and concepts of a domain which are basic to any of the other facets (listed earlier and treated, in some detail, below), with such domain intrinsics initially covering at least one specific, hence named, stake-holder view* ■

Example 68 ***Railway Net Intrinsics.*** We narrate and formalise three railway net intrinsics.

- From the view of *potential train passengers* a railway net consists of lines, stations and trains. A line connects exactly two distinct stations.
- From the view of *actual train passengers* a railway net — in addition to the above — allows for several lines between any pair of stations and, within stations, provides for one or more platform tracks from which to embark or alight a train.
- From the view of *train operating staff* a railway net — in addition to the above — has lines and stations consisting of suitably connected rail units. A rail unit is either a simple (i.e., linear, straight) unit, or is a switch unit, or is a simple crossover unit, or is a switchable crossover unit, etc. Simple units have two connectors. Switch units have three connectors. Simple and switchable crossover units have four connectors. A path (through a unit) is a pair of connectors of that unit. A state of a unit is the set of paths, in the direction of which a train may travel. A (current) state may be empty: The unit is closed for traffic. A unit can be in either one of a number of states of its state space.

A summary formalisation of the three narrated railway net intrinsics could be:

- *Potential train passengers:*

```
type
  N, L, S, Sn, Ln
value
  obs_Ls: N → L-set, obs_Ss: N → S-set
  obs_Ln: L → Ln, obs_Sn: S → Sn
  obs_Sns: L → Sn-set, obs_Lns: S → Ln-set
axiom
  ...
```

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

- *Actual train passengers:*

```
type
  Tr, Trn
value
```



```
    obs_Trs: S → Tr-set, obs_Trn: Tr → Trn
axiom
    ...
```

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

- *Train operating staff:*

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

Unit and connector sorts have been added as have concrete types for paths, unit states, unit state spaces and related observer functions, including unit state and unit state space observers. The reader is invited to compare the three narrative descriptions with the three formal descriptions, line by line

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

Example 69. ***Comparable Intrinsics.*** We refer to Example 68. We claim that the concept of nets, lines and stations in the three models of Example 68 must relate. The simplest possible relationships are to let the third model be the common "unifier" and to mandate

- that the model of nets, lines and stations of the *potential train passengers* formalisation is that of nets, lines and stations of the *train operating staff* model; and
- that the model of nets, lines, stations and tracks of the *actual train passengers* formalisation is that of nets, lines, stations of the *train operating staff* model.

Thus the third model is seen as the definitive model for the stake-holder views

Example 70. ***Intrinsics of Switches.*** The intrinsic attribute of a rail switch is that it can take on a number of states. A simple switch (${}^{c_|}Y_{c}^{c_/}$) has three connectors: $\{c,c_|,c_/\}$. c is the connector of the common rail from which one can either "go straight" $c_|$, or "fork" $c_/$ (Fig. 2.1). So we have that a possible state space of such a switch could be ω_{g_s} :

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

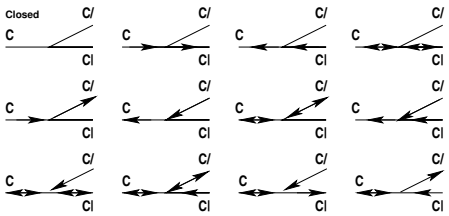

Figure 2.1: Possible states of a rail switch

The above models a general switch ideally. Any particular switch ω_{p_s} may have $\omega_{p_s} \subset \omega_{g_s}$. Nothing is said about how a state is determined: who sets and resets it, whether determined solely by the physical position of the switch gear, or also by visible or virtual (i.e., invisible, intangible) signals up or down the rail, away from the switch ■

Conceptual Versus Actual Intrinsics

In order to bring an otherwise seemingly complicated domain across to the reader, one may decide to present it piecemeal:[2] First, one presents the very basics, the fewest number of inescapable entities, functions and behaviours. Then, in a step of enrichment, one adds a few more (intrinsic) entities, functions and behaviours. And so forth. In a final step one adds the last (intrinsic) entities, functions and behaviours. In order to develop what initially may seem to be a complicated domain, one may decide to develop it piecemeal: We basically do as for the presentation steps: Steps of enrichment — from a big lie, via increasingly smaller lies, till one reaches a truth!

On Modelling Intrinsics

Domains can be characterised by intrinsically being endurant, or function, or event, or behaviour intensive. Software support for activities in such domains then typically amount to database systems, computation-bound systems, real-time embedded systems, respectively distributed process monitoring and control systems. Modelling the domain intrinsics in respective cases can often be done property-oriented specification languages (like CafeOBJ [82] or CASL [63]), model-oriented specification languages (like Alloy [100], B [1], VDM [48, 49, 77], RSL [85], or Z [157]), event-based languages (like Petri nets [132] or CSP [97]), respectively process-based specfication languages (like MSCs [99], LSCs [65, 93, 108], statecharts [92], or CSP [97]).

2.2.2 Support Technologies

Definition 21 . ***Support technology:*** *By a domain* ***support technology*** *we shall understand ways and means of implementing certain observed phenomena or certain conceived concepts* ■

Example 71. ***Railway Support Technology.*** We give a rough sketch description of possible rail unit switch technologies.

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

(ii) With the advent of reasonably reliable mechanics, pulleys and levers[3] (and steel wires), switches were made to change state by means of "throwing" levers in a cabin tower located centrally at the station (with the lever then connected through wires etc., to the actual switch).

[2] That seemingly complicated domain may seem very complicated, containing hundreds of entities. Instead of presenting all the entities in one "fell swoop", one presents them in stages.

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

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

It must be stressed that Example 71 is just a rough sketch. In a proper narrative description the software (cum domain) engineer must describe, in detail, the subsystem of electronics, electro-mechanics and the human operator interface (buttons, lights, sounds, etc.).

An aspect of supporting technology includes recording the state-behaviour in response to external stimuli. We give an example.

Example 72. ***Probabilistic Rail Switch Unit State Transitions.*** Figure 2.2 indicates a way of formalising this aspect of a supporting technology. Figure 2.2 intends to model the probabilistic (erroneous and correct) behaviour of a switch when subjected to settings (to switched (s) state) and resettings (to direct (d) state). A switch may go to the switched state from the direct state when subjected to a switch setting s with probability psd ■

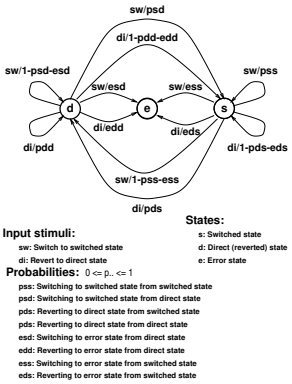

Figure 2.2: Probabilistic state switching

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

Example 73. ***Railway Optical Gates.*** Train traffic (itf:iTF), intrinsically, is a total function over some time interval, from time (t:T) to continuously positioned (p:P) trains (tn:TN).

Conventional optical gates sample, at regular intervals, the intrinsic train traffic. The result is a sampled traffic (stf:sTF). Hence the collection of all optical gates, for any given railway, is a partial function from intrinsic to sampled train traffics (stf).

We need to express quality criteria that any optical gate technology should satisfy — relative to a necessary and sufficient description of a closeness predicate. The following axiom does that:

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

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

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

$\mathscr{D}$ is not an RSL operator. It is a mathematical way of expressing the definition set of a general function. Hence it is not a computable function. Check-ability is an issue of testing the optical gates when delivered for conformance to the closeness predicate, i.e., to the axiom ■

On Modelling Support Technologies

Support technologies in their relation to the domain in which they reside typically reflect real-time embeddedness. As such the techniques and languages for modelling support technologies resemble those for modelling event and process intensity, while temporal notions are brought into focus. Hence typical modelling notations include event-based languages (like Petri nets [132] or CSP [97]), respectively process-based specification languages (like MSCs [99], LSCs [65, 93, 108], Statecharts [92], or CSP [97]), as well as temporal languages (like the Duration Calculus, DC [160] and Temporal Logic of Actions, TLA+ [110]).

2.2.3 Management and Organisation

Example 74 ***Train Monitoring, I.*** In China, as an example, rescheduling of trains occurs at stations and involves telephone negotiations with neighbouring stations ("up and down the lines"). Such rescheduling negotiations, by phone, imply reasonably strict management and organisation (M&O). This kind of M&O reflects the geographical layout of the rail net ■

Definition 22 . ***Domain Management:*** *By domain* ***management*** *we shall understand such people (such decisions) (i) who (which) determine, formulate and thus set standards (cf. rules and regulations, Sect. 2.2.4) concerning strategic, tactical and operational decisions; (ii) who ensure that these decisions are passed on to (lower) levels of management, and to floor staff; (iii) who make sure that such orders, as they were, are indeed carried out; (iv) who handle undesirable deviations in the carrying out of these orders cum decisions; and (v) who "backstop" complaints from lower management levels and from floor staff* ■

Definition 23 . ***Domain Organisation:*** *By domain* ***organisation*** *we shall understand the structuring of management and non-management staff levels; the allocation of strategic, tactical and operational concerns to within management and non-management staff levels; and hence the "lines of command": who does what, and who reports to whom, administratively and functionally*
■

Example 75. ***Railway Management and Organisation: Train Monitoring, II.*** We single out a rather special case of railway management and organisation. Certain (lowest-level operational and station-located) supervisors are responsible for the day-to-day timely progress of trains within a station and along its incoming and outgoing lines, and according to given timetables. These supervisors and their immediate (middle-level) managers (see below for regional managers) set guidelines (for local station and incoming and outgoing lines) for the monitoring of train traffic, and for controlling trains that are either ahead of or behind their schedules. By an incoming and an outgoing line we mean part of a line between two stations, the remaining part being handled by neighbouring station management. Once it has been decided, by such a manager, that a train is not following its schedule, based on information monitored by non-management staff, then that manager directs that staff: (i) to suggest a new schedule for the train in question, as well as for possibly affected other trains, (ii) to negotiate the new schedule with appropriate neighbouring stations, until a proper reschedule can be decided upon, by the managers at respective stations, (iii) and to enact that new schedule.[4] A (middle-level operations) manager for regional traffic, i.e., train traffic involving several stations and lines, resolves possible disputes and conflicts ■

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

Conceptual Analysis, First Part

People staff enterprises, the components of infrastructures with which we are concerned, i.e., for which we develop software. The larger these enterprises — these infrastructure components — the more need there is for management and organisation. The rôle of management is roughly, for our purposes, twofold: first, to perform strategic, tactical and operational work, to set strategic, tactical and operational policies — and to see to it that they are followed. The rôle of management is, second, to react to adverse conditions, that is, to unforeseen situations, and to decide how they should be handled, i.e., conflict resolution.

Policy-setting should help non-management staff operate normal situations — those for which no management interference is thus needed. And management "backstops" problems: management takes these problems off the shoulders of non-management staff.

To help management and staff know who's in charge wrt. policy setting and problem handling, a clear conception of the overall organisation is needed. Organisation defines lines of communication within management and staff, and between these. Whenever management and staff has to turn to others for assistance they usually, in a reasonably well-functioning enterprise, follow the command line: the paths of organigrams — the usually hierarchical box and arrow/line diagrams.

Methodological Consequences

The *management and organisation* model of a domain is a partial specification; hence all the usual abstraction and modelling principles, techniques and tools apply. More specifically, management is a set of predicates, observer and generator functions which either parameterise other, the operations functions, that is, determine their behaviour, or yield results that become arguments to these other functions

Organisation is thus a set of constraints on communication behaviours. Hierarchical, rather than linear, and matrix structured organisations can also be modelled as sets (of recursively invoked sets) of equations.

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

Conceptual Analysis, Second Part

To relate classical organigrams to formal descriptions we first show such an organigram (Fig. 2.3), and then we show schematic processes which — for a rather simple scenario — model managers and the managed!

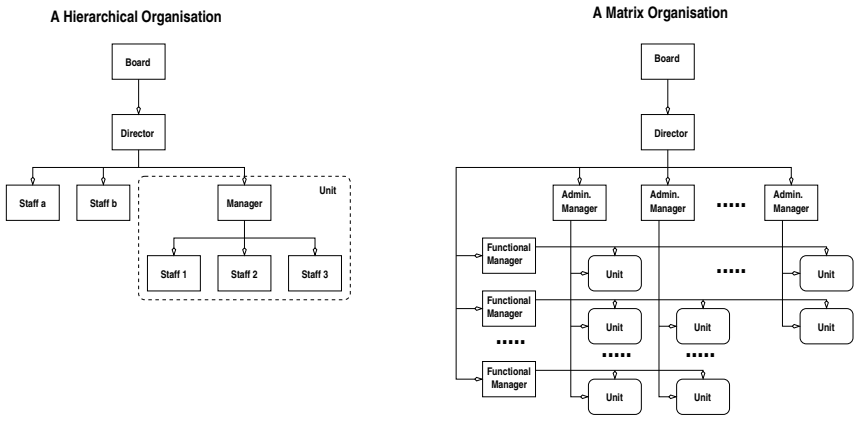

Figure 2.3: Organisational structures

Based on such a diagram, and modelling only one neighbouring group of a manager and the staff working for that manager we get a system in which one manager, mgr, and many staff, stf, coexist or work concurrently, i.e., in parallel. The mgr operates in a context and a state modelled by ψ. Each staff, stf(i) operates in a context and a state modelled by $s\sigma(i)$.

type
  Msg, Ψ, Σ, Sx
  $S\Sigma = Sx \xrightarrow{m} \Sigma$
channel
  { ms[i]:Msg | i:Sx }
value
  $s\sigma$:SΣ, ψ:Ψ

  sys: **Unit** → **Unit**
  sys() ≡ ‖ { st(i)($s\sigma$(i)) | i:Sx } ‖ mg(ψ)

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

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

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

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

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

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

Both manager and staff processes recurse (i.e., iterate) over possibly changing states. The management process non-deterministically, external choice, "alternates" between "broadcast"-issuing orders to staff and receiving individual messages from staff. Staff processes likewise non-deterministically, external choice, alternate between receiving orders from management and issuing individual messages to management.

The conceptual example also illustrates modelling stake-holder behaviours as interacting (here CSP-like [97]) processes.

On Modelling Management and Organisation

Management and organisation basically spans entity, function, event and behaviour intensities and thus typically require the full spectrum of modelling techniques and notations — summarised in the two "On Modelling ..." paragraphs at the end of the two previous sections.

2.2.4 Rules and Regulations

Definition 24 . ***Domain Rules:*** *By a domain **rule** we shall understand some text (in the domain) which prescribes how people or equipment are expected to behave when dispatching their duty, respectively when performing their function* ■

Definition 25 . ***Domain Regulation:*** *By a domain **regulation** we shall understand some text (in the domain) which prescribes what remedial actions are to be taken when it is decided that a rule has not been followed according to its intention* ■

Example 76. ***Trains at Stations.***

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

  > *In any three-minute interval at most one train may either arrive to or depart from a railway station.*

- Regulation: *If it is discovered that the above rule is not obeyed*, then there is some regulation which prescribes administrative or legal management and/or staff action, as well as some correction to the railway traffic ■

Example 77. ***Trains Along Lines.***

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

  There must be at least one free sector (i.e., without a train) between any two trains along a line.

- Regulation: *If it is discovered that the above rule is not obeyed*, then there is some regulation which prescribes administrative or legal management and/or staff action, as well as some correction to the railway traffic ■

A Meta-characterisation of Rules and Regulations

At a meta-level, i.e., explaining the general framework for describing the syntax and semantics of the human-oriented domain languages for expressing rules and regulations, we can say the following: There are, abstractly speaking, usually three kinds of languages involved wrt. (i.e., when expressing) rules and regulations (respectively when invoking actions that are subject to rules and regulations). Two languages, Rules and Reg, exist for describing rules, respectively regulations; and one, Stimulus, exists for describing the form of the [always current] domain action stimuli.

A syntactic stimulus, sy_sti, denotes a function, se_sti:STI: $\Theta \to \Theta$, from any configuration to a next configuration, where configurations are those of the system being subjected to stimulations. A syntactic rule, sy_rul:Rule, stands for, i.e., has as its semantics, its meaning, rul:RUL, a predicate over current and next configurations, $(\Theta \times \Theta) \to$ **Bool**, where these next configurations have been brought about, i.e., caused, by the stimuli. These stimuli express: If the predicate holds then the stimulus will result in a valid next configuration.

```
type
  Stimulus, Rule, Θ
  STI = Θ → Θ
  RUL = (Θ × Θ) → Bool
value
  meaning: Stimulus → STI
  meaning: Rule → RUL

  valid: Stimulus × Rule → Θ → Bool
  valid(sy_sti,sy_rul)(θ) ≡ meaning(sy_rul)(θ,(meaning(sy_sti))(θ))

  valid: Stimulus × RUL → Θ → Bool
  valid(sy_sti,se_rul)(θ) ≡ se_rul(θ,(meaning(sy_sti))(θ))
```

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

The predicate is almost the inverse of the rules functions. The action function serves to undo the stimulus function.

```
type
  Reg
  Rul_and_Reg = Rule × Reg
  REG = Pre_REG × Act_REG
  Pre_REG = Θ × Θ → Bool
```

```
  Act_REG = Θ → Θ
value
  interpret: Reg → REG
```

The idea is now the following: Any action of the system, i.e., the application of any stimulus, may be an action in accordance with the rules, or it may not. Rules therefore express whether stimuli are valid or not in the current configuration. And regulations therefore express whether they should be applied, and, if so, with what effort.

More specifically, there is usually, in any current system configuration, given a set of pairs of rules and regulations. Let (sy_rul,sy_reg) be any such pair. Let sy_sti be any possible stimulus. And let θ be the current configuration. Let the stimulus, sy_sti, applied in that configuration result in a next configuration, θ', where θ' = (meaning(sy_sti))(θ). Let θ' violate the rule, ∼valid(sy_sti,sy_rul)(θ), then if predicate part, pre_reg, of the meaning of the regulation, sy_reg, holds in that violating next configuration, pre_reg(θ,(meaning(sy_sti))(θ)), then the action part, act_reg, of the meaning of the regulation, sy_reg, must be applied, act_reg(θ), to remedy the situation.

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

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

On Modelling Rules and Regulations

Usually rules (as well as regulations) are expressed in terms of domain entities, including those grouped into "the state", functions, events, and behaviours. Thus the full spectrum of modelling techniques and notations may be needed. Since rules usually express properties one often uses some combination of axioms and well-formedness predicates. Properties sometimes include temporality and hence temporal notations (like Duration Calculus [160] or Temporal Logic of Actions [110]) are used. And since regulations usually express state (restoration) changes one often uses state changing notations (such as found in B [1], RSL [85], VDM-SL [48, 49, 77], and Z [157]). In some cases it may be relevant to model using some constraint satisfaction notation [2] or some Fuzzy Logic notations [151].

2.2.5 Scripts and Licensing Languages

Definition 26 . ***Domain Script:*** *By a domain* ***script*** *we shall understand the structured, almost, if not outright, formally expressed, wording of a rule or a regulation that has legally binding power, that is, which may be contested in a court of law* ■

Example 78 ***A Casually Described Bank Script.*** We deviate, momentarily, from our line of railway examples, to exemplify one from banking. Our formulation amounts to just a (casual) rough sketch. It is followed by a series of four large examples. Each of these elaborate on the theme of (bank) scripts.

The problem area is that of how repayments of mortgage loans are to be calculated. At any one time a mortgage loan has a balance, a most recent previous date of repayment, an interest rate and a handling

fee. When a repayment occurs, then the following calculations shall take place: (i) the interest on the balance of the loan since the most recent repayment, (ii) the handling fee, normally considered fixed, (iii) the effective repayment — being the difference between the repayment and the sum of the interest and the handling fee — and the new balance, being the difference between the old balance and the effective repayment. We assume repayments to occur from a designated account, say a demand/deposit account. We assume that bank to have designated fee and interest income accounts. (i) The interest is subtracted from the mortgage holder's demand/deposit account and added to the bank's interest (income) account. (ii) The handling fee is subtracted from the mortgage holder's demand/deposit account and added to the bank's fee (income) account. (iii) The effective repayment is subtracted from the mortgage holder's demand/deposit account and also from the mortgage balance. Finally, one must also describe deviations such as overdue repayments, too large, or too small repayments, and so on ■

Example 79. ***A Formally Described Bank Script.*** First we must informally and formally define the bank state:

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

```
type
  C, A, M
  AY′ = Real, AY = {| ay:AY′ • 0<ay≤10 |}
  MI′ = Real, MI = {| mi:MI′ • 0<mi≤10 |}
  Bank′ = A_Register × Accounts × M_Register × Loans
  Bank = {| β:Bank′ • wf_Bank(β)|}
  A_Register = C →m A-set
  Accounts = A →m Balance
  M_Register = C →m M-set
  Loans = M →m (Loan × Date)
  Loan,Balance = P
  P = Nat
```

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

```
value
  ay:AY, mi:MI

  wf_Bank: Bank → Bool
  wf_Bank(ρ,α,μ,ℓ) ≡ ∪ rng ρ = dom α ∧ ∪ rng μ = dom ℓ
axiom
  ai<mi
```

Operations on banks are denoted by the commands of the bank script language. First the syntax:

```
type
  Cmd = OpA | CloA | Dep | Wdr | OpM | CloM | Pay
  OpA == mkOA(c:C)
  CloA == mkCA(c:C,a:A)
  Dep == mkD(c:C,a:A,p:P)
  Wdr == mkW(c:C,a:A,p:P)
  OpM == mkOM(c:C,p:P)
  Pay == mkPM(c:C,a:A,m:M,p:P)
```

```
CloM == mkCM(c:C,m:M,p:P)
Reply = A | M | P | OkNok
OkNok == ok | notok
```

And then the semantics:

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

And so forth for remaining commands ■

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

Licensing Languages

A special form of scripts are increasingly appearing in some domains, notably the domain of electronic, or digital media, where these licenses express that the licensor permits the licensee to render (i.e., play) works of proprietary nature CD ROM-like music, DVD-like movies, etc. while obligating the licensee to pay the licensor on behalf of the owners of these, usually artistic works. We refer to [89, 129, 139, 51, 4, 61, 159] for papers and reports on license languages.

On Modelling Scripts

Scripts (as are licenses) are like programs (respectively like prescriptions for program executions). Hence the full variety of techniques and notations for modelling programming (or specification) languages apply [67, 90, 134, 141, 148, 156]. Chapters 6–9 of Vol. 2 of [21, 23, 24] cover pragmatics, semantics and syntax techniques for defining languages.

2.2.6 Human Behaviour

Definition 27 . ***Human Behaviour:*** *By domain **human behaviour** we shall understand any of a quality spectrum of carrying out assigned work: from (i) careful, diligent and accurate, via (ii) sloppy dispatch, and (iii) delinquent work, to (iv) outright criminal pursuit* ■

Example 80. ***Banking — or Programming — Staff Behaviour.*** Let us assume a bank clerk, "in ye olde" days, when calculating, say mortgage repayments (cf. Example 78).

We would characterise such a clerk as being *diligent*, etc., if that person carefully follows the mortgage calculation rules, and checks and double-checks that calculations "tally up", or lets others do so. We would characterise a clerk as being *sloppy* if that person occasionally forgets the checks alluded to above. We would characterise a clerk as being *delinquent* if that person systematically forgets these checks. And we

would call such a person a *criminal* if that person intentionally miscalculates in such a way that the bank (and/or the mortgage client) is cheated out of funds which, instead, may be diverted to the cheater. Let us, instead of a bank clerk, assume a software programmer charged with implementing an automatic routine for effecting mortgage repayments (cf. Example 79). We would characterise the programmer as being *diligent* if that person carefully follows the mortgage calculation rules, and throughout the development verifies and tests that the calculations are correct with respect to the rules. We would characterise the programmer as being *sloppy* if that person forgets certain checks and tests when otherwise correcting the computing program under development. We would characterise the programmer as being *delinquent* if that person systematically forgets these checks and tests. And we would characterise the programmer as being a *criminal* if that person intentionally provides a program which miscalculates the mortgage interest, etc., in such a way that the bank (and/or the mortgage client) is cheated out of funds ■

Example 81. ***A Human Behaviour Mortgage Calculation.*** Example 79 gave a semantics to the mortgage calculation request (i.e., command) as would a diligent bank clerk be expected to perform it. To express, that is, to model, how sloppy, delinquent, or outright criminal persons (staff?) could behave we must modify the int_Cmd(mkPM(c,a,m,p,d′))(ρ,α,μ,ℓ) definition.

```
int_Cmd(mkPM(c,a,m,p,d′))(ρ,α,μ,ℓ) ≡
  let (b,d) = ℓ(m) in
  if q(α(a),p) /* α(a)≤p∨α(a)=p∨α(a)≤p∨... */
    then
      let i = f1(interest(mi,b,d′−d)),
          ℓ′ = ℓ † [m↦f2(ℓ(m)−(p−i))]
          α′ = α † [a↦f3(α(a)−p),ai↦f4(α(ai)+i),
                    a“staff”↦f“staff”(α(ai)+i)] in
      ((ρ,α′,μ,ℓ′),ok) end
    else
      ((ρ,α′,μ,ℓ),nok)
  end end
  pre c ∈ dom μ ∧ m ∈ μ(c)
```

The predicate q and the functions f_1, f_2, f_3, f_4 and $f_{\text{“staff”}}$ of Example 81 are deliberately left undefined.

```
q: P × P ~→ Bool
f1,f2,f3,f4,f“staff”: P ~→ P
```

They are being defined by the “staffer” when performing (incl., programming) the mortgage calculation routine ■

The point of Example 81 is that one must first define the mortgage calculation script precisely as one would like to see the diligent staff (programmer) to perform (incl., correctly program) it before one can “pinpoint” all the places where lack of diligence may “set in”. The invocations of q, f_1, f_2, f_3, f_4 and $f_{\text{“staff”}}$ designate those places.

The point of Example 81 is also that we must first domain-define, “to the best of our ability” all the places where human behaviour may play other than a desirable rôle. If we cannot, then we cannot claim that some requirements aim at countering undesirable human behaviour.

A Meta-characterisation of Human Behaviour

Commensurate with the above, humans interpret rules and regulations differently, and not always “consistently” — in the sense of repeatedly applying the same interpretations.

Our final specification pattern is therefore:

type
  Action = Θ $\xrightarrow{\sim}$ **Θ-infset**
value
  hum_int: Rule → Θ → RUL**-infset**
  action: Stimulus → Θ → Θ
  hum_beha: Stimulus × Rules → Action → Θ $\xrightarrow{\sim}$ **Θ-infset**
  hum_beha(sy_sti,sy_rul)(α)(θ) **as** θset
    post
      θset = α(θ) ∧ action(sy_sti)(θ) ∈ θset
      ∧ ∀ θ′:Θ•θ′ ∈ θset ⇒
        ∃ se_rul:RUL•se_rul ∈ hum_int(sy_rul)(θ)⇒se_rul(θ,θ′)

The above is, necessarily, sketchy: There is a possibly infinite variety of ways of interpreting some rules. A human, in carrying out an action, interprets applicable rules and chooses one which that person believes suits some (professional, sloppy, delinquent or criminal) intent. "Suits" means that it satisfies the intent, i.e., yields **true** on the pre/post-configuration pair, when the action is performed — whether as intended by the ones who issued the rules and regulations or not. We do not cover the case of whether an appropriate regulation is applied or not

The above-stated axioms express how it is in the domain, not how we would like it to be. For that we have to establish requirements.

On Modelling Human Behaviour

To model human behaviour is, "initially", much like modelling management and organsiation. But only 'initially'. The most significant human behaviour modelling aspects is then that of modelling non-determinism and looseness, even ambiguity. So a specification language which allows specifying non-determinism and looseness (like CafeOBJ [82] and RSL [85]) is to be preferred.

2.2.7 Completion

Domain acquisition resulted in typically up to thousands of units of domain descriptions. Domain analysis subsequently also serves to classify which facet any one of these description units primarily characterise. But some such "compartmentalisations" may be difficult, and may be deferred till the step of "completion". It may then be — "at the end of the day", that is, after all of the above facets have been modelled — that some description units are left as not having been described, not deliberately, but "circumstantially". It then behooves the domain engineer to fit these "dangling" description units into suitable parts of the domain description. This "slotting in" may be simple, and all is fine. Or it may be difficult. Such difficulty may be a sign that the chosen model, the chosen description, in its selection of endurants, functions, events and behaviours to model — in choosing these over other possible selections of phenomena and concepts is not appropriate. Another attempt must be made. Another selection, another abstraction of entities, functions, etc., may need be chosen. Usually however, after having chosen the abstractions of the intrinsic phenomena and concepts, one can start checking whether "dangling" description units can be fitted in "with ease".

2.2.8 Integrating Formal Descriptions

We have seen that to model the full spectrum of domain facets one needs not one, but several specification languages. No single specification language suffices. It seems highly unlikely and it appears not to be desirable to obtain a single, "universal" specification language capable of "equally" elegantly, suitably abstractly modelling all aspects of a domain. Hence one must conclude that the full modelling of domains shall deploy several formal notations. The issues are then the following which combinations of notations to select, and how to make sure that the combined specification denotes something meaningful. The ongoing

series of “Integrating Formal Methods” conferences [3] is a good source for techniques, compositions and meaning.

2.3 Closing Discussion

TO BE TYPED

Chapter 3

Prompt Semantics

Chapter Status

This chapter is under revision and is to be completed.

3.1 A Model of The Analysis & Description Process

3.1.1 A Summary of Prompts

In Chap. 1 we outlined two classes of prompts: the domain [endurant] analysis prompts:[1]

a. `is_ entity`, 23
b. `is_ endurant`, 24
c. `is_ perdurant`, 24
d. `is_ discrete`, 24
e. `is_ continuous`, 24
f. `is_ part`, 25
g. `is_ component`, 25
h. `is_ material`, 26
i. `is_ atomic`, 26
j. `is_ composite`, 26
k. `observe_ parts`, 27
l. `has_ concrete_ type`, 29
m. `has_ mereology`, 34
n. `attribute_ names`, 38
o. `has_ components`, 44
p. `has_ materials`, 45

and the domain [endurant] description prompts:

a. **obs_ part_** P, 28
b. **is_** P, 28
c. **obs_ part_** T, 29
d. **uid_ P**, 33
e. **mereo_ P**, 35
f. **upd_ mereology**, 36
g. **attr_** A, 38
h. **components**, 44
i. **obs_ part_** M, 46
j. **materials**, 46

These prompts are imposed upon the domain analyser cum describer. They are "figuratively" applied to the domain. Their orderly, sequenced application follows the method hinted at in the previous section and expressed in a pseudo-formal notation in this section. The notation looks formal but since we have not formalised these prompts it is only pseudo-formal. In [35] we shall formalise these prompts.

[1] The prompts are sorted in order of appearence. The one or two digits following the prompt names refer to page numbers minus the number of the first page of this paper + 1.

3.1.2 Preliminaries

Let P be a sort, that is, a collection of endurants. By ηP we shall understand a syntactic quantity: the name of P. By ιp:P we shall understand the semantic quantity: an (arbitrarily selected) endurant in P. And by $\eta^{-1}\eta$P we shall understand P. To guide the TripTych domain analysis & description process we decompose it into steps. Each step "handles" a sort p:P or a material m:M. Steps handling discovery of composite sorts generate a set of sort names ηP$_1$, ηP$_2$, ..., ηP$_n$ and ηM$_1$, ηM$_2$, ..., ηM$_n$. These are put in a reservoir for sorts to be inspected. The handled sort ηP or ηM is removed from that reservoir. Handling of material sorts concerns only their attributes. Each domain description prompt results in domain specification text (here we show only the formal texts) being deposited in the domain description reservoir, a global variable τ. The clause: domain_description_prompt(p) : τ := $\tau \oplus$ ["text ; "] means that the formal text "text ; " is joined to the global variable τ where that "text ; " is prompted by domain_description_prompt(p). The meaning of $\oplus$ will be discussed at the end of this section.

3.1.3 Initialising the Domain Analysis & Description Process

We remind the reader that we are dealing only with endurant domain entities. The domain analysis approach covered in Chap. 1 was based on decomposing an understanding of a domain from the "overall domain" into its components, and these, if not atomic, into their subcomponents. So we need to initialise the domain analysis & description by selecting (or choosing) the domain Δ.

Here is how we think of that "initialisation" process. The domain analyser & describer spends some time focusing on the domain, maybe at the "white board"[2], rambling, perhaps in an un-structured manner, across its domain, Δ, and its subdomains. Informally jotting down more-or-less final sort names, building, in the domain analysers' & describers' mind an image of that domain. After some time, doing this, the domain analyser & describer is ready. An image of the domain is in the form of "a domain" endurant, δ:Δ. Those are the quantities, $\eta\Delta$ (name of Δ) [Item 112] and ιp:P (for (δ:Δ)) [Item 119 on the following page], referred to below.

Thus this initialisation process is truly a creative one.

3.1.4 A Domain Analysis & Description State

112 A global variable αps will accumulate all the sort names being discovered.

113 A global variable νps will hold names of sorts yet to be analysed and described.

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

variable
112. αps := [$\eta\Delta$] ηP-**set** or ηP*
113. νps := [$\eta\Delta$] (ηP|ηM)-**set** or (ηP|ηM)*
114. τ := [] **Text-set** or **Text***

We shall explain the use of [...]s and the operations of $\setminus$ and $\oplus$ on the above variables in Sect. 3.1.6 on Page 98.

3.1.5 Analysis & Description of Endurants

115 To analyse and describe endurants means to first

116 examine those endurant which have yet to be so analysed and described

[2]Here 'white board' is a conceptual notion. It could be physical, it could be yellow "post-it" stickers, or it could be an electronic conference "gadget".

117 by selecting and removing from νps (Item 122.) an as yet unexamined sort (by name);

118 then analyse and describe an endurant entity (ιp:P) of that sort — this analysis, when applied to composite parts, leads to the insertion of zero[3] or more sort names[4];

119 then to analyse and describe the mereology of each part sort,

120 and finally to analyse and describe the attributes of each sort.

```
value
115.  analyse_and_describe_endurants: Unit → Unit
115.  analyse_and_describe_endurants() ≡
116.    while ∼is_empty(νps) do
117.      let ηS = select_and_remove_ηS() in
118.      analyse_and_describe_endurant_sort(ιs:S) end end ;
119.    for all ηP • ηP ∈ αps do analyse_and_describe_mereology(ιp:P) end
120.    for all ηP • ηP ∈ αps do analyse_and_describe_attributes(ιp:P) end
```

The ι of Items 118, 119 and 120 are crucial. The domain analyser is focused on sort S (and P) and is "directed" (by those items) to choose (select) an endurant ιs (ιp) of that sort. The ability of the domain analyser to find such an entity is a measure of that person's professional creativity.

As was indicated in Chap. 1, the mereology of a part may involve unique identifiers of any part sort, hence must be done after all such part sort unique identifiers have been identified. Similarly for attributes which also may involve unique identifiers. Each iteration of analyse_and_describe_endurant_sort(ιp:P) involves the selection of a sort (by name) (which is that of either a part sort or a material sort) with this sort name then being removed.

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

122 The affected global state component is that of the reservoir, νps.

```
value
121.  select_and_remove_ηS: Unit → ηP
121.  select_and_remove_ηS() ≡
122.    let ηS • ηS ∈ νps in νps := νps \ {ηS} ; ηS end
```

The analysis and description of all sorts also performs an analysis and description of their possible unique identifiers (if part sorts) and attributes. The analysis and description of sort mereologies potentially requires the unique identifiers of any set of sorts. Therefore the analysis and description of sort mereologies follows that of analysis and description of all sorts.

123 To analyse and describe an endurant

124 is to find out whether it is a part.

125 If so then it is to analyse and describe it as a part,

126 else it is to analyse and describe it as a material.

[3] If the sub-parts of p are all either atomic or already analysed, then no new sort names are added to the repository νps.
[4] These new sort names are then "picked-up" for sort analysis &c. in a next iteration of the while loop.

```
123.  analyse_and_describe_endurant_sort: (P|M) → Unit
123.  analyse_and_describe_endurant_sort(e:(P|M)) ≡
124.    if is_part(e)
124.        assert: is_part(e) ≡ is_endurant(e)∧is_discrete(e)
125.      then analyse_and_describe_part_sort(e:P)
126.      else analyse_and_describe_material_parts(e:M)
123.    end
```

Analysis & Description of Part Sorts

127 The analysis and description of a part sort

128 is based on there being a set, ps, of parts[5] to analyse —

129 of which an archetypal one, p′, is arbitrarily selected.

130 analyse and describe part p′

```
127.  analyse_and_describe_part_sort: P → Unit
127.  analyse_and_describe_part_sort(p:P) ≡
128.    let ps = observe_parts(p) in
129.    let p′:P • p′ ∈ ps in
130.    analyse_and_describe_part(p′)
127.    end end
```

131 The analysis (&c.) of a part

132 first analyses and describes its unique identifiers.

133 If atomic

134 and

135 if the part embodies materials,

136 we analyse and describe these.

137 If not atomic then the part is composite

138 and is analysed and described as such.

```
131.  analyse_and_describe_part: P → Unit
131.  analyse_and_describe_part(p) ≡
132.    analyse_and_describe_unique_identifier(p) ;
133.    if is_atomic(p)
134.      then
135.          if has_materials(p)
136.              then analyse_and_describe_part_materials(p) end
137.      else assert: is_composite(p)
138.          analyse_and_describe_composite_endurant(p) end
131.    pre: is_discrete(p)
```

We do not associate materials with composite parts.

[5] We can assume that there is at least one element of that set. For the case that the sort being analysed is a domain Δ, say *"The Transport Domain"*, p′ is some representative *"transport domain"* δ. Similarly for any other sort for which ps is now one of the sorts of δ.

Analysis & Description of Part Materials

139 The analysis and description of the material part sorts, one or more, of atomic parts p of sort P containing such materials,

140 simply observes the material sorts of p,

141 that is, generates the one or more continuous endurants

142 and the corresponding observer function text.

143 The reservoir of sorts to be inspected is augmented by the material sorts — except if already previously entered (the $\setminus$ α**ps** clause).

```
139.  analyse_and_describe_part_materials: P → Unit
139.  analyse_and_describe_part_materials(p) ≡
140.      observe_material_sorts(p) :
141.      τ := τ ⊕ ["type M1,M2,...,Mm;
142.                value obs_part_M1:P→M1,obs_part_M2:P→M2,...,obs_part_Mm:P→Mm;"]
143.      νps := νps ⊕ ([ M1,M2,...,Mm ] \ αps)
139.      pre: has_materials(p)
```

Analysis & Description of Material Parts

144 To analyse and describe materials, *m*, i.e., continuous endurants,

145 is only necessary if *m* has parts.

146 Then we observe the sorts of these parts.

147 The identified part sort names update both name reservoirs.

```
144.  analyse_and_describe_material_parts: M → Unit
144.  analyse_and_describe_material_parts(m:M) ≡
145.    if has_parts(m)
146.      then observe_part_sorts(m):
146.          τ := τ ⊕ [" type  P1,P2,...,PN ;
146.                   value  obs_part_Pi: M→Pi i:{1..N};"]
147.        ‖ νps := νps ⊕ ([ ηP1,ηP2,...,ηPN ]\ αps)
147.        ‖ αps := αps ⊕ [ ηP1,ηP2,...,ηPN ]
144.      end
144.    assert: is_continuous(m)
```

Analysis & Description of Composite Endurants

148 To analyse and describe a composite endurant of sort P

149 is to analyse and describe the unique identifier of that composite endurant,

150 then to analyse and describe the sort. If the sort has a concrete type

151 then we analyse and describe that concrete sort type

152 else we analyse and describe the abstract sort.

```
148.  analyse_and_describe_composite_endurant: P → Unit
148.  analyse_and_describe_composite_endurant(p) ≡
149.    analyse_and_describe_unique_identifier(p) ;
150.    if has_concrete_type(p)
151.      then analyse_and_describe_concrete_sort(p)
152.      else analyse_and_describe_abstract_sort(p)
150.    end
```

Analysis & Description of Concrete Sort Types

153 The concrete sort type being analysed and described

154 is either

155 expressible by some compound type expression

154. or is

156 expressible by some alternative type expression.

```
153.  analyse_and_describe_concrete_sort: P → Unit
153.  analyse_and_describe_concrete_sort(p:P) ≡
155.    analyse_and_describe_concrete_compound_type(p)
154.    ⌈⌉
156.    analyse_and_describe_concrete_alternative_type(p)
153.    pre: has_concrete_type(p)
```

157 The concrete compound sort type

158 is expressible by some simple type expression, T=$\mathscr{E}$(Q,R,...,S) over either concrete types or existing or new sorts Q, R, ..., S.

159 The emerging sort types are identified

160 and assigned to both νps

161 and αps.

```
155.  analyse_and_describe_concrete_compound_type: P → Unit
155.  analyse_and_describe_concrete_compound_type(p:P) ≡
157.    observe_part_type(p):
157.      τ := τ ⊕ [”type  Q,R,..,S, T = 𝒞(Q,R,...,S);
157.                value obs_part_T: P → T ;”] ;
158.    let {Pa,Pb,...,Pc} = sorts_of({Q,R,...,S})
159.          assert: {Pa,Pb,...,Pc} ⊆ {Q,R,...,S} in
160.    νps := νps ⊕ [ηPa, ηPb, ..., ηPc] ‖
161.    αps := αps ⊕ ([ηPa, ηPb, ..., ηPc] \ αps) end
155.    pre: has_concrete_type(p)
```

162 The concrete alternative sort type expression

163 is expressible by an alternative type expression T=P1|P2|...|PN where each of the alternative types is made disjoint wrt. existing types by means of the description language Pi::mkPi(s$_u$:P$_i$) construction.

164 The emerging sort types are identified and assigned

165 to both νps

166 and αps.

```
156.  analyse_and_describe_concrete_alternative_type: P → Unit
156.  analyse_and_describe_concrete_alternative_type(p:P) ≡
162.    observe_part_type(p):
163.      τ := τ ⊕ [”type  T=P1 | P2 | ... | PN, Pi::mkPi(s_u:P_i) (1≤i≤N);
163.                value obs_part_T: P→T ;”] ;
164.    let {P_a,P_b,...,P_c} = sorts_of({P_i|1≤i≤n})
164.        assert: {P_a,P_b,...,P_c} ⊆ {P_i|1≤i≤n} in
165.    νps := νps ⊕ ([ηP_a, ηP_b, ..., ηP_c] \ αps) ‖
166.    αps := αps ⊕ [ηP_a, ηP_b, ..., ηP_c] end
153.    pre: has_concrete_type(p)
```

Analysis & Description of Abstract Sorts

167 To analyse and describe an abstract sort

168 amounts to observe part sorts and to

169 update the sort name repositories.

```
167.  analyse_and_describe_abstract_sort: P → Unit
167.  analyse_and_describe_abstract_sort(p:P) ≡
168.    observe_part_sorts(p):
168.      τ := τ ⊕ [”type P_1, P_2, ..., P_n;
168.                value obs_part_P_i:P→P_i (0≤i≤n);”]
169.    ‖ νps := νps ⊕ ([ηP_1, ηP_2, ..., ηP_n] \ αps)
169.    ‖ αps := αps ⊕ [ηP_1, ηP_2, ..., ηP_n]
```

Analysis & Description of Unique Identifiers

170 To analyse and describe the unique identifier of parts of sort P is

171 to observe the unique identifier of parts of sort P

172 where we assume that all parts have unique identifiers.

```
170.  analyse_and_describe_unique_identifier: P → Unit
170.  analyse_and_describe_unique_identifier(p) ≡
171.    observe_unique_identifier(p):
171.      τ := τ ⊕ [”type PI; value uid_P:P→PI;”]
172.    assert: has_unique_identifier(p)
```

Analysis & Description of Mereologies

173 To analyse and describe a part mereology

174 if it has one

175 amounts to observe that mereology

176 and otherwise do nothing.

177 The analysed quantity must be a part.

```
173. analyse_and_describe_mereology: P → Unit
173. analyse_and_describe_mereology(p) ≡
174.   if has_mereology(p)
175.     then observe_mereology(p) :
175.           τ := τ ⊕ "type MT = 𝒞(PI_a,PI_b,...,PI_c) ;
175.                    value obs_mereo_P: P→MT ;"
176.     else skip end
173.   pre: is_part(p)
```

Analysis & Description of Part Attributes

178 To analyse and describe the attributes of parts of sort P is

179 to observe the attributes of parts of sort P

180 where we assume that all parts have attributes.

```
178.  analyse_and_describe_part_attributes: P → Unit
178.  analyse_and_describe_part_attributes(p) ≡
179.    observe_attributes(p):
179.       τ := τ ⊕ ["type  A_1, ..., A_m ;
179.                 value attr_A_1:P→A_1,,...,attr_A_m:P→A_m;"]
180.    assert: has_attributes(p)
```

3.1.6 Discussion of The Model

The above model lacks a formal understanding of the individual prompts as listed in Sect. 3.1.1. Such an understanding is attempted in [35].

Termination

The sort name reservoir *vps* is "reduced" by one name in each iteration of the **while** loop of the analyse_and_describe_endurants, cf. Item 117 on Page 92, and is augmented, in each iteration of that loop, by sort names – if not already dispensed of iterations of in earlier itetrations, cf. formula Items 143 on Page 94, 147 on Page 94, 160 on Page 95, 165 on the facing page and 160 on Page 95. We take it as a dogma that domains contain a finite number of differently typed parts and matyerials. This introduction and removal of sort names and the finiteness of sort names is then the basis for a proper proof of terminaton of the the analysis & description process.

Axioms and Proof Obligations

We have omitted from the above treatment of axioms concerning well-formedness of parts, materials and attributes and proof obligations concerning disjointness of observed part and material sorts and attribute types. A more proper treatment would entail adding a line of proof obligation text right after Item lines 176 on the preceding page and 179 on the previous page, and of axiom text right after Item lines 142, 146, 157, 159, 171, 179, No axiom is needed in connection with Item line 163 on Page 96.

[36] covers axioms and proof obligations in some detail.

Order of Analysis & Description: A Meaning of '$\oplus$'

The variables αps, vps and τ are defined to hold either sets or lists. The operator $\oplus$ can be thought of as either set union ($\cup$ and [,]≡{,}) — in which case the domain description text in τ is a set of domain description texts or as list concatenation ($\frown$ and [,]≡⟨,⟩) of domain description texts. The operator $\ell_1 \oplus \ell_2$ now has at least two interpretations: either $\ell_1 \frown \ell_2$ or $\ell_2 \frown \ell_1$. In the case of lists the $\oplus$ (i.e., $\frown$) does not (suffix or prefix) append ℓ_2 elements already in ℓ_1. The select_and_remove_ηP function on Page 92 applies to the set interpretation. A list interpretation is:

```
value
117.  select_and_remove_ηP: Unit → ηP
117.  select_and_remove_ηP() ≡
117.    let ηP = hd vps in  vps := tl vps; ηP end
```

In the first case ($\ell_1 \frown \ell_2$) the analysis and description process proceeds from the root, breadth first, In the second case ($\ell_2 \frown \ell_1$) the analysis and description process proceeds from the root, depth first.

3.2 A Model of The Analysis & Description Prompts

TO BE WRITTEN

3.2.1 On the Domain Analyser's Image of Domains

TO BE WRITTEN

3.2.2 An Abstract Syntax of Domains

Domain Nodes

The core quantity of the domain analyser's image of domains is here called a node. Nodes designate entities. Some designated nodes are so-called duplicate nodes. A **duplicate node** designates a sort name that is defined elsewhere in the domain description tree. See Sect. 3.2.2 on Page 100. Five kinds of nodes are said to define sorts. Atomic nodes, Item 185 Page 99, define some qualities, Q (cf. Item 192 Page 99), and may refer to material nodes (either directly or by reference (#Mn)). Material nodes, Item 186 Page 99, define material qualities, MAT, Item 192 on the facing page, and may refer to part nodes (either directly or by reference (#Pn)). Composite nodes, Item 188 Page 99, define define some qualities and a set of entity nodes (either directly or by reference (#Sn)). Type nodes, Item 190 Page 99, define define some qualities, a concrete type, and a set of part nodes (either directly or by reference (#Pn)) — where their part names occur in the type expression. Alternative nodes, Item 191 Page 99, define define some qualities and a set of [alternative] part nodes (either directly or by reference (#Pn)) A more tersely expressed narrative and the a;ready reference formalisation follows.

181 A node is either an endurant or a perdurant node.

182 An endurant node is either a discrete (i.e., a part) or a continuous (i.e., a material) node.

183 We shall not consider perdurant nodes in this paper.

184 A part node is either an atomic node or a composite node or a duplicate node (represented as [#Sn]).

185 An atomic node contains some quality description and a usually empty set of uniquely material-named material nodes, some may be duplicate such nodes.

186 A material node contains some material attributes and a possibly empty set of uniquely part-named part nodes.

187 A composite type node is either a compound node or is a [concrete] type node.

188 A composite node contains a set of uniquely sort-named nodes and some quality description.

189 A "k"oncrete type node is either a simple compound type node or is an alternative sorts node.

190 A type node is some type expression over sorts and concrete type names and a possibly empty set of uniquely part-named endurants and some quality description.

191 An alternative node is a set of two or more uniquely named endurant nodes.

192 A quality description consists of a unique identifier, a possibly "empty" mereology, and some attributes.

type
181. N == mkEnN(EnN) | mkPeN(PeN)[6]
182. EnN == mkPaN(PaN) | mkMaN(MaN)
183. PeN
184. PaN == mkAtN(AtN) | mkCoN(CoN)
185. **AtN** = Q × ((Mn $\xrightarrow[m]{}$ MaN)[7] × [#Mn]-**set**)
186. **MaN** = MAT × ((Pn $\xrightarrow[m]{}$ PaN) × [#Pn]-**set**)
187. CoN == mkCmN(CmN)| mkKTN(KTN)
188. **CmN** = Q × ((Sn $\xrightarrow[m]{}$ EnN)[8] × [#Sn]-**set**)
189. KTN == mkTyN(st:TyN) | mkAlT(AlT)
190. **TyN** = Q × TE × ((Pn $\xrightarrow[m]{}$ PaN)[9] × [#Pn]-**set**)
191. **AlT** = Q × ((Pn $\xrightarrow[m]{}$ PaN)[10] × [#Pn]-**set**)
192. Q = UI × ME × PAT

Sn=Pn|Mn[11], Pn, Mn, Tn, TE, UI, MAT, PAT

Part and material names, Pn, Mn, type expressions, TE, and unique identifiers, UI, are further undefined quantities. We shall define mereology and material and part attributes, MAT and PAT, below.

[6] The type equation A=mkB(...)|mkC(...)|...|mkD(...) defines A to consist of the disjoint types designated by mkB(...), mkC(...), ... and mkD(...). In mkE(s:E) s denotes a selector function. mkB, etc., are called constructor functions. s(mkX(x))≡. isX(mkX(x))≡**true**. isX, etc., are discriminator predicates.

[7] Empty map when part "carries" no materials. Usually a singleton map if it does carry materials.

[8] The sort names are those sort names of the type expression which are being defined here (i.e., "appear" for the first time).

[9] See footnote 8.

[10] At least two map elements

[11] The part name and the material name types are disjoint, that is $\mathscr{M}$(Pn)∩$\mathscr{M}$(Mn)

The Root Domain Node

193 The root domain node is a singleton map from a sort name to an endurant node.

type
193. RDN = Sn $\xrightarrow{m}$ EnN **axiom** ∀ rdn:RDN • **card dom** rdn = 1
value
193. rdn:RDN = [sn↦mkEnN(en)]
193. sn:Sn, en:EnN

Domain Description Trees

Due to the recursive definition of sort nodes the abstract syntax can be visualised as defining **description trees** ■Endurant nodes of a part node and part nodes of a material node can be said to designate subtrees. We can therefore define a notion of **description tree path**s.

Syntax

194 A description [tree] path is a sequence of one or more sort names.

type
194. DP = (Sn|[#Sn])*
axiom
194. ∀ dtp:DTP •
194. dtp≠⟨⟩
194. ∧ ∃ sn:Sn•[#sn]∈ **elems** dtp ⇒
194. [#sn] ∉ **elems fst**[12]dtp

Generating Description Tree Paths

195

196

197

198

199

200

201

202

203

[12] **fst** list is the list of all but the last element of list.

```
value
195.  G: EnN → DP-infset
195.  G(en) ≡
196.    case en of
198.      mkPaN(mkAtN(_,m)) → G(m),
199.      mkPaN(mkMaN(_,m)) → G(m),
200.      mkPaN(mkCoN(mkCmN(_,m))) → G(m),
201.      mkPaN(mkCoN(mkKTN(mkTyN(_,_,m)))) → G(m),
202.      mkPaN(mkCoN(mkKTN(mkAlT(_,m)))) → G(m),
203.      mkMaN(_,m) → G(m)
195.    end ∪ {⟨⟩}
```

The Generate path function is overload-defined, four variants the above and the three below:

204 for the root node,

205 for the six node-defining nodes, and

206 their duplicate components.

```
204.  G: RDN → DP-infset
204.  G([n↦en]) ≡ {⟨n⟩^dp|dp:DP•n′ ∈ dom rdn∧dp ∈ G(en)}

205.  G: Nodes × Duplicates → DP-infset
205.  G(ns,ds) ≡
205.    {⟨n⟩^dp|n:Sn,dp:DP•n ∈ dom ns∧dp ∈ G(ns(n))∪ G(ds)}

206.  G: Duplicates → DP-set
206.  G(ds) ≡ {⟨[#sn]⟩|[#sn] ∈ ds}
```

Well-formedness of Domain Nodes

Well-formed Composite and Material Nodes

207 Composite nodes must contain at least one sort node;

208 material nodes may contain no part nodes.

```
value
207.  wf_comp_nds: CmN → Bool
207.  wf_comp_nds(_,(sm,ss)) ≡ dom sm ∩ Sns(ss) ≠ {}

207.  Sns: [#Sn]-set −< Sn-set
207.  Sns(msn) ≡ {sn|sn:Sn•[#sn]∈ msn}

208.  wf_mat_nds: MaN ×[#Pn]-set →Bool
208.  wf_mat_nds(_,_) ≡ true
```

No Recursively Defined Sorts Sorts, whether parts sorts or material sorts, cannot be recursively defined.

209 A sort is recursively defined if its name occur more than once on any path,

210 by properties of lists and sets this is tantamount to saying that the length of a violating path is higher than the cardinality of the set of all names in the path.

```
209.  no_recursively_defined_sorts: DPN → Bool
209.  no_recursively_defined_sorts(dpn) ≡
209.    let paths = G(dpn) in
209.    ∼∃ path:DP • path ∈ paths
210.       ⇒ len path > card elems path end
```

No Duplicate Definitions Once a part node has been defined it cannot be redefined. The purpose of the "#" "blocks" in the part node maps is to mark where a sort name would otherwise be redefined.

211 We check for duplicate definitions across domain description nodes.

212 Let dps be the set of all root-to-leaf paths of a domain tree.

213 There does not exist two paths dp′,dp″ in dps

214 with distinct prefixes

215 such that there exists a common sort name, sn, which when appended to dp′,dp″ is a path in the domain.

216 A path dp′ is a prefix of a peth dp if the exists a path dp″ when when appended to dp′ becomes dp.

```
211.  no_duplicate_definitions: DPN → Bool
211.  no_duplicate_definitions(dpn) ≡
212.    let dps = G(dpn) in
213.    ∼ ∃ dp′,dp″:DP • {dp′,dp″}⊆dps ⇒
214.       let pdps′=prefixes(dp′),pdps″=prefixes(dp″) in
214.       ∃ pdp′,pdp″:DP•pdp′ ∈ pdps′∧pdp″ ∈ pdps″ ⇒ pdp′≠pdp″
215.          ⇒ ∃ sn:SN•pdp′⌢⟨sn⟩∈ pdps′∧pdp″⌢⟨sn⟩∈ pdps″
211.    end end

216.  prefixes: DP → DP-set
216.  prefixes(dp) ≡ {dp′|dp′,dp″:DP•dp′⌢dp″ = dp}
```

Defined Duplicate Sort Names

217 defined_sorts is defined over root domain nodes.

218 If, for some sort name, sn:Sn, and some domain tree path, dtp, of a domain rdn:RDN, the last elements of dtp is [#sn] then sn must be defined elsewhere in the domain tree of rdn:RDN.

219 In this case there must exist a unique another path, dtp′,

220 such that sn is properly defined withing dtp′.

```
value
217.  defined_sorts: RDN → Bool
217.  defined_sorts([sn↦en]) ≡
217.    let dtps = paths(en) in
218.    if ∃ dtp:DTP,sn:Sn•dtp^⟨[#sn]⟩∈ dtps
219.      then ∃![13]dtp′:DTP•dtp′isin dtps
220.           sn ∈ elems fst dtp′ else skip end end
```

3.2.3 Node Selection

221

222

223

224

225

```
221. select_node: DP × DPN ~→ EnN
221. select_node(dp)([sn↦en]) ≡
222.   case dp of
222.     ⟨[#sn]⟩ → select_node([#sn])(en),
222.     ⟨sn⟩ → en,
222.     ⟨sn′⟩^dp′ →
196.       case en of
198.         mkPaN(mkAtN(_,[sn′↦en′]∪m)) → select_node(dp′)([sn′↦en′]),
199.         mkPaN(mkMaN([sn′↦en′]∪m,_)) → select_node(dp′)([sn′↦en′]),
200.         mkPaN(mkCoN(mkSCN([sn′↦en′]∪m,_))) → select_node(dp′)([sn′↦en′]),
201.         mkPaN(mkCoN(mkKTN(mkCmpT(_,[sn′↦en′]∪m,_))))
198.           → select_node(dp′)([sn′↦en′]),
202.         mkPaN(mkCoN(mkKTN(mkAlT([sn′↦en′]∪m,_))))
198.           → select_node(dp′)([sn′↦en′]),
203.         mkMaN([sn′↦en′]∪m,_) → select_node(dp′)([sn′↦en′]),
194.         _ → chaos
196.       end
196.     _ → chaos end
```

```
222.  select_node: {|[#sn]|} → DPN → EnN
222.  select_node([#sn])([sn↦en]) ≡
222.    let dp:DP • dp ∈ paths([sn↦en])
222.                ∧ ∃ i:Nat•i ∈ inds dp\{len dp}∧dp(i)=sn in
222.    select_node(dp)([sn↦en]) end
```

3.2.4 Index of Prompts

In Chap. 1 we motivated and briefly explained a number of domain analysis and domain description prompts.

[13] ∃!... reads: there exists a unique ...

Analysis Prompts

a. is_ entity, 23
b. is_ endurant, 24
c. is_ perdurant, 24
d. is_ discrete, 24
e. is_ continuous, 24
f. is_ part, 25
g. is_ component, 25
h. is_ material, 26
i. is_ atomic, 26
j. is_ composite, 26
k. observe_ parts, 27
l. has_ concrete_ type, 29
m. has_ mereology, 34
n. attribute_ names, 38
o. has_ components, 44
p. has_ materials, 45

MORE TO COME

Description Prompts

a. **obs_ part_** P, 28
b. **is_** P, 28
c. **obs_ part_** T, 29
d. **uid_ P**, 33
e. **mereo_ P**, 35
f. **upd_ mereology**, 36
g. **attr_** A, 38
h. **components**, 44
i. **obs_ part_** M, 46
j. **materials**, 46

MORE TO COME

•••

We shall now present a formal description of a meaning for these prompts.

3.2.5 A Formal Description of a Meaning of Prompts

MORE TO COME

The Iterative Nature of The Description Process

Assume that the domain analysers cum describers are analysing & describing a particular endurant; that is, as we shall understand it, are examining a given endurant node in the domain description tree !

To make this claim: *the domain analysers cum describers are examining a given endurant node in the domain description tree* amounts to saying that *the domain analysers cum describers have in their mind a reasonably "stable" "picture" of a domain in terms of a domain description tree.*

We need explain this assumption. In this assumption there is "buried" an understanding that the domain analysers cum describers during the — what we can call "the final" — domain analysis & description process, that leads to a "deliverable" domain description, are not investigating the domain to be described for the first time. That is, we certainly assume that any "final" domain analysis & description process has been preceded by a number of iterations of "trial" domain analysis & description processes.

Hopefully this iteration of experimental domain analysis & description processes converges. Each iteration leads to some domain description, that is, some domain description tree. A first iteration is thus based on a rather incomplete domain description tree which, however, "quickly" emerges into a less incomplete one in that first iteration. When the domain analysers cum describers decide that a "final" iteration seems possible then a "final" description emerges If acceptable, OK, otherwise yet an "final" iteration must be performed. Common to all iterations is that the domain analysers cum describers have in mind some more-or-less "complete" domain description tree and apply the prompts introduces in Chap. 1.

How Are We Modelling the Prompts

TO BE WRITTEN

The Model

TO BE WRITTEN

3.2.6 Discussion

TO BE WRITTEN

3.3 Discussion of the Models

Chapter 4

Domains: Their Simulation, Monitoring and Control

Abstract

This divertimento – on the occasion of the 70th anniversary of Prof., Dr Hermann Maurer – sketches some observations over the concepts of domain, requirements and modelling – where abstract interpretations of these models cover both a priori, a posteriori and real-time aspects of the domain as well as 1–1, microscopic and macroscopic simulations, real-time monitoring and real-time monitoring & control of that domain. The reference frame for these concepts are domain models: carefully narrated and formally described domains. I survey more-or-less standard ideas of verifiable development and conjecture product families of demos, simulators, monitors and monitors & controllers – but now these "standard ideas" are recast in the context of core requirements prescriptions being "derived" from domain descriptions.

4.1 Introduction

A background setting for this paper is the concern for professionally developing the right software, i.e., software which satisfies users expectations, and software that is right: i.e., software which is correct with respect to user requirements and thus has no "bugs", no "blue screens".

The present paper must be seen on the background of the main line of experimental research around the topics of domain engineering, requirements engineering and their relation. For details I refer to [24, Chaps. 9–16: Domain Engineering, Chaps. 17-24: Requirements Engineering].

The aims of this paper is to present (1) some ideas about software that (1a) "demo", (1b) simulate, (1c) monitor and (1d) monitor & control domains; (2) some ideas about "time scaling": demo and simulation time versus domain time; and (3) how these kinds of software relate.

The paper is exploratory. There will be no theorems and therefore there will be no proofs. We are presenting what might eventually emerge into (α) a theory of domains, i.e., a domain science [25, 47, 27, 33], and (β) a software development theory of domain engineering versus requirements engineering [32, 26, 28, 31].

The paper is not a "standard" research paper: it does not compare its claimed achievements with corresponding or related achievements of other researchers – simply because we do not claim "achievements" which have been fully, or at least reasonably well theorised – etcetera. But I would suggest that you might find some of the ideas of the paper (in Sect. 4.3) worthwhile publishing. Hence the "divertimento" suffix to the paper title.

The structure of the paper is as follows.

In Sect. 4.3 we then outline a series of interpretations of domain descriptions. These arise, when developed in an orderly, professional manner, from requirements prescriptions which are themselves orderly developed from the domain description[1]. The essence of Sect. 4.3 is (i) the (albeit informal) presentation of such tightly related notions as *demos* (Sect. 4.3.1), *simulators* (Sect. 4.3.2), *monitors* (Sect. 4.3.3) and *monitors & controllers* (Sect. 4.3.3) (these notions can be formalised), and (ii) the conjectures on a product family of domain-based software developments (Sect. 4.3.5). A notion of *script-based simulation* extends demos and is the basis for monitor and controller developments and uses. The script used in our example here is related to time, but one can define non-temporal scripts – so the "carrying idea" of Sect. 4.3 extends to a widest variety of software. We claim that Sect. 4.3 thus brings these new ideas: a tightly related software engineering concept of *demo-simulator-monitor-controller* machines, and an extended notion of *reference models for requirements and specifications* [88].

4.2 Domain Descriptions

By a domain description we shall mean a combined narrative, that is, precise, but informal, and a formal description of the application domain ***as it is:*** no reference to any possible requirements let alone software that is desired for that domain. (Thus a requirements prescription is a likewise combined narrative, that is, precise, but informal, and a formal prescription of what we expect from a machine (hardware + software) that is to support simple entities, actions, events and behaviours of a possibly business process re-engineering application domain. Requirements expresses a domain ***as we would like ti to be.***)

We bring in Chapter 6 an example domain description.

We further refer to the literature for examples: [109, *railways* (2000)], [19, *the 'market'* (2000)], [28, *public government, IT security, hospitals* (2006) chapters 8–10], [26, *transport nets* (2008)] and [31, *pipelines* (2010)]. On the net you may find technical reports covering "larger" domain descriptions. Recent papers on the concept of domain descriptions are [31, 33, 29, 47, 26, 25, 30].

To emphasize: domain descriptions describe domains as they are with no reference to (requirements to) possibly desired software. Domain descriptions do not necessarily describe computable objects. They relate to the described domain in a way similar to the way in which mathematical descriptions of physical phenomena stand to "the physical world".

4.3 Interpretations

4.3.1 What Is a Domain-based Demo?

A *domain-based demo* is a software system which *"present"* (1) simple entities, (2) actions, (3) events and (4) behaviours of a domain. The *"presentation"* abstracts these phenomena and their related concepts in various computer generated forms: visual, acoustic, etc.

Examples

A domain description might, as that of Appendix 6, be of transport nets (of hubs [street intersections, train stations, harbours, airports] and links [road segments, rail tracks, shipping lanes, air-lanes]), their development, traffic [of vehicles, trains, ships and aircraft], etc. We shall assume such a transport domain description below.

(1) Simple entities are, for example, presented as follows: (*a*) transport nets by two dimensional (2D) road, railway or airline maps, (*b*) hubs and links by highlighting parts of 2D maps and by related photos – and their unique identifiers by labelling hubs and links, (*c*) routes by highlighting sequences of paths (hubs and links) on a 2D map, (*d*) buses by photographs and by dots at hubs or on links of a 2D map, and (*e*) bus timetables by, well, indeed, by showing a 2D bus timetable.

[1] We do not show such orderly "derivations" but outline their basics in Sect. 4.3.4.

(2) Actions are, for example, presented as follows: (f) The insertion or removal of a hub or a link by showing "instantaneous" triplets of "before", "during" and "after" animation sequences. (g) The start or end of a bus ride by showing flashing animations of the appearance, respectively the flashing disappearance of a bus (dot) at the origin, respectively the destination bus stops.

(3) Events are, for example, presented as follows: (h) A mudslide [or fire in a road tunnel, or collapse of a bridge] along a (road) link by showing an animation of part of a (road) map with an instantaneous sequence of (α) the present link , (β) a gap somewhere on the link, (γ) and the appearance of two ("symbolic") hubs "on either side of the gap". (i) The congestion of road traffic "grinding to a halt" at, for example, a hub, by showing an animation of part of a (road) map with an instantaneous sequence of the massive accumulation of vehicle dots moving (instantaneously) from two or more links into a hub.

(4) Behaviours are, for example, presented as follows: (k) A bus tour: from its start, on time, or "thereabouts", from its bus stop of origin, via (all) intermediate stops, with or without delays or advances in times of arrivals and departures, to the bus stop of destination (ℓ) The composite behaviour of "all bus tours", meeting or missing connection times, with sporadic delays, with cancellation of some bus tours, etc. – by showing the sequence of states of all the buses on the net.

We say that behaviours (3(j)–4(ℓ)) are *script-based* in that they (try to) satisfy a bus timetable (1(e)).

Towards a Theory of Visualisation and Acoustic Manifestation

The above examples shall serve to highlight the general problem of visualisation and acoustic manifestation. Just as we need sciences of visualising scientific data and of diagrammatic logics, so ***we need more serious studies of visualisation and acoustic manifestation — so amply, but, this author thinks, inconsistently demonstrated by current uses of interactive computing media.***

4.3.2 Simulations

"Simulation is the imitation of some real thing, state of affairs, or process; the act of simulating something generally entails representing certain key characteristics or behaviours of a selected physical or abstract system" [Wikipedia] for the purposes of testing some hypotheses usually stated in terms of the model being simulated and pairs of statistical data and expected outcomes.

Explication of Figure 4.1

Figure 4.1 on the facing page attempts to indicate four things: (i) Left top: the rounded edge rectangle labelled "The Domain" alludes to some specific domain ("out there"). (ii) Left middle: the small rounded rectangle labelled "A Domain Description" alludes to some document which narrates and formalises a description of "the domain". (iii) Left bottom: the medium sized rectangle labelled "A Domain Demo based on the Domain Description" (for short "Demo") alludes to a software system that, in some sense (to be made clear later) "simulates" "The Domain." (iv) Right: the large rectangle (a) shows a horisontal time axis which basically "divides" that large rectangle into two parts: (b) Above the time axis the **"fat"** rounded edge rectangle alludes to the time-wise behaviour, a *domain trace*, of "The Domain" (i.e., the actual, or real, domain). (c) Below the time axis there are eight **"thin"** rectangles. These are labels S1, S2, S3, S4, S5, S6, S7 and S8. (d) Each of these denote a "run", i.e., a time-stamped "execution", a *program trace*, of the "Demo". Their "relationship" to the time axis is this: their execution takes place in the real time as related to that of "The Domain" behaviour.

A *trace* (whether a domain or a program execution trace) is a time-stamped sequence of states: domain states, respectively demo, simulator, monitor and monitor & control states.

From Fig. 4.1 on the next page and the above explication we can conclude that "executions" S4 and S5 each share exactly one time point, t, at which "The Domain" and "The Simulation" "share" time, that is, the time-stamped execution S4 and S5 reflect a "Simulation" state which at time t should reflect (some abstraction of) "The Domain" state.

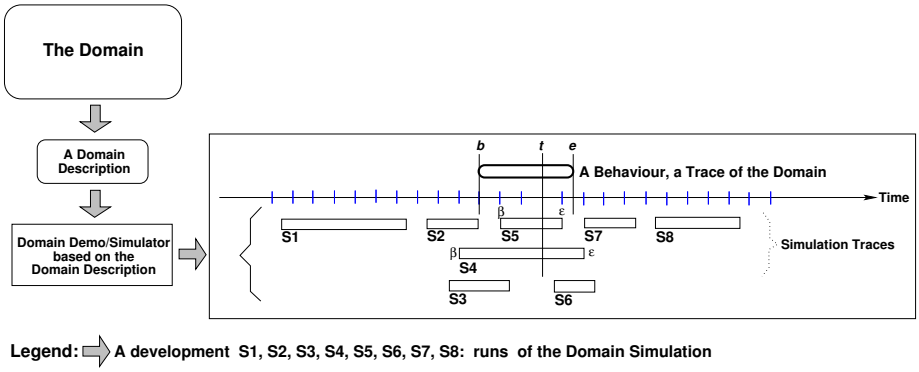

Figure 4.1: Simulations

Only if the domain behaviour (i.e., trace) fully "surrounds" that of the simulation trace, or, vice-versa (cf. Fig. 4.1[S4,S5]), is there a "shared" time. Only if the 'begin' and 'end' times of the domain behaviour are identical to the 'start' and 'finish' times of the simulation trace, is there an infinity of shared 1–1 times.

In Fig 4.2 on the following page we show "the same" "Domain Behaviour" (three times) and a (1) simulation, a (2) monitoring and a (3) monitoring & control, all of whose 'begin/start' (b/β) and 'end/finish' (e/ε) times coincide. In such cases the "Demo/Simulation" takes place in real-time throughout the 'begin$\cdots\cdots$end' interval.

Let β and ε be the 'start' and 'finish' times of either S4 or S5. Then the relationship between t, β, ε, b and e is $\frac{t-\mathsf{b}}{\mathsf{e}\text{-}t} = \frac{t-\beta}{\varepsilon-t}$ — which leads to a second degree polynomial in t which can then be solved in the usual, high school manner.

Script-based Simulation

A script-based simulation is the behaviour, i.e., an execution, of, basically, a demo which, step-by-step, follows a script: that is a prescription for highlighting simple entities, actions, events and behaviours.

Script-based simulations where the script embodies a notion of time, like a bus timetable, and unlike a route, can be thought of as the execution of a demos where "chunks" of demo operations take place in accordance with "chunks"[2] of script prescriptions. The latter (i.e., the script prescriptions) can be said to represent simulated (i.e., domain) time in contrast to "actual computer" time. The actual times in which the script-based simulation takes place relate to domain times as shown in Simulations S1 to S8 in Fig. 4.1 and in Fig. 4.2 on the following page(1–3). Traces Fig. 4.2(1–3) and S8 Fig. 4.1 are said to be *real-time:* there is a one-to-one mapping between computer time and domain time. S1 and S4 Fig. 4.1 are said to be *microscopic:* disjoint computer time intervals map into distinct domain times. S2, S3, S5, S6 and S7 are said to be *macroscopic:* disjoint domain time intervals map into distinct computer times.

In order to concretise the above "vague" statements let us take the example of simulating bus traffic as based on a bus timetable script. A simulation scenario could be as follows. Initially, not relating to any domain time, the simulation "demos" a net, available buses and a bus timetable. The person(s) who are requesting the simulation are asked to decide on the ratio of the domain time interval to simulation time interval. If the ratio is 1 a real-time simulation has been requested. If the ratio is less than 1 a microscopic simulation has been requested. If the ratio is larger than 1 a microscopic simulation has been requested. A chosen ratio of, say 48 to 1 means that a 24 hour bus traffic is to be simulated in 30 minutes of elapsed simulation time. Then the person(s) who are requesting the simulation are asked to decide on the starting domain time, say 6:00am, and the domain time interval of simulation, say 4 hours – in which case the simulation of bus traffic from 6am till 10am is to be shown in 5 minutes (300 seconds) of elapsed

[2] We deliberately leave the notion of chunk vague so as to allow as wide an spectrum of simulations.

simulation time. The person(s) who are requesting the simulation are then asked to decide on the *"sampling times"* or *"time intervals"*: If *'sampling times'* 6:00 am, 6:30 am, 7:00 am, 8:00 am, 9:00 am, 9:30 am and 10:00 am are chosen, then the simulation is stopped at corresponding simulation times: 0 sec., 37.5 sec., 75 sec., 150 sec., 225 sec., 262.5 sec. and 300 sec. The simulation then shows the state of selected entities and actions at these domain times. If *'sampling time interval'* is chosen and is set to every 5 min., then the simulation shows the state of selected entities and actions at corresponding domain times. The simulation is resumed when the person(s) who are requesting the simulation so indicates, say by a "resume" icon click. The time interval between adjacent simulation stops and resumptions contribute with 0 time to elapsed simulation time – which in this case was set to 5 minutes. Finally the requestor provides some statistical data such as numbers of potential and actual bus passengers, etc.

Then two clocks are started: a domain time clock and a simulation time clock. The simulation proceeds as driven by, in this case, the bus time table. To include "unforeseen" events, such as the wreckage of a bus (which is then unable to complete a bus tour), we allow any number of such events to be randomly scheduled. Actually scheduled events "interrupts" the "programmed" simulation and leads to thus unscheduled stops (and resumptions) where the unscheduled stop now focuses on showing the event.

The Development Arrow

The arrow, ⇨, between a pair of boxes (of Fig. 4.1 on the previous page) denote a step of development: (i) from the domain box to the domain description box it denotes the development of a domain description based on studies and analyses of the domain; (ii) from the domain description box to the domain demo box it denotes the development of a software system — where that development assumes an intermediate requirements box which has not been show; (iii) from the domain demo box to either of a simulation traces it denotes the development of a simulator as the related demo software system, again depending on whichever special requirements have been put to the simulator.

4.3.3 Monitoring & Control

Figure 4.2 shows three different kinds of uses of software systems (where (2) [Monitoring] and (3) [Monitoring & Control] represent further) developments from the demo or simulation software system mentioned in Sect. 4.3.1 and Sect. 4.3.2 on the previous page.

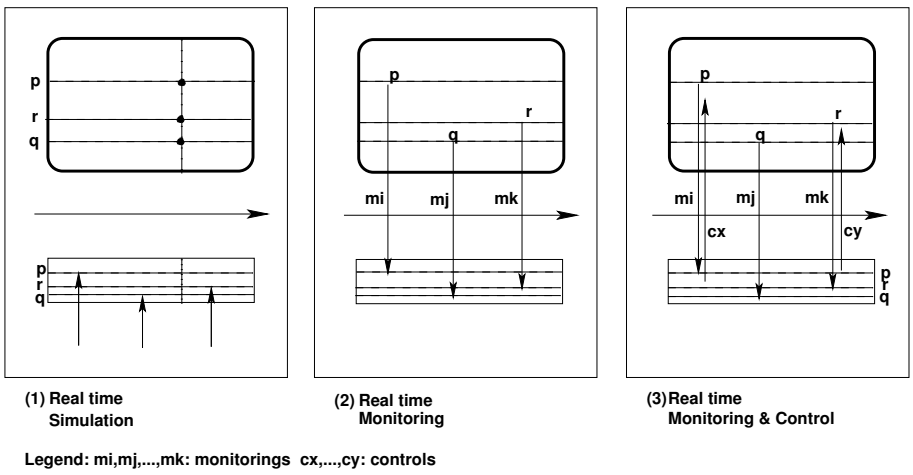

Figure 4.2: Simulation, Monitoring and Monitoring & Control

We have added some (three) horisontal and labelled (p, q and r) lines to Fig. 4.2(1,2,3) (with respect to the

traces of Fig. 4.1 on Page 109). They each denote a trace of a simple entity, an action or an event, that is, they are traces of values of these phenomena or concepts. A (named) action value could, for example, be the pair of the before and after states of the action and some description of the function ("insertion of a link", "start of a bus tour") involved in the action. A (named) event value could, for example, be a pair of the before and after states of the entities causing, respectively being effected by the event and some description of the predicate ("mudslide", "break-down of a bus") involved in the event. A cross section, such as designated by the vertical lines (one for the domain trace, one for the "corresponding" program trace) of Fig. 4.2 on the facing page(1) denotes a state: a domain, respectively a program state.

Figure 4.2(1) attempts to show a real-time demo or simulation for the chosen domain. Figure 4.2(2) purports to show the deployment of real-time software for monitoring (chosen aspects of) the chosen domain. Figure 4.2(3) purports to show the deployment of real-time software for monitoring as well as controlling (chosen aspects of) the chosen domain.

Monitoring

By *domain monitoring* we mean *"to be aware of the state of a domain"*, its simple entities, actions, events and behaviour. Domain monitoring is thus a process, typically within a distributed system for collecting and storing state data. In this process "observation" points — i.e., simple entities, actions and where events may occur — are identified in the domain, cf. points p, q and r of Fig. 4.2. Sensors are inserted at these points. The "downward" pointing vertical arrows of Figs. 4.2(2–3), from "the domain behaviour" to the "monitoring" and the "monitoring & control" traces express communication of what has been sensed (measured, photographed, etc.) [as directed by and] as input data (etc.) to these monitors. The monitor (being "executed") may store these "sensings" for future analysis.

Control

By *domain control* we mean *"the ability to change the value"* of simple entities and the course of actions and hence behaviours, including prevention of events of the domain. Domain control is thus based on domain monitoring. Actuators are inserted in the domain "at or near" monitoring points or at points related to these, viz. points p and r of Fig. 4.2 on the preceding page(3). The "upward" pointing vertical arrows of Fig. 4.2 on the facing page(3), from the "monitoring & control" traces to the "domain behaviour" express communication, to the domain, of what has been computed by the controller as a proper control reaction in response to the monitoring.

4.3.4 Machine Development

Machines

By a *machine* we shall understand a combination of hardware and software. For demos and simulators the machine is "mostly" software with the hardware typically being graphic display units with tactile instruments. For monitors the "main" machine, besides the hardware and software of demos and simulators, additionally includes *sensors* distributed throughout the domain and the technological machine means of *communicating* monitored signals from the sensors to the "main" machine and the processing of these signals by the main machine. For monitors & controllers the machine, besides the monitor machine, further includes actuators placed in the domain and the machine means of computing and communicating control signals to the actuators.

Requirements Development

Essential parts of $\mathbb{R}$equirements to a $\mathbb{M}$achine can be systematically "derived" from a $\mathbb{D}$omain description. These essential parts are the *domain requirements* and the *interface requirements*. Domain requirements are those requirements which can be expressed, say in narrative form, by mentioning technical terms only of

the domain. These technical terms cover only phenomena and concepts (simple entities, actions, events and behaviours) of the domain. Some domain requirements are *projected*, *instantiated*, made more *deterministic* and *extended*[3].

(a) By *domain projection* we mean a sub-setting of the domain description: parts are left out which the requirements stake-holders, collaborating with the requirements engineer, decide is of no relevance to the requirements. For our example it could be that our domain description had contained models of road net attributes such as "the wear & tear" of road surfaces, the length of links, states of hubs and links (that is, [dis]allowable directions of traffic through hubs and along links), etc. Projection might then omit these attributes.

(b) By *domain instantiation* we mean a specialisation of entities (simple, actions, events and behaviours), refining them from abstract simple entities to more concrete such, etc. For our example it could be that we only model freeways or only model road-pricing nets – or any one or more other aspects.

(c) By *domain determination* we mean that of making the domain description cum domain requirements prescription less non-deterministic, i.e., more deterministic (or even the other way around!). For our example it could be that we had domain-described states of street intersections as not controlled by traffic signals – where the determination is now that of introducing an abstract notion of traffic signals which allow only certain states (of red, yellow and green).

(d) By *domain extension* we basically mean that of extending the domain with phenomena and concepts that were not feasible without information technology. For our examples we could extend the domain with bus mounted GPS gadgets that record and communicate (to, say a central bus traffic computer) the more-or-less exact positions of buses – thereby enabling the observation of bus traffic.

Interface requirements are those requirements which can be expressed, say in narrative form, by mentioning technical terms both of the domain and of the machine. These technical terms thus cover shared phenomena and concepts, that is, phenomena and concepts of the domain which are, in some sense, also (to be) represented by the machine. Interface requirements represent (i) the initialisation and "on-the-fly" update of simple machine entities on the basis of *shared* domain entities; (ii) the interaction between the machine and the domain while the machine is carrying out a (previous domain) action; (iii) machine responses, if any, to domain events — or domain responses, if any, to machine events cum "outputs"; and (iv) machine monitoring and machine control of domain phenomena. Each of these four (i–iv) interface requirement facets themselves involve projection, instantiation, determination, extension and fitting.

Machine requirements are those requirements which can be expressed, say in narrative form, by mentioning technical terms only of the machine. (An example is: visual display units.)

4.3.5 Verifiable Software Development

An Example Set of Conjectures

(A) From a domain, $\mathscr{D}$, one can develop a domain description $\mathbb{D}$. $\mathbb{D}$ cannot be verified. It can at most be validated. Individual properties, $\mathbb{P}_{\mathbb{D}}$, of the domain description $\mathbb{D}$ and hence, purportedly, of the domain, $\mathscr{D}$, can be expressed and possibly proved

$$\mathbb{D} \models \mathbb{P}_{\mathbb{D}}$$

and these may be validated to be properties of $\mathscr{D}$ by observations in (or of) that domain.

(B) From a domain description, $\mathbb{D}$, one can develop requirements, $\mathbb{R}_{\text{DE}}$, for, and from $\mathbb{R}_{\text{DE}}$ one can develop a domain demo machine specification $\mathbb{M}_{\text{DE}}$ such that

$$\mathbb{D}, \mathbb{M}_{\text{DE}} \models \mathbb{R}_{\text{DE}}.$$

The formula $\mathbb{D}, \mathbb{M} \models \mathbb{R}$ can be read as follows: in order to prove that the $\mathbb{M}$achine satisfies the $\mathbb{R}$equirements, assumptions about the $\mathbb{D}$omain must often be made explicit in steps of the proof.

[3] We omit consideration of *fitting*.

(C) From a domain description, $\mathbb{D}$, and a domain demo machine specification, $\mathbb{S}_{\text{DE}}$, one can develop requirements, $\mathbb{R}_{\text{SI}}$, for, and from such a $\mathbb{R}_{\text{SI}}$ one can develop a domain simulator machine specification $\mathbb{M}_{\text{SI}}$ such that

$$(\mathbb{D};\mathbb{M}_{\text{DE}}),\mathbb{M}_{\text{SI}} \models \mathbb{R}_{\text{SI}}.$$

We have "lumped" $(\mathbb{D};\mathbb{M}_{\text{DE}})$ as the two constitute the extended domain for which we, in this case of development, suggest the next stage requirements and machine development to take place.

(D) From a domain description, $\mathbb{D}$, and a domain simulator machine specification, $\mathbb{M}_{\text{SI}}$, one can develop requirements, $\mathbb{R}_{\text{MO}}$, for, and from such a $\mathbb{R}_{\text{MO}}$ one can develop a domain monitor machine specification $\mathbb{M}_{\text{MO}}$ such that

$$(\mathbb{D};\mathbb{M}_{\text{SI}}),\mathbb{M}_{\text{MO}} \models \mathbb{R}_{\text{MO}}.$$

(E) From a domain description, $\mathbb{D}$, and a domain monitor machine specification, $\mathbb{M}_{\text{MO}}$, one can develop requirements, $\mathbb{R}_{\text{MC}}$, for, and from such a $\mathbb{R}_{\text{MC}}$ one can develop a domain monitor & controller machine specification $\mathbb{M}_{\text{MC}}$ such that

$$(\mathbb{D};\mathbb{M}_{\text{MO}}),\mathbb{M}_{\text{MC}} \models \mathbb{R}_{\text{MC}}.$$

Chains of Verifiable Developments

The above illustrated just one chain of development. There are others. All are shown in Fig. 4.3. The above development is shown as the longest horisontal chain (third row).

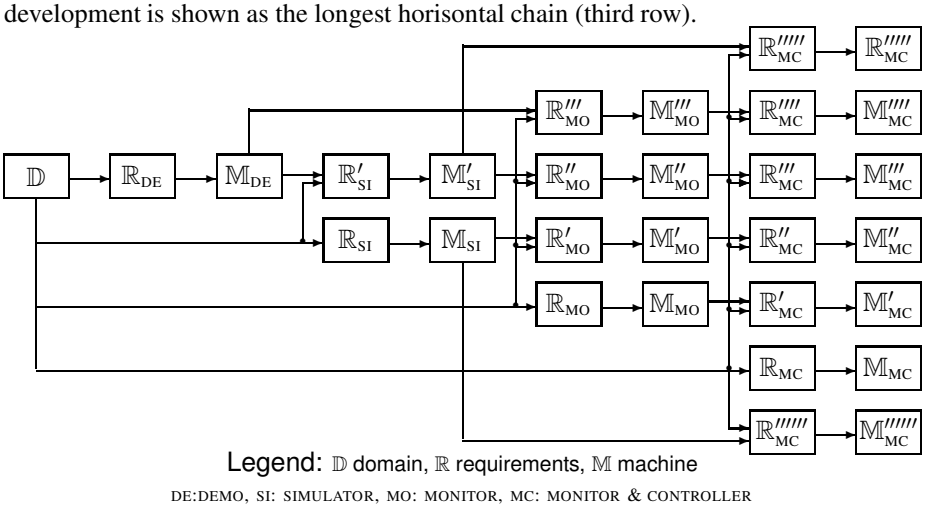

Figure 4.3: Chains of Verifiable Developments

Figure 4.3 can also be interpreted as prescribing a widest possible range of machine cum software products [54, 124] for a given domain. One domain may give rise to many different kinds of DEmo machines, SImulators, MOnitors and Monitor & Controllers (the unprimed versions of the $\mathbb{M}_{\text{T}}$ machines (where T ranges over DE, SI, MO, MC)). For each of these there are similarly, "exponentially" many variants of successor machines (the primed versions of the $\mathbb{M}_{\text{T}}$ machines).

What does it mean that a machine is a primed version? Well, here it means, for example, that $\mathbb{M}'_{\text{SI}}$ embodies facets of the demo machine $\mathbb{M}_{\text{DE}}$, and that $\mathbb{M}'''_{\text{MC}}$ embodies facets of the demo machine $\mathbb{M}_{\text{DE}}$, of the simulator $\mathbb{M}'_{\text{SI}}$, and the monitor $\mathbb{M}''_{\text{MO}}$. Whether such requirements are desirable is left to product customers and their software providers [54, 124] to decide.

4.4 Conclusion

Our divertimento is almost over. It is time to conclude.

4.4.1 Discussion

The $\mathbb{D},\mathbb{M} \models \mathbb{R}$ ('correctness' of) development relation appears to have been first indicated in the Computational Logic Inc. `Stack` [11, 87] and the EU ESPRIT `ProCoS` [16, 17] projects; [88] presents this same idea with a purpose much like ours, but with more technical details and full discussion.

The term 'domain engineering' appears to have at least two meanings: the one used here [25, 30] and one [94, 73, 55] emerging out of the Software Engineering Institute at CMU where it is also called *product line engineering*[4]. Our meaning, is, in a sense, more narrow, but then it seems to also be more highly specialised (with detailed description and formalisation principles and techniques). Fig. 4.3 on the previous page illustrates, in capsule form, what we think is the CMU/SEI meaning. The relationship between, say Fig. 4.3 and *model-based software development* seems obvious but need be explored.

What Have We Achieved

We have characterised a spectrum of strongly domain-related as well as strongly inter-related (cf. Fig. 4.3) software product families: *demos*, *simulators*, *monitors* and *monitor & controllers*. We have indicated varieties of these: simulators based on demos, monitors based on simulators, monitor & controllers based on monitors, in fact any of the latter ones in the software product family list as based on any of the earlier ones. We have sketched temporal relations between simulation traces and domain behaviours: *a priori*, *a posteriori*, *macroscopic* and *microscopic*, and we have identified the real-time cases which lead on to monitors and monitor & controllers.

What Have We Not Achieved — Some Conjectures

We have not characterised the software product family relations other than by the $\mathbb{D}$, $\mathbb{M} \models \mathbb{R}$ and $(\mathbb{D};\mathbb{M}_{\mathrm{XYZ}}), \mathbb{M} \models \mathbb{R}$ clauses. That is, we should like to prove conjectured type theoretic inclusion relations like:

$$\wp([\![\mathscr{M}_{\mathrm{X}_{\underline{\mathsf{mod}}\ ext.}}]\!]) \supseteq \wp([\![\mathscr{M}'^{\ldots'}_{\mathrm{X}_{\underline{\mathsf{mod}}\ ext.}}]\!]), \quad \wp([\![\mathscr{M}'^{\ldots'}_{\mathrm{X}_{\underline{\mathsf{mod}}\ ext.}}]\!]) \supseteq \wp([\![\mathscr{M}''^{\ldots'}_{\mathrm{X}_{\underline{\mathsf{mod}}\ ext.}}]\!])$$

where X and Y range appropriately, where $[\![\mathscr{M}]\!]$ expresses the meaning of $\mathscr{M}$, where $\wp([\![\mathscr{M}]\!])$ denote the space of all machine meanings and where $\wp([\![\mathscr{M}_{x_{\underline{\mathsf{mod}}\ ext.}}]\!])$ is intended to denote that space modulo ("free of") the **y** facet (here *ext.*, for extension).

That is, it is conjectured that the set of more specialised, i.e., n primed, machines of kind x is type theoretically "contained" in the set of m primed (unprimed) x machines ($0 \leq m < n$).

There are undoubtedly many such interesting relations between the DEMO, SIMULATOR, MONITOR and MONITOR & CONTROLLER machines, unprimed and primed.

What Should We Do Next

This paper has the subtitle: *A Divertimento of Ideas and Suggestions.* It is not a proper theoretical paper. It tries to throw some light on families and varieties of software, i.e., their relations, and. It focuses, in particular, on so-called DEMO, SIMULATOR, MONITOR and MONITOR & CONTROLLER software and their relation to the "originating" domain, i.e., that in which such software is to serve, and hence that which is being *extended* by such software, cf. the compounded 'domain' $(\mathbb{D};\mathbb{M}_i)$ of in $(\mathbb{D};\mathbb{M}_i),\mathbb{M}_j \models \mathbb{D}$. These notions should be studied formally. All of these notions: requirements projection, instantiation, determination and extension can be formalised; and the specification language, in the form used here (without `CSP` processes, [97]) has a formal semantics and a proof system — so the various notions of development, $(\mathbb{D};\mathbb{M}_i),\mathbb{M}_j \models \mathbb{R}$ and $\wp(\mathbb{M})$ can be formalised.

[4] http://en.wikipedia.org/wiki/Domain_engineering.

Chapter 5

A Rôle for Mereology in Domain Science and Engineering

Chapter Status

This chapter must be revised.
It contains a technical error.
The model of parts and subparts, Sec. 5.3.1, is too simple.

Abstract

We give an abstract model of parts and part-hood relations of software application domains such as the financial service industry, railway systems, road transport systems, health care, oil pipelines, secure [IT] systems, etcetera. We relate this model to axiom systems for mereology [57], showing satisfiability, and show that for every mereology there corresponds a class of `Communicating Sequential Processes` [97], that is: a λ–expression.

5.1 Introduction

The term 'mereology' is accredited to the Polish mathematician, philosopher and logician Stanisław Leśniewski (1886–1939) who "was a nominalist: he rejected axiomatic set theory and devised three formal systems,*Protothetic*, *Ontology*, and *Mereology* as a concrete alternative to set theory". In this contribution I shall be concerned with only certain aspects of mereology, namely those that appears most immediately relevant to domain science (a relatively new part of current computer science). Our knowledge of 'mereology' has been through studying, amongst others, [57, 113].

5.1.1 Computing Science Mereology

"Mereology (from the Greek $\mu\varepsilon\rho o\varsigma$ 'part') is the theory of parthood relations: of the relations of part to whole and the relations of part to part within a whole"[1]. In this contribution we restrict 'parts' to be those that, firstly, are spatially distinguishable, then, secondly, while "being based" on such spatially distinguishable parts, are conceptually related. The relation: "being based", shall be made clear in this contribution.

[1] Achille Varzi: Mereology, http://plato.stanford.edu/entries/mereology/ 2009 and [57]

Accordingly two parts, p_x and p_y, (of a same "whole") are are either "adjacent", or are "embedded within" one another as loosely indicated in Fig. 5.1.

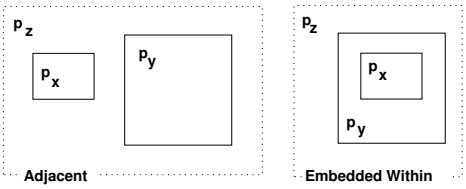

Figure 5.1: 'Adjacent' and "Embedded Within' parts

'Adjacent' parts are direct parts of a same third part, p_z, i.e., p_x and p_y are "embedded within" p_z; or one (p_x) or the other (p_y) or both (p_x and p_y) are parts of a same third part, p'_z "embedded within" p_z; etcetera; as loosely indicated in Fig. 5.2. or one is "embedded within" the other — etc. as loosely indicated in Fig. 5.2.

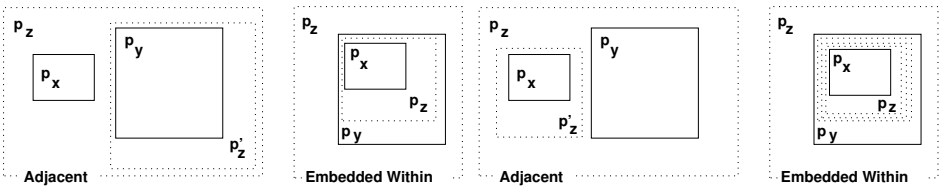

Figure 5.2: 'Adjacent' and "Embedded Within' parts

Parts, whether adjacent or embedded within one another, can share properties. For adjacent parts this sharing seems, in the literature, to be diagrammatically expressed by letting the part rectangles "intersect". Usually properties are not spatial hence 'intersection' seems confusing. We refer to Fig. 5.3.

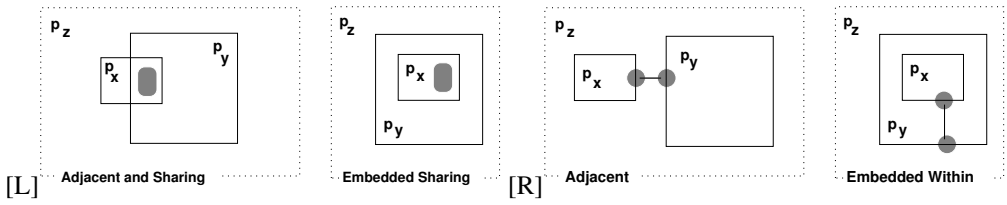

Figure 5.3: Two models, [L,R], of parts sharing properties

Instead of depicting parts sharing properties as in Fig. 5.3[L]eft where dashed rounded edge rectangles stands for 'sharing', we shall (eventually) show parts sharing properties as in Fig. 5.3[R]ight where •—• connections connect those parts.

5.1.2 From Domains via Requirements to Software

One reason for our interest in mereology is that we find that concept relevant to the modelling of domains. A derived reason is that we find the modelling of domains relevant to the development of software. Conventionally a first phase of software development is that of requirements engineering. To us domain engineering is (also) a prerequisite for requirements engineering [26, 46]. Thus to properly ***design*** 𝕊oftware

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

5.1.3 Domains: Science and Engineering

Domain science is the study and knowledge of domains. **Domain engineering** is the practice of **"walking the bridge"** from domain science to domain descriptions: to **create domain descriptions** on the background of scientific knowledge of domains, the specific domain "at hand", or domains in general; and to **study domain descriptions** with a view to broaden and deepen scientific results about domain descriptions. This contribution is based on the engineering and study of many descriptions, of air traffic, banking, commerce (the consumer/retailer/wholesaler/producer supply chain), container lines, health care, logistics, pipelines, railway systems, secure [IT] systems, stock exchanges, etcetera.

5.1.4 Contributions of This Contribution

A general contribution is that of providing elements of a domain science. Three specific contributions are those of (i) giving a model that satisfies published formal, axiomatic characterisations of mereology; (ii) showing that to every (such modelled) mereology there corresponds a CSP [97] program and to conjecture the reverse; and, related to (ii), (iii) suggesting complementing **syntactic** and **semantic** theories of mereology.

5.1.5 Structure of This Contribution

We briefly overview the structure of this contribution. First, on Sect. 5.2, **we loosely characterise how we look at mereologies: *"what they are to us !"***. Then, in Sect. 5.3, **we give an abstract, model-oriented specification of a class of mereologies** in the form of composite parts and composite and atomic subparts and their possible connections. The abstract model as well as the axiom system (Sect. 5.4) focuses on the **syntax of mereologies.** Following that, in Sect. 5.4 **we indicate how the model of** Sect. 5.3 **satisfies the axiom system of that section.** In preparation for Sect. 5.6, Sect. 5.5 **presents characterisations of attributes of parts, whether atomic or composite.** Finally Sect. 5.6 presents **a semantic model of mereologies,** one of a wide variety of such possible models. This one emphasize the possibility of considering parts and subparts as processes and hence a mereology as a system of processes. Section 5.7 concludes with some remarks on what we have achieved.

5.2 Our Concept of Mereology

5.2.1 Informal Characterisation

Mereology, to us, is the study and knowledge about how physical and conceptual parts relate and what it means for a part to be related to another part: *being disjoint, being adjacent, being neighbours, being contained properly within, being properly overlapped with*, etcetera. By physical parts we mean such spatial individuals which can be pointed to. **Examples:** *a road net (consisting of street segments and street intersections); a street segment (between two intersections); a street intersection; a road (of sequentially neigbouring street segments of the same name) a vehicle; and a platoon (of sequentially neigbouring vehicles).*

By a conceptual part we mean an abstraction with no physical extent, which is either present or not. **Examples:** *a bus timetable (not as a piece or booklet of paper, or as an electronic device, but) as an image in the minds of potential bus passengers; and routes of a pipeline, that is, neighbouring sequences of pipes,*

valves, pumps, forks and joins, for example referred to in discourse: the gas flows through "such-and-such" a route". The tricky thing here is that a route may be thought of as being both a concept or being a physical part — in which case one ought give them different names: a planned route and an actual road, for example.

The mereological notion of subpart, that is: *contained within* can be illustrated by **examples:** *the intersections and street segments are subparts of the road net; vehicles are subparts of a platoon; and pipes, valves, pumps, forks and joins are subparts of pipelines.* The mereological notion of adjacency can be illustrated by **examples.** We consider *the various controls of an air traffic system, cf. Fig. 5.4, as well as its aircrafts as adjacent within the air traffic system; the pipes, valves, forks, joins and pumps of a pipeline, cf. Fig. 5.9 on Page 121, as adjacent within the pipeline system; two or more banks of a banking system, cf. Fig. 5.6 on Page 120, as being adjacent.* The mereo-topological notion of neighbouring can be illustrated by **examples:** *Some adjacent pipes of a pipeline are neighbouring (connected) to other pipes or valves or pumps or forks or joins, etcetera; two immediately adjacent vehicles of a platoon are neighbouring.* The mereological notion of proper overlap can be illustrated by **examples** some of which are of a general kind: *two routes of a pipelines may overlap; and two conceptual bus timetables may overlap with some, but not all bus line entries being the same;* and some of really reflect adjacency: *two adjacent pipe overlap in their connection, a wall between two rooms overlap each of these rooms — that is, the rooms overlap each other "in the wall".*

5.2.2 Six Examples

We shall, in Sect. 5.3, present a model that is claimed to abstract essential mereological properties of air traffic, buildings and their installations, machine assemblies, financial service industry, the oil industry and oil pipelines, and railway nets.

Air Traffic

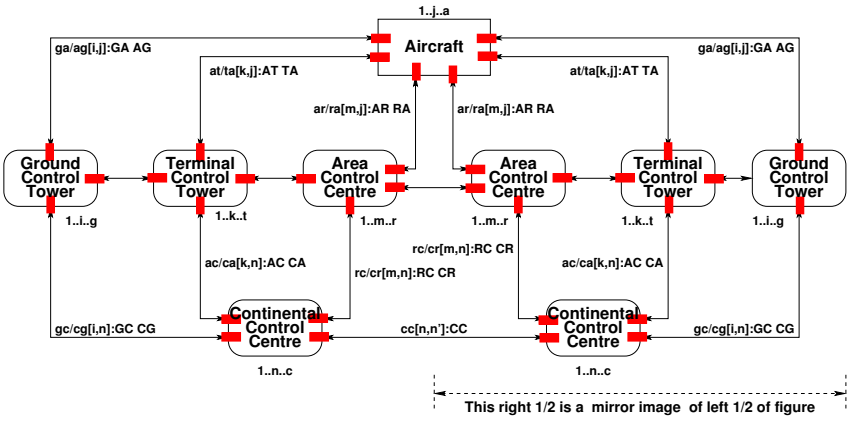

Figure 5.4: A schematic air traffic system

Figure 5.4 shows nine adjacent (9) boxes and eighteen adjacent (18) lines. Boxes and lines are parts. The line parts "neighbours" the box parts they "connect". Individually boxes and lines represent adjacent parts of the composite air traffic "whole". The rounded corner boxes denote buildings. The sharp corner box denote an aircraft. Lines denote radio telecommunication. The "overlap" between neigbouring line and box parts are indicated by "connectors". Connectors are shown as small filled, narrow, either horisontal

or vertical "filled" rectangle[2] at both ends of the double-headed-arrows lines, overlapping both the line arrows and the boxes. The index ranges shown attached to, i.e., labelling each unit, shall indicate that there are a multiple of the "single" (thus representative) box or line unit shown. These index annotations are what makes the diagram of Fig. 5.4 on the preceding page schematic. Notice that the 'box' parts are fixed installations and that the double-headed arrows designate the ether where radio waves may propagate. We could, for example, assume that each such line is characterised by a combination of location and (possibly encrypted) radio communication frequency. That would allow us to consider all lines for not overlapping. And if they were overlapping, then that must have been a decision of the air traffic system.

Buildings

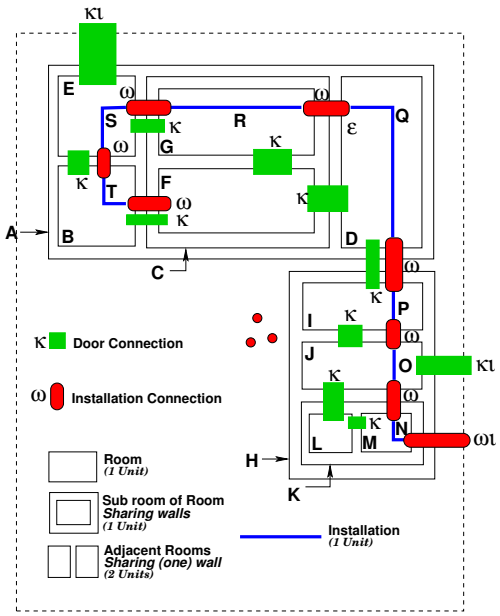

Figure 5.5: A building plan with installation

Figure 5.5 shows a building plan — as a composite part. The building consists of two buildings, A and H. The buildings A and H are neighbours, i.e., shares a common wall. Building A has rooms B, C, D and E, Building H has roomsI, J and K; Rooms L and M are within K. Rooms F and G are within C.

The thick lines labelled N, O, P, Q, R, S, and T models either electric cabling, water supply, air conditioning, or some such "flow" of gases or liquids.

Connection $\kappa\iota o$ provides means of a connection between an environment, shown by dashed lines, and B or J, i.e. "models", for example, a door. Connections κ provides "access" between neighbouring rooms. Note that 'neighbouring' is a transitive relation. Connection $\omega\iota o$ allows electricity (or water, or oil) to be conducted between an environment and a room. Connection ω allows electricity (or water, or oil) to be conducted through a wall. Etcetera.

Thus "the whole" consists of A and B. Immediate subparts of A are B, C, D and E. Immediate subparts of C are G and F. Etcetera.

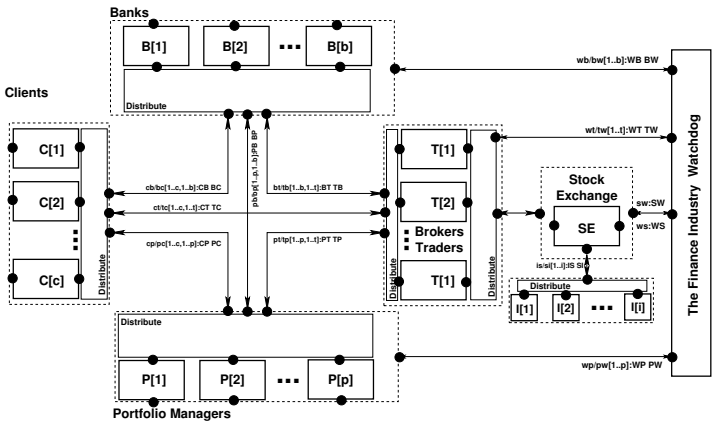

Figure 5.6: A financial service industry

Financial Service Industry

Figure 5.6 is rather rough-sketchy! It shows seven (7) larger boxes [6 of which are shown by dashed lines], six [6] thin lined "distribution" boxes, and twelve (12) double-arrowed lines. Boxes and lines are parts. (We do not described what is meant by "distribution".) Where double-arrowed lines touch upon (dashed) boxes we have connections. Six (6) of the boxes, the dashed line boxes, are composite parts, five (5) of them consisting of a variable number of atomic parts; five (5) are here shown as having three atomic parts each with bullets "between" them to designate "variability". Clients, not shown, access the outermost (and hence the "innermost" boxes, but the latter is not shown) through connections, shown by bullets, •.

Machine Assemblies

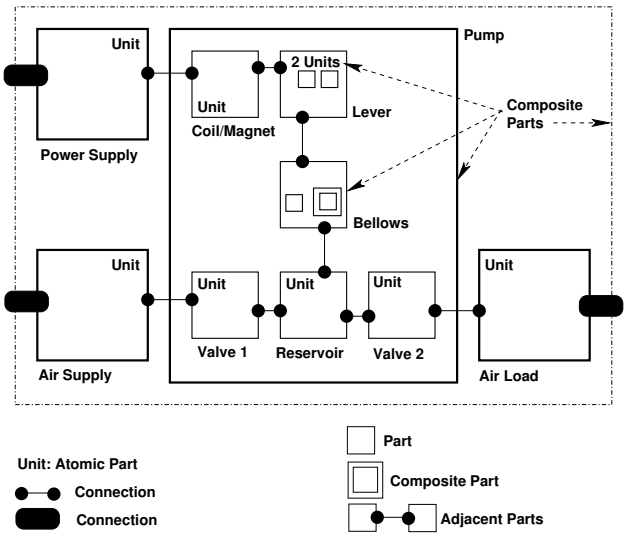

Figure 5.7: An air pump, i.e., a physical mechanical system

[2]There are 38 such rectangles in Fig. 5.4 on Page 118.

Figure 5.7 on the facing page shows a machine assembly. Square boxes show composite and atomic parts. Black circles or ovals show connections. The full, i.e., the level 0, composite part consists of four immediate parts and three internal and three external connections. The Pump is an assembly of six (6) immediate parts, five (5) internal connections and three (3) external connectors. Etcetera. Some connections afford "transmission" of electrical power. Other connections convey torque. Two connections convey input air, respectively output air.

Oil Industry

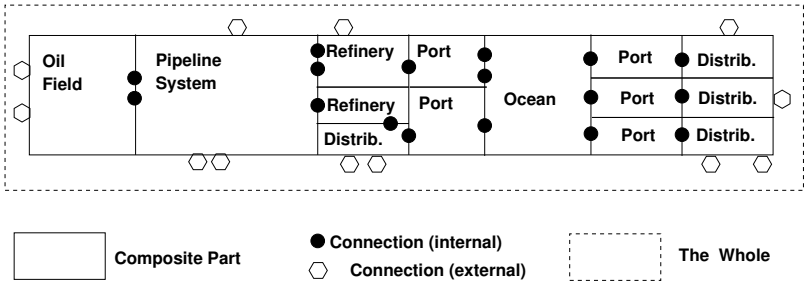

Figure 5.8: A Schematic of an Oil Industry

"The" Overall Assembly Figure 5.8 shows a composite part consisting of fourteen (14) composite parts, left-to-right: one oil field, a crude oil pipeline system, two refineries and one, say, gasoline distribution network, two seaports, an ocean (with oil and ethanol tankers and their sea lanes), three (more) seaports, and three, say gasoline and ethanol distribution networks.

Between all of the neighbouring composite parts there are connections, and from some of these composite parts there are connections (to an external environment). The crude oil pipeline system composite part will be concretised next.

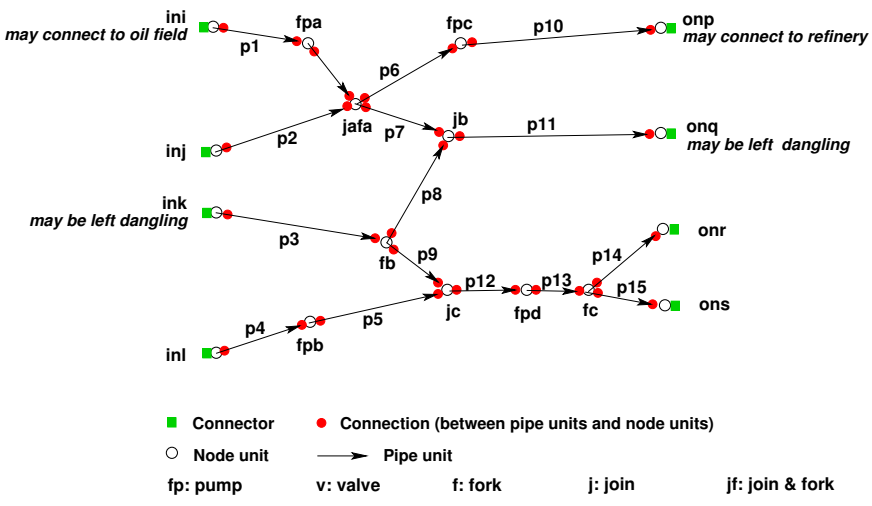

Figure 5.9: A pipeline system

A Concretised Composite parts Figure 5.9 on the preceding page shows a pipeline system. It consists of 32 atomic parts: fifteen (15) pipe units (shown as directed arrows and labelled p1–p15), four (4) input node units (shown as small circles, ○, and labelled in$i$–inℓ), four (4) flow pump units (shown as small circles, ○, and labelled fp$a$–fp$d$), five (5) valve units (shown as small circles, ○, and labelled vx–vw), three (3) join units (shown as small circles, ○, and labelled jb–jc), two (2) fork units (shown as small circles, ○, and labelled fb–fc), one (1) combined join & fork unit (shown as small circles, ○, and labelled jafa), and four (4) output node units (shown as small circles, ○, and labelled on$p$–on$s$).

In this example the routes through the pipeline system start with node units and end with node units, alternates between node units and pipe units, and are connected as shown by fully filled-out dark coloured disc connections. Input and output nodes have input, respectively output connections, one each, and shown as lighter coloured connections.

Railway Nets

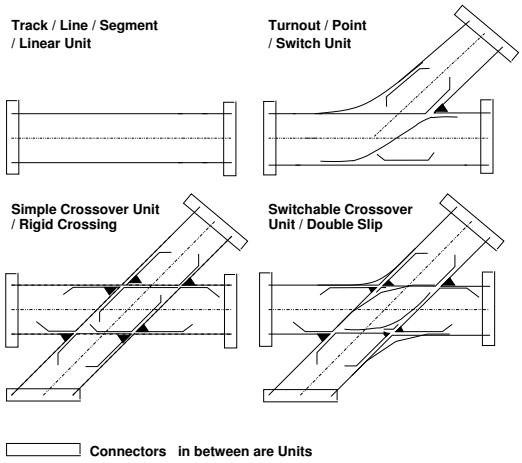

Figure 5.10: Four example rail units

Figure 5.10 diagrams four rail units, each with two, three or four connectors shown as narrow, somewhat "longish" rectangles. Multiple instances of these rail units can be assembled (i.e., composed) by their connectors as shown on Fig. 5.11 on the facing page into proper rail nets.

Figure 5.11 on the next page diagrams an example of a proper rail net. It is assembled from the kind of units shown in Fig. 5.10. In Fig. 5.11 consider just the four dashed boxes: The dashed boxes are assembly units. Two designate stations, two designate lines (tracks) between stations. We refer to to the caption four line text of Fig. 5.10 for more "statistics". We could have chosen to show, instead, for each of the four "dangling' connectors, a composition of a connection, a special "end block" rail unit and a connector.

Discussion

We have brought these examples only to indicate the issues of a "whole" and atomic and composite parts, adjacency, within, neighbour and overlap relations, and the ideas of attributes and connections. We shall make the notion of 'connection' more precise in the next section. [158] gives URLs to a number of domain models illustrating a great variety of mereologies.

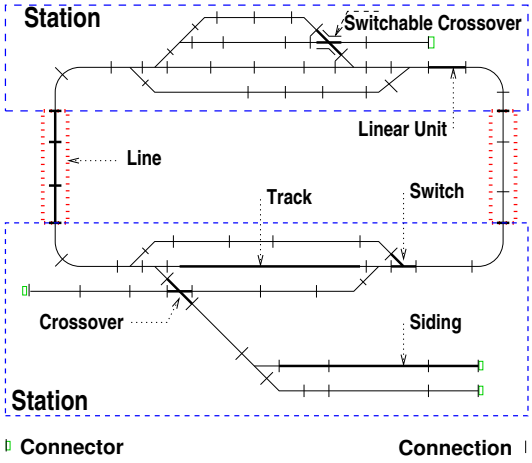

Figure 5.11: A "model" railway net. An Assembly of four Assemblies:
two stations and two lines; Lines here consist of linear rail units;
stations of all the kinds of units shown in Fig. 5.10 on the facing page.
There are 66 connections and four "dangling" connectors

5.3 An Abstract, Syntactic Model of Mereologies

We distinguish between **atomic** and **composite parts.** Atomic parts do not contain separately distinguishable parts. Composite parts contain at least one separately distinguishable part. It is the domain analyser who decides what constitutes "the whole", that is, how parts relate to one another, what constitutes parts, and whether a part is atomic or composite. We refer to the proper parts of a composite part as subparts.

5.3.1 Parts and Subparts

Figure 5.12 on the next page illustrates composite and atomic parts. The *slanted sans serif* uppercase identifiers of Fig. 5.12 *A1, A2, A3, A4, A5, A6* and *C1, C2, C3* are meta-linguistic, that is. they stand for the parts they "decorate"; they are not identifiers of "our system".

The Model

The formal models of this contribution are expressed in the RAISE Specification Language, RSL [86, 85, 22].

226 The "whole" contains a set of parts.

227 A part is either an atomic part or a composite part.

228 One can observe whether a part is atomic or composite.

229 Atomic parts cannot be confused with composite parts.

230 From a composite part one can observe one or more parts.

type
226. W = P-**set**
227. P = A | C

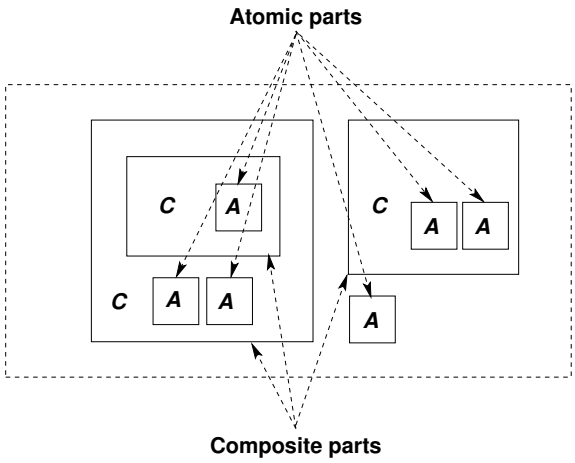

Figure 5.12: Atomic and composite parts

value
228. is_A: P → **Bool**, is_C: P → **Bool**
axiom
229. ∀ a:A,c:C•a≠c, i.e., A∩C={||} ∧ is_A(a)≡∼is_C(a)∧is_C(c)≡∼is_A(c)
value
230. obs_Ps: C → P-**set** **axiom** ∀ c:C • obs_Ps(c)≠{}

Fig. 5.12 and the expressions below illustrate the observer function obs_Ps:

- obs_Ps(*C1*) = {*A2, A3, C3*},
- obs_Ps(*C2*) = {*A4, A5*},
- obs_Ps(*C3*) = {*A6*}.

Please note that this example is meta-linguistic. We can define an auxiliary function.

231 From a composite part, c, we can extract all atomic and composite parts

a observable from c or

b extractable from parts observed from c.

value
231. xtr_Ps: C → P-**set**
231. xtr_Ps(c) ≡
231a. **let** ps = obs_Ps(c) **in**
231b. ps ∪ ∪ {obs_Ps(c′)|c′:C • c′ ∈ ps} **end**

5.3.2 'Within' and 'Adjacency' Relations

'Within'

232 One part, p, is said to be *immediately within*, imm_within(p,p′), another part,

a if p′ is a composite part
b and p is observable in p′.

value
232. imm_within: P × P $\xrightarrow{\sim}$ **Bool**
232. imm_within(p,p′) ≡
232a. is_C(p′)
232b. ∧ p ∈ obs_Ps(p′)

'Transitive Within'

We can generalise the 'immediate within' property.

233 A part, p, is transitively within a part p′, within(p,p′),
a either if p, is immediately within p′
b or if there exists a (proper) composite part p″ of p′ such that within(p″,p).

value
233. within: P × P $\xrightarrow{\sim}$ **Bool**
233. within(p,p′) ≡
233a. imm_within(p,p′)
233b. ∨ ∃ p″:C • p″ ∈ obs_Ps(p′) ∧ within(p,p″)

'Adjacency'

234 Two parts, p,p′, are said to be *immediately adjacent*, imm_adjacent(p,p′)(c), to one another, in a composite part c, such that p and p′ are distinct and observable in c.

value
234. imm_adjacent: P × P → C $\xrightarrow{\sim}$ **Bool**,
234. imm_adjacent(p,p′)(c) ≡ p≠p′ ∧ {p,p′}⊆obs_Ps(c)

Transitive 'Adjacency'

We can generalise the immediate 'adjacent' property.

235 Two parts, p,p′, of a composite part, c, are adjacent(p, p′) in c
a either if imm_adjacent(p,p′)(c),
b or if there are two p″ and p‴ of c such that
i p″ and p‴ are immediately adjacent parts of c and
ii p is equal to p″ or p″ is properly within p and p′ is equal to p‴ or p‴ is properly within p′

value
235. adjacent: P × P → C $\xrightarrow{\sim}$ **Bool**
235. adjacent(p,p′)(c) ≡
235a. imm_adjacent(p,p′)(c) ∨
235b. ∃ p″,p‴:P •
235(b)i. imm_adjacent(p″,p‴)(c) ∧
235(b)ii. ((p=p″)∨within(p,p″)(c)) ∧ ((p′=p‴)∨within(p′,p‴)(c))

5.3.3 Unique Identifications

Each physical part can be uniquely distinguished for example by an abstraction of its properties at a time of origin. In consequence we also endow conceptual parts with unique identifications.

236 In order to refer to specific parts we endow all parts, whether atomic or composite, with **u**nique **id**entifications.

237 We postulate functions which observe these **u**nique **id**entifications, whether as parts in general or as atomic or composite parts in particular.

238 such that any to parts which are distinct have **u**nique **id**entifications.

type
236. Π
value
237. uid_Π: P → Π
axiom
238. ∀ p,p′:P • p≠p′ ⇒ uid_Π(p)≠uid_Π(p′)

Figure 5.13 illustrates the unique identifications of composite and atomic parts.

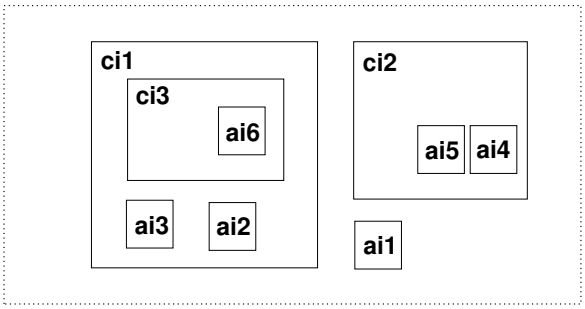

Figure 5.13: ai_j: atomic part identifiers, ci_k: composite part identifiers

We exemplify the observer function obs_Π in the expressions below and on Fig. 5.13:

- obs_Π(*C1*) = *ci1*, obs_Π(*C2*) = *ci2*, etcetera; and
- obs_Π(*A1*) = *ai1*, obs_Π(*A2*) = *ai2*, etcetera.

Please note that also this example is meta-linguistic.

239 We can define an auxiliary function which extracts all part identifiers of a composite part and parts within it.

value
239. xtr_Πs: C → Π-**set**
239. xtr_Πs(c) ≡ {uid_Π(c)} ∪ ∪ {uid_Π(p)|p:P•p ∈ xtr_Πs(c)}

5.3.4 Attributes

In Sect. 5.5 we shall explain the concept of properties of parts, or, as we shall refer to them, attributes For now we just postulate that

240 parts have sets of attributes, atr:ATR, (whatever they are!),

241 that we can observe attributes from parts, and hence

242 that two distinct parts may share attributes

243 for which we postulate a membership function ∈.

type
240. ATR
value
241. atr_ATRs: P → ATR-**set**
242. share: P×P → **Bool**
242. share(p,p′) ≡ p≠p′∧∃ atr:ATR•atr ∈ atr_ATRs(p)∧atr ∈ atr_ATRs(p′)
243. ∈: ATR × ATR-**set** → **Bool**

5.3.5 Connections

In order to illustrate other than the within and adjacency part relations we introduce the notions of connectors and, hence, connections. Figure 5.14 illustrates connections between parts. A connector is, visually, a •—• line that connects two distinct part boxes.

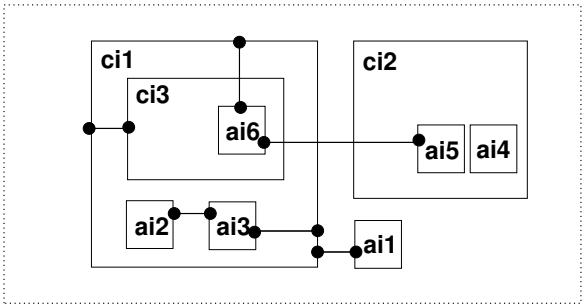

Figure 5.14: Connectors

244 We may refer to the connectors by the two element sets of the unique identifiers of the parts they connect.

For **example:**

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

245 From a part one can observe the unique identities of the other parts to which it is connected.

```
type
244.  K = {| k:Π-set • card k = 2 |}
value
245.  mereo_Ks: P → K-set
```

246 The set of all possible connectors of a part can be calculated.

```
value
246.  xtr_Ks: P → K-set
246.  xtr_Ks(p) ≡ {{uid_Π(p),π}|π:Π•π ∈ mereo_Πs(p)}
```

Connector Wellformedness

247 For a composite part, s:C,

248 all the observable connectors, ks,

249 must have their two-sets of part identifiers identify parts of the system.

```
value
247.  wf_Ks: C → Bool
247.  wf_Ks(c) ≡
248.    let ks = xtr_Ks(c), πs = mereo_Πs(c) in
249.    ∀ {π′,π″}:Π-set • {π′,π″}⊆ks ⇒
249.      ∃ p′,p″:P • {π′,π″}={uid_Π(p′),uid_Π(p″)} end
```

Connector and Attribute Sharing Axioms

250 We postulate the following axiom:

a If two parts share attributes, then there is a connector between them; and

b if there is a connector between two parts, then they share attributes.

251 The function xtr_Ks (Item 246) can be extended to apply to Wholes.

```
axiom
250.  ∀ w:W•
250.    let ps = xtr_Ps(w), ks = xtr_Ks(w) in
250a.   ∀ p,p′:P • p≠p′ ∧ {p,p′}⊆ps ∧ share(p,p′) ⇒
250a.     {uid_Π(p),uid_Π(p′)} ∈ ks ∧
250b.   ∀ {uid,uid′} ∈ ks ⇒
250b.     ∃ p,p′:P • {p,p′}⊆ps ∧ {uid,uid′}={uid_Π(p),uid_Π(p′)}
250b.       ⇒ share(p,p′) end
value
251.  xtr_Ks: W → K-set
251.  xtr_Ks(w) ≡ ∪{xtr_Ks(p)|p:P•p ∈ obs_Ps(p)}
```

In other words: modelling sharing by means of intersection of attributes or by means of connectors is "equivalent".

Sharing

252 When two distinct parts share attributes,

253 then they are said to be sharing:

252. sharing: P × P → **Bool**
253. sharing(p,p′) ≡ p≠p′∧share(p,p′)

5.3.6 Uniqueness of Parts

There is one property of the model of wholes: W, Item 226 on Page 123, and hence the model of composite and atomic parts and their unique identifiers "spun off" from W (Item 227 [Page 123] to Item 250b [Page 128]). and that is that any two parts as revealed in different, say adjacent parts are indeed unique, where we — simplifying — define uniqueness sôlely by the uniqueness of their identifiers.

Uniqueness of Embedded and Adjacent Parts

254 By the definition of the obs_Ps function, as applied obs_Ps(c) to composite parts, c:C, the atomic and composite subparts of c are all distinct and have distinct identifiers (uiids: unique immediate identifiers).

value
254. uiids: C → **Bool**
254. uiids(c) ≡ ∀ p,p′:P•p≠p′∧{p,p′}⊆obs_Ps(c)⇒**card**{uidΠ(p),uidΠ(p′),uidΠ(c)}=3

255 We must now specify that that uniqueness is "propagated" to parts that are proper parts of parts of a composite part (uids: unique identifiers).

255. uids: C → **Bool**
255. uids(c) ≡
255. ∀ c′:C•c′ ∈ obs_Ps(c) ⇒ uiids(c′)
255. ∧ **let** ps′=xtr_Ps(c′),ps″=xtr_Ps(c″) **in**
255. ∀ c″:C•c″ ∈ ps′⇒uids(c″)
255. ∧ ∀ p′,p″:P•p′ ∈ ps′∧p″ ∈ ps″⇒uid_Π(p′)≠uid_Π(p″) **end**

5.4 An Axiom System

Classical axiom systems for mereology focus on just one sort of "things", namely 𝒫arts. Leśniewski had in mind, when setting up his mereology to have it supplant set theory. So parts could be composite and consisting of other, the sub-parts — some of which would be atomic; just as sets could consist of elements which were sets — some of which would be empty.

5.4.1 Parts and Attributes

In our axiom system for mereology we shall avail ourselves of two sorts: 𝒫arts, and 𝒜ttributes.[3]

[3] Identifiers P and A stand for model-oriented types (parts and atomic parts), whereas identifiers 𝒫 and 𝒜 stand for property-oriented types (parts and attributes).

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

$\mathscr{A}$ttributes are associated with $\mathscr{P}$arts. We do not say very much about attributes: We think of attributes of parts to form possibly empty sets. So we postulate a primitive predicate, $\in$, relating $\mathscr{P}$arts and $\mathscr{A}$ttributes.

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

5.4.2 The Axioms

The axiom system to be developed in this section is a variant of that in [57]. We introduce the following relations between parts:

part_of:	$\mathbb{P}:$	$\mathscr{P} \times \mathscr{P}$	$\to$	**Bool**	Page 130
proper_part_of:	$\mathbb{PP}:$	$\mathscr{P} \times \mathscr{P}$	$\to$	**Bool**	Page 130
overlap:	$\mathbb{O}:$	$\mathscr{P} \times \mathscr{P}$	$\to$	**Bool**	Page 130
underlap:	$\mathbb{U}:$	$\mathscr{P} \times \mathscr{P}$	$\to$	**Bool**	Page 130
over_crossing:	$\mathbb{OX}:$	$\mathscr{P} \times \mathscr{P}$	$\to$	**Bool**	Page 130
under_crossing:	$\mathbb{UX}:$	$\mathscr{P} \times \mathscr{P}$	$\to$	**Bool**	Page 130
proper_overlap:	$\mathbb{PO}:$	$\mathscr{P} \times \mathscr{P}$	$\to$	**Bool**	Page 131
proper_underlap:	$\mathbb{PU}:$	$\mathscr{P} \times \mathscr{P}$	$\to$	**Bool**	Page 131

Let $\mathbb{P}$ denote ***part-hood***; p_x is part of p_y, is then expressed as $\mathbb{P}(p_x, p_y)$.[4] (5.1) Part p_x is part of itself (reflexivity). (5.2) If a part p_x is part p_y and, vice versa, part p_y is part of p_x, then $p_x = p_y$ (antisymmetry). (5.3) If a part p_x is part of p_y and part p_y is part of p_z, then p_x is part of p_z (transitivity).

$$\forall p_x : \mathscr{P} \bullet \mathbb{P}(p_x, p_x) \tag{5.1}$$

$$\forall p_x, p_y : \mathscr{P} \bullet (\mathbb{P}(p_x, p_y) \wedge \mathbb{P}(p_y, p_x)) \Rightarrow p_x = p_y \tag{5.2}$$

$$\forall p_x, p_y, p_z : \mathscr{P} \bullet (\mathbb{P}(p_x, p_y) \wedge \mathbb{P}(p_y, p_z)) \Rightarrow \mathbb{P}(p_z, p_z) \tag{5.3}$$

Let $\mathbb{PP}$ denote ***proper part-hood***. p_x is a proper part of p_y is then expressed as $\mathbb{PP}(p_x, p_y)$. $\mathbb{PP}$ can be defined in terms of $\mathbb{P}$. $\mathbb{PP}(p_x, p_y)$ holds if p_x is part of p_y, but p_y is not part of p_x.

$$\mathbb{PP}(p_x, p_y) \triangleq \mathbb{P}(p_x, p_y) \wedge \neg \mathbb{P}(p_y, p_x) \tag{5.4}$$

Overlap, $\mathbb{O}$, expresses a relation between parts. Two parts are said to overlap if they have "something" in common. In classical mereology that 'something' is parts. To us parts are spatial entities and these cannot "overlap". Instead they can 'share' attributes.

$$\mathbb{O}(p_x, p_y) \triangleq \exists a : \mathscr{A} \bullet a \in p_x \wedge a \in p_y \tag{5.5}$$

Underlap, $\mathbb{U}$, expresses a relation between parts. Two parts are said to underlap if there exists a part p_z of which p_x is a part and of which p_y is a part.

$$\mathbb{U}(p_x, p_y) \triangleq \exists p_z : \mathscr{P} \bullet \mathbb{P}(p_x, p_z) \wedge \mathbb{P}(p_y, p_z) \tag{5.6}$$

Think of the underlap p_z as an "umbrella" which both p_x and p_y are "under".
Over-cross, $\mathbb{OX}$, p_x and p_y are said to over-cross if p_x and p_y overlap and p_x is not part of p_y.

$$\mathbb{OX}(p_x, p_y) \triangleq \mathbb{O}(p_x, p_y) \wedge \neg \mathbb{P}(p_x, p_y) \tag{5.7}$$

Under-cross, $\mathbb{UX}$, p_x and p_y are said to under cross if p_x and p_y underlap and p_y is not part of p_x.

$$\mathbb{UX}(p_x, p_y) \triangleq \mathbb{U}(p_x, p_z) \wedge \neg \mathbb{P}(p_y, p_x) \tag{5.8}$$

[4] Our notation now is not RSL but a conventional first-order predicate logic notation.

Proper Overlap, $\mathbb{PO}$, expresses a relation between parts. p_x and p_y are said to properly overlap if p_x and p_y over-cross and if p_y and p_x over-cross.

$$\mathbb{PO}(p_x, p_y) \triangleq \mathbb{OX}(p_x, p_y) \wedge \mathbb{OX}(p_y, p_x) \tag{5.9}$$

Proper Underlap, $\mathbb{PU}$, p_x and p_y are said to properly underlap if p_x and p_y under-cross and p_x and p_y under-cross.

$$\mathbb{PU}(p_x, p_y) \triangleq \mathbb{UX}(p_x, p_y) \wedge \mathbb{UX}(p_y, p_x) \tag{5.10}$$

5.4.3 Satisfaction

We shall sketch a proof that the *model* of the previous section, Sect. 5.3, *satisfies* is a model for the *axioms* of this section. To that end we first define the notions of *interpretation*, *satisfiability*, *validity* and *model*.

Interpretation: By an interpretation of a predicate we mean an assignment of a truth value to the predicate where the assignment may entail an assignment of values, in general, to the terms of the predicate.

Satisfiability: By the satisfiability of a predicate we mean that the predicate is true for some interpretation.

Valid: By the validity of a predicate we mean that the predicate is true for all interpretations.

Model: By a model of a predicate we mean an interpretation for which the predicate holds.

A Proof Sketch

We assign

256 P as the meaning of $\mathscr{P}$

257 ATR as the meaning of $\mathscr{A}$,

258 imm_within as the meaning of $\mathbb{P}$,

259 within as the meaning of $\mathbb{PP}$,

260 $\in_{(\text{of type:}\mathit{ATR}\times\mathit{ATR}-\textbf{set}\to\textbf{Bool})}$ as the meaning of $\in_{(\text{of type:}\mathscr{A}\times\mathscr{P}\to\textbf{Bool})}$ and

261 sharing as the meaning of $\mathbb{O}$.

With the above assignments is is now easy to prove that the other axiom-operators $\mathbb{U}$, $\mathbb{PO}$, $\mathbb{PU}$, $\mathbb{OX}$ and $\mathbb{UX}$ can be modelled by means of imm_within, within, $\in_{(\text{of type:}\mathit{ATR}\times\mathit{ATR}-\textbf{set}\to\textbf{Bool})}$ and sharing.

5.5 An Analysis of Properties of Parts

So far we have not said much about *"the nature"* of parts other than composite parts having one or more subparts and parts having attributes. In preparation also for the next section, Sect. 5.6 we now take a closer look at the concept of 'attributes'. We consider three kinds of attributes: their unique identifications [uid_Π] — which we have already considered; their connections, i.e., their mereology [mereo_P] — which we also considered; and their "other" attributes which we shall refer to as properties. [prop_P]

5.5.1 Mereological Properties

An Example

Road nets, n:N, consists of a set of street intersections (hubs), h:H, uniquely identified by hi's (in HI), and a set of street segments (links), l:L, uniquely identified by li's (in LI). such that from a street segment one can observe a two element set of street intersection identifiers, and from a street intersection one can observe a set of street segment identifiers. Constraints between values of link and hub identifiers must be satisfied. The two element set of street intersection identifiers express that the street segment is connected to exactly two existing and distinct street intersections, and the zero, one or more element set of street segment identifiers express that the street intersection is connected to zero, one or more existing and distinct street segments. An axiom expresses these constraints. We call the hub identifiers of hubs and links, the link identifiers of links and hubs, and their fulfilment of the axiom the connection **mereo**logy.

```
type
    N, H, L, HI, LI
value
    obs_Hs: N→H-set, obs_Ls: N→L-set
    uid_HI: H→HI, uid_LI: L→LI
    mereo_HIs: L→HI-set axiom ∀ l:L•card mereo_HIs(l)=2
    mereo_LIs: H→LI-set
axiom
  ∀ n:N•
      let hs=obs_Hs(n),ls=obs_Ls(n) in
      ∀ h:H•h ∈ hs⇒∀ li:LI•li ∈ mereo_LIs(h)⇒∃ l:L•uid_LI(l)=li
      ∧ ∀ l:L•l ∈ ls⇒∃ h,h′:H•{h,h′}⊆hs∧mereo_HIs(l)={uid_HI(h),uid_HI(h′)}
      end                                                              •
```

Unique Identifier and Mereology Types

In general we allow for any embedded (within) part to be connected to any other embedded part of a composite part or across adjacent composite parts. Thus we must, in general, allow for a family of part types P1, P2, ..., Pn, for a corresponding family of part identifier types Π1, Π2, ..., Πn, and for corresponding observer **u**nique **id**entification and **mer**eology functions:

```
type
    P = P1 | P2 | ... | Pn
    Π = Π1 | Π2 | ... | Πn
value
    uid_Πj: Pj → Πj for 1≤j≤n
    mereo_Πs: P → Π-set
```

Example: Our example relates to the abstract model of Sect. 5.3.

262 With each part we associate a unique identifier, π.

263 And with each part we associate a set, $\{\pi_1, \pi_2, \ldots, \pi_n\}, n \leq 0$ of zero, one ore more other unique identifiers, different from π.

264 Thus with each part we can associate a set of zero, one or more connections, viz.: $\{\pi, \pi_j\}$ for $0 \leq j \leq n$.

type
262. Π
value
262. uid_Π: P → Π
263. mereo_Πs: P → Π-**set**
axiom
263. ∀ p:P•uid_Π(p) ∉ mereo_Πs(p)
value
264. xtr_Ks: P → K-**set**
264. xtr_Ks(p) ≡
264. **let** (π,πs)=(uid_Π,mereo_Πs)(p) **in**
264. {{π′,π″}|π′,π″:Π•π′=π∧π″ ∈ πs} **end**

•

5.5.2 Properties

By the properties of a part we mean such properties additional to those of unique identification and mereology. Perhaps this is a cryptic characterisation. Parts, whether atomic or composite, are there for a purpose. The unique identifications and mereologies of parts are there to refer to and structure (i.e., relate) the parts. So they are there to facilitate the purpose. The properties of parts help towards giving these parts "their final meaning". (We shall support his claim ("their final meaning") in Sect. 5.6.) Let us illustrate the concept of properties.

Examples: (i) Typical properties of street segments are: length, cartographic location, surface material, surface condition, traffic state — whether open in one, the other, both or closed in all directions. (ii) Typical properties of street intersections are: design[5] location, surface material, surface condition, traffic state — open or closed between any two pairs of in/out street segments. (iii) Typical properties of road nets are: name, owner, public/private, free/tool road, area, etcetera. •

265 Parts are characterised (also) by a set of one or more distinctly named and not necessarily distinctly typed property values.

- a Property names are further undefined tokens (i.e., simple quantities).
- b Property types are either sorts or are concrete types such as integers, reals, truth values, enumerated simple tokens, or are structured (sets, Cartesians, lists, maps) or are functional types.
- c From a part
  - i one can observe its sets of property names
  - ii and its set (i.e., enumerable map) of distinctly named and typed property values.
- d Given an property name of a part one can observe the value of that part for that property name.
- e For practical reasons we suggest **prop**erty named **prop**erty value observer function — where we further take the liberty of using the **prop**erty type name in lieu of the **prop**erty name.

type
265. Props = PropNam $\xrightarrow[m]{}$ PropVAL
265a. PropNam
265b. PropVAL
value
265(c)i. obs_Props: P → Props
265(c)ii. xtr_PropNams: P → PropNam-**set**

[5] for example, a simple 'carrefour', or a (circular) roundabout, or a free-way interchange a cloverleaf or a stack or a clover-stack or a turbine or a roundabout or a trumpet or a directional or a full Y or a hybrid interchange.

265(c)ii. xtr_PropNams(p) ≡ **dom** obs_Props(p)
265d. xtr_PropVAL: P → PropNam $\xrightarrow{\sim}$ PropVAL
265d. xtr_PropVAL(p)(pn) ≡ (obs_Props(p))(pn)
265d. **pre**: pn ∈ xtr_PropNams(p)

Here we leave PropNames and PropVALues undefined.

Example:

type
NAME, OWNER, LEN, DESIGN, PP == public | private, ...
LΣ, HΣ, LΩ, HΩ
value
obs_Props: N → {| [″name″↦nm,″owner″↦ow,″public/private″↦pp,...]
| nm:NAME, ow:OWNER, ..., pp:PP |}
obs_Props: L → {| [″length″↦len,...,″state″↦lσ,″state space″↦lω:LΩ]
| **len**:LEN,...,lσ:LΣ,lω:LΩ |}
obs_Props: H → {| [″design″↦des, ...,″state″↦hσ,″state space″↦hω]
| des:DESIGN,...,hσ:HΣ,hω:HΩ |}
prop_NAME: N → NAME
prop_OWNER: N → OWNER
prop_LEN: L → LEN
prop_LΣ: L → LΣ, obs_LΩ: L → LΩ
prop_DESIGN: H → DESIGN
prop_HΣ: H → HΣ, obs_HΩ: H → HΩ
...

We trust that the reader can decipher this example. •

5.5.3 Attributes

There are (thus) three kinds of part attributes:

- **u**nique **id**entifier "observers" (uid_),
- **mereo**logy "observers (**mereo_**), and
- **prop**erty "observers" (**prop_...**, obs_Props)

We refer to Sect. 5.3.4, and to Items 240–241.

type
240.′ ATR = Π × Π-**set** × Props
value
241.′ atr_ATR: P → ATR
axiom
∀ p:P • **let** (π,πs,props) = atr_ATR(p) **in** π ∉ πs **end**

In preparation for redefining the share function of Item 242 on Page 127 we must first introduce a modification to property values.

266 A property value, pv:PropVal, is either a simple property value (as was hitherto assumed), or is a unique part identifier.

type
265. Props = PropNam $\xrightarrow[m]{}$ PropVAL_or_Π
266. PropVAL_or_Π :: mk_Simp:PropVAL | mk_Π:Π

267 The idea a property name *pn*, of a part p', designating a Π-valued property value π is

a that π refers to a part p'

b one of whose property names must be pn

c and whose corresponding property value must be a proper, i.e., simple property value, v,

d which is then the property value in p' for pn.

```
value
267. get_VAL: P × PropName → W → PropVAL
267. get_VAL(p,pn)(w) ≡
269.    let pv = (obs_Props(p))(pn) in
267.    case pv of
267.       mk_Simp(v) → v,
267a.      mk_Π(π) →
267a.         let p′:P•p′ ∈ xtr_Ps(w)∧uid_Π(p′)=π in
267c.         (obs_Props(p′))(pn) end
267.    end end
267c.    pre: pn ∈ obs_PropNams(p)
267b.       ∧ pn ∈ obs_PropNams(p′)
267c.       ∧ is_PropVAL((obs_Props(p′))(pn))
```

The three bottom lines above, Items 267b–267c, imply the general constraint now formulated.

268 We now express a constraint on our modelling of attributes.

a Let the attributes of a part p be $(\pi, \pi s, \mathsf{props})$.

b If a property name pn in props has (associates to) a Π value, say π'

c then π' must be in πs.

d and there must exist another part, p', distinct from p, with unique identifier π', such that

e it has some property named pn with a simple property value.

```
value
268.  wf_ATR: ATR → W → Bool
268a.  wf_ATR(π,πs,props)(w) ≡
268a.    π ∉ πs ∧
268b.    ∀ π′:Π • π′ ∈ rng props ⇒
268c.       let pn:PropNam•props(pn)=π′ in
268c.       pi′∈ πs
268d.     ∧ ∃ p′:P•p′∈ xtr_Ps(w)∧uid_Π(p′)=π′ ⇒
268e.          pn ∈ obs_PropNams(obs_Props(p′))
268e.        ∧ ∃ mk_SimpVAL(v):VAL•(obs_Props(p′))(pn)=mk_SimpVAL(v) end
```

269 Two distinct parts share attributes

a if the unique part identifier of one of the parts is in the mereology of the other part, or

b if a property value of one of the parts refers to a property of the other part.

```
value
269.   share: P × P → Bool
269.   share(p,p′) ≡
269.     p ≠ p′ ∧
269.     let (π,πs,props) = atr_ATR(p),(π′,πs′,props′) = atr_ATR(p′),
269.         pns = xtr_PropNams(p), pns′ = xtr_PropNams(p′) in
269a.    π ∈ πs′ ∨ π′ ∈ πs ∨
269b.    ∃ pn:PropNam•pn ∈ pns ∩ pns′ ⇒
269b.      let vop = props(pn), vop′ = props′(pn) in
```

```
269b.    case (vop,vop′) of
269b.      (mk_Π(π″),mk_Simp(v)) → π″=π′,
269b.      (mk_Simp(v),mk_Π(π″)) → π=π″,
269b.      _ → false
269.     end end end
```

Comment: *v* is a shared attribute.

5.5.4 Discussion

We have now witnessed four kinds of observer function:

- he above three kinds of mereology and property 'observers' and the
- part (and subpart) **obs**_ervers.

These observer functions are postulated. They cannot be defined. They "just exist" by the force of our ability to observe and decide upon their values when applied by us, the domain observers.

Parts are either composite or atomic. Analytic functions are postulated. They help us decide whether a part is composite or atomic, and, from composite parts their immediate subparts.

Both atomic and composite parts have all three kinds of attributes: unique identification, mereology (connections), and properties. Analytic functions help us observe, from a part, its unique identification, its mereology, and its properties.

Some attribute values may be static, that is, constant, others may be inert dynamic, that is, can be changed. It is exactly the inert dynamic attributes which are the basis for the next sections semantic model of parts as processes.

In the above model (of this and Sect. 5.3) we have not modelled distinctions between static and dynamic properties. You may think, instead of such a model, that an **always** temporal operator, □, being applied to appropriate predicates.

5.6 A Semantic CSP Model of Mereology

The model of Sect. 5.3 can be said to be an abstract model-oriented definition of the syntax of mereology. Similarly the axiom system of Sect. 5.4 can be said to be an abstract property-oriented definition of the syntax of mereology. With the analysis of attributes of parts, Sect. 5.5, we have begun a semantic analysis of mereology. We now bring that semantic analysis a step further.

5.6.1 A Semantic Model of a Class of Mereologies

We show that to every mereology there corresponds a program of cooperating sequential processes CSP. We assume that the reader has practical knowledge of Hoare's CSP [97].

Parts ≃ Processes

The model of mereology presented in Sect. 5.3 (Pages 123–129) focused on (i) parts and (ii) connectors. To parts we associate CSP processes. Part processes are indexed by the unique part identifiers. The connectors form the mereological attributes of the model.

Connectors ≃ Channels

The CSP channels are indexed by the two-set (hence distinct) part identifier connectors. From a whole we can extract (xtr_Ks, Item 251 on Page 128) all connectors. They become indexes into an array of channels. Each of the connector channel index identifiers indexes exactly two part processes. Let w:W be the whole under analysis.

```
value
  w:W
  ps:P-set = ∪{xtr_Ps(c)|c:C•c ∈ w} ∪ {a|a:A•a ∈ w}
  ks:K-set = xtr_Ks(w)
type
```

```
  K = Π-set axiom ∀ k:K•card k=2
  ChMap = Π →m K-set
value
  cm:ChMap = [uid_Π(p)↦xtr_Ks(p)|p:P•p ∈ ps]
channel
  ch[k|k:K•k ∈ ks] MSG
```

We leave channel messages. m:MSG, undefined.

Process Definitions

```
value
  system: W → process
  system(w) ≡
      ||{comp_process(uid_Π(c))(c)|c:C•c ∈ w} || ||{atom_process(uid_Π(a),a)|a:A•a ∈ w}

  comp_process: π:Π → c:C→ in,out {ch(k)|k:K•k ∈ cm(π)} process
  comp_process(π)(c) ≡ [ assert: π = uid_Π(c) ]
     𝓜𝒞(π)(c)(atr_ATR(c)) ||
     || {comp_process(uid_Π(c′))(c′)|c′:C•c′ ∈ obs_Ps(c)} ||
     || {atom_process(uid_Π(a))(a)|a:A•a ∈ obs_Ps(c)}

  𝓜𝒞: π:Π → C → ATR → in,out {ch(k)|k:K•k ∈ cm(pi)} process
  𝓜𝒞(π)(c)(c_attrs) ≡ 𝓜𝒞(c)(C𝓕(c)(c_attrs))  assert: atr_ATR(c) ≡ c_attrs


  C𝓕: c:C → ATR → in,out {ch[em(i)]|i:KI•i ∈ cm(uid_Π(c))} ATR
```

ATR and atr_ATR are defined in Items 240.′ and 241.′ (Page 134).

```
  atom_process: a:A → in,out {ch[cm(k)]|:K•k ∈ cm(uid_Π(a))} process
  atom_process(a) ≡ 𝓜𝒜(a)(atr_ATR(a))

  𝓜𝒜: a:A → ATR → in,out {ch[cm(k)]|k:K•k ∈ cm(uid_Π(a))} process
  𝓜𝒜(a)(a_attrs) ≡ 𝓜𝒜(a)(A𝓕(a)(a_attrs))  assert: atr_ATR(a) ≡ a_attrs

  A𝓕: a:A → ATR → in,out {ch[em(k)]|k:K • k ∈ cm(uid_Π(a))} ATR
```

The meaning processes $\mathscr{M}_{\mathscr{C}}$ and $\mathscr{M}_{\mathscr{A}}$ are generic. Their sôle purpose is to provide a never ending recursion. "In-between" they "make use" of *C*omposite, respectively *A*tomic specific $\mathscr{F}$unctions here symbolised by $C\mathscr{F}$, respectively $A\mathscr{F}$.

Both $C\mathscr{F}$ and $A\mathscr{F}$ are expected to contain input/output clauses referencing the channels of their signatures; these clauses enable the sharing of attributes. We illustrate this "sharing" by the schematised function $\mathscr{F}$ standing for either $C\mathscr{F}$ or $A\mathscr{F}$.

```
  value
      𝓕: p:(C|A) → ATR → in,out {ch[em(k)]|k:K • k ∈ cm(uid_Π(p))} ATR
      𝓕(p)(π,πs,props) ≡
          [] {let av = ch[em({π,j})] ? in
                ... ; [optional] ch[em({π,j})] ! in_reply(props)(av);
                    (π,πs,in_update_ATR(props)(j,av)) end | {π,j}:K•{π,j} ∈ πs}
        ⌈⌉ [] { ... ; ch[em({π,j})] ! out_reply(props);
                (π,πs,out_update_ATR(props)(j)) | {π,j}:K•{π,j} ∈ πs}
          ⌈⌉ (π,πs,own_work(props))
      assert: π = uid_Π(p)
```

in_reply: Props → Π × VAL → VAL
in_update_ATR: Props → Π × VAL → Props
out_reply: Props → VAL
out_update_ATR: Props →Π → Props
own_work: Props → Props

We leave VAL undefined.

5.6.2 Discussion

General

A little more meaning has been added to the notions of parts and connections. The within and adjacent to relations between parts (composite and atomic) reflect a phenomenological world of geometry, and the connected relation between parts reflect both physical and conceptual world understandings: physical world in that, for example, radio waves cross geometric "boundaries", and conceptual world in that ontological classifications typically reflect lattice orderings where *overlaps* likewise cross geometric "boundaries".

Partial Evaluation

The composite_processes function "first" "functions" as a compiler. The 'compiler' translates an assembly structure into three process expressions: the $\mathscr{M}_{\mathscr{C}}$(c)(c_attrs) invocation, the parallel composition of composite processes, c', one for each composite sub-part of c, and the parallel composition of atomic processes, a, one for each atomic sub-part of c — with these three process expressions "being put in parallel". The recursion in composite_processes ends when a sub-...-composites consist of no sub-sub-...-composites. Then the compiling task ends and the many generated $\mathscr{M}_{\mathscr{C}}$(c)(c_attrs) and $\mathscr{M}_{\mathscr{A}}$(a)(a_attrs) process expressions are invoked.

5.7 Concluding Remarks

5.7.1 Relation to Other Work

The present contribution has been conceived in the following context.

My first awareness of the concept of 'mereology' was from listening to many presentations by **Douglas T. Ross** (1929–2007) at IFIP working group WG3.2 meetings over the years 1980–1999. In [137] Douglas T. Ross and John E. Ward reports on the 1958–1967 MIT project for *computer-aided design (CAD) for numerically controlled production*.[6] Pages 13–17 of [137] reflects on issues bordering to and behind the concerns of mereology. Ross' thinking is clearly seen in the following text: *"... our consideration of fundamentals begins not with design or problem-solving or programming or even mathematics, but with philosophy (in the old-fashioned meaning of the word) – we begin by establishing a "world-view". We have repeatedly emphasized that there is no way to bound or delimit the potential areas of application of our system, and that we must be prepared to cope with any conceivable problem. Whether the system will assist in any way in the solution of a given problem is quite another matter, ..., but in order to have a firm and uniform foundation, we must have a uniform philosophical basis upon which to approach any given problem. This "world-view" must provide a working framework and methodology in terms of which any aspect of our awareness of the world may be viewed. It must be capable of expressing the utmost in reality, giving expression to unending layers of ever-finer and more concrete detail, but at the same time abstract chimerical visions bordering on unreality must fall within the same scheme. "Above all, the world-view itself must be concrete and workable, for it will form the basis for all involvement of the computer in the problem-solving process, as well as establishing a viewpoint for approaching the unknown human component of the problem-solving team."* Yes, indeed, the philosophical disciplines of ontology, epistemology and mereology, amongst others, ought be standard curricula items in the computer science and software engineering studies, or better: domain engineers cum software system designers ought be imbued by the wisdom of those disciplines as was Doug. *"... in the summer of 1960 we coined the word plex to serve as a generic term for these philosophical ruminations. "Plex" derives from the word plexus, "An interwoven combination of parts in a structure", (Webster). ... The purpose of a* ***'modeling***

[6] Doug is said to have coined the term and the abbreviation CAD [135].

plex' *is to represent completely and in its entirety a "thing", whether it is concrete or abstract, physical or conceptual. A 'modeling plex' is a trinity with three primary aspects, all of which must be present. If any one is missing a complete representation or modeling is impossible. The three aspects of plex are* ***data, structure,*** *and* ***algorithm.*** *. . . "* which *". . . is concerned with the behavioral characteristics of the plex model– the interpretive rules for making meaningful the data and structural aspects of the plex, for assembling specific instances of the plex, and for interrelating the plex with other plexes and operators on plexes. Specification of the algorithmic aspect removes the ambiguity of meaning and interpretation of the data structure and provides a complete representation of the thing being modeled."* In the terminology of the current paper a plex is a part (whether composite or atomic), the data are the properties (of that part), the structure is the mereology (of that part) and the algorithm is the process (for that part). Thus Ross was, perhaps, a first instigator (around 1960) of object-orientedness. A first, "top of the iceberg" account of the mereology-ideas that Doug had then can be found in the much later (1976) three page note [136]. Doug not only 'invented' CAD but was also the father of AED (Algol Extended for Design), the Automatically Programmed Tool (APT) language, SADT (Structured Analysis and Design Technique) and helped develop SADT into the IDEF0 method for the Air Force's Integrated Computer-Aided Manufacturing (ICAM) program's IDEF suite of analysis and design methods. Douglas T. Ross went on for many years thereafter, to deepen and expand his ideas of relations between mereology and the programming language concept of type at the IFIP WG2.3 working group meetings. He did so in the, to some, enigmatic, but always fascinating style you find on Page 63 of [136].

In [114] **Henry S. Leonard** and **Henry Nelson Goodman:** *A Calculus of Individuals and Its Uses* present the American Pragmatist version of Leśniewski's mereology. It is based on a single primitive: discreet, ⟧. The idea the calculus of individuals is, as in Leśniewski's mereology, to avoid having to deal with the empty sets while relying on explicit reference to classes (or parts).

[57] **R. Casati** and **A. Varzi:** *Parts and Places: the structures of spatial representation* has been the major source for this paper's understanding of mereology. Although our motivation was not the spatial or topological mereology, [143], and although the present paper does not utilize any of these concepts' axiomatision in [57, 143] it is best to say that it has benefitted much from these publications.

Domain descriptions, besides mereological notions, also depend, in their successful form. on FCA: Formal Concept Analysis. Here a main inspiration has been drawn , since the mid 1990s from **B. Ganter** and **R. Wille's** *Formal Concept Analysis — Mathematical Foundations* [84]. *The approach takes as input a matrix specifying a set of objects and the properties thereof, called attributes, and finds both all the "natural" clusters of attributes and all the "natural" clusters of objects in the input data, where a "natural" object cluster is the set of all objects that share a common subset of attributes, and a "natural" property cluster is the set of all attributes shared by one of the natural object clusters. Natural property clusters correspond one-for-one with natural object clusters, and a concept is a pair containing both a natural property cluster and its corresponding natural object cluster. The family of these concepts obeys the mathematical axioms defining a lattice, a Galois connection).* Thus the choice of adjacent and embedded ('within') parts and their connections is determined after serious formal concept analysis. In [47] we present a 'concept analysis' approach to domain description, where the present paper presents the mereological approach.

The present paper is based on [29] of which it is an extensive revision and extension.

5.7.2 What Has Been Achieved ?

We have given a model-oriented specification of mereology. We have indicated that the model satisfies a widely known axiom system for mereology. We have suggested that (perhaps most) work on mereology amounts to syntactic studies. So we have suggested one of a large number of possible, schematic semantics of mereology. And we have shown that to every mereology there corresponds a set of communicating sequential process (CSP).

5.7.3 Future Work

We need to characterise, in a proper way, the class of CSP programs for which there corresponds a mereology. Are you game ?

One could also wish for an extensive editing and publication of Doug Ross' surviving notes.

Chapter 6

A Domain Description

310

Chapter Status

The domain description will be augmented.
A model or routes will be added.

6.1 Endurants

6.1.1 Domain, Net, Fleet and Monitor

The root domain, $\Delta_{\mathscr{D}}$, whose description is to be exemplified, is that of a composite traffic system (270a.) with a road net, (270b.) with a fleet of vehicles and (270c.) of whose individual position on the road net we
311 can speak, that is, monitor.

270 We analyse the traffic system into

- a a composite road net,
- b a composite fleet (of vehicles), and
- c an atomic monitor.

271 The road net consists of two composite parts,

- a an aggregation of hubs and
- b an aggregation of links.

312

type
270. Δ_Δ
270a. N_Δ
270b. F_Δ
270c. M_Δ
value
270a. **obs_part_**N_Δ: $\Delta_\Delta \to N_\Delta$
270b. **obs_part_**F_Δ: $\Delta_\Delta \to F_\Delta$
270c. **obs_part_**M_Δ: $\Delta_\Delta \to M_\Delta$
type
271a. HA_Δ
271b. LA_Δ
value

271a. **obs_part**_HA$_\Delta$: N$_\Delta$ → HA$_\Delta$
271b. **obs_part**_LA$_\Delta$: N$_\Delta$ → LA$_\Delta$

6.1.2 Hubs and Links

313

272 Hub aggregates are sets of hubs.

273 Link aggregates are sets of links.

274 Fleets are set of vehicles.

275 We introduce some auxiliary functions.

- a links extracts the links of a network.
- b hubs extracts the hubs of a network.

314

type
272. H$_\Delta$, HS$_\Delta$ = H$_\Delta$-**set**
273. L$_\Delta$, LS$_\Delta$ = L$_\Delta$-**set**
274. V$_\Delta$, VS$_\Delta$ = V$_\Delta$-**set**
value
272. **obs_part**_HS$_\Delta$: HA$_\Delta$ → HS$_\Delta$
273. **obs_part**_LS$_\Delta$: LA$_\Delta$ → LS$_\Delta$
274. **obs_part**_VS$_\Delta$: F$_\Delta$ → VS$_\Delta$
275a. links$_\Delta$: Δ_Δ → L-**set**
275a. links$_\Delta$(δ_Δ) ≡ **obs_part**_LS(**obs_part**_LA(δ_Δ))
275b. hubs$_\Delta$: Δ_Δ → H-**set**
275b. hubs$_\Delta$(δ_Δ) ≡ **obs_part**_HS(**obs_part**_HA(δ_Δ))

6.1.3 Unique Identfiers

315

We cover the unique identifiers of all parts, whether needed or not.

276 Nets, hub and link aggregates, hubs and links, fleets, vehicles and the monitor all

- a have unique identifiers
- b such that all such are distinct, and
- c with corresponding observers.

277 We introduce some auxiliary functions:

- a xtr_lis extracts all link identifiers of a traffic system.
- b xtr_his extracts all hub identifiers of a traffic system.
- c given an appropriate link identifier and a net get_link 'retrieves' the designated link.
- d given an appropriate hub identifier and a net get_hub 'retrieves' the designated hub.

316

type
276a. NI, HAI, LAI, HI, LI, FI, VI, MI
value
276c. **uid**_NI: N$_\Delta$ → NI
276c. **uid**_HAI: HA$_\Delta$ → HAI
276c. **uid**_LAI: LA$_\Delta$ → LAI
276c. **uid**_HI: H$_\Delta$ → HI
276c. **uid**_LI: L$_\Delta$ → LI
276c. **uid**_FI: F$_\Delta$ → FI

276c. **uid_**VI: $V_\Delta \to$ VI
276c. **uid_**MI: $M_\Delta \to$ MI
axiom
276b. NI$\cap$HAI=Ø, NI$\cap$LAI=Ø, NI$\cap$HI=Ø, etc.

317 where axiom 276b. is expressed semi-formally, in mathematics.

value
277a. xtr_lis: $\Delta_\Delta \to$ LI-**set**
277a. xtr_lis(δ_Δ) $\equiv$
277a. **let** ls = links(δ_Δ) **in** {**uid_**LI(l)|l:L•l $\in$ ls} **end**
277b. xtr_his: $\Delta_\Delta \to$ HI-**set**
277b. xtr_his(δ_Δ) $\equiv$
277b. **let** hs = hubs(δ_Δ) **in** {**uid_**HI(h)|h:H•k $\in$ hs} **end**
277c. get_link: LI $\to \Delta_\Delta \xrightarrow{\sim}$ L
277c. get_link(li)(δ_Δ) $\equiv$
277c. **let** ls = links(δ_Δ) **in**
277c. **let** l:L • l $\in$ ls $\wedge$ li=**uid_**LI(l) **in** l **end end**
277c. **pre**: li $\in$ xtr_lis(δ_Δ)
277d. get_hub: HI $\to \Delta_\Delta \xrightarrow{\sim}$ H
277d. get_hub(hi)(δ_Δ) $\equiv$
277d. **let** hs = hubs(δ_Δ) **in**
277d. **let** h:H • h $\in$ hs $\wedge$ hi=**uid_**HI(h) **in** h **end end**
277d. **pre**: hi $\in$ xtr_his(δ_Δ)

6.1.4 Mereology

318

We cover the mereologies of all part sorts introduced so far. We decide that nets, hub aggregates, link aggregates and fleets have no mereologies of interest.

278 Hub mereologies reflect that they are connected to zero, one or more links.

279 Link mereologies reflect that they are connected to exactly two distinct hubs.

280 Vehicle mereologies reflect that they are connected to the monitor.

281 The monitor mereology reflects that it is connected to all vehicles.

282 For all hubs of any net it must be the case that their mereology designates links of that net.

283 For all links of any net it must be the case that their mereologies designates hubs of that net.

284 For all transport domains it must be the case that

a the mereology of vehicles of that system designates the monitor of that system, and that

b the mereology of the monitor of that system designates vehicles of that system.

319

value
278. **obs_mereo_**H_Δ: $H_\Delta \to$ LI-**set**
279. **obs_mereo_**L: L $\to$ HI-**set axiom** $\forall$ l:L•**card obs_mereo_**L(l)=2
280. **obs_mereo_**V: V $\to$ MI
281. **obs_mereo_**M: M $\to$ VI-**set**
axiom
282. $\forall$ δ:Δ, hs:HS_Δ•hs=hubs(δ), ls:LS_Δ•ls=links(δ) •
282. $\forall$ h:H_Δ•h $\in$ hs•**obs_mereo_**H(h)$\subseteq$xtr_his(δ) $\wedge$
283. $\forall$ l:L_Δ•l $\in$ ls•**obs_mereo_**L(l)$\subseteq$xtr_lis(δ) $\wedge$
284a. **let** f:F_Δ•f=**obs_part_**F(δ) $\Rightarrow$
284a. **let** m:M_Δ•m=**obs_part_**M(δ),
284a. vs:VS•vs=**obs_part_**VS(f) **in**

284a. ∀ v:V_Δ•v ∈ vs⇒**uid_**V(v) ∈ **obs_mereo_**M(m)
284b. ∧ **obs_mereo_**M(m) = {**uid_**V(v)|v:V•v ∈ vs}
284b. **end end**

6.1.5 Attributes, I

320

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

- **Hubs:** *location*s[1] are considered static, *wear and tear* (condition of road surface) is considered inert, *hub state*s and *hub state space*s are considered programmable;
- **Links:** *length*s and *location*s are considered static, *wear and tear* (condition of road surface) is considered inert, *link state*s and *link state space*s are considered programmable;
- **Vehicles:** *manufacturer name*, *engine type* (whether diesel, gasoline or electric) and *engine power* (kW/horse power) are considered static; *velocity* and *acceleration* may be considered reactive (i.e., a function of gas pedal position, etc.), *global position* (informed via a `GNSS: Global Navigation Satellite System`) and *local position* (calculated from a global position) are considered biddable

321

6.1.6 Attributes, II

322

We treat one attribute each for hubs, links, vehicles and the monitor. First we treat hubs.

285 Hubs
  a have *hub states* which are sets of pairs of identifiers of links connected to the hub[2],
  b and have *hub state spaces* which are sets of hub states[3].
286 For every net,
  a link identifiers of a hub state must designate links of that net.
  b Every hub state of a net must be in the hub state space of that hub.
287 Hubs have geodetic and cadestral location.
288 We introduce an auxiliary function: xtr_lis extracts all link identifiers of a hub state.

323

type
285a. HΣ = (LI×LI)-**set**
285b. HΩ = HΣ-**set**
value
285a. **attr_**HΣ: H → HΣ
285b. **attr_**HΩ: H → HΩ
axiom
286. ∀ δ:Δ,
286. **let** hs = hubs(δ) **in**
286. ∀ h:H • h ∈ hs •
286a. xtr_lis(h)⊆xtr_lis(δ)
286b. ∧ **attr_**Σ(h) ∈ **attr_**Ω(h)
286. **end**
type
287. HGCL
value
287. **attr_**HGCL: H → HGCL
288. xtr_lis: H → LI-**set**

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

```
288. xtr_lis(h) ≡
288.     {li | li:LI,(li′,li″):LI×LI •
288.         (li′,li″) ∈ attr_HΣ(h) ∧ li ∈ {li′,li″}}
```

Then links.

289 Links have lengths.

290 Links have geodetic and cadestral location.

291 Links have states and state spaces:

- a States modeled here as pairs, (hi',hi''), of identifiers the hubs with which the links are connected and indicating directions (from hub h' to hub h''.) A link state can thus have 0, 1, 2, 3 or 4 such pairs.
- b State spaces are the set of all the link states that a link may enjoy.

```
type
289.  LEN
290.  LGCL
291a. LΣ = (HI×HI)-set
291b. LΩ = LΣ-set
value
289.  attr_LEN: L → LEN
290.  attr_LGCL: L → LGCL
291a. attr_LΣ: L → LΣ
291b. attr_LΩ: L → LΩ
axiom
291.  ∀ n:N •
291.    let ls = xtr−links(n), hs = xtr_hubs(n) in
291.    ∀ l:L•l ∈ ls ⇒
291a.     let lσ = attr_LΣ(l) in
291a.     0≤card lσ≤4
291a.   ∧ ∀ (hi′,hi″):(HI×HI)•(hi′,hi″) ∈ lσ ⇒
291a.       {get_H(hi′)(n),get_H(hi″)(n)}=obs_mereo_L(l)
291b.   ∧ attr_LΣ(l) ∈ attr_LΩ(l)
291.    end end
```

Then vehicles.

292 Every vehicle of a traffic system has a position which is either ‘on a link’ or ‘at a hub’.

- a An ‘on a link’ position has four elements: a unique link identifier which must designate a link of that traffic system and a pair of unique hub identifiers which must be those of the mereology of that link.
- b The ‘on a link’ position real is the fraction, thus properly between 0 (zero) and 1 (one) of the length from the first identified hub “down the link” to the second identifier hub.
- c An ‘at a hub’ position has three elements: a unique hub identifier and a pair of unique link identifiers — which must be in the hub state.

```
type
292.  VPos = onL | atH
292a.  onL :: LI HI HI R
292b.  R = Real      axiom ∀ r:R • 0≤r≤1
292c.  atH :: HI LI LI
value
292.  attr_VPos: V_Δ → VPos
axiom
292a.  ∀ n_Δ:N_Δ, onL(li,fhi,thi,r):VPos •
```

292a. ∃ l_Δ:L_Δ•l_Δ∈**obs_part_**LS(**obs_part_**N_Δ(n_Δ))
292a. ⇒ li=**uid_**L_Δ(l)∧{fhi,thi}=**obs_mereo_**L_Δ(l_Δ),
292c. ∀ n_Δ:N_Δ, atH(hi,fli,tli):VPos •
292c. ∃ h_Δ:H_Δ•h_Δ∈**obs_part_**HS_Δ(**obs_part_**N(n_Δ))
292c. ⇒ hi=**uid_**H_Δ(h_Δ)∧(fli,tli) ∈ **attr_**LΣ(h_Δ)

328

293 We introduce an auxiliary function `distribute`.

- a `distribute` takes a net and a set of vehicles and
- b generates a map from vehicles to distinct vehicle positions on the net.
- c We sketch a "formal" `distribute` function, but, for simplicity we omit the technical details that secures distinctness — and leave that to an axiom !

294 We define two auxiliary functions:

- a xtr_links extracts all links of a net and
- b xtr_hub extracts all hubs of a net.

329

type
293b. MAP = VI $\xrightarrow{m}$ VPos
293b. ∀ map:MAP • **card dom** map = **card rng** map
value
293. distribute: VS_Δ → N_Δ → MAP
293. distribute(vs_Δ)(n_Δ) ≡
293a. **let** (hs,ls) = (xtr_hubs(n_Δ),xtr_links(n_Δ)) **in**
293a. **let** vps = {onL(**uid_**(l_Δ),fhi,thi,r)|l_Δ:L_Δ•l_Δ∈ls∧{fhi,thi}⊆**obs_mereo_**L(l)∧0≤r≤1}
293a. ∪ {atH(**uid_**H(h_Δ),fli,tli)|h_Δ:H_Δ•h_Δ∈hs∧{fli,tli}⊆**obs_mereo_**H_Δ(h_Δ)} **in**
293b. [**uid_**V_Δ(v)↦vp|v_Δ:V_Δ,vp:VPos•v_Δ∈vs∧vp∈vps]
293. **end end**

330

294a. xtr_links$_\Delta$: N_Δ → L_Δ**-set**
294a. xtr_links$_\Delta$(n_Δ)≡**obs_part_**LS(**obs_part_**LA(n_Δ))
294b. xtr_hubs$_\Delta$: N_Δ → H_Δ**-set**
294a. xtr_hubs$_\Delta$(n_Δ)≡**obs_part_**HS_Δ(**obs_part_**HA_Δ(n_Δ))

331

And finally monitors. We consider only one monitor attribute.

295 The monitor has a vehicle traffic attribute.

- a For every vehicle of the road transport system the vehicle traffic attribute records a possibly empty list of time marked vehicle positions.
- b These vehicle positions are alternate sequences of 'on link' and 'at hub' positions
  - i such that any sub-sequence of 'on link' positions record the same link identifier, the same pair of ''to' and 'from' hub identifiers and increasing fractions,
  - ii such that any sub-segment of 'at hub' positions are identical,
  - iii such that vehicle transition from a link to a hub is commensurate with the link and hub mereologies, and
  - iv such that vehicle transition from a hub to a link is commensurate with the hub and link mereologies.

332

type
295. Traffic = VI $\xrightarrow{m}$ (T × VPos)*
value
295. **attr_**Traffic: M → Traffic
axiom

```
295b.  ∀ δ:Δ •
295b.    let m = obs_part_M_Δ(δ) in
295b.    let tf = attr_Traffic(m) in
295b.    dom tf ⊆ xtr_vis(δ) ∧
295b.    ∀ vi:VI • vi ∈ dom tf •
295b.      let tr = tf(vi) in
295b.      ∀ i,i+1:Nat • {i,i+1}⊆dom tr •
295b.        let (t,vp)=tr(i),(t′,vp′)=tr(i+1) in
295b.        t<t′
295(b)i.       ∧ case (vp,vp′) of
295(b)i.           (onL(li,fhi,thi,r),onL(li′,fhi′,thi′,r′))
295(b)i.             → li=li′∧fhi=fhi′∧thi=thi′∧r≤r′
295(b)i.               ∧ li ∈ xtr_lis(δ)
295(b)i.               ∧ {fhi,thi} = obs_mereo_L(get_link(li)(δ)),
295(b)ii.          (atH(hi,fli,tli),atH(hi′,fli′,tli′))
295(b)ii.            → hi=hi′∧fli=fli′∧tli=tli′
295(b)ii.            ∧ hi ∈ xtr_his(δ)
295(b)ii.            ∧ (fli,tli) ∈ obs_mereo_H(get_hub(hi)(δ)),
295(b)iii.         (onL(li,fhi,thi,1),atH(hi,fli,tli))
295(b)iii.           → li=fli∧thi=hi
295(b)iii.           ∧ {li,tli} ⊆ xtr_lis(δ)
295(b)iii.           ∧ {fhi,thi}=obs_mereo_L(get_link(li)(δ))
295(b)iii.           ∧ hi ∈ xtr_his(δ)
295(b)iii.           ∧ (fli,tli) ∈ obs_mereo_H(get_hub(hi)(δ)),
295(b)iv.          (atH(hi,fli,tli),onL(li′,fhi′,thi′,0))
295(b)iv.            → etcetera,
295b.            _ → false
295b.    end end end end end
```

6.1.7 Routes 333

We bring a model of routes.

TO BE WRITTEN

6.2 Perdurants 334

6.2.1 Vehicle to Monitor Channel

296 Let δ be the traffic system domain.

297 Then focus on the set of vehicles

298 and the monitor —

299 and we obtain an appropriate channel array for communication between vehicles and the traffic observing monitor.

```
value
297.  let vs:VS • vs = obs_part_VS(obs_part_F(δ)),
298.      m:M • m = obs_part_M(δ) in
channel
299.  {v_m_ch[uid_VI(v),uid_MI(m)]|v:V•v ∈ vs} end
```

6.2.2 Link Disappearance Event 335

We formalise aspects of the above-mentioned link disappearance event:

300 The result net, n':N', is not well-formed.

301 For a link to disappear there must be at least one link in the net;

302 and such a link may disappear such that

303 it together with the resulting net makes up for the "original" net.

```
value
300. link_diss_event: N × N′ × Bool
300. link_diss_event(n,n′) as tf
301.   pre: obs_part_Ls(obs_part_LS(n))≠{}
302.   post: ∃ l:L•l ∈ obs_part_Ls(obs_part_LS(n)) ⇒
303.         l ∉ obs_part_Ls(obs_part_LS(n′))
303.         ∧ n′ ∪ {l} = obs_part_Ls(obs_part_LS(n))
```

6.2.3 Road Traffic

336

Global Values There is given some globally observable parts.

304 besides the domain, δ_Δ:Δ_Δ,

305 a net, n:N,

306 a set of vehicles, vs:V-**set**,

307 a monitor, m:M, and

308 a clock, clock, behaviour.

309 From the net and vehicles we generate an initial distribution of positions of vehicles.

The n:N, vs:V-**set** and m:M are observable from any road traffic system domain δ.

337

```
value
304. δΔ:ΔΔ
305. n:N = obs_part_N(δΔ),
305. ls:L-set=linksLs(δ),hs:H-set=hubs(δΔ),
305. lis:LI-set=xtr_lis(δ),his:HI-set=xtr_his(δΔ)
306. vs:V-set=obs_part_Vs(obs_part_VS(obs_part_F(δ)Δ)),
306. vis:VI-set = {uid_VI(v)|v:V•v ∈ vs},
307. m:obs_part_M(δ), mi=uid_MI(m), ma:attributes(m)
308. clock: 𝕋 → out {clk_ch[vi]|vi:VI•vi ∈ vis} Unit
309. vm:MAP•vpos_map = distribute(vs)(n);
```

338

Channels

310 We additionally declare a set of vehicle to monitor channels indexed

a by the unique identifiers of vehicles

b and the (single) monitor identifier.[4]

and communicating vehicle positions.

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

339

Behaviour Signatures

311 The road traffic system behaviour, rts, takes no arguments (hence the first **Unit**)[5]; and "behaves", that is, continues forever (hence the last **Unit**).

312 The vehicle behaviour

a is indexed by the unique identifier, uid_V(v):VI,

b the vehicle mereology, in this case the single monitor identifier mi:MI,

c the vehicle attributes, obs_attribs(v)

d and — factoring out one of the vehicle attributes — the current vehicle position.

e The vehicle behaviour offers communication to the monitor behaviour (on channel vm_ch[vi]); and behaves "forever".

340

313 The monitor behaviour takes

a the monitor identifier,

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

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

b the monitor mereology,

c the monitor attributes,

d and — factoring out one of the vehicle attributes — the discrete road traffic, drtf:dRTF, being repeatedly "updated" as the result of **in**put communications from (all) vehicles;

e the behaviour otherwise behaves forever.

```
value
311.  trs: Unit → Unit
312.  veh_Δ: vi:VI × mi:MI → vp:VPos →
312.              out vm_ch[vi,mi] Unit
313.  mon_Δ: m:M_Δ × vis:VI-set → RTF →
313.              in {v_m_ch[vi,mi]|vi:VI•vi ∈ vis},clk_ch Unit
```

341

The Road Traffic System Behaviour

314 Thus we shall consider our **road traffic system**, rts, as

a the concurrent behaviour of a number of vehicles and, to "observe", or, as we shall call it, to monitor their movements,

b the monitor behaviour.

```
value
314. trs() =
314a.    ‖ {veh_Δ(uid_VI(v),mi)(vm(uid_VI(v)))|v:V•v ∈ vs}
314b.    ‖ mon_Δ(mi,vis)([vi↦⟨⟩|vi:VI•vi ∈ vis])
```

342

where, wrt, the monitor, we dispense with the mereology and the attribute state arguments and instead just have a monitor traffic argument which records the discrete road traffic, MAP, initially set to "empty" traces (⟨⟩, of so far "no road traffic"!).

In order for the monitor behaviour to assess the vehicle positions these vehicles communicate their positions to the monitor via a
343 vehicle to monitor channel. In order for the monitor to time-stamp these positions it must be able to "read" a clock.

315 We describe here an abstraction of the vehicle behaviour **at** a Hub (hi).

a Either the vehicle remains at that hub informing the monitor of its position,

b or, internally non-deterministically,

i moves onto a link, tli, whose "next" hub, identified by thi, is obtained from the mereology of the link identified by tli;

ii informs the monitor, on channel vm[vi,mi], that it is now at the very beginning (0) of the link identified by tli,

iii whereupon the vehicle resumes the vehicle behaviour positioned at the very beginning of that link,

c or, again internally non-deterministically,

d the vehicle "disappears — off the radar" !

344

```
315.  veh_Δ(vi,mi)(vp:atH(hi,fli,tli)) ≡
315a.       v_m_ch[vi,mi]!vp ; veh_Δ(vi,mi)(vp)
315b.    ⌈⌉
315(b)i.    let {hi′,thi}=obs_mereo_L(get_link(tli)(n)) in
315(b)i.                  assert: hi′=hi
315(b)ii.   v_m_ch[vi,mi]!onL(tli,hi,thi,0) ;
315(b)iii.  veh_Δ(vi,mi)(onL(tli,hi,thi,0)) end
315c.    ⌈⌉
315d.       stop
```

345

316 We describe here an abstraction of the vehicle behaviour **on** a Link (ii). Either

a the vehicle remains at that link position informing the monitor of its position,

b or, internally non-deterministically,

c if the vehicle's position on the link has not yet reached the hub,

i then the vehicle moves an arbitrary increment ℓ_ε (less than or equal to the distance to the hub) along the link informing the monitor of this, or

ii else, 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),

A the vehicle informs the monitor that it is now at the hub identified by thi,
B whereupon the vehicle resumes the vehicle behaviour positioned at that hub.

317 or, internally non-deterministically,
318 the vehicle "disappears — off the radar" !

346

```
316.  veh_Δ(vi,mi)(vp:onL(li,fhi,thi,r)) ≡
316a.     v_m_ch[vi,mi]!vp ; veh(_Δvi,mi,va)(vp)
316b.   ⌈⌉
316c.     if r + ℓ_ε≤1
316(c)i.        then v_m_ch[vi,mi]!onL(li,fhi,thi,r+ℓ_ε) ;
316(c)i.             veh_Δ(vi,mi)(onL(li,fhi,thi,r+ℓ_ε))
316(c)ii.       else let li′:LI•li′ ∈ obs_mereo_H(get_hub(thi)(n)) in
316(c)iiA.           v_m_ch[vi,mi]!atH(li,thi,li′);
316(c)iiB.           veh_Δ(vi,mi)(atH(li,thi,li′)) end end
317.    ⌈⌉
318.      stop
```

347

The Monitor Behaviour

319 The monitor behaviour evolves around

a the monitor identifier,
b the monitor mereology,
c and the attributes, ma:ATTR
d — where we have factored out as a separate arguments — a table of traces of time-stamped vehicle positions,
e while accepting messages
  i about time
  ii and about vehicle positions
f and otherwise progressing "in[de]finitely".

348

320 Either the monitor "does own work"
321 or, internally non-deterministically accepts messages from vehicles.

a A vehicle position message, vp, may arrive from the vehicle identified by vi.
b That message is appended to that vehicle's movement trace – prefixed by time (obtained from the time channel),
c whereupon the monitor resumes its behaviour —
d where the communicating vehicles range over all identified vehicles.

349

```
319.  mon_Δ(mi,vis)(trf) ≡
320.     mon_Δ(mi,vis)(trf)
321.   ⌈⌉
321a.    []{let tvp = (clk_ch?,v_m_ch[vi,mi]?) in
321b.      let trf′ = trf † [vi ↦ trf(vi)^<tvp>] in
321c.      mon_Δ(mi,vis)(trf′)
321d.      end end | vi:VI • vi ∈ vis}
```

350

We are about to complete a long, i.e., a four page example. We can now comment on the full example: The domain, $\delta : \Delta$ is a manifest part. The road net, $n : N$ is also a manifest part. The fleet, $f : F$, of vehicles, $vs : VS$, likewise, is a manifest part. But the monitor, $m : M$, is a concept. One does not have to think of it as a manifest "observer". The vehicles are on — or off — the road (i.e., links and hubs). We know that from a few observations and generalise to all vehicles. They either move or stand still. We also, similarly, know that. Vehicles move. Yes, we know that. Based on all these repeated observations and generalisations we introduce the concept of vehicle traffic. Unless positioned high above a road net — and with good binoculars — a single person cannot really observe the traffic. There are simply too many links, hubs, vehicles, vehicle positions and times. Thus we conclude that, even in a richly manifest domain, we can also "speak of", that is, describe concepts over manifest phenomena, including time !

351

Part II

Requirements Engineering

Chapter 7

Requirements 352

Chapter Status

Sections 7.3 and 7.4 need be revised.
Chapter lacks a concluding section.

Abstract

In Chapter 1 we introduced a method for analysing and describing manifest domains. In this chapter we show how to systematically, but of course, not automatically, "derive" requirements prescriptions from domain descriptions. There are, as we see it, three kinds of requirements: (i) domain requirements, (ii) interface requirements and (iii) machine requirements. The **machine** is the hardware and software to be developed from the requirements ■ (i) **Domain requirements** are those requirements which can be expressed sôlely using technical terms of the 353
domain ■ (ii) **Interface requirements** are those requirements which can be expressed only using technical terms of both the domain and the machine ■ (iii) **Machine requirements** are those requirements which can be expressed sôlely using technical terms of the machine ■ We show principles, techniques and tools for "deriving" domain 354
requirements and interface requirements. The domain requirements development focus on projection, instantiation, determination, extension and fitting. These domain-to-requirements operators can be described briefly: projection 355
removes such descriptions which are to be omitted for consideration in the requirements, instantiation mandates specific mereologies, determination specifies less non–determinism, extension extends the evolving requirements prescription with further domain description aspects and fitting resolves "loose ends" as they may have emerged during the domain-to-requirements operations.

7.1 Introduction 356

Definition 28 . ***Requirements (I):*** *By a* ***requirements*** *we understand (cf. IEEE Standard 610.12 [98]): "A condition or capability needed by a user to solve a problem or achieve an objective"* ■

7.1.1 General Considerations

The objective of requirements engineering is to create a requirements prescription: A **requirements prescription** specifies externally observable properties of endurants and perdurants: functions, events and behaviours of ***the machine*** such as the requirements stake-holders wish them to be ■ The **machine** is what is required: that is, the ***hardware*** and ***software*** that is to be designed and which are to satisfy the requirements ■ A *requirements prescription* thus (***putatively***) expresses what there should be. A requirements prescription expresses nothing about the design of the possibly desired (required) software. We shall show how a major part of a requirements prescription can be "derived" from "its" prerequisite domain description. 357 358

Rule 1 ***The "Golden Rule" of Requirements Engineering:*** *Prescribe only those requirements that can be objectively shown to hold for the designed software* ■

"Objectively shown" means that the designed software can either be tested, or be model checked, or be proved (verified), to satisfy the requirements.

Rule 2 ***An "Ideal Rule" of Requirements Engineering:*** *When prescribing (including formalising) requirements, also formulate tests and properties for model checking and theorems whose actualisation should show adherence to the requirements* ■

The rule is labelled "ideal" since such precautions will not be shown in this paper. The rule is clear. It is a question for proper management to see that it is adhered to.

Rule 3 ***Requirements Adequacy:*** *Make sure that requirements cover what users expect* ■

That is, do not express a requirement for which you have no users, but make sure that all users' requirements are represented or somehow accommodated. In other words: the requirements gathering process needs to be like an extremely "fine-meshed net": One must make sure that all possible stake-holders have been involved in the requirements acquisition process, and that possible conflicts and other inconsistencies have been obviated.

Rule 4 ***Requirements Implementability:*** *Make sure that requirements are implementable* ■

That is, do not express a requirement for which you have no assurance that it can be implemented. In other words, although the requirements phase is not a design phase, one must tacitly assume, perhaps even indicate, somehow, that an implementation is possible. But the requirements in and by themselves, stay short of expressing such designs.

Rule 5 ***Requirements Verifiability and Validability:*** *Make sure that requirements are verifiable and can be validated* ■

That is, do not express a requirement for which you have no assurance that it can be verified and validated. In other words, once a first-level software design has been proposed, one must show that it satisfies the requirements. Thus specific parts of even abstract software designs are usually provided with references to specific parts of the requirements that they are (thus) claimed to implement.

Definition 29 . ***Requirements (II):*** *By* ***requirements*** *we shall understand a document which prescribes desired properties of a machine: (i) what endurants the machine shall "maintain", and what the machine shall (must; not should) offer of (ii) functions and of (iii) behaviours (iv) while also expressing which events the machine shall "handle"* ■

By a machine that "maintains" endurants we shall mean: a machine which, "between" users' use of that machine, "keeps" the data that represents these entities. From earlier we repeat:

Definition 30 . ***Machine:*** *By machine we shall understand a, or the, combination of hardware and software that is the target for, or result of the required computing systems development* ■

So this, then, is a main objective of requirements development: to start towards the design of the hardware + software for the computing system.

Definition 31 . ***Requirements (III):*** *To specify the machine* ■

When we express requirements and wish to "convert" such requirements to a realisation, i.e., an implementation, then we find that some requirements (parts) imply certain properties to hold of the hardware on which the software to be developed is to "run", and, obviously, that remaining — probably the larger parts of the — requirements imply certain properties to hold of that software. So we find that although we may believe that our job is software engineering, important parts of our job are to also "design the machine"!

7.1.2 Four Stages of Requirements Development 366

We shall unravel requirements in four stages — the first three stages are sketchy (and thus informal) while the last stage is systematic, mandates both strict narrative, and formal descriptions, and is "derivable" from the domain description. The four stages are: (i) the *problem/objective* sketch, (ii) the narrative system requirements sketch, (iii) the narrative user requirements sketch, and (iv) the systematic narrative and formal functional requirements prescription.

Problem and/or Objective Sketch 367

Definition 32 . ***Problem/Objective Sketch:*** *By a* ***problem/objective sketch*** *we understand a narrative which emphasises what the problem or objectie is and thereby names its main concepts* ■

368

Example 82 . **The Problem/Objective Requirements: A Sketch**: The objective is to create a **road-pricing product.** By a road-pricing product we shall understand an information technology-based system containing computers and communications equipment and software that enables the recording of *vehicle* movements within a well-delineated *road net* and thus enables the *owner* of the road net to charge the *owner* of the vehciles *fees* for the usage of that road net

Systems Requirements 369

Definition 33 . ***System Requirements:*** *By a* ***system requirements narrative*** *we understand a narrative which emphasises the overall hardware and software system components* ■

370

Example 83 . **The Road-pricing System Requirements: A Narrative**: The requirements are based on the following a-priori given constellation of system components: (i) there is assumed a GNSS: a `Global Navigation Satellite System`; (ii) there are specially equipped *vehicles*; (iii) there is a well-delineated road net called a *toll-road* net with specially equipped toll-gates with *barrier*s which afford (only the specially equipped) vehicles to enter into and exit from the toll-road net; and (iv) there is a [road-pricing] *calculator*. These four system components are required to 371
behave and interact as follows: (a) The `GNSS` is assumed to continuously offer vehicles timed information about their global positions; (b) *vehicles* shall contain a `GNSS receiver` which based on the global position information shall regularly calculate their timed local position and offer this to the *calculator* — while otherwise cruising the general road net as well as the toll-road net, the latter while carefully moving through toll-gate barriers; (c) *toll-road gates* shall register the identity of vehicles entering and exiting the toll-road and offer this information to the calculator; and (d) the *calculator* shall accept all messages from vehicles and gates and use this information to record the movements of vehicles and bill these whenever they exit the toll-road. The requirements are therefore to include requirements to [1] the `GNSS` radio telecom- 372
munications equipment, [2] the vehicle `GNSS receiver` equipment, [3] the vehicle software, [4] the toll-gate in and out sensor equipment, [5] the electro-mechanical toll-gate barrier equipment, [6] the toll-gate barrier actuator equipment, [7] the toll-gate software, [8] the actuator software, and [9] the communications It is in this sense that the requirements are for 373
an information technology-based system of both software and hardware — not just hard computer and communications equipment, but also movement sensors and electro-mechanical "gear"

User and External Equipment Requirements 374

Definition 34 . ***User and External Equipment Requirements:*** *By a* ***user and external equipment requirements narrative*** *we understand a narrative which emphasises the human user and external equipment interfaces to the system components* ■

375

Example 84 . **The Road-pricing User and External Equipment Requirements: Narrative**:
The human users of the road-pricing system are vehicle drivers, toll-gate sensor, actuator and barrier service staff,

and the road-pricing service calculator staff. The external equipment are the GNSS satellites and the telecommunications equipment (i) which enables communication between (ii) the GNSS satellite sand vehicles , (iii) vehicles and the road-pricing calculator, (iv) toll-gates and the road-pricing calculator and (v) the road-pricing calculator and vehicles (for billing), We defer expression of human user and external equipment requirements till our treatment of relevant functional requirements

Functional Requirements

Definition 35 . ***Functional Requirements:*** *By* ***functional requirements*** *we understand precise prescriptions of the endurants and perdurants of the system components* ■

There are, as we see it, three kinds of requirements: (i) domain requirements, (ii) interface requirements and (iii) machine requirements (i) **Domain requirements** are those requirements which can be expressed sôlely using technical terms of the domain ■ (ii) **Interface requirements** are those requirements which can be expressed only using technical terms of both the domain and the machine ■ (iii) **Machine requirements** are those requirements which can be expressed sôlely using technical terms of the machine ■

7.2 Domain Requirements

Definition 36 . ***Domain Requirements Prescription:*** *A* ***domain requirements prescription*** *is that subset of the requirements prescription which can be expressed sôlely using terms from the domain description* ■

To determine a relevant subset all we need is collaboration with requirements stake-holders. Experimental evidence, in the form of example developments of requirements prescriptions from domain descriptions, appears to show that one can formulate techniques for such developments around a few domain description to requirements prescription operations. We suggest these: projection, instantiation, determination, extension, fitting and, perhaps, other domain description to requirements prescription operations.

7.2.1 Domain Projection

Definition 37 . ***Domain Projection:*** *By a* ***domain projection*** *we mean a subset of the domain description, one which leaves out all those endurants: parts, materials and components, as well as perdurants: functions, events and behaviours that the stake-holders do not wish represented by the machine. The resulting document is a partial domain requirements prescription* ■

In determining an appropriate subset the requirements engineer must secure that the final prescription is complete and consistent — that is, that there are no "dangling references", i.e., that all entities that are referred to are all properly defined.

Domain Projection — Narrative

We now start on a series of examples that illustrate domain requirements development.

Example 85 . **Domain Requirements. Projection A Narrative Sketch**: We require that the Road-pricing IT, computing & communications system shall embody the following domain entities, in one form or another: the net, its links and hubs, and their properties (unique identifiers, mereologies and attributes), the vehicles, as endurants, as endurants, and the general vehicle behaviour, i.e., the vehicle signature. To formalise this we copy the domain description, Δ_Δ, From that domain description we remove all mention of the link insertion and removal functions, the link disappearance event, the vehicle behaviour, and the monitor to obtain the $\Delta_{\not\Delta}$ version of the domain requirements prescription.[1]

[1] Restrictions of the net to the toll road nets, hinted at earlier, will follow in the next domain requirements steps.

Domain Projection — Formalisation

385

The requirements prescription hinges, crucially, not only on a systematic narrative of all the projected, instantiated, determinated, extended and fitted specifications, but also on their formalisation. In the series of domain projection examples following below we, regretfully, omit the narrative texts. In bringing the formal texts we keep the item numbering from Sect. 1.2, where you can find the associated narrative texts. 386

Example 86 . Domain Requirements. Projection Root Sorts:

```
type
270.  Δ𝒟
270a.  N𝒟
270b.  F𝒟
value
270a.  obs_part_N𝒟: Δ𝒟→N𝒟
270b.  obs_part_F𝒟: Δ𝒟→F𝒟
```

```
type
271a.  HA𝒟
271b.  LA𝒟
value
271a.  obs_part_HA: N𝒟 → HA
271b.  obs_part_LA: N𝒟 → LA
```

387

Example 87 . Domain Requirements. Projection Sub-domain Sorts and Types:

```
type
272.  H𝒟, HS𝒟 = H𝒟-set
273.  L𝒟, LS𝒟 = L𝒟-set
274.  V𝒟, VS𝒟 = V𝒟-set
value
272.  obs_part_HS𝒟: HA𝒟 → HS𝒟
```

```
273.  obs_part_LS𝒟: LA𝒟 → LS𝒟
274.  obs_part_VS𝒟: F𝒟 → VS𝒟
275a.  links: Δ𝒟 → L-set
275a.  links(δ𝒟) ≡ obs_part_LS𝒮(obs_part_LA𝒮(δ𝒮))
275b.  hubs: Δ𝒟 → H-set
275b.  hubs(δ𝒟) ≡ obs_part_HS𝒟(obs_part_HA𝒟(δ𝒟))
```

388

Example 88 . Domain Requirements. Projection Unique Identifications:

```
type
276a.  HI, LI, VI, MI
value
276c.  uid_HI: H𝒟 → HI
276c.  uid_LI: L𝒟 → LI
```

```
276c.  uid_VI: V𝒟 → VI
276c.  uid_MI: M𝒟 → MI
axiom
276b.  HI∩LI=Ø, HI∩VI=Ø, HI∩MI=Ø,
276b.  LI∩VI=Ø, LI∩MI=Ø, VI∩MI=Ø
```

389

Example 89 . Domain Requirements. Projection Road Net Mereology:

```
value
278.  obs_mereo_H𝒟: H𝒟 → LI-set
279.  obs_mereo_L𝒟: L𝒟 → HI-set
279.  axiom ∀ l:L𝒟 • card obs_mereo_L𝒟(l)=2
280.  obs_mereo_V𝒟: V𝒟 → MI
281.  obs_mereo_M𝒟: M𝒟 → VI-set
axiom
282.  ∀ δ𝒟:Δ𝒟, hs:HS•hs=hubs(δ), ls:LS•ls=links(δ𝒟) ⇒
282.  ∀ h:H𝒟•h ∈ hs ⇒
282.  obs_mereo_H𝒟(h)⊆xtr_his(δ𝒟) ∧
```

```
283.  ∀ l:L𝒟•l ∈ ls •
282.  obs_mereo_L𝒟(l)⊆xtr_lis(δ𝒟) ∧
284a.  let f:F𝒟•f=obs_part_F𝒟(δ𝒟) ⇒
284a.  vs:VS𝒟•vs=obs_part_VS𝒟(f) in
284a.  ∀ v:V𝒟•v ∈ vs ⇒
284a.  uid_V𝒟(v) ∈ obs_mereo_M𝒟(m) ∧
284b.  obs_mereo_M𝒟(m)
284b.  = {uid_V𝒟(v)|v:V•v ∈ vs}
284b.  end
```

390

Example 90 . Domain Requirements. Projection Attributes of Hubs:

```
type
285a. HΣ𝒟 = (HI×HI)-sett
285b. HΩ𝒟 = HΣ𝒟-set
value
285a. attr_HΣ𝒟: H𝒟 → HΣ𝒟
285b. attr_HΩ𝒟: H𝒟 → HΩ𝒟
type
287.  HGCL
value
287.  attr_HGCL: H → HGCL
axiom
286.  ∀ δ𝒟:Δ𝒟,
286.    let hs = hubs(δ𝒟) in
286.    ∀ h:H𝒟 • h ∈ hs •
286a.       xtr_lis(h)⊆xtr_lis(δ𝒟)
286b.     ∧ attr_Σ𝒟(h) ∈ attr_Ω𝒟(h)
286.    end
```

391

Example 91 . Domain Requirements. Projection Attributes of Links:

```
type
289.  LEN
290.  LGCL
291a. LΣ𝒟 = (HI×HI)-set
291b. LΩ𝒟 = LΣ𝒟-set
value
289.  attr_LEN: L𝒟 → LEN
290.  attr_LGCL: L 𝒟→ LGCL
291a. attr_LΣ𝒟: L𝒟 → LΣ𝒟
291b. attr_LΩ𝒟: L𝒟 → LΩ𝒟
axiom
      291a.– 291b on Page 144.
```

392

Example 92 . Domain Requirements. Projection Behaviour:

Global Values

```
value
304.  δ𝒟:Δ𝒟,
305.  n:N𝒟 = obs_part_N𝒟(δ𝒟),
305.  ls:L𝒟-set = links(δ𝒟),
305.  hs:H𝒟-set = hubs(δ𝒟),
305.  lis:LI-set = xtr_lis(δ𝒟),
305.  his:HI-set = xtr_his(δ𝒟)
```

Behaviour Signatures

```
value
311.  trs𝒟: Unit → Unit
312.  veh𝒟: VI×MI×ATTR → ... Unit
```

The System Behaviour

```
value
314a. trs𝒟()=‖{veh𝒟(uid_VI(v),obs_mereo_V(v),attr_ATTRS(v)) | v:V𝒟•v ∈ vs}
```

We observe that the vehicle behaviour is left unspecified.

A Projection Operator 393

Domain projection thus take a domain description, $\mathscr{D}$, and yields a projected requirements prescription, ,$\mathscr{R}_{\mathscr{D}}$.

- **type** projection: $\mathscr{D} \rightarrow \mathscr{R}_{\mathscr{D}}$.

Semantically $\mathscr{D}$ denotes a possibly infinite set of meanings, say $\mathbb{D}$ and $\mathscr{R}_{\mathscr{D}}$ denotes a possibly infinite set of meanings, say $\mathbb{R}_{\mathbb{P}}$, such that some relation $\mathbb{R}_{\mathbb{P}} \sqsubseteq \mathbb{D}$ is satisfied.

7.2.2 Domain Instantiation 394

Definition 38 . ***Instantiation:*** *By* ***domain instantiation*** *we mean a refinement of the partial domain requirements prescription, resulting from the projection step, in which the refinements aim at rendering the endurants: parts, materials and components, as well as the perdurants: actions, events and behaviours of the domain requirements prescription more concrete, more specific* ■ *Instantiations usually render these concepts less general.*

395

Refinement of endurants can be expressed (i) either in the form of concrete types, (ii) or of further "delineating" axioms over sorts, (iii) or of a combination of concretisation and axioms. We shall exemplify the third possibility. Examples 93–94 express requirements that the road net on which the road-pricing system is to be based must satisfy.

Domain Instantiation — Narrative

396

Example 93 . Domain Requirements. Instantiation Road Net, Narrative: We now require that there is, as before, a road net, $n_{\mathscr{J}}$:$N_{\mathscr{J}}$, which can be understood as consisting of two, "connected sub-nets". A toll-road net, $trn_{\mathscr{J}}$:$TRN_{\mathscr{J}}$, cf. Fig. 7.1, and an ordinary road net, n'_{Δ}. The two are connected as follows: The toll-road net, $trn_{\mathscr{J}}$, borders some toll-road plazas, in Fig. 7.1 shown by white filled circles (i.e., hubs). These toll-road plaza hubs are proper hubs of the 'ordinary' road net, n'_{Δ}.

397

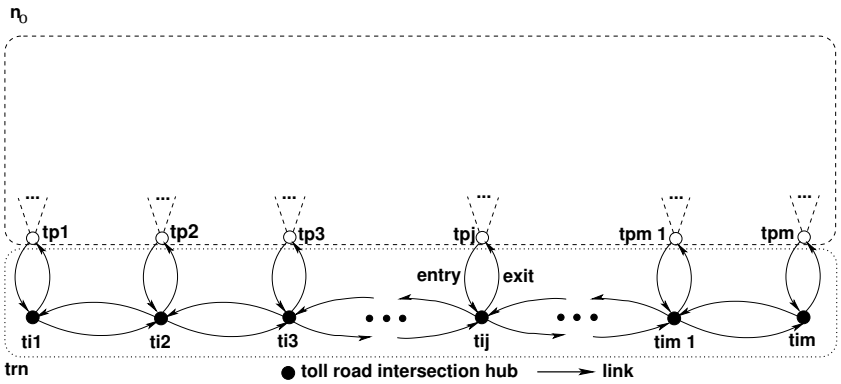

Figure 7.1: A simple, linear toll-road net trn. tp_j: $t$oll $p$laza j, ti_j: $t$oll road $i$ntersection j.
Upper dashed sub-figure hint at an ordinary road net n_o.
Lower dotted sub-figure hint at a toll-road net trn.
Dash-dotted (- - - - - -) images above tp_js hint at links to remaining "parts" of n_o.

398

322 The instantiated domain, $\delta_{\mathscr{J}}$:$\Delta_{\mathscr{J}}$ has just the net, $n_{\mathscr{J}}$:$N_{\mathscr{J}}$ being instantiated.

323 The road net consists of two "sub-nets"

- a an "ordinary" road net, n'_{Δ}:N'_{Δ} and
- b a toll-road net proper, $trn_{\mathscr{J}}$:$TRN_{\mathscr{J}}$ —
- c "connected" by an interface hil:HIL:
  - i That interface consists of a number of toll-road plazas (i.e., hubs), modeled as a list of hub identifiers, hil:HI*.
  - ii The toll-road plaza interface to the toll-road net, trn:$TRN_{\mathscr{J}}$[2], has each plaza, hil[i], connected to a pair of toll-road links: an entry and an exit link: $(l_e{:}L, l_x{:}L)$.
  - iii The toll-road plaza interface to the 'ordinary' net, n'_{Δ}:N'_{Δ}, has each plaza, i.e., the hub designated by the hub identifier hil[i], connected to one or more ordinary net links, $\{l_{i_1}, l_{i_2}, \cdots, l_{i_\ell}\}$.

323b The toll-road net, trn:$TRN_{\mathscr{J}}$, consists of three collections (modeled as lists) of links and hubs:

- i a list of pairs of toll-road entry/exit links: $\langle(l_{e_1}, l_{x_1}), \cdots, (l_{e_\ell}, l_{x_\ell})\rangle$,
- ii a list of toll-road intersection hubs: $\langle h_{i_1}, h_{i_2}, \cdots, h_{i_\ell}\rangle$, and
- iii a list of pairs of main toll-road ("up" and "$d$own") links: $\langle(ml_{i_{1u}}, ml_{i_{1d}}), (m_{i_{2u}}, m_{i_{2d}}), \cdots, (m_{i_{\ell u}}, m_{i_{\ell d}})\rangle$.

d The three lists have commensurate lengths.

ℓ is the number of toll plazas, hence also the number of toll-road intersection hubs and therefore a number one larger than the number of pairs of main toll-road ("up" and "$d$own") links

Domain Instantiation — Formalisation

401

Example 94 . Domain Requirements. Instantiation Road Net, Formal Types:

[2] We (sometimes) omit the subscript $\mathscr{J}$ when it should be clear from the context what we mean.

type
322 $\Delta_{\mathscr{I}}$
323 $N_{\mathscr{I}} = N'_{\Delta} \times HIL \times TRN$
323a N'_{Δ}
323b $TRN_{\mathscr{I}} = (L \times L)^* \times H^* \times (L \times L)^*$
323c $HIL = HI^*$

axiom
323d $\forall\, n_{\mathscr{I}}{:}N_{\mathscr{I}}$ •
323d **let** $(n_{\Delta},hil,(exll,hl,lll)) = n_{\mathscr{I}}$ **in**
323d **len** hil = **len** exll = **len** hl = **len** lll + 1
323d **end**

We have named the "ordinary" net sort N'_{Δ}. It is "almost" like N_{Δ} — except that the interface hubs are also connected to the toll-road net entry and exit links.

Domain Instantiation — Formalisation: Well-formedness 402

Example 95 . Domain Requirements. Instantiation Road Net, Well-formedness: The partial concretisation of the net sorts, N, into $N_{\mathscr{R}_1}$ requires, in addition to the length relations of the three lists of interface hubs, entry and exit links and , some well-formedness conditions to be satisfied.

324 The toll-road intersection hubs must all have distinct hub identifiers.

value
324. wf_dist_toll_road_isect_hub_ids: $H^* \to$ **Bool**
324. wf_dist_toll_road_isect_hub_ids(hl) ≡
324. **len** hl = **card** xtr_his(hl)

325 The toll-road 'up' and 'down' links must all have distinct link identifiers.

value
325. wf_dist_toll_road_u_d_link_ids: $(L \times L)^* \to$ **Bool**
325. wf_dist_toll_road_u_d_link_ids(lll) ≡
325. 2 × **len** lll = **card** xtr_lis(lll)

403

326 The toll-road entry/exit links must all have distinct link identifiers.

value
326. wf_dist_e_x_link_ids: $(L \times L)^* \to$ **Bool**
326. wf_dist_e_x_link_ids(exll) ≡
326. 2 × **len** exll = **card** xtr_lis(exll)

327 Proper net links must not designate toll-road intersection hubs.

value
327. wf_isoltd_toll_road_isect_hubs: $HI^* \times H^* \to N_{\mathscr{I}} \to$ **Bool**
327. wf_isoltd_toll_road_isect_hubs(hil,hl)($n_{\mathscr{I}}$) ≡
327. **let** ls=xtr_links($n_{\mathscr{I}}$) **in**
327. **let** his = ∪ {**obs_mereo_**L(l)|l:L•l ∈ ls} **in**
327. his ∩ xtr_his(hl) = {} **end end**

404

328 The plaza hub identifiers must designate hubs of the 'ordinary' net.

value
328. wf_p_hubs_pt_of_ord_net: $HI^* \to N'_{\Delta} \to$ **Bool**
328. wf_p_hubs_pt_of_ord_net(hil)(n'_{Δ}) ≡
328. **elems** hil ⊆ xtr_his(n'_{Δ})

329 The plaza hub mereologies must each,
- a besides identifying at least one hub of the ordinary net,
- b also identify the two entry/exit links with which they are supposed to be connected.

value
329. wf_p_hub_interf: $N'_{\Delta} \to$ **Bool**
329. wf_p_hub_interf(n_o,hil,(exll,_,_)) ≡
329. ∀ i:**Nat** • i ∈ **inds** exll ⇒
329. **let** h = get_H(hil(i))(n'_{Δ}) **in**
329. **let** lis = **obs_mereo_**H(h) **in**
329. **let** lis′ = lis \ xtr_lis(n') **in**
329. lis′ = xtr_lis(exll(i)) **end end end**

405

330 The mereology of each toll-road intersection hub must identify
- a the entry/exit links
- b and exactly the toll-road 'up' and 'down' links
- c with which they are supposed to be connected.

value
330. wf_toll_road_isect_hub_iface: $N_{\mathscr{I}} \to$ **Bool**
330. wf_toll_road_isect_hub_iface(_,_,(exll,hl,lll)) ≡
330. ∀ i:**Nat** • i ∈ **inds** hl ⇒
330. **obs_mereo_**H(hl(i)) =
330a. xtr_lis(exll(i)) ∪
330. **case** i **of**
330b. 1 → xtr_lis(lll(1)),
330b. **len** hl → xtr_lis(lll(**len** hl−1))
330b. _ → xtr_lis(lll(i)) ∪ xtr_lis(lll(i−1))
330. **end**

406

331 The mereology of the entry/exit links must identify exactly the
  a interface hubs and the
  b toll-road intersection hubs
  c with which they are supposed to be connected.

```
value
331. wf_exll: (L×L)*×Hl*×H*→Bool
331. wf_exll(exll,hil,hl) ≡
331.   ∀ i:Nat • i ∈ inds exll
331.     let (hi,(el,xl),h) = (hil(i),exll(i),hl(i)) in
331.     obs_mereo_L(el) = obs_mereo_L(xl)
331.     = {hi} ∪ {uid_H(h)} end
331.   pre: len eell = len hil = len hl
```

407

332 The mereology of the toll-road 'up' and 'down' links must
  a identify exactly the toll-road intersection hubs
  b with which they are supposed to be connected.

```
value
332. wf_u_d_links: (L×L)*×H*→Bool
332. wf_u_d_links(lll,hl) ≡
332.   ∀ i:Nat • i ∈ inds lll ⇒
332.     let (ul,dl) = lll(i) in
332.     obs_mereo_L(ul) = obs_mereo_L(dl) =
332a.    uid_H(hl(i)) ∪ uid_H(hl(i+1)) end
332.   pre: len lll = len hl+1
```

408

We have used additional auxiliary functions:

```
value
  xtr_his: H*→Hl-set
  xtr_his(hl) ≡ {uid_Hl(h)|h:H•h ∈ elems hl}

  xtr_lis: (L×L)→Ll-set
  xtr_lis(l′,l″) ≡ {uid_Ll(l′)}∪{uid_Ll(l″)}
  xtr_lis: (L×L)*→ Ll-set
  xtr_lis(lll) ≡
  ∪{xtr_lis(l′,l″)|(l′,l″):(L×L)•(l′,l″)∈ elems lll}
```

409

Summary Well-formedness Predicate

333 The well-formedness of instantiated nets is now the conjunction of the individual well-formedness predicates above.

```
value
333. wf_instantiated_net: N𝒻 → Bool
333. wf_instantiated_net(n′Δ,hil,(exll,hl,lll))
324.   wf_dist_toll_road_isect_hub_ids(hl)
325.   ∧ wf_dist_toll_road_u_d_link_ids(lll)
326.   ∧ wf_dist_e_e_link_ids(exll)
327.   ∧ wf_isolated_toll_road_isect_hubs(hil,hl)(n′)
328.   ∧ wf_p_hubs_pt_of_ord_net(hil)(n′)
329.   ∧ wf_p_hub_interf(n′Δ,hil,(exll,_,_))
330.   ∧ wf_toll_road_isect_hub_iface(_,_,(exll,hl,lll))
331.   ∧ wf_exll(exll,hil,hl)
332.   ∧ wf_u_d_links(lll,hl)
```

Domain Instantiation — Abstraction

410

Example 96 . Domain Requirements. Instantiation Road Net, Abstraction: Domain instantiation has refined an abstract definition of net sorts, n_Δ:N_Δ, into a partially concrete definition of nets, $n_{\mathscr{f}}$:$N_{\mathscr{f}}$. We need to show the refinement relation:

- abstraction($n_{\mathscr{f}}$) = n_Δ.

411

```
value
334  abstraction: N𝒻 → NΔ
335  abstraction(n′Δ,hil,(exll,hl,lll)) ≡
336    let nΔ:NΔ •
336      let hs = obs_part_HSΔ(obs_part_HAΔ(n′Δ)),
336          ls = obs_part_LSΔ(obs_part_LAΔ(n′Δ)),
336          ths = elems hl,
336          eells = xtr_links(eell), llls = xtr_links(lll) in
337          hs∪ths=obs_part_HSΔ(obs_part_HAΔ(nΔ))
338        ∧ ls∪eells∪llls=obs_part_LSΔ(obs_part_LAΔ(nΔ))
339    nΔ end end
```

334 The abstraction function takes a concrete net, $n_{\mathscr{f}}$:$N_{\mathscr{f}}$, and yields an abstract net, n_Δ:N_Δ.

335 The abstraction function doubly decomposes its argument into constituent lists and sub-lists.

336 There is postulated an abstract net, n_Δ:N_Δ, such that

337 the hubs of the concrete net and toll-road equals those of the abstract net, and

338 the links of the concrete net and toll-road equals those of the abstract net.

339 And that abstract net, n_Δ:N_Δ, is postulated to be an abstraction of the concrete net.

An Instantiation Operator

413

Domain instantiation take a requirements prescription, $\mathscr{R}_{\mathscr{P}}$, and yields a more concrete requirements prescription $\mathscr{R}_{\mathscr{I}}$.

- **type** instantiation: $\mathscr{R}_{\mathscr{P}} \rightarrow \mathscr{R}_{\mathscr{I}}$

Semantically $\mathscr{R}_{\mathscr{P}}$ denotes a possibly infinite set of meanings, say $\mathbb{R}_{\mathbb{P}}$, $\mathscr{R}_{\mathscr{I}}$ denotes a possibly infinite set of meanings, say $\mathbb{R}_{\mathbb{I}}$ and such that some relation $\mathbb{R}_{\mathbb{I}} \sqsubseteq \mathbb{R}_{\mathbb{P}}$ is satisfied.

7.2.3 Domain Determination

414

Definition 39 . ***Determination:*** *By* ***domain determination*** *we mean a refinement of the partial domain requirements prescription, resulting from the instantiation step, in which the refinements aim at rendering the endurants: parts, materials and components, as well as the perdurants: functions, events and behaviours of the partial domain requirements prescription less non-determinate, more determinate.*

■

415

Determinations usually render these concepts less general. That is, the value space of endurants that are made more determinate is "smaller", contains fewer values, as compared to the endurants before determination has been "applied".

Domain Determination: Example

We show an example of 'domain determination'. It is expressed sôlely in terms of axioms over the concrete
416 toll-road net type.

Example 97 . **Domain Requirements. Determination Toll-roads**:

All Toll-road Links are One-way Links We focus only on the toll-road net. We single out only two 'determinations':

340 *The entry/exit and toll-road links*

- a are always all one way links,
- b as indicated by the arrows of Fig. 7.1 on Page 157,
- c such that each pair allows traffic in opposite directions.

```
value
340.  opposite_traffics: (L×L)* × (L×L)* → Bool
340.  opposite_traffics(exll,lll) ≡
340.      ∀ (lt,lf):(L×L) • (lt,lf) ∈ elems exll^lll ⇒
340a.       let (ltσ,lfσ) = (attr_LΣ(lt),attr_LΣ(lf)) in
340a′.      attr_LΩ(lt)={ltσ}∧attr_LΩ(ft)={ftσ}
340a″.    ∧ card ltσ = 1 = card lfσ
340.      ∧ let ({(hi,hi′)},{(hi″,hi‴)}) = (ltσ,lfσ) in
340c.       hi=hi‴ ∧ hi′=hi″
340.      end end
```

Predicates 340a′. and 340a″. express the same property.

418

All Toll-road Hubs are Free-flow

341 *The hub state spaces* are singleton sets of the toll-road hub states which always allow exactly these (and only these) crossings:

- a from *e*ntry links back to the paired e*x*it *l*ink*s*,
- b from *e*ntry links to emanating *t*oll-road *l*ink*s*,
- c from incident *t*oll-road links to e*x*it *l*ink*s*, and
- d from incident *t*oll-road link to emanating *t*oll-road *l*ink*s*.

```
value
341.  free_flow_toll_road_hubs: (L×L)*×(L×L)*→Bool
341.  free_flow_toll_road_hubs(exl,ll) ≡
341.      ∀ i:Nat•i ∈ inds hl ⇒
341.        attr_HΣ(hl(i)) =
341a.         hσ_ex_ls(exl(i))
341b.       ∪ hσ_et_ls(exl(i),(i,ll))
341c.       ∪ hσ_tx_ls(exl(i),(i,ll))
341d.       ∪ hσ_tt_ls(i,ll)

value
341a.  hσ_ex_ls: (L×L)→LΣ
341a.  hσ_ex_ls(e,x) ≡ {(uid_LI(e),uid_LI(x))}

value
341b.  hσ_et_ls: (L×L)×(Nat×(em:L×in:L)*)→LΣ
```

```
341b. hσ_et_ls((e,_),(i,ll)) ≡
341b.   case i of
341b.     2      → {(uid_LI(e),uid_LI(em(ll(1))))},
341b.     len ll+1 → {(uid_LI(e),uid_LI(em(ll(len ll))))},
341b.     _      → {(uid_LI(e),uid_LI(em(ll(i−1)))),
341b.                 (uid_LI(e),uid_LI(em(ll(i))))}
341b.   end
```

The *em* and *in* in the toll-road link list (em:L×in:L)* designate selectors for *em*anating, respectively *in*cident links.

```
value
341c. hσ_tx_ls: (L×L)×(Nat×(em:L×in:L)*)→LΣ
341c. hσ_tx_ls((_,x),(i,ll)) ≡
341c.   case i of
341c.     2      → {(uid_LI(in(ll(1))),uid_LI(x))},
341c.     len ll+1 → {(uid_LI(in(ll(len ll))),uid_LI(x))},
341c.     _      → {(uid_LI(in(ll(i−1))),uid_LI(x)),
341c.                 (uid_LI(in(ll(i))),uid_LI(x))}
341c.   end
```

```
value
341d. hσ_tt_ls: Nat×(em:L×in:L)*→LΣ
341d. hσ_tt_ls(i,ll) ≡
341d.   case i of
341d.     2      → {(uid_LI(in(ll(1))),uid_LI(em(ll(1))))},
341d.     len ll+1 → {(uid_LI(in(ll(len ll))),uid_LI(em(ll(len ll))))},
341d.     _      → {(uid_LI(in(ll(i−1))),uid_LI(em(ll(i−1)))),
341d.                 (uid_LI(in(ll(i))),uid_LI(em(ll(i))))}
341d.   end
```

A Domain Determination Operator 423

Domain determination take a requirements description, $\mathscr{R}_{\mathscr{I}}$, and yields a more deterministic requirements prescription, $\mathscr{R}_{\mathscr{D}}$.

- **type** instantiation: $\mathscr{R}_{\mathscr{I}} \rightarrow \mathscr{R}_{\mathscr{D}}$

Semantically $\mathscr{R}_{\mathscr{I}}$ denotes a possibly infinite set of meanings, say $\mathbb{R}_{\mathbb{I}}$, $\mathscr{R}_{\mathscr{D}}$ denotes a possibly infinite set of meanings, say $\mathbb{R}_{\mathbb{D}}$ and such that some relation $\mathbb{R}_{\mathbb{I}} \sqsubseteq \mathbb{R}_{\mathbb{D}}$ is satisfied.

7.2.4 Domain Extension 424

Definition 40 . ***Extension:*** *By* ***domain extension*** *we understand the introduction of endurants and perdurants that were not feasible in the original domain, but for which, with computing and communication, and with new, emerging technologies, for example, sensors, actuators and satellites, there is the possibility of feasible implementations, hence requirement, that what is introduced becomes*[3] *part of the unfolding requirements prescription* ■

The Core Requirements Example: Domain Extension 425

Example 98 . **Domain Requirements. Extension Vehicles: Parts, Properties and Channels**:

342 There is a domain, $\delta_{\mathscr{E}}$:$\Delta_{\mathscr{E}}$, which contains

343 a fleet, $f_{\mathscr{E}}$:$F_{\mathscr{E}}$,

344 of a set, $vs_{\mathscr{E}}$:$VS_{\mathscr{E}}$, of

345 extended vehicles, $v_{\mathscr{E}}$:$V_{\mathscr{E}}$ — their extension amounting to

  a a dynamic, active and biddable attribute[4], whose value, ti-gpos:TiGpos, at any time, reflects that vehicle's *time-stamped global positions*[5]

  b The vehicle's GNSS receiver calculates its local position, lpos:LPOS, based on these signals.

  c Vehicles access these external attributes via the external attribute channel, attr_TiGPos_ch, cf. Item 100 on Page 56 Sect. 1.3.7 (*"Access to External Attribute Values"*).

  d The vehicle can, on its own volition, offer the timed local position, ti-lpos:TiLPos to the price calculator, $c_{\mathscr{E}}$:$C_{\mathscr{E}}$ along a vehicles-to-calculator channel, v_c_ch.

[3] become or becomes ?

[4] See Sect. 1.2.9 Page 40.

[5] We refer to literature, [71], on GNSS, *global navigation satellite systems*. The local vehicle position, lpos:LPos, is determined from three to four time-stamped signals received from a like number of GNSS satellites.

type
342. $\Delta_{\mathscr{E}}$
343. $F_{\mathscr{E}}$
344. $VS_{\mathscr{E}} = V_{\mathscr{E}}$-**set**
345. $V_{\mathscr{E}}$
345a. TiGPos = $\mathbb{T}$ × GPOS
345a. TiLPos = $\mathbb{T}$ × LPOS
345b. GPOS, LPOS
value
343. **obs_part**_$F_{\mathscr{E}}$: $\Delta_{\mathscr{E}} \to F_{\mathscr{E}}$
344. **obs_part**_$VS_{\mathscr{E}}$: $F_{\mathscr{E}} \to VS_{\mathscr{E}}$
344. vs:**obs_part**_$VS_{\mathscr{E}}(F_{\mathscr{E}})$
channel
345c. {attr_TiGPos_ch[vi]|viLVI•vi ∈ xtr_VIs(vs)}: TiGPos
345d. {v_c_ch[vi,ci]
345d. | vi:VI,ci:CI•vi∈vis∧ci=**uid**_C(c)}:(VI×TiLPos)
value
345a. attr_TiGPos_ch[vi]?
345b. loc_pos: GPOS → LPOS

where vis:VI-**set** is the set unique vehicle identifiers of all vehicles of the requirements domain fleet, f:$F_{\mathscr{R}_{\mathscr{E}}}$. We define two auxiliary functions,

346 xtr_vs, which given a domain, or a fleet, extracts its set of vehicles, and

347 xtr_vis which given a set of vehicles generates their unique identifiers.

value
346. xtr_vs: ($\Delta_{\mathscr{E}}|F_{\mathscr{E}}|VS_{\mathscr{E}}$) → $V_{\mathscr{E}}$-**set**
346. xtr_vs(arg) ≡
346. **is**_$\Delta_{\mathscr{E}}$(arg) → **obs_part**_$VS_{\mathscr{E}}$(**obs_part**_$F_{\mathscr{E}}$(arg)),
346. **is**_$F_{\mathscr{E}}$(arg) → **obs_part**_$VS_{\mathscr{E}}$(arg),
346. **is**_$VS_{\mathscr{E}}$(arg) → arg
347. xtr_vis: ($\Delta_{\mathscr{E}}|F_{\mathscr{E}}|VS_{\mathscr{E}}$) → VI-**set**
347. xtr_vis(arg) ≡ {**uid**_VI(v)|v ∈ xtr_vs(arg)}

Example 99 . Domain Requirements. Extension Toll-road Net: Parts, Properties and Channels: We extend the domain with toll-gates for vehicles entering and exiting the toll-road entry and exit links. Figure 7.2 illustrates the idea of gates.

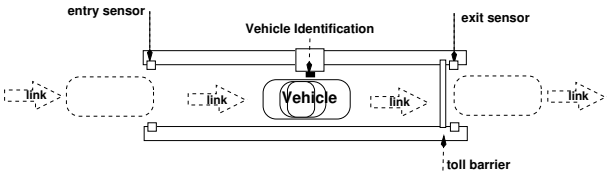

Figure 7.2: A toll plaza gate

Figure 7.2 is intended to illustrate a vehicle entering (or exiting) a toll-road entry link. The toll-gate is equipped with three sensors: an entry sensor, a vehicle identification sensor and an exit sensor. The entry sensor serves to prepare the vehicle identification sensor. The exit sensor serves to prepare the gate for closing when a vehicle has passed. The vehicle identification sensor identifies the vehicle and "delivers" a pair: the current time and the vehicle identifier. Once the vehicle identification sensor has identified a vehicle the gate opens.

348 There is the domain, δ:$\Delta_{\mathscr{E}}$,

349 which contains the extended net, n:$N_{\mathscr{E}}$, with the net extension amounting to the toll-road net, $TRN_{\mathscr{E}}$,

350 that is, the instantiated toll-road net, trn:$TRN_{\mathscr{J}}$, is extended, into trn:$TRN_{\mathscr{E}}$, with entry, eg:EG, and exit, xg:XG, toll-gates.

From entry- and exit-gates we can observe

a their unique identifier and their mereology: being paired with the entry-, respectively exit link and the calculator (by their unique identifiers); further

b a pair of gate enter and leave sensors modeled as external attribute channels, (ges:ES,gls:XS), and

c a time-stamped vehicle identity sensor modeled as external attribute channels.

```
type
348  Δ𝒮
349  N𝒮
350  TRN𝒮 = (EG×XG)* × TRN𝒥
350a  GI
value
348  obs_part_N𝒮: Δ𝒮 → N𝒮
349  obs_part_TRN𝒮: N𝒮 → TRN𝒮
```

```
350a  uid_G: (EG|XG) → GI
350a  obs_mereo_G: (EG|XG) → (LI×CI)
channel
350b  {attr_enter_ch[gi]|gi:GI•...} "enter"
350b  {attr_leave_ch[gi]|gi:GI•...} "leave"
350c  {attr_passing_ch[gi]|gi:GI•...} TIVI
type
350c  TIVI = 𝕋 × VI
```

432

We define some auxiliary functions over toll-road nets, trn:TRN$_{\mathscr{S}}$:

351 xtr_eGℓ extracts the ℓist of entry gates,

352 xtr_xGℓ extracts the ℓist of exit gates,

353 xtr_eGIds$s$ extracts the $s$et of entry gate identifiers,

354 xtr_xGIds$s$ extracts the $s$et of exit gate identifiers,

355 xtr_Gs extracts the $s$et of all gates, and

356 xtr_GIds$s$ extracts the $s$et of all gate identifiers.

433

```
value
351  xtr_eGℓ: TRN𝒮 → EG*
351  xtr_eGℓ(pgl,_) ≡
351    {eg|(eg,xg):(EG,XG)•(eg,xg)∈ elems pgl}
352  xtr_xGℓ: TRN𝒮 → XG*
352  xtr_xGℓ(pgl,_) ≡
352    {xg|(eg,xg):(EG,XG)•(eg,xg)∈ elems pgl}
353  xtr_eGIds: TRN𝒮 → GI-set
353  xtr_eGIds(pgl,_) ≡
353    {uid_GI(g)|g:EG•g ∈ xtr_eGs(pgl,_)}
```

```
354  xtr_xGIds: TRN𝒮 → GI-set
354  xtr_xGIds(pgl,_) ≡
354    {uid_GI(g)|g:EG•g ∈ xtr_xGs(pgl,_)}
355  xtr_Gs: TRN𝒮 → G-set
355  xtr_Gs(pgl,_) ≡
355    xtr_eGs(pgl,_) ∪ xtr_xGs(pgl,_)
356  xtr_GIds: TRN𝒮 → GI-set
356  xtr_GIds(pgl,_) ≡
356    xtr_eGIds(pgl,_) ∪ xtr_xGIds(pgl,_)
```

434

357 A well-formedness condition expresses

- a that there are as many entry end exit gate pairs as there are toll-plazas,
- b that all gates are uniquely identified, and
- c that each entry [exit] gate is paired with an entry [exit] link and has that link's unique identifier as one element of its mereology, the other elements being the calculator identifier and the vehicle identifiers.

The well-formedness relies on awareness of

358 the unique identifier, ci:CI, of the road pricing calculator, c:C, and

359 the unique identifiers, vis:VI-set, of the fleet vehicles.

435

```
value
358  ci:CI
359  vis:VI-set
axiom
357  ∀ n:N𝒥3, trn:TRN𝒥3 •
357    let (exgl,(exl,hl,lll)) = obs_part_TRN𝒥3(n) in
```

```
357a    len exgl = len exl = len hl = len lll + 1
357b  ∧ card xtr_GIds(exgl) = 2 * len exgl
357c  ∧ ∀ i:Nat•i ∈ inds exgl•
357c      let ((eg,xg),(el,xl)) = (exgl(i),exl(i)) in
357c      obs_mereo_G(eg) = (uid_U(el),ci,vis)
357c    ∧ obs_mereo_G(xg) = (uid_U(xl),ci,vis) end end
```

436

Example 100 . Domain Requirements. Extension Parts, Properties and Channels:

360 The road pricing calculator repeatedly receives

- a information, (vi,(τ,pos)):VITIPOS,
- b sent by vehicles as to their identify and time-stamped position
- c over a channel, v_c_ch indexed by the c:C$_{\mathscr{S}}$ and the vehicle identities.

361 The road pricing calculator has a number of attributes:

- a a traffic map, trm:TRM, which, for each vehicle inside the toll-road net, records a chronologically ordered list of each vehicle's timed position, (τ,vp), and
- b a (total) road location function, vplf:VPLF.
  - i The $v$ehicle $p$osition $l$ocation $f$unction, vplf:VPLF, is subject to another function, locate_VPos, which, given a local position, lpos:LPos, yields the vehicle position designated by the GNSS-provided position, or yields the response that the

provided position is off the toll-road net[6].

ii This result is used by the road-pricing calculator to conditionally
   A either update the traffic map, trm:TRM, recording also the relevant time,
   B or reset that vehicle's traffic recording while send a bill for the just completed journey.

type
360a VITIPos = VI × (𝕋 × LPos)
value
360a ... v_c_ch[ci,vi] ? ...
360b ... v_c_ch[ci,vi] ! (vi,(τ,p)) ...
channel
360c {v_c_ch[ci,vi]|vi:VI•vi ∈ vis}:VITIPos

type
361a TRM = VI $\xrightarrow{m}$ (𝕋 × VPos)*
361b VPLF = LPos → VPos | "off_TRN"
value
361(b)i locate_LH: LPos×RLF → (VPos|"off_TRN")
361(b)iiA update_TRM: VI×(𝕋×VPos)→TRM→TRM
361(b)iiB reset_TRM: VI→TRM→TRM

Example 101 . Domain Requirements. Extension Main Sorts:

362 The main sorts of the road-pricing domain, $\Delta_{\mathscr{E}}$, are
   a the net, projected, instantiated (to include the specific toll-road net), made more determinate and now extended, $N_{\mathscr{E}}$, with toll-gates;
   b the fleet, $F_{\mathscr{E}}$,
   c of sets, VS, of extended vehicles, $V_{\mathscr{E}}$;
   d the extended toll-road net, $TRN_{\mathscr{E}}$, extending the instantiated toll-road net, $TRN_{\mathscr{I}}$, with toll-gates; and
   e the road pricing calculator, $C_{\mathscr{E}}$.

type
362. $\Delta_{\mathscr{E}}$
362a. $N_{\mathscr{E}}$
362b. $F_{\mathscr{E}}$
362c. $VS_{\mathscr{E}} = V_{\mathscr{E}}$-**set**
362d. $TRN_{\mathscr{E}} = (EG{\times}XG)^* \times TRN_{\mathscr{I}}$
362e. $C_{\mathscr{E}}$

value
362a. **obs_part**_$N_{\mathscr{E}}$: $\Delta \to N_{\mathscr{E}}$
362b. **obs_part**_$F_{\mathscr{E}}$: $\Delta \to F_{\mathscr{E}}$
362c. **obs_part**_$VS_{\mathscr{E}}$: $\Delta \to VS_{\mathscr{E}}$
362d. **obs_part**_$TRN_{\mathscr{E}}$: $N_{\mathscr{E}} \to TRN_{\mathscr{E}}$
362e. **obs_part**_$C_{\mathscr{E}}$: $\Delta \to C_{\mathscr{E}}$

Example 102 . Domain Requirements. Extension Global Values: We exemplify a road-pricing system behaviour, in Example 103, based on the following global values.

363 There is a given domain, $\delta_{\mathscr{E}}$:$\Delta_{\mathscr{E}}$;
364 there is the net, $n_{\mathscr{E}}$:$N_{\mathscr{E}}$, of that domain;
365 there is toll-road net, $trn_{\mathscr{E}}$:$TRN_{\mathscr{E}}$, of that net;
366 there is a set, $egs_{\mathscr{E}}$:$EG_{\mathscr{E}}$-**set**, of entry gates;
367 there is a set, $xgs_{\mathscr{E}}$:$XG_{\mathscr{E}}$-**set**, of exit gates;
368 there is a set, $gis_{\mathscr{E}}$:$GI_{\mathscr{E}}$-**set**, ofgate identifiers;
369 there is a set, $vs_{\mathscr{E}}$:$V_{\mathscr{E}}$-**set**, of vehicles;
370 there is a set, $vis_{\mathscr{E}}$:$VI_{\mathscr{E}}$-**set**, of vehicle identifiers;
371 there is the road-pricing calculator, $c_{\mathscr{E}}$:$C_{\mathscr{E}}$ and
372 there is its unique identifier, $ci_{\mathscr{E}}$:CI.

value
363. $\delta_{\mathscr{E}}$:$\Delta_{\mathscr{E}}$
364. $n_{\mathscr{E}}$:$N_{\mathscr{E}}$ = **obs_part**_$N_{\mathscr{E}}$($\delta_{\mathscr{E}}$)
365. $trn_{\mathscr{E}}$:$TRN_{\mathscr{E}}$ = **obs_part**_$TRN_{\mathscr{E}}$($n_{\mathscr{E}}$)
366. $egs_{\mathscr{E}}$:EG-**set** = xtr_egs($trn_{\mathscr{E}}$)
367. $xgs_{\mathscr{E}}$:XG-**set** = xtr_xgs($trn_{\mathscr{E}}$)
368. $gis_{\mathscr{E}}$:XG-**set** = xtr_gis($trn_{\mathscr{E}}$)
369. $vs_{\mathscr{E}}$:$V_{\mathscr{E}}$-**set** = **obs_part**_VS(**obs_part**_$F_{\mathscr{E}}$($\delta_{\mathscr{E}}$))
370. $vis_{\mathscr{E}}$:VI-**set** = {**uid**_VI($v_{\mathscr{E}}$)|$v_{\mathscr{E}}$:$V_{\mathscr{E}}$•$v_{\mathscr{E}}$ ∈ $vs_{\mathscr{E}}$}
371. $c_{\mathscr{E}}$:$C_{\mathscr{E}}$ = **obs_part**_$C_{\mathscr{E}}$($\delta_{\mathscr{E}}$)
372. $ci_{\mathscr{E}}$:$CI_{\mathscr{E}}$ = **uid**_CI($c_{\mathscr{E}}$)

Example 103 . Domain Requirements. Extension System Behaviour: We shall model the behaviour of the road-pricing system as follows: we shall only model behaviours related to atomic parts; we shall not model behaviours of hubs and links; thus we shall model only the set of behaviours of vehicles, veh, the set of behaviours of toll-gates, gate, and the behaviour of the road-pricing calculator, calc.

[6] The vplf:VPLF function is constructed from awareness of the topology extended net, $n_{\mathscr{E}}$:$N_{\mathscr{E}}$, including the mereology and the geodetic and cadestral attributes of links and hubs.

373 The road-pricing system behaviour, sys, is expressed as
  a the parallel, ‖, (distributed) composition of the behaviours of all vehicles, with the parallel composition of
  b the parallel (likewise distributed) composition of the behaviours of all entry gates, with the parallel composition of
  c the parallel (likewise distributed) composition of the behaviours of all exit gates, with the parallel composition of
  d the behaviour of the road-pricing calculator,

444

```
value
373.  sys: Unit → Unit
373.  sys() ≡
373a.       ‖ {veh(uid_V(v),(ci,gis),𝕋TiGPos)|v:V•v ∈ vs𝒮}
373b.    ‖ ‖ {gate("Entry")(uid_EG(eg),obs_mereo_G(eg),(𝕋enter,𝕋passing,𝕋leave))|eg:EG•eg ∈ egs𝒮}
373c.    ‖ ‖ {gate("Exit")(uid_EG(xg),obs_mereo_G(xg),(𝕋enter,𝕋passing,𝕋leave))|xg:XG•xg ∈ xgs𝒮}
373d.       ‖ calc(ci𝒮,(vis𝒮,gis𝒮))(rlf)(trm)
```

445

Example 104 . Domain Requirements. Extension Vehicle Behaviour: We refer to the vehicle behaviour, in the domain, described in Chapter 6's **The Road Traffic System Behaviour** Pages 148–149.

374 Instead of moving around by explicitly expressed internal non-determinism[7] vehicles move around by unstated internal non-determinism and instead receive their current position from the global positioning subsystem.
  a At each moment the vehicle receives its time-stamped local position, tilpos:TiLPos,
  b which it then proceeds to communicate, with its vehicle identification, (vi,tilpos), to the road pricing subsystem —
  c whereupon it resumes its vehicle behaviour.

446

```
value
374.  veh: vi:VI×(ci:CI×gis:GI-set)×𝕋TiGPos →
374.     out v_c_ch[ci,vi] Unit
374.  veh(vi,(ci,gis),attr_TiGPos_ch[vi]) ≡
374a.     let (τ,gpos) = attr_TiGPos_ch[vi]? in
374a.     let lpos = loc_pos(gpos) in
374b.     v_c_ch[ci,vi] ! (vi,(τ,lpos)) ;
374c.     veh(vi,(ci,gis),attr_TiGPos_ch[vi]) end end
374.    pre vi ∈ vis𝒮 ∧ ci = ci𝒮 ∧ gis = gis𝒮
```

447

Example 105 . Domain Requirements. Extension Gate Behaviour: The entry and the exit gates have "vehicle enter", "vehicle leave" and "vehicle time and identification" sensors. The following assumption can now be made: during the time interval between a gate's vehicle "enter" sensor having first sensed a vehicle entering that gate and that gate's "leave" sensor having last sensed that vehicle leaving that gate that gate's "vehicle time and identification" sensor registers the time when the vehicle is entering the gate and that vehicle's unique identification. We sketch the toll-gate behaviour: 448

375 We parameterise the toll-gate behaviour as either an entry or an exit gate.
376 Toll-gates
  a inform the calculator of place (i.e., link) and time of entering and exiting of identified vehicles
  b over an appropriate array of channels.
377 Toll-gates operate autonomously and cyclically.
  a The **attr_**Enter event "triggers" the behaviour specified in formula line Item 377b–377d.
  b The time-of-entry and the identity of the entering (or exiting) vehicle is sensed via external attribute channel inputs.
  c Then the road pricing calculator is informed of time-of-entry and of vehicle vi entering (or exiting) link li.
  d And finally, after that vehicle has left the entry or exit gate that toll-gate's behaviour is resumed.

[7] We refer to Items 315b, 315c on Page 148 and 316b, 316(c)ii, 317 on Page 149

449

The toll-gate behaviour, gate:

```
type
375   EE = "Enter" | "Exit"
376a  GCM = EE × (𝕋 × VI × LI)
channel
376b  {g_c_ch[uid_GI(g),ci]|g:G,ci:CI•g ∈ gates(trn)} GCM
value
377   gate: ee:EE×gi:GI×(ci:CI×VI-set×LI)×(𝕌enter×𝕌passing×𝕌leave) → out g_c_ch[gi,ci] Unit
377   gate(ee,gi,(ci,vis,li),ea:(attr_enter_ch[gi],attr_passing_ch[gi],attr_leave_ch[gi])) ≡
377a    attr_enter_ch[gi] ? ;
377b    let (τ,vi) = attr_passing_ch[gi] ? in assert vi ∈ vis
377c    g_c_ch[gi,ci] ! (ee,(τ,(vi,li)));
377d    attr_leave_ch[gi] ?
377d    gate(ee)(gi,(ci,vis,li),ea)
377     end
377     pre ci = ci𝒻 ∧ vis = vis𝒻 ∧ li ∈ lis𝒻
```

450

Example 106 . Domain Requirements. Extension Calculator Behaviour:

378 The road-pricing calculator alternates between (offering to accept communication with)

- a either any vehicle
- b or any toll-gate.

```
378.   calc: ci:CI×(vis:VI-set×gis:GI-set)→RLF→TRM→
378a.       in {v_c_ch[ci,vi]|vi:VI•vi ∈ vis},
378b.          {g_c_ch[ci,gi]|gi:GI•gi ∈ gis} Unit
378.   calc(ci,(vis,gis))(rlf)(trm) ≡
378a.     react_to_vehicles(ci,(vis,gis))(rlf)(trm)
378.   []
378b.     react_to_gates(ci,(vis,gis))(rlf)(trm)
378.    pre ci = ci𝒻 ∧ vis = vis𝒻 ∧ gis = gis𝒻
```

451

379 If the communication is from a vehicle inside the toll-road net

- a then its toll-road net position, vp, is found from the road location function, rlf,
- b and the calculator resumes its work with the traffic map, trm, suitable updated,
- c otherwise the calculator resumes its work with no changes.

```
378a.  react_to_vehicles(ci,(vis,gis))(rlf)(trm) ≡
378a.    let (vi,(τ,lpos)) =
378a.      []{v_c_ch[ci,vi]|vi:VI•vi∈ vis} in
379.     if vi ∈ dom trm
379a.      then let vp = rlf(lpos) in
379b.        calc(ci,(vis,gis))(rlf)(trm†[vi↦trm⌢⟨(τ,vp)⟩]) end
379c.      else calc(ci,(vis,gis))(rlf)(trm) end end
```

452

380 If the communication is from a gate,

- a then that gate is either an entry gate or an exit gate;
- b if it is an entry gate
- c then the calculator resumes its work with the vehicle (that passed the entry gate) now recorded, afresh, in the traffic map, trm.
- d Else it is an exit gate and
- e the calculator concludes that the vehicle has ended its to-be-paid for journey inside the toll-road net, and hence to be billed;
- f then the calculator resumes its work with the vehicle (that passed the exit gate) now removed from the traffic map, trm.

```
378b.  react_to_gates(ci,(vis,gis))(rlf)(trm) ≡
378b.    let (ee,(τ,(vi,li))) =
378b.      []{g_c_ch[ci,gi]|gi:GI•gi∈ gis} in
380a.    case ee of
380b.      "Enter" →
380c.        calc(ci,(vis,gis))(rlf)(trm∪[vi↦⟨(τ,(li,0))⟩]),
380d.      "Exit" →
380e.        billing(vi,trm(vi)⌢⟨(τ,(li,1))⟩);
380f.        calc(ci,(vis,gis))(rlf)(trm\{vi}) end end
```

•••

We have made relevant external attributes explicit parameters of their (corresponding part) processes. We refer to Sect. 1.3.7.

A Domain Extension Operator

454

Domain extension takes a (more-or-less) deterministic requirements description, $\mathscr{R}_{\mathscr{D}}$, and yields an extended requirements prescription, $\mathscr{R}_{\mathscr{E}}$, which extends the domain description, $\mathscr{D}$, and, "at the same time", "extends" the requirements prescription, $\mathscr{R}_{\mathscr{D}}$,

- **type** extension: $\mathscr{R}_{\mathscr{D}} \rightarrow \mathscr{R}_{\mathscr{E}}$

Semantically $\mathscr{R}_{\mathscr{D}}$ denotes a possibly infinite set of meanings, say $\mathbb{R}_{\mathbb{D}}$, and $\mathscr{R}_{\mathscr{E}}$ denotes a possibly infinite set of meanings, say $\mathbb{R}_{\mathbb{E}}$, but now the relation $\mathbb{R}_{\mathscr{E}} \sqsubseteq \mathbb{R}_{\mathscr{D}}$ is not necessarily satisfied — but instead some conservative extension relation $\mathbb{R}_{\mathbb{E}} \sqsupseteq \mathbb{D}_{\mathbb{D}}$ is satisfied.

7.2.5 Requirements Fitting

455

Often a domain being described "fits" onto, is "adjacent" to, "interacts" in some areas with, another domain: *transportation* with *logistics*, *health-care* with *insurance*, *banking* with *securities trading* and/or *insurance*, and so on. The issue of requirements fitting arises when two or more software development 456
projects are based on what appears to be the same domain. The problem then is to harmonise the two or more software development projects by harmonising, if not too late, their requirements developments.

Some Definitions

457

We thus assume that there are n domain requirements developments, $d_{r_1}, d_{r_2}, \ldots, d_{r_n}$, being considered, and that these pertain to the same domain — and can hence be assumed covered by a same domain description.

458

Definition 41 . ***Requirements Fitting:*** *By* ***requirements fitting*** *we mean a* harmonisation *of* $n > 1$ *domain requirements that have overlapping (shared) not always consistent parts and which results in n* partial domain requirements*', $p_{d_{r_1}}, p_{d_{r_2}}, \ldots, p_{d_{r_n}}$, and m* shared domain requirements*, $s_{d_{r_1}}, s_{d_{r_2}}, \ldots, s_{d_{r_m}}$, that "fit into" two or more of the partial domain requirements* ■ *The above definition pertains to the result of 'fitting'. The next definition pertains to the act, or process, of 'fitting'.*

459

Definition 42 . ***Requirements Harmonisation:*** *By* ***requirements harmonisation*** *we mean a number of alternative and/or co-ordinated prescription actions, one set for each of the domain requirements actions:* Projection, Instantiation, Determination *and* Extension*. They are – we assume n separate* 460
software product requirements: Projection*: If the n product requirements do not have the same projections, then identify a common projection which they all share, and refer to it is the* common projection*. Then develop, for each of the n product requirements, if required, a* specific projection *of the common one. Let there be m such specific projections, $m \leq n$.* Instantiation*: First instantiate the common projection,* 461
if any instantiation is needed. Then for each of the m specific projections instantiate these, if required. Determination*: Likewise, if required, "perform" "determination" of the possibly instantiated common projection, and, similarly, if required, "perform" "determination" of the up to m possibly instantiated projections.* Extension*: Finally "perform extension" likewise: First, if required, of the common projection* 462
(etc.), then, if required, on the up m specific projections (etc.). These harmonization developments may possibly interact and may need to be iterated ■

By a **partial domain requirement**s we mean a domain requirements which is short of (that is, is missing) some prescription parts: text and formula ■ By a **shared domain requirement**s we mean a domain requirements ■ By **requirements fitting** *m* shared *d*omain *r*equirements texts, *sdrs*, into *n* partial domain 463
requirements we mean that there is for each *p*artial *d*omain *r*equirements, pdr_i, an identified *s*ubset of *sdrs* (could be all of *sdrs*), $ssdrs_i$, such that textually conjoining $ssdrs_i$ to pdr_i, i.e., $ssdrs_i \oplus pdr_i$ can be claimed to yield the "original" d_{r_i}, that is, $\mathscr{M}(ssdrs_i \oplus pdr_i) \subseteq \mathscr{M}(d_{r_i})$, where $\mathscr{M}$ is a suitable meaning function over prescriptions ■

Requirements Fitting Procedure — A Sketch 464

Requirements fitting consists primarily of a pragmatically determined sequence of analytic and synthetic ('fitting') steps. It is first decided which n domain requirements documents to fit. Then a 'manual' analysis is made of the selected, n domain requirements. During this analysis tentative shared domain requirements are identified. It is then decided which m shared domain requirements to single out. This decision results in a tentative construction of n partial domain requirements. An analysis is made of the tentative partial and shared domain requirements. A decision is then made whether to accept the resulting documents or to iterate the steps above.

Requirements Fitting – An Example 465

Example 107 . Domain Requirements. Fitting A Sketch: We postulate two domain requirements: We have outlined a domain requirements development for software support for a road-pricing system. We have earlier hinted at domain operations related to insertion of new and removal of existing links and hubs. We can therefore postulate that there are two domain requirements developments, both based on the transport domain:

- one, $d_{r_{\text{toll}}}$, for a road-pricing system, and,
- another, $d_{r_{\text{maint.}}}$, for a toll-road link and hub building and maintenance system monitoring and controlling link and hub quality and for development.

466 The fitting procedure now identifies the shared awareness by both $d_{r_{\text{toll}}}$ and $d_{r_{\text{maint.}}}$ of nets (N), hubs (H) and links (L). We conclude from this that we can single out a common requirements for software that manages net, hubs and links. Such software requirements basically amounts to requirements for a database system. A suitable such system, say a 467 relational database management system, DB_{rel}, may already be available with the customer.

In any case, where there before were two requirements ($d_{r_{\text{toll}}}, d_{r_{\text{maint.}}}$) there are now four: (i) $d'_{r_{\text{toll}}}$, a modification of $d_{r_{\text{toll}}}$ which omits the description sections pertaining to the net; (ii) $d'_{r_{\text{maint.}}}$, a modification of $d_{r_{\text{maint.}}}$ which likewise omits the description sections pertaining to the net; (iii) $d_{r_{\text{net}}}$, which contains what was basically omitted in $d'_{r_{\text{toll}}}$ and $d'_{r_{\text{maint.}}}$; and (iv) $d_{r_{\text{db:i/f}}}$ (db:i/f for database interface) which prescribes a mapping between type names of $d_{r_{\text{net}}}$ and relation and attribute names of DB_{rel} ■

Much more can and should be said, but this suffices as an example in a software engineering methodology paper.

7.2.6 Domain Requirements Consolidation 468

After projection, instantiation, determination, extension and fitting, it is time to review, consolidate and possibly restructure (including re-specify) the domain requirements prescription before the next stage of requirements development.

7.3 Interface Requirements 469

By an **interface requirements** we mean a requirements prescription which refines and extends the domain requirements by considering those requirements of the domain requirements whose endurants (parts, materials) and perdurants (actions, events and behaviours) are ***"shared"*** between the domain and the machine (being requirements prescribed) ■

7.3.1 Shared Phenomena 470

By **sharing** we mean (a) that an ***endurant*** is represented both in the domain and "inside" the machine, and that its machine representation must at suitable times reflect its state in the domain; and/or (b) that an ***action*** requires a sequence of several "on-line" interactions between the machine (being requirements 471 prescribed) and the domain, usually a person or another machine; and/or (c) that an ***event*** arises either in the domain, that is, in the environment of the machine, or in the machine, and need be communicated to the

machine, respectively to the environment; and/or (d) that a ***behaviour*** is manifested both by actions and events of the domain and by actions and events of the machine ■ So a systematic reading of the domain requirements shall result in an identification of all shared endurants, parts, materials and components; and perdurants actions, events and behaviours. Each such shared phenomenon shall then be individually dealt with: ***endurant sharing*** shall lead to interface requirements for data initialisation and refreshment; ***action sharing*** shall lead to interface requirements for interactive dialogues between the machine and its environment; ***event sharing*** shall lead to interface requirements for how such event are communicated between the environment of the machine and the machine; and ***behaviour sharing*** shall lead to interface requirements for action and event dialogues between the machine and its environment.

•••

We shall now illustrate these domain interface requirements development steps with respect to our ongoing example.

7.3.2 Shared Endurants

We "split" our interface requirements development into two separate steps: the development of $d_{r_{\text{net}}}$ (the common domain requirements for the shared hubs and links), and the co-development of $d_{r_{\text{db:i/f}}}$ (the common domain requirements for the interface between $d_{r_{\text{net}}}$ and DB_{rel} — under the assumption of an available relational database system DB_{rel})

Example 108 . Interface Requirements. Shared Endurants: The main shared endurants are the net (hubs, links) and the vehicles. As domain endurants hubs and links undergo changes, all the time, with respect to the values of several attributes: *length, cadestral information, names, wear and tear* (where-ever applicable), *last/next scheduled maintenance* (where-ever applicable), *state* and *state space*, and many others. Similarly for vehicles: their position, velocity and acceleration, and many other attributes. When planning the common domain requirements for the net, i.e., the hubs and links, we enlarge our scope of requirements concerns beyond the two so far treated ($d_{r_{\text{toll}}}$, $d_{r_{\text{maint.}}}$) in order to make sure that the shared relational database of nets, their hubs and links, may be useful beyond those requirements. We then come up with something like hubs and links are to be represented as tuples of relations; each net will be represented by a pair of relations a hubs relation and a links relation; each hub and each link may or will be represented by several tuples; etcetera. In this database modeling effort it must be secured that "standard" operations on nets, hubs and links can be supported by the chosen relational database system DB_{rel}

Data Initialisation

As part of $d_{r_{\text{net}}}$ one must prescribe data initialisation, that is provision for an interactive user interface dialogue with a set of proper display screens, one for establishing net, hub or link attributes names and their types, and, for example, two for the input of hub and link attribute values. Interaction prompts may be prescribed: next input, on-line vetting and display of evolving net, etc. These and many other aspects may therefore need prescriptions. Essentially these prescriptions concretise the insert and remove link and hub actions.

Example 109 . Interface Requirements. Shared Endurant Initialisation: The domain is that of the road net, n:N, say of Chapter 6 — see also Example 108 By 'shared road net initialisation' we mean the "ab initio" establishment, "from scratch" of a data base recording the properties of all links, l:L, and hubs, h:H, their unique identifications, **uid_L**(l) and **uid_H**(h), their mereologies, **obs_mereo_L**(l) and **obs_mereo_H**(h) , and the initial values of all their attributes, **attributes**(l) and **attributes**(h).

381 There are r_l and r_h "recorders" recording link, respectively hub properties with each recorder having a unique identity,

382 Each recorder is charged with a set of links or a set of hubs according to some partitioning of all such.

383 The recorders inform a central data base, net_db, of their recordings:

   a (ri,nol,(u_j,m_j,$attrs_j$)) where

   b ri is the identity of the recorder,

c nol is either `link` or `hub`,

d u_j = **uid_**L(l) or **uid_**H(h) for some link or hub,

e m_j = **obs_mereo_**L(l) or **obs_mereo_**H(h) for that link or hub and

f $attrs_j$ = **attributes**(l) or **attributes**(h) for that link or hub.

```
type
381. RI
value
381. rl,rh:NAT axiom rl>0 ∧ rh>0
type
383a. M = RI×"link"×LNK | RI×"hub"×HUB
383a. LNK = LI × HI-set × LATTRS
383a. HUB = HI × LI-set × HATTRS
```

```
value
382. partitioning: L-set → Nat → (L-set)*
382.                | H-set → Nat → (H-set)*
382. partitioning(s)(r) as sl
382.   post: len sl = r
382.     ∧ ∪ elems sl = s
382.     ∧ ∀ si,sj:(L-set|H-set) •
382.         si≠{}
382.       ∧ sj≠{}
382.       ∧ {si,sj}⊆elems ss ⇒ si ∩ sj = {}
```

384 The $r_l + r_h$ recorder behaviours interact with the one net_db behaviour

```
channel
384. r_db: RI×(LNK|HUB)
value
384. LNK_recorder: RI → L-set → out r_db Unit
384. HUB−recorder: RI → H-set → out r_db Unit
384. net_db: Unit → in r_db Unit
```

385 The data base behaviour, net_db, offers to receive messages from the link an hub recorders.

386 And the data base behaviour, net_db, deposits these messages in respective variables.

387 Initially there is a net, $n : N$,

388 from which is observed its links and hubs.

389 These sets are partitioned into r_l, respectively r_h length lists of non-empty links and hubs.

390 The ab-initio data initialisation behaviour, ab_initio_data, is then the parallel composition of link recorder, hub recorder and data base behaviours with link and hub recorder being allotted appropriate link, respectively hub sets.

391 We construct, for technical reasons, as the reader will soon see, disjoint lists of link, respectively hub recorder identities.

```
value
385. net_db:
variable
386. lnk_db: (RI×LNK)-set
386. hub_db: (RI×HUB)-set
value
387. n:N
388. ls:L-set = obs_Ls(obs_LS(n))
388. hs:H-set = obs_Hs(obs_HS(n))
389. lsl:(L-set)* = partition(ls)(rl)
389. lhl:(H-set)* = partition(hs)(rh)
391. rill:RI* axiom len rill = rl = card elems rill
391. rihl:RI* axiom len rihl = rh = card elems rihl
```

```
390.  ab_initio_data: Unit → Unit
390.  ab_initio_data() ≡
390.    ‖ {lnk_rec(rill[i])(lsl[i])|i:Nat•1≤i≤rl}
390.    ‖ {hub_rec(rihl[i])(lhl[i])|i:Nat•1≤i≤rh}
390.    ‖ net_db()
```

392 The link and the hub recorders are near-identical behaviours.

- a They both revolve around an imperatively stated **for all ... do ... end**. The selected link (or hub) is inspected and the "data" for the data base is prepared from
- b the unique identifier,
- c the mereology, and
- d the attributes.
- e These "data" are sent, as a message, prefixed the senders identity, to the data base behaviour.
- f We presently leave the ... unexplained.

```
value
384.  link_rec: RI → L-set → Unit
392.  link_rec(ri,ls) ≡
392a.    for ∀ l:L•l ∈ ls do uid_L(l)
392b.      let lnk = (uid_L(l),
392c.                 obs_mereo_L(l),
392d.                 attributes(l)) in
392e.      rdb ! (ri,"link",lnk);
392f.      ... end
392a.    end
```

```
384.  hub_rec: RI × H-set → Unit
392.  hub_rec(ri,hs) ≡
392a.    for ∀ h:H•h ∈ hs do uid_H(h)
392b.      let hub = (uid_L(h),
392c.                 obs_mereo_H(h),
392d.                 attributes(h)) in
392e.      rdb ! (ri,"hub",hub);
392f.      ... end
392a.    end
```

393 The net_db data base behaviour revolves around a seemingly "never-ending" cyclic process.

394 Each cycle "starts" with acceptance of some,

395 either link or hub data.

396 If link data then it is deposited in the link data base,

397 if hub data then it is deposited in the hub data base.

```
value
393.  net_db() ≡
394.    let (ri,loh,data) = r_db ? in
395.    case loh of
396.      "link" → ... ; lnk_db := lnk_db ∪ (ri,data),
397.      "hub"  → ... ; hub_db := hub_db ∪ (ri,data)
395.    end end ;
393′.    ... ;
393.    net_db()
```

The above model is an idealisation. It assumes that the link and hub data represent a well-formed net. Included in this well-formedness are the following issues: (a) that all link or hub identifiers are communicated exactly once, (b) that all mereologies refer to defined parts, and (c) that all attribute values lie within an appropriate value range. If we were to cope with possible recording errors then we could, for example, extend the model as follows: (i) when a link or a hub recorder has completed its recording then it increments an initially zero counter (say at Item 392f, Page 171); (ii) before the net data base recycles it tests whether all recording sessions has ended and then proceeds to check the data base for well-formedness issues (a–b–c) (say at Item 393′, Page 171) ■

The above example illustrates the 'interface' phenomenon: In the formulas, for example, we show both manifest domain entities, viz., n, l, h etc., and abstract (required) software objects, viz., $(ui, me, attrs)$.

Data Refreshment

As part of $d_{r_{\text{net}}}$ one must also prescribe data refreshment: an interactive user interface dialogue with a set of proper display screens one for selecting the updating of net, of hub or of link attribute names and their types and, for example, two for the respective update of hub and link attribute values. Interaction-prompts may be prescribed: next update, on-line vetting and display of revised net, etc. These and many other aspects may therefore need prescriptions. These prescriptions also concretise insert and remove link and hub actions.

7.3.3 Shared Actions, Events and Behaviours

We illustrate the ideas of shared actions, events and behaviours through the domain requirements extension of Sect. 7.2.4, more specifically Examples 103–105 Pages 164–166.

Example 110 . **Interface Requirements. Shared Actions, Events and Behaviours**:

This Example has yet to be written

Examples 104–106, Pages 165–166,
illustrate shared interactive actions, events and behaviours.

7.4 Machine Requirements

7.4.1 Delineation of Machine Requirements

On Machine Requirements

Definition 43 . ***Machine Requirements:*** *By* ***machine requirements*** *we shall understand such requirements which can be expressed "sôlely" using terms from, or of the machine* ■

Definition 44 . ***The Machine:*** *By the* ***machine*** *we shall understand the hardware and software to be built from the requirements* ■

The expression *which can be expressed "sôlely" using terms from, or of the machine* shall be understood with "a grain of salt". Let us explain. The machine requirements statements may contain references to domain entities but these are meant to be generic references, that is, references to certain classes of entities in general. We shall illustrate this "genericitiy" in some of the examples below.

Machine Requirements Facets

We shall, in particular, consider the following five kinds of machine requirements: *performance requirements, dependability requirements, maintenance requirements, platform requirements* and *documentation requirements.*

7.4.2 Performance Requirements

Definition 45 . ***Performance Requirements:*** *By* performance requirements *we mean machine requirements that prescribe storage consumption, (execution, access, etc.) time consumption, as well as consumption of any other machine resource: number of CPU units (incl. their quantitative characteristics such as cost, etc.), number of printers, displays, etc., terminals (incl. their quantitative characteristics), number of "other", ancillary software packages (incl. their quantitative characteristics), of data communication bandwidth, etcetera* ■

501

Example 111 . Machine Requirements. Road-pricing System Performance: Possible road pricing system performance requirements could evolve around: maximum number of cars entering and leaving the sum total of all gates within a minimum period — for example 10.000 maximum within any interval of 10 seconds minimum; maximum time between a car entering a gate and the raising of the gate barrier — for example 3 seconds; etcetera, We cannot be more specific: that would require more details about gate sensors and gate barriers.

7.4.3 Dependability Requirements

502

MORE TO COME

Failures, Errors and Faults

To properly define the concept of *dependability* we need first introduce and define the concepts of *failure*, *error*, and *fault*. 503

Definition 46 . *Failure:* *A machine failure occurs when the delivered service deviates from fulfilling the machine function, the latter being what the machine is aimed at [131]* ■

504

Definition 47 . *Error:* *An error is that part of a machine state which is liable to lead to subsequent failure. An error affecting the service is an indication that a failure occurs or has occurred [131]* ■

505

Definition 48 . *Fault:* *The adjudged (i.e., the 'so-judged') or hypothesised cause of an error is a fault [131]* ■

The term hazard is here taken to mean the same as the term fault. One should read the phrase: "adjudged or hypothesised cause" carefully: In order to avoid an unending trace backward as to the cause,[8] we stop at *the cause which is intended to be prevented or tolerated.* 506

Definition 49 . *Machine Service:* *The service delivered by a machine is its behaviour as it is perceptible by its user(s), where a user is a human, another machine or a(nother) system which interacts with it [131]* ■

507

Definition 50 . *Dependability:* *Dependability is defined as the property of a machine such that reliance can justifiably be placed on the service it delivers [131]* ■

We continue, less formally, by characterising the above defined concepts [131]. "A given machine, operating in some particular environment (a wider system), may fail in the sense that some other machine (or system) makes, or could in principle have made, a *judgement* that the activity or inactivity of the given machine constitutes a *failure*". The concept of *dependability* can be simply defined as "the quality or the characteristic of being dependable", where the adjective 'dependable' is attributed to a machine whose failures are judged sufficiently rare or insignificant. *Impairments* to dependability are the unavoidably ex- 508
pectable circumstances causing or resulting from "undependability": faults, errors and failures. *Means* for dependability are the techniques enabling one to provide the ability to deliver a service on which reliance can be placed, and to reach confidence in this ability. *Attributes* of dependability enable the properties which are expected from the system to be expressed, and allow the machine quality resulting from the impairments and the means opposing them to be assessed. Having already discussed the "threats" aspect, 509
we shall therefore discuss the "means" aspect of the *dependability tree*.

[8] An example: "The reason the computer went down was the current supply did not deliver sufficient voltage, and the reason for the drop in voltage was that a transformer station was overheated, and the reason for the overheating was a short circuit in a plant nearby, and the reason for the short circuit in the plant was that . . . , etc."

- Attributes:
  - Accessibility
  - Availability
  - Integrity
  - Reliability
  - Safety
  - Security
- Means:
  - Procurement
    - Fault prevention
    - Fault tolerance
  - Validation
    - Fault removal
    - Fault forecasting
- Threats:
  - Faults
  - Errors
  - Failures

Despite all the principles, techniques and tools aimed at *fault prevention, faults* are created. Hence the need for *fault removal. Fault removal* is itself imperfect. Hence the need for *fault forecasting*. Our increasing dependence on computing systems in the end brings in the need for *fault tolerance*. We refer to special texts [111] on the above four topics.

Definition 51 . ***Dependability Attribute:*** *By a dependability attribute we shall mean either one of the following: accessibility, availability, integrity, reliability, robustness, safety and security. That is, a machine is dependable if it satisfies some degree of "mixture" of being accessible, available, having integrity, and being reliable, safe and secure*

The crucial term above is "satisfies". The issue is: To what "degree"? As we shall see — in a later section — to cope properly with dependability requirements and their resolution requires that we deploy mathematical formulation techniques, including analysis and simulation, from statistics (stochastics, etc.). In the next seven subsections we shall characterise the dependability attributes further. In doing so we have found it useful to consult [111].

Accessibility

Usually a desired, i.e., the required, computing system, i.e., the machine, will be used by many users — over "near-identical" time intervals. Their being granted access to computing time is usually specified, at an abstract level, as being determined by some internal nondeterministic choice, that is: essentially by *"tossing a coin"!* If such internal nondeterminism was carried over, into an implementation, some *"coin tossers"* might never get access to the machine.

Definition 52 . ***Accessibility:*** *A system being accessible — in the context of a machine being dependable — means that some form of "fairness" is achieved in guaranteeing users "equal" access to machine resources, notably computing time (and what derives from that)*

Example 112 . **Machine Requirements. Road-pricing System Accessibility**: Fairness of the calculator behaviour, cf. formula Item 378 on Page 166 ([]) shall mean that "earlier" (wrt. time-stamped) messages from either vehicles or from gates shall be accepted by the calculator before "later" such messages. This is guaranteed by the semantics of RSL. And, hence, shall be guaranteed by any implementation of the deterministic choice []

Availability

Usually a desired, i.e., the required, computing system, i.e., the machine, will be used by many users — over "near-identical" time intervals. Once a user has been granted access to machine resources, usually computing time, that user's computation may effectively make the machine unavailable to other users — by "going on and on and on"!

Definition 53 . ***Availability:*** *By availability — in the context of a machine being dependable — we mean its readiness for usage. That is, that some form of "guaranteed percentage of computing time" per time interval (or percentage of some other computing resource consumption) is achieved — hence some form of "time slicing" is to be effected* ■

518

Example 113 . Machine Requirements. Road-pricing System Availability: Formula Item 374b (Page 165) specify that vehicles "continuously" inform the calculator (cf. formula Items 378 on Page 166) of their time-stamped local position. This may lead you to think that these messages may effectively "block out" "concurrent" messages from toll-road gates. In an implementation we may choose to discretize vehicle-to-calculator messages. That is, to "space them apart", some time interval — so long as an "intentional semantics is maintained"

Integrity

519

Definition 54 . *Integrity:* *A system has* integrity *— in the context of a machine being dependable — if it is and remains unimpaired, i.e., has no faults, errors and failures, and remains so, without these, even in the situations where the environment of the machine has faults, errors and failures* ■

Integrity seems to be a highest form of dependability, i.e., a machine having integrity is 100% dependable! The machine is sound and is incorruptible. 520

Example 114 . Machine Requirements. Road-pricing System Integrity: We divide the integrity concerns for the road-pricing computing and communications system into two "spheres" (I–II): (I) the integrity of the sensor and actuator equipment attached to (I.1) vehicles (i.e., their GNSS attributes), and to (I.2) toll-road gates: (I.2.1) in/out sensors, (I.2.2) vehicle identifiers and (I.2.3) gates; and (II) the software of the road-pricing computing and communications system, that is, the software which interfaces with vehicles, toll-gates and the calculator. As for the integrity of 521
the the sensor and actuator equipment we do not require that the road-pricing computing and communications system is 100% dependable, It is satisfactory if it retains its (i) accessibility, (ii) availability, (iii) reliability, (iv) safety and (v) security in the presence of maintenance. As for the integrity of the software we require that it (a) is **proven correct** with 522
respect to domain and requirements specifications under the assumption that sensor and actuator equipment functions with 100%'s integrity; (b) and where correctness proofs may not be feasible or possible, that the software is appropriately **model-checked**; (c) and where "complete" model-checks may not be feasible or possible, that the software is **formally tested**

523

Definition 55 . *Reliability:* *A system being* reliable *— in the context of a machine being dependable — means some measure of continuous correct service, that is, measure of time to failure*

524

Example 115 . Machine Requirements. Road-pricing System Reliability: *Mean-time between failures*, MTBF, (i) of any vehicle's GNSS correct recording of local position must be at least 30.000 hours; (ii) of any toll-gate complex, that is, it's ability to correctly identify a passing vehicle, or it's ability to correctly close and open gates must be at least 20.000 hours

Safety

525

Definition 56 . *Safety:* *By safety — in the context of a machine being dependable — we mean some measure of continuous delivery of service of either correct service, or incorrect service after benign failure, that is: Measure of time to catastrophic failure* ■

526

Example 116 . Machine Requirements. Road-pricing System Safety: *Mean time to catastrophic failure*, MTCF, (i) for a vehicle's GNSS to function properly shall be 60.000 hours; and (ii) of any toll-gate complex, that is, it's ability to correctly identify a passing vehicle, or it's ability to correctly close and open gates must be at least 40.000 hours

Security

527

We shall take a rather limited view of security. We are not including any consideration of security against brute-force terrorist attacks. We consider that an issue properly outside the realm of software engineering. Security, then, in our limited view, requires a notion of *authorised user*, with authorised users being fine-grained authorised to access only a well-defined subset of system resources (data, functions, etc.). An
528 *unauthorised user* (for a resource) is anyone who is not authorised access to that resource.

Definition 57 . ***Security:*** *A system being* secure *— in the context of a machine being dependable — means that an* unauthorised user*, after believing that he or she has had access to a requested system resource: (i) cannot find out what the system resource is doing, (ii) cannot find out how the system resource is working and (iii) does not know that he/she does not know! That is, prevention of unauthorised access to computing and/or handling of information (i.e., data)* ■

529

Example 117 . **Machine Requirements. Road-pricing System Security**: Vehicles are authorised (i) to receive GNSS timed global positions, but not to tamper with, e.g. misrepresent them, are authorised (ii) to, and shall correctly compute their local positions based on the received global positions, and are finally authorised (iii) to, and shall correctly inform the calculator of their timed local positions

Robustness

530

Definition 58 . ***Robustness:*** *A system is* robust *— in the context of dependability — if it retains its attributes after failure, and after maintenance* ■

Thus a robust system is "stable" across failures and "across" possibly intervening "repairs" and "across"
531 other forms of maintenance.

Example 118 . **Machine Requirements. Road-pricing System Robustness**: The road-pricing computing and communications system shall retain its (i) performance and (ii) dependability, that is, (ii.1) accessibility, (ii,2) availability, (ii,3) reliability, and (ii,4) safety requirements in the presence of maintenance.

7.4.4 Maintenance Requirements

532

TO BE TYPED

Delineation and Facets of Maintenance Requirements

Definition 59 . ***Maintenance Requirements:*** *By* maintenance requirements *we understand a combination of requirements with respect to: (i)* adaptive maintenance*, (iii)* corrective maintenance*, (ii)* perfective maintenance*, (iv)* preventive maintenance *and (v)* extensional maintenance ■

533

Maintenance of building, mechanical, electrotechnical and electronic artifacts — i.e., of artifacts based on the natural sciences — is based both on documents and on the presence of the physical artifacts. Maintenance of software is based just on software, that is, on all the documents (including tests) entailed by software — see Definition 71 on Page 179.

Adaptive Maintenance

534

Definition 60 . ***Adaptive Maintenance:*** *By* adaptive maintenance *we understand such maintenance that changes a part of that software so as to also, or instead, fit to some other software, or some other hardware equipment (i.e., other software or hardware which provides new, respectively replacement, functions)* ■

535

Example 119 . Machine Requirements. Road-pricing System Adaptive Maintenance: Two forms of adaptive maintenance occur in connection with the road-pricing computing and communication system: (i) adaptive maintenance of vehicle and toll-gate sensors and actuators, and (ii) adaptive maintenance of the "interfacing" software, that is, the vehicle software as prescribed by Item 374 on Page 165, the toll-gate software as prescribed by Item 377 on Page 165, and the calculator software as prescribed by Item 378 on Page 166. Adaptive maintenance of vehicle and toll-gate sensors and actuators occurs when existing sensors or actuators are replaced due to failure. Adaptive maintenance of interfacing software is required when existing sensors or actuators have been replaced and their characteristics are different from those of the replaced equipment, hence requires modifications of interfacing software

536

Corrective Maintenance

537

Definition 61 . *Corrective Maintenance:* *By corrective maintenance we understand such maintenance which corrects a software error* ■

538

Example 120 . Machine Requirements. Road-pricing System Corrective Maintenance: Corrective maintenance of the road-pricing computing and communications system is required in two "spheres": (i) when system, that is, toll-gate and vehicles sensors or actuators fail, and (i) when, despite all verification efforts, the interfacing, that is, the vehicle, the gate, or the calculator software fails. In the former case, (i), the failing sensor or actuator is replaced possibly implying adaptive maintenance. In the latter case, (ii), the failing software is analysed in order to locate the erroneous code, whereupon that code is replaced by such code that can lead to a verification of the full system

539

Perfective Maintenance

540

Definition 62 . *Perfective Maintenance:* *By perfective maintenance we understand such maintenance which helps improve (i.e., lower) the need for hardware storage, time and (hard) equipment* ■

541

Example 121 . Machine Requirements. Road-pricing System Perfective Maintenance: We focus on perfective maintenance of vehicle, toll-gate and calculator software. We focus, in particular, on (i) the reaction time in connection with response to external stimuli for the gate software (i.1) the timed local position, Item 374a on Page 165, of vehicles; (i.2) the attr_enter_ch[gi] event from a toll-gate's in coming sensor, Item 377a on Page 165; (i.3) the timed vehicle identity for a attr_TIVI_ch[gi] event form a toll-gate sensor, Item 377b on Page 165; and (i.4) the attr_leave_ch[gi] event from a toll-gate's out going sensor, Item 377d on Page 165; (ii) the reaction time, of the calculator, Item 378 on Page 166, to incoming, alternating, communications from either vehicles, Item 378a on Page 166, or gates, Item 378b on Page 166. and (iii) the calculation time of the calculator for billing, cf. Item 380e on Page 166.

542

543

Preventive Maintenance

544

Definition 63 . *Preventive Maintenance:* *By preventive maintenance we understand such maintenance which helps detect, i.e., forestall, future occurrence of software or hardware failures* ■

Example 122 . Machine Requirements. Road-pricing System Preventive Maintenance:

TO BE WRITTEN

Extensional Maintenance

545

Definition 64 . *Extensional Maintenance:* *By extensional maintenance we understand such maintenance which adds new functionalities to the software, i.e., which implements additional requirements* ■

Example 123 . Machine Requirements. Road-pricing System Extensional Maintenance:

TO BE WRITTEN

7.4.5 Platform Requirements

546

TO BE WRITTEN

Delineation and Facets of Platform Requirements

Definition 65 . ***Platform:*** *By a [computing] platform is here understood a combination of hardware and systems software so equipped as to be able to develop and execute software, in one form or another* ■

547 What the "in one form or another" is transpires from the next characterisation.

Definition 66 . ***Platform Requirements:*** *By platform requirements we mean a combination of the following: (i) development platform requirements, (ii) execution platform requirements, (iii) maintenance platform requirements and (iv) demonstration platform requirements* ■

Development Platform

548

Definition 67 . ***Development Platform Requirements:*** *By development platform requirements we shall understand such machine requirements which detail the specific software and hardware for the platform on which the software is to be developed* ■

Execution Platform

549

Definition 68 . ***Execution Platform Requirements:*** *By execution platform requirements we shall understand such machine requirements which detail the specific (other) software and hardware for the platform on which the software is to be executed* ■

Maintenance Platform

550

Definition 69 . ***Maintenance Platform Requirements:*** *By maintenance platform requirements we shall understand such machine requirements which detail the specific (other) software and hardware for the platform on which the software is to be maintained* ■

Demonstration Platform

551

Definition 70 . ***Demonstration Platform Requirements:*** *By demonstration platform requirements we shall understand such machine requirements which detail the specific (other) software and hardware for the platform on which the software is to be demonstrated to the customer — say for acceptance tests, or for management demos, or for user training* ■

• • •

552

Example 124 . **Machine Requirements. Road-pricing System Platform Requirements**: The platform requirements are the following:

the **development platform** to be typed

the **execution platform** to be typed

the **maintenance platform** to be typed

and the **demonstration platform** to be typed.

7.4.6 Documentation Requirements

553

Definition 71 . ***Software:*** *By* ***software*** *we shall understand (i) not only* ***code*** *that may be the basis for executions by a computer, (ii) but also its full* ***development documentation****: (ii.1) the stages and steps of* ***application domain description,*** *(ii.2) the stages and steps of* ***requirements prescription,*** *and (ii.3) the stages and steps of* ***software design*** *prior to code, with all of the above including all validation and verification (incl., test) documents. (iii) In addition, as part of our wider concept of* 554
software, we also include a comprehensive collection of supporting documents: (iii.1) ***training manuals,*** *(iii.2)* ***installation manuals,*** *(iii.3)* ***user manuals,*** *(iii.4)* ***maintenance manuals,*** *and (iii.5–6)* ***development and maintenance logbooks.*** ■

555

Definition 72 . ***Documentation Requirements:*** *By documentation requirements we mean requirements of any of the software documents that together make up software and hardware*[9] ■

Example 125 . **Machine Requirements — Documentation**:

TO BE WRITTEN

7.4.7 Discussion

556

TO BE TYPED

[9] — we omit a definition of what we mean by hardware such as the one we gave for software, cf. Definition 71.

Part III

Conclusion

Chapter 8

Discussion of Research Topics 562

Chapter Status

This chapter is incomplete.

There are a number of research topics: some relate to domain analysis & description, cf. Chapter 1 and some of these are listed in Sect. 8.1, other relate to requirements engineering, cf. Chapter 7 and some of these are listed in Sect. 8.2.

8.1 Domain Science & Engineering Topics 563

The TripTych approach to software development, based on an initial, serious phase of domain engineering, a new phase of software engineering, for which we claim to now have laid a solid foundation for domain engineering — opens up for a variety of issues that need further study. The entries in this section are not ordered according to any specific principle.

8.1.1 Analysis & Description Calculi for Other Domains 564

The analysis and description calculus of this paper appears suitable for manifest domains. For other domains other calculi appears necessary. There is the introvert, composite domain of systems software: operating systems, compilers, database management systems, Internet-related software, etcetera. The classical computer science and software engineering disciplines related to these components of systems software appears to have provided the necessary analysis and description "calculi." There is the domain of financial 565
systems software accounting & bookkeeping, banking systems, insurance, financial instruments handling (stocks, etc.), etcetera. We refer to Sect. 9.1.2 on Page 186 [Item 8]. Etcetera. For each domain characterisable by a distinct set of analysis & description calculus prompts such calculi must be identified. 566

It seems straightforward: to base a method for analysing & describing a category of domains on the idea of prompts like those developed in this paper.

8.1.2 On Domain Description Languages 567

We have in this paper expressed the domain descriptions in the RAISE [86] specification language RSL [85]. With what is thought of as basically inessential, editorial changes, one can reformulate these domain description texts in either of Alloy [100] or The B-Method [1] or VDM [48, 49, 77] or Z [157]. One 568

could also express domain descriptions algebraically, for example in CafeOBJ [81, 68, 80, 56]. The analysis and the description prompts remain the same. The description prompts now lead to CafeOBJ texts.

We did not go into much detail with respect to perdurants, let alone behaviours. For all the very many domain descriptions, covered elsewhere, RSL (with its CSP sub-language) suffices. But there are cases where we have conjoined our RSL domain descriptions with descriptions in Petri Nets [132] or MSC [99] (Message Sequence Charts) or StateCharts [92]. Since this paper only focused on endurants there was no need, it appears, to get involved in temporal issues. When that becomes necessary, in a study or description of perdurants, then we either deploy DC: The Duration Calculus [160] or TLA+: Temporal Logic of Actions [110].

8.1.3 Ontology Relations

A more exact understanding of the relations between the "classical" AI/information science/ontology view of domains [13, 14, 105], and the algorithmic view of domains, as presented in the current paper, seems required. The almost disparate jargon of the two "camps" seems, however, to be a hindrance.

8.1.4 Analysis of Perdurants

A study of perdurants, Sect. 1.3, as detailed as that of our study of endurants, ought be carried out. One difficulty, as we see it, is the choice of formalisms: whereas the basic formalisms for the expression of endurants and their qualities was type theory and simple functions and predicates, there is no such simple set of formal constructs that can "carry" the expression of behaviours.Besides the textual CSP, [97], there is graphic notations of Petri Nets, [132], Message Sequence Charts, [99], State-charts, [92], and others.

8.1.5 Commensurate Discrete and Continuous Models

Section 1.3.6 on Page 56 hinted at co-extensive descriptions of discrete and continuous behaviours, the former in, for example, RSL, the latter in, typically, the calculus mathematics of partial different equations (PDEs). The problem that arises in this situation is the following: there will be, say variable identifiers, e.g., x, y, ..., z which in the RSL formalisation has one set of meanings, but which in the PDE "formalisation" has another set of meanings. Current formal specification languages[1] do not cope with continuity. Some research is going on. But to substantially cover, for example, the proper description of laminar and turbulent flows in networks (e.g., pipelines, Example 61 on Page 56) requires more substantial results.

8.1.6 Interplay between Parts, Materials and Components

Examples 49 on Page 46, 50 on Page 47, 51 on Page 48 and 61 on Page 56 revealed but a small fraction of the problems that may arise in connection with modeling the interplay between parts and materials. Subject to proper formal specification language and, for example PDE specification, we may expect more interesting laws, as for example those of Examples 50 on Page 47, 51 on Page 48, and even proof of these as if they were theorems. Formal specifications have focused on verifying properties of requirements and software designs. With co-extensive (i.e., commensurate) formal specifications of both discrete and continuous behaviours we may expect formal specifications to also serve as bases for predictions.

8.1.7 Dynamics

There is a serious limitation in what can be modeled with the present approach. Although we can model the dynamic introduction of new atomic or removal of existing parts, when members of a composite set of such parts, we cannot model the dynamic introduction or removal of the processes corresponding to such

[1] Alloy [100],Event B [1],RSL [85], VDM-SL [48, 49, 77], Z [157], etc.

parts. Also we have not shown how to model global time. And, although we can model spatial positions, we have not shown how to model spatial locations. These deliberate omissions are due to the facts that the description language, RSL, cannot model continuity and that it cannot provide for arbitrary models of time [150]. Here is an area worth studying.

8.1.8 Precise Descriptions of Manifest Domains

The focus on the principles, techniques and tools of domain analysis & description has been such domains in which humans play an active rôle. Formal descriptions of domains may serve to prove properties of domains, in other words, to understand better these domains, and to validate requirements derived from such domain descriptions, and thereby to ensure that software derived from such requirements is not only correct, but also meet users expectations. Improved understanding of man-made domains — without necessarily leading to new software — may serve to improve the "business processes" of these domains, make them more palatable for the human actors, make them more efficient wrt. resource-usage. Descriptions of domains are descriptions of the syntax and semantics of the technical languages used in speaking about and in the domain. The domain analysis required for the design of programming languages is based on computability: mathematical logic and recursive function theory. The domain analysis required for "real-world" domains is not based on computability: that "world" is not computable. Requirements engineering based on domain descriptions is based on deriving computable subsets of refined domain descriptions. The classical theory and practice of programming language semantics and compiler development [15] and [23, Part VII (Chapters 16–19)] can now be further developed into a theory and practice for deriving general software from formal domain descriptions [26].

Descriptions of domains are descriptions of the syntax and semantics of the technical languages used in speaking about and in the domain. The domain analysis required for the design of programming languages is based on computability: mathematical logic and recursive function theory. The domain analysis required for "real-world" domains is not based on computability: that "world" is not computable. Requirements engineering based on domain descriptions is based on deriving computable subsets of refined domain descriptions. The classical theory and practice of programming language semantics and compiler development [15] and [23, Part VII (Chapters 16–19)] can now be further developed into a theory and practice for deriving general software from formal domain descriptions [26].

Physicists study 'Mother Nature', the world without us. Domain scientists study man-made part and material based universes with which we interact — the world within and without us. Classical engineering builds on laws of physics to design and construct buildings, chemical compounds, machines and electrical and electronic products. So far software engineers have not expressed software requirements on any precise description of the basis domain. This paper strongly suggests such a possibility. Regardless: it is interesting to also formally describe domains; and, as shown, it can be done.

8.1.9 Towards Mathematical Models of Domain Analysis & Description

There are two aspects to a precise description of the ***domain analysis prompt***s and ***domain description prompt***s. There is that of describing the individual prompts as if they were "machine instructions" for an albeit strange machine; and there is that of describing the interplay between prompts: the sequencing of ***domain description prompt***s as determined by the outcome of the ***domain analysis prompt***s. We have described and formalised the latter in [43, Processes]; and we are in the midst of describing and formalising the former in [35, Prompts].

8.1.10 Laws of Descriptions: A Calculus of Prompts

Laws of descriptions deal with the order and results of applying the domain analysis and description prompts. Some laws are covered in [33]. It is expected that establishing formal models of the prompts, for example as outlined in [35, 43], will help identify such laws. The various description prompts apply

to parts (etc.) of specified sorts (etc.) and to a "hidden state". The "hidden state" has two major elements: the domain and the evolving description texts. An "execution" of a prompt potentially changes that "hidden state". Let P, PA and PB be composite part sorts where PA and PB are derived from P. Let $\mathfrak{R}_i$, $\mathfrak{R}_j$, etc., be suitable functions which rename sort, type and attribute names. In a proper prompt calculus we would expect `observe_part_sorts_PA;observe_part_sorts_PB`, when "executed" by one and the same domain engineer, to yield the same "hidden state" as `observe_part_sorts_PB`;$\mathfrak{R}_i$;`observe_part_sorts_PA`;$\mathfrak{R}_j$. Also one would expect `observe_part_sorts_PA`;$\mathfrak{R}_i$;`observe_part_sorts_PA`;$\mathfrak{R}_j$. to yield the same state as just `observe_part_sorts_PA` given suitable renaming functions.

Well ? or does one really ?

There are some assumptions that are made here. One pair of assumptions is that the domain is fixed and to one observer. yields the same analysis and description results no matter in which order prompts are "executed". Another assumption is that the domain engineer does not get wiser as analysis and description progresses. If, as one can very well expect, the domain engineer does get wiser, then former results may be discarded and either replaced by newer analysis and descriptions or prompts repeated. In such cases these laws do not hold.

8.1.11 Domains and Galois Connections

Section 1.1.8 on Page 23 very briefly mentioned that formal concepts form Galois Connections. In the seminal [83] a careful study is made of this fact and beautiful examples show the implications for domains. It seems that our examples have all been too simple. They do not easily lead on to the "discovery" of "new" domain concepts from appropriate concept lattices. We refer to [47, Section 9]. Further study need be done.

8.1.12 Laws of Domain Description Prompts

Typically `observe_part_sorts` applies to a composite part, p:P, and yield descriptions of one or more part sorts: p_1:P_1,p_2:P_2,...,p_m:P_m. Let p_i:P_i,p_j:P_j,...,p_k:P_k (of these) be composite. Now `observe_part_sorts`(p_i) and `observe_part_sorts`(p_j), etc., can be applied and yield texts $text_i$, respectively $text_j$. A law of domain description prompts now expresses that the order in which the two or more observers is applied is immaterial, that is, they commute. In [33] we made an early exploration of such laws of domain description prompts. More work, see also below, need be done.

8.1.13 Domain Theories:

An ultimate goal of domain science & engineering is to prove properties of domains. Well, maybe not properties of domains, but then at least properties of domain descriptions. If one can be convinced that a posited domain description indeed is a faithful description of a domain, then proofs of properties of the domain description are proofs of properties of that domain. Ultimately domain science & engineering must embrace such studies of *laws of domains*. Here is a fertile ground for zillions of Master and PhD theses !

Example 126 . **A Law of Train Traffic at Stations**:Let a transport net, n:N, be that of a railroad system. Hubs are train stations. Links are rail lines between stations. Let a train timetable record train arrivals and train departures from stations. And let such a timetable be modulo some time interval, say typically 24 hours. Now let us (idealistically) assume that actual trains arrive at and depart from train stations according the train timetable and that the train traffic includes all and only such trains as are listed in the train timetable. Now a law of train traffic expresses *"Over the modulo time interval of a train timetable it is the case that the number of trains arriving at a station minus the number of trains ending their journey at that station plus the number of trains starting their journey at that station equals number of trains departing from that station."* ■

8.1.14 External Attributes 597

More study is needed in order to clarify the relations between the various external attributes and control theory.

8.2 Requirements Topics 598

8.2.1 Domain Requirements Methodology

Further principles, techniques and tools for the projection, instantiation, determination, extension and fitting operations.

8.2.2 Domain Requirements Operator Theory 599

A model of the domain to domain-to-requirements operators: projection, instantiation, determination, extension and fitting. (Sect. 7.2).

8.2.3 Methodology for Interface Requirements 600

Sect. 7.3 did not go into sufficient detail as to method principles, techniques and tools.

Chapter 9

Bibliography 601

9.1 Bibliographical Notes

Web page **www.imm.dtu.dk/~dibj/domains/** lists the published papers and reports mentioned in the next two subsections.

9.1.1 Published Papers

I have thought about domain engineering for more than 20 years. But serious, focused writing only started to appear since [24, Part IV] — with [20, 18] being exceptions: [25] suggests a number of domain science and engineering research topics; [30] covers the concept of domain facets summarised in Chap. 2; [47] explores compositionality and Galois connections. [26, 46] show how to systematically, but, of course, not automatically, "derive" requirements prescriptions from domain descriptions; [32] takes the triptych software development as a basis for outlining principles for believable software management; [29, 39] presents a model for Stanisław Leśniewski's [57] concept of mereology; [31, 33] present an extensive example and is otherwise a precursor for the present paper; [34] presents, based on the `TripTych` view of software development as ideally proceeding from domain description via requirements prescription to software design, concepts such as software demos and simulators; [37] analyses the `TripTych`, especially its domain engineering approach, with respect to Maslow's [1] and Peterson's and Seligman's [2] notions of humanity: how can computing relate to notions of humanity; the first part of [40] is a precursor for the present paper with its second part presenting a first formal model of the elicitation process of analysis and description based on the prompts more definitively presented in the current paper; and [41] focus on domain safety criticality.

The present paper basically replaces the domain analysis and description section of all of the above reference — including [24, Part IV].

9.1.2 Reports 602

We list a number of reports all of which document descriptions of domains. These descriptions were carried out in order to research and develop the domain analysis and description concepts now summarised in the present paper. These reports ought now be revised, some slightly, others less so, so as to follow all of the prescriptions of the current paper. Except where a URL is given in full, please prefix the web reference with: `http://www2.compute.dtu.dk/~dibj/`.

[1] *Theory of Human Motivation.* Psychological Review 50(4) (1943):370-96; and *Motivation and Personality,* Third Edition, Harper and Row Publishers, 1954.

[2] *Character strengths and virtues: A handbook and classification.* Oxford University Press, 2004

1 *A Railway Systems Domain:* `http://euler.fd.cvut.cz/railwaydomain/` (2003)

2 *Models of IT Security. Security Rules & Regulations:* `it-security.pdf` (2006)

3 *A Container Line Industry Domain:* `container-paper.pdf` (2007)

4 *The "Market": Consumers, Retailers, Wholesalers, Producers:* `themarket.pdf` (2007)

5 *What is Logistics ?:* `logistics.pdf` (2009)

6 *A Domain Model of Oil Pipelines:* `pipeline.pdf` (2009)

7 *Transport Systems:* `comet/comet1.pdf` (2010)

8 *The Tokyo Stock Exchange:* `todai/tse-1.pdf` and `todai/tse-2.pdf` (2010)

9 *On Development of Web-based Software. A Divertimento:* `wfdftp.pdf` (2010)

10 *Documents (incomplete draft):* `doc-p.pdf` (2013)

9.2 References

603

[1] J.-R. Abrial. The B Book: Assigning Programs to Meanings *and* Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge, England, 1996 and 2009.

[2] K. R. Apt. *Principles of Constraint Programming*. Cambridge University Press, August 2003. ISBN 0521825830.

[3] K. Araki et al., editors. *IFM 1999–2013: Integrated Formal Methods*, LNCS Vols. 1945, 2335, 2999, 3771, 4591, 5423, 6496, 7321 and 7940. Springer, 1999–2013.

[4] Y. Arimoto and D. Bjørner. Hospital Healthcare: A Domain Analysis and a License Language. Technical note, JAIST, School of Information Science, 1-1, Asahidai, Tatsunokuchi, Nomi, Ishikawa, Japan 923-1292, Summer 2006.

[5] R. Audi. *The Cambridge Dictionary of Philosophy*. Cambridge University Press, The Pitt Building, Trumpington Street, Cambridge CB2 1RP, England, 1995.

[6] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. . Cambrige University Press, January 2003. 570 pages, 14 tables, 53 figures; ISBN: 0521781760.

[7] F. Baader, I. Horrocks, and U. Sattler. Description Logics as Ontology Languages for the Semantic Web. In D. Hutter and W. Stephan, editors, *Mechanizing Mathematical Reasoning*, pages 228–248. Springer, Heidelberg, 2005.

[8] C. Bachman. Data structure diagrams. *Data Base, Journal of ACM SIGBDP*, 1(2), 1969.

[9] A. Badiou. *Being and Event*. Continuum, 2005. (Lêtre et l'événements, Edition du Seuil, 1988).

[10] V. Benjamins and D. Fensel. The Ontological Engineering Initiative (KA)2. Internet publication + Formal Ontology in Information Systems, University of Amsterdam, SWI, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands and University of Karlsruhe, AIFB, 76128 Karlsruhe, Germany, 1998. http://www.aifb.uni-karlsruhe.de/WBS/broker/KA2.htm.

[11] W. Bevier, W. H. Jr., J. S. Moore, and W. Young. An approach to system verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989. Special Issue on System Verification.

[12] G. Birkhoff. *Lattice Theory*. American Mathematical Society, Providence, R.I., 3 edition, 1967.

[13] T. Bittner, M. Donnelly, and B. Smith. Endurants and Perdurants in Directly Depicting Ontologies. *AI Communications*, 17(4):247–258, December 2004. IOS Press, in [133].

[14] T. Bittner, M. Donnelly, and B. Smith. Individuals, Universals, Collections: On the Foundational Relations of Ontology. In A. Varzi and L. Vieu, editors, *Formal Ontology in Information Systems*, Proceedings of the Third International Conference, pages 37–48. IOS Press, 2004.

[15] D. Bjørner. Programming Languages: Formal Development of Interpreters and Compilers. In *International Computing Symposium 77 (eds. E. Morlet and D. Ribbens)*, pages 1–21. European ACM, North-Holland Publ.Co., Amsterdam, 1977.

[16] D. Bjørner. A ProCoS Project Description. *Published in two slightly different versions: (1) EATCS Bulletin, October 1989, (2) (Ed. Ivan Plander:) Proceedings: Intl. Conf. on AI & Robotics, Strebske Pleso, Slovakia, Nov. 5-9, 1989, North-Holland, Publ.*, Dept. of Computer Science, Technical University of Denmark, October 1989.

[17] D. Bjørner. Trustworthy Computing Systems: The ProCoS Experience. In *14'th ICSE: Intl. Conf. on Software Eng., Melbourne, Australia*, pages 15–34. ACM Press, May 11–15 1992.

[18] D. Bjørner. Michael Jackson's Problem Frames: Domains, Requirements and Design. In L. ShaoYang and M. Hinchley, editors, *ICFEM'97: International Conference on Formal Engineering Methods*, Los Alamitos, November 12–14 1997. IEEE Computer Society. Final Version.

[19] D. Bjørner. Domain Models of "The Market" — in Preparation for E–Transaction Systems. In *Practical Foundations of Business and System Specifications (Eds.: Haim Kilov and Ken Baclawski)*, The Netherlands, December 2002. Kluwer Academic Press. Final draft version.

[20] D. Bjørner. Domain Engineering: A "Radical Innovation" for Systems and Software Engineering ? In *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, Heidelberg, October 7–11 2003. Springer–Verlag. The Zohar Manna International Conference, Taormina, Sicily 29 June – 4 July 2003. Final draft version.

[21] D. Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. .

[22] D. Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling; Vol. 2: Specification of Systems and Languages; ol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.

[23] D. 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.

[24] D. Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.

[25] D. Bjørner. Domain Theory: Practice and Theories, Discussion of Possible Research Topics. In *ICTAC'2007*, volume 4701 of *Lecture Notes in Computer Science (eds. J.C.P. Woodcock et al.)*, pages 1–17, Heidelberg, September 2007. Springer.

[26] D. Bjørner. From Domains to Requirements. In *Montanari Festschrift*, volume 5065 of *Lecture Notes in Computer Science (eds. Pierpaolo Degano, Rocco De Nicola and José Meseguer)*, pages 1–30, Heidelberg, May 2008. Springer.

[27] D. Bjørner. An Emerging Domain Science – A Rôle for Stanisław Leśniewski's Mereology and Bertrand Russell's Philosophy of Logical Atomism. *Higher-order and Symbolic Computation*, 2009.

[28] D. Bjørner. *Domain Engineering: Technology Management, Research and Engineering*. Research Monograph (#4); JAIST Press, 1-1, Asahidai, Nomi, Ishikawa 923-1292 Japan, This Research Monograph contains the following main chapters:

1 *On Domains and On Domain Engineering – Prerequisites for Trustworthy Software – A Necessity for Believable Management*, pages 3–38.
2 *Possible Collaborative Domain Projects – A Management Brief*, pages 39–56.
3 *The Rôle of Domain Engineering in Software Development*, pages 57–72.
4 *Verified Software for Ubiquitous Computing – A VSTTE Ubiquitous Computing Project Proposal*, pages 73–106.
5 *The Triptych Process Model – Process Assessment and Improvement*, pages 107–138.
6 *Domains and Problem Frames – The Triptych Dogma and M.A.Jackson's PF Paradigm*, pages 139–175.
7 *Documents – A Rough Sketch Domain Analysis*, pages 179–200.
8 *Public Government – A Rough Sketch Domain Analysis*, pages 201–222.
9 *Towards a Model of IT Security — – The ISO Information Security Code of Practice – An Incomplete Rough Sketch Analysis*, pages 223–282.
10 *Towards a Family of Script Languages – – Licenses and Contracts – An Incomplete Sketch*, pages 283–328.

2009.

[29] D. Bjørner. On Mereologies in Computing Science. In *Festschrift: Reflections on the Work of C.A.R. Hoare*, History of Computing (eds. Cliff B. Jones, A.W. Roscoe and Kenneth R. Wood), pages 47–70, London, UK, 2009. Springer.

[30] D. Bjørner. Domain Engineering. In P. Boca and J. Bowen, editors, *Formal Methods: State of the Art and New Directions*, Eds. Paul Boca and Jonathan Bowen, pages 1–42, London, UK, 2010. Springer.

[31] D. Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics, Part I of II: The Engineering Part*. *Kibernetika i sistemny analiz*, (4):100–116, May 2010.

[32] D. Bjørner. Believable Software Management. *Encyclopedia of Software Engineering*, 1(1):1–32, 2011.

[33] D. Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics Part II of II: The Science Part*. *Kibernetika i sistemny analiz*, (2):100–120, May 2011.

[34] D. 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.

[35] D. Bjørner. Domain Analysis: A Model of Prompts (paper[3], slides[4]). Research Report 2013-6, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Fall 2013.

[36] D. Bjørner. Domain Analysis (paper[5] slides[6]). Research Report 2013-1, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, April 2013.

[37] D. Bjørner. *Domain Science and Engineering as a Foundation for Computation for Humanity*, chapter 7, pages 159–177. Computational Analysis, Synthesis, and Design of Dynamic Systems. CRC [Francis & Taylor], 2013. (eds.: Justyna Zander and Pieter J. Mosterman).

[38] D. Bjørner. Pipelines – a Domain Description[7]. Experimental Research Report 2013-2, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013.

[39] D. Bjørner. *A Rôle for Mereology in Domain Science and Engineering*. Synthese Library (eds. Claudio Calosi and Pierluigi Graziani). Springer, Amsterdam, The Netherlands, October 2014.

[40] D. Bjørner. Domain Analysis: Endurants – An Analysis & Description Process Model. In S. Iida, J. Meseguer, and K. Ogata, editors, *Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi*. Springer, May 2014.

[41] D. Bjørner. Domain Engineering – A Basis for Safety Critical Software. Invited Keynote, ASSC2014: Australian System Safety Conference, Melbourne, 26–28 May, December 2014.

[42] D. Bjørner. Manifest Domains: Analysis & Description. Research Report, 2014. Part of a series of research reports: [44, 45], Being submitted.

[43] D. Bjørner. Domain Analysis: Endurants – a Consolidated Model of Prompts. Research Report, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, May 2015.

[44] D. Bjørner. Domain Analysis & Description: Models of Processes and Prompts. Research Report, To be completed early 2014. Part of a series of research reports: [42, 45].

[45] D. Bjørner. From Domains to Requirements – A Different View of Requirements Engineering. Research Report, To be completed mid 2015. Part of a series of research reports: [42, 44].

[46] D. Bjørner. The Role of Domain Engineering in Software Development. Why Current Requirements Engineering Seems Flawed! In *Perspectives of Systems Informatics*, volume 5947 of *Lecture Notes in Computer Science*, pages 2–34, Heidelberg, Wednesday, January 27, 2010. Springer.

[47] D. Bjørner and A. Eir. Compositionality: Ontology and Mereology of Domains. Some Clarifying Observations in the Context of Software Engineering in July 2008, eds. Martin Steffen,

[3] http://www.imm.dtu.dk/~dibj/da-mod-p.pdf
[4] http://www.imm.dtu.dk/~dibj/da-mod-s.pdf
[5] http://www.imm.dtu.dk/~dibj/da-p.pdf
[6] http://www.imm.dtu.dk/~dibj/da-s.pdf
[7] http://www.imm.dtu.dk/~dibj/pipe-p.pdf

Dennis Dams and Ulrich Hannemann. In *Festschrift for Prof. Willem Paul de Roever Concurrency, Compositionality, and Correctness*, volume 5930 of *Lecture Notes in Computer Science*, pages 22–59, Heidelberg, July 2010. Springer.

[48] D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.

[49] D. Bjørner and C. B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.

[50] D. Bjørner and J. F. Nilsson. Algorithmic & Knowledge Based Methods — Do they "Unify" ? In *International Conference on Fifth Generation Computer Systems: FGCS'92*, pages 191–198. ICOT, June 1–5 1992.

[51] D. Bjørner, A. Yasuhito, C. Xiaoyi, and X. Jianwen. A Family of License Languages. Technical report, JAIST, Graduate School of Information Science, 1-1, Asahidai, Tatsunokuchi, Nomi, Ishikawa, Japan 923-1292, August 2006.

[52] W. D. Blizard. A Formal Theory of Objects, Space and Time. *The Journal of Symbolic Logic*, 55(1):74–89, March 1990.

[53] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[54] J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. ACM Press/Addison-Wesley, New York, NY, 2000.

[55] F. Buschmann, K. Henney, and D. Schmidt. *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*. John Wiley & Sons Ltd., England, 2007.

[56] CafeOBJ. http://cafeobj.org/, 2014.

[57] R. Casati and A. Varzi. *Parts and Places: the structures of spatial representation*. MIT Press, 1999.

[58] R. Casati and A. Varzi. Events. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2010 edition, 2010.

[59] R. Casati and A. C. Varzi, editors. *Events*. Ashgate Publishing Group – Dartmouth Publishing Co. Ltd., Wey Court East, Union Road, Farnham, Surrey, GU9 7PT, United Kingdom, 23 March 1996.

[60] P. P. Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Trans. Database Syst*, 1(1):9–36, 1976.

[61] X. Chen and D. Bjørner. Public Government: A Domain Analysis and a License Language. Technical note, JAIST, School of Information Science, 1-1, Asahidai, Tatsunokuchi, Nomi, Ishikawa, Japan 923-1292, Summer 2006.

[62] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[63] CoFI (The Common Framework Initiative). CASL *Reference Manual*, volume 2960 of *Lecture Notes in Computer Science (IFIP Series)*. Springer–Verlag, 2004.

[64] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, 2000.

[65] W. Damm and D. Harel. LSCs: Breathing life into Message Sequence Charts. *Formal Methods in System Design*, 19:45–80, 2001. Early version appeared as Weizmann Institute Tech. Report CS98-09, April 1998. An abridged version appeared in *Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-based Distributed Systems* (FMOODS'99), Kluwer, 1999, pp. 293–312.

[66] D. Davidson. *Essays on Actions and Events*. Oxford University Press, 1980.

[67] J. de Bakker. *Control Flow Semantics*. The MIT Press, Cambridge, Mass., USA, 1995.

[68] R. Diaconescu and K. Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. AMAST Series in Computing, 6. World Scientific, Singapore, 1998.

[69] M. Dorfman and R. H. Thayer, editors. *Software Requirements Engineering*. IEEE Computer Society Press, 1997.

[70] F. Dretske. Can Events Move? *Mind*, 76(479-492), 1967. reprinted in [59], pp. 415-428.

[71] ESA. Global Navigation Satellite Systems. Web, European Space Agency. There are several global navigation satellite systems (http://en.wikipedia.org/wiki/Satellite_navigation) either in operation or being developed: (1.) the US developed and operated `GPS` (`NAVSTAR`) system, http://en.wikipedia.org/wiki/Global_Positioning_System; (2.) the EU developed and (to be) operated `Galileo` system, http://en.wikipedia.org/wiki/Galileo_positioning_system; (3.) the Russian developed and (to be) operated `GLONASS`, http://en.wikipedia.org/wiki/-GLONASS; and (4.) the Chinese `Compass Navigation System`, http://en.wikipedia.-org/wiki/Compass_navigation_system.

[72] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts 02142, 1996. 2nd printing.

[73] R. Falbo, G. Guizzardi, and K. Duarte. An Ontological Approach to Domain Engineering. In *Software Engineering and Knowledge Engineering*, Proceedings of the 14th international conference SEKE'02, pages 351–358, Ischia, Italy, July 15-19 2002. ACM.

[74] D. J. Farmer. *Being in time: The nature of time in light of McTaggart's paradox*. University Press of America, Lanham, Maryland, 1990. 223 pages.

[75] E. A. Feigenbaum and P. McCorduck. *The fifth generation*. Addison-Wesley, Reading, MA, USA, 1st ed. edition, 1983.

[76] W. Feijen, A. van Gasteren, D. Gries, and J. Misra, editors. *Beauty is Our Business*, Texts and Monographs in Computer Science, New York, NY, USA, 1990. Springer. A Birthday Salute to Edsger W. Dijkstra.

[77] J. Fitzgerald and P. G. 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.

[78] M. Fowler. *Domain Specific Languages*. Signature Series. Addison Wesley, October 20120.

[79] C. A. Furia, D. Mandrioli, A. Morzenti, and M. Rossi. *Modeling Time in Computing*. Monographs in Theoretical Computer Science. Springer, 2012.

[80] K. Futatsugi, D. Găină, and K. Ogata. Principles of proof scores in CafeOBJ. *Theor. Comput. Sci.*, 464:90–112, 2012.

[81] K. Futatsugi and A. Nakagawa. An overview of CAFE specification environment - an algebraic approach for creating, verifying, and maintaining formal specifications over networks. In *Proc. of 1st International Conference on Formal Engineering Methods (ICFEM '97), November 12-14, 1997, Hiroshima, JAPAN*, pages 170–182. IEEE, 1997.

[82] K. Futatsugi, A. Nakagawa, and T. Tamai, editors. *CAFE: An Industrial–Strength Algebraic Formal Method*, Sara Burgerhartstraat 25, P.O. Box 211, NL–1000 AE Amsterdam, The Netherlands, 2000. Elsevier. Proceedings from an April 1998 Symposium, Numazu, Japan.

[83] B. Ganter and R. Wille. *Formal Concept Analysis — Mathematical Foundations*. Springer-Verlag, January 1999. ISBN: 3540627715, 300 pages, Amazon price: US $ 44.95.

[84] B. Ganter and R. Wille. *Formal Concept Analysis — Mathematical Foundations*. Springer-Verlag, January 1999. ISBN: 3540627715, 300 pages, Amazon price: US $ 44.95.

[85] C. W. George, P. Haff, K. Havelund, A. E. Haxthausen, R. Milne, C. B. Nielsen, S. Prehn, and K. R. Wagner. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.

[86] C. W. George, A. E. Haxthausen, S. Hughes, R. Milne, S. Prehn, and J. S. Pedersen. *The RAISE Development Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.

[87] D. I. Good and W. D. Young. Mathematical Methods for Digital Systems Development. In *VDM '91: Formal Software Development Methods*, pages 406–430. Springer-Verlag, October 1991. Volume 2.

[88] C. A. Gunter, E. L. Gunter, M. A. Jackson, and P. Zave. A Reference Model for Requirements and Specifications. *IEEE Software*, 17(3):37–43, May–June 2000.

[89] C. A. Gunter, S. T. Weeks, and A. K. Wright. Models and Languages for Digtial Rights. In *Proc. of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)*, pages 4034–4038, Maui, Hawaii, USA, January 2001. IEEE Computer Society Press.

[90] C. Gunther. *Semantics of Programming Languages*. The MIT Press, Cambridge, Mass., USA, 1992.

[91] P. Hacker. Events and Objects in Space and Time. *Mind*, 91:1–19, 1982. reprinted in [59], pp. 429-447.

[92] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[93] D. Harel and R. Marelly. *Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.

[94] M. Harsu. A Survey on Domain Engineering. Review, Institute of Software Systems, Tampere University of Technology, Finland, December 2002.

[95] D. Haywood. *Domain-Driven Design Using Naked Objects*. The Pragmatic Bookshelf (an imprint of 'The Pragmatic Programmers, LLC.'), http://pragprog.com/, 2009.

[96] M. Heidegger. *Sein und Zeit (Being and Time)*. Oxford University Press, 1927, 1962.

[97] C. A. R. Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985. Published electronically: http://www.usingcsp.-com/cspbook.pdf (2004).

[98] IEEE Computer Society. IEEE–STD 610.12-1990: Standard Glossary of Software Engineering Terminology. Technical report, IEEE, IEEE Headquarters Office, 1730 Massachusetts Avenue, N.W., Washington, DC 20036-1992, USA. Phone: +1-202-371-0101, FAX: +1-202-728-9614, 1990.

[99] ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992, 1996, 1999.

[100] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-10114-9.

[101] M. A. Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. ACM Press. Addison-Wesley, Reading, England, 1995.

[102] M. A. Jackson. *Problem Frames — Analyzing and Structuring Software Development Problems*. ACM Press, Pearson Education. Addison-Wesley, England, 2001.

[103] M. A. Jackson. Program Verification and System Dependability. In P. Boca and J. Bowen, editors, *Formal Methods: State of the Art and New Directions*, pages 43–78, London, UK, 2010. Springer.

[104] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[105] I. Johansson. Qualities, Quantities, and the Endurant-Perdurant Distinction in Top-Level Ontologies. In D. A. B. R. N. M. R.-B. T. Althoff, K.-D., editor, *Professional Knowledge Management WM 2005*, volume 3782 of *Lecture Notes in Artificial Intelligence*, pages 543–550. Springer, 2005. 3rd Biennial Conference, Kaiserslautern, Germany, April 10-13, 2005, Revised Selected Papers.

[106] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. FODA: Feature-Oriented Domain Analysis. Feasibility Study CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, November 1990. http://www.sei.cmu.edu/library/abstracts/reports/90tr021.cfm.

[107] J. Kim. *Supervenience and Mind*. Cambridge University Press, 1993.

[108] J. Klose and H. Wittke. An automata based interpretation of Live Sequence Charts. In T. Margaria and W. Yi, editors, *TACAS 2001*, LNCS 2031, pages 512–527. Springer-Verlag, 2001.

[109] D. B. . Formal Software Techniques in Railway Systems. In E. 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.

[110] L. Lamport. *Specifying Systems*. Addison–Wesley, Boston, Mass., USA, 2002.

[111] J. Laprie, editor. *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault–Tolerant Systems*. Springer–Verlag, Vienna, 1992. In English, French, German, Italian and Japanese.

[112] S. Lauesen. *Software Requirements - Styles and Techniques*. Addison-Wesley, UK, 2002.

[113] C. Lejewski. A note on Leśniewksi's Axiom System for the Mereological Notion of Ingredient or Element. *Topoi*, 2(1):63–71, June, 1983.

[114] H. S. Leonard and N. Goodman. The Calculus of Individuals and its Uses. *Journal of Symbolic Logic*, 5:45–44, 1940.

[115] W. Little, H. Fowler, J. Coulson, and C. Onions. *The Shorter Oxford English Dictionary on Historical Principles*. Clarendon Press, Oxford, England, 1987.

[116] N. Medvidovic and E. Colbert. Domain-Specific Software Architectures (DSSA). Power Point Presentation, found on The Internet, Absolute Software Corp., Inc.: `Abs[S/W]`, 5 March 2004.

[117] D. Mellor. Things and Causes in Spacetime. *British Journal for the Philosophy of Science*, 31:282–288, 1980.

[118] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, December 2005.

[119] Merriam Webster Staff. Online Dictionary: `http://www.m-w.com/home.htm`, 2004. Merriam–Webster, Inc., 47 Federal Street, P.O. Box 281, Springfield, MA 01102, USA.

[120] E. Mettala and M. H. Graham. The Domain Specific Software Architecture Program. Project Report CMU/SEI-92-SR-009, Software Engineering Institute Carnegie Mellon University Pittsburgh, Pennsylvania 15213, June 1992.

[121] J. M. Neighbors. The Draco Approach to Constructing Software from Reusable Components. *IEEE Transactions of Software Engineering*, SE-10(5), September 1984.

[122] S. L. Pfleeger. *Software Engineering, Theory and Practice*. Prentice–Hall, 2nd edition, 2001.

[123] C.-Y. T. Pi. *Mereology in Event Semantics*. Phd, McGill University, Montreal, Canada, August 1999.

[124] K. Pohl, G. Bockle, and F. van der Linden. *Software Product Line Engineering*. Springer, Berlin, Heidelberg, New York, 2005.

[125] R. S. Pressman. *Software Engineering, A Practitioner's Approach*. International Edition, Computer Science Series. McGraw–Hill, 5th edition, 1981–2001.

[126] R. Prieto-Díaz. Domain Analysis for Reusability. In *COMPSAC 87*. ACM Press, 1987.

[127] R. Prieto-Díaz. Domain analysis: an introduction. *Software Engineering Notes*, 15(2):47–54, 1990.

[128] R. Prieto-Díaz and G. Arrango. *Domain Analysis and Software Systems Modelling*. IEEE Computer Society Press, 1991.

[129] R. Pucella and V. Weissman. A Logic for Reasoning about Digital Rights. In *Proc. of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, pages 282–294. IEEE Computer Society Press, 2002.

[130] A. Quinton. Objects and Events. *Mind*, 88:197–214, 1979.

[131] B. Randell. On Failures and Faults. In *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 18–39. Formal Methods Europe, Springer, 2003. Invited paper.

[132] W. Reisig. *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*. Leitfäden der Informatik. Vieweg+Teubner, 1st edition, 15 June 2010. 248 pages; ISBN 978-3-8348-1290-2.

[133] J. Renz and H. W. Guesgen, editors. *Spatial and Temporal Reasoning*, volume 14, vol. 4, Journal: AI Communications, Amsterdam, The Netherlands, Special Issue. IOS Press, December 2004.

[134] J. C. Reynolds. *The Semantics of Programming Languages*. Cambridge University Press, 1999.

[135] D. T. Ross. Computer-aided design. *Commun. ACM*, 4(5):41–63, 1961.

[136] D. 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.

[137] D. T. Ross and J. E. Ward. Investigations in computer-aided design for numerically controlled production. Final Technical Report ESL-FR-351, , May 1968. 1 December 1959 – 3 May 1967. Electronic Systems Laboratory Electrical Engineering Department, MIT, Cambridge, Massachusetts 02139.

[138] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[139] P. Samuelson. Digital rights management {and, or, vs.} the law. *Communications of ACM*, 46(4):41–45, Apr 2003.

[140] D. Sannella and A. Tarlecki. *Foundations of Algebraic Semantcs and Formal Software Development*. Monographs in Theoretical Computer Science. Springer, Heidelberg, 2012.

[141] D. A. Schmidt. *Denotational Semantics: a Methodology for Language Development*. Allyn & Bacon, 1986.

[142] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[143] B. Smith. Mereotopology: A Theory of Parts and Boundaries. *Data and Knowledge Engineering*, 20:287–303, 1996.

[144] I. Sommerville. *Software Engineering*. Pearson, 8th edition, 2006.

[145] J. F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Brooks/Cole Thompson Learning, August 17, 1999.

[146] D. Spinellis. Notable design patterns for domain specific languages. *Journal of Systems and Software*, 56(1):91–99, Feb. 2001.

[147] Staff of Encyclopœdia Brittanica. Encyclopœdia Brittanica. Merriam Webster/Brittanica: Access over the Web: http://www.eb.com:180/, 1999.

[148] R. Tennent. *The Semantics of Programming Languages*. Prentice–Hall Intl., 1997.

[149] W. Tracz. Domain-specific software architecture (DSSA) frequently asked questions (FAQ). *Software Engineering Notes*, 19(2):52–56, 1994.

[150] J. van Benthem. *The Logic of Time*, volume 156 of *Synthese Library: Studies in Epistemology, Logic, Methhodology, and Philosophy of Science (Editor: Jaakko Hintika)*. Kluwer Academic Publishers, P.O.Box 17, NL 3300 AA Dordrecht, The Netherlands, second edition, 1983, 1991.

[151] F. Van der Rhee, H. Van Nauta Lemke, and J. Dukman. Knowledge based fuzzy control of systems. *IEEE Trans. Autom. Control*, 35(2):148–155, Feb. 1990.

[152] A. van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[153] H. Wang, J. S. Dong, and J. Sun. Reasoning Support for Semantic Web Ontology Family Languages Using Alloy. *International Journal of Multiagent and Grid Systems, IOS Press*, 2(4):455–471, 2006.

[154] A. Whitehead. *The Concept of Nature*. Cambridge University Press, Cambridge, 1920.

[155] G. Wilson and S. Shpall. Action. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2012 edition, 2012.

[156] G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, Cambridge, Mass., USA, 1993.

[157] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.

[158] WWW. Domain Descriptions:

1 *A Container Line Industry Domain:* `www2.imm.dtu.dk/~db/container--paper.pdf`
2 *What is Logistics:* `www2.imm.dtu.dk/~db/logistics.pdf`
3 *The "Market": Consumers, Retailers, Wholesalers, Producers* `www2.imm.dtu.-dk/~db/themarket.pdf`
4 *MITS: Models of IT Security. Security Rules & Regulations:* `www2.imm.dtu.-dk/~db/it-security.pdf`
5 *A Domain Model of Oil Pipelines:* `www2.imm.dtu.dk/~db/pipeline.pdf`
6 *A Railway Systems Domain:* `http://euler.fd.cvut.cz/railwaydomain`
7 *Transport Systems:* `www2.imm.dtu.dk/~db/comet/comet1.pdf`
8 *The Tokyo Stock Exchange* `www2.imm.dtu.dk/~db/todai/tse-1.pdf` and `www2.imm.dtu.dk/~db/todai/tse-2.pdf`
9 *On Development of Web-based Software. A Divertimento of Ideas and Suggestions:* `www2.imm.dtu.dk/~db/wfdftp.pdf`

. R&D Experiments, Dines Bjørner, DTU Informatics, Technical University of Denmark, 2007–2010.

[159] J. Xiang and D. Bjørner. The Electronic Media Industry: A Domain Analysis and a License Language. Technical note, JAIST, School of Information Science, 1-1, Asahidai, Tatsunokuchi, Nomi, Ishikawa, Japan 923-1292, Summer 2006.

[160] C. C. Zhou and M. R. Hansen. *Duration Calculus: A Formal Approach to Real–time Systems*. Monographs in Theoretical Computer Science. An EATCS Series. Springer–Verlag, 2004.

Part IV

Appendix

Appendix A

RSL

A.1 RSL: The Raise Specification Language

A.1.1 Type Expressions

Type expressions are expressions whose value are type, that is, possibly infinite sets of values (of "that" type).

Atomic Types

Atomic types have (atomic) values. That is, values which we consider to have no proper constituent (sub-)values, i.e., cannot, to us, be meaningfully "taken apart".

RSL has a number of *built-in* atomic types. There are the Booleans, integers, natural numbers, reals, characters, and texts.

type
[1] **Bool** **true**, **false**
[2] **Int** ... , −2, −2, 0, 1, 2, ...
[3] **Nat** 0, 1, 2, ...
[4] **Real** ..., −5.43, −1.0, 0.0, 1.23···, 2,7182···, 3,1415···, 4.56, ...
[5] **Char** "a", "b", ..., "0", ...
[6] **Text** "abracadabra"

Composite Types

Composite types have composite values. That is, values which we consider to have proper constituent (sub-)values, i.e., can be meaningfully "taken apart". There are two ways of expressing composite types: either explicitly, using concrete type expressions, or implicitly, using sorts (i.e., abstract types) and observer functions.

Concrete Composite Types From these one can form type expressions: finite sets, infinite sets, Cartesian products, lists, maps, etc.

Let A, B and C be any type names or type expressions, then:

[7] A-**set**
[8] A-**infset**

[9] A × B × ... × C
[10] A*
[11] A^ω
[12] A ~~→~~ B
[13] A ~~→~~ B
[14] A ~→ B
[15] (A)
[16] A | B | ... | C
[17] mk_id(sel_a:A,...,sel_b:B)
[18] sel_a:A ... sel_b:B

The following are generic type expressions:

1 The Boolean type of truth values **false** and **true**.

2 The integer type on integers ..., –2, –1, 0, 1, 2, ...

3 The natural number type of positive integer values 0, 1, 2, ...

4 The real number type of real values, i.e., values whose numerals can be written as an integer, followed by a period (“.”), followed by a natural number (the fraction).

5 The character type of character values ″a″, ″b″, ...

6 The text type of character string values ″aa″, ″aaa″, ..., ″abc″, ...

7 The set type of finite cardinality set values.

8 The set type of infinite and finite cardinality set values.

9 The Cartesian type of Cartesian values.

10 The list type of finite length list values.

11 The list type of infinite and finite length list values.

12 The map type of finite definition set map values.

13 The function type of total function values.

14 The function type of partial function values.

15 In (A) A is constrained to be:

- either a Cartesian B × C × ... × D, in which case it is identical to type expression kind 9,
- or not to be the name of a built-in type (cf., 1–6) or of a type, in which case the parentheses serve as simple delimiters, e.g., (A ~~→~~ B), or (A*)**-set**, or (A-**set**)list, or (A|B) ~~→~~ (C|D|(E ~~→~~ F)), etc.

16 The postulated disjoint union of types A, B, ..., and C.

17 The record type of mk_id-named record values mk_id(av,...,bv), where av, ..., bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.

18 The record type of unnamed record values (av,...,bv), where av, ..., bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.

Sorts and Observer Functions

type
  A, B, C, ..., D
value
  obs_B: A → B, obs_C: A → C, ..., obs_D: A → D

The above expresses that values of type A are composed from at least three values — and these are of type B, C, ..., and D. A concrete type definition corresponding to the above presupposing material of the next section

type
  B, C, ..., D
  A = B × C × ... × D

A.1.2 Type Definitions

Concrete Types

Types can be concrete in which case the structure of the type is specified by type expressions:

type
  A = Type_expr

Some schematic type definitions are:

[1] Type_name = Type_expr /∗ without | s or subtypes ∗/
[2] Type_name = Type_expr_1 | Type_expr_2 | ... | Type_expr_n
[3] Type_name ==
  mk_id_1(s_a1:Type_name_a1,...,s_ai:Type_name_ai) |
  ... |
  mk_id_n(s_z1:Type_name_z1,...,s_zk:Type_name_zk)
[4] Type_name :: sel_a:Type_name_a ... sel_z:Type_name_z
[5] Type_name = {| v:Type_name′ • $\mathscr{P}$(v) |}

where a form of [2–3] is provided by combining the types:

  Type_name = A | B | ... | Z
  A == mk_id_1(s_a1:A_1,...,s_ai:A_i)
  B == mk_id_2(s_b1:B_1,...,s_bj:B_j)
  ...
  Z == mk_id_n(s_z1:Z_1,...,s_zk:Z_k)

Types A, B, ..., Z are disjoint, i.e., shares no values, provided all mk_id_k are distinct and due to the use of the disjoint record type constructor ==.

axiom
  ∀ a1:A_1, a2:A_2, ..., ai:Ai •
    s_a1(mk_id_1(a1,a2,...,ai))=a1 ∧ s_a2(mk_id_1(a1,a2,...,ai))=a2 ∧
    ... ∧ s_ai(mk_id_1(a1,a2,...,ai))=ai ∧
  ∀ a:A • **let** mk_id_1(a1′,a2′,...,ai′) = a **in**
    a1′ = s_a1(a) ∧ a2′ = s_a2(a) ∧ ... ∧ ai′ = s_ai(a) **end**

Subtypes

In RSL, each type represents a set of values. Such a set can be delimited by means of predicates. The set of values b which have type B and which satisfy the predicate $\mathscr{P}$, constitute the subtype A:

type
A = {| b:B • $\mathscr{P}$(b) |}

Sorts — Abstract Types

Types can be (abstract) sorts in which case their structure is not specified:

type
A, B, ..., C

A.1.3 The RSL Predicate Calculus

Propositional Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values (**true** or **false** [or **chaos**]). Then:

false, **true**
a, b, ..., c ∼a, a∧b, a∨b, a⇒b, a=b, a≠b

are propositional expressions having Boolean values. ∼, ∧, ∨, ⇒, = and ≠ are Boolean connectives (i.e., operators). They can be read as: *not*, *and*, *or*, *if then* (or *implies*), *equal* and *not equal*.

Simple Predicate Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values, let x, y, ..., z (or term expressions) designate non-Boolean values and let i, j, ..., k designate number values, then:

false, **true**
a, b, ..., c
∼a, a∧b, a∨b, a⇒b, a=b, a≠b
x=y, x≠y,
i<j, i≤j, i≥j, i≠j, i≥j, i>j

are simple predicate expressions.

Quantified Expressions

Let X, Y, ..., C be type names or type expressions, and let $\mathscr{P}(x)$, $\mathscr{Q}(y)$ and $\mathscr{R}(z)$ designate predicate expressions in which x, y and z are free. Then:

∀ x:X • $\mathscr{P}(x)$
∃ y:Y • $\mathscr{Q}(y)$
∃ ! z:Z • $\mathscr{R}(z)$

are quantified expressions — also being predicate expressions.

They are "read" as: For all x (values in type X) the predicate $\mathscr{P}(x)$ holds; there exists (at least) one y (value in type Y) such that the predicate $\mathscr{Q}(y)$ holds; and there exists a unique z (value in type Z) such that the predicate $\mathscr{R}(z)$ holds.

A.1.4 Concrete RSL Types: Values and Operations

Arithmetic

type
  Nat, **Int**, **Real**
value
  $+,-,*$: **Nat**×**Nat**→**Nat** | **Int**×**Int**→**Int** | **Real**×**Real**→**Real**
  /: **Nat**×**Nat**$\xrightarrow{\sim}$**Nat** | **Int**×**Int**$\xrightarrow{\sim}$**Int** | **Real**×**Real**$\xrightarrow{\sim}$**Real**
  $<,\leq,=,\neq,\geq,>$ (**Nat**|**Int**|**Real**) → (**Nat**|**Int**|**Real**)

Set Expressions

Set Enumerations Let the below a's denote values of type A, then the below designate simple set enumerations:

$\{\{\}, \{a\}, \{e_1,e_2,...,e_n\}, ...\} \in$ **A-set**
$\{\{\}, \{a\}, \{e_1,e_2,...,e_n\}, ..., \{e_1,e_2,...\}\} \in$ **A-infset**

Set Comprehension The expression, last line below, to the right of the ≡, expresses set comprehension. The expression "builds" the set of values satisfying the given predicate. It is abstract in the sense that it does not do so by following a concrete algorithm.

type
  A, B
  P = A → **Bool**
  Q = A $\xrightarrow{\sim}$ B
value
  comprehend: **A-infset** × P × Q → **B-infset**
  comprehend(s,P,Q) ≡ { Q(a) | a:A • a ∈ s ∧ P(a)}

Cartesian Expressions

Cartesian Enumerations Let e range over values of Cartesian types involving A, B, ..., C, then the below expressions are simple Cartesian enumerations:

type
  A, B, ..., C
  A × B × ... × C
value
  (e1,e2,...,en)

List Expressions

List Enumerations Let a range over values of type A, then the below expressions are simple list enumerations:

$\{\langle\rangle, \langle e\rangle, ..., \langle e1,e2,...,en\rangle, ...\} \in A^*$
$\{\langle\rangle, \langle e\rangle, ..., \langle e1,e2,...,en\rangle, ..., \langle e1,e2,...,en,...\rangle, ...\} \in A^\omega$

$\langle a\_i .. a\_j \rangle$

The last line above assumes a_i and a_j to be integer-valued expressions. It then expresses the set of integers from the value of e_i to and including the value of e_j. If the latter is smaller than the former, then the list is empty.

List Comprehension The last line below expresses list comprehension.

```
type
  A, B, P = A → Bool, Q = A ~→ B
value
  comprehend: A^ω × P × Q ~→ B^ω
  comprehend(l,P,Q) ≡
    ⟨ Q(l(i)) | i in ⟨1..len l⟩ • P(l(i))⟩
```

Map Expressions

Map Enumerations Let (possibly indexed) u and v range over values of type $T1$ and $T2$, respectively, then the below expressions are simple map enumerations:

```
type
  T1, T2
  M = T1 →m T2
value
  u,u1,u2,...,un:T1, v,v1,v2,...,vn:T2
  [], [u↦v], ..., [u1↦v1,u2↦v2,...,un↦vn] ∀ ∈ M
```

Map Comprehension The last line below expresses map comprehension:

```
type
  U, V, X, Y
  M = U →m V
  F = U ~→ X
  G = V ~→ Y
  P = U → Bool
value
  comprehend: M×F×G×P → (X →m Y)
  comprehend(m,F,G,P) ≡
    [ F(u) ↦ G(m(u)) | u:U • u ∈ dom m ∧ P(u)]
```

Set Operations

Set Operator Signatures

```
value
  19 ∈: A × A-infset → Bool
  20 ∉: A × A-infset → Bool
  21 ∪: A-infset × A-infset → A-infset
  22 ∪: (A-infset)-infset → A-infset
  23 ∩: A-infset × A-infset → A-infset
  24 ∩: (A-infset)-infset → A-infset
  25 \: A-infset × A-infset → A-infset
```

26 $\subset$: A-**infset** $\times$ A-**infset** $\to$ **Bool**
27 $\subseteq$: A-**infset** $\times$ A-**infset** $\to$ **Bool**
28 $=$: A-**infset** $\times$ A-**infset** $\to$ **Bool**
29 $\neq$: A-**infset** $\times$ A-**infset** $\to$ **Bool**
30 **card**: A-**infset** $\xrightarrow{\sim}$ **Nat**

Set Examples

examples
a $\in$ {a,b,c}
a $\notin$ {}, a $\notin$ {b,c}
{a,b,c} $\cup$ {a,b,d,e} = {a,b,c,d,e}
$\cup${{a},{a,b},{a,d}} = {a,b,d}
{a,b,c} $\cap$ {c,d,e} = {c}
$\cap${{a},{a,b},{a,d}} = {a}
{a,b,c} $\setminus$ {c,d} = {a,b}
{a,b} $\subset$ {a,b,c}
{a,b,c} $\subseteq$ {a,b,c}
{a,b,c} = {a,b,c}
{a,b,c} $\neq$ {a,b}
card {} = 0, **card** {a,b,c} = 3

Informal Explication

19 $\in$: The membership operator expresses that an element is a member of a set.

20 $\notin$: The nonmembership operator expresses that an element is not a member of a set.

21 $\cup$: The infix union operator. When applied to two sets, the operator gives the set whose members are in either or both of the two operand sets.

22 $\cup$: The distributed prefix union operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.

23 $\cap$: The infix intersection operator. When applied to two sets, the operator gives the set whose members are in both of the two operand sets.

24 $\cap$: The prefix distributed intersection operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.

25 $\setminus$: The set complement (or set subtraction) operator. When applied to two sets, the operator gives the set whose members are those of the left operand set which are not in the right operand set.

26 $\subseteq$: The proper subset operator expresses that all members of the left operand set are also in the right operand set.

27 $\subset$: The proper subset operator expresses that all members of the left operand set are also in the right operand set, and that the two sets are not identical.

28 $=$: The equal operator expresses that the two operand sets are identical.

29 $\neq$: The nonequal operator expresses that the two operand sets are *not* identical.

30 **card**: The cardinality operator gives the number of elements in a finite set.

Set Operator Definitions

The operations can be defined as follows (≡ is the definition symbol):

```
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 ∗/
```

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
```

List Operations

List Operator Signatures

```
value
  hd: A^ω ~→ A
  tl: A^ω ~→ A^ω
  len: A^ω ~→ Nat
  inds: A^ω → Nat-infset
  elems: A^ω → A-infset
  .(.): A^ω × Nat ~→ A
  ⌢: A* × A^ω → A^ω
```

List Operation Examples

```
examples
  hd⟨a1,a2,...,am⟩=a1
  tl⟨a1,a2,...,am⟩=⟨a2,...,am⟩
  len⟨a1,a2,...,am⟩=m
  inds⟨a1,a2,...,am⟩={1,2,...,m}
  elems⟨a1,a2,...,am⟩={a1,a2,...,am}
```

⟨a1,a2,...,am⟩(i)=ai
⟨a,b,c⟩⌢⟨a,b,d⟩ = ⟨a,b,c,a,b,d⟩
⟨a,b,c⟩=⟨a,b,c⟩
⟨a,b,c⟩ ≠ ⟨a,b,d⟩

Informal Explication

- **hd**: Head gives the first element in a nonempty list.
- **tl**: Tail gives the remaining list of a nonempty list when Head is removed.
- **len**: Length gives the number of elements in a finite list.
- **inds**: Indices give the set of indices from 1 to the length of a nonempty list. For empty lists, this set is the empty set as well.
- **elems**: Elements gives the possibly infinite set of all distinct elements in a list.
- $\ell(i)$: Indexing with a natural number, i larger than 0, into a list ℓ having a number of elements larger than or equal to i, gives the ith element of the list.
- $\widehat{\ }$: 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.
- $\neq$: The nonequal operator expresses that the two operand lists are *not* identical.

The operations can also be defined as follows:

List Operator Definitions

```
value
  is_finite_list: A^ω → Bool

  len q ≡
    case is_finite_list(q) of
      true → if q = ⟨⟩ then 0 else 1 + len tl q end,
      false → chaos end

  inds q ≡
    case is_finite_list(q) of
      true → { i | i:Nat • 1 ≤ i ≤ len q },
      false → { i | i:Nat • i≠0 } end

  elems q ≡ { q(i) | i:Nat • i ∈ inds q }


  q(i) ≡
    if i=1
      then
        if q≠⟨⟩
          then let a:A,q′:Q • q=⟨a⟩⌢q′ in a end
          else chaos end
```

else q(i−1) **end**

fq ⌢ iq ≡
  ⟨ **if** 1 ≤ i ≤ **len** fq **then** fq(i) **else** iq(i − **len** fq) **end**
  | i:**Nat** • **if len** iq≠**chaos then** i ≤ **len** fq+**len end** ⟩
  pre is_finite_list(fq)

iq′ = iq″ ≡
  inds iq′ = **inds** iq″ ∧ ∀ i:**Nat** • i ∈ **inds** iq′ ⇒ iq′(i) = iq″(i)

iq′ ≠ iq″ ≡ ∼(iq′ = iq″)

Map Operations
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′]

Map Operation Explication

- $m(a)$: Application gives the element that a maps to in the map m.
- **dom**: Domain/Definition Set gives the set of values which *maps to* in a map.
- **rng**: Range/Image Set gives the set of values which *are mapped to* in a map.

- †: Override/Extend. When applied to two operand maps, it gives the map which is like an override of the left operand map by all or some "pairings" of the right operand map.
- ∪: Merge. When applied to two operand maps, it gives a merge of these maps.
- \: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements that are not in the right operand set.
- /: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements of the right operand set.
- =: The equal operator expresses that the two operand maps are identical.
- ≠: The nonequal operator expresses that the two operand maps are *not* identical.
- °: Composition. When applied to two operand maps, it gives the map from definition set elements of the left operand map, m_1, to the range elements of the right operand map, m_2, such that if a is in the definition set of m_1 and maps into b, and if b is in the definition set of m_2 and maps into c, then a, in the composition, maps into c.

Map Operation Redefinitions The map operations can also be defined as follows:

```
value
  rng m ≡ { m(a) | a:A • a ∈ dom m }

  m1 † m2 ≡
    [ a↦b | a:A,b:B •
      a ∈ dom m1 \ dom m2 ∧ 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
```

A.1.5 λ-Calculus + Functions

The λ-Calculus Syntax

```
type /∗ A BNF Syntax: ∗/
    ⟨L⟩ ::= ⟨V⟩ | ⟨F⟩ | ⟨A⟩ | ( ⟨A⟩ )
    ⟨V⟩ ::= /∗ variables, i.e. identifiers ∗/
    ⟨F⟩ ::= λ⟨V⟩ • ⟨L⟩
    ⟨A⟩ ::= ( ⟨L⟩⟨L⟩ )
value /∗ Examples ∗/
```

⟨L⟩: e, f, a, ...
⟨V⟩: x, ...
⟨F⟩: λ x • e, ...
⟨A⟩: f a, (f a), f(a), (f)(a), ...

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 $\lambda y \cdot e$ 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).

Substitution

In RSL, the following rules for substitution apply:

- **subst**([N/x]x) ≡ N;
- **subst**([N/x]a) ≡ a,

  for all variables a≠ x;
- **subst**([N/x](P Q)) ≡ (**subst**([N/x]P) **subst**([N/x]Q));
- **subst**([N/x](λ*x*•*P*)) ≡ λ y•P;
- **subst**([N/x](λ y•P)) ≡ λ*y*• **subst**([N/x]P),

  if x≠y and y is not free in N or x is not free in P;
- **subst**([N/x](λy•P)) ≡ λz•**subst**([N/z]**subst**([z/y]P)),

  if y≠x and y is free in N and x is free in P

  (where z is not free in (N P)).

α-Renaming and β-Reduction

- α-renaming: λx•M

  If x, y are distinct variables then replacing x by y in λx•M results in λy•**subst**([y/x]M). We can rename the formal parameter of a λ-function expression provided that no free variables of its body M thereby become bound.
- β-reduction: (λx•M)(N)

  All free occurrences of x in M are replaced by the expression N provided that no free variables of N thereby become bound in the result. (λx•M)(N) ≡ **subst**([N/x]M)

Function Signatures

For sorts we may want to postulate some functions:

type
  A, B, C
value
  obs_B: A → B,
  obs_C: A → C,
  gen_A: B×C → A

Function Definitions

Functions can be defined explicitly:

value
  f: Arguments → Result
  f(args) ≡ DValueExpr

  g: Arguments $\xrightarrow{\sim}$ Result
  g(args) ≡ ValueAndStateChangeClause
  pre P(args)

Or functions can be defined implicitly:

value
  f: Arguments → Result
  f(args) **as** result
  post P1(args,result)

  g: Arguments $\xrightarrow{\sim}$ Result
  g(args) **as** result
  pre P2(args)
  post P3(args,result)

The symbol $\xrightarrow{\sim}$ indicates that the function is partial and thus not defined for all arguments. Partial functions should be assisted by preconditions stating the criteria for arguments to be meaningful to the function.

A.1.6 Other Applicative Expressions

Simple let Expressions

Simple (i.e., nonrecursive) **let** expressions:

  let a = $\mathscr{E}_d$ **in** $\mathscr{E}_b$(a) **end**

is an "expanded" form of:

  (λa.$\mathscr{E}_b$(a))($\mathscr{E}_d$)

Recursive let Expressions

Recursive **let** expressions are written as:

let f = λa:A • E(f) **in** B(f,a) **end**

is "the same" as:

let f = **Y**F **in** B(f,a) **end**

where:

F ≡ λg•λa•(E(g)) and YF = F(YF)

Predicative let Expressions

Predicative **let** expressions:

let a:A • $\mathscr{P}$(a) **in** $\mathscr{B}$(a) **end**

express the selection of a value a of type A which satisfies a predicate $\mathscr{P}$(a) for evaluation in the body $\mathscr{B}$(a).

Pattern and "Wild Card" let Expressions

Patterns and *wild cards* can be used:

let {a} ∪ s = set **in** ... **end**
let {a,_} ∪ s = set **in** ... **end**

let (a,b,...,c) = cart **in** ... **end**
let (a,_,...,c) = cart **in** ... **end**

let ⟨a⟩^ℓ = list **in** ... **end**
let ⟨a,_,b⟩^ℓ = list **in** ... **end**

let [a↦b] ∪ m = map **in** ... **end**
let [a↦b,_] ∪ m = map **in** ... **end**

Conditionals

Various kinds of conditional expressions are offered by RSL:

if b_expr **then** c_expr **else** a_expr
end

if b_expr **then** c_expr **end** ≡ /∗ same as: ∗/
if b_expr **then** c_expr **else skip end**

if b_expr_1 **then** c_expr_1

```
elsif b_expr_2 then c_expr_2
elsif b_expr_3 then c_expr_3
...
elsif b_expr_n then c_expr_n end

case expr of
   choice_pattern_1 → expr_1,
   choice_pattern_2 → expr_2,
   ...
   choice_pattern_n_or_wild_card → expr_n
end
```

Operator/Operand Expressions

⟨Expr⟩ ::=
⟨Prefix_Op⟩ ⟨Expr⟩
| ⟨Expr⟩ ⟨Infix_Op⟩ ⟨Expr⟩
| ⟨Expr⟩ ⟨Suffix_Op⟩
| ...
⟨Prefix_Op⟩ ::=
− | ∼ | ∪ | ∩ | **card** | **len** | **inds** | **elems** | **hd** | **tl** | **dom** | **rng**
⟨Infix_Op⟩ ::=
= | ≠ | ≡ | + | − | ∗ | ↑ | / | < | ≤ | ≥ | > | ∧ | ∨ | ⇒
| ∈ | ∉ | ∪ | ∩ | \ | ⊂ | ⊆ | ⊇ | ⊃ | ⌢ | † | ◦
⟨Suffix_Op⟩ ::= !

A.1.7 Imperative Constructs

Statements and State Changes

Often, following the RAISE method, software development starts with highly abstract-applicative constructs which, through stages of refinements, are turned into concrete and imperative constructs. Imperative constructs are thus inevitable in RSL.

```
   Unit
value
   stmt: Unit → Unit
   stmt()
```

- Statements accept no arguments.
- Statement execution changes the state (of declared variables).
- **Unit** → **Unit** designates a function from states to states.
- Statements, stmt, denote state-to-state changing functions.
- Writing () as "only" arguments to a function "means" that () is an argument of type **Unit**.

Variables and Assignment

0. **variable** v:Type := expression
1. v := expr

Statement Sequences and skip

Sequencing is expressed using the ';' operator. **skip** is the empty statement having no value or side-effect.

2. **skip**
3. stm_1;stm_2;...;stm_n

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**

Iterative Conditionals

6. **while** expr **do** stm **end**
7. **do** stmt **until** expr **end**

Iterative Sequencing

8. **for** e **in** list_expr • P(b) **do** S(b) **end**

A.1.8 Process Constructs

Process Channels

Let A and B stand for two types of (channel) messages and i:KIdx for channel array indexes, then:

channel c:A
channel { k[i]:B • i:KIdx }

declare a channel, c, and a set (an array) of channels, k[i], capable of communicating values of the designated types (A and B).

Process Composition

Let P and Q stand for names of process functions, i.e., of functions which express willingness to engage in input and/or output events, thereby communicating over declared channels. Let P() and Q stand for process expressions, then:

P ‖ Q Parallel composition
P [] Q Nondeterministic external choice (either/or)
P ⨅ Q Nondeterministic internal choice (either/or)
P ⫲ Q Interlock parallel composition

express the parallel (‖) of two processes, or the nondeterministic choice between two processes: either external ([]) or internal (⨅). The interlock (⫲) composition expresses that the two processes are forced to communicate only with one another, until one of them terminates.

Input/Output Events

Let c, k[i] and e designate channels of type A and B, then:

c ?, k[i] ? Input
c ! e, k[i] ! e Output

expresses the willingness of a process to engage in an event that "reads" an input, respectively "writes" an output.

Process Definitions

The below signatures are just examples. They emphasise that process functions must somehow express, in their signature, via which channels they wish to engage in input and output events.

value
P: **Unit** → **in** c **out** k[i]
Unit
Q: i:KIdx → **out** c **in** k[i] **Unit**

P() ≡ ... c ? ... k[i] ! e ...
Q(i) ≡ ... k[i] ? ... c ! e ...

The process function definitions (i.e., their bodies) express possible events.

A.1.9 Simple RSL Specifications

Often, we do not want to encapsulate small specifications in schemes, classes, and objects, as is often done in RSL. An RSL specification is simply a sequence of one or more types, values (including functions), variables, channels and axioms:

type
...
variable
...
channel
...
value
...
axiom
...

In practice a full specification repeats the above listings many times, once for each "module" (i.e., aspect, facet, view) of specification. Each of these modules may be "wrapped" into scheme, class or object definitions.[1]

[1] For schemes, classes and objects we refer to [23, Chap. 10]

Appendix B

Indexes

B.1. Endurant Analysis Prompts 216
B.2. Description Language Observers and "Built-in" Functions 216
B.3. Description Prompts and Their Schemas 216
B.4. Attribute Analysis Prompts 217
B.5. [Well-formedness] Axioms 217
B.6. [Disjoint Sort] Proof Obligations 217
B.7. Definitions 217
B.8. Examples 222
B.9. Concepts 225
B.10 Index of RSL Language Constructs 233

B.1 Index of Endurant Analysis Prompts

a. `is_ entity`, 23
b. `is_ endurant`, 24
c. `is_ perdurant`, 24
d. `is_ discrete`, 24
e. `is_ continuous`, 24
f. `is_ part`, 25
g. `is_ component`, 25
h. `is_ material`, 26
i. `is_ atomic`, 26
j. `is_ composite`, 26
k. `observe_ parts`, 27
l. `has_ concrete_ type`, 29
m. `has_ mereology`, 34
n. `attribute_ names`, 38
o. `has_ components`, 44
p. `has_ materials`, 45

B.2 Description Language Observers and "Built-in" Functions

a. **obs_ part_** P, 28
b. **is_** P, 28
c. **obs_ part_** T, 29
d. **uid_ P**, 33
e. **mereo_ P**, 35
f. **upd_ mereology**, 36
g. **attr_** A, 38
h. **components**, 44
i. **obs_ part_** M, 46
j. **materials**, 46

B.3 Domain Description Prompts and Their Schemas

[1] observe_ part_ sorts, 28
[2] observe_ part_ type, 29
[3] observe_ unique_ identifier, 33
[4] observe_ mereology, 35
[5] observe_ attributes, 39
[6] observe_ component_ sorts, 44
[7] observe_ material_ sorts, 45

1. Part Sort Observers, 28
2. Part Type Observers, 29
3. Part Unique Identifier, 33
4. Part Mereology, 35
5. Part Attributes, 39
6. Component Observers, 44
7. Material Observers, 45

B.4 Attribute Analysis Prompts

A. is_ static_ attribute, 40
B. is_ dynamic_ attribute, 40
C. is_ inert_ attribute, 40
D. is_ reactive_ attribute, 40
E. is_ active_ attribute, 40
F. is_ autonomous_ attribute, 40
G. is_ biddable_ attribute, 40
H. is_ programmable_ attribute, 40
I. is_ external_ attribute, 41

B.5 [Well–formedness] Axioms

Domain Mereologies, 35
Hub States, HΣ, 40
Links, L, and Hubs, H, 34
Pipeline Route Descriptors, 49
Pipeline Systems, PLS (0), 36
Pipeline Systems, PLS (1), 47
Pipeline Systems, PLS (2), 48
Pipeline Systems, PLS (3), 50
Road Nets, N, 35

B.6 [Disjoint Sort] Proof Obligations

Disjointness of Attribute Types, 39
Disjointness of Component Sorts, 44
Disjointness of Material Sorts, 46
Disjointness of Part Sorts, 28

B.7 Definitions

description
  domain
    prompt, 29, 33, 39, 44, 45, 52, 71, 183
  prompt
    domain, 29, 33, 39, 44, 45, 52, 71, 183
domain
  description
    prompt, 29, 33, 39, 44, 45, 52, 71, 183
  prompt
    description, 29, 33, 39, 44, 45, 52, 71, 183
prompt
  description
    domain, 29, 33, 39, 44, 45, 52, 71, 183
  domain
    description, 29, 33, 39, 44, 45, 52, 71, 183
abstract
  type, 27
Accessibility, 174
action

discrete, 53
active
attribute, 40
Actor, 53
actor, 53
Adaptive Maintenance, 176
analysis
domain
prompt, 20, 23–27, 29, 34, 38, 44, 45, 52, 71, 183
prompt
domain, 20, 23–27, 29, 34, 38, 44, 45, 52, 71, 183
Atomic
part, 26
Atomic Part, 26
attribute
active, 40
behaviour
external, 56
biddable, 40
dynamic, 40
external, 41
behaviour, 56
inert, 40
programmable, 40
reactive, 40
shared, 42
static, 40
autonomous
attribute, 40
Availability, 174

behaviour
attribute
external, 56
continuous, 56
discrete, 54
external
attribute, 56
biddable
attribute, 40

Component, 25
component, 25
Composite
part, 26
Composite Part, 26
computer
science, 17
computing
science, 17
concept
formal, 22
concrete
type, 27
confusion, 48
context
formal, 22
continuous
behaviour, 56
endurant, 24
Continuous Endurant, 24
Corrective Maintenance, 177

Demonstration Platform Requirements, 178
Dependability, 173
Dependability Attribute, 174
derived, 30
description
domain, 18, 19
prompt, 20, 35
path
tree, 100
prompt
domain, 20, 35
text, 18
tree
path, 100
trees, 100
Determination, 160
determination
domain, 160
Development Platform Requirements, 178
discrete
action, 53
behaviour, 54
endurant, 24
Discrete Action, 53
Discrete Behaviour, 54
Discrete Endurant, 24
Documentation Requirements, 179
domain
analysis
prompt, 20, 23–27, 29, 34, 38, 44, 45, 52, 71, 183
description, 18, 19
prompt, 20, 35
determination, 160
extension, 161
facet, 74
instantiation, 156

intrinsics, 75
manifest, 17
partial
requirement, 167
prescription
requirements, 154
projection, 154
prompt
analysis, 20, 23–27, 29, 34, 38, 44, 45, 52, 71, 183
description, 20, 35
requirement
partial, 167
shared, 167
requirements, 151, 154
prescription, 154
shared
requirement, 167
stake-holder, 74
Domain Management, 79
Domain Organisation, 79
Domain Projection, 154
Domain Regulation, 82
Domain Requirements Prescription, 154
Domain Rules, 82
Domain Script, 84
duplicate
node, 98
dynamic
attribute, 40

Endurant, 23
endurant, 23
continuous, 24
discrete, 24
Entity, 23
entity, 23
Epistemology, 52
Error, 173
Event, 54
event, 54
Execution Platform Requirements, 178
expression
function
type, 57
type
function, 57
Extension, 161
extension
domain, 161
Extensional Maintenance, 177
extent, 22
external
attribute, 41
behaviour, 56
behaviour
attribute, 56
part
quality, 32
quality
part, 32

Facet, 74
facet
domain, 74
Failure, 173
Fault, 173
fitting
requirements, 167
formal
concept, 22
context, 22
function
expression
type, 57
signature, 57
type
expression, 57
Function Signature, 57
Function Type Expression, 57
functional
requirements, 154
Functional Requirements, 154

harmonisation
requirements, 167
has_concrete_type
prerequisite
prompt, 29
prompt
prerequisite, 29
has_mereology
prerequisite
prompt, 35
prompt
prerequisite, 35
head
pump, 56
Human Behaviour, 86

inert
attribute, 40
Instantiation, 156

instantiation
domain, 156
Integrity, 175
intent, 22
interface
requirements, 151, 154, 168
internal
part
quality, 32
quality
part, 32
Intrinsics, 75
intrinsics
domain, 75
is_ composite
prerequisite
prompt, 28
prompt
prerequisite, 28
is_ discrete
prerequisite
prompt, 27
prompt
prerequisite, 27
is_ entity
prerequisite
prompt, 23, 24
prompt
prerequisite, 23, 24

junk, 48

knowledge, 66

Machine, 152
machine, 151, 172
requirements, 151, 154, 172
Machine Requirements, 172
Machine Service, 173
Maintenance Platform Requirements, 178
Maintenance Requirements, 176
manifest
domain, 17
Material, 25
material, 25, 45
mereology, 34
type, 34
method, 16
methodology, 17

narrative
requirements
system, 153
user and external equipment, 153
system
requirements, 153
user and external equipment
requirements, 153
node
duplicate, 98

observe_ part_ type
prerequisite
prompt, 29
prompt
prerequisite, 29
Ontology, 52

Part, 25
part, 25
Atomic, 26
Composite, 26
external
quality, 32
internal
quality, 32
qualities, 32
quality
external, 32
internal, 32
partial
domain
requirement, 167
requirement
domain, 167
path
description
tree, 100
tree
description, 100
Perdurant, 23
perdurant, 23
Perfective Maintenance, 177
Performance Requirements, 172
phenomenon, 23
Platform, 178
Platform Requirements, 178
prerequisite
has_ concrete_ type
prompt, 29
has_ mereology
prompt, 35
is_ composite

prompt, 28
is_ discrete
prompt, 27
is_ entity
prompt, 23, 24
observe_ part_ type
prompt, 29
prompt
has_ concrete_ type, 29
has_ mereology, 35
is_ composite, 28
is_ discrete, 27
is_ entity, 23, 24
observe_ part_ type, 29
prescription
domain
requirements, 154
requirements, 151
domain, 154
Preventive Maintenance, 177
problem/objective
sketch, 153
Problem/Objective Sketch, 153
programmable
attribute, 40
projection
domain, 154
prompt
analysis
domain, 20, 23–27, 29, 34, 38, 44, 45, 52, 71, 183
description
domain, 20, 35
domain
analysis, 20, 23–27, 29, 34, 38, 44, 45, 52, 71, 183
description, 20, 35
has_ concrete_ type
prerequisite, 29
has_ mereology
prerequisite, 35
is_ composite
prerequisite, 28
is_ discrete
prerequisite, 27
is_ entity
prerequisite, 23, 24
observe_ part_ type
prerequisite, 29
prerequisite
has_ concrete_ type, 29
has_ mereology, 35
is_ composite, 28
is_ discrete, 27
is_ entity, 23, 24
observe_ part_ type, 29
pump
head, 56

qualities
part, 32
quality
external
part, 32
internal
part, 32
part
external, 32
internal, 32

reactive
attribute, 40
Reliability, 175
requirement
domain
partial, 167
shared, 167
partial
domain, 167
shared
domain, 167
requirements, 151, 152
domain, 151, 154
prescription, 154
fitting, 167
functional, 154
harmonisation, 167
interface, 151, 154, 168
machine, 151, 154, 172
narrative
system, 153
user and external equipment, 153
prescription, 151
domain, 154
system
narrative, 153
user and external equipment
narrative, 153
Requirements (I), 151
Requirements (II), 152
Requirements (III), 152
Requirements Fitting, 167

Requirements Harmonisation, 167
Robustness, 176

Safety, 175
science
    computer, 17
    computing, 17
Security, 176
share, 42
shared
    attribute, 42
    domain
        requirement, 167
    requirement
        domain, 167
sharing, 168
signature
    function, 57
sketch
    problem/objective, 153
Software, 179
software, 179
sort, 27
stake-holder, 74
    domain, 74
State, 52
state, 52
State-holder, 74
static
    attribute, 40
sub-part, 26
Support technology, 77
system
    narrative
        requirements, 153
    requirements
        narrative, 153
System Requirements, 153

text
    description, 18
The Machine, 172
tree
    description
        path, 100
    path
        description, 100
trees
    description, 100
type, 27
    abstract, 27
    concrete, 27
    expression
        function, 57
    function
        expression, 57
    mereology, 34

user and external equipment
    narrative
        requirements, 153
    requirements
        narrative, 153
User and External Equipment Requirements, 153

B.8 Examples

126 A Law of Train Traffic at Stations, 184
66 A Law of Train Traffic, 72
54 Actors, 53
36 Atomic Part Attributes, 38
18 Atomic Parts, 26
35 Attribute Propositions and Other Values, 38
41 Autonomous and Programmable Hub Attributes, 41
60 Bank System Channels, 55
58 Behaviours, 54
59 Bus System Channels, 55
65 Bus Timetable Coordination, 61
15 Components, 25
37 Composite Part Attributes, 38
19 Composite Parts, 26
20 Composite and Atomic Part Sorts of Transportation, 28
21 Concrete Part Types of Transportation, 30
46 Container Components, 45
22 Container Line Sorts, 30
13 Continuous Endurants, 24
28 Derivation Chains, 32
12 Discrete Endurants, 24
4 Endurant Entity Qualities, 17
61 Flow in Pipelines, 56
7 Formal Description of Bank System Endurants, 18
9 Formal Description of Bank System Perdurants, 19

39 Inert and Reactive Attributes, 40
62 Insert Hub Action Formalisation, 57
64 Link Disappearence Formalisation, 59
2 Manifest Domain Endurants, 17
1 Manifest Domain Names, 17
3 Manifest Domain Perdurants, 17
16 Materials, 25
34 Mereology Update, 37
6 Narrative Description of Bank System Endurants, 18
8 Narrative Description of Bank System Perdurants, 19
52 No Pipeline Junk, 49
17 Parts Containing Materials, 25
45 Parts and Components, 44
47 Parts and Materials, 45
55 Parts, Attributes and Behaviours, 53
14 Parts, 25
5 Perdurant Entity Qualities, 18
49 Pipeline Material Flow, 46
48 Pipeline Material, 46
33 Pipeline Parts Mereology, 36
27 Pipeline Parts, 31
51 Pipelines: Inter Unit Flow and Leak Law, 48
50 Pipelines: Intra Unit Flow and Leak Law, 47
38 Road Hub Attributes, 39
56 Road Net Actions, 53
32 Road Net Part Mereologies, 35
57 Road Net and Road Traffic Events, 54
30 Shared Attribute Mereology, 34
42 Shared Attributes, 42
44 Shared Passbooks, 43
43 Shared Timetables, 42
63 Some Function Signatures, 58
67 Some Stake-holders, 74
53 States, 52
40 Static, Programmable and Inert Link Attributes, 40
31 Topological Connectedness Mereology, 34
10 Traffic System Endurants, 23
11 Traffic System Perdurants, 24
29 Unique Transportation Net Part Identifiers, 33
77 Trains Along Lines, 82–83
72 Probabilistic Rail Switch Unit State Transitions, 78
73 Railway Optical Gates, 78–79
74 Train Monitoring, I, 79
76 Trains at Stations, 82
69 Comparable Intrinsics, 76
70 Intrinsics of Switches, 76–77
81 A Human Behaviour Mortgage Calculation, 87
79 A Formally Described Bank Script, 85–86
78 A Casually Described Bank Script, 84–85
80 Banking — or Programming — Staff Behaviour, 86–87
75 Railway Management and Organisation: Train Monitoring, II, 80
68 Railway Net Intrinsics, 75–76
71 Railway Support Technology, 77–78

A Law of Train Traffic (# 66), 72
A Law of Train Traffic at Stations (# 126), 184
Actors (# 54), 53
Atomic Part Attributes (# 36), 38
Atomic Parts (# 18), 26
Attribute Propositions and Other Values (# 35), 38
Autonomous and Programmable Hub Attributes (# 41), 41

Bank System Channels (# 60), 55
Behaviours (# 58), 54
Bus System Channels (# 59), 55
Bus Timetable Coordination (# 65), 61–62

ch15fac.10 (# 77), 82–83
ch15fac.6 (# 72), 78
ch15fac.7 (# 73), 78–79
ch15fac.8 (# 74), 79
ch15fac.9 (# 76), 82
ch5-diff-models (# 69), 76
ch5-v-i (# 68), 75–76
ch5-v-i-2 (# 70), 76–77
ch5-v-ii-bank-f (# 81), 87
ch5-v-ii-ds (# 78), 84–85
ch5-v-ii-ds-f (# 79), 85–86
ch5-v-ii-hb (# 80), 86–87
ch5-v-ii-mao (# 75), 80
ch5-v-st (# 71), 77–78
Components (# 15), 25
Composite and Atomic Part Sorts of Transportation (# 20), 28–29
Composite Part Attributes (# 37), 38
Composite Parts (# 19), 26
Concrete Part Types of Transportation (# 21), 30
Container Components (# 46), 45
Container Line Sorts (# 22), 30–31
Continuous Endurants (# 13), 24

Derivation Chains (# 28), 32
Discrete Endurants (# 12), 24
Domain Requirements
  Determination
    Toll-roads (# 97), 160–161

Extension
Calculator Behaviour (# 106), 166
Gate Behaviour (# 105), 165–166
Global Values (# 102), 164
Main Sorts (# 101), 164
Parts, Properties and Channels (# 100), 163–164
System Behaviour (# 103), 164–165
Toll-road Net: Parts, Properties and Channels (# 99), 162–163
Vehicle Behaviour (# 104), 165
Vehicles: Parts, Properties and Channels (# 98), 161–162
Fitting
A Sketch (# 107), 168
Instantiation
Road Net, Abstraction (# 96), 159–160
Road Net, Formal Types (# 94), 157–158
Road Net, Narrative (# 93), 157
Road Net, Well-formedness (# 95), 158–159
Projection
A Narrative Sketch (# 85), 154
Attributes of Hubs (# 90), 155–156
Attributes of Links (# 91), 156
Behaviour (# 92), 156
Road Net Mereology (# 89), 155
Root Sorts (# 86), 155
Sub-domain Sorts and Types (# 87), 155
Unique Identifications (# 88), 155

Endurant Entity Qualities (# 4), 17

Flow in Pipelines (# 61), 56
Formal Description of Bank System Endurants (# 7), 18–19
Formal Description of Bank System Perdurants (# 9), 19

Inert and Reactive Attributes (# 39), 40
Insert Hub Action Formalisation (# 62), 57–58
Interface Requirements
Shared
Actions, Events and Behaviours (# 110), 172
Endurant Initialisation (# 109), 169–171
Endurants (# 108), 169

Link Disappearence Formalisation (# 64), 59

Machine Requirements
Documentation (# 125), 179
Road-pricing System
Accessibility (# 112), 174
Adaptive Maintenance (# 119), 177
Availability (# 113), 175
Corrective Maintenance (# 120), 177
Extensional Maintenance (# 123), 177
Integrity (# 114), 175
Perfective Maintenance (# 121), 177
Performance (# 111), 173
Platform Requirements (# 124), 178
Preventive Maintenance (# 122), 177
Reliability (# 115), 175
Robustness (# 118), 176
Safety (# 116), 175
Security (# 117), 176
Manifest Domain Endurants (# 2), 17
Manifest Domain Names (# 1), 17
Manifest Domain Perdurants (# 3), 17
Materials (# 16), 25
Mereology Update (# 34), 37

Narrative Description of Bank System Endurants (# 6), 18
Narrative Description of Bank System Perdurants (# 8), 19
No Pipeline Junk (# 52), 49–51

Parts (# 14), 25
Parts and Components (# 45), 44
Parts and Materials (# 47), 45
Parts Containing Materials (# 17), 25
Parts, Attributes and Behaviours (# 55), 53
Perdurant Entity Qualities (# 5), 18
Pipeline Material (# 48), 46
Pipeline Material Flow (# 49), 46–47
Pipeline Parts (# 27), 31–32
Pipeline Parts Mereology (# 33), 36
Pipelines: Inter Unit Flow and Leak Law (# 51), 48
Pipelines: Intra Unit Flow and Leak Law (# 50), 47–48

Requirements
The Problem/Objective
A Sketch (# 82), 153
The Road-pricing System
A Narrative (# 83), 153
The Road-pricing User and External Equipment
Narrative (# 84), 153–154
Road Hub Attributes (# 38), 39–40
Road Net Actions (# 56), 53
Road Net and Road Traffic Events (# 57), 54
Road Net Part Mereologies (# 32), 35

Shared Attribute Mereology (#30), 34
Shared Attributes (#42), 42
Shared Passbooks (#44), 43
Shared Timetables (#43), 42–43
Some Function Signatures (#63), 58
Some Stake-holders (#67), 74
States (#53), 52
Static, Programmable and Inert Link Attributes (#40), 40–41

Topological Connectedness Mereology (#31), 34
Traffic System Endurants (#10), 23
Traffic System Perdurants (#11), 24

Unique Transportation Net Part Identifiers (#29), 33–34

B.9 Concepts

[endurant]
  analysis prompts
    domain, 90
  description prompts
    domain, 90
  domain
    analysis prompts, 90
    description prompts, 90

description
  domain
    prompt, 29, 33, 39, 44, 45, 52, 71, 183
  prompt
    domain, 29, 33, 39, 44, 45, 52, 71, 183
domain
  description
    prompt, 29, 33, 39, 44, 45, 52, 71, 183
  prompt
    description, 29, 33, 39, 44, 45, 52, 71, 183
prompt
  description
    domain, 29, 33, 39, 44, 45, 52, 71, 183
  domain
    description, 29, 33, 39, 44, 45, 52, 71, 183

abstract
  value, 33
abstraction, 23
access
  attribute
    value, 42
  value
    attribute, 42
accessibility, 174
action, 19, 52
adaptive maintenance, 176
algorithmic
  engineering, 66
analyser
  domain, 16, 19
analysis
  domain, 16, 19, 22, 67–69
    prompt, 20, 23–27, 29, 34, 38, 44, 45, 52, 71, 183
  problem
    world, 68
  product line, 67
  prompt
    domain, 20, 23–27, 29, 34, 38, 44, 45, 52, 71, 183
  world
    problem, 68
analysis prompts
  [endurant]
    domain, 90
  domain
    [endurant], 90
architecture
  software, 68
atomic, 19
attribute, 56, 65
  access
    value, 42
  behaviour
    external, 56, 57
  constant, 56
  dynamic, 56
  external, 161, 162, 165
    behaviour, 56, 57
  programmable, 56, 57
  value
    access, 42
authorised user, 176
autonomous, 56
availability, 174
axiom
  sort

well-formedness, 48
well-formedness
sort, 48

bases
knowledge, 66
behaviour, 19, 52, 173
attribute
external, 56, 57
external
attribute, 56, 57
individual, 56
biddable, 56

change
state, 56
channel
external attribute, 56
class
diagram, 69
code, 179
commitments
ontological, 64
common
projection, 167
communication, 63
component
reusable
software, 67
software, 68
reusable, 67
composite, 19
composite, 140
computer
science, 17, 64–66
computing
science, 17, 66
conceive, 23
concept
formal, 22
concrete
prescription
requirements, 160
requirements
prescription, 160
concurrency, 63
confusion, 49
conservative
extension, 167
constant
attribute, 56
constructor
function
type, 57
type
function, 57
context, 22
continuant, 23
continuous, 19
time, 56
corrective maintenance, 176, 177
criminal human behaviour, 87

defined, 102
delinquent human behaviour, 86, 87
demo
domain, 16
demonstration platform
requirements, 178
demonstration platform requirements, 178
dependability, 173
attribute, 174
requirements, 172
tree, 173
describer, 16
domain, 16, 19
description
development
domain, 68
domain, 16, 19, 67–69
development, 68
prompt, 20, 35
path
tree, 100
prompt
domain, 20, 35
tree
path, 100
description prompts
[endurant]
domain, 90
domain
[endurant], 90
descriptions
domain, 68, 69
design
software, 16, 68, 69
determination, 151, 154
deterministic
prescription
requirements, 161
requirements

prescription, 161
development
description
domain, 68
document, 179
domain
description, 68
logbook, 179
model-oriented
software, 68
platform requirements, 178
requirements, 68, 69
software
model-oriented, 68
diagram
class, 69
diligent human behaviour, 86, 87
discrete, 19
documentation
requirements, 172, 179
domain, 69
[endurant]
analysis prompts, 90
description prompts, 90
analyser, 16, 19
analysis, 16, 19, 22, 67–69
prompt, 20, 23–27, 29, 34, 38, 44, 45, 52, 71, 183
analysis prompts
[endurant], 90
demo, 16
describer, 16, 19
description, 16, 19, 67–69, 179
development, 68
prompt, 20, 35
description prompts
[endurant], 90
descriptions, 68, 69
development
description, 68
engineer, 16, 19, 68
engineering, 16, 19, 66–68
facet, 74
language
specific, 67
modeling, 47, 67, 68
partial
requirement, 167
prescription
requirements, 154
prompt
analysis, 20, 23–27, 29, 34, 38, 44, 45, 52, 71, 183
description, 20, 35
requirement
partial, 167
shared, 167
requirements, 151, 154
prescription, 154
science, 19, 65, 66
scientist, 19
shared
requirement, 167
simulator, 16
software
specific, 16, 68
specific
language, 67
software, 16, 68
duplicate, 99
dynamic
attribute, 56
value, 42

endurant, 17, 19
engineer
domain, 16, 19, 68
requirements, 68
software, 19, 68
engineering
algorithmic, 66
domain, 16, 19, 66–68
knowledge, 66
ontological, 66
ontology, 64
product line
software, 67, 68
requirements, 16, 67, 69
software
product line, 67, 68
entities, 19
error, 173
event, 19, 52
execution platform requirements, 178
expression
function
type, 57
type, 57
function, 57
extended
prescription
requirements, 167

requirements
prescription, 167
extension, 151, 154
conservative, 167
extensional
maintenance, 176, 177
external
attribute, 161, 162, 165
behaviour, 56, 57
behaviour
attribute, 56, 57
part
quality, 32
quality
part, 32
external attribute
channel, 56

facet
domain, 74
failure, 173
fault, 173, 174
forecasting, 174
prevention, 174
removal, 174
tolerance, 174
fitting, 151, 154
formal
concept, 22
text, 21
formal concept analysis, 23
frame
problem, 68
frames
problem, 68
function
constructor
type, 57
expression
type, 57
name, 57
type
constructor, 57
expression, 57
functional
prescription
requirements, 153
requirements
prescription, 153

goal, 69
golden rule of requirements, 152

hardware, 68
has_concrete_type
prerequisite
prompt, 29
prompt
prerequisite, 29
has_mereology
prerequisite
prompt, 35
prompt
prerequisite, 35
head, 56
human behaviour
criminal, 87
delinquent, 86, 87
diligent, 86, 87
sloppy, 86, 87

ideal rule of requirements, 152
identifier
unique, 33, 65
imperative
language
programming, 66
programming
language, 66
individual
behaviour, 56
inert, 56
information
science, 64–66
installation
manual, 179
instantiation, 151, 154
integrity, 174, 175
interface
requirements, 151, 154
internal
part
quality, 32
qualities, 25, 33, 37
quality
part, 32
interval
time, 54
is_composite
prerequisite
prompt, 28
prompt

prerequisite, 28
is_ discrete
prerequisite
prompt, 27
prompt
prerequisite, 27
is_ entity
prerequisite
prompt, 23, 24
prompt
prerequisite, 23, 24

join
lattice, 32
junk, 48

knowledge, 66
bases, 66
engineering, 66
representation, 66

language
domain
specific, 67
imperative
programming, 66
programming
imperative, 66
specific
domain, 67
lattice
join, 32

machine, 68
=hardware+software, 152
requirements, 151, 154, 172
maintenance
adaptive, 176
corrective, 176, 177
extensional, 176, 177
logbook, 179
manual, 179
perfective, 176, 177
preventive, 176, 177
requirements, 172, 176
maintenance platform
requirements, 178
manifest
phenomena, 17
manual
installation, 179
maintenance, 179
training, 179
user, 179
mereology, 65
observer, 34
type, 34
methodology, 16
model-oriented
development
software, 68
software
development, 68
modeling
domain, 47, 67, 68
requirements, 47

name
function, 57
narrative
requirements
system, 153
user and external equipment, 153
system
requirements, 153
text, 21
user and external equipment
requirements, 153
non-manifest
qualities, 17

obligation
proof, 49
observe, 23
observe_ part_ type
prerequisite
prompt, 29
prompt
prerequisite, 29
observer
mereology, 34
occurrent, 23
ontological
commitments, 64
engineering, 66
ontology
engineering, 64
science, 64
upper, 64–66

parallelism, 63
part, 26
external
quality, 32

internal
quality, 32
quality
external, 32
internal, 32
sort, 27
partial
domain
requirement, 167
requirement
domain, 167
path
description
tree, 100
tree
description, 100
perdurant, 17, 19
perfective maintenance, 176, 177
performance requirements, 172
phenomena
manifest, 17
philosophy, 64
platform requirements, 172, 178
demonstration, 178
development, 178
execution, 178
maintenance, 178
prerequisite
has_ concrete_ type
prompt, 29
has_ mereology
prompt, 35
is_ composite
prompt, 28
is_ discrete
prompt, 27
is_ entity
prompt, 23, 24
observe_ part_ type
prompt, 29
prompt
has_ concrete_ type, 29
has_ mereology, 35
is_ composite, 28
is_ discrete, 27
is_ entity, 23, 24
observe_ part_ type, 29
prescription
concrete
requirements, 160
deterministic
requirements, 161
domain
requirements, 154
extended
requirements, 167
functional
requirements, 153
projected
requirements, 156
requirements, 16, 67–69, 151
concrete, 160
deterministic, 161
domain, 154
extended, 167
functional, 153
projected, 156
preventive maintenance, 176, 177
problem
analysis
world, 68
frame, 68
frames, 68
world, 68
analysis, 68
process
schema, 70
product line
analysis, 67
engineering
software, 67, 68
software, 68
engineering, 67, 68
programmable
attribute, 56, 57
programming
imperative
language, 66
language
imperative, 66
projected
prescription
requirements, 156
requirements
prescription, 156
projection, 151, 154
common, 167
specific, 167
prompt, 20–21
analysis
domain, 20, 23–27, 29, 34, 38, 44, 45, 52, 71, 183

description
domain, 20, 35
domain
analysis, 20, 23–27, 29, 34, 38, 44, 45, 52, 71, 183
description, 20, 35
has_ concrete_ type
prerequisite, 29
has_ mereology
prerequisite, 35
is_ composite
prerequisite, 28
is_ discrete
prerequisite, 27
is_ entity
prerequisite, 23, 24
observe_ part_ type
prerequisite, 29
prerequisite
has_ concrete_ type, 29
has_ mereology, 35
is_ composite, 28
is_ discrete, 27
is_ entity, 23, 24
observe_ part_ type, 29
proof
obligation, 49

qualities, 19
internal, 25, 33, 37
non-manifest, 17
quality, 65
external
part, 32
internal
part, 32
part
external, 32
internal, 32

reactive, 56
redefined, 102
reliability, 174, 175
representation
knowledge, 66
requirement
domain
partial, 167
shared, 167
partial
domain, 167
shared
domain, 167
requirements, 68, 69
concrete
prescription, 160
demonstration platform, 178
dependability, 172
deterministic
prescription, 161
development, 68, 69
platform, 178
documentation, 172, 179
domain, 151, 154
prescription, 154
engineer, 68
engineering, 16, 67, 69
execution platform, 178
extended
prescription, 167
functional
prescription, 153
golden rule, 152
ideal rule, 152
interface, 151, 154
machine, 151, 154, 172
maintenance, 172, 176
platform, 178
modeling, 47
narrative
system, 153
user and external equipment, 153
performance, 172
platform, 172, 178
prescription, 16, 67–69, 151, 179
concrete, 160
deterministic, 161
domain, 154
extended, 167
functional, 153
projected, 156
projected
prescription, 156
sketch
system, 153
user, 153
system
narrative, 153
sketch, 153
user
sketch, 153
user and external equipment

narrative, 153
reusable
component
software, 67
software
component, 67
reuse, 67
robustness, 174, 176

safety, 174, 175
schema
process, 70
science
computer, 17, 64–66
computing, 17, 66
domain, 19, 65, 66
information, 64–66
ontology, 64
scientist
domain, 19
security, 174, 176
shared
domain
requirement, 167
requirement
domain, 167
sharing, 33
signature, 52, 66
simulator
domain, 16
sketch
requirements
system, 153
user, 153
system
requirements, 153
user
requirements, 153
sloppy human behaviour, 86, 87
software, 68
architecture, 68
component, 68
reusable, 67
design, 16, 68, 69, 179
development
model-oriented, 68
domain
specific, 16, 68
engineer, 19, 68
engineering
product line, 67, 68
model-oriented
development, 68
product line, 68
engineering, 67, 68
reusable
component, 67
specific
domain, 16, 68
sort, 22
axiom
well-formedness, 48
part, 27
well-formedness
axiom, 48
specific
domain
language, 67
software, 16, 68
language
domain, 67
projection, 167
software
domain, 16, 68
stake-holder, 74
state, 52
change, 56
static, 56
value, 42
sub-part, 26
support
document, 179
synchronisation, 63
system
narrative
requirements, 153
requirements
narrative, 153
sketch, 153
sketch
requirements, 153

test
document, 179
text
formal, 21
narrative, 21
time, 52, 54
continuous, 56
interval, 54
training manual, 179
tree

description
path, 100
path
description, 100
TripTych, 15, 16, 22, 37, 64–70, 91, 181, 186
type, 22
constructor
function, 57
expression, 57
function, 57
function
constructor, 57
expression, 57
mereology, 34

unauthorised user, 176
Unified Modeling Language
UML, 69
old.UML, 69
unique
identifier, 33, 65
upper
ontology, 64–66
user
authorised, 176
manual, 179
requirements
sketch, 153
sketch
requirements, 153
unauthorised, 176
user and external equipment
narrative
requirements, 153
requirements
narrative, 153

validation
document, 179
value
abstract, 33
access
attribute, 42
attribute
access, 42
dynamic, 42
static, 42
verification
document, 179

well-formedness
axiom
sort, 48
sort
axiom, 48
world
analysis
problem, 68
problem, 68
analysis, 68

B.10 RSL Language Constructs

Arithmetics
...,-2,-1,0,1,2,..., 190
a_i*a_j , 193
a_i+a_j , 193
a_i/a_j , 193
a_i=a_j , 192
a_i≥a_j , 192
a_i>a_j , 192
a_i≤a_j , 192
a_i<a_j , 192
a_i≠a_j , 192
a_i−a_j , 193
Cartesians
(e_1,e_2,...,e_n) , 193
Chaos
chaos, 196, 197
Clauses
... **elsif** ... , 203
case b_e **of** pa_1 → c_1, ... pa_n → c_n **end** , 203
if b_e **then** c_c **else** c_a **end** , 202
Combinators
let a:A • P(a) **in** c **end** , 202
let pa = e **in** c **end** , 201
Functions
f(args) **as** result, 201
post P(args,result), 201
pre P(args), 201
f(a), 200
f(args) ≡ expr, 201
Imperative
case b_e **of** pa_1 → c_1, ... pa_n → c_n **end** , 204
do stmt **until** b*e* **end** , 204

for e **in** list$_{expr}$ • P(b) **do** stm(e) **end** , 204
if b$_e$ **then** c$_c$ **else** c$_a$ **end** , 204
skip , 204
variable v:Type := expression , 204
while be **do** stm **end** , 204
f(), 203
stm$_1$;stm$_2$;...;stm$_n$; , 204
v := expression , 204

Lists
<Q(l(i))|i **in**<1..**len**l> •P(a)> , 194
hAB, 193
ℓ(i) , 196
⟨ei ..ej ⟩, 193
$\langle e_1, e_2, ..., e_n B$, 193
elems ℓ , 196
hd ℓ , 196
inds ℓ , 196
len ℓ , 196
tl ℓ , 196

Logics
$b_i \vee b_j$, 192
∀ a:A • P(a) , 192
∃! a:A • P(a) , 192
∃ a:A • P(a) , 192
∼ b , 192
false, 190, 192
true, 190, 192
a_i=a_j , 193
$a_i \geq a_j$, 193
$a_i > a_j$, 193
$a_i \leq a_j$, 193
$a_i < a_j$, 193
$a_i \neq a_j$, 193
$b_i \Rightarrow b_j$, 192
$b_i \wedge b_j$, 192

Maps
[F(e)↦G(m(e))|e:E•e∈**dom** m∧P(e)] , 194
[ ] , 194
[u$_1$↦v$_1$,u$_2$↦v$_2$,...,u$_n$↦v$_n$] , 194
$m_i \setminus m_j$, 198
$m_i \circ m_j$, 198
m_i / m_j , 198
dom m , 198
rng m , 198
$m_i = mj$, 198
$m_i \cup m_j$, 198
$m_i \dagger m_j$, 198
$m_i \neq mj$, 198
m(e) , 198

Processes
channel c:T , 204
channel {k[i]:T•i:KIdx} , 204
c ! e , 205
c ? , 205
k[i] ! e , 205
k[i] ? , 205
P⫿Q, 204
P⫲ Q, 204
P: **Unit** → **in** c **out** k[i] **Unit** , 205
P[]Q, 204
P‖ Q, 204
Q: i:KIdx → **out** c **in** k[i] **Unit**, 205

Sets
{Q(a)|a:A•a∈s∧P(a)} , 193
{} , 193
$\{e_1, e_2, ..., e_n\}$, 193
∩{s$_1$,s$_2$,...,s$_n$} , 194
∪{s$_1$,s$_2$,...,s$_n$} , 194
card s , 195
e∈s , 194
e∉s , 194
s_i=s_j , 195
$s_i \cap s_j$, 194
$s_i \cup s_j$, 194
$s_i \subset s_j$, 195
$s_i \subseteq s_j$, 195
$s_i \neq s_j$, 195
$s_i \setminus s_j$, 194

Types
(T$_1$×T$_2$×... ×T$_n$), 190
T*, 190
T$^\omega$, 190
T$_1$ × T$_2$ × ... × T$_n$, 189
Bool, 189
Char, 189
Int, 189
Nat, 189
Real, 189
Text, 189
Unit, 203, 205
mk_ id(s$_1$:T$_1$,s$_2$:T$_2$,...,s$_n$:T$_n$), 190
s$_1$:T$_1$ s$_2$:T$_2$ ... s$_n$:T$_n$, 190
T = Type_ Expr, 191
T$_1$ | T$_2$ | ... | T$_1$ | T$_n$, 190
T={| v:T′• P(v)|} , 191, 192
T==TE$_1$ | TE$_2$ | ... | TE$_n$, 191
Ti⇀Tj, 190
Ti→Tj, 190
T-**infset**, 189
T-**set**, 189