Domain Science & Engineering

A Prerequisite for Requirements Engineering

ASSC Melbourne Tutorial Notes, May 28, 2014

Dines Bjørner
Fredsvej 11, DK–2840 Holte, Denmark
bjorner@gmail.com, www.imm.dtu.dk/~dibj

April 8, 2014: 12:17

Abstract

These notes cover a **new science & engineering of domains** as well as a **new foundation for software development**. We treat the latter first. Instead of commencing with requirements engineering, whose pursuit may involve repeated, but unstructured forms of domain analysis, we propose a predecessor phase of domain engineering.

That is, we single out domain analysis as an activity to be pursued prior to requirements engineering. In emphasising domain engineering as a predecessor phase we, at the same time, introduce **a number of facets** that are **not present,** we think, **in current software engineering** studies and practices.

(i) **One facet** is **the construction of separate domain descriptions.** Domain descriptions are void of any reference to requirements and encompass the modelling of domain phenomena without regard to their being computable.

(ii) **Another facet** is **the pursuit of domain descriptions as a free-standing activity.** In these notes we emphasize domain description development need not necessarily lead to software development. This gives a new meaning to business process engineering, and should lead to a deeper understanding of a domain and to possible non-IT related business process re-engineering of areas of that domain. In these notes we shall investigate a method for analysing domains, for constructing domain descriptions and some emerging scientific bases. We shall also, less detailed, cover basic principles, techniques and tools for "deriving" requirements from domain descriptions.

Our contribution to **domain analysis** is that we view domain analysis as a variant of formal concept analysis [27], a contribution which can be formulated by the "catch phrase" **domain entities and their qualities form Galois connections**, and further contribute with a methodology of necessary corresponding principles and techniques of domain analysis. Those corresponding principles and techniques hinge on our view of domains as having the following **ontology**. There are the entities that we can describe and then there is "the rest" which we leave un-described. We analyse entities into endurant entities and perdurant entities , that is, parts and materials as endurant entities and discrete actions, discrete events and behaviours as perdurant entities , respectively. Another way of looking at entities is as discrete entities , or as continuous entities. We also contribute to the analysis of discrete endurants in terms of the following notions: part types and material types, part unique identifiers, part mereology and part attributes and material attributes and **material laws**. Of the above we point to the introduction, into computing science and software engineering of the notions of materials and continuous behaviours **as novel.**

The example formalisations are expressed in **RAISE** [29] (with [5, 6, 7] being a rather comprehensive monograph cum textbook), but could as well have been expressed in, for

example, **Alloy** [36], **Event B** [1] , **VDM** [15, 16, 24] or **Z** [68].

Contents

1 Introduction **8**
1.1 Domains: Some Definitions . . . 8
Example 1: Some Domains . . . 8
1.1.1 Domain Analysis . . . 8
Example 2: A Container Line Analysis . . . 8
1.1.2 Domain Descriptions . . . 8
Example 3: A Transport Domain Description . . . 8
1.1.3 Domain Engineering . . . 9
1.1.4 Domain Science . . . 9
1.2 The Triptych of Software Development . . . 9
1.3 Issues of Domain Science & Engineering . . . 10
1.4 Structure of Lecture Notes . . . 11

2 The Main Example: Road Traffic System **11**
Example 4: The Main Example . . . 11
2.1 Parts . . . 11
2.1.1 Root Sorts . . . 11
2.1.2 Sub-domain Sorts and Types . . . 12
2.1.3 Further Sub-domain Sorts and Types . . . 13
2.2 Properties . . . 13
2.2.1 Unique Identifications . . . 13
2.2.2 Mereology . . . 14
[1] Road Net Mereology: . . . 14
[2] Fleet of Vehicles Mereology: . . . 14
2.2.3 Attributes . . . 15
[1] Attributes of Links: . . . 15
[2] Attributes of Hubs: . . . 15
[3] Attributes of Vehicles: . . . 16
[4] Vehicle Positions: . . . 17
2.3 Definitions of Auxiliary Functions . . . 18
2.4 Some Derived Traffic System Concepts . . . 18
2.4.1 Maps . . . 18
2.4.2 Traffic Routes . . . 19
[1] Circular Routes: . . . 20
[2] Connected Road Nets: . . . 21
[3] Set of Connected Nets of a Net: . . . 21
[4] Route Length: . . . 21
[5] Shortest Routes: . . . 22
2.5 States . . . 23
2.6 Actions . . . 23
2.7 Events . . . 24
2.8 Behaviours . . . 25
2.8.1 Traffic . . . 25
[1] Continuous Traffic: . . . 25
[2] Discrete Traffic: . . . 25
[3] Time: An Aside: . . . 26
2.8.2 Globally Observable Parts . . . 26
2.8.3 Road Traffic System Behaviours . . . 27
2.8.4 Channels . . . 27
2.8.5 Behaviour Signatures . . . 28
2.8.6 The Vehicle Behaviour . . . 28
2.8.7 The Monitor Behaviour . . . 29

3 Domains 30
[1] Domain: 30
[2] Domain Phenomena: 30
[3] Domain Entity: 30
[4] Endurant Entity: 30
[5] Perdurant Entity: 30
[6] Discrete Endurant: 31
[7] Continuous Endurant: 31
[8] Domain Parts and Materials: 31
[9] Domain Analysis: 31
[10] Domain Description: 31
[11] Domain Engineering: 31
[12] Domain Science: 31
[13] Values & Types: 31
[14] Discrete Perdurant: 31
[15] Continuous Perdurant: 31
[16] Extensionality: 32
[17] Intentionality: 32

4 Discrete Endurant Entities 32
4.1 Parts – Syntactic Aspects 33
4.1.1 What is a Part ? 33
Example 5: Parts 33
4.1.2 Classes of "Same Kind" Parts 33
Example 6: Part Properties 33
4.1.3 A Preview of Part Properties 33
4.1.4 An Analysis Process: Endurants 33
4.1.5 Part Property Values 34
Example 7: Part Property Values 34
Example 8: Distinct Parts 34
4.1.6 Part Sorts 34
Example 9: Part Sorts 34
4.1.7 Atomic Parts 34
Example 10: Atomic Types 34
4.1.8 Composite Parts 34
Example 11: Composite Types 35
4.1.9 Part Observers 35
Example 12: Implementation of Observer Functions 35
Example 13: Observer Functions 35
4.1.10 Part Types 35
Example 14: Concrete Types 35
Example 15: Has Composite Types 35
4.2 Part Properties 36
Example 16: Property Value Scales 36
4.2.1 Unique Identifiers 36
Example 17: Unique Identifier Functions 37
[1] A Dogma of Unique Existence: 37
[2] A Simplification on Specification of Intentional Properties: 37
[3] Discussion: 37
[4] The uid_P Operator: 37
[5] Constancy of Unique Identifiers — Some Dogmas: 37
4.2.2 Mereology 37
Example 18: Manifest and Conceptual Parts 38
[1] Extensional and Intentional Part Relations: 38
Example 19: Shared Route Maps and Bus Time Tables 38
Example 20: Monitor and Vehicle Mereologies 39
[2] Unique Part Identifier Mereologies: 39
Example 21: Road Traffic System Mereology 39

Example 22: Pipeline Mereology . . . 39
[3] Concrete Part Type Mereologies: . . . 40
Example 23: A Container Line Mereology . . . 40
[4] Variability of Mereologies: . . . 42
Example 24: Insert Link . . . 42
4.2.3 Attributes . . . 43
Example 25: Road Transport System Part Attributes . . . 43
[1] Stages of Attribute Analysis: . . . 43
Example 26: Static and Dynamic Attributes . . . 44
Example 27: Concrete Attribute Types . . . 44
[2] The attr_A Operator: . . . 44
[3] Variability of Attributes: . . . 44
Example 28: Setting Road Intersection Traffic Lights . . . 44
4.2.4 Properties of Parts . . . 45
4.3 States . . . 45
Example 29: A Variety of Road Traffic Domain States . . . 45
4.4 An Example Domain: Pipelines . . . 45
Example 30: Pipeline Units and Their Mereology . . . 45
Example 31: Pipelines: Nets and Routes . . . 46

5 Discrete Perdurant Entities 48
5.1 On Domain Analysis: Discrete Perdurants . . . 49
5.2 Actions . . . 49
Example 32: Transport Net and Container Vessel Actions . . . 49
5.2.1 Abstraction: On Modelling Domain Actions . . . 49
5.2.2 Agents: An Aside on Actions . . . 49
5.2.3 Action Signatures . . . 50
Example 33: Action Signatures: Nets and Vessels . . . 50
5.2.4 Action Definitions . . . 50
Example 34: Transport Nets Actions . . . 50
Example 35: Container Line: Remove Container . . . 50
Modelling Actions . . . 51
5.3 Events . . . 52
Example 36: Events . . . 52
5.3.1 An Aside on Events . . . 52
5.3.2 Event Signatures . . . 53
5.3.3 Event Definitions . . . 53
Example 37: Road Transport System Event . . . 53
Modelling Events . . . 53
5.4 Discrete Behaviours . . . 54
5.4.1 What is Meant by 'Behaviour'? . . . 54
5.4.2 Behaviour Narratives . . . 54
5.4.3 Channels . . . 54
5.4.4 Behaviour Signatures . . . 55
5.4.5 Behaviour Definitions . . . 55
[1] Atomic Part Behaviours: . . . 56
Example 38: Atomic Part Behaviours . . . 56
[2] Composite Part Behaviours: . . . 56
Example 39: Compositional Behaviours . . . 56
5.4.6 A Model of Parts and Behaviours . . . 56
Example 40: Syntax and Semantics of Mereology . . . 57
[1] A Syntactic Model of Parts: . . . 57
[2] A Semantics Model of Parts: . . . 58

6 Continuous Entities 60
6.1 Materials . . . 60
Example 41: Materials . . . 60
6.1.1 Materials-based Domains . . . 60
Example 42: Material Processing . . . 61
6.1.2 "Somehow Related" Parts and Materials . . . 61
Example 43: Somehow Related Materials and Parts . . . 61
6.1.3 Material Observers . . . 61
Example 44: Pipelines: Core Continuous Endurant . . . 61
Example 45: Pipelines: Parts and Materials . . . 62
6.1.4 Material Properties . . . 62
Example 46: Pipelines: Parts and Material Properties . . . 63
6.1.5 Material Laws of Flows and Leaks . . . 64
Example 47: Pipelines: Intra Unit Flow and Leak Law . . . 64
Example 48: Pipelines: Inter Unit Flow and Leak Law . . . 65
6.2 Continuous Behaviours . . . 65
6.2.1 Fluid Dynamics . . . 65
[1] Descriptions of Continuous Domain Behaviours: . . . 66
[2] Prescriptions of Required Continuous Domain Behaviours: . . . 66
Example 49: Pipelines: Fluid Dynamics and Automatic Control . . . 66
6.2.2 A Pipeline System Behaviour . . . 66
Example 50: A Pipeline System Behaviour . . . 66

7 Requirements Engineering 69
7.1 A Requirements "Derivation" . . . 69
7.1.1 Definition of Requirements . . . 69
IEEE Definition of 'Requirements' . . . 69
7.1.2 The Machine = Hardware + Software . . . 69
7.1.3 Requirements Prescription . . . 69
7.1.4 Some Requirements Principles . . . 69
The "Golden Rule" of Requirements Engineering . . . 69
An "Ideal Rule" of Requirements Engineering . . . 70
7.1.5 A Decomposition of Requirements Prescription . . . 70
7.1.6 An Aside on Our Example . . . 70
7.2 Domain Requirements . . . 70
7.2.1 Projection . . . 71
7.2.2 Instantiation . . . 71
[1] Model Well-formedness wrt. Instantiation:: . . . 72
7.2.3 Determination . . . 72
[1] Model Well-formedness wrt. Determination:: . . . 72
7.2.4 Extension . . . 74
Backgorund: . . . 74
The Extension: . . . 74
The Formalisation: . . . 75
7.3 Interface Requirements Prescription . . . 76
7.3.1 Shared Parts . . . 77
[1] Data Initialisation:: . . . 77
[2] Data Refreshment:: . . . 77
7.3.2 Shared Actions . . . 78
[1] Interactive Action Execution:: . . . 78
7.3.3 Shared Events . . . 78
7.3.4 Shared Behaviours . . . 78
7.4 Machine Requirements . . . 78
7.5 Discussion of Requirements "Derivation" . . . 78

8 Conclusion **79**
8.1 What Have We Achieved . . . 79
8.2 General Remarks . . . 79
8.3 Acknowledgements . . . 80

9 Bibliography **80**
9.1 References . . . 80

A A TripTychTripTych@TripTych Ontology **86**

B On A Theory of Container Stowage **87**
B.1 Some Pictures . . . 87
B.2 Parts . . . 88
B.2.1 A Basis . . . 88
B.2.2 Mereological Constraints . . . 89
B.2.3 Stack Indexes . . . 90
B.2.4 Stowage Schemas . . . 92
B.3 Actions . . . 93
B.3.1 Remove Container from Vessel . . . 93
B.3.2 Remove Container from CTP . . . 94
B.3.3 Stack Container on Vessel . . . 95
B.3.4 Stack Container in CTP . . . 95
B.3.5 Transfer Container from Vessel to CTP . . . 95
B.3.6 Transfer Container from CTP to Vessel . . . 96

C Indexes **97**
C.1 RSL Index . . . 97
C.2 Formalisation Index . . . 98
C.3 Definition Index . . . 100
C.4 Example Index . . . 101
C.5 Concept Index . . . 103
C.6 Language, Method and Technology Index . . . 118
C.7 Selected Author Index . . . 118

D RSL: The Raise Specification Language **120**
D.1 Type Expressions . . . 120
D.1.1 Atomic Types . . . 120
D.1.2 Composite Types . . . 120
[1] Concrete Composite Types: . . . 120
[2] Sorts and Observer Functions: . . . 121
D.2 Type Definitions . . . 122
D.2.1 Concrete Types . . . 122
D.2.2 Subtypes . . . 123
D.2.3 Sorts — Abstract Types . . . 123
D.3 The RSL Predicate Calculus . . . 123
D.3.1 Propositional Expressions . . . 123
D.3.2 Simple Predicate Expressions . . . 123
D.3.3 Quantified Expressions . . . 124
D.4 Concrete RSL Types: Values and Operations . . . 124
D.4.1 Arithmetic . . . 124
D.4.2 Set Expressions . . . 124
[1] Set Enumerations: . . . 124
[2] Set Comprehension: . . . 124
D.4.3 Cartesian Expressions . . . 125
[1] Cartesian Enumerations: . . . 125
D.4.4 List Expressions . . . 125
[1] List Enumerations: . . . 125
[2] List Comprehension: . . . 125

D.4.5 Map Expressions . . . 125
[1] Map Enumerations: . . . 125
[2] Map Comprehension: . . . 126
D.4.6 Set Operations . . . 126
[1] Set Operator Signatures: . . . 126
[2] Set Examples: . . . 126
[3] Informal Explication: . . . 127
[4] Set Operator Definitions: . . . 127
D.4.7 Cartesian Operations . . . 127
D.4.8 List Operations . . . 128
[1] List Operator Signatures: . . . 128
[2] List Operation Examples: . . . 128
[3] Informal Explication: . . . 128
[4] List Operator Definitions: . . . 129
D.4.9 Map Operations . . . 130
[1] Map Operator Signatures and Map Operation Examples: . . . 130
[2] Map Operation Explication: . . . 130
[3] Map Operation Redefinitions: . . . 131
D.5 λ-Calculus + Functions . . . 131
D.5.1 The λ-Calculus Syntax . . . 131
D.5.2 Free and Bound Variables . . . 132
D.5.3 Substitution . . . 132
D.5.4 α-Renaming and β-Reduction . . . 132
D.5.5 Function Signatures . . . 132
D.5.6 Function Definitions . . . 133
D.6 Other Applicative Expressions . . . 133
D.6.1 Simple let Expressions . . . 133
D.6.2 Recursive let Expressions . . . 133
D.6.3 Predicative let Expressions . . . 134
D.6.4 Pattern and "Wild Card" let Expressions . . . 134
D.6.5 Conditionals . . . 134
D.6.6 Operator/Operand Expressions . . . 135
D.7 Imperative Constructs . . . 135
D.7.1 Statements and State Changes . . . 135
D.7.2 Variables and Assignment . . . 136
D.7.3 Statement Sequences and skip . . . 136
D.7.4 Imperative Conditionals . . . 136
D.7.5 Iterative Conditionals . . . 136
D.7.6 Iterative Sequencing . . . 136
D.8 Process Constructs . . . 136
D.8.1 Process Channels . . . 136
D.8.2 Process Composition . . . 137
D.8.3 Input/Output Events . . . 137
D.8.4 Process Definitions . . . 137
D.9 Simple RSL Specifications . . . 137

1 Introduction

We beg the reader to re-read the **abstract**, Page 1, as for the **contribution** of these notes.

This is primarily a methodology set of lecture notes. By a method$_\delta$ we shall understand a set of **principles** for **selecting** and **applying** a number of **techniques** and **tools** in order to **analyse** a **problem** and **construct** an **artefact**. By methodology$_\delta$ we shall understand the study and knowledge about methods.

These notes contributes to the study and knowledge of software engineering development methods. Its contributions are those of suggesting and exploring domain engineering and domain engineering as a basis for requirements engineering. We are not saying *"thou must develop software this way"*, but we do suggest that since it is possible and makes sense to do so it may also be wise to do so.

1.1 Domains: Some Definitions

By a domain$_\delta$ we shall here understand an area of human activity characterised by observable phenomena: entities whether endurants (manifest parts and materials) or perdurants (actions, events or behaviours), whether discrete or continuous; and of their properties.

Example: 1 Some Domains Some examples are:

air traffic,	fish industry,	securities trading,
airport,	health care,	transportation
banking,	logistics,	etcetera. ■
consumer market,	manufacturing,	
container lines,	pipelines,	

1.1.1 Domain Analysis

By domain analysis$_\delta$ we shall understand an inquiry into the domain, its entities and their properties.

Example: 2 A Container Line Analysis. *parts:* container, vessel, terminal port, etc.; *actions:* container loading, container unloading, vessel arrival in port, etc.; *events:* container falling overboard; container afire; etc.; *behaviour:* vessel voyage, across the seas, visiting ports, etc. Length of a container is a container *property*. Name of a vessel is a vessel *property*. Location of a container terminal port is a port *property*.

1.1.2 Domain Descriptions

By a domain description$_\delta$ we shall understand a narrative description tightly coupled (say line-number-by-line-number) to a formal description. To develop a domain description requires a thorough amount of domain analysis.

Example: 3 A Transport Domain Description.

- *Narrative:*

⋙ a transport net, n:N,
consists of an aggregation of hubs, hs:HS,
which we "concretise" as a set of hubs, H-**set**, and
an aggregation of links, ls:LS, that is, a set L-**set**,

- *Formalisation:*

⋙ **type** N, HS, LS, Hs = H-**set**, Ls = L-**set**, H, L
value
obs_HS: N→HS,
obs_LS: N→LS.
obs_Hs: HS→H-**set**,
obs_Ls: LS→L-**set**. ■

An interesting domain description is usually a document of a hundred pages or so. Each page "listing" pairs of enumerated informal, i.e., narrative descriptions with formal descriptions.

1.1.3 Domain Engineering

By domain engineering$_\delta$ we shall understand the engineering of a domain description, that is, the rigorous construction of domain descriptions, and the further analysis of these, creating theories of domains. The size (usually, say a hundred pages), structure (usually a finely sectioned document of may subsub···subsections) and complexity (having many cross-references between subsub···subsections) of interesting domain descriptions is usually such as to put a special emphasis on engineering: the management and organisation of several, typically 5–6 collaborating domain describers, the ongoing check of description quality, completeness and consistency, etcetera.

1.1.4 Domain Science

By domain science$_\delta$ we shall understand two things: the general study and knowledge of how to create and handle domain descriptions (a general theory of domain descriptions) and the specific study and knowledge of a particular domain. The two studies intertwine.

1.2 The `Triptych` of Software Development

We suggest a "dogma": before software can be designed one must understand[1] the requirements; and before requirements can be expressed one must understand[2] the domain.

We can therefore view software development as ideally proceeding in three (i.e., `TripTych`) phases: an initial phase of domain engineering, followed by a phase of requirements engineering, ended by a phase of software design.

In the domain engineering phase[3] ($\mathcal{D}$) a domain is analysed, described and "theorised", that is, the beginnings of a specific domain theory is established. In the requirements engineering phase[4] ($\mathcal{R}$) a requirements prescription is constructed — significant fragments of which are "derived", systematically, from the domain description. In the software design phase[5] ($\mathcal{S}$)

[1] Or maybe just: have a reasonably firm grasp of
[2] See previous footnote!
[3] See Sects. 4–6
[4] See Sect. 7
[5] We do not illustrate the software design phase in this paper.

a software design is derived, systematically, rigorously or formally, from the requirements prescription. Finally the $\mathcal{S}$oftware is proven correct with respect to the $\mathcal{R}$equirements under assumption of the $\mathcal{D}$omain: $\mathcal{D}, \mathcal{S} \models \mathcal{R}$.

By a machine$_\delta$ we shall understand the hardware and software[6] of a target, i.e., a required IT system.

In [8, 14, 11] we indicate how one can "derive" significant parts of requirements from a suitably comprehensive domain description – basically as follows. Domain projection: from a domain description one projects those areas that are to be somehow manifested in the software. Domain initialisation: for that resulting projected requirements prescription one initialises a number of part types as well as action and behaviour definitions, from less abstract to more concrete, specific types, respectively definitions. Domain determination: hand-in-hand with domain initialisation a[n interleaved] stage of making values of types less non-deterministic, i.e., more deterministic, can take place. Domain extension: Requirements often arise in the context of new business processes or technologies either placing old or replacing human processes in the domain. Domain extension is now the 'enrichment' of the domain requirements, so far developed, with the description of these new business processes or technologies. Etcetera. The result of this part of "requirements derivation" is the domain requirements.

A set of domain-to-requirements operators similarly exists for constructing interface requirements from the domain description and, independently, also from knowledge of the machine for which the required IT system is to be developed. We illustrate the techniques of domain requirements and interface requirements in Sect. 7.

Finally machine requirements are "derived" from just the knowledge of the machine, that is, the target hardware and the software system tools for that hardware. Since the domain does not "appear" in the construction of the machine requirements we shall not illustrate that aspect of requirements prescription in Sect. 7. When you review this section ('A Triptych of Software Development') then you will observe how 'the domain' predicates both the requirements and the software design. For a specific domain one may develop many (thus related) requirements and from each such (set of) requirements one may develop many software designs. We may characterise this multitude of domain-predicated requirements and designs as a product line [12]. You may also characterise domain-specific developments as representing another 'definition' of domain engineering.

1.3 Issues of Domain Science & Engineering

We specifically focus on the following issues of domain science &[7] engineering: (i) which are the "things" to be described[8], (ii) how to analyse these "things" into constituent description structures[9], (iii) how to describe these "things" informally and formally, (iv) how to further structure descriptions[10], and a further study of (v) mereology[11].

[6] By software$_\delta$ we shall understand all the development documentation, from domain descriptions via requirements prescriptions to software design; all verification data: the formal tests, model checkings and proofs; the development contracts, the management plans, the budgets and accounts; the staffing plans; the installation manuals, the user manuals, the (perfective, adaptive, corrective, etc.) maintenance manuals, and the development methodology manuals; as well as all the software development tools used in the actual development.

[7] When we put '&' between two terms that the compound term forms a whole concept.

[8] endurants [manifest entities henceforth called parts and materials] and perdurants [actions, events, behaviours]

[9] atomic and composite, unique identifiers, mereology, attributes

[10] *intrinsics, support technology, rules & regulations, organisation & management, human behaviour* etc.

[11] the study and knowledge of parts and relations of parts to other parts and a "whole".

1.4 Structure of Lecture Notes

First, Sect. 1, we introduce the problem. And that was done above.

We start with an Example, Sect. 2. The example is that of a domain of road traffic.

Then, in Sects. 3–6 we bring a rather careful analysis of the concept of the observable, manifest phenomena that we shall refer to as entities. We strongly think that these sections (of these notes) brings, to our taste, a simple and elegant reformulation of what is usually called *"data modelling"*, in this case for domains — but with major aspects applicable as well to requirements development and software design. That analysis focuses on endurant entities, also called parts and materials, those that can be observed at no matter what time, i.e., entities of substance or continuant, and perdurant entities: action, event and behaviour entities, those that occur, that happen, that, in a sense, are accidents. **We think** that this "decomposition" of the "data analysis" problem into discrete parts and continuous materials, atomic and composite parts, their unique identifiers and mereology, and their attributes **is novel**, and differs from past practices in domain analysis.

In Sect. 7 we bring a brief survey of the kind of requirements engineering that one can now pursue based on a reasonably comprehensive domain description. We show how one can systematically, but not automatically "derive" significant fragments of requirements prescriptions from domain descriptions.

• • •

The formal descriptions will here be expressed in the **RAISE** [29] Specification Language, RSL. We otherwise refer to [5]. Appendix D brings a short primer, mostly on the syntactic aspects of RSL. But other model-oriented formal specification languages can be used with equal success; for example: **Alloy** [36], **Event B** [1] , **VDM** [15, 16, 24] and **Z** [68].

2 The Main Example: Road Traffic System

Example: 4 The Main Example. The main example presents a terse narrative and formalisation of a road traffic domain. Since the example description conceptually covers also major aspects of railroad nets, shipping nets, and air traffic nets, we shall use such terms as hubs and links to stand for road (or street) intersection and road (or street) segments, train stations and rail lines, harbours and shipping lanes, and airports and air lanes.

2.1 Parts

2.1.1 Root Sorts

The domain, the stepwise unfolding of whose description is to be exemplified, is that of a composite traffic system (i) with a road net, (ii) with a fleet of vehicles (iii) of whose individual position on the road net we can speak, that is, monitor.

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

type
1. Δ
1a. N
1b. F
1c. M
value
1a. **obs_**N: Δ → N
1b. **obs_**F: Δ → F
1c. **obs_**M: Δ → M

2.1.2 Sub-domain Sorts and Types

2. From the road net we can observe
   [a] a composite part, HS, of road (i.e., street) intersections (hubs) and
   [b] an composite part, LS, of road (i.e., street) segments (links).

type
2. HS, LS
value
2a. **obs_**HS: N → HS
2b. **obs_**LS: N → LS

3. From the fleet sub-domain, F, we observe a composite part, VS, of vehicles

type
3. VS
value
3. **obs_**VS: F → VS

4. From the composite sub-domain VS we observe
   [a] the composite part Vs, which we concretise as a set of vehicles
   [b] where vehicles, V, are considered atomic.

type
4a. Vs = V-**set**
4b. V
value
4a. **obs_**Vs: VS → V-**set**

The "monitor" is considered atomic; it is an abstraction of the fact that we can speak of the positions of each and every vehicle on the net without assuming that we can indeed pin point these positions by means of for example sensors.

2.1.3 Further Sub-domain Sorts and Types

We now analyse the sub-domains of HS and LS.

5. From the hubs aggregate we decide to observe
   [a] the concrete type of a set of hubs,
   [b] where hubs are considered atomic; and
6. from the links aggregate we decide to observe
   [a] the concrete type of a set of links,
   [b] where links are considered atomic;

type
5a. Hs = H-**set**
6a. Ls = L-**set**
5b. H
6b. L
value
5. **obs_**Hs: HS → H-**set**
6. **obs_**Ls: LS → L-**set**

We have no composite parts left to further analyse into parts whether they be again composite or atomic. That is, at various, what we shall refer to as, domain indexes we have discovered the following part types:

- $\langle\Delta\rangle$: N, F, M
- $\langle\Delta, N\rangle$: HS, LS
- $\langle\Delta, F\rangle$: VS
- $\langle\Delta, HS\rangle$: Hs, H
- $\langle\Delta, LS\rangle$: Ls, L
- $\langle\Delta, VS\rangle$: Vs, V

Thus we have ended up with atomic parts.

2.2 Properties

Parts are distinguished by their properties: the types and the values of these. We consider three kinds of properties: unique identifiers, mereology and attributes.

2.2.1 Unique Identifications

There is, for any traffic system, exactly one composite aggregation, HS, of hubs, exactly one composite aggregation, Hs, of hubs, exactly one composite aggregation, LS, of links, exactly one composite aggregation, Ls, of links, exactly one composite aggregation, VS, of vehicles and exactly one composite aggregation, Vs, of vehicles, Therefore we shall not need to associate unique identifiers with any of these.

7. We decide the following:
   [a] each hub has a unique hub identifier,

[b] each link has a unique link identifier and

[c] each vehicle has a unique vehicle identifier.

type
7a. HI
7b. LI
7c. VI
value
7a. **uid_**H: H → HI
7b. **uid_**L: L → LI
7c. **uid_**V: V → VI

2.2.2 Mereology

[1] Road Net Mereology: By *mereology* we mean the study, knowledge and practice of understanding parts and part relations.

The relations between, that is, the mereology of, the composite parts of the road net, n:N, are simple: there is one HS part of n:N; there is one Hs part of the only HS part of n:N; there is one LS part of n:N; and there is one Ls part of the only LS part of n:N. Therefore we shall not associate any special mereology based on unique identifiers which we therefore also decided to not express for these composite parts.

8. Each link is connected to exactly two hubs, that is,
   [a] from each link we can observe its mereology, that is, the identities of these two distinct hubs,
   [b] and these hubs must be of the net of the link;
9. and each hub is connected to zero, one or more links, that is,
   [a] from each hub we can observe its mereology, that is, the identities of these links,
   [b] and these links must be of the net of the hub.

value
8a. **mereo_**L: L → HI-**set**, **axiom** ∀ l:L•**card** **mereo_**L(l)=2
axiom
8b. ∀ n:N,l:L,hi:HI • l ∈ **obs_**Ls(**obs_**LS(n)) ∧ hi ∈ **mereo_**L(l)
8b. ⇒ ∃ h:H•h ∈ **obs_**Hs(**obs_**HS(n))∧**uid_**H(h)=hi
value
9a. **mereo_**H: H → LI-**set**
axiom
9b. ∀ n:N,h:H,li:LI • h ∈ **obs_**Hs(**obs_**HS(n)) ∧ li ∈ **mereo_**H(h)
9b. ⇒ ∃ l:L•l ∈ **obs_**Ls(**obs_**LS(n))∧**uid_**L(l)=li

[2] Fleet of Vehicles Mereology: We shall omit treatment of mereology of vehicles.

In the traffic system that we are building up there are no relations to be expressed between vehicles, only between vehicles and the (single and only) monitor.

2.2.3 Attributes

We shall model attributes of links, hubs and vehicles. The composite parts, aggregations of hubs, HS and Hs, aggregations of links, LS and Ls and aggregations of vehicles, VS and Vs, also have attributes, but we shall omit modelling them here.

[1] Attributes of Links:

10. The following are attributes of links.

    [a] Link states, $l\sigma$:$L\Sigma$, which we model as possibly empty sets of pairs of distinct identifiers of the connected hubs. A link state expresses the directions that are open to traffic across a link.

    [b] Link state spaces, $l\omega$:$L\Omega$ which we model as the set of link states. A link state space expresses the states that a link may attain across time.

    [c] Further link attributes are length, location, etcetera.

Link states are usually dynamic attributes whereas link state spaces, link length and link location (usually some curvature rendition) are considered static attributes.

type
10a. $L\Sigma = (HI \times HI)$**-set**
axiom
10a. $\forall$ $l\sigma$:$L\Sigma$ • $0 \leq$ **card** $l\sigma \leq 2$
value
10a. **attr_**$L\Sigma$: $L \rightarrow L\Sigma$
axiom
10a. $\forall$ l:L • **let** {hi,hi′}=**mereo_**L(l) **in** **attr_**$L\Sigma$(l)⊆{(hi,hi′),(hi′,hi)} **end**
type
10b. $L\Omega = L\Sigma$**-set**
value
10b. **attr_**$L\Omega$: $L \rightarrow L\Omega$
axiom
10b. $\forall$ l:L • **let** {hi,hi′}=**mereo_**L(l) **in** **attr_**$L\Sigma$(l)∈ **attr_**$L\Omega$(l) **end**
type
10c. LOC, LEN, ...
value
10c. **attr_**LOC: $L \rightarrow$ LOC, **attr_**LEN: $L \rightarrow$ LEN, ...

[2] Attributes of Hubs:

11. The following are attributes of hubs:

    [a] Hub states, $h\sigma$:$H\Sigma$, which we model as possibly empty sets of pairs of identifiers of the connected links. A hub state expresses the directions that are open to traffic across a hub.

    [b] Hub state spaces, $h\omega$:$H\Omega$ which we model as the set of hub states. A hub state space expresses the states that a hub may attain across time.

[c] Further hub attributes are location, etcetera.

Hub states are usually dynamic attributes whereas hub state spaces and hub location are considered static attributes.

type
11a. HΣ = (LI × LI)**-set**
value
11a. **attr_**HΣ: H → HΣ
axiom
11a. ∀ h:H • **attr_**HΣ(h)⊆{(li,li′)|li,li′:LI•{li,li′}⊆**mereo_**H(h)}
type
11b. HΩ = HΣ**-set**
value
11b. **attr_**HΩ: H → HΩ
axiom
11b. ∀ h:H • **attr_**HΣ(h) ∈ **attr_**HΩ(h)
type
11c. LOC, ...
value
11c. **attr_**LOC: L → LOC, ...

[3] Attributes of Vehicles:

12. Dynamic attributes of vehicles include
    [a] position
        i. at a hub (about to enter the hub — referred to by the link it is coming from, the hub it is at and the link it is going to, all referred to by their unique identifiers or
        ii. some fraction "down" a link (moving in the direction from a from hub to a to hub — referred to by their unique identifiers)
        iii. where we model fraction as a real between 0 and 1 included.
    [b] velocity, acceleration, etcetera.
13. All these vehicle attributes can be observed.

type
12a. VP = atH | onL
12(a)i. atH :: fli:LI × hi:HI × tli:LI
12(a)ii. onL :: fhi:HI × li:LI × frac:FRAC × thi:HI
12(a)iii. FRAC = **Real**, **axiom** ∀ frac:FRAC • 0 ≤ frac ≤ 1
12b. VEL, ACC, ...
value
13. **attr_**VP:V→VP, **attr_**onL:V→onL, **attr_**atH:V→atH
13. **attr_**VEL:V→VEL, **attr_**ACC:V→ACC

[4] Vehicle Positions:

14. Given a net, n:N, we can define the possibly infinite set of potential vehicle positions on that net, vps(n).
    [a] vps(n) is expressed in terms of the links and hubs of the net.
    [b] vps(n) is the
    [c] union of two sets:
        i. the potentially[12] infinite set of "on link" positions
        ii. for all links of the net

        and
        i. the finite set of "at hub" positions
        ii. for all hubs in the net.

```
value
14.   vps: N → VP-infset
14b.  vps(n) ≡
14a.      let ls=obs_Ls(obs_LS(n)), hs=obs_Hs(obs_HS(n)) in
14(c)i.     { onL(fhi,uid(l),f,thi) | fhi,thi:HI,l:L,f:FRAC •
14(c)ii.          l ∈ ls ∧ {fhi,thi}=mereo_L(l) }
14c.      ∪
14(c)i.     { atH(fli,uid_H(h),tli) | fli,tli:LI,h:H •
14(c)ii.          h ∈ hs ∧ {fli,tli}⊆mereo_H(h) }
14a.      end
```

Given a net and a finite set of vehicles we can distribute these over the net, i.e., assign initial vehicle positions, so that no two vehicles "occupy" the same position, i.e., are "crashed" ! Let us call the non-deterministic assignment function, i.e., a relation, for vpr.

15. vpm:VPM is a bijective map from vehicle identifiers to (distinct) vehicle positions.
16. vpr has the obvious signature.
17. vpr(vs)(n) is defined in terms of
18. a non-deterministic selection, vpa, of vehicle positions, and
19. a non-deterministic assignment of these vehicle positions to vehicle identifiers —
20. being the resulting distribution.

```
type
15.   VPM′ = VI →m VP
15.   VPM = {| vpm:VPM′ • card dom vpm = card rng vpm |}
value
```

[12] The 'potentiality' arises from the nature of FRAC. If fractions are chosen as, for example, 1/5'th, 2/5'th, ..., 4/5'th, then there are only a finite number of "on link" vehicle positions. If instead fraction are arbitrary infinitesimal quantities, then there are infinitely many such.

```
16.  vpr: V-set × N → VMP
17.  vpr(vs)(n) ≡
18.     let vpa:VP-set • vpa ⊆ vps(vs)(n) ∧ card vpa = vard vs in
19.     let vpm:VPM • dom vpm = vps ∧ rng vpm = vpa in
20.     vpm end end
```

2.3 Definitions of Auxiliary Functions

21. From a net we can extract all its link identifiers.

22. From a net we can extract all its hub identifiers.

```
value
21.  xtr_LIs: N → LI-set
21.  xtr_LIs(n) ≡ {uid_L(l)|l:L•l ∈ obs_Ls(obs_LS(n))}
22.  xtr_HIs: N → HI-set
22.  xtr_HIs(n) ≡ {uid_H(l)|h:H•h ∈ obs_Hs(obs_HS(n))}
```

23. Given a link identifier and a net get the link with that identifier in the net.

24. Given a hub identifier and a net get the hub with that identifier in the net.

```
value
26.   get_H: HI → N ~→ H
26.   get_H(hi)(n) ≡ ι h:H•h ∈ obs_Hs(obs_HS(n))∧uid_H(h)=hi
26.      pre: hi ∈ xtr_HIs(n)
26a.  get_L: LI → N ~→ L
26a.  get_L(li)(n) ≡ ι l:L•l ∈ obs_Ls(obs_LS(n))∧uid_L(l)=li
26a.     pre: hl ∈ xtr_LIs(n)
```

The ι a:A•$\mathcal{P}$(a) expression yields the unique value a:A which satisfies the predicate $\mathcal{P}$(a). If none, or more than one exists then the function is undefined.

2.4 Some Derived Traffic System Concepts

2.4.1 Maps

25. A road map is an abstraction of a road net. We define one model of maps below.

    [a] A road map, RM, is a finite definition set function, M, (a specification language map) from
    - hub identifiers (the source hub)
    - to (such finite definition set) functions from link identifiers
    - to hub identifiers (the target hub).

```
type
25a.  RM′ = HI →m (LI →m HI)
```

If a hub identifier in the source or an rm:RM maps into the empty map then the "corresponding" hub is "isolated": has no links emanating from it.

26. These road maps are subject to a well-formedness criterion.
    [a] The target hubs must be defined also as source hubs.
    [b] If a link is defined from source hub (referred to by its identifier) shi via link li to a target hub thi, then, vice versa, link li is also defined from source thi to target shi.

```
type
26.   RM = {| rm:RM′ • wf_RM(rm) |}
value
26.   wf_RM: RM′ → Bool
26.   wf_RM(rm) ≡
26a.      ∪ { rng(rm(hi))|hi:HI•hi ∈ dom rm } ⊆ dom rm
26b.    ∧ ∀ shi:HI•shi ∈ dom rm ⇒
26b.        ∀ li:LI • li ∈ dom rm(shi) ⇒
26b.            li ∈ dom rm((rm(shi))(li)) ∧ (rm((rm(shi))(li)))(li)=shi
```

27. Given a road net, n, one can derive "its" road map.
    [a] Let hs and ls be the hubs and links, respectively of the net n.
    [b] Every hub with no links emanating from it is mapped into the empty map.
    [c] For every link identifier uid_L(l) of links, l, of ls and every hub identifier, hi, in the mereology of l
    [d] hi is mapped into a map from uid_L(l) into hi'
    [e] where hi' is the other hub identifier of the mereology of l.

```
value
27.   derive_RM: N → RM
27.   derive_RM(n) ≡
27a.      let hs = obs_Hs(obs_HS(n)), ls = obs_Ls(obs_LS(n)) in
27b.      [ hi ↦ [ ] | hi:HI • ∃ h:H • h ∈ hs ∧ mereo_H(h) = {} ] ∪
27d.      [ hi ↦ [ uid_L(l) ↦ hi′
27e.                          | hi′:HI • hi′ = mereo_L(l)\{hi} ]
27c.          | l:L,hi:HI • l ∈ ls ∧ hi ∈ mereo_L(l) ] end
```

Theorem: If the road net, n, is well-formed then wf_RM(derive_RM(n)).

2.4.2 Traffic Routes

28. A traffic route, tr, is an alternating sequence of hub and link identifiers such that
    [a] li:LI is in the mereology of the hub, h:H, identified by hi:HI, the predecessor of li:LI in route r, and
    [b] hi':HI, which follows li:LI in route r, is different from hi, and is in the mereology of the link identified by li.

```
type
28.   R′ = (HI|LI)*
28.   R = {| r:R′ • ∃ n:N • wf_R(r)(n) |}
value
28.   wf_R: R′ → N → Bool
28.   wf_R(r)(n) ≡
28.     ∀ i:Nat • {i,i+1}⊆inds r ⇒
28a.        is_HI(r(i)) ⇒ is_LI(r(i+1)) ∧ r(i+1) ∈ mereo_H(get_H(r(i))(n)),
28b.        is_LI(r(i)) ⇒ is_HI(r(i+1)) ∧  r(i+1) ∈ mereo_L(get_L(r(i))(n))
```

29. From a well-formed road map (i.e., a road net) we can generate the possibly infinite set of all routes through the net.
    [a] **Basis Clauses:**
        i. The empty sequence of identifiers is a route.
        ii. The one element sequences of link and hub identifiers of links and hubs of a road map (i.e., a road net) are routes.
        iii. If hi maps into some li in rm then ⟨hi,li⟩ and ⟨li,hi⟩ are routes of the road map (i.e., of the road net).
    [b] **Induction Clause:**
        i. Let r⌢⟨i⟩ and ⟨i′⟩⌢r′ be two routes of the road map.
        ii. If the identifiers i and i′ are identical, then r⌢⟨i⟩⌢r′ is a route.
    [c] **Extremal Clause:**
        i. Only such routes that can be formed from a finite number of applications of the above clauses are routes.

```
value
29.   gen_routes: M → Routes-infset
29.   gen_routes(m) ≡
29(a)i.      let rs = {⟨⟩}
29(a)ii.           ∪ {⟨li,hi⟩,⟨hi,li⟩|li:LI,hi:HI•...}
29(b)i.            ∪ {let r⌢⟨li⟩,⟨li′⟩⌢r′:R • {r⌢⟨li⟩,⟨li′⟩⌢r′}⊆rs,
29(b)i.                  r″⌢⟨hi⟩,⟨hi′⟩⌢r‴:R • {r″⌢⟨hi⟩,⟨hi′⟩⌢r‴}⊆rs in
29(b)ii.               r⌢⟨li⟩⌢r′,r″⌢⟨hi⟩⌢r‴ end} in
29(c)i.      rs end
```

[1] Circular Routes:

30. A route is circular if the same identifier occurs more than once.

```
value
30.   is_circular_route: R → Bool
30.   is_circular_route(r) ≡ ∃ i,j:Nat • {i,j}⊆inds r ∧ i≠j ⇒ r(i)=r(j)
```

[2] Connected Road Nets:

31. A road net is connected if there is a route from any hub (or any link) to any other hub or link in the net.

```
31.  is_conn_N: N → Bool
31.  is_conn_N(n) ≡
31.      let m = derive_RM(n) in
31.      let rs = gen_routes(m) in
31.      ∀ i,i′:(LI|HI) • {i,i′}⊆xtr_LIs(n)∪ xtr_HIs(n)
31.         ∃ r:R • r ∈ rs ∧ r(1)=i ∧ r(len r)=i′ end end
```

[3] Set of Connected Nets of a Net:

32. The set, cns, of connected nets of a net, n, is
    [a] the smallest set of connected nets, cns,
    [b] whose hubs and links together "span" those of the net n.

```
value
32.  conn_Ns: N → N-set
32.  conn_Ns(n) as cns
32a.     pre: true
32b.     post: conn_spans_HsLs(n)(cns)
32a.           ∧ ∼∃ kns:N-set • card kns < card cns
32a.              ∧ conn_spans_HsLs(n)(kns)
```

```
32b.  conn_spans_HsLs: N → N → Bool
32b.  conn_spans_HsLs(n)(cns) ≡
32b.      ∀ cn:N•cn ∈ cns ⇒ is_connected_N(n)(cn)
32b.    ∧ let (hs,ls) = (obs_Hs(obs_HS(n)),obs_Ls(obs_LS(n))),
32b.          chs = ∪{obs_Hs(obs_HS(cn))|cn ∈ cns},
32b.          cls = ∪{obs_Ls(obs_LS(cn))|cn ∈ cns} in
32b.      hs = chs ∧ ls = cls end
```

[4] Route Length:

33. The length attributes of links can be
    [a] added and subtracted,
    [b] multiplied by reals to obtain lengths,
    [c] divided to obtain fractions,
    [d] compared as to whether one is shorter than another, etc., and
    [e] there is a "zero length" designator.

value
33a. $+,- : \text{LEN} \times \text{LEN} \rightarrow \text{LEN}$
33b. $* : \text{LEN} \times \textbf{Real} \rightarrow \text{LEN}$
33c. $/ : \text{LEN} \times \text{LEN} \rightarrow \textbf{Real}$
33d. $<,\leq,=,\neq,\geq,> : \text{LEN} \times \text{LEN} \rightarrow \textbf{Bool}$
33e. $\ell_0 : \text{LEN}$

34. One can calculate the length of a route.

value
34. length: R → N → LEN
34. length(r)(n) ≡
34. **case** r **of**:
34. ⟨⟩ → ℓ_0,
34. ⟨si⟩⌢r′ →
34. is_LI(si)→**attr_**LEN(get_L(si)(n))+length(r′)(n)
34. is_HI(si)→length(r′)(n)
34. **end**

[5] Shortest Routes:

35. There is a predicate, is_R, which,

[a] given a net and two distinct hub identifiers of the net,

[b] tests whether there is a route between these.

value
35. is_R: N → (HI×HI) → **Bool**
35. is_R(n)(fhi,thi) ≡
35a. fhi ≠ thi ∧ {fht,thi}⊆xtr_HIs(n)
35b. ∧ ∃ r:R • r ∈ routes(n) ∧ **hd** r = fhi ∧ r(**len** r) = thi

36. The shortest between two given hub identifiers

[a] is an acyclic route, r,

[b] whose first and last elements are the two given hub identifiers

[c] and such that there is no route, r′ which is shorter.

value
36. shortest_route: N → (HI×HI) → R
36a. shortest_route(n)(fhi,thi) **as** r
36b. **pre**: pre_shortest_route(n)(fhi,thi)
36c. **post**: pos__shortest_route(n)(r)(fhi,thi)

36b. pre_shortest_route: N → (HI×HI) → **Bool**
36b. pre_shortest_route(n)(fhi,thi) ≡
36b. is_R(n)(fhi,thi) ∧ fhi≠thi ∧ {fhi,thi}⊂xtr_HIs(n)

36c. pos_shortest_route: N → R → (HI×HI) → **Bool**
36c. pos_shortest_route(n)(r)(fhi,thi) ≡
36c. r ∈ routes(n)
36c. ∧ ∼∃ r′:R • r′ ∈ routes(n) ∧ length(r′) < length(r)

2.5 States

There are different notions of state. In our example these are some of the states: the road net composition of hubs and links; the state of a link, or a hub; and the vehicle position.

2.6 Actions

An action is what happens when a function invocation changes, or potentially changes a state. Examples of traffic system actions are: insertion of hubs, insertion of links, removal of hubs, removal of links, setting of hub state (hσ), setting of link state (lσ), moving a vehicle along a link, moving a vehicle from a link to a hub and moving a vehicle from a hub to a link.

37. The insert action applies to a net and a hub and conditionally yields an updated net.

    [a] The condition is that there must not be a hub in the "argument" net with the same unique hub identifier as that of the hub to be inserted and

    [b] the hub to be inserted does not initially designate links with which it is to be connected.

    [c] The updated net contains all the hubs of the initial net "plus" the new hub.

    [d] and the same links.

value
37. ins_H: N → H $\stackrel{\sim}{\to}$ N
37. ins_H(n)(h) **as** n′, **pre**: pre_ins_H(n)(h), **post**: post_ins_H(n)(h)

37a. pre_ins_H(n)(h) ≡
37a. ∼∃ h′:H • h′ ∈ **obs_**Hs(n) ∧ **uid_**HI(h)=**uid_**HI(h′)
37b. ∧ **mereo_**H(h) = {}

37c. post_ins_H(n)(h)(n′) ≡
37c. **obs_**Hs(n) ∪ {h} = **obs_**Hs(n′)
37d. ∧ **obs_**Ls(n) = **obs_**Ls(n′)

2.7 Events

By an event we understand a state change resulting indirectly from an unexpected application of a function, that is, that function was performed "surreptitiously". Events can be characterised by a pair of (before and after) states, a predicate over these and, optionally, a time or time interval. Events are thus like actions: change states, but are usually either caused by "previous" actions, or caused by "an outside action".

38. Link disappearance is expressed as a predicate on the "before" and "after" states of the net. The predicate identifies the "missing" ℓink (!).

39. Before the disappearance of link ℓ in net n
    [a] the hubs h' and h'' connected to link ℓ
    [b] were connected to links identified by $\{l'_1, l'_2, \ldots, l'_p\}$ respectively $\{l''_1, l''_2, \ldots, l''_q\}$
    [c] where, for example, l'_i, l''_j are the same and equal to $\mathsf{uid\_\Pi}(\ell)$.

38. link_dis: N × N → **Bool**
38. link_dis(n,n′) ≡
38. ∃ ℓ:L • pre_link_dis(n,ℓ) ⇒ post_link_dis(n,ℓ,n′)
39. pre_link_dis: N × L → **Bool**
39. pre_link_dis(n,ℓ) ≡ ℓ ∈ **obs_**Ls(n)

40. After link ℓ disappearance there are instead
    [a] two separate links, ℓ_i and ℓ_j, "truncations" of ℓ
    [b] and two new hubs h''' and h''''
    [c] such that ℓ_i connects h' and h''' and
    [d] ℓ_j connects h'' and h'''';
    [e] Existing hubs h' and h'' now have mereology
        i. $\{l'_1, l'_2, \ldots, l'_p\} \setminus \{\mathsf{uid\_\Pi}(\ell)\} \cup \{\mathsf{uid\_\Pi}(\ell_i)\}$ respectively
        ii. $\{l''_1, l''_2, \ldots, l''_q\} \setminus \{\mathsf{uid\_\Pi}(\ell)\} \cup \{\mathsf{uid\_\Pi}(\ell_j)\}$

41. All other hubs and links of n are unaffected.

42. We shall "explain" *link disappearance* as the combined, instantaneous effect of
    [a] first a remove link "event" where the removed link connected hubs hi_j and hi_k;
    [b] then the insertion of two new, "fresh" hubs, h_α and h_β;
    [c] "followed" by the insertion of two new, "fresh" links $\mathsf{l}_{j\alpha}$ and $\mathsf{l}_{k\beta}$ such that
        i. $\mathsf{l}_{j\alpha}$ connects hi_j and h_α and
        ii. $\mathsf{l}_{k\beta}$ connects hi_k and $\mathsf{h}_{k\beta}$

value
42. post_link_dis(n,ℓ,n$'$) $\equiv$
42. **let** h_a,h_b:H •
42. **let** {li_a,li_b}=**mereo_**L(ℓ) **in**
42. (get_H(li_a)(n),get_H(li_b)(n)) **end in**
42a. **let** n$''$ = rem_L(n)(**uid_**L(ℓ)) **in**
42b. **let** h_α,h_β:H • {h_α,h_β}$\cap$**obs_**Hs(n)={} **in**
42b. **let** n$'''$ = ins_H(n$''$)(h_α) **in**
42b. **let** n$''''$ = ins_H(n$'''$)(h_β) **in**
42c. **let** $\mathrm{l}_{j\alpha}$,$\mathrm{l}_{k\beta}$:L • {$\mathrm{l}_{j\alpha}$,$\mathrm{l}_{k\beta}$}$\cap$**obs_**Ls(n)={}
42c. $\wedge$ **mereo_**L($\mathrm{l}_{j\alpha}$) = {**uid_**H(h_a),**uid_**H(h_α)}
42c. $\wedge$ **mereo_**L($\mathrm{l}_{k\beta}$) = {**uid_**H(h_b),**uid_**H(h_β)} **in**
42(c)i. **let** n$'''''$ = ins_L(n$''''$)($\mathrm{l}_{j\alpha}$) **in**
42(c)ii. n$'$ = ins_L(n$'''''$)($\mathrm{l}_{k\beta}$) **end end end end end end end**

2.8 Behaviours

2.8.1 Traffic

[1] Continuous Traffic: For the road traffic system perhaps the most significant example of a behaviour is that of its traffic

43. the continuous time varying discrete positions of vehicles, vp:VP[13],
44. where time is taken as a dense set of points.

type
44. c$\mathbb{T}$
43. cRTF = c$\mathbb{T}$ $\rightarrow$ (V $\vec{m}$ VP)

[2] Discrete Traffic: We shall model, not continuous time varying traffic, but

45. discrete time varying discrete positions of vehicles,
46. where time can be considered a set of linearly ordered points.

46. d$\mathbb{T}$
45. dRTF = d$\mathbb{T}$ $\vec{m}$ (V $\vec{m}$ VP)

47. The road traffic that we shall model is, however, of vehicles referred to by their unique identifiers.

type
47. RTF = d$\mathbb{T}$ $\vec{m}$ (VI $\vec{m}$ VP)

[13] For VP see Item 12a on Page 16.

[3] Time: An Aside: We shall take a rather simplistic view of time [17, 45, 53, 63].

48. We consider d$\mathbb{T}$, or just $\mathbb{T}$, to stand for a totally ordered set of time points.
49. And we consider $\mathbb{TI}$ to stand for time intervals based on $\mathbb{T}$.
50. We postulate an infinitesimal small time interval δ.
51. $\mathbb{T}$, in our presentation, has lower and upper bounds.
52. We can compare times and we can compare time intervals.
53. And there are a number of "arithmetics-like" operations on times and time intervals.

type
48. $\mathbb{T}$
49. $\mathbb{TI}$
value
50. δ:$\mathbb{TI}$
51. $\mathbb{MIN}, \mathbb{MAX}$: $\mathbb{T} \rightarrow \mathbb{T}$
51. $<,\leq,=,\geq,>$: $(\mathbb{T}\times\mathbb{T})|(\mathbb{TI}\times\mathbb{TI}) \rightarrow$ **Bool**
52. $-$: $\mathbb{T}\times\mathbb{T} \rightarrow \mathbb{TI}$
53. $+$: $\mathbb{T}\times\mathbb{TI},\mathbb{TI}\times\mathbb{T} \rightarrow \mathbb{T}$
53. $-,+$: $\mathbb{TI}\times\mathbb{TI} \rightarrow \mathbb{TI}$
53. $*$: $\mathbb{TI}\times$**Real** $\rightarrow \mathbb{TI}$
53. $/$: $\mathbb{TI}\times\mathbb{TI} \rightarrow$ **Real**

54. We postulate a global clock behaviour which offers the current time.
55. We declare a channel clk_ch.

value
54. clock: $\mathbb{T} \rightarrow$ **out** clk_ch **Unit**
54. clock(t) $\equiv$ ... clk_ch!t ... clock(t $\sqcap$ t+δ)
channnel
55. clk_ch:$\mathbb{T}$

2.8.2 Globally Observable Parts

There is given

56. a net, n:N,
57. a set of vehicles, vs:V-**set**, and
58. a monitor, m:M.

The n:N, vs:V-**set** and m:M are observable from the road traffic system domain.

value
56. n:N = **obs_**N(Δ)
56. ls:L-**set** = **obs_**Ls(**obs_**LS(n)), hs:H-**set** = **obs_**Hs(**obs_**HS(n)),
56. lis:LI-**set** = {**uid_**L(l)|l:L•l ∈ ls}, his:HI-**set** = {**uid_**H(h)|h:H•h ∈ hs}
57. vs:V-**set** = **obs_**Vs(**obs_**VS(**obs_**F(Δ))), vis:V-**set** = {**uid_**V(v)|v:V•v ∈ vs}
58. m:**obs_**M(Δ)

2.8.3 Road Traffic System Behaviours

59. 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, based on

    [c] the monitor and its unique identifier,

    [d] an initial vehicle position map, and

    [e] an initial starting time.

value
59c. mi:MI = **uid_**(m)
59d. vpm:VPM = vpr(vs)(n)
59e. t_0:T = clk_ch?

59. rts() =
59a. ‖ {veh(**uid_**V(v))(v)(vpm(**uid_**V(v)))|v:V•v ∈ vs}
59b. ‖ mon(mi)(m)([t_0 ↦ vpm])

where the "extra" monitor argument records the discrete road traffic, RTF, initially set to the singleton map from an initial start time, t_0 to the initial assignment of vehicle positions.

2.8.4 Channels

In order for the monitor behaviour to assess the vehicle positions these vehicles communicate their positions to the monitor via a vehicle to monitor channel. In order for the monitor to time-stamp these positions it must be able to "read" a clock.

60. Thus we declare a set of channels indexed by the unique identifiers of vehicles and communicating vehicle positions.

channel
60. {vm_ch[mi,vi]|vi:VI•vi ∈ vis}:VP

2.8.5 Behaviour Signatures

61. The road traffic system behaviour, rts, takes no arguments (hence the first **Unit**); and "behaves", that is, continues forever (hence the last **Unit**).

62. The vehicle behaviours are indexed by the unique identifier, uid_V(v):VI, the vehicle part, v:V and the vehicle position; offers communication to the monitor behaviour (on channel vm_ch[vi]); and behaves "forever".

63. The monitor behaviour takes the so far unexplained monitor part, m:M, as one argument and the discrete road traffic, drtf:dRTF, being repeatedly "updated" as the result of **in**put communications from (all) vehicles; the behaviour otherwise runs forever.

value
61. rts: **Unit** → **Unit**
62. veh: vi:VI → v:V → VP → **out** vm_ch[vi],mi:MI **Unit**
63. mon: mi:MI → m:M → dRTF → **in** {vm_ch[mi,vi]|vi:VI•vi ∈ vis},clk_ch **Unit**

2.8.6 The Vehicle Behaviour

64. A vehicle process is indexed by the unique vehicle identifier vi:VI, the vehicle "as such", v:V and the vehicle position, vp:VPos.

    The vehicle process communicates with the monitor process on channel vm[vi] (sends, but receives no messages), and otherwise evolves "in[de]finitely" (hence **Unit**).

65. We describe here an abstraction of the vehicle behaviour at a Hub (hi).

    [a] Either the vehicle remains at that hub informing the monitor,

    [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], that it is now on the link identified by tli,

        iii. whereupon the vehicle resumes the vehicle behaviour positioned at the very beginning (0) of that link,

    [c] or, again internally non-deterministically,

    [d] the vehicle "disappears — off the radar" !

```
65.      veh(vi)(v)(vp:atH(fli,hi,tli)) ≡
65a.           vm_ch[ mi,vi ]!vp ; veh(vi)(v)(vp)
65b.         ⌈⌉
65(b)i.        let {hi′,thi}=mereo_L(get_L(tli)(n)) in assert: hi′=hi
65(b)ii.       vm_ch[ mi,vi ]!onL(tli,hi,0,thi) ;
65(b)iii.      veh(vi)(v)(onL(tli,hi,0,thi)) end
65c.         ⌈⌉
65d.           stop
```

66. 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,
    [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 δ 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.
67. or, internally non-deterministically,
68. the vehicle "disappears — off the radar" !

```
64.    veh(vi)(v)(vp:onL(fhi,li,f,thi)) ≡
66a.        vm_ch[mi,vi]!vp ; veh(vi)(v)(vp)
66b.    ⌈⌉
66c.        if f + δ<1
66(c)i.         then vm_ch[mi,vi]!onL(fhi,li,f+δ,thi) ;
66(c)i.              veh(vi)(v)(onL(fhi,li,f+δ,thi))
66(c)ii.        else let li′:LI•li′ ∈ mereo_H(get_H(thi)(n)) in
66(c)iiA.            vm_ch[mi,vi]!atH(li,thi,li′);
66(c)iiB.            veh(vi)(v)(atH(li,thi,li′)) end end
67.     ⌈⌉
68.         stop
```

2.8.7 The Monitor Behaviour

69. The monitor behaviour evolves around the attributes of an own "state", m:M, a table of traces of vehicle positions, while accepting messages about vehicle positions and otherwise progressing "in[de]finitely".
70. Either the monitor "does own work"
71. 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,
    [c] whereupon the monitor resumes its behaviour —
    [d] where the communicating vehicles range over all identified vehicles.

```
69.   mon(mi)(m)(rtf) ≡
70.       mon(mi)(own_mon_work(m))(rtf)
71.    ⌈⌉
71a.    ⌈⌉ { let ((vi,vp),t) = (vm_ch[mi,vi]?,clk_ch?) in
71b.         let rtf′ = rtf † [t ↦ rtf(max dom rtf) † [vi ↦ vp]] in
71c.         mon(mi)(m)(rtf′) end
71d.         end | vi:VI • vi ∈ vis }

70.   own_mon_work: M → dRTF → M
```

We do not describe the clock behaviour by other than stating that it continually offers the current time on channel clkm_ch. ■

3 Domains

We characterise a number of terms.

[1] Domain: By a domain$_\delta$ we shall here understand an area of human activity characterised by observable phenomena: entities whether endurants [**is_endurant(e)**] (manifest parts and materials) or perdurants [**is_perdurant(e)**] (actions, events or behaviours), whether discrete [**is_discrete(e)**] or continuous [**is_continuous(e)**] ; and of their properties.

[2] Domain Phenomena: By a domain phenomenon$_\delta$ we shall understand something that can be observed by the human senses or by equipment based on laws of physics and chemistry. Those phenomena that can be observed by the human eye or touched, for example, by human hands, we call parts and materials. Those phenomena that can be observed of parts and materials can usually be measured and we call them properties of these parts and those materials.

[3] Domain Entity: By a domain entity$_\delta$ we shall understand a manifest domain phenomenon or a domain concept, i.e., an abstraction, derived from a domain entity.

The distinction between a manifest domain phenomenon and a concept thereof, i.e., a domain concept, is important. Really, what we describe are the domain concepts derived from domain phenomena or from other domain concepts.

[4] Endurant Entity: We distinguish between endurants and perdurants.

From Wikipedia: *By an endurant$_\delta$ (also known as a continuant$_\delta$ or a substance$_\delta$) we shall understand an entity that can be observed, i.e., perceived or conceived, as a complete concept, at no matter which given snapshot of time. Were we to freeze time we would still be able to observe the entire endurant.*

[5] Perdurant Entity: From Wikipedia: *Perdurant: Also known as occurrent, accident or happening. Perdurants are those entities for which only a fragment exists if we look at them at any given snapshot in time. When we freeze time we can only see a fragment of the perdurant. Perdurants are often what we know as processes, for example 'running'. If we freeze time then we only see a fragment of the running, without any previous knowledge one might not even be able to determine the actual process as being a process of running. Other examples include an activation, a kiss, or a procedure.*

[6] Discrete Endurant: We distinguish between discrete endurants and continuous endurants.

By a discrete endurant$_\delta$, that is, a part, we shall understand something which is separate or distinct in form or concept, consisting of distinct or separate parts.

[7] Continuous Endurant: By a continuous endurant$_\delta$, that is, a material, we shall understand an endurant whose spatial characteristics are prolonged, without interruption, in an unbroken spatial series or pattern.

[8] Domain Parts and Materials: By a part$_\delta$ we mean a discrete endurant, a manifest entity which is fixed in shape and extent. By a material$_\delta$ a continuous endurant, a manifest entity which typically varies in shape and extent.

[9] Domain Analysis: By domain analysis$_\delta$ we shall understand an examination of a domain, its entities, their possible composition, properties and relations between entities,

[10] Domain Description: By a domain description$_\delta$ we shall understand a narrative description tightly coupled (say line-number-by-line-number) to a formal description.

[11] Domain Engineering: By domain engineering$_\delta$ we shall understand the engineering of a domain description, that is, the rigorous construction of domain descriptions, and the further analysis of these, creating theories of domains[14], etc.

[12] Domain Science: By domain science$_\delta$ we shall understand two things: the general study and knowledge of how to create and handle domain descriptions (a general theory of domain descriptions) and the specific study and knowledge of a particular domain. The two studies intertwine.

[13] Values & Types: By a value$_\delta$ we mean some mathematical quantity. By a type$_\delta$ we mean a largest set of values, each characterised by the same predicate, such that there are no other values, not members of the set, but which still satisfy that predicate. We do not give examples here of the kind of type predicates that may characterise types.

When we observe a domain we observe instances of entities; but when we describe those instances (which we shall call values) we describe, not the values, but their type and properties: parts and materials have types and values; actions, events and behaviours, all, have types and values, namely as expressed by their signatures; and actions, events and behaviours have properties, namely as expressed by their function definitions. Values are phenomena and types are concepts thereof.

[14] Discrete Perdurant: By a discrete perdurant$_\delta$ we shall understand a perdurant which we consider as taking place instantaneously, in no time, or where whatever time interval it may take to complete is considered immaterial.

[15] Continuous Perdurant: By a continuous perdurant$_\delta$ we shall understand a perdurant whose temporal characteristics are likewise prolonged, without interruption, in an unbroken temporal series or pattern.

[14] Section 2 (Pages 11–30) is an example of the basis for a theory of road traffic systems.

[16] Extensionality: By extensionality$_\delta$ `Merriam-Webster`[15] means *"something which relates to, or is marked by extension," "that is, concerned with objective reality"*. Our use basically follows this characterisation: We think of extensionality as a syntactic notion, one that characterises an exterior appearance or form We shall therefore think of part types and material types whether parts are atomic or composite, and how composite parts are composed as extensional features.

[17] Intentionality: By intentionality$_\delta$ `Merriam-Webster`[16] means: *"done by intention or design", "intended", "of or relating to epistemological intention", "having external reference"*. Our use basically follows this characterisation: we think of intentionality as a semantic notion, one that characterises an intention. We shall therefore think of part attributess and material attributes as intentional features.

• • •

The crucial characterisation is that of domain entity, see Sect. 3[3] (Page 30). It is pivotal since all we describe: narrate and formalise, are domain entities. If we get the characterisation wrong we get everything wrong! What might get the characterisation, or its interpretation, wrong is the interpretation of domain entities: *"those phenomena that can be observed by the human eye or touched, for example, by human hands,"* and *"manifest domain phenomena or domain concepts, i.e., abstractions, derived from a domain entities"*.

The whole thing hinges of *what can be described, what constitutes a description and when is a text a bona fide description.*

Another set of questions are *of what we have chosen to constitute entities which should we describe, which not ?*

Philosophers have dealt with these questions. Recent writings are [4, 59, 25] and [20, 41, 67]. Going back in time we find [42, 39, 21]. The classics are [56, 55, 18, 43].

We shall only indirectly contribute to this philosophical discussion and do so by presenting the material of this paper; having studied, over the years, fragments of the above cited publications we have concluded with the suggestions of this paper: following the principles, techniques and tools presented here can lead the domain engineer to a large class of domain descriptionss, large enough for our "immediate future" needs ! We shall, in the conclusion, return to the questions of what can be described, what constitutes a description and when is a text a bona fide description ?

4 Discrete Endurant Entities

For pragmatics reasons we structure our treatment of discrete endurant domain entities as follows: *First* we treat the extensional aspects of parts, *then their properties:* the intentional aspects. One could claim that when we say 'first' we mean: first a syntactic analysis of parts into atomic and composite parts, etcetera; and when we say 'then their properties' we mean: then a partial semantic analysis, something which "throws" light over parts, since parts really are distinguishable only through their properties.

[15] Extensionality. Merriam-Webster.com. 2011, http://www.merriam-webster.com (16 August 2012).
[16] Intentionality. Merriam-Webster.com. 2011, http://www.merriam-webster.com (16 August 2012).

4.1 Parts – Syntactic Aspects

4.1.1 What is a Part ?

By a part$_\delta$ we mean an observable manifest endurant.

Discussion: We use the term 'part' where others use different terms, for example, 'individual', 'object', 'particular', 'thing', 'unit', or other.

Example: 5 Parts. Example parts have their types defined in the items as follows: N Item 1a Page 11, F Item 1b Page 11, M Item 1c Page 11, HS Item 2a Page 12, LS Item 2b Page 12, VS Item 3 Page 12, Vs Item 4a Page 12, V Item 4b Page 12, Hs Item 5 Page 13, Ls Item 6 Page 13, H Item 5a Page 13, L Item 6b Page 13.

4.1.2 Classes of "Same Kind" Parts

We repeat: the domain describer does not describe instances of parts, but seeks to describe classes of parts of the same kind. Instead of the term 'same kind' we shall use either the terms part sort or part type.

By a same kind class of parts$_\delta$, that is a part sort or part type we shall mean a class all of whose members, i.e., parts, enjoy "exactly" the same properties where a property is expressed as a proposition.

Example: 6 Part Properties. We continue Example 4. Examples of part properties are: *has unique identity* (was exemplified, will be properly defined), *has mereology* (was exemplified, will be properly defined), *has length*, *has location*, *has traffic movement restriction* (as for vehicles along a link, one direction, both directions or closed), *has position* (example: vehicle position), *has velocity* and *has acceleration* (the last two holds for vehicles). ■

4.1.3 A Preview of Part Properties

For pragmatic reasons we group endurant properties into two categories: a group which we shall refer to as meta properties: **is discrete**, **is continuous**, **is atomic**, **is composite**, **has observers**, **is sort** and **has concrete type**; and a group which we shall refer to as part properties **has unique existence**, **has mereology** and **has attributes**. The first group is treated in this section; the second group in Sect. 6.

4.1.4 An Analysis Process: Endurants

The domain analyser examines collections of parts. (i) In doing so the domain analyser discovers and thus identifies and lists a number of properties. (ii) Each of the parts examined usually satisfies only a subset of these properties. (iii) The domain analyser now groups parts into collections such that each collection have its parts satisfy the same set of properties, such that no two distinct collections are indexed, as it were, by the same set of properties, and such that all parts are put in some collection. (iv) The domain analyser now assigns distinct type names (same as sort names) to distinct collections. That is how we assign types to parts. The quality of the part type universe depends on how thoroughly the domain analysers do their job: (α) collecting sufficiently many examples of parts, (β) enumerating sufficiently many examples of property propositions, and (γ) "assigning" appropriate properties to parts. This step of domain description development is crucial to the appropriateness and acceptability of the resulting domain description. Examining too few parts, enumerating too few and/or

irrelevant property propositions sloppiness in general can often result in domain models that turn out to be "unwieldy", models that do not capture, sufficiently elegantly the core domain concepts. For good advice in seeking elegance in models see [37, M.A. Jackson: Lexicon ...].

We shall return later to a proper treatment of formal concept analysis [27].

4.1.5 Part Property Values

By a part property value$_\delta$, i.e., a property value$_\delta$ of a part, we mean the value associated with an intentional property of the part.

Example: 7 Part Property Values. A link, l:L, may have the following intentional property values: LOCation value *loc_set*, LENgth value *123 meters* and *mereology* value $\{\kappa_i, \kappa_j\}$. ■

Two parts of the same type are different if for at least one of the intentional properties of that part type they have different part property values. slut

Example: 8 Distinct Parts. Two links, l_a,l_b:L, may have the following respective property values: LOCation values *loc_set*$_a$, and *loc_set*$_b$, LENgth value *123 meters* and *123 meters*, i.e., the same, and *mereology* values $\{\kappa_i, \kappa_j\}$ and $\{\kappa_m, \kappa_n\}$ where $\{\kappa_i, \kappa_j\} \neq \{\kappa_m, \kappa_n\}$. When so, they are distinct, and the cadestral space *loc_set*$_a$ must not share any point with cadestral space *loc_set*$_b$.

4.1.6 Part Sorts

By an abstract type$_\delta$, or a sort$_\delta$, we shall understand a type which has been given a name but is otherwise undefined, that is, is a set of values of further undefined quantities [50, 49]. where these are given properties which we may express in terms of axioms over sort (including property) values. All of the above examples are examples of sorts.

Example: 9 Part Sorts. The discovery of N, F and M was made as a result of examining the domain, Δ , at domain index ⟨Δ⟩; HS and LS at domain index ⟨Δ,N⟩; Hs and H (Ls and L) at domain indexes ⟨Δ,HS⟩ (⟨Δ,LS⟩); and Vs and V at domain index ⟨Δ,VS⟩. ■

4.1.7 Atomic Parts

By an atomic part$_\delta$ we mean a part which, in a given context, is deemed *not* to consist of meaningful, separately observable proper sub-parts. A sub-part is a part.

Example: 10 Atomic Types. We have exemplified the following atomic types: H (Item 5b on Page 13), L (Item 6b on Page 13), V (Item 4b on Page 12) and M (Item 1c on Page 11).

Implicit tests, at domain indexes, by the domain analyser, for atomicity were performed as follows: for H at ⟨Δ, N,HS,Hs,H⟩; for L at ⟨Δ, N,LS,Ls,L⟩; for V at ⟨Δ, F,VS,Vs,V⟩; and for M at ⟨Δ, M⟩. ■

4.1.8 Composite Parts

By a composite part$_\delta$ we mean *a part which, in a given context, is deemed to indeed consist of meaningful, separately observable proper sub-parts.*

Example: 11 Composite Types. We have exemplified the following composite types: N (Items 2a– 2b on Page 12), HS (Item 5 on Page 13), LS (Item 6 on Page 13), Hs (Item 5a on Page 13), Ls (Item 6a on Page 13), F (Item 3 on Page 12), VS (Item 4a on Page 12), Va (Item 4a on Page 12), respectively. Tests for compostionality of these were implicitly performed; for N at index ⟨Δ, N⟩; for HS and LS at index ⟨Δ, N,HS⟩ and ⟨Δ, N,LS⟩; for Hs and Ls at indexes ⟨Δ, N,HS,Hs⟩ and ⟨Δ, N,LS,Ls⟩; for F at index ⟨Δ, F⟩; for VS at index ⟨Δ, F,VS⟩; and for Vs at index ⟨Δ, F,VS,Vs⟩. ■

4.1.9 Part Observers

By a part observer$_\delta$ or a material observer$_\delta$ we mean a meta-physical operator (a meta function),

72. **obs_B**: P → B

that is, one performed by the domain analyser, which "applies" (i.e., who applies it) to a composite part value[17], P, and which yields the sub-part of type B, of the examined part. The **obs_** "keyword" prefix to a part type name B is intended to alert the reader to the fact that **obs_**B is a meta function.

We name these **obs_**erver functions obs_X to indicate that they are observing parts of type X. The **obs_**erver functions are not computable. They can not be mechanised. Therefore we refer to them as mental. They can be "implemented" as, for example, follows:

Example: 12 Implementation of Observer Functions. I take you around a particular road net, n, say in my town. I point out to you, one-by-one, all the street intersections, $h_1, h_2, \ldots, h_n$, of that net. You "write" them down: as many characteristics as you (and I) can come across, including some choice of unique identifiers, their mereologies, and attributes, "one-by-one". In the end we have identified, i.e., visited, all the hubs in my town's road net n. ■

Example: 13 Observer Functions. We have exemplified the following obs_erver functions: **obs_**N (Item 1a on Page 11), **obs_**F (Item 1b on Page 11), **obs_**M (Item 1c on Page 11), **obs_**HS (Item 2a on Page 12), **obs_**LS (Item 2b on Page 12), **obs_**VS (Item 3 on Page 12), **obs_**Vs (Item 4a on Page 12), **obs_**Hs (Item 5 on Page 13) and **obs_**Ls (Item 6 on Page 13), where we list their "definitions", not their many uses. ■

4.1.10 Part Types

By a concrete type$_\delta$ we shall understand a type, T, which has been given both a name and a defining type expression of, for example the form T = A-**set**, T = A-**infset**, T = A×B×···×C, T = A*, T = A$^\omega$, T = A $\underset{m}{\rightarrow}$ B, T = A→B, T = A$\overset{\sim}{\rightarrow}$B, or T = A|B|···|C. where A, B, ..., C are type names or type expressions.

Example: 14 Concrete Types. Example concrete part types were exemplified in Vs = V-**set**: Item 4a on Page 12, Hs = H-**set**: Item 5a Page 13, Ls = L-**set**: Item 6a Page 13. ■

Example: 15 Has Composite Types. The discovery of concrete types were done as follows: for HS, Hs = H-**set** at ⟨Δ,N,HS⟩, for LS, Ls = L-**set** at ⟨Δ,N,LS⟩, and for VS, Vs = V-**set** at ⟨Δ,F,VS⟩. ■

[17] or composite part type

4.2 Part Properties

(I) By a property[18] we mean a pair a (finite) collection of one or more propositions.

(II) By an endurant property a property which holds of an endurant — which we *model* as a *pair* of a type and a value (of that type)[19].

(III) By a perdurant property$_\delta$ we shall mean a property which holds of an perdurant — which we, as a minimum, *model* as a *pair* of a perdurant name and a function type, that is, as a function signature.

Property Value Scales: With intentional properties we associate a property value scale. By a property value scale$_\delta$ of a part type we shall mean a value range that parts of that type will have their property values range over.

Example: 16 Property Value Scales. We continue Example 4. (i) The mereology property value scale$_\delta$ for hubs of a net range over finite sets of link identifiers of that net. (ii)The mereology property value scale$_\delta$ for links of a net range over two element sets of hub identifiers for that net. (iii)The range of location values for any one hub of a net is restricted to not share any cadestral point with any other hub's location value for that net.

Discussion: The notion of 'property' is central to much philosophical discussion; we mention a few (that we have studied): [25, The Ontology of Language: Properties, Individuals and Discourse], [58, Parts: A Study in Ontology] and [46, Properties].[20] Their reading has influenced our work.

The notion of 'property' is also central to the recent notion of concept analysis [27, Formal Concept Analysis – Mathematical Foundations]. Here the term concept is understood as *a property of a part*. There is no associated type and value notions such as we have expressed in (II) on the current page and Footnote 19.

We shall now unravel our 'Property Theory'[21] of parts.

We see three categories of part properties: unique identifiers, mereology and (general) attributes.

Each and every part has unique existence — which we model through unique identifiers. Parts relate (somehow) to other parts, that is, mereology — which we model a relations between unique identifiers. And parts usually have other, additional properties which we shall refer to as attributes — which we model as pairs of attribute types and attribute values.

4.2.1 Unique Identifiers

Given that we can assume that each and every part, p:P, has unique existence we can postulate the unique identifier observer function:

- uid_P: P →PI

[18] By saying 'a property' we definitely mean to distinguish our use of the term from one which refers to legal property such as physical (land) or intangible (legal rights) property.

[19] The type value may be a singleton, or lie within a range of discrete values, or lie within a range of continuous values. The ranges may be finite or may be infinite.

[20] A reading of the contents listing of [46] reveals an interpretation of *parts and properties:*

I Function and Concept, Gottlob Frege
II The World of Universals, Bertrand Russell
III On our Knowledge of Universals, Bertrand Russell
IV Universals, F. P. Ramsey
V On What There Is, W. V. Quine
VI Statements about Universals, Frank Jackson
VII 'Ostrich Nominalism' or 'Mirage Realism', Michael Devitt
VIII Against 'Ostrich' Nominalism, D. M. Armstrong
IX On the Elements of Being: I, Donald C. Williams
X The Metaphysic of Abstract Particulars, Keith Campbell
XI Tropes, Chris Daly
XII Properties, D. M. Armstrong
XIII Modal Realism at Work: Properties, David Lewis
XIV New Work for a Theory of Universals, David Lewis
XV Causality and Properties, Sydney Shoemaker
XVI Properties and Predicates, D. H. Mellor.

[21] — with apologies to [61, 62, 25].

.

Example: 17 Unique Identifier Functions. We have only exemplified the following unique identifier meta-functions and types: **uid_**H, HI Item 7a on Page 13, **uid_**L, LI Item 7b on Page 14 and **uid_**V, VI Item 7c on Page 14. We did not find a need for defining unique identifier meta-functions for N, F, M, HS, Hs, LS, Ls, VS, and Vs. ■

[1] A Dogma of Unique Existence: We take, as a dogma, that every two parts whose intentional property values differ for at least one property, other than their unique identifiers, are distinct and thus have distinct unique identifiers.

[2] A Simplification on Specification of Intentional Properties: So we make a simplification in our treatment of intentional part properties By postulating distinct unique identifiers we are forcing distinctness of parts and can dispense with, that is, do not have to explicitly ascribe such intentional properties whose associated values would then have to differ in order to guarantee distinctness of parts,

[3] Discussion: Parts have unique existence. Whether they be spatial or conceptual. Two manifest parts cannot overlap spatially. A part is a conceptual part if it is an abstraction of a part. Two conceptual parts are identical if they have identical properties, that is, abstract the same manifest part, otherwise they are distinct. We shall therefore associate with each part a unique identifier, whether we may need to refer to that property or not. There are only manifest parts and conceptual parts. The above deserves a whole separate inquiry. In defense of the above, perhaps somewhat dogmatically phrased position, we refer to Russel's [57].

[4] The uid_P Operator: More specifically we postulate, for every part, p:P, a meta-function:

73. **uid_**P: P → Π

where Π is the type of the unique identifiers of parts p:P. The **uid_** "keyword" prefix to a part type name P is intended to alert the reader to the fact that **uid_**P is a meta function. In practice we "construct" the unique identifier type name for parts of type P by "suffixing" I to P, and we explicitly "postulate define" the meta-function shown in Item 73. How is the uid_PI meta-function "implemented"? Well, for a domain description it suffices to postulate it. If we later were to develop software in support of the described domain, then there are many ways of "implementing" the uid_PIs.

[5] Constancy of Unique Identifiers — Some Dogmas: We postulate the following dogmas: parts may be "added" to or "removed" from a domain; parts that are "added" to a domain have unique identifiers that are not identifiers of any other part of the history of the domain; parts that are "removed" from a domain will not have their identifiers reused should parts subsequently be "added" to the domain; and domains do not allow for the changing (update) of unique identifier values.

4.2.2 Mereology

Mereology: By mereology$_\delta$ (Greek: $\mu\epsilon\rho o\varsigma$) we shall understand the study and knowledge about the theory of *part-hood* relations: of the relations of *part* to *whole* and the relations of *part* to *part* within a *whole*.

In the following please observe the type font distinctions: *part*, etc., and part (etc.).

In the above definition of the term mereology we have used the terms *part-hood, part* and *whole* in a more general sense than we use the term part.

In this the "more general sense" we interpret *part* to include, besides what the term part covers in these notes, also concepts, abstractions, derived from the concept of part.

That is, by *part* we mean not only manifest phenomena but also intangible phenomena that may be abstract models of parts, or may be (further) abstract models of *part*s.

Example: 18 Manifest and Conceptual Parts. We refer to Example 4. A net, n:N (Item 1a on Page 12), is a manifest part whereas a map, rm:RM (Item 26 on Page 19), is a *part*. ■

[1] Extensional and Intentional Part Relations: Henceforth we shall "merge" the two terms *part* and part into one meaning.

So henceforth the term part shall refer to both manifest, tangible and discrete endurants and to abstractions of these. We are forced to do so by necessity. Instead of describing the manifest phenomena we are describing conceptual models of these; that is, instead of describing manifest parts we are describing their part types and part properties.

Thus we choose "mereology" to model relations between both parts and *part*s. We can thus distinguish between two kinds of such relations: extensional part relations which typically are spatial relations between manifest parts and intentional part relations which typically are conceptual relations between abstract parts.

Extensional relations between manifest parts are of the kind: one part, p:P, is *"adjacent to"* (*"physically neighbouring"*) another part, q:Q, one part, p:P, is *"embedded within"* (*"physically surrounded by"*) another part, q:Q, and one part, p:P, *"overlaps with"* another part, q:Q.[22] We model these relations, "equivalently", as follows: in the mereology of p, **mereo_**P(p), there is a reference, **uid_**Q(q), to q, and in the mereology of q, **mereo_**Q(q), there is a reference, **uid_**P(p), to p.

Intentional relations between abstractions are of the kind: part p:P has an attribute whose value always stand in a certain relation (for example, a copy of a fragment or the whole) to another part q:Q's "corresponding" attribute value.

Example: 19 Shared Route Maps and Bus Time Tables. We continue and we extend Example 4. The 'Road Transport Domain' of Example 4 has its fleet of vehicles be that of a metropolitan city's busses which ply some of the routes according to the city road map (i.e., the net) and according to a bus time table — which we leave undefined. We can now re-interpret the road traffic monitor to represent a coordinating bus traffic authority, CBTA. CBTA is now the "new" monitor, i.e., is a part. Two of its attributes are: a metropolitan area road map and a metropolitan area bus time table Vehicles are now busses and each bus follows a route of the metropolitan area road map of which it has a copy, as a vehicle attribute, "shared" with CBTA; each bus additionally runs according to the metropolitan area bus time table of which it has a copy, as a vehicle attribute, "shared" with CBTA. ■

We model these attribute value relations, " equivalently", as above: in the mereology of p, **mereo_**P(p), there is a reference, **uid_**Q(q), to q, and in the mereology of q, **mereo_**Q(q), there is a reference, **uid_**P(p), to p.

[22] The reader may wonder: How can a manifest physical part "overlap" another such part ? We shall comment on this conundrum later in these notes. [Conundrum: a question or problem having only a conjectural answer.]

Example: 20 Monitor and Vehicle Mereologies. We continue Example 19 on the preceding page.

```
74. value  mereo_M: VI-set
75. type   MI
76. value  uid_M: M → MI
77. value  mereo_V: V → MI
```

■

[2] Unique Part Identifier Mereologies: To express a unique part identifier mereology assumes that the related parts have been endowed, say explicitly, with unique part identifiers., say of unique identifier types $\Pi_j, \Pi_k, \ldots, \Pi_\ell$. A mereology meta function is now postulated:

```
78. value  mereo_P: P → (Πj | Πk | ... | Πℓ)-set,
```

or of some such signature, one which applies to parts, p:P, and yields unique identifiers of other, "the related", parts — where these "other parts" can be of any part type, including P. The **mereo_** "keyword" prefix to a part type name P is intended to alert the reader to the fact that **mereo_**P is a meta function.

Example: 21 Road Traffic System Mereology. We have exemplified unique part identifier mereologies for hubs, mereo_H Item 8a on Page 14 and links, mereo_L Item 9a on Page 14. ■

Example: 22 Pipeline Mereology. This is a somewhat lengthy example from a domain now being exemplified. We start by narrating a pipeline domain of pipelines and pipeline units.

79. A pipeline consists of pipeline units.

80. A pipeline unit is either

    [a] a well unit output connected to a pipe or a pump unit;

    [b] a pipe, a pump or a valve unit input and output connected to two distinct pipeline units other than a well;

    [c] a fork unit input connected to a pipeline unit other than a well and output connected to two pipeline units other than wells and sinks;

    [d] a join unit input connected to two pipeline units other than wells and output connected to a a pipeline unit other than a sink; and

    [e] a sink unit input connected to a valve.

```
type
79.  PL
value
79.  obs_Us: PL → U-set
type
80.  U = WeU | PiU | PuU | VaU | FoU | JoU | SiU
value
80.  uid_U: U → UI
80.  mereo_U: U → UI-set × UI-set
80.  i_mereo_U,o_mereo_U: U → UI-set
80.  i_UIs(u) ≡ let (ius,_) = mereo_U(u) in ius end
80.  o_UIs(u) ≡ let (_,ous) = mereo_U(u) in ous end
```

axiom
∀ pl:PL,u:U • u ∈ **obs_**Us(pl) ⇒
80a. **is_**WeU(u) → **card** i_UIs(u)=0 ∧ **card** o_UIs(u)=1,
80b. (**is_**PiU|**is_**PuU|**is_**VaU)(u) → **card** i_UIs(u)=1=**card** o_UIs(u),
80c. **is_**FoU(u) → **card** i_UIs(u)=1 ∧ **card** o_UIs(u)=2,
80d. **is_**JoU(u) → **card** i_UIs(u)=2 ∧ **card** o_UIs(u)=1,
80e. **is_**SiU(u) → **card** i_UIs(u)=1 ∧ **card** o_UIs(u)=0

The UI "typed" **value** and **axiom** Items 80 "reveal" the mereology of pipelines. ■

[3] Concrete Part Type Mereologies: Let A_i and B_j, for suitable i, j denote distinct part types and let B_jI Let there be the following concrete type definitions:

type
a_1:A_1 = bs:B_1-**set**
a_2:A_2 = bc:B_{2_1} × B_{2_2} × ... × B_{2_n}
a_3:A_3 = bl:$B_3{}^*$
a_4:A_4 = bm:BI_4 $\overrightarrow{m}$ B_4

The above part type definitions can be interpreted mereologically: Part a:A_1 has sub-parts b_{1_i},b_{1_2},...,b_{1_m}:B_1 of bs parthood related to just part a:A_1. Parts a:A_2 has sub-parts b_{2_1},b_{2_2},. . . ,b_{2_m}:B_2 of bc parthood related only to parts a:A_1 Parts a:A_3 has sub-parts b_{3_i}, for all indices i of the list bℓ, parthood related to parts a:A_3, and to part $b_{3_{i-1}}$ and part $b_{3_{i+1}}$, for $1<i<$**len** bℓ by being "neighbours" and also to other b_{3_j} if the index j is known to b_{3_i} for $i \neq j$. Parts a:A_4 have all parts bm(bi_j) for index bi_j in the definition set **dom** bm, be parthood related to a:A_4 and to other such bm:B_4 parts if they know their indexes.

Example: 23 A Container Line Mereology. This example brings yet another domain into consideration.

81. Two parts, sets of container vessels, CV-**set**, and sets of container terminal ports, CTP-**set**, are crucial to container lines, CL.
82. Crucial parts of container vessels and container terminal ports are their structures of *bays*, bs:BS.
83. A bay structure consists of an indexed set of *bay*s.
84. A *bay* consists of an indexed set of *rows*
85. A *row* consists of an index set of *stacks*.
86. A *stack* consists of a linear sequence of *container*s.

type
81. CP, CVS, CTPS
value
81. **obs_**CVS: CL → CVS
81. **obs_**CTPS: CL → CTPS

type
81. CVS = CV-**set**
81. CTPS = CTP-**set**
value
82. **obs_**BS: (CV|CTP) → BS
type
83. BI, BS, B = BI $\overrightarrow{m}$ B
value
84. **obs_**RS: B → RS
type
84. RI, RS, R = RI $\overrightarrow{m}$ R
value
85. **obs_**SS: R → SS
type
85. SI, SS, C = SI $\overrightarrow{m}$ S
86. S = C*

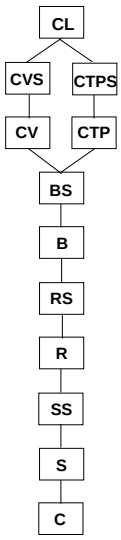


Figure 1: A container line domain index lattice

In Fig. 1 is shown a container line domain index lattice. At the top ("root") there is the container line domain type name. Immediately below it are the, in this case, two sub-domains (that we consider), CVS and CTPS. For each of these two there are the corresponding CV and CTP sun-domains. For each of these one can observe the container bays, hence, definition-wise, shared sub-domain. It is then defined in terms of a sequence of increasingly more "narrowly" defined sub-domains. The lattice "ends" with the atomic sub-domain of containers, C. ■

• • •

Discussion: Mereology is a discipline of study within both philosophy and logic. Mereology, in one form or another, has been studied, by philosophers, over the millennia, in 'Ancient Greece' (Plato, Aristotle), 'Roman Times' (Boethius), 'Medieval Ages' (Abelard, Aquinas)

and in the 'Age of Enlightenment' (Kant), mereology became the subject also of a rigorous mathematical treatment, in the 1920s, by the Polish mathematician *Stanisław Leshniewski* [44, 48, 60]. Now it is also becoming a study within computer science [9, 13]. Modern study of mereology [66, 65, 19] treats it axiomatically. We shall, in contrast, suggest model-oriented descriptions of mereology. In [13] we indicate how a general model, $\mathcal{M}$, of mereology satisfies an axiomatic presentation, $\mathcal{A}$, a theory, that is, $\mathcal{M} \models \mathcal{A}$.

We present two classes of models of domain mereologies. One class of mereology models are based on the use of unique part identifiers. The other class of mereology models are based on concrete part type definitions. In either set of models the mereology that we shall express is about how a part is related to other parts and we "lightly" understand that relationship as a kind of connection: whether spatial connection in the form of a part, p, being either "somehow" *contained within* another, an "embracing" part, p', or "somehow" *adjacent to* another, a "neigbouring" part, p'; or conceptual connection in the form of properties of one part, p, being related to properties of one part, p, whether these properties be spatial or otherwise.

[4] Variability of Mereologies: The mereology of parts (of type P) may be a constant, i.e., static, or a variable, i.e., dynamic. That is, for some, or all, parts of a part type may need to be updated. We express the update of a part mereology as follows:

87. **value** **upd_mereo**_P: $(\Pi_i|\Pi_i|\ldots|\Pi_i)$**-set** $\rightarrow$ P $\rightarrow$ P

where **upd_mereo**_P$(\{\pi_a, \pi_b, \ldots, \pi_c\})$(p) results in a part p′:P where all part properties of p′ other than its mereology are as they "were" in p but the mereology of p′ is $\{\pi_a, \pi_b, \ldots, \pi_c\}$.

Example: 24 **Insert Link.** We continue Example 4, Item 42 on Page 24: In the post_link_dis predicate we referred to the undefined link insert function, ins_L. We now define that function:

88. The insert_Link action applies to a net, n, and a link, l,
89. and yields a new net, n′.
90. The conditions for a successful insertion are
    [a] that the link, l, is not in the links of net n,
    [b] that the unique identifier of l is not in the set of unique identifiers of the net n, and
    [c] that the mereology of link l has been prepared to be, i.e., is the two element set of unique identifiers of two hubs in net n.
91. The result of a successful insertion is
    [a] that the links of the new net, n′, are those of the previous net, n, "plus" link l;
    [b] that the hubs, "originally" h_a,h_b, connected by l, are only mereo-logically updated to each additional include the unique identifier of l; and
    [c] that all other hubs of n and n′ are unchanged.

88. ins_L: N $\rightarrow$ L $\rightarrow$ N
89. ins_L(n)(l) **as** n′
90. **pre**:
90a. l $\notin$ **obs**_Ls(**obs**_LS(n))

90b. $\wedge$ **uid_**L(l) $\notin$ **in** xtr_LIs(n)
90c. $\wedge$ **mereo_**L(l) $\subseteq$ xtr_HIs(n)
91. **post**:
91a. **obs_**Ls(**obs_**LS(n′))=**obs_**Ls(**obs_**LS(n))∪{l}
91. $\wedge$ **let** {hi_a,hi_b}=**mereo_**L(l) **in**
91. **let** {h_a,h_b}={get_H(hi_a)(n),get_H(hi_b)(n)} **in**
91b. get_H(hi_a)(n′)=**upd_mereo_**H(h_a)(**mereo_**H(h_a)∪{**uid_**L(l)})
91b. $\wedge$ get_H(hi_b)(n′)=**upd_mereo_**H(h_b)(**mereo_**H(h_b)∪{**uid_**L(l)})
91c. $\wedge$ **obs_**Hs(**obs_**HS(n))=**obs_**Hs(**obs_**HS(n))\{hi_a,hi_b}∪{h_a′,h_b′} **end end**

As for the very many other function definitions in these notes we illustrate one form of function definition annotations, and not always consistently the same "style". We do not pretend that our function definitions are novel, let alone a contribution of these notes; instead we rely on the reader having learnt, more laboriously than we these notes can muster, an appropriate function definition narrative style. ■

• • •

This point in these notes may also be an appropriate one for briefly discussing another aspect the form of of formal function definitions. Even to us, even though we certainly do not always adhere to this *desiderata,* a function definition ought be formulated in a few lines: 2–3, at most 4. If, as above, we do not achieve that, in a "first attempt",[23] then the developer ought split that function definition into several such. To do so often amounts to the separate development of *a domain theory:* a number of more-or-less "ultra-short" definitions and their repeated re-use in many contexts while also developing a number of theorems based also on axioms of that *domain theory*.

4.2.3 Attributes

Attribute: By a part attribute$_\delta$ we mean a part property other than part unique identifier and part mereology, and its associated attribute property value.

Example: 25 Road Transport System Part Attributes. We have exemplified, Example 4, a number of part attribute observation functions: attr_LΣ Item 10a on Page 15, attr_LΩ Item 10b on Page 15, attr_LOC, attr_LEN Item 10c on Page 15, attr_HΣ Item 11a on Page 15, attr_HΩ Item 11b on Page 15, attr_LOC Item 11c on Page 16, attr_VP, attr_onL, attr_atH, attr_VEL and attr_ACC Item 13 on Page 16. ■

[1] Stages of Attribute Analysis: There are four facets to deciding upon part attributes: (i) determining on which attributes to focus; (ii) selecting appropriate attribute type names, (**viz.**, LΣ, LΩ, HΣ, HΩ, LEN, LOC, VP, atH, onL, VEL and ACC from the above example); (iii) determining whether an attribute type is a static attribute type (having constant value) (**viz.**, LEN, LOC), or a dynamic attribute type (having variable values)) (**viz.**, LΣ, LΩ, HΣ, HΩ, VP, atH, onL, VEL, ACC); and (iv) deciding upon possible concrete type definitions for (some of) those attribute types (**viz.**, LΣ, LΩ, HΣ, HΩ, VP, atH, onL).

[23] We refer to some such "not too tersely expressed" function definitions: wf_RM Item 26 on Page 19 (where we suggest that the three line Item 26b become the body of an auxiliary predicate), and, notably, the above ins_L Item 88 on the preceding page.

Example: 26 Static and Dynamic Attributes. Continuing Example 4 we have: Dynamic attributes: LΣ Item 10a on Page 15; HΣ Item 11a on Page 15; VP, atH, onL Items 12a–12(a)ii on Page 16; and VEL and ACC both Item 13 on Page 16. All other attributes are considered static. ■

Example: 27 Concrete Attribute Types. From Example 4: LΣ=(HI×HI) Item 10a on Page 15, LΩ=LΣ-**set** Item 10b on Page 15, HΣ=(LI×LI)-**set** Item 11a on Page 15 and HΩ=HΣ-**set** Item 11b on Page 15. ■

[2] The attr_A Operator: To observe a part attribute we therefore describe the attribute observer signature

92. **attr_A**: P → A,

where P is the part type being examined for attributes, and A is one of the chosen attribute type names. The **attr_** "keyword" prefix to an attribute type name A is intended to alert the reader to the fact that **attr_**A is a meta function. The "hunt" for part attributes, i.e., attribute types, the resulting attribute function signatures and the chosen concrete attribute types is crucial for achieving successful domain descriptions.

[3] Variability of Attributes: Static attributes are constants. Dynamic attributes are variables. To express the update of any one specific dynamic attributevalue we use the meta-operator:

93. **value** **upd_attr_**A: A → P → P

where **upd_attr_**A(a)(p) results in a part p′:P where all part properties of p′ other than its the attribute value for attribute A are as they "were" in p but the attribute value for attribute A is a. The **upd_attr_** "keyword" prefix to an attribute type name A is intended to alert the reader to the fact that **upd_attr_**A is a meta function.

Example: 28 Setting Road Intersection Traffic Lights. We refer to Example 4, Items 11a (HΣ) and 11b (HΩ) on Page 16. The intent of the hub state model (a hub state as a set of pairs of unique link identifiers) is that it expresses the possibly empty set of allowed hub traversals, from a link incident upon the hub to a link emanating from that hub.

94. In order to "change" a hub state the set_hub_state action is performed,

95. It takes a hub and a hub state and yields a changed hub.
    The argument hub state must be in the state space of the hub.
    The result of setting the hub state is that the resulting hub has the argument state as its (updated) hub state.

value
94. set_hub_state: H → HΣ → H
95. set_hub_state(h)(hσ) ≡ **upd_attr_**HΣ(h)(hσ)
95. **pre**: hσ ∈ **attr_**HΩ(h)

The hub state has not changed if **attr_**HΣ(h) = hσ. ■

4.2.4 Properties of Parts

The properties of parts and materials are fully captured by (i) the unique part identifiers, (ii) the part mereology and (iii) the full set ofpart attributes and material attributes We therefore postulate a property function when when applied to a part or a material yield this triplet, (i–iii), of properties in a suitable structure.

type
  Props = {|PI|**nil**|} × {|(PI-**set**×...×PI-**set**)|**nil**|} × Attrs
value
  props: Part|Material → Props

where Part stands for a part type, Material stands for a material type, PI stand for unique part identifiers and PI-**set**×...×PI-**set** for part mereologies. The {|...|} denotes a proper specification language sub-type and **nil** denotes the empty type.

4.3 States

By a state$_\delta$ we mean a collection of such parts some of whose part attribute values are dynamic, that is, can vary.

Example: 29 A Variety of Road Traffic Domain States. We continue Example 4. A link, l:L, constitutes a state by virtue of if its link traffic state lσ:**attr_**LΣ. A hub, h:H, constitutes a state by virtue of its hub traffic state hσ:**attr_**HΣ, and independently, its hub mereology lis:LI-**set**:**mereo_**H. A net, n:N, constitutes a state by virtue of if its link and hub states. A monitor, m:M, constitutes a state by virtue of if its vehicle position map vpm:**attr_**VPM. ■

4.4 An Example Domain: Pipelines

We close Sect. 4 with a "second main example", albeit "smaller", in text size, than Example 4. The domain is that of pipelines. The reason we bring this example is the following: Not all domain endurants are discrete domain endurants. Some domains possess continuous domain endurants. We shall call them materials. Two such materials are liquids, like `oil` (or `petroleum`), and gaseous, like `natural gas`. The description of such, as we shall later call them, materials-based domains requires additional description concepts and new description techniques. The examples of this subsection, i.e., Sect. 4.4 illustrates these new concepts and techniques as do the examples of Sect. 6.1.

Example: 30 Pipeline Units and Their Mereology.

96. A pipeline consists of connected units, u:U.
97. Units have unique identifiers.
98. And units have mereologies, ui:UI:
    [a] pump[24], pu:Pu, pipe, pi:Pi, and valve[25], va:Va, units have one input connector and one output connector;

[24] We abstract from such distinctions between *oil pipeline pumps* and *gas pipeline compressors*.
[25] We abstract *regulator stations* (where the pipeline operator can release some of the pressure from the pipeline) and *block valve stations* (where the operator can isolate any segment of a pipeline for maintenance work or isolate a rupture or leak) into valves.

[b] fork, fo:Fo, [join, jo:Jo] units have one [two] input connector[s] and two [one] output connector[s];

[c] well[26], we:We, [sink[27], si:Si] units have zero [one] input connector and one [zero] output connector.

[d] Connectors of a unit are designated by the unit identifier of the connected unit.

[e] The auxiliary sel_UIs_in selector funtion selects the unique identifiers of pipeline units providing input to a unit;

[f] sel_UIs_out selects unique identifiers of output recipients.

```
type
96.  U = Pu | Pi | Va | Fo | Jo | Si | We
97.  UI
value
97.  uid_U: U → UI
98.  mereo_U: U → UI-set × UI-set
98.  wf_mereo_U: U → Bool
98.  wf_mereo_U(u) ≡
98a.     let (iuis,ouis) = mereo_U(u) in
98a.     is_(Pu|Pi|Va)(u) → card iusi = 1 = card ouis,
98b.     is_Fo(u) → card iuis = 1 ∧ card ouis = 2,
98b.     is_Jo(u) → card iuis = 2 ∧ card ouis = 1,
98c.     is_We(u) → card iuis = 0 ∧ card ouis = 1,
98d.     is_Si(u) → card iuis = 1 ∧ card ouis = 0 end
98e.  sel_UIs_in: U → UI-set
98e.  sel_UIs_in(u) ≡ let (iuis,_)=mereo_U(u) in iuis end
98f.  sel_UIs_out: U → UI-set
98f.  sel_UIs_out(u) ≡ let (_,ouis)=mereo_U(u) in ouis end
```

Example: 31 Pipelines: Nets and Routes.

99. A pipeline net consists of several properly connected pipeline units.

    Example 30 on the preceding page already described pipeline units.

    Here we shall concentrate on their connectedness, i.e., the wellformednes of pipeline nets.

100. A pipeline net is well-formed if

    [a] all routes of the net are acyclic, and

    [b] there are a non-empty set of well-to-sink routes that connect any well to some sink, and

    [c] all other routes of the net are embedded in the well-to-sink routes

[26] We abstract wells into initial injection stations where the liquid or gaseous material is injected into the line.

[27] We abstract partial and final delivery stations into sinks, places where the material is delivered to an agent outside the pipeline system.

```
type
99.    PLN′
99.    PLN = {| pln:PLN′ • is_wf_PLN(pln) |}
value
99.    obs_Us: PLN → U-set
100.   is_wf_PLN: PLN′ → Bool
100.   is_wf_PLN(pln) ≡
100.          let rs = routes{pln} in
100b.          well_to_sink_routes(pln)≠{}
100c.       ∧ embedded_routes(pln) end
```

101. An acyclic route is a route where any element occurs at most once.

102. A well-to-sink route of a net, pln, is a route whose first element designates a well in pln and whose last element designates a sink in pln.

103. One non-empty route, r′, is embedded in another route, r if the latter can be expressed as the concatenation of three routes: r = r″⌢r′⌢r‴ where r″ or r‴ may be empty routes (⟨⟩).

```
type
105.    R′ = UI*
100a.   R = {r:R′•is_acyclic(r)}
value
100a.   is_acyclic: R → Bool
100a.   is_acyclic(r) ≡ ∀ i,j:Nat•i≠j∧{i,j}⊆inds r⇒r[i]≠r[j]

100b.   well_to_sink_routes: PLN → R-set
100b.   well_to_sink_routes(pln) ≡
100b.      {r|r:R•r ∈ routes(pln) ∧ ∃ we:WE,si:Si •
100b.          {we,si}⊆obs_Us(pln) ⇒ r[1]=we ∧ r[len r]=si}
```

104. One non-empty route, er, is_embedded in another route, r,

   [a] if there are two indices, i, j, into r

   [b] such that the sequence of r elements from and including i to and including j is er.

```
value
104.   is_embedded: R × R → Bool
104.   is_embedded(er,r) ≡
104a.      ∃ i,j:Nat•{i,j}⊆inds r
104b.          ⇒ er = ⟨r[k]|k:Nat • i≤k≤j⟩
104.      pre: er≠⟨⟩
```

105. A route, r, of a pipeline net is a sequence of unique unit identifiers, satisfying the following properties:

[a] if r[i]=ui_i has ui_i designate a unit, u, of the pipeline then $\langle \mathsf{ui}_i \rangle$ is a route of the net;

[b] if $\mathsf{r}_i \frown \langle \mathsf{ui}_i \rangle$ and $\langle \mathsf{ui}_j \rangle \frown \mathsf{r}_j$ are routes of the net

  i. where u_i and u_j are the units (of the net) designated by ui_i and ui_j

  ii. and ui_j is in the output mereology of u_i and ui_i is in the input mereology of u_j

  iii. then $\mathsf{r}_i \frown \langle \mathsf{ui}_i \rangle \frown \langle \mathsf{ui}_j \rangle \frown \mathsf{r}_j$ is a route of the net.

[c] Only such routes that can be constructed by a finite number of "applications" of Items 105a and 105b are routes.

105. routes: PLN → R-**set**
105. routes(pln) ≡
105a. **let** rs = {⟨**uid_**UI(u)⟩|u:U•u ∈ **obs_**Us(pln)}
105(b)iii. ∪ { $r_i \frown \langle ui_i \rangle \frown \langle ui_j \rangle \frown r_j$
105b. | $r_i \frown \langle ui_i \rangle, \langle ui_j \rangle \frown r_i$:R • $\{r_i \frown \langle ui_i \rangle, \langle ui_j \rangle \frown r_j\} \subseteq$rs
105(b)i. ∧ **let** u_i,u_j:U•$\{u_i,u_i\} \subseteq$**obs_**Us(pln)∧ui_i=**uid_**U(u_i)∧ui_j=**uid_**U(u_j)
105(b)ii. **in** $ui_i \in$ iuis(u_j) ∧ $ui_j \in$ ouis(u_i) **end** }
105c. **in** rs **end**

Section 6.1 will continue with several examples (Example 44 on Page 61, Example 45 on Page 62, Example 46 on Page 63, Example 47 on Page 64 and Example 48 on Page 65) following up on the two examples of this section.

LECTURE 3

5 Discrete Perdurant Entities

From Wikipedia: *Perdurant: Also known as occurrent, accident or happening. Perdurants are those entities for which only a fragment exists if we look at them at any given snapshot in time. When we freeze time we can only see a fragment of the perdurant. Perdurants are often what we know as processes, for example 'running'. If we freeze time then we only see a fragment of the running, without any previous knowledge one might not even be able to determine the actual process as being a process of running. Other examples include an activation, a kiss, or a procedure.*

A discrete perdurant$_\delta$ is a perdurant which is a discrete entity. We shall consider the following discrete perdurants. actions (Sect. 5.2), events (Sect. 36), and discrete behaviours (Sect. 5.4).

Actions and events occur instantaneously, that is, in time, but taking no time, and to therefore be discrete action$_\delta$s and discrete event$_\delta$s.

5.1 On Domain Analysis: Discrete Perdurants

The domain analyser examines collections of discrete perdurants. (i) In doing so the domain analyser discovers and thus identifies and lists a number of perdurant properties. (ii) Each of the discrete perdurants examined usually satisfies only a subset of these properties. (iii) The domain analyser now groups discrete perdurant into collections such that each collection have its discrete perdurants satisfy the same set of properties, such that no two distinct collections are indexed, as it were, by the same set of properties, and such that all discrete perdurants are put in some collection. (iv) The domain analyser now classify collections as actions, events or behaviours, and assign signatures to distinct collections. That is how we assign signatures to discrete perdurants.

5.2 Actions

By a function$_\delta$ we understand a mathematical concept, a thing which when applied to a value, called its argument, yields a value, called its result. A discrete action$_\delta$ can be understood as a function invoked on a state value and is one that potentially changes that value. Other terms for action are function invocation$_\delta$ and function application$_\delta$.

Example: 32 Transport Net and Container Vessel Actions.

- *Inserting* and *removing* hubs and links in a net are considered actions.
- *Setting* the traffic signals for a hub (which has such signals) is considered an action.
- *Loading* and *unloading* containers from or unto the top of a container stack are considered actions. ■

5.2.1 Abstraction: On Modelling Domain Actions

We claim that we describe domain actions, but we actually describe functions, which are "somewhat far removed" from domains. So what are we actually claiming ? We are claiming that there is an interesting class of actions and that they can all be abstracted into one, possibly non-deterministic function whose properties are then claimed to "mimic" those of the actions in the interesting class.

5.2.2 Agents: An Aside on Actions

Think'st thou existence doth depend on time?
It doth; but actions are our epochs.

George Gordon Noel Byron,
Lord Byron (1788-1824) Manfred. Act II. Sc. 1.

"An action is something an agent does that was 'intentional under some description'" [23, Davidson 1980, Essay 3]. That is, actions are performed by agents. We shall not yet go into any deeper treatment of agency or agents. We shall do so in Sect. 5.4. Agents will here, for simplicity, be considered behaviours, and are treated in Sect. 5.4. As to the relation between intention and action we note that Davidson wrote: 'intentional under some description' and take that as our cue: the agent follows a script, that is, a behaviour description, and invokes actions accordingly, that is, follow, or honours that script.

5.2.3 Action Signatures

By an action signature we understand a quadruple: a function name, a function definition set type expression, a total or partial function designator ($\rightarrow$, respectively $\stackrel{\sim}{\rightarrow}$), and a function image set type expression: fct_name: A $\rightarrow$ Σ $(\rightarrow|\stackrel{\sim}{\rightarrow})$ Σ [$\times$ R], where $(X \mid Y)$ means either X or Y, and $[Z]$ means that for some signatures there may be a Z component meaning that the action also has the effect of "leaving" a type Z value.[28]

Example: 33 Action Signatures: Nets and Vessels.

insert_Hub: N$\rightarrow$H$\stackrel{\sim}{\rightarrow}$N;
remove_Hub: N$\rightarrow$HI$\stackrel{\sim}{\rightarrow}$N;
set_Hub_Signal: N$\rightarrow$HI$\stackrel{\sim}{\rightarrow}H\Sigma\stackrel{\sim}{\rightarrow}$N
load_Container: V$\rightarrow$C$\rightarrow$StackId$\stackrel{\sim}{\rightarrow}$V; and
unload_Container: V$\rightarrow$StackId$\stackrel{\sim}{\rightarrow}$(V$\times$C). ■

5.2.4 Action Definitions

There are a number of ways in which to characterise an action. One way is to characterise its underlying function by a pair of predicates: precondition: a predicate over function arguments — which includes the state, and postcondition: a predicate over function arguments, a proper argument state and the desired result state. If the precondition holds, i.e., is **true**, then the arguments, including the argument state, forms a proper 'input' to the action. If the postcondition holds, assuming that the precondition held, then the resulting state [and possibly a yielded, additional "result" (R)] is as they would be had the function been applied.

Example: 34 Transport Nets Actions. In Example 4 we gave an explicit example of an action: ins_H: Items 37–37d, while implicit references to net actions were made in the event predicates link_dis, pre_link_dis: Items 38–39c, post_link_dis (Items 38–39c): rem_L Item 42a and ins_L Items 42(c)i–42(c)ii. ■

What is not expressed, but tacitly assume in the above pre- and post-conditions is that the state, here n, satisfy invariant criteria before (i.e. n) and after (i.e., n') actions, whether these be implied by axioms or by well-formedness predicates. over parts. This remark applies to any definition of actions, events and behaviours.

Example: 35 Container Line: Remove Container. We refer to Example 23 (Pages 40–41).

106. The remove_Container_from_Vessel action applies to a vessel and a stack address and conditionally yields an updated vessel and a container.
    [a] We express the 'remove from vessel' function primarily by means of an auxiliary function remove_C_from_BS, remove_C_from_BS(obs_BS(v))(stid), and some further post-condition on the before and after vessel states (cf. Item 106d).
    [b] The remove_C_from_BS function yields a pair: an updated set of bays and a container.
    [c] When obs_erving the BayS from the updated vessel, v′, and pairing that with what is assumed to be a vessel, then one shall obtain the result of remove_C_from_BS(obs_BS(v))(stid).

[28] We shall not here speculate on "what happens" to that resulting value.

[d] Updating, by means of remove_C_from_BS(obs_BS(v))(stid), the bays of a vessel must leave all other properties of the vessel unchanged.

107. The pre-condition for remove_C_from_BS(bs)(stid) is

[a] that stid is a valid_address in bs, and

[b] that the stack in bs designated by stid is non_empty.

108. The post-condition for remove_C_from_BS(bs)(stid) wrt. the updated bays, bs′, is

[a] that the yielded container, i.e., c, is obtained, get_C(bs)(stid), from the top of the non-empty, designated stack,

[b] that the mereology of bs′ is unchanged, unchanged_mereology(bs,bs′). wrt. bs.

[c] that the stack designated by stid in the "input" state, bs, is popped, popped_designated_stack(bs,bs′)(stid), and

[d] that all other stacks are unchanged in bs′ wrt. bs, unchanged_non_designated_stacks(bs,bs′)(stid).

value
106. remove_C_from_V: V → StackId $\stackrel{\sim}{\rightarrow}$ (V×C)
106. remove_C_from_V(v)(stid) **as** (v′,c)
106c. (**obs_**Bs(**obs_**BS(v′),c)) = remove_C_from_BS(**obs_**Bs(**obs_**BS(v)))(stid)
106d. ∧ props(v)=props(v″)

106b. remove_C_from_BS: BS → StackId → (BS×C)
106a. remove_C_from_BS(bs)(stid) **as** (bs′,c)
107a. **pre**: valid_address(bs)(stid)
107b. ∧ non_empty_designated_stack(bs)(stid)
108a. **post**: c = get_C(bs)(stid)
108b. ∧ unchanged_mereology(bs,bs′)
108c. ∧ popped_designated_stack(bs,bs′)(stid)
108d. ∧ unchanged_non_designated_stacks(bs,bs′)(stid)

The props function was introduced in Sect. 4.2.4 on Page 45.

This example hints at *a theory of container vessel bays, rows and stacks*. More on that is found in Appendix B. There you will find explanations of the valid_address (Item 177 on Page 90), non_empty_designated_stack (Item 178), unchanged_mereology (Item 179), popped_designated_stack (Item 180) and unchanged_non_designated_stacks (Item 181) functions. ■

There are other ways of defining functions. But the form of these are not germane to the aims of these notes.

Modelling Actions

- We refer to Sect. 5.1: Formal Concept Analysis of Discrete Perdurants on Page 49.
- The domain describer has decided that an entity is a perdurant and is, or represents an

action: was *"done by an agent and intentionally under some description"* [23].

- The domain describer has further decided that the observed action is of a class of actions — of the "same kind" — that need be described.
- By actions of the 'same kind' is meant that these can be described by the same function signature and function definition.

- The domain describer must decide on the underlying function signature.
  - The argument type and the result type of the signature are those of either previously identified
    - parts and/or materials,
    - unique part identifiers, and/or
    - attributes.

- Sooner or later the domain describer must decide on the function definition.
  - The form[29] must be decided upon.
  - For pre/post-condition forms it appears to be convenient to have developed, "on the side", a theory of mereology for the part types involved in the function signature.

5.3 Events

By an event$_\delta$ we understand *a state change resulting indirectly from an unexpected application of a function, that is, that function was performed "surreptitiously"*.

Events can be characterised by a pair of (before and after) states, a predicate over these and, optionally, a time or time interval.

Events are thus like actions: change states, but are usually either caused by "previous" actions, or caused by "an outside action".

Example: 36 Events. *Container vessel:* A container falls overboard sometimes between times t and t'. *Financial service industry:* A bank goes bankrupt sometimes between times t and t'. *Health care:* A patient dies sometimes between times t and t'. *Pipeline system:* A pipe breaks sometimes between times t and t'. *Transportation:* A link "disappears" sometimes between times t and t'.

5.3.1 An Aside on Events

We may observe an event, and then we do so at a specific time or during a specific time interval. But we wish to describe, not a specific event but a class of events of "the same kind". In these notes we therefore do not ascribe time points or time intervals with the occurrences of events[30].

[29] Only the pre/post-condition form has so far been illustrated. Other function definition forms, incl. predicate functions, will emerge in further examples below.

[30] As we do not ascribe time points or time intervals with neither actions nor behaviours.

5.3.2 Event Signatures

An event signature$_\delta$ *is a predicate signature having an event name (evt), a pair of state types* $(\Sigma \times \Sigma)$*, a total function space operator* $(\rightarrow)$ *and a* **Bool***ean type constant: evt:* $(\Sigma\times\Sigma) \rightarrow$ **Bool**.

Sometimes there may be a good reason for indicating the type, ET, of an event cause value, if such a value can be identified: evt: ET $\times$ $(\Sigma \times \Sigma) \rightarrow$ **Bool**.

5.3.3 Event Definitions

An event definition$_\delta$ takes the form of *a predicate definition: a predicate name and argument list, usually just a state pair, an existential quantification over some part (of the state) or over some dynamic attribute of some part (of the state) or combinations of the above a pre-condition expression over the input argument(s), an implication symbol* $(\Rightarrow)$*, and a post-condition expression over the argument(s): evt*$(\sigma, \sigma') = \exists$ *(ev:ET)* $\bullet$ *pre_evt(ev)*$(\sigma) \Rightarrow$ *post_evt(ev)*(σ, σ').

There may be variations to the above form.

Example: 37 Road Transport System Event. Example 4, Sect. 2.7, Items 38–42(c)ii (Pages 24–25) exemplified an event definition.

Modelling Events

- We refer to Sect. 5.1: Formal Concept Analysis of Discrete Perdurants on Page 49.
- The domain describer has decided that an entity is a perdurant and is, or represents an event: occurred surreptitiously, that is, was not an action that was *"done by an agent and intentionally under some description"* [23].
  - The domain describer has further decided that the observed event is of a class of events — of the "same kind" — that need be described.
  - By events of the 'same kind' is meant that these can be described by the same predicate function signature and predicate function definition.
- First the domain describer must decide on the underlying predicate function signature.
  - The argument type and the result type of the signature are those of either previously identified
    - parts,
    - unique part identifiers, or
    - attributes.
- Sooner or later the domain describer must decide on the predicate function definition.
  - For predicate function definitions it appears to be convenient to have developed, "on the side", a theory of mereology for the part types involved in the function signature.

5.4 Discrete Behaviours

We shall distinguish between discrete behaviours (this section) and continuous behaviours (Sect. 6.2). Roughly discrete behaviours proceed in discrete (time) steps — where, in this section, we omit considerations of time. Each step corresponds to an action or an event or a time interval between these. Actions and events may take some (usually inconsiderable time), but the domain analyser has decided that it is not of interest to understand what goes on in the domain during that time (interval). Hence the behaviour is considered discrete.

Continuous behaviours are continuous in the sense of the calculus of mathematical analysis; to qualify as a continuous behaviour time must be an essential aspect of the behaviour.

Discrete behaviours can be modelled in many ways, for example using CSP [31]. MSC [35], Petri Nets [54] and Statechart [30]. We refer to Chaps. 12–14 of [6]. In these notes we shall use RSL/CSP.

5.4.1 What is Meant by 'Behaviour'?

We give two characterisations of the concept of 'behaviour'. a "loose" one and a "slanted one.

A loose characterisation runs as follows: by a behaviour$_\delta$ we understand a set of sequences of actions, events and behaviours.

A "slanted" characterisation runs as follows: by a behaviour$_\delta$ we shall understand either a sequential behaviour$_\delta$ consisting of a possibly infinite sequence of zero or more actions and events; or one or more communicating behaviour$_\delta$s whose output actions of one behaviour may synchronise and communicate with input actions of another behaviour; or two or more behaviours acting either as internal non-deterministic behaviour$_\delta$s ($\sqcap$) or as external non-deterministic behaviour$_\delta$s ($\sqcap\!\sqcap$).

This latter characterisation of behaviours is "slanted" in favour of a CSP, i.e., a communicating sequential behaviour, view of behaviours. We could similarly choose to "slant" a behaviour characterisation in favour of Petri Nets, or MSCs, or Statecharts, or other.

5.4.2 Behaviour Narratives

Behaviour narratives may take many forms. A behaviour may best be seen as composed from several interacting behaviours. Instead of narrating each of these, as was done in Example 4, one may proceed by first narrating the interactions of these behaviours. Or a behaviour may best be seen otherwise, for which, therefore, another style of narration may be called for, one that "traverses the landscape" differently. Narration is an art. Studying narrations – and practice – is a good way to learn effective narration.

5.4.3 Channels

We remind the reader that we are focusing exclusively on domain behaviours. Domain behaviours, as we shall see in Sect. 5.4.6, take their "root" in parts. We shall find, even when "parts" take the form of concepts, that these do not "overlap". They may share properties, but we can consider them "disjoint".[31] Hence communication between processes can be

[31] These previous sentences really beg more careful, at times philosophical arguments. Once this present, and at present, excluding Sect. 7, 90 page document, has found a reasonably stable form (after now 4–5 iterations, we plan to separate out a number of the places, such as this, which warrant careful motivations.

thought of as communication between "disjoint parts", and, as such, can be abstracted as taking place in a non-physical medium which we shall refer to as channels.

By a channel$_\delta$ we shall understand *a means of communicating entities between [two] behaviours.*

To express channel communications we, at present, make use of **RSL** [28]'s **out**put (ch ! v) / **in**put (ch ?) clauses and **channel** declarations,

```
type    M
channel ch M,
value   ch!v, ch?,
```

Variations of the above clauses are

```
type    ChIdx, ChJdx
channel {ch[i]|i:ChIdx•𝒫(i,...)}:M, {ch[i,j]|i:ChIdx,j:ChJdx•𝒫(i,j,...)}:M
value   ch[i]!v, ch[i]?, ch[i,j]!v, ch[i,j]?
```

where $\mathcal{P}$ is a suitable predicate over channel indices and possibly global domain values.

5.4.4 Behaviour Signatures

By a behaviour signature$_\delta$ we shall understand *a a function signature augmented by a clause which declares the* **in** *channels on which the function accepts inputs and the* **out** *channels on which the function offers output.*

value behaviour: A → **in** in_chs **out** out_chs B

where (i) the form **in** in_chs **out** out_chs may be just **in** in_chs or **out** out_chs or both **in** in_chs **out** out_chs that is, behaviour accepts input(s), or offers output(s), or both; where (ii) A typically is of the forms **Unit** if the behaviour "takes no arguments", that is: behaviour(), or PI×P if the behavior is directly based on a part, p:P, for that is: behaviour(uid_P(p),p); where (iii) in_chs and out_chs are of the form either

- ch or
- {ch[i]|i:ChIdx•$\mathcal{Q}$(i,...)} or
- {ch[i,j]|i:ChIdx,j:ChJdx•$\mathcal{R}$(i,j,...)},

$\mathcal{Q}$, $\mathcal{R}$ are appropriate predicates; and where (iv) either B is either just **Unit** when the behaviour is typically a never-ending (i.e., cyclic) behaviours, or is some result type C.

5.4.5 Behaviour Definitions

This section is about the basic form of behaviour function definitions. We shall only be concerned with behaviours which define part behaviours.

By a part behaviour$_\delta$ we shall understand *a behaviour whose state is that of the part for which it is the behaviour.*

There are basically two cases for which we are interested in the form of the behaviour definition: (i) the atomic part behaviour, and (ii) the composite part behaviour.

[1] Atomic Part Behaviours: Let p:P be an atomic part of type P. Then the basic form of a cyclic atomic behaviour definition is

```
value
    atomic_core_part_behaviour(uid_P(p))(p) ≡
        let p′ = 𝒜(uid_P(p))(p) in
        atomic_core_part_behaviour(uid_P(p))(p′) end
        post: uid_P(p) = uid_P(p′),

    𝒜: PI → P → in ... out ... P,
```

where $\mathcal{A}$ usually is a terminating function which synchronises and communicates with other part behaviours.

Example: 38 Atomic Part Behaviours. Example 4, Sect. 2.8.6 on Page 28 and Sect. 2.8.7 on Page 29 illustrates cyclic atomic behaviours: vehicle at Hub: Items 65–65d, on Page 28, vehicle on Link: Items 64–68, on Page 29 and monitor: Items 69–71d, on Page 30. ■

[2] Composite Part Behaviours: Let p:P be a composite part of type P. Then the basic form of a cyclic atomic behaviour definition for p:P is

```
value
    composite_part_behaviour(uid_P(p))(p) ≡
        composite_core_part_behaviour(uid_P(p))(p)
      ‖ { part_behaviour(uid_P(p′))(p′)|p′:P•p′ ∈ obs_(p)}

    composite_core_part_behaviour: PI → P → in ... out ... Unit
    composite_core_part_behaviour(uid_P(p))(p) ≡
        let p′ = 𝒞(uid_P(p))(p) in
        composite_core_part_behaviour(uid_P(p))(p′) end
        post: uid_P(p) = uid_P(p′)

    𝒞: PI → P → in ... out ... P,
```

where $\mathcal{C}$ usually is a terminating function which synchronises and communicates with other part behaviours.

Example: 39 Compositional Behaviours. Example 4, Sect. 2.8.3 on Page 27 illustrated compositionality, cf. Items 59– 59b on Page 27. ■

The next section illustrates the basic principles that we recommend when modelling behaviours of domains consisting of composite and atomic parts.

5.4.6 A Model of Parts and Behaviours

How often have you not "confused", linguistically, the perdurant notion of a train process: progressing from railway station to railway station, with the endurant notion of the train, say as it appears listed in a train time table, or as it is being serviced in workshops, etc. There is a reason for that — as we shall now see: parts may be considered syntactic quantities denoting

semantic quantities. We therefore describe a general model of parts of domains and we show that for each instance of such a model we can 'compile' that instance into a CSP'program'.

The example additionally has a more general aim, *namely that of showing that to every mereology (or parts) there is a λ-expression here in the form of basically a* **CSP** [31] *program.*

Example: 40 Syntax and Semantics of Mereology.

[1] A Syntactic Model of Parts:

109. The *whole* contains a set of *parts*.
110. *Parts* are either *atomic* or *composite*.
111. From *composite parts* one can observe a set of *parts*.
112. All *parts* have *unique identifiers*

type
109. W, P, A, C
110. P = A | C
value
111. **obs_**Ps: (W|C) → P-**set**
type
112. PI
value
112. **uid_**Π: P → Π

113. From a *whole* and from any *part* of that *whole* we can e**xtr**act all contained *parts*.
114. Similarly one can e**xtr**act the *unique identifiers* of all those contained *parts*.
115. Each part may have a *mereology* which may be "empty".
116. A *mereology*'s *unique part identifiers* must refer to some other parts other than the part itself.

value
113. xtr_Ps: (W|P) → P-**set**
113. xtr_Ps(w) ≡ {xtr_Ps(p)|p:P•p ∈ **obs_**Ps(p)}
113. **pre**: is_W(p)
113. xtr_Ps(p) ≡ {xtr_Ps(p)|p:C•p∈ **obs_**Ps(p)}∪{p}
113. **pre**: is_P(p)
114. xtr_Πs: (W|P) → Π-**set**
114. xtr_Πs(wop) ≡ {**uid_**P(p)|p ∈ xtr_Ps(wop)}
115. **mereo_**P: P → Π-**set**
axiom
116. ∀ w:W
116. **let** ps = xtr_Ps(w) **in**
116. ∀ p:P • p ∈ ps • ∀ π:Π • π ∈ **mereo_**P(p) ⇒ π ∈ xtr_Πs(p) **end**

117. An attribute map of a *part* associates with *attribute names*, i.e., *type names*, their *values*, whatever they are.

118. From a *part* one can extract its attribute map.

119. Two *parts share attributes* if their respective attribute maps share *attribute names*.

120. Two *parts share properties* if the y

    [a] either *share attributes*

    [b] or the *unique identifier* of one is in the *mereology* of the other.

type
117. AttrNm, AttrVAL,
117. AttrMap = AttrNm $\underset{m}{\rightarrow}$ AttrVAL
value
118. **attr_**AttrMap: P → AttrMap
119. share_Attributes: P×P → **Bool**
119. share_Attributes(p,p′) ≡
119. **dom attr_**AttrMap(p) ∩
119. **dom attr_**AttrMap(p′) ≠ {}
120. share_Properties: P×P → **Bool**
120. share_Properties(p,p′) ≡
120a. share_Attributes(p,p′)
120b. ∨ **uid_**P(p) ∈ **mereo_**P(p′)
120b. ∨ **uid_**P(p′) ∈ **mereo_**P(p)

[2] A Semantics Model of Parts:

121. We can define the set of two element sets of *unique identifiers* where

    - one of these is a *unique part identifier* and
    - the other is in the mereology of some other *part*.
    - We shall call such two element "pairs" of *unique identifiers* connectors.
    - That is, a connector is a two element set, i.e., "pairs", of *unique identifiers* for which the identified parts share properties.

122. Let there be given a 'whole', w:W.

123. To every such "pair" of *unique identifiers* we associate a *channel*

    - or rather a position in a matrix of *channels* indexed over the "pair sets" of *unique identifiers*.
    - and communicating messages m:M.

type
121. K = Π-**set axiom** ∀ k:K•**card** k=2
value

121. xtr_Ks: (W|P) → K-**set**
121. xtr_Ks(wop) ≡
121. **let** ps = xtr_Ps(w) **in**
121. {{**uid_**P(p),π}|p:P,π:Π•p∈ ps ∧ ∃ p′:P•p′≠p∧π=**uid_**P(p′) ∧ **uid_**P(p)∈uid_P(p′)} **end**

122. w:W

123. **channel** {ch[k]|k:xtr_Ks(w)}:M

124. Now the 'whole' *behaviour* whole is the parallel composition of *part processes*, one for each of the immediate parts of the *whole*.

125. A *part process* is

    [a] either an *atomic part process*, atom, if the *part* is an *atomic part*,

    [b] or it is a *composite part process*, comp, if the *part* is a *composite part*.

124. whole: W → **Unit**
124. whole(w) ≡ ‖ {part(**uid_**P(p))(p) | p:P•p ∈ xtr_Ps(w)}

125. part: π:Π → P → **Unit**
125. part(π)(p) ≡
125a. is_A(p) → atom(π)(p),
125b. _ → comp(π)(p)

126. A *composite process*, part, consists of

    [a] a *composite core process*, comp_core, and

    [b] the parallel composition of *part processes* one for each *contained part* of part.

.

value
126. comp: π:Π → p:P → **in**,**out** {ch[{π,π′}|{π′∈ **mereo_**P(p)}]} **Unit**
126. comp(π)(p) ≡
126a. comp_core(π)(p) ‖
126b. ‖ {part(**uid_**P(p′))(p′) | p′:P•p′ ∈ **obs_**Ps(p)}

127. An *atomic process* consists of just an *atomic core process*, *atom_core*

127. atom: π:Π → p:P → **in**,**out** {ch[{π,π′}|{π′∈ **mereo_**P(p)}]} **Unit**
127. atom(π)(p) ≡ atom_core(π)(p)

128. The core behaviours both

[a] update the part properties and

[b] recurses with the updated properties,

[c] without changing the part identification.

We leave the update action undefined.

value
128. core: π:Π $\rightarrow$ p:P $\rightarrow$ **in**,**out** {ch[{π,π'}|{$\pi'\in$ **mereo_**P(p)}]} **Unit**
128. core(π)(p) $\equiv$
128a. **let** p$'$ = update(π)(p)
128b. **in** core(π)(p$'$) **end**
128b. **assert:** **uid_**P(p)=π=**uid_**P(p$'$)

The model of parts can be said to be a syntactic model. No meaning was "attached" to parts. The conversion of parts into CSP programs can be said to be a semantic model of parts, one which to every part associates a behaviour which evolves "around" a state which is that of the properties of the part.

6 Continuous Entities

There are two kinds of continuous entities: materials (Sect. 6.1) and continuous behaviours (Sect. 6.2). By a material$_\delta$ we small mean a continuous endurant, a manifest entity which typically varies in shape and extent. By a continuous behaviour$_\delta$ we small mean a continuous perdurant, which we may think of as a function from continuous $\mathbb{T}$ime to some structure, simple or complicated, of parts and materials.

6.1 Materials

Let us start with examples of materials.

Example: 41 **Materials.** Examples of endurant continuous entities are such as coal, air, natural gas, grain, sand, iron ore[32], minerals, crude oil, solid waste, sewage, steam and water. ■

The above materials are either liquid materials (crude oil, sewage, water), gaseous materials (air, gas, steam), or granular materials (coal, grain, sand, iron ore, mineral, or solid waste).

Endurant continuous entities, or materials as we shall call them, are the core endurants of process domains, that is, domains in which those materials *form the basis* for their *"raison d'être"*.

6.1.1 Materials-based Domains

By a materials based domain$_\delta$ we shall mean a domain *many of whose parts serve to transport materials, and some of whose actions, events and behaviours serve to monitor and control the part transport of materials.*

[32]– whether molten or not

Example: 42 Material Processing. (i) Oil or gas materials are ubiquitous to pipeline systems — so pipeline systems are oil or gas-based systems. (ii) Sewage is ubiquitous to waste management systems — so waste management systems are sewage-based systems. (iii) Water is ubiquitous to systems composed from reservoirs, tunnels and aqueducts which again are ubiquitous to hydro-electric power plants, irrigation systems or water supply utilities — so hydro-electric power plants, irrigation systems and water supply utilities are water-based systems. ■

Ubiquitous means 'everywhere'. A continuous entity, that is, a material is a core material, if it is "somehow related" to one or more parts of a domain.

6.1.2 "Somehow Related" Parts and Materials

We explain our use of the term "somehow related".

Example: 43 Somehow Related Materials and Parts. With `teletype font` we designate materials and with *slanted font* we imply parts or part processes. (i) `Oil` is pumped from *wells*, runs through *pipes*, is "lifted" by *pumps*, diverted by *forks*, "runs together" by means of *joins*, and is delivered to *sinks*. (ii) `Grain` is delivered to silos by trucks, piped through a network of pipes, forks and valves to vessels, etc. (iii) `Minerals` are *mined*, *conveyed* by *belts* to *lorries* or *trains* or *cargo vessels* and finally *deposited*. (iv) `Iron ore`, for example, is *'conveyed'*[33] into *smelters*, *'roasted'*, *'reduced'* and *'fluxed'*, *'mixed'* with other mineral ores to produce a molten, pure metal, which is then *'collected'* into *ingots*. ■

6.1.3 Material Observers

When analysing domains a key question, in view of the above notion of core continuous endurants (i.e., materials) is therefore: does the domain embody a notion of core continuous endurants (i.e., materials); if so, then identify these "early on" in the domain analysis. Identifying materials — their types and attributes — is slightly different from identifying discrete endurants, i.e., parts.

Example: 44 Pipelines: Core Continuous Endurant. We continue Examples 30 on Page 45 and 31 on Page 46. The core continuous endurant, i.e., material, of (say oil) pipelines is, yes, oil:

type
  O **material**
value
  obs_O: PLN → O

The keyword **material** is a pragmatic. ■

Materials are "few and far between" as compared to parts, we choose to mark the type definitions which designate materials with the keyword **material**. In contrast, we do not mark the type definitions which designate parts with the keyword **discrete**. First we do not associate the notion of atomicity or composition with a material. Materials are continuous. Second, amongst the attributes, none have to do with geographic (or cadestral) matters. Materials are moved. And materials have no unique identification or mereology. No "part"[34]

[33] The single quote terms are verbs to which there corresponds part processes.
[34] The term part is not the technical term for discrete endurants, but the more conventional term.

of a material distinguishes it from other "parts". But they do have other attributes when occurring in connection with, that is, related to parts, for example, volume or weight.

Example: 45 Pipelines: Parts and Materials. We continue Examples 30 on Page 45 and 31 on Page 46.

129. From an oil pipeline system one can, amongst others,
    [a] observe the finite set of all its pipeline bodies,
    [b] units are composite and consists of a unit,
    [c] and the oil, even if presently, at time of observation, empty of oil.
130. Whether the pipeline is an oil or a gas pipeline is an attribute of the pipeline system.
    [a] The volume of material that can be contained in a unit is an attribute of that unit.
    [b] There is an auxiliary function which estimates the volume of a given "amount" of oil.
    [c] The observed oil of a unit must be less than or equal to the volume that can be contained by the unit.

type
129. PLS, B, U, Vol
129. O **material**
value
129a. **obs_**Bs: PLS → B-**set**
129b. **obs_**U: B → U
129c. **obs_**O: B → O
130. **attr_**PLS_Type: PLS → {"oil"|"gas"}
130a. **attr_**Vol: U → Vol
130b. vol: O → Vol
axiom
130c. ∀ pls:PLS,b:B•b ∈ **obs_**Bs(pls)⇒vol(**obs_**O(b))≤**attr_**Vol(**obs_**U(b))

Notice how bodies are composite and consists of a discrete, atomic part, the unit, and a material endurant, the oil. We refer to Example 46 on the next page. ■

6.1.4 Material Properties

These are some of the key concerns in domains focused on materials: transport, flows, leaks and losses, and input to systems and output from systems, Other concerns are in the direction of dynamic behaviours of materials focused domains (mining and production), including stability, periodicity, bifurcation and ergodicity. In these notes we shall, when dealing with systems focused on materials, concentrate on modelling techniques for transport, flows, leaks and losses, and input to systems and output from systems.

Formal specification languages like **Alloy** [36], **Event B** [1] , CASL [22]**CafeOBJ** [26], **RAISE** [29], **VDM** [15, 16, 24] and **Z** [68] do not embody the mathematical calculus notions of continuity, hence do not "exhibit" neither differential equations nor integrals. Hence cannot formalise dynamic systems within these formal specification languages.

Example: 46 Pipelines: Parts and Material Properties. We refer to Examples 30 on Page 45, 31 on Page 46 and 45 on the facing page.

131. Properties of pipeline units additionally include such which are concerned with flows (F) and leaks (L) of materials[35]:
    [a] current flow of material into a unit input connector,
    [b] maximum flow of material into a unit input connector while maintaining laminar flow,
    [c] current flow of material out of a unit output connector,
    [d] maximum flow of material out of a unit output connector while maintaining laminar flow,
    [e] current leak of material at a unit input connector,
    [f] maximum guaranteed leak of material at a unit input connector,
    [g] current leak of material at a unit input connector,
    [h] maximum guaranteed leak of material at a unit input connector,
    [i] current leak of material from "within" a unit,
    [j] maximum guaranteed leak of material from "within" a unit.
132. There are "the usual" arithmetic and comparison operators of flows and leaks, and there is a smallest detectable (flow and) leak.

type
132. F, L

value
132. $\oplus,\ominus$: (F|L)×(F|L) → (F|L)
132. $<,\leq,=$: (F|L)×(F|L) → **Bool**
132. $\otimes$: (F|L)×**Real** → (F|L)
132. /: (F|L)×(F|L) → **Real**
132. ℓ_0:L

131a. **attr_**cur_iF: U → UI → F
131b. **attr_**max_iF: U → UI → F
131c. **attr_**cur_oF: U → UI → F
131d. **attr_**max_oF: U → UI → F
131e. **attr_**cur_iL: U → UI → L
131f. **attr_**max_iL: U → UI → L
131g. **attr_**cur_oL: U → UI → L
131h. **attr_**max_oL: U → UI → L
131i. **attr_**cur_L: U → L
131j. **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 as dynamic attributes.

133. Properties of pipeline materials may additionally include
    [a] kind of material[36],
    [b] paraffins,
    [c] naphtenes,
    [d] aromatics,
    [e] asphatics,
    [f] viscosity,
    [g] etcetera.

[35] Here we think of flows and leaks as measured in terms of volume per time unit.
[36] For example `Brent Blend` Crude Oil

We leave it to the reader to provide the formalisations. ■

6.1.5 Material Laws of Flows and Leaks

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 modelling. That is in contrast to incorporating such notions of flows and leaks in requirements modelling where one has to show implementability.

Modelling flows and leaks is important to the modelling of materials-based domains.

Example: 47 Pipelines: Intra Unit Flow and Leak Law. We continue our line of Pipeline System examples (cf. the opening line of Example 46 on the previous page).

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

135. 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
134. ∀ pls:PLS,b:B\We\Si,u:U •
134. b ∈ **obs_**Bs(pls)∧u=**obs_**U(b)⇒
134. **let** (iuis,ouis) = **mereo_**U(u) **in**
135. sum_cur_iF(iuis)(u) =
135a. sum_cur_iL(iuis)(u)
135b. ⊕ **attr_**cur_L(u)
135c. ⊕ sum_cur_oF(ouis)(u)
135d. ⊕ sum_cur_oL(ouis)(u)
134. **end**

136. The sum_cur_iF (cf. Item 135) sums current input flows over all input connectors.

137. The sum_cur_iL (cf. Item 135a) sums current input leaks over all input connectors.

138. The sum_cur_oF (cf. Item 135c) sums current output flows over all output connectors.

139. The sum_cur_oL (cf. Item 135d) sums current output leaks over all output connectors.

136. sum_cur_iF: UI-**set** → U → F
136. sum_cur_iF(iuis)(u) ≡ ⊕ ⟨**attr_**cur_iF(ui)(u)|ui:UI•ui ∈ iuis⟩
137. sum_cur_iL: UI-**set** → U → L
137. sum_cur_iL(iuis)(u) ≡ ⊕ ⟨**attr_**cur_iL(ui)(u)|ui:UI•ui ∈ iuis⟩
138. sum_cur_oF: UI-**set** → U → F
138. sum_cur_oF(ouis)(u) ≡ ⊕ ⟨**attr_**cur_iF(ui)(u)|ui:UI•ui ∈ ouis⟩

139. sum_cur_oL: UI-**set** → U → L
139. sum_cur_oL(ouis)(u) ≡ ⊕ ⟨**attr_**cur_iL(ui)(u)|ui:UI•ui ∈ ouis⟩
⊕: (F×F)|F* → F | (L×L)|L* → L

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

Example: 48 Pipelines: Inter Unit Flow and Leak Law.

140. 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.

140. ∀ pls:PLS,b,b′:B,u,u′:U•
140. {b,b′}⊆**obs_**Bs(pls)∧b≠b′∧u′=**obs_**U(b′)
140. ∧ **let** (iuis,ouis)=**mereo_**U(u),(iuis′,ouis′)=**mereo_**U(u′),
140. ui=**uid_**U(u),ui′=**uid_**U(u′) **in**
140. ui ∈ iuis ∧ ui′ ∈ ouis′ ⇒
140a. **attr_**cur_oF(us′)(ui′) ⊖ **attr_**leak_oF(us′)(ui′)
140b. = **attr_**cur_iF(us)(ui) ⊕ **attr_**leak_iF(us)(ui)
140. **end**
140. **comment:** b′ precedes b

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. ■

6.2 Continuous Behaviours

This section is still under research and development.

The aim of this section is to relate discrete behaviour domain models of some fragments of a domain to continuous behaviour domain models of other fragments of that domain.

By a continuous behaviour model$_\delta$ we mean *a domain description that emphasises the behaviour of materials, that is, how they flow through parts, and related matters.*

6.2.1 Fluid Dynamics

Continuous behaviour domain models classically express the fluid dynamics$_\delta$ of flows of fluids, that is, the natural science of liquids and gasses.

The natural science of fluids (from `Wikipedia:`) *"are based on foundational axioms of fluid dynamics which are the conservation laws, specifically, conservation of mass, conservation of linear momentum (also known as Newton's Second Law of Motion), and conservation of energy (also known as First Law of Thermodynamics). These are based on classical mechanics. They are expressed using the Reynolds Transport Theorem."*

[1] Descriptions of Continuous Domain Behaviours: We are not going to exemplify such descriptive natural science models. Their mathematics, besides being elegant and beautiful, includes familiarity with Bernoulli Equations, Navier Stokes Equations, etc.

For continuous behaviour domain models we shall refer to such mathematical models of the natural science of fluids.

[2] Prescriptions of Required Continuous Domain Behaviours: By a prescriptive domain model$_\delta$ we mean *a desirable behaviour specification as in, for example, a requirements prescription* of a continuous time dynamic system.

We are also not going to illustrate prescriptive domain models. Their mathematics, besides also being elegant and beautiful, is based on the descriptive natural science models; but are now part of the engineering realm of *Control Theory*. It includes such disciplines as fuzzy control [47], stochastic control [38] and adaptive control [3], etc.

Example: 49 Pipelines: Fluid Dynamics and Automatic Control. We refer to Example 50. In that example, next, we expect domain models for the fluid dynamics of individual pipeline units: wells, pumps, pipes, valves, forks, joins and sinks, as well as models (one or more) for sequences of such units, extending, preferably to entire nets: from wells to sinks. And we expect requirements description models again for each of some of the individual units: pumps and valves in particular: when they need and how they are controlled: regulating pumps and valves and which unit attributes need be monitored. ■

6.2.2 A Pipeline System Behaviour

We shall model the behaviours of a composite pipeline system. We shall be using basically the same form of the description as first illustrated in Sects. 2.8.2—2.8.7 (Pages 26–30) of Example 4. That system, Sects. 2.8.2—2.8.7, can be interpreted as illustrating the central monitoring of vehicles spread over a wide geographical area. The system to be illustrated in Example 50 can likewise be interpreted as illustrating the central monitoring of pipeline units (and their oil) spread over a wide geographical area.

Example: 50 A Pipeline System Behaviour. We consider (cf. Examples 30 on Page 45 and 31 on Page 46) the pipeline system units to represent also the following behaviours: pls:PLS, Item 129a on Page 62, to also represent the system process, pipeline_system, and for each kind of unit, cf. Example 30, there are the unit processes: unit, well (Item 98c on Page 46), pipe (Item 98a), pump (Item 98a), valve (Item 98a), fork (Item 98b), join (Item 98b) and sink (Item 98d on Page 46).

```
channel
    { pls_u_ch[ui]:ui:UI•i ∈ UIs(pls) } MUPLS
    { u_u_ch[ui,uj]:ui,uj:UI•{ui,uj}⊆UIs(pls) } MUU
type
    MUPLS, MUU
value
    pipeline_system: PLS → in,out { pls_u_ch[ui]:ui:UI•i ∈ UIs(pls) }  Unit
    pipeline_system(pls) ≡ ‖ { unit(u)|u:U•u ∈ obs_Us(pls) }
    unit: U → Unit
    unit(u) ≡
```

98c. is_We(u) → well(uid_U(u))(u),
98a. is_Pu(u) → pump(uid_U(u))(u),
98a. is_Pi(u) → pipe(uid_U(u))(u),
98a. is_Va(u) → valve(uid_U(u))(u),
98b. is_Fo(u) → fork(uid_U(u))(u),
98b. is_Jo(u) → join(uid_U(u))(u),
98d. is_Si(u) → sink(uid_U(u))(u)

We illustrate essentials of just one of these behaviours.

98b. fork: ui:UI → u:U → **out**,**in** pls_u_ch[ui],
in { u_u_ch[iui,ui] | iui:UI • iui ∈ sel_UIs_in(u) }
out { u_u_ch[ui,oui] | iui:UI • oui ∈ sel_UIs_out(u) } **Unit**
98b. fork(ui)(u) ≡
98b. **let** u′ = core_fork_behaviour(ui)(u) **in**
98b. fork(ui)(u′) **end**

The core_fork_behaviour(ui)(u) distributes what oil (or gas) in receives, on the one input sel_UIs_in(u) = {iui}, along channel u_u_ch[iui] to its two outlets sel_UIs_out(u) = {oui_1,oui_2}, along channels u_u_ch[oui_1], u_u_ch[oui_2].

The core_···_behaviour[s](ui)(u) also communicate with the pipeline_system behaviour. What we have in mind here is to model a traditional supervisory control and data acquisition, SCADA system.

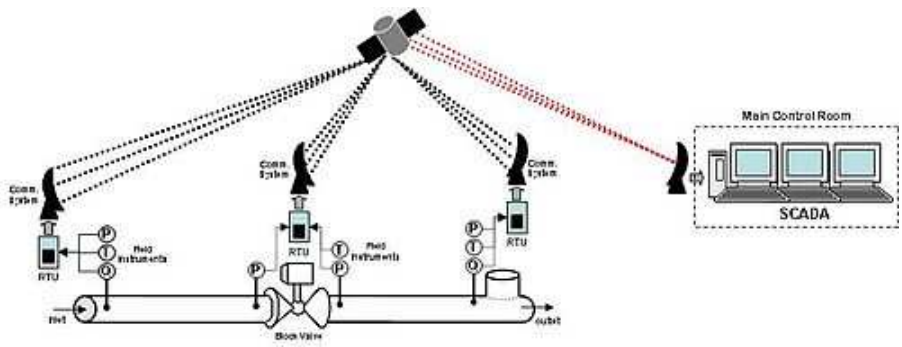


Figure 2: A supervisory control and data acquisition system

141. SCADA is then part of the scada_pipeline_system behaviour.

141. scada_pipeline_system: PLS →
141. **in**,**out** { pls_u_ch[ui]:ui:UI•i ∈ UIs(pls) } **Unit**
141. scada_pipeline_system(pls) ≡
141. scada(props(pls)) ‖ pipeline_system(pls)

props was defined in Sect. 4.2.4 Page 45.

We refer to Example 49 on the facing page: for all the core_···_behaviours we expect the scada monitor to be expressed in terms of a prescriptive domain model which prescribes some optimal form of control of the pipeline net.

142. scada non-deterministically (internal choice, ⌈⌉), alternates between continually

   [a] doing own work,
   [b] acquiring data from pipeline units, and
   [c] controlling selected such units.

```
type
142.   Props
value
142.   scada: Props → in,out { pls_ui_ch[ui] | ui:UI•ui ∈ ∈ uis }  Unit
142.   scada(props) ≡
142a.      scada(scada_own_work(props))
142b.    ⌈⌉ scada(scada_data_acqui_work(props))
142c.    ⌈⌉ scada(scada_control_work(props))
```

We leave it to the readers imagination to describe scada_own_work.

143. The scada_data_acqui_work

   [a] non-deterministically, external choice, [], offers to accept data,
   [b] and scada_input_updates the scada state —
   [c] from any of the pipeline units.

```
value
143.   scada_data_acqui_work: Props → in,out { pls_ui_ch[ui] | ui:UI•ui ∈ ∈ uis }  Props
143.   scada_data_acqui_work(props) ≡
143a.      [] { let (ui,data) = pls_ui_ch[ui] ? in
143b.           scada_input_update(ui,data)(props) end
143c.           | ui:UI • ui ∈ uis }

143b.   scada_input_update: UI × Data → Props → Props
type
143a.   Data
```

144. The scada_control_work

   [a] analyses the scada state (props) thereby selecting a pipeline unit, ui, and the controls, ctrl, that it should be subjected to;
   [b] informs the units of this control, and
   [c] scada_output_updates the scada state.

```
144.   scada_control_work: Props → in,out { pls_ui_ch[ui] | ui:UI•ui ∈ ∈ uis }  Props
144.   scada_control_work(props) ≡
144a.      let (ui,ctrl) = analyse_scada(ui,props) in
144b.      pls_ui_ch[ui] ! ctrl ;
144c.      scada_output_update(ui,ctrl)(props) end
```

144c. scada_output_update UI × Ctrl → Props → Props
type
144a. Ctrl

We leave it to the reader to suggest definitions of the core SCADA functions: scada_own_work, analyse_scada and scada_internal_update These functions depend on the system being monitored & controlled. Typically they are formulated in the realm of automatic control theory.

7 Requirements Engineering

We shall give a terse overview of some facets of requirements engineering. Namely those which "relate" domain engineering to requirements engineering. The relation is the following: one can "derive", not automatically, but systematically, domain requirements and significant aspects of interface requirements from domain descriptions.

7.1 A Requirements "Derivation"

7.1.1 Definition of Requirements

IEEE Definition of 'Requirements'

By a requirements we understand (cf. IEEE Standard 610.12 [34]): *"A condition or capability needed by a user to solve a problem or achieve an objective"*.

7.1.2 The Machine = Hardware + Software

By 'the machine' we shall understand the software to be developed and hardware (equipment + base software) to be configured for the domain application.

7.1.3 Requirements Prescription

The core part of the requirements engineering of a computing application is the requirements prescription. A requirements prescription tells us which parts of the domain are to be supported by 'the machine'. A requirements is to satisfy some goals. Usually the goals cannot be prescribed in such a manner that they can serve directly as a basis for software design. Instead we derive the requirements from the domain descriptions and then argue (incl. prove) that the goals satisfy the requirements. In this paper we shall not show the latter but shall show the former.

7.1.4 Some Requirements Principles

The "Golden Rule" of Requirements Engineering

Prescribe only such requirements that can be objectively shown to hold for the designed software.

An "Ideal Rule" of Requirements Engineering

When prescribing (including formalising) requirements, formulate tests (theorems, properties for model checking) whose actualisation show adherence to the requirements.

We shall not show adherence to the above rules.

7.1.5 A Decomposition of Requirements Prescription

We consider three forms of requirements prescription: the domain requirements, the interface requirements and the machine requirements. Recall that the machine is the hardware and software (to be required). Domain requirements are those whose technical terms are from the domain only. Machine requirements are those whose technical terms are from the machine only. Interface requirements are those whose technical terms are from both.

7.1.6 An Aside on Our Example

We shall continue our "ongoing" example. Our requirements is for a tollway system. By a requirements goal we mean *"an objective the system under consideration should achieve"* [64]. The goals of having a tollway system are: to decrease transport times between selected hubs of a general net; and to decrease traffic accidents and fatalities while moving on the tollway net as compared to comparable movements on the general net. The tollway net, however, must be paid for by its users. Therefore tollway net entries and exits occur at tollway plazas with these plazas containing entry and exit toll collectors where tickets can be issued, respectively collected and travel paid for. We shall very briefly touch upon these toll collectors, in the Extension part (as from Page 74) of the next section, Sect. 7.2. So all the other parts of the next section serve to build up to the Extension section, Sect. 7.2.4 on Page 74.

7.2 Domain Requirements

Domain requirements cover all those aspects of the domain — parts and materials, actions, events and behaviours — which are to be supported by 'the machine'. Thus domain requirements are developed by systematically "revising" cum "editing" the domain description: which parts are to be **projected:** left in or out; which general descriptions are to be **instantiated** into more specific ones; which non-deterministic properties are to be made more **determinate**; and which parts are to be **extended** with such computable domain description parts which are not feasible without IT.

Thus projection, instantiation, determination and extension are basic engineering tasks of domain requirements engineering. An example may best illustrate what is at stake. The example is that of a tollway system — in contrast to the general nets. See Fig. 3 on the next page.

The links of the general net of Fig. 3 on the facing page are all two-way links, so are the plaza-to-tollway links of the tollway net of Fig. 3. The tollway links are all one-way links. The hubs of the general net of Fig. 3 are assumed to all allow traffic to move in from any link and onto any link. The plaza hubs do not show links to "an outside" — but they are assumed. Vehicles enter the tollway system from the outside and leave to the outside. The tollway hubs

allow traffic to move in from the plaza-to-tollway link and back onto that or onto the one or two tollway links emanating from that hub, as well as from tollway links incident upon that hub onto tollway links emanating from that hub or onto the tollway-to-plaza link.

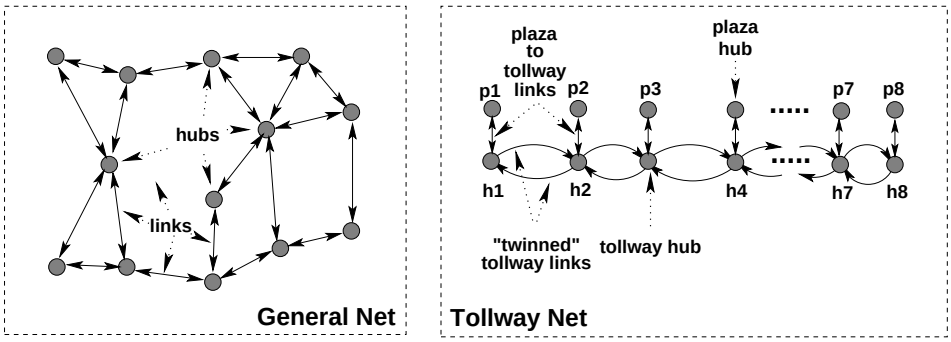


Figure 3: General and Tollway Nets

7.2.1 Projection

By domain projection$_\delta$ we mean that a subset of the domain description is kept. In the tollway example we actually keep all the parts, their properties and therefore the types and functions derived from these, Thus we keep: 1a–1c (N, F, M) 2–2b (HS, LS), 5a–6b (Hs, Ls, H, L), 7a–7b (HI, LI), 10a–10c (LΣ, LΩ, LEN, LOC) and 11a–11c (HΣ, HΩ, LOC) , 3–4b, 7c (VS, Vs, V), 8a–9b (**mereo_**L), 12a–12(a)iii, 13 (VP, atH, onL, FRAC, **attr_**VP), We do not keep any actions or events (!), But we keep the behaviours: 59–59b (trs), 61–63 (trs, veh, mon), 65–65d, 64–68 (veh), 69–71d (mon).

7.2.2 Instantiation

From the general net model of earlier formalisations we instantiate, that is, make more concrete, the tollway net model now described.

145. The net is now concretely modelled as a pair of sequences.
146. One sequence models the plaza hubs, their plaza-to-tollway link and the connected tollway hub.
147. The other sequence models the pairs of "twinned" tollway links.
148. From plaza hubs one can observe their hubs and the identifiers of these hubs.
149. The former sequence is of m such plaza "complexes" where $m \geq 2$; the latter sequence is of $m-1$ "twinned" links.
150. From a tollway net one can abstract a proper net.

```
type
145.  TWN = PC* × TL*
146.  PC = PH × L × H
147.  TL = L × L
value
146.  obs_H: PH → H, obs_HI: PH → HI
axiom
149.  ∀ (pcl,tll):TWN •
149.     2≤len pcl∧len pcl=len tll+1
value
150.  abs_HsLs: TWN → (Hs×Ls)
150.  abs_HsLs(pcl,tll) as (hs,ls)
150.     pre: wf_TWN(pcl,tll)
150.     post:
```

```
150.      hs = {h,h′|(h,_,h′):PC • (h,_,h′)∈ elems pcl}   150.          {l,l′|(l,l′):TL•(l,l′)∈ elems tll}
150.   ∧ ls = {l|(_,l,_):PC • (_,l,_)∈ elems pcl} ∪
```

[1] Model Well-formedness wrt. Instantiation:: Instantiation restricts general nets to tollway nets. Well-formedness deals with proper mereology: that observed identifier references are proper. The well-formedness of instantiation of the tollway system model can be defined as follows:

151. The i'plaza complex, (p_i, l_i, h_i), is instantiation-well-formed if
    [a] link l_i identifies hubs p_i and h_i, and
    [b] hub p_i and hub h_i both identifies link l_i; and if
152. the i'th pair of twinned links, tl_i, tl'_i,
    [a] has these links identify the tollway hubs of the i'th and i+1'st plaza complexes $((p_i, l_i, h_i)$ respectively $(p_{i+1}, l_{i+1}, h_{i_1}))$.

```
value
   Instantiation_wf_TWN: TWN → Bool
   Instantiation_wf_TWN(pcl,tll) ≡
151.    ∀ i:Nat • i ∈ inds pcl⇒
151.      let (pi,li,hi)=pcl(i) in
151a.       obs_LIs(li)={obs_HI(pi),obs_HI(hi)}
151b.     ∧ obs_LI(li)∈ obs_LIs(pi)∩ obs_LIs(hi)
152.    ∧ let (li′,li″) = tll(i) in
152.        i < len pcl ⇒
152.          let (pi′,li‴,hi′) = pcl(i+1) in
152a.           obs_HIs(li) = obs_HIs(li′)
152a.           = {obs_HI(hi),obs_HI(hi′)}
      end end end
```

7.2.3 Determination

By domain determination$_\delta$ we mean, as illustrated in this example, making part property values less in-determinate, i.e., more determinate.

The state sets contain only one set. Twinned tollway links allow traffic only in opposite directions. Plaza to tollway hubs allow traffic in both directions. tollway hubs allow traffic to flow freely from plaza to tollway links and from incoming tollway links to outgoing tollway links and tollway to plaza links. The determination-well-formedness of the tollway system model can be defined as follows[37]:

[1] Model Well-formedness wrt. Determination:: We need define well-formedness wrt. determination. Please study Fig. 4 on the next page.

[37] i ranges over the length of the sequences of twinned tollway links, that is, one less than the length of the sequences of plaza complexes. This "discrepancy" is reflected in out having to basically repeat formalisation of both Items 154a and 154b.

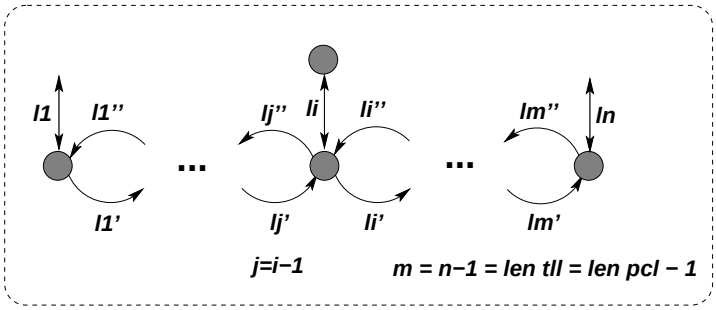


Figure 4: Hubs and Links

153. All hub and link state spaces contain just one hub, respectively link state.
154. The i'th plaza complex, pcl(i):(p_i, l_i, h_i) is determination-well-formed if
    [a] l_i is open for traffic in both directions and
    [b] p_i allows traffic from h_i to "revert"; and if
155. the i'th pair of twinned links (li', li'') (in the context of the i+1st plaza complex, pcl(i+1):$(p_{i+1}, l_{i+1}, h_{i+1})$) are determination-well-formed if
    [a] link l'_i is open only from h_i to h_{i+1} and
    [b] link l''_i is open only from h_{i+1} to h_i; and if
156. the jth tollway hub, h_j (for $1 \leq j \leq$ **len** pcl) is determination-well-formed if, depending on whether j is the first, or the last, or any "in-between" plaza complex positions,
    [a] [the first:] hub $i = 1$ allows traffic in from l_1 and l''_1, and onto l_1 and l'_1.
    [b] [the last:] hub $j = i + 1 =$ **len** pcl allows traffic in from $l_{\mathbf{len}\,\mathsf{tll}}$ and $l''_{\mathbf{len}\,\mathsf{tll}-1}$, and onto $l_{\mathbf{len}\,\mathsf{tll}}$ and $l'_{\mathbf{len}\,\mathsf{tll}-1}$.
    [c] [in-between:] hub $j = i$ allows traffic in from l_i, l''_i and l'_i and onto l_i, l'_{i-1} and l''_i.

```
value
154.  Determination_wf_TWN: TWN → Bool
154.  Determination_wf_TWN(pcl,tll) ≡
154.    ∀ i:Nat• i ∈ inds tll ⇒
154.      let (pi,li,hi) = pcl(i),
154.          (npi,nli,nhi) = pcl(i+1), in
154.          (li′,li″) = tll(i) in
153.        obs_HΩ(pi)={obs_HΣ(pi)}∧obs_HΩ(hi)={obs_HΣ(hi)}
153.    ∧ obs_LΩ(li)={obs_LΣ(li)}∧obs_LΩ(li′)={obs_LΣ(li′)}
153.    ∧ obs_LΩ(li″)={obs_LΣ(li″)}
154a.    ∧ obs_LΣ(li)
154a.          = {(obs_HI(pi),obs_HI(hi)),(obs_HI(hi),obs_HI(pi))}
154a.    ∧ obs_LΣ(nli)
154a.          = {(obs_HI(npi),obs_HI(nhi)),(obs_HI(nhi),obs_HI(npi))}
154b.    ∧ {(obs_LI(li),obs_LI(li))}⊆obs_HΣ(pi)
154b.    ∧ {(obs_LI(nli),obs_LI(nli))}⊆obs_HΣ(npi)
155a.    ∧ obs_LΣ(li′)={(obs_HI(hi),obs_HI(nhi))}
155b.    ∧ obs_LΣ(li″)={(obs_HI(nhi),obs_HI(hi))}
```

```
156.    ∧ case i+1 of
156a.      2 → obs_HΣ(h_1)=
156a.            {(obs_LΣ(l_1),obs_LΣ(l_1)), (obs_LΣ(l_1),obs_LΣ(l_1″)),
156a.            (obs_LΣ(l″_1),obs_LΣ(l_1)), (obs_LΣ(l″_1),obs_LΣ(l′_1))},
156b.      len pcl → obs_HΣ(h_i+1)=
156b.           {(obs_LΣ(l_len pcl),obs_LΣ(l_len pcl)),
156b.            (obs_LΣ(l_len pcl),obs_LΣ(l′_len tll)),
156b.            (obs_LΣ(l″_len tll),obs_LΣ(l_len pcl)),
156b.            (obs_LΣ(l″_len tll),obs_LΣ(l′_len tll))},
156c.      _ → obs_HΣ(h_i)=
156c.           {(obs_LΣ(l_i),obs_LΣ(l_i)), (obs_LΣ(l_i),obs_LΣ(l′_i)),
156c.            (obs_LΣ(l_i),obs_LΣ(l″_i−1)), (obs_LΣ(l″_i),obs_LΣ(l′_i)),
156c.            (obs_LΣ(l″_i),obs_LΣ(l′_i−1)), (obs_LΣ(l″_i),obs_LΣ(l′_i))}
154.    end end
```

7.2.4 Extension

By domain extension$_\delta$ we understand the *introduction of domain entities, actions, events and behaviours that were not feasible in the original domain, but for which, with computing and communication, there is the possibility of feasible implementations, and such that what is introduced become part of the emerging domain requirements prescription.*

Backgorund: The road traffic monitoring domain of Example 4, notably Sects. 2.8.6–2.8.7, (Items 65–71d Pages 28–29), illustrated the intangible abstraction of road traffic in the form of the recording of a discrete version of that traffic:[38]

```
46.  dT
45.  dRTF =  dT  m→  (VI  m→  VP)
```

by the road traffic system:

```
value
59.   trs() =
59a.     ‖ {veh(uid_V(v))(v)(vpm(uid_V(v)))|v:V•v ∈ vs}
59b.     ‖ mon(mi)(m)([ t0 ↦ vpm ])
```

We say that the road traffic, dRTF is intangible since the dRTF function, being a function, is an intangible. The domain extension is now making that "function" a tangible notion. There is no presumption, in defining the monitor behaviour, that there is indeed a mechanised behaviour, i.e., a computerised process that "implements" that monitor. Since one can speak of the monitor behaviour, one can, as well define it.

The Extension: We now "implement" a version of the above monitor behaviour. The proposed domain extension builds upon the monitor and the ability of vehicles to communicate their vehicle positions to the monitor, cf. Items 65a and 65a Page 28, Items 66a, 66(c)i and 66(c)iiA Page 29 and Item 71a Page 30. Instead of this "directness" we interpret links and

[38] In dRTF we change V into a reference to vehicles VI.

hubs of the tollway system as behaviours endowed with sensors. Vehicle behaviours now interact with link and hub behaviours communicating their positions which the link and hub behaviours communicate to a tollway system monitor. The domain extension then consists of the extension of links and hubs with sensors and the modelling of their vehicle interactions and their interaction with the tollway system monitor.

The Formalisation: We introduce

157. rather simple link and hub behaviours, and
158. an array of channels for the interaction of vehicle behaviours with link and hub behaviours.

And we modify

159. the vehicle and monitor behaviours and
160. the vehicle/monitor channel

the latter to now serve at the channel for link and hub interactions with the refined monitor behaviour.

```
value
150.   (hs,ls):(Hs×Ls) = abs_HsLs(twn)
22.    his:HI-set = {uid_H(h)|h:H•h ∈ hs}
21.    lis:LI-set = {uid_L(l)|l:L•l ∈ ls}
channel
158.   {vlh_ch[vi,si]|vi:VI,si:(LI|HI)•vi ∈ vis∧si ∈ lis ∪ his}:VP
160.   {lhm_ch[si,mi]|si:(LI|HI)•si ∈ lis ∪ his}:(VI×VP)
value
158.   link: li:LI → L → in { vlh_ch[vi,si]|si:LI•si ∈ lis }  Unit
158.   hub: hi:HI → H → in { vlh_ch[vi,si]|si:HI•si ∈ his }  Unit
157.   link(li)(l) ≡
157.      (...⌈⌉ [] {let (vi,vp) = vlh_ch[vi,li]? in lhm_ch[li,mi]!(vi,vp)|vi:VI•vi ∈ vis end});link(li)(l)
157.   hub(hi)(h) ≡
157.      (...⌈⌉ [] {let (vi,vp) = vlh_ch[vi,hi]? in lhm_ch[hi,mi]!(vi,vp)|vi:VI•vi ∈ vis end});hub(hi)(h)
59.    trs() =
59a.      ‖ {veh(uid_V(v))(v)(vpm(uid_V(v)))|v:V•v ∈ vs}
59b.      ‖ mon(mi)(m)([t0 ↦ vpm])
157.      ‖ {link(uid_L(l))(l)|l:L•l ∈ ls}
157.      ‖ {hub(uid_H(h))(h)|h:H•h ∈ hs}
```

The modifications to the vehicle behaviour is shown in Items 65a′, 65(b)ii′, 66a′, 66(c)i′, 66(c)iiA′ and 71a′ (Pages 75–76).

```
65.    veh(vi)(v)(vp:atH(fli,hi,tli)) ≡
65a′.        vlh_ch[vi,hi]!(vi,vp) ; veh(vi)(v)(vp)
65b.       ⌈⌉
65(b)i.      let {hi′,thi}=mereo_L(get_L(tli)(n)) in assert: hi′=hi
65(b)ii′.    vlh_ch[vi,tli]!(vi,onL(hi,tli,0,thi)) ;
```

```
65(b)iii.      veh(vi)(v)(onL(hi,tli,0,thi)) end
65c.        ⌈⌉
65d.          stop
```

```
64.   veh(vi)(v)(vp:onL(fhi,li,f,thi)) ≡
66a′.        vlh_ch[vi,li]!(vi,vp) ; veh(vi)(v)(vp)
66b.     ⌈⌉
66c.         if f + δ<1
66(c)i′.           then vlh_ch[vi,li]!(vi,onL(fhi,li,f+δ,thi)) ;
66(c)i.                 veh(vi)(v)(onL(fhi,li,f+δ,thi))
66(c)ii.           else let li′:LI•li′ ∈ mereo_H(get_H(thi)(n)) in
66(c)iiA′.              vlh_ch[vi,thi]!(vi,atH(li,thi,li′)) ;
66(c)iiB.               veh(vi)(v)(atH(li,thi,li′)) end end
67.         ⌈⌉
68.           stop
```

```
69.   mon(mi)(m)(rtf) ≡
70.         mon(mi)(own_mon_work(m))(rtf)
71.     ⌈⌉
71a′.     ⌷ { let ((vi,vp),t) = (lhm_ch[si,mi]?,clk_ch?) in
71b.          let rtf′ = rtf † [t ↦ rtf(max dom rtf) † [vi ↦ vp]] in
71c.          mon(mi)(m)(rtf′) end
71d.          end | si:(LI|HI) • si ∈ lis ∪ his}
```

The extension, in this example, does not really amount to much. We say that we have extended links and hubs with sensors. But we have not really modelled these sensors. We have modelled their intent, but not their extent. A more complete extension, which has to be done, but which is not shown in these notes, would now model these sensors as they rely on the unique vehicle identifier to be sensed. We shall, regrettably, omit this aspect of our presentation of the extension. There are so very many ways in which sensors and their object: the vehicles, can interact. Vehicles can be equipped with radio frequency identification tags, etcetera. Whichever sensor technology is chosen, it must be described. A description includes both it proper and its erroneous functioning. Such (IT equipment &c.) descriptions may be expressed in a number of steps: First, as here, a `RSL/CSP` [33, 5]. model. Then a "derived" description models temporal properties — using `Duration Calculus, DC` [69], or `Temporal Logic of Actions, TLA+` [40]. Finally a timed-automata [2, 51] model which "implements" the `DC` model.

7.3 Interface Requirements Prescription

A systematic reading of the domain requirements shall result in an identification of all shared parts and materials, actions, events and behaviours. An entity is said to be a shared entity$_\delta$ if it is present in some related forms, in both the domain and the machine.

Each such shared phenomenon shall then be individually dealt with: **part** and **materials 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 events are communicated between the environment of the machine and the machine. 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.

7.3.1 Shared Parts

The main shared parts of the main example of this section are the net, hence the hubs and the links. As domain parts they repeatedly undergo changes with respect to the values of a great number of attributes and otherwise possess attributes — most of which have not been mentioned so far: length, cadestral information, namings, wear and tear (where-ever applicable), last/next scheduled maintenance (where-ever applicable), state and state space, and many others.

We "split" our interface requirements development into two separate steps: the development of $d_{r.net}$ (the common domain requirements for the shared hubs and links), and the co-development of $d_{r.db:i/f}$ (the common domain requirements for the interface between $d_{r.net}$ and DB_{rel} — under the assumption of an available relational database system DB_{rel} 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 ($d_{r.toll}$, $d_{r.maint.}$) 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 modelling effort it must be secured that "standard" actions on nets, hubs and links can be supported by the chosen relational database system DB_{rel}.

[1] Data Initialisation:: As part of $d_{r.net}$ 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 link action.

[2] Data Refreshment:: As part of $d_{r.net}$ one must also prescribe data refreshment: an interactive user interface dialogue with a set of proper display screens one for updating net, hub or link attributes (names) and their types and, for example, two for the 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 concretise remove and insert link actions.

7.3.2 Shared Actions

The main shared actions are related to the entry of a vehicle into the tollway system and the exit of a vehicle from the tollway system.

[1] Interactive Action Execution:: As part of $d_{r.toll}$ we must therefore prescribe the varieties of successful and less successful sequences of interactions between vehicles (or their drivers) and the toll gate machines.

The prescription of the above necessitates determination of a number of external events, see below.

(Again, this is an area of embedded, real-time safety-critical system prescription.)

7.3.3 Shared Events

The main shared external events are related to the entry of a vehicle into the tollway system, the crossing of a vehicle through a tollway hub and the exit of a vehicle from the tollway system.

As part of $d_{r.toll}$ we must therefore prescribe the varieties of these events, the failure of all appropriate sensors and the failure of related controllers: gate opener and closer (with sensors and actuators), ticket "emitter" and "reader" (with sensors and actuators), etcetera.

The prescription of the above necessitates extensive fault analysis.

7.3.4 Shared Behaviours

The main shared behaviours are therefore related to the journey of a vehicle through the tollway system and the functioning of a toll gate machine during "its lifetime". Others can be thought of, but are omitted here.

In consequence of considering, for example, the journey of a vehicle behaviour, we may "add" some further, extended requirements: (a) requirements for a vehicle statistics "package"; (b) requirements for tracing supposedly "lost" vehicles; (c) requirements limiting tollway system access in case of traffic congestion; etcetera.

7.4 Machine Requirements

The machine requirements make hardly any concrete reference to the domain description; so we omit its treatment altogether.

7.5 Discussion of Requirements "Derivation"

We have indicated how the domain engineer and the requirements engineer can work together to "derive" significant fragments of a requirements prescription. This puts requirements engineering in a new light. Without a previously existing domain descriptions the requirements engineer has to do double work: both domain engineering and requirements engineering but without the principles of domain description, as laid down in these notes that job would not be so straightforward as we now suggest.

8 Conclusion

8.1 What Have We Achieved

We claim that there are four major contributions being reported upon: strongly hinting that *domain types and signatures form Galois connections*, the separation of domain engineering from requirements engineering, the separate treatment of domain science & engineering: as "free-standing" with respect, ultimately, to computer science, and endowed with quite a number of domain analysis principles and domain description principles; and the identification of a number of techniques for "deriving" significant fragments of requirements prescriptions from domain descriptions — where we consider this whole relation between domain engineering and requirements engineering to be novel. Yes, we really do consider the possibility of a systematic 'derivation' of significant fragments of requirements prescriptions from domain descriptions to cast a different light on requirements engineering.

What we have not shown in these notes is the concept of domain facets; this concept is dealt with in [10] — but more work has to be done to give a firm theoretical understanding of domain facets of domain intrinsics, domain support technology, domain scripts, domain rules and regulations, domain management and organisation, and human domainbehaviour. Facets will be illustrated in my keynote tomorrow!

8.2 General Remarks

Perhaps belaboring the point: one can pursue creating and studying domain descriptions without subsequently aiming at requirements development, let alone software design. That is, domain descriptions can be seen as "free-standing", of their "own right", useful in simply just understanding domains in which humans act. Just like it is deemed useful that we study "Mother Nature", the physical world around us, given before humans "arrived"; so we think that there should be concerted efforts to study and create domain models, for use in studying "our man-made domains of discourses"; possibly proving laws about these domains; teaching, from early on, in middle-school, the domains in which the middle-school students are to be surrounded by; etcetera

How far must one formalise such domain descriptions ? Well, enough, so that possible laws can be mathematically proved. Recall that domain descriptions usually will or must be developed by domain researchers — not necessarily domain engineers — in research centres, say universities, where one also studies physics. And, when we base requirements development on domain descriptions, as we indeed advocate, then the requirements engineers must understand the formal domain descriptions, that is, be able to perform formal domain projection, domain instantiation, domain determination, domain extension, etcetera. This is similar to the situation in classical engineering which rely on the sciences of physics, and where, for example, *Bernoulli's equations*, *Navier-Stokes equations*, *Maxwell's equations*, etcetera were developed by physicists and mathematicians, but are used, daily, by engineers: read and understood, massaged into further differential equations, etcetera, in order to calculate (predict, determine values), etc. Nobody would hire non-skilled labour for the engineering development of airplane designs unless that "labourer" was skilled in *Navier-Stokes equations*, or for the design of mobile telephony transmission towers unless that person was skilled in *Maxwell's equations*.

So we must expect a future, we predict, where a subset of the software engineering candidates from universities are highly skilled in the development of formal domain descriptions

formal requirements prescriptions in at least one domain, such as *transportation*, for example, air traffic, railway systems, road traffic and shipping; or *manufacturing*, *services* (health care, public administration, etc.), *financial industries*, or the like.

8.3 Acknowledgements

I thank the organisers of ASSC and especially Dr. Clive V. Boughton for having invited me to give this tutorial.

9 Bibliography

9.1 References

[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] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994. (Preliminary versions appeared in Proc. 17th ICALP, LNCS 443, 1990, and Real Time: Theory in Practice, LNCS 600, 1991).

[3] K. Åström and B. Wittenmark. *Adaptive Control*. Addison-Wesley Publishing Company, 1989.

[4] A. Badiou. *Being and Event*. Continuum, 2005. (Lêtre et l'événements, Edition du Seuil, 1988).

[5] D. Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. .

[6] 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.

[7] D. Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.

.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] D. Bjørner. *A Rôle for Mereology in Domain Science and Engineering*. Synthese Library (eds. Claudio Calosi and Pierluigi Graziani). Springer, Amsterdam, The Netherlands, May 2013.

[14] 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.

[15] D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.

[16] D. Bjørner and C. B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.

[17] W. D. Blizard. A Formal Theory of Objects, Space and Time. *The Journal of Symbolic Logic*, 55(1):74–89, March 1990.

[18] R. Carnap. *Der Logische Aufbau der Welt*. Weltkreis, Berlin, 1928.

[19] R. Casati and A. Varzi. *Parts and Places: the structures of spatial representation*. MIT Press, 1999.

[20] R. Casati and A. Varzi. Events. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2010 edition, 2010.

[21] B. L. Clarke. A Calculus of Individuals Based on 'Connection'. *Notre Dame J. Formal Logic*, 22(3):204–218, 1981.

[22] CoFI (The Common Framework Initiative). CASL *Reference Manual*, volume 2960 of *Lecture Notes in Computer Science (IFIP Series)*. Springer–Verlag, 2004.

[23] D. Davidson. *Essays on Actions and Events*. Oxford University Press, 1980.

[24] 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.

[25] C. Fox. *The Ontology of Language: Properties, Individuals and Discourse*. CSLI Publications, Center for the Study of Language and Information, Stanford University, California, ISA, 2000.

[26] 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.

[27] B. Ganter and R. Wille. *Formal Concept Analysis — Mathematical Foundations*. Springer-Verlag, January 1999. ISBN: 3540627715, 300 pages, Amazon price: US $ 44.95.

[28] 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.

[29] 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.

[30] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[31] C. 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).

[32] T. Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985.

[33] T. Hoare. Communicating Sequential Processes. Published electronically: `http://www.usingcsp.com/cspbook.pdf`, 2004. Second edition of [32]. See also `http://www.usingcsp.com/`.

[34] 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.

[35] ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992, 1996, 1999.

[36] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-10114-9.

[37] M. A. Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. ACM Press. Addison-Wesley, Reading, England, 1995.

[38] S. Karlin and H. M. Taylor. *An Introduction to Stochastic Modeling*. Academic Press, 1998. ISBN 0-12-684887-4.

[39] S. Kripke. *Naming and Necessity*. Harvard University Press, Cambridge, MA, USA, 1980. (See also: http://plato.stanford.edu/entries/rigid-designators).

[40] L. Lamport. *Specifying Systems*. Addison–Wesley, Boston, Mass., USA, 2002.

[41] H. Laycock. Object. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2011 edition, 2011.

[42] H. S. Leonard and N. Goodman. The Calculus of Individuals and its Uses. *Journal of Symbolic Logic*, 5:45–44, 1940.

[43] S. Leśniewksi. 0 Podstawack Matematyki (Foundations of Mathematics). *Prezeglad Filosoficzny*, 30-34, 1927-1931.

[44] E. Luschei. *The Logical Systems of Leśniewksi*. North Holland, Amsterdam, The Netherlands, 1962.

[45] J. M. E. McTaggart. The Unreality of Time. *Mind*, 18(68):457–84, October 1908. New Series. See also: [52].

[46] D. H. Mellor and A. Oliver, editors. *Properties*. Oxford Readings in Philosophy. Oxford Univ Press, May 1997. ISBN: 0198751761, 320 pages.

[47] K. Michels, F. Klawonn, R. Kruse, and A. Nürnberger. *Fuzzy Control: Fundamentals, Stability and Design of Fuzzy Controllers*. Springer, 19 October 2010.

[48] D. Miéville and D. Vernant. *Stanisław Leśniewksi aujourd'hui*. Grenoble, October 8-10, 1992.

[49] R. Milne. RSL Proof Rules. Research Report RAISE/CRI/DOC/5/V1, CRI A/S, 30 March 1990.

[50] R. Milnes. Semantic Foundations for RSL. Research Report RAISE/CRI/DOC/4/V1, CRI A/S, 30 March 1990.

[51] E.-R. Olderog and H. Dierks. *Real-Time Systems: Formal Specification and Automatic Verification*. Cambridge University Press, UK, 2008.

[52] R. L. Poidevin and M. MacBeath, editors. *The Philosophy of Time*. Oxford University Press, 1993.

[53] A. N. Prior. *Papers on Time and Tense*. Clarendon Press, Oxford, UK, 1968.

[54] W. Reisig. *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*. Leitfäden der Informatik. Vieweg+Teubner, 1st edition, 15 June 2010. 248 pages; ISBN 978-3-8348-1290-2.

[55] B. Russel. *"Preface," Our Knowledge of the External World*. G. Allen & Unwin, Ltd., London, 1952.

[56] B. Russell. On Denoting. *Mind*, 14:479–493, 1905.

[57] B. Russell. The Philosophy of Logical Atomism. *The Monist: An International Quarterly Journal of General Philosophical Inquiry,*, xxxviii–xxix:495–527, 32–63, 190–222, 345–380, 1918–1919.

[58] P. M. Simons. *Parts: A Study in Ontology*. Clarendon Press, 1987.

[59] B. Smith. Ontology and the Logistic Analysis of Reality. In G. Haefliger and P. M. Simons, editors, *Analytic Phenomenology*. Dordrecht/Boston/London: Kluwer, Padua, Italy, 1993.

[60] J. Srzednicki and Z. Stachniak, editors. *Leśniewksi's Lecture Notes in Logic*. Dordrecht, 1988.

[61] R. Turner. *Truth and Modality for Knowledge Representation*. Pitman, 1990.

[62] R. Turner. *Computational Linguistics and Formal Semantics*, chapter Properties, Propositions and Semantic Theory, pages 159–180. Studies in Natural Langhuage Processing, eds. M. Rosner and R. Johnson. Cambridge University Press, 1992.

[63] 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.

[64] A. van Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. In *5th IEEE International Symposium of Requirements Engineering*, volume RE'01, pages 249–263, Toronto, Canada, August 2001. IEEE CS Press.

[65] A. C. Varzi. *On the Boundary between Mereology and Topology*, pages 419–438. Hölder-Pichler-Tempsky, Vienna, 1994.

[66] A. C. Varzi. *Spatial Reasoning in a Holey*[39] *World*, volume 728 of *Lecture Notes in Artificial Intelligence*, pages 326–336. Springer, 1994.

[67] G. Wilson and S. Shpall. Action. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2012 edition, 2012.

[68] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.

[69] 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.

[39] holey: something full of holes

Appendices

- **A TripTych Ontology** 86–86
- **On A Theory of Container Stowage** 87–96
- **Indexes** 97–119
  - RSL Index 97
  - Formalisation Index 98
  - Definition Index 100
  - Example Index 101
  - Concept Index 103
  - Language, Method and Technology Index 118
  - Selected Author Index 118
- **An RSL Primer** 120–138

A A TripTych Ontology

161. domains — Sect. 3 pg 30
- [a] domain — Sect. 3.1 pg 30
- [b] domain phenomenon — Sect. 3.2 pg 30
- [c] domain entity — Sect. 3.3 pg 30
- [d] domain analysis — Sect. 3.4 pg 31
- [e] domain description — Sect. 3.5 pg 31
- [f] domain engineering — Sect. 3.6 pg 31
- [g] domain science — Sect. 3.7 pg 31
- [h] domain values and types — Sect. 3.8 pg 31
- [i] endurant entity — Sect. 3.9 pg 30
- [j] perdurant entity — Sect. 3.10 pg 30
- [k] discrete endurant — Sect. 3.11 pg 31
- [l] continuous endurant — Sect. 3.12 pg 31
- [m] discrete perdurant — Sect. 3.13 pg 31
- [n] continuous perdurant — Sect. 3.14 pg 31

162. discrete endurant domain entities — Sect. 4 pg 32
- [a] parts — Sect. 4.1 pg 33
  - i. abstract sorts — Sect. 4.1.6 pg 34
  - ii. atomic parts — Sect. 4.1.7 pg 34
  - iii. composite parts — Sect. 4.1.8 pg 34
  - iv. part observers — Sect. 4.1.9 pg 35
  - v. concrete types — Sect. 4.1.10 pg 35
- [b] part properties — Sect. 4.2 pg 36
  - i. unique identifiers — Sect. 4.2.1 pg 36
  - ii. mereology — Sect. 4.2.2 pg 37
  - iii. attributes — Sect. 4.2.3 pg 43
- [c] states — Sect. 4.3 pg 45

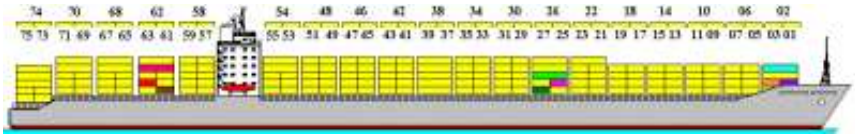
163. discrete perdurant domain entities — Sect. 5 pg 48
- [a] actions — Sect. 5.2 pg 49
  - i. action signatures — Sect. 5.2.3 pg 50
  - ii. action definitions — Sect. 5.2.4 pg 50
- [b] events — Sect. 5.3 pg 52
  - i. event signatures — Sect. 5.3.2 pg 53
  - ii. event predicate definitions — Sect. 5.3.3 pg 53
- [c] discrete behaviours — Sect. 5.4 pg 54
  - i. behaviour signatures — Sect. 5.4.4 pg 55
  - ii. behaviour definitions — Sect. 5.4.5 pg 55

164. continous entities — Sect. 6 pg 60
- [a] materials — Sect. 6.1 pg 60
  - i. materials-based domains — Sect. 6.1.1 pg 60
  - ii. part/material relations — Sect. 6.1.2 pg 61
  - iii. material observers — Sect. 6.1.3 pg 61
  - iv. material properties — Sect. 6.1.4 pg 62
  - v. laws of material flows and losses — Sect. 6.1.5 pg 64
- [b] continuous behaviours — Sect. 6.2 pg 65

B On A Theory of Container Stowage

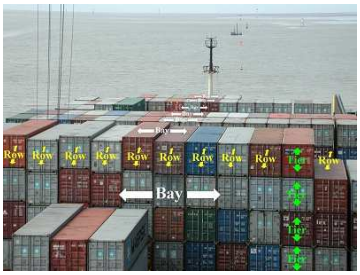
This section is under development. The idea of this section is not so much to present a container domain description, but rather to present fragments, "bits and pieces", of a theory of such a domain. The purpose of having a theory is to "draw" upon the 'bits and pieces' when expressing properties of endurants and definitions of actions, events and behaviours. Again: this section is very much in embryo.

B.1 Some Pictures

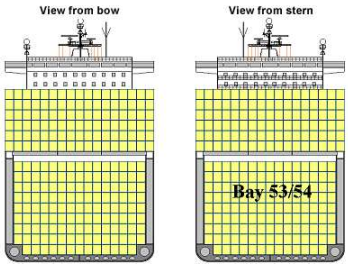


A container vessel with 'bay' numbering

Container vessels ply the seven seas and in-numerous other waters. They carry containers from port to port. The history of containers[40] goes back to the late 1930s. The first container vessels made their first transports in 1956. Malcolm P. McLean is credited to have invented the container. To prove the concept of container transport he founded the container line `Sea-Land Inc.` which was sold to `Maersk Lines` at the end of the 1990s.



Bay numbers.



Ship stowage cross section

Down along the vessel, horisontally, from front to aft, containers are grouped, in numbered bays.

Row and tier numbers

Bays are composed from rows, horisontally, across the vessel. Rows are composed from stacks, horisontally, along the vessel. And stacks are composed, vertically, from [tiers of] containers

[40] http://www.containerhandbuch.de/chb_e/stra/index.html?/chb_e/stra/stra_01_01_00.html

B.2 Parts

B.2.1 A Basis

165. From a container vessel (cv:CV) and from a container terminal port (ctp:CTP) one can observe their bays (bays:BAYS).

type
165. CV, CTP, BAYS
value
165. obs_BAYS: (CV|CTP) → BAYS

166. The bays, bs:BS, (of a container vessel or a container terminal port) are mereologically structured as an (BId) indexed set of individual bays (b:B).

type
166. BId, B
166. BS = BId $\overrightarrow{m}$ B
value
166. obs_BS: BAYS → BS (i.e., BId $\overrightarrow{m}$ B)

167. From a bay, b:B, one can observe its rows, rs:ROWS.

168. The rows, rs:RS, (of a bay) are mereologically structured as an (RId) indexed set of individual rows (r:R).

type
167. ROWS, RId, R
168. RS = RId $\overrightarrow{m}$ R
value
167. obs_ROWS: B → ROWS
168. obs_RS: ROWS → RS (i.e., RId $\overrightarrow{m}$ R)

169. From a row, r:R, one can observe its stacks, STACKS.

170. The stacks, ss:SS (of a row) are mereologically structured as an (SId) indexed set of individual stacks (s:S).

type
169. STACKS, SId, S
170. SS = SId $\overrightarrow{m}$ S
value
169. obs_STACKS: R → STACKS
170. obs_SS: STACKS → SS (i.e., SId $\overrightarrow{m}$ S)

171. A stack (s:S) is mereologically structured as a linear sequence of containers (c:C).

type
171. C
171. S = C*

The containers of the same stack index across stacks are called the tier at that index, cf. photo on Page 87..

172. A container is here considered a composite part
    [a] of the container box, k:K
    [b] and freight, f:F.
173. Freight is considered composite
    [a] and consists of zero, one or more colli (package, indivisible unit of freight),
    [b] each having a unique colli identifier (over all colli of the entire world !).
    [c] Container boxes likewise have unique container identifiers.

type
172. C, K, F, P
value
172a. obs_K: C → K
172b. obs_F: C → F
173a. obs_Ps: F → P-**set**
type
173b. PI
173c. CI
value
173b. uid_P: P → PI
173c. uid_C: C → CI

B.2.2 Mereological Constraints

174. For any bay of a vessel the index sets of its rows are identical.

175. For a bay of a vessel the index sets of its stacks are identical.

axiom
174. ∀ cv:CV •
174. ∀ b:B•b ∈ **rng** obs_BS(obs_BAYS(cv))⇒
174. **let** rws=obs_ROWS(b) **in**
174. ∀ r,r′:R•{r,r′}⊆**rng** obs_RS(b)⇒**dom** r=**dom** r′
175. ∧ **dom** obs_SS(r) = **dom** obs_SS(r′) **end**

B.2.3 Stack Indexes

176. A container stack (and a container) is designated by an index triple: a bay index, a row index and a stack index.

177. A container index triple is valid, for a vessel, if its indices are valid indices.

```
type
176.  StackId = BId×RId×SId
value
177.  valid_address: BS → StackId → Bool
177.  valid_address(bs)(bid,rid,sid) ≡
177.     bid ∈ dom bs
177.  ∧ rid ∈ dom (obs_RS(bs))(bid)
177.  ∧ sid ∈ dom (obs_SS((obs_RS(bs))(bid)))(rid)
```

The above can be defined in terms of the below.

```
type
    BayId = BId
    RowId = BId×RId
value
177.  valid_BayId: V → BayId → Bool
177.  valid_BayId(v)(bid) ≡  bid ∈ dom obs_BS(obs_BAYS(v))

177.  get_B: V → BayId ~→ B
177.  get_B(v)(bid) ≡ (get_B(bs))(bid) pre: valid_BId(v)(bid)

177.  get_B: BS → BayId ~→ B
177.  get_B(bs)(bid) ≡ (obs_BS(obs_BAYS(v)))(bid) pre: bid ∈ dom bs


177.  valid_RowId: V → RowId → Bool
177.  valid_RowId(v)(bid,rid) ≡ rid ∈ dom obs_RS(get_B(v)(bid))
177.     pre: valid_BayId(v)(bid)

177.  get_R: V → RowId ~→ R
177.  get_R(v)(bid,rid) ≡ get_R(obs_BS(v))(bid,rid) pre: valid_RowId(v)(bid,rid)

177.  get_R: BS → RowId ~→ R
177.  get_R(bs)(bid,rid) ≡ (obs_RS(get_RS(bs(bid))))(rid)
177.     pre: valid_RowId(v)(bid,rid)


177.  get_S: V → StackId ~→ S
177.  get_S(v)(bid,rid,sid) ≡ (obs_SS(get_R(get_B(v)(bid,rid))))(sid)
177.     pre: valid_address(v)(bid,rid,sid)
```

177. get_C: V → StackId $\stackrel{\sim}{\to}$ C
177. get_C(v)(stid) ≡ get_C(obs_BS(v))(stid) **pre**: get_S(v)(bid,rid,sid) ≠ ⟨⟩

177. get_C: BS → StackId $\stackrel{\sim}{\to}$ C
177. get_C(bs)(bid,rid,sid) ≡ **hd**(obs_SS(get_R((bs(bid))(rid))))(sid)
177. **pre**: get_S(bs)(bid,rid,sid) ≠ ⟨⟩

177. valid_addresses: V → StackId-**set**
177. valid_addresses(v) ≡ {adr|adr:StackId•valid_address(adr)(v)}

178. The predicate non_empty_designated_stack checks whether the designated stack is non-empty.

178. non_empty_designated_stack: V → StackId → **Bool**
178. non_empty_designated_stack(v)(bid,rid,sid) ≡ get_S(v)(bid,rid,sid) ≠ ⟨⟩

179. Two vessels have the same mereology if they have the same set of valid-addresses.

value
179. unchanged_mereology: BS × BS → **Bool**
179. unchanged_mereology(bs,bs′) ≡ valid_addresses(bs) = valid_addresses(bs′)

180. The designated stack, s′, of a vessel, v′ is popped with respect the "same designated" stack, s, of a vessel, v

   [a] if the ordered sequence of the containers of s′ are identical to the ordered sequence of containers of all but the first container of s.

180. popped_designated_stack: BS × BS → StackId → **Bool**
180. popped_designated_stack(bs,bs′)(stid) ≡
180a. **tl** get_S(v)(stid) = get_S(bs′)(stid)

181. For a given stack index, valid for two bays (bs, bs′) of two vessels or two container terminal ports, and say stid, these two bays enjoy the unchanged_non_designated_stacks(bs,bs′)(stid) property

   [a] if the stacks (of the two bays) not identified by stid are identical.

181. unchanged_non_designated_stacks: BS × BS → StackId → **Bool**
181. unchanged_non_designated_stacks(bs,bs′)(stid) ≡
181a. ∀ adr:StackId•adr ∈ valid_addresses(v)\{stid}⇒
181a. get_S(bs)(adr) = get_S(bs′)(adr)
181. **pre**: unchanged_mereology(bs,bs′)

B.2.4 Stowage Schemas

182. By a stowage schema of a vessel we understand a "table"

[a] which for every bay identifier of that vessel records a bay schema

[b] which for every row identifier of an identified bay records a row schema

[c] which for every stack identifier of an identified row records a stack schema

[d] which for every identified stack records its tier schema.

[e] A stack schema records for every tier index (which is a natural number) the type of container (contents) that may be stowed at that position.

[f] The tier indexes of a stack schema form a set of natural numbers from one to the maximum number in the index set.[41]

```
value
182.  obs_StoSchema: V → StoSchema
type
182a.  StoSchema = BId →m BaySchema
182b.  BaySchema = RId →m RowSchema
182c.  RowSchema = SId →m StaSchema
182d.  StaSchema = Nat →m C_Type
182e.  C_Type
axiom
182f.  ∀ stsc:StaSchema • dom stsc = {1..max dom stsc}
```

183. One can define a function which from an actual vessel "derives" its "current stowage schema".

```
183.  cur_sto_schema: V → StoSchema
183.  cur_sto_schema(v) ≡
183.     let bs = obs_BS(obs_BAYS(v)) in
183.     [ bid ↦ let rws = obs_RS(obs_ROWS(bs(bid))) in
183.           [ rid ↦ let ss = obs_SS(obs_STACKS(rws)(rid)) in
183.                 [ sid ↦ ⟨ analyse_container(ss(i))|i:Nat•i ∈ inds ss ⟩
183.                 | sid:SId•sid ∈ ss ] end
183.           | rid:RId•rid ∈ dom rws ] end
183.      | bid:BId•bid ∈ dom ds ] end

183.  analyse_container: C → C_Type
```

184. Given a stowage schema and a current stowage schema one can check the latter for conformance wrt. the former.

[41] That maximum number designates the maximum height of the stack at that stack position. For any actual stack the height is between zero and the maximum height, inclusive.

```
184. conformance: StoSchema × StoSchema → Bool
184. conformance(stosch,cur_stosch) ≡
184.    dom cur_stosch = dom stosch
184. ∧ ∀ bid:BId • bid ∈ dom stosch ⇒
184.       dom cur_stosch(bid) = dom stosch(bid)
184.    ∧ ∀ rid:RId • rid ∈ dom(stosch(bid))(rid) ⇒
184.          dom(cur_stosch(bid))(rid) = dom(stosch(bid))(rid)
184.       ∧ ∀ sid:SId • sid ∈ dom(cur_stosch(bid))(rid)
184.             ∀ i:Nat • i ∈ inds((cur_stosch(bid))(rid))(sid) ⇒
184.                conform((((cur_stosch(bid))(rid))(sid))(i),
184.                         (((stosch(bid))(rid))(sid))(i))

184. conform: C_Type × C_Type → Bool
```

185. From a vessel one can observe its mandated stowage schema.

186. The current stowage schema of a vessel must always conform to its mandated stowage schema.

```
value
185.  obs_StoSchema: V → StoSchema

186.  stowage_conformance: V → Bool
186.  stowage_conformance(v) ≡
186.     let mandated = obs_StoSchema(v),
186.         current = cur_sto_schema(v) in
186.     conformance(mandated,current) end
```

B.3 Actions

B.3.1 Remove Container from Vessel

106. The remove_Container_from_Vessel action applies to a vessel and a stack address and conditionally yields an updated vessel and a container.

    106a. We express the 'remove from vessel' function primarily by means of an auxiliary function remove_C_from_BS, remove_C_from_BS(obs_BS(v))(stid), and some further post-condition on the before and after vessel states (cf. Item 106d).

    106b. The remove_C_from_BS function yields a pair: an updated set of bays and a container.

    106c. When obs_erving the BayS from the updated vessel, v′, and pairing that with what is assumed to be a vessel, then one shall obtain the result of remove_C_from_BS(obs_BS(v))(stid).

    106d. Updating, by means of remove_C_from_BS(obs_BS(v))(stid), the bays of a vessel must leave all other properties of the vessel unchanged.

107. The pre-condition for remove_C_from_BS(bs)(stid) is

    107a. that stid is a valid_address in bs, and

    107b. that the stack in bs designated by stid is non_empty.

108. The post-condition for remove_C_from_BS(bs)(stid) wrt. the updated bays, bs′, is

    108a. that the yielded container, i.e., c, is obtained, get_C(bs)(stid), from the top of the non-empty, designated stack,

    108b. that the mereology of bs′ is unchanged, unchanged_mereology(bs,bs′). wrt. bs.

    108c. that the stack designated by stid in the "input" state, bs, is popped, popped_-designated_stack(bs,bs′)(stid), and

    108d. that all other stacks are unchanged in bs′ wrt. bs, unchanged_non_designated_-stacks(bs,bs′)(stid).

value
106. remove_C_from_V: V → StackId $\stackrel{\sim}{\rightarrow}$ (V×C)
106. remove_C_from_V(v)(stid) **as** (v′,c)
106c. (**obs**_Bs(**obs**_BS(v′),c)) = remove_C_from_BS(**obs**_Bs(**obs**_BS(v)))(stid)
106d. ∧ props(v)=props(v″)

106b. remove_C_from_BS: BS → StackId → (BS×C)
106a. remove_C_from_BS(bs)(stid) **as** (bs′,c)
107a. **pre**: valid_address(bs)(stid)
107b. ∧ non_empty_designated_stack(bs)(stid)
108a. **post**: c = get_C(bs)(stid)
108b. ∧ unchanged_mereology(bs,bs′)
108c. ∧ popped_designated_stack(bs,bs′)(stid)
108d. ∧ unchanged_non_designated_stacks(bs,bs′)(stid)

The props function was introduced in Sect. 4.2.4 on Page 45.

B.3.2 Remove Container from CTP

We define a remove action similar to that of Sect. B.3.1 on the previous page.

187. Instead of vessel bays we are now dealing with the bays of container terminal ports.

We omit the narrative — which is very much like that of narrative Items 106c and 106d.

value
187. remove_C_from_CTP: CTP → StackId $\stackrel{\sim}{\rightarrow}$ (CTP×C)
187. remove_C_from_CTP(ctp)(stid) **as** (ctp′,c)
106c. (obs_BS(ctp′),c) = remove_C_from_BS(obs_BS(ctp))(stid)
106d. ∧ props(ctp)=props(ctp″)

B.3.3 Stack Container on Vessel

188. Stacking a container at a vessel bay stack location

[a]

[b]

[c]

value
188. stack_C_on_vessel: BS → StackId $\stackrel{\sim}{\to}$ C $\stackrel{\sim}{\to}$ BS
188a. stack_C_on_vessel(bs)(stid)(c) **as** bs′
188a. **comment:** bs is bays of a v:V, i.e., bs = obs_BS(v)
188b. **pre**:
188c. **post**:

B.3.4 Stack Container in CTP

189.

190.

191.

192.

value
189. stack_C_in_CTP: CTP → StackId → C $\stackrel{\sim}{\to}$ CTP
190. stack_C_in_CTP(ctp)(stid)(c) **as** ctp′
191. **pre**:
192. **post**:

B.3.5 Transfer Container from Vessel to CTP

193.

194.

195.

196.

value
193. transfer_C_from_V_to_CTP: V→StackId$\stackrel{\sim}{\to}$CTP→StackId$\stackrel{\sim}{\to}$(V×CTP)
194. transfer_C_from_V_to_CTP(v)(v_stid)(ctp)(ctp_stid) ≡
195. **let** (c,v′) = remove_C_from_V(v)(v_stid) **in**
195. (v′,stack_C_in_CTP(ctp)(ctp_stid)(c)) **end**

B.3.6 Transfer Container from CTP to Vessel

197.

198.

199.

value

197. transfer_C_from_CTP_to_V: CTP→StackId$\stackrel{\sim}{\to}$V→StackId$\stackrel{\sim}{\to}$(CTP×V)
198. transfer_C_from_CTP_to_V(ctp)(ctp_stid)(v)(v_stid) ≡
199. **let** (c,ctp′) = remove_C_from_CTP(ctp)(ctp_stid) **in**
199. (ctp′,stack_C_in_CTP(ctp)(ctp_stid)(c)) **end**

C Indexes

C.1 RSL Index

Arithmetics
- ...,-2,-1,0,1,2,..., 122
- a_i*a_j , 125
- a_i+a_j , 125
- a_i/a_j , 125
- a_i=a_j , 124
- a_i≥a_j , 124
- a_i>a_j , 124
- a_i≤a_j , 124
- a_i<a_j , 124
- a_i≠a_j , 124
- a_i−a_j , 125

Cartesians
- (e_1,e_2,...,e_n) , 126

Chaos
- **chaos**, 128, 130

Clauses
- ... **elsif** ... , 135
- **case** b_e **of** pa_1 → c_1, ... pa_n → c_n **end** , 136
- **if** b_e **then** c_c **else** c_a **end** , 135

Combinators
- **let** a:A • P(a) **in** c **end** , 135
- **let** pa = e **in** c **end** , 134

Functions
- f(args) **as** result, 134
- **post** P(args,result), 134
- **pre** P(args), 134
- f(a), 132
- f(args) ≡ expr, 134

Imperative
- **case** b_e **of** pa_1 → c_1, ... pa_n → c_n **end** , 137
- **do** stmt **until** b*e* **end** , 137
- **for** e **in** $list_{expr}$ • P(b) **do** stm(e) **end** , 137
- **if** b_e **then** c_c **else** c_a **end** , 137
- **skip** , 137
- **variable** v:Type := expression , 137
- **while** b*e* **do** stm **end** , 137
- f(), 136
- stm_1;stm_2;...;stm_n; , 137
- v := expression , 137

Lists
- <Q(l(i))|i **in**<1..**len**l> •P(a)> , 126
- hAB, 126
- ℓ(i) , 129
- ⟨ei ..ej ⟩, 126
- ⟨$e_1, e_2, ..., e_n B$, 126
- **elems** ℓ , 129
- **hd** ℓ , 129
- **inds** ℓ , 129
- **len** ℓ , 129
- **tl** ℓ , 129

Logics
- b_i ∨ b_j , 124
- ∀ a:A • P(a) , 125
- ∃! a:A • P(a) , 125
- ∃ a:A • P(a) , 125
- ∼ b , 124
- **false**, 121, 124
- **true**, 121, 124
- a_i=a_j , 125
- a_i≥a_j , 125
- a_i>a_j , 125
- a_i≤a_j , 125
- a_i<a_j , 125
- a_i≠a_j , 125
- b_i ⇒ b_j , 124
- b_i ∧ b_j , 124

Maps
- [F(e)↦G(m(e))|e:E•e∈**dom** m∧P(e)] , 127
- [] , 126
- [u_1↦v_1,u_2↦v_2,...,u_n↦v_n] , 126
- m_i \ m_j , 131
- m_i ∘ m_j , 131
- m_i / m_j , 131
- **dom** m , 131
- **rng** m , 131
- m_i = mj , 131
- m_i ∪ m_j , 131
- m_i † m_j , 131
- m_i ≠ mj , 131

m(e) , 131

Processes

channel c:T , 137
channel {k[i]:T•i:KIdx} , 137
c ! e , 138
c ? , 138
k[i] ! e , 138
k[i] ? , 138
P ⨅ Q, 138
P ⨅ Q, 138
P: **Unit** → **in** c **out** k[i] **Unit** , 138
P ⨅ Q, 138
P ⨅ Q, 138
Q: i:KIdx → **out** c **in** k[i] **Unit**, 138

Sets

{Q(a)|a:A•a∈s∧P(a)} , 125
{} , 125
$\{e_1, e_2, ..., e_n\}$, 125
∩{s_1,s_2,...,s_n} , 127
∪{s_1,s_2,...,s_n} , 127
card s , 127
e∈s , 127
e∉s , 127
s_i=s_j , 127
s_i∩s_j , 127
s_i∪s_j , 127
s_i⊂s_j , 127
s_i⊆s_j , 127
s_i≠s_j , 127
s_i\s_j , 127

Types

(T_1×T_2×... ×T_n), 121
T^*, 121
T^ω, 121
T_1 × T_2 × ... × T_n, 121
Bool, 121
Char, 121
Int, 121
Nat, 121
Real, 121
Text, 121
Unit, 136, 138
mk_ id(s_1:T_1,s_2:T_2,...,s_n:T_n), 121
s_1:T_1 s_2:T_2 ... s_n:T_n, 121
T = Type_ Expr, 123
T_1 | T_2 | ... | T_1 | T_n , 121
T={| v:T′• P(v)|} , 123, 124
T==TE_1 | TE_2 | ... | TE_n , 123
Ti⇝Tj, 121
Ti→Tj, 121
T-**infset**, 121
T-**set**, 121

C.2 Formalisation Index

Concept
- Functions
  - conn_ Ns ι32, 21
  - derive_ RM ι27, 20
  - gen_ routes ι29, 21
  - is_ circular_ route ι30, 21
  - is_ conn_ N ι31, 21
  - spans_ HsLs ι32b, 22
  - vpr ι16, 18
  - vps ι14, 17
- Types
  - 𝕋𝕀 ι49, 26
  - 𝕋 ι48, 26
  - c𝕋 ι44, 25
  - cRTF ι43, 25
  - d𝕋 ι46, 26, 75
  - dRTF ι45, 26, 75
  - dRTF ι47, 26
  - R ι28, 20
  - RM ι26, 19
  - RM' ι25a, 19
  - Routes-**infset** ι29, 21
  - VPM ι15, 18
  - VP-**infset** ι14, 17
- Values
  - δ ι50, 26
  - lis:LI-**set** ι56, 27
  - t_0:T ι59e, 27
  - vpm:VPM ι59d, 27

Domain
- Δ ι1, 12

Endurant Extraction Functions
- xtr_ HIs ι22, 18

xtr_ LIs ι21, 18
Endurant Part
Attribute Observer
attr_ACC ι13, 17
attr_HΩ ι11b, 16
attr_HΣ ι11a, 16
attr_LΩ ι10b, 15
attr_LΣ ι10a, 15
attr_LEN ι10c, 15
attr_LOC ι10c, 15
attr_LOC ι11c, 16
attr_VEL ι13, 17
attr_VP ι13, 17
attr_atH ι13, 17
attr_onL ι13, 17
Attribute Type Axioms
HΩ ι11b, 16
HΣ ι11a, 16
LΩ ι10b, 15
LΣ ι10a, 15
Attribute Types
ACC ι12b, 17
atH ι12(a)ii, 16
HΩ ι11b, 16
HΣ ι11a, 16
LΩ ι10b, 15
LΣ ι10a, 15
LEN ι10c, 15
LOC ι10c, 15
onL ι12(a)i, 16
VEL ι12b, 17
VP ι12a, 16
Auxiliary Functions
get_ H ι26, 18
get_ L ι26a, 18
Mereology Axioms
H ι9b, 14
L ι8a, 14
Mereology Observers
mereo_H ι9a, 14
mereo_L ι8a, 14
Observers
obs_F ι1b, 12
obs_HS ι2a, 12
obs_ Hs ι5, 13
obs_LS ι2b, 12
obs_ Ls ι6, 13
obs_M ι1c, 12
obs_N ι1a, 12
obs_VS ι3, 12
obs_ Vs ι4a, 12
Types
F ι1b, 12
H ι5b, 13
HS ι2, 12
Hs ι5a, 13
L ι6b, 13
LS ι2, 12
Ls ι6a, 13
M ι1c, 12
N ι1a, 12
V ι4b, 12
VS ι3, 12
Vs ι4a, 12
Unique Identifier Observer
uid_H ι7a, 14
uid_L ι7b, 14
uid_V ι7c, 14
Unique Identifier Types
HI ι7a, 14
LI ι7b, 14
LV ι7c, 14
Values
ls:L-**set** ι56, 27
m:M ι58, 27
n:N ι56, 27
vs:V-**set** ι57, 27

Meta Functions Definitions:
attr_A ι92, 44
mereo_P ι78, 39
uid_P ι73, 37
upd_attr_A ι93, 44
upd_mereo_P ι87, 42

Perdurant Channnels
clk_ ch ι55, 26
vm_ ch[...] ι60, 28
Perdurant Functions
Actions
ins_ H ι37, 24
post_ ins_ H ι37c, 24
pre_ ins_ H ι37a, 24
Behaviours

clock ι54, 26
mon ι63, 28
mon ι69, 30, 77
own_ mon_ work ι70, 30
tra ι59, 27, 75, 76
tra ι61, 28
veh ι62, 28
veh ι64, 29, 76
veh ι65, 29, 76
Events
link_ dis ι38, 24
post_ link_ dis ι42, 25
pre_ link_ dis ι39, 24

Wellformedness
wf_ R ι28, 20
wf_ RM ι26, 19

C.3 Definition Index

abstract
type, 34
atomic
part, 35

behaviour, 55
signature, 56

channel, 55
communicating
behaviour, 55
composite
part, 35
concrete
type, 36
connector, 59
continuant, 31
continuous
behaviour, 61
model, 66
endurant, 31
perdurant, 32

data
initialisation, 78
refreshment, 78
determination, 71
discrete
action, 49, 50
endurant, 31
event, 49
perdurant, 32, 49
domain, 8, 30
analysis, 8, 31
description, 8, 31
determination, 73
engineering, 9, 31
entity, 31
extension, 75
instantiation, 72
phenomenon, 30
projection, 71
requirements, 71
science, 9, 31

endurant, 31
event, 53
definition, 54
signature, 53
event, 24
extension, 71
extensionality, 32
external
non-deterministic
behaviour, 55

fluid
dynamics, 66
function, 50
application, 50
invocation, 50

goal
requirements, 71

instantiation, 71
intentionality, 32
interface
requirements, 71
internal
non-deterministic

behaviour, 55

machine, 10, 70
requirements, 71
material, 31, 61
observer, 35
materials
based
domain, 61
mereology, 14, 38
method, 8
methodology, 8

part, 31, 33
attribute, 44
behaviour, 56
observer, 35
property
value, 34
perdurant
property, 36
prescriptive
domain
model, 67
projection, 71
property
value, 34
scale, 36

requirements, 70
domain, 71
goal, 71
interface, 71
machine, 71

same kind
class of parts, 33
sequential
behaviour, 55
shared
entity, 77
software, 10
sort, 34
state, 45
substance, 31

type, 32

value, 32

C.4 Example Index

2 A Container Line Analysis, 8
23 A Container Line Mereology, 41
50 A Pipeline System Behaviour, 67–69
3 A Transport Domain Description, 8–9
29 A Variety of Road Traffic Domain States, 45
33 Action Signatures: Nets and Vessels, 50–51
38 Atomic Part Behaviours, 57
10 Atomic Types, 35
11 Composite Types, 35
39 Compositional Behaviours, 57
27 Concrete Attribute Types, 44
14 Concrete Types, 36
35 Container Line: Remove Container, 51–52
8 Distinct Parts, 34
36 Events, 53
15 Has Composite Types, 36
12 Implementation of Observer Functions, 35
24 Insert Link, 42–43
18 Manifest and Conceptual Parts, 38
41 Materials, 61
20 Monitor and Vehicle Mereologies, 39
13 Observer Functions, 35
6 Part Properties, 33
7 Part Property Values, 34
9 Part Sorts, 34
5 Parts, 33
22 Pipeline Mereology, 39–40
30 Pipeline Units and Their Mereology, 46–47
44 Pipelines: Core Continuous Endurant, 62
49 Pipelines: Fluid Dynamics and Automatic Control, 67
48 Pipelines: Inter Unit Flow and Leak Law, 66
47 Pipelines: Intra Unit Flow and Leak Law, 65
31 Pipelines: Nets and Routes, 47–48
46 Pipelines: Parts and Material Properties,

63–64
45 Pipelines: Parts and Materials, 62–63
16 Property Value Scales, 36
21 Road Traffic System Mereology, 39
37 Road Transport System Event, 54
25 Road Transport System Part Attributes, 44
28 Setting Road Intersection Traffic Lights, 45
19 Shared Route Maps and Bus Time Tables, 39
43 Somehow Related Materials and Parts, 62
26 Static and Dynamic Attributes, 44
40 Syntax and Semantics of Mereology, 57–61
4 The Main Example, 11–30
32 Transport Net and Container Vessel Actions, 50
34 Transport Nets Actions, 51
17 Unique Identifier Functions, 37
42 Material Processing, 61
Material Processing (# 42), 61
1 Some Domains, 8

A Container Line Analysis (# 2), 8
A Container Line Mereology (# 23), 41
A Pipeline System Behaviour (# 50), 67–69
A Transport Domain Description (# 3), 8–9
A Variety of Road Traffic Domain States (# 29), 45
Action Signatures: Nets and Vessels (# 33), 50–51
Atomic Part Behaviours (# 38), 57
Atomic Types (# 10), 35

Composite Types (# 11), 35
Compositional Behaviours (# 39), 57
Concrete Attribute Types (# 27), 44
Concrete Types (# 14), 36
Container Line: Remove Container (# 35), 51–52

Distinct Parts (# 8), 34

Events (# 36), 53

Has Composite Types (# 15), 36

Implementation of Observer Functions (# 12), 35
Insert Link (# 24), 42–43

Manifest and Conceptual Parts (# 18), 38
Materials (# 41), 61
Monitor and Vehicle Mereologies (# 20), 39

Observer Functions (# 13), 35

Part Properties (# 6), 33
Part Property Values (# 7), 34
Part Sorts (# 9), 34
Parts (# 5), 33
Pipeline Mereology (# 22), 39–40
Pipeline Units and Their Mereology (# 30), 46–47
Pipelines: Core Continuous Endurant (# 44), 62
Pipelines: Fluid Dynamics and Automatic Control (# 49), 67
Pipelines: Inter Unit Flow and Leak Law (# 48), 66
Pipelines: Intra Unit Flow and Leak Law (# 47), 65
Pipelines: Nets and Routes (# 31), 47–48
Pipelines: Parts and Material Properties (# 46), 63–64
Pipelines: Parts and Materials (# 45), 62–63
Property Value Scales (# 16), 36

Road Traffic System Mereology (# 21), 39
Road Transport System Event (# 37), 54
Road Transport System Part Attributes (# 25), 44

Setting Road Intersection Traffic Lights (# 28), 45
Shared Route Maps and Bus Time Tables (# 19), 39
Somehow Related Materials and Parts (# 43), 62
Static and Dynamic Attributes (# 26), 44
Syntax and Semantics of Mereology (# 40), 57–61

The Main Example (# 4), 11–30
Transport Net and Container Vessel Actions (# 32), 50
Transport Nets Actions (# 34), 51

Unique Identifier Functions (# 17), 37

C.5 Concept Index

abstract, 10
  model, 38
  part, 38
abstraction, 31, 38, 39
  intangible, 75
account, 10
action, 8, 10, 11, 30, 49, 50, 52–55, 61
  discrete, 1
  domain, 50
  input, 55
  output, 55
  shared, 78
  sharing, 77
  signature, 50
adaptive
  control, 67
agency, 50
agent, 50
analyse, 1, 8
analyser
  domain, 34, 35, 49, 55
analysis, 1
  concept, 36, 37
    formal, 1, 34
  domain, 1, 8, 11, 37, 62
    principle, 79
  formal
    concept, 1, 34
  mathematical, 55
  principle
    domain, 79
and data acquisition
  control
    supervisory, 68
  supervisory
    control, 68
annotation
  definition
    function, 43
  function
    definition, 43
apply, 8
area
  bus time table
    metropolitan, 39
  metropolitan
    bus time table, 39
    road map, 39
  road map
    metropolitan, 39
argument, 50
  type, 53, 54
artefact, 8
atomic, 10, 32
  behaviour
    definition, 56, 57
    part, 56
  definition
    behaviour, 56, 57
  part, 11, 56
    behaviour, 56
attribute, 10, 11, 35, 37, 39, 44, 45, 67
  concrete
    type, 44
  dynamic, 44
    type, 44
  function
    signatures, 44
  map, 58
  material, 1, 32, 45
  name
    type, 44, 45
  observation function
    part, 44
  part, 1, 44, 45
    observation function, 44
    value, 45
  property
    value, 44
  relation
    value, 39
  signatures
    function, 44
  static
    type, 44
  type, 37, 44
    concrete, 44
    dynamic, 44

name, 44, 45
static, 44
value, 37, 39, 45
part, 45
property, 44
relation, 39
vehicle, 39
attributes
part, 32
automatic
control
theory, 69
theory
control, 69
axiom, 34, 44

behaviour, 1, 8, 10, 11, 30, 49, 50, 53, 55, 61
atomic
definition, 56, 57
part, 56
communicating
sequential, 55
composite
part, 56
continuous, 1, 54, 55
domain model, 66
core, 60
definition
atomic, 56, 57
function, 56
desirable
specification, 67
discrete, 49, 54
domain model, 66
domain model
continuous, 66
discrete, 66
dynamic, 63
function
definition, 56
narrative, 55
part, 56, 57
atomic, 56
composite, 56
sequential
communicating, 55
shared, 79
sharing, 77
specification
desirable, 67
behaviours
continuous, 61
bifurcation, 63
budget, 10
bus, 39
coordinating
traffic authority, 39
table
time, 39
time
table, 39
traffic authority
coordinating, 39
bus time table
area
metropolitan, 39
metropolitan
area, 39
business
engineering
process, 1
process
engineering, 1
re-engineering, 1
re-engineering
process, 1

calculus, 55
channel, 55
checking
model, 10
class
interesting, 50
communicate, 55
communicating
behaviour
sequential, 55
sequential
behaviour, 55
composite, 10, 32
behaviour
part, 56
part, 11, 32, 57
behaviour, 56

type, 35
value, 35
type
part, 35
value
part, 35
composite, 11
composition, 31
computing
science, 1
concept, 31, 36
analysis, 36, 37
formal, 1, 34
domain, 31, 34
formal
analysis, 1, 34
mathematical, 50
conceptual
connection, 42
part, 37
relation, 38
concrete, 10
attribute
type, 44
definition
part type, 42
type, 44
part
type, 36
part type
definition, 42
type
attribute, 44
definition, 44
part, 36
connection
conceptual, 42
spatial, 42
connector, 59
constant, 42
value, 44
construct, 8
container
description
domain, 88
domain
description, 88
continuous, 8, 30, 55
behaviour, 1, 54, 55
domain model, 66
behaviours, 61
core
endurant, 62
domain
endurant, 46
domain model
behaviour, 66
dynamic system
time, 67
endurant, 31, 61
core, 62
domain, 46
entities, 61
entities, 1, 61
endurant, 61
entity, 61
material, 11
perdurant, 32, 61
time
dynamic system, 67
contract
development, 10
control, 67
adaptive, 67
and data acquisition
supervisory, 68
automatic
theory, 69
fuzzy, 67
stochastic, 67
supervisory
and data acquisition, 68
theory
automatic, 69
coordinating
bus
traffic authority, 39
traffic authority
bus, 39
core
behaviour, 60
continuous
endurant, 62
endurant, 61

continuous, 62
material, 61

data
initialisation, 78
refreshment, 78
verification, 10
definition
annotation
function, 43
atomic
behaviour, 56, 57
behaviour
atomic, 56, 57
function, 56
concrete
part type, 42
type, 44
event, 54
formal
function, 43
function, 32, 43, 44, 52, 53
annotation, 43
behaviour, 56
formal, 43
narrative style, 43
predicate, 54
narrative style
function, 43
part type
concrete, 42
predicate
function, 54
type, 62
concrete, 44
definition set
function
type expression, 50
type expression
function, 50
derivation
requirements, 10
describer
domain, 33, 52, 54
description
container
domain, 88
development
domain, 1, 34
domain, 1, 8–11, 34, 38, 44, 70, 71, 79, 80
container, 88
development, 1, 34
principle, 79
formal, 8, 31
model
requirements, 67
narrative, 8, 31
principle
domain, 79
requirements
model, 67
descriptions
domain, 33
descriptive
model
natural science, 66, 67
natural science
model, 66, 67
design
phase
software, 9
software, 9–11, 80
phase, 9
desirable
behaviour
specification, 67
specification
behaviour, 67
determinate, 73
determination, 71
domain, 80
deterministic, 10
developer, 43
development
contract, 10
description
domain, 1, 34
documentation, 10
domain
description, 1, 34
manual
methodology, 10
methodology
manual, 10

requirements, 11, 80
software, 1
tool, 10
tool
software, 10
discrete, 8, 30
action, 1
behaviour, 49, 54, 55
domain model, 66
domain
endurant, 46
domain model
behaviour, 66
endurant, 1, 31, 38, 62
domain, 46
entities, 1
entity, 49
event, 1
part, 11
perdurant, 32, 49
documentation
development, 10
domain, 61, 77
action, 50
analyser, 34, 35, 49, 55
analysis, 1, 8, 11, 37, 62
principle, 79
concept, 31, 34
container
description, 88
continuous
endurant, 46
describer, 33, 52, 54
description, 1, 8–11, 34, 38, 44, 70, 71, 79, 80
container, 88
development, 1, 34
principle, 79
principles, 79
descriptions, 33
determination, 10, 80
development
description, 1, 34
discrete
endurant, 46
endurant, 46
continuous, 46
discrete, 46
engineer, 33, 79, 80
engineering, 1, 8–10, 70, 79
phase, 9
entity, 31, 32
extension, 10, 75, 80
facet, 80
human, 80
index, 13, 34, 35
initialisation, 10
instantiation, 80
intrinsics, 80
management and organisation, 80
manifest
phenomenon, 31
mereologies, 42
model, 34, 80
prescriptive, 67, 68
modelling, 64
phase
engineering, 9
phenomena, 1
phenomenon
manifest, 31
prescriptive
model, 67, 68
principle
analysis, 79
description, 79
projection, 10, 80
requirements, 10, 70, 71
researcher, 80
rules and regulations, 80
script, 80
specific
theory, 9
support technology, 80
theory, 9, 31
specific, 9
domain model
behaviour
continuous, 66
discrete, 66
continuous
behaviour, 66
discrete
behaviour, 66

dynamic, 42, 45
  attribute, 44
    type, 44
  behaviour, 63
  system, 63
  type
    attribute, 44
dynamic system
  continuous
    time, 67
  time
    continuous, 67

endurant, 8, 10, 30, 31, 36
  continuous, 31, 61
    core, 62
    domain, 46
    entities, 61
    entity, 61
  core, 61
    continuous, 62
  discrete, 1, 31, 38, 62
    domain, 46
  domain, 46
    continuous, 46
    discrete, 46
  entities, 1
    continuous, 61
  entity, 11
  manifest
    observable, 33
  observable
    manifest, 33
  properties, 33
  property, 36
engineer
  domain, 33, 79, 80
  requirements, 79, 80
engineering, 9, 31
  business
    process, 1
  domain, 1, 8–10, 70, 79
    phase, 9
  phase
    domain, 9
    requirements, 9
  process
    business, 1
  requirements, 1, 8, 9, 11, 70, 79
    phase, 9
  software, 1
entities, 1, 55
  continuous, 1, 61
    endurant, 61
  discrete, 1
  endurant, 1
    continuous, 61
  perdurant, 1
entity, 8, 11, 31, 52, 54
  continuous, 61
  discrete, 49
  domain, 31, 32
  instance, 32
  manifest, 31, 61
ergodicity, 63
event, 8, 10, 11, 24, 30, 49, 54, 55, 61
  definition, 54
  discrete, 1
  external
    shared, 79
  name, 53
  shared
    external, 79
  sharing, 77
expression
  type, 36
extension, 71
  domain, 75, 80
extensional
  feature, 32
  part
    relation, 38
  relation, 38
    part, 38
external
  event
    shared, 79
  shared
    event, 79

facet
  domain, 80
feature
  extensional, 32

intentional, 32
fleet, 39
flow, 63
fluid, 66
formal
analysis
concept, 1, 34
concept
analysis, 1, 34
definition
function, 43
description, 8, 31
function
definition, 43
languages
specification, 63
specification
languages, 63
test, 10
formal specification
language
model-oriented, 11
model-oriented
language, 11
function, 50, 61
annotation
definition, 43
application, 50
attribute
signatures, 44
behaviour
definition, 56
definition, 32, 43, 44, 52, 53
annotation, 43
behaviour, 56
formal, 43
narrative style, 43
predicate, 54
definition set
type expression, 50
formal
definition, 43
image set
type expression, 50
invocation, 50
meta, 35, 38, 39, 44, 45
name, 50
narrative style
definition, 43
non-deterministic, 50
partial, 50
predicate
definition, 54
signature, 54
property, 32, 45
signature, 36, 52, 56
predicate, 54
signatures
attribute, 44
space
total, 53
total
space, 53
type, 36
type expression
definition set, 50
image set, 50
fuzzy
control, 67

gas, 66
gaseous, 46
material, 61
goal, 70, 71
requirements, 71
golden rule
requirements, 70
granular
material, 61

hardware, 10, 70
human
domain, 80

ideal rule
of requirements, 70
identifier
part
unique, 1, 39, 42, 44, 45
type
unique, 39
type name
unique, 38
unique, 10, 11, 35, 37–39, 43
part, 1, 39, 42, 44, 45

type, 39
type name, 38
unit, 48
value, 38
vehicle, 77
unit
unique, 48
value
unique, 38
vehicle
unique, 77
identifiers
part
unique, 45
unique
part, 45
image set
function
type expression, 50
type expression
function, 50
in-determinate, 73
index, 35
domain, 13, 34, 35
initialisation
data, 78
initialise, 10
input
action, 55
installation
manual, 10
instance
of entity, 32
instantiation, 71
domain, 80
intangible, 75
abstraction, 75
phenomena, 38
intention, 50
intentional
feature, 32
part
properties, 37
relation, 38
properties, 36, 37
part, 37
property, 34
value, 34, 37
relation, 39
part, 38
value
property, 34, 37
interesting
class, 50
interface
requirements, 10, 70, 71
interval
time, 24, 53
intrinsics
domain, 80
IT
system, 10

language
formal specification
model-oriented, 11
model-oriented
formal specification, 11
languages
formal
specification, 63
specification
formal, 63
laws
material, 1
leak, 63
line
product, 10
liquid, 46, 66
material, 61

machine, 10, 70, 77
requirements, 10, 71
maintenance
manual, 10
management
plan, 10
management and organisation
domain, 80
manifest, 38
domain
phenomenon, 31
endurant
observable, 33

entity, 31, 61
observable
endurant, 33
part, 37, 38
phenomena, 38
phenomenon
domain, 31
manual
development
methodology, 10
installation, 10
maintenance, 10
methodology
development, 10
user, 10
map
attribute, 58
road, 39
material, 1, 8, 10, 11, 30, 31, 45, 61, 63
attribute, 1, 32, 45
continuous, 11
core, 61
gaseous, 61
granular, 61
laws, 1
liquid, 61
type, 1, 32, 45
mathematical
analysis, 55
concept, 50
model, 66
quantity, 32
mereologies
domain, 42
part, 45
mereology, 10, 11, 35, 37, 38, 42, 43
model, 42
part, 1, 42, 44, 45
part identifier
unique, 39
unique
part identifier, 39
meta
function, 35, 38, 39, 44, 45
properties, 33
methodology, 8
development
manual, 10
manual
development, 10
metropolitan
area
bus time table, 39
road map, 39
bus time table
area, 39
road map
area, 39
model
abstract, 38
checking, 10
description
requirements, 67
descriptive
natural science, 66, 67
domain, 34, 80
prescriptive, 67, 68
mathematical, 66
mereology, 42
natural science
descriptive, 66, 67
prescriptive
domain, 67, 68
requirements
description, 67
model-oriented
formal specification
language, 11
language
formal specification, 11
modelling, 1
domain, 64
requirements, 64
monitor, 39, 67
road
traffic, 39
traffic
road, 39

name
attribute
type, 44, 45
event, 53
function, 50

part
type, 35, 38, 39
perdurant, 36
sort, 34
type, 34, 36
attribute, 44, 45
part, 35, 38, 39
narrative
description, 8, 31
narrative style
definition
function, 43
function
definition, 43
natural
science, 66
natural science
descriptive
model, 66, 67
model
descriptive, 66, 67
net, 39
non-deterministic, 10
function, 50

observable
endurant
manifest, 33
manifest
endurant, 33
phenomenon, 11
observation function
attribute
part, 44
part
attribute, 44
ontology, 1
output
action, 55

part, 1, 8, 10, 11, 30–39, 42, 45, 55, 61, 62
abstract, 38
atomic, 56
behaviour, 56
attribute, 1, 44, 45
observation function, 44
value, 45
attributes, 32
behaviour, 56, 57
atomic, 56
composite, 56
composite, 11, 32, 57
behaviour, 56
type, 35
value, 35
conceptual, 37
concrete
type, 36
discrete, 11
extensional
relation, 38
identifier
unique, 1, 39, 42, 44, 45
identifiers
unique, 45
intentional
properties, 37
relation, 38
manifest, 37, 38
mereologies, 45
mereology, 1, 42, 44, 45
name
type, 35, 38, 39
observation function
attribute, 44
properties, 33, 34, 37, 38, 42, 45, 60
intentional, 37
property, 44
value, 34, 73
relation
extensional, 38
intentional, 38
shared, 78
sharing, 77
sort, 33
type, 1, 32–34, 36, 38, 39, 42, 44, 45
composite, 35
concrete, 36
name, 35, 38, 39
universe, 34
unique
identifier, 1, 39, 42, 44, 45
identifiers, 45
universe

type, 34
value
attribute, 45
composite, 35
property, 34, 73
part identifier
mereology
unique, 39
unique
mereology, 39
part type
concrete
definition, 42
definition
concrete, 42
partial
function, 50
perdurant, 8, 10, 30–32, 36, 49, 52, 54
continuous, 61
discrete, 49
entities, 1
name, 36
properties, 49
periodicity, 63
phase
design
software, 9
domain
engineering, 9
engineering
domain, 9
requirements, 9
requirements
engineering, 9
software
design, 9
phenomena
domain, 1
intangible, 38
manifest, 38
phenomenon
domain
manifest, 31
manifest
domain, 31
shared, 77
plan
management, 10
staffing, 10
point
time, 53
postcondition, 51
precondition, 51
predicate
definition
function, 54
function
definition, 54
signature, 54
signature, 53
function, 54
type, 32
prescription
requirements, 9, 10, 67, 70, 79, 80
prescriptions
requirements, 11
prescriptive
domain
model, 67, 68
model
domain, 67, 68
principle, 8
analysis
domain, 79
description
domain, 79
domain
analysis, 79
description, 79
problem, 8
process
business
engineering, 1
re-engineering, 1
engineering
business, 1
re-engineering
business, 1
product
line, 10
project, 10
projection, 71
domain, 80
proof, 10

properties, 33, 34, 49
  endurant, 33
  intentional, 36, 37
    part, 37
  meta, 33
  part, 33, 34, 37, 38, 42, 45, 60
    intentional, 37
  perdurant, 49
property, 8, 30–34, 36
  attribute
    value, 44
  endurant, 36
  function, 45
  intentional, 34
    value, 34, 37
  part, 44
    value, 34, 73
  proposition, 34
  propositions, 34
  scale
    value, 36
  value, 34, 36
    attribute, 44
    intentional, 34, 37
    part, 34, 73
    scale, 36
proposition, 33
  property, 34
propositions
  property, 34
props, 52, 95

quantities
  semantic, 57
  syntactic, 57
quantity
  mathematical, 32

range
  value, 36
re-engineering
  business
    process, 1
  process
    business, 1
refreshment
  data, 78

relation
  attribute
    value, 39
  conceptual, 38
  extensional
    part, 38
  intentional
    part, 38
  part
    extensional, 38
    intentional, 38
  spatial, 38
  value
    attribute, 39
requirements
  derivation, 10
  description
    model, 67
  development, 11, 80
  domain, 10, 70, 71
  engineer, 79, 80
  engineering, 1, 8, 9, 11, 70, 79
    phase, 9
  goal, 71
  golden rule, 70
  ideal rule, 70
  interface, 10, 70, 71
  machine, 10, 71
  model
    description, 67
  modelling, 64
  phase
    engineering, 9
  prescription, 9, 10, 67, 70, 79, 80
  prescriptions, 11
researcher
  domain, 80
result, 50
  type, 53, 54
road
  map, 39
  monitor
    traffic, 39
  traffic
    monitor, 39
road map
  area

metropolitan, 39
metropolitan
area, 39
route, 39
rule
of requirements, golden, 70
of requirements, ideal, 70
rules and regulations
domain, 80

scale
property
value, 36
value
property, 36
science
computing, 1
natural, 66
script
domain, 80
select, 8
semantic
quantities, 57
sequential
behaviour
communicating, 55
communicating
behaviour, 55
shared
action, 78
behaviour, 79
event
external, 79
external
event, 79
part, 78
phenomenon, 77
signature, 32, 49
action, 50
function, 36, 52, 56
predicate, 54
predicate, 53
function, 54
signatures
attribute
function, 44
function
attribute, 44
software, 10, 70
design, 9–11, 80
phase, 9
development, 1
tool, 10
engineering, 1
phase
design, 9
tool
development, 10
somehow related, 61, 62
sort, 34
name, 34
part, 33
space
function
total, 53
total
function, 53
spatial
connection, 42
relation, 38
specific
domain
theory, 9
theory
domain, 9
specification
behaviour
desirable, 67
desirable
behaviour, 67
formal
languages, 63
languages
formal, 63
stability, 63
staffing
plan, 10
state
type, 53
value, 50
static, 42
attribute
type, 44
type

attribute, 44
stochastic
control, 67
sub-part, 35
supervisory
and data acquisition
control, 68
control
and data acquisition, 68
support technology
domain, 80
synchronise, 55
syntactic
quantities, 57
system
dynamic, 63
IT, 10

table
bus
time, 39
time
bus, 39
tangible, 38, 75
techniques, 8, 10
test
formal, 10
theorem, 44
theory
automatic
control, 69
control
automatic, 69
domain
specific, 9
mereology, 53, 54
specific
domain, 9
time, 24, 53, 55
bus
table, 39
continuous
dynamic system, 67
dynamic system
continuous, 67
interval, 24, 53, 55
point, 53
table
bus, 39
tool
development
software, 10
software
development, 10
tools, 8
total
function, 50
space, 53
space
function, 53
traffic
monitor
road, 39
road
monitor, 39
traffic authority
bus
coordinating, 39
coordinating
bus, 39
`TripTych`, 9, 86, 87
type, 32–36
argument, 53, 54
attribute, 37, 44
concrete, 44
dynamic, 44
name, 44, 45
static, 44
composite
part, 35
concrete
attribute, 44
definition, 44
part, 36
definition, 62
concrete, 44
dynamic
attribute, 44
expression, 36
function, 36
identifier
unique, 39
material, 1, 32, 45
name, 34, 36

attribute, 44, 45
part, 35, 38, 39
part, 1, 32–34, 36, 38, 39, 42, 44, 45
composite, 35
concrete, 36
name, 35, 38, 39
universe, 34
predicate, 32
result, 53, 54
state, 53
static
attribute, 44
unique
identifier, 39
universe
part, 34
value, 36
type expression
definition set
function, 50
function
definition set, 50
image set, 50
image set
function, 50
type name
identifier
unique, 38
unique
identifier, 38
type P, 42

ubiquitous, 61
unique
identifier, 10, 11, 35, 37–39, 43
part, 1, 39, 42, 44, 45
type, 39
type name, 38
unit, 48
value, 38
vehicle, 77
identifiers
part, 45
mereology
part identifier, 39
part
identifier, 1, 39, 42, 44, 45
identifiers, 45
part identifier
mereology, 39
type
identifier, 39
type name
identifier, 38
unit
identifier, 48
value
identifier, 38
vehicle
identifier, 77
unit
identifier
unique, 48
unique
identifier, 48
universe
part
type, 34
type
part, 34
update, 42
user
manual, 10

value, 32, 34, 36, 50
attribute, 37, 39, 45
part, 45
property, 44
relation, 39
composite
part, 35
constant, 44
identifier
unique, 38
intentional
property, 34, 37
part
attribute, 45
composite, 35
property, 34, 73
property, 34, 36
attribute, 44
intentional, 34, 37
part, 34, 73

scale, 36
range, 36
relation
attribute, 39
scale
property, 36
state, 50
type, 36
unique
identifier, 38
variable, 44
variable, 42
value, 44
vehicle, 39
attribute, 39
identifier
unique, 77
unique
identifier, 77
verification
data, 10

yield, 50

C.6 Language, Method and Technology Index

`Alloy`, 2, 11, 63
`B`
Bourbaki, 2, 11, 63
`CASL`
Common Algebraic Specification Language, 63
`CSP`, 57
Communicating Sequential Processes, 55
`CafeOBJ`, 63
`Event B`, 2, 11, 63
`MSC`
Message Sequence Charts, 55
`Petri Net`, 55
`RAISE`
Rigorous Approach to Industrial Software Engineering, 2, 11, 63
`RSL`
`CSP`, 55
the `RAISE` Specification Language, 2, 11, 55, 63
`SCADA`, 68, 69
`Statechart`, 55
`TLA+`
Temporal Logic of Actions, 77
`VDM`
Vienna Development Method, 2, 11, 63
`Z`
Zermelo, 2, 11, 63

C.7 Selected Author Index

Jean-Raymond Abrial, 2, 11, 63
R. Alur, 77

A. Badiou, 32
Dines Bjørner, 2, 10, 11, 42, 55, 63, 77, 80
Wayne D. Blizard, 26

Rudolf Carnap, 32
R. Casati, 42
Bowman L. Clarke, 32

Jim Davies, 2, 11, 63
H. Dierks, 77
D.L. Dill, 77

John Fitzgerald, 2, 11, 63
Chris Fox, 36, 37

B. Ganter, 34, 36
Chris W. George, 2, 11, 55, 63
N. Goodman, 32

Michael Reichhardt Hansen, 77
David Harel, 55
C.A.R. Hoare, 55, 57, 77

Michael A. Jackson, 34
Daniel Jackson, 2, 11, 63
Cliff B. Jones, 2, 11, 63

Kokichi Futatsugi, 63
S. Kripke, 32

Leslie A. Lamport, 77
Peter Gorm Larsen, 2, 11, 63
H. Laycock, 32
H.S. Leonard, 32
S. Leśniewksi, 42
Staniław Leśniewksi, 32
E. Luschei, 42

J.M.E. McTaggart, 26
D.H. Mellor, 36
R.E. Milne, 34
Till Mossakowski, 63
Peter David Mosses, 63

Ernst-Rüdiger Olderog, 77
A. Oliver, 36

Søren Prehn, 2, 11, 55, 63
A.N. Prior, 26

Wolfgang Reisig, 55
B. Russel, 32, 37

Scpall, 32
B. Smith, 32

R. Turner, 37

Axel van Laamsverde, 71
Johan van Benthem, 26
A.C. Varzi, 42

R. Wille, 34, 36
Wilson, 32
Jim Woodcock, 2, 11, 63

Zhou ChaoChen, 77

D RSL: The Raise Specification Language

D.1 Type Expressions

Type expressions are expressions whose value are type, that is, possibly infinite sets of values (of "that" type).

D.1.1 Atomic Types

Atomic types have (atomic) values. That is, values which we consider to have no proper constituent (sub-)values, i.e., cannot, to us, be meaningfully "taken apart".

`RSL` has a number of *built-in* atomic types. There are the Booleans, integers, natural numbers, reals, characters, and texts.

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"

D.1.2 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.

[1] 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 $\xrightarrow{m}$ B
[14] A $\xrightarrow{\sim}$ B
[15] (A)
[16] A | B | ... | C
[17] mk_id(sel_a:A,...,sel_b:B)
[18] sel_a:A ... sel_b:B

The following are generic type expressions:

1. The Boolean type of truth values **false** and **true**.

2. The integer type on integers ..., –2, –1, 0, 1, 2, ...
3. The natural number type of positive integer values 0, 1, 2, ...
4. The real number type of real values, i.e., values whose numerals can be written as an integer, followed by a period (“.”), followed by a natural number (the fraction).
5. The character type of character values ″a″, ″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 $\underset{m}{\rightarrow}$ B), or (A*)-**set**, or (A-**set**)list, or (A|B) $\underset{m}{\rightarrow}$ (C|D|(E $\underset{m}{\rightarrow}$ 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.

[2] 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

D.2 Type Definitions

D.2.1 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′ • $\mathcal{P}$(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**

D.2.2 Subtypes

In RSL, each type represents a set of values. Such a set can be delimited by means of predicates. The set of values b which have type B and which satisfy the predicate $\mathcal{P}$, constitute the subtype A:

type
A = {| b:B • $\mathcal{P}$(b) |}

D.2.3 Sorts — Abstract Types

Types can be (abstract) sorts in which case their structure is not specified:

type
A, B, ..., C

D.3 The RSL Predicate Calculus

D.3.1 Propositional Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values (**true** or **false** [or **chaos**]). Then:

false, **true**
a, b, ..., c ∼a, a∧b, a∨b, a⇒b, a=b, a≠b

are propositional expressions having Boolean values. ∼, ∧, ∨, ⇒, = and ≠ are Boolean connectives (i.e., operators). They can be read as: *not*, *and*, *or*, *if then* (or *implies*), *equal* and *not equal*.

D.3.2 Simple Predicate Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values, let x, y, ..., z (or term expressions) designate non-Boolean values and let i, j, ..., k designate number values, then:

false, **true**
a, b, ..., c
∼a, a∧b, a∨b, a⇒b, a=b, a≠b
x=y, x≠y,
i<j, i≤j, i≥j, i≠j, i≥j, i>j

are simple predicate expressions.

D.3.3 Quantified Expressions

Let X, Y, ..., C be type names or type expressions, and let $\mathcal{P}(x)$, $\mathcal{Q}(y)$ and $\mathcal{R}(z)$ designate predicate expressions in which x, y and z are free. Then:

∀ x:X • $\mathcal{P}(x)$
∃ y:Y • $\mathcal{Q}(y)$
∃ ! z:Z • $\mathcal{R}(z)$

are quantified expressions — also being predicate expressions.

They are "read" as: For all x (values in type X) the predicate $\mathcal{P}(x)$ holds; there exists (at least) one y (value in type Y) such that the predicate $\mathcal{Q}(y)$ holds; and there exists a unique z (value in type Z) such that the predicate $\mathcal{R}(z)$ holds.

D.4 Concrete RSL Types: Values and Operations

D.4.1 Arithmetic

type
  Nat, **Int**, **Real**
value
  +,−,∗: **Nat**×**Nat**→**Nat** | **Int**×**Int**→**Int** | **Real**×**Real**→**Real**
  /: **Nat**×**Nat**$\stackrel{\sim}{\to}$**Nat** | **Int**×**Int**$\stackrel{\sim}{\to}$**Int** | **Real**×**Real**$\stackrel{\sim}{\to}$**Real**
  <,≤,=,≠,≥,> (**Nat**|**Int**|**Real**) → (**Nat**|**Int**|**Real**)

D.4.2 Set Expressions

[1] Set Enumerations: Let the below a's denote values of type A, then the below designate simple set enumerations:

{{}, {a}, {e_1,e_2,...,e_n}, ...} ∈ A-**set**
{{}, {a}, {e_1,e_2,...,e_n}, ..., {e_1,e_2,...}} ∈ A-**infset**

[2] Set Comprehension: The expression, last line below, to the right of the ≡, expresses set comprehension. The expression "builds" the set of values satisfying the given predicate. It is abstract in the sense that it does not do so by following a concrete algorithm.

type
  A, B
  P = A → **Bool**
  Q = A $\stackrel{\sim}{\to}$ B
value
  comprehend: A-**infset** × P × Q → B-**infset**
  comprehend(s,P,Q) ≡ { Q(a) | a:A • a ∈ s ∧ P(a)}

D.4.3 Cartesian Expressions

[1] 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)
```

D.4.4 List Expressions

[1] List Enumerations: Let a range over values of type A, then the below expressions are simple list enumerations:

```
   {⟨⟩, ⟨e⟩, ..., ⟨e1,e2,...,en⟩, ...} ∈ A*
   {⟨⟩, ⟨e⟩, ..., ⟨e1,e2,...,en⟩, ..., ⟨e1,e2,...,en,... ⟩, ...} ∈ A^ω

   ⟨ a_i .. a_j ⟩
```

The last line above assumes a_i and a_j to be integer-valued expressions. It then expresses the set of integers from the value of e_i to and including the value of e_j. If the latter is smaller than the former, then the list is empty.

[2] 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))⟩
```

D.4.5 Map Expressions

[1] Map Enumerations: Let (possibly indexed) u and v range over values of type $T1$ and $T2$, respectively, then the below expressions are simple map enumerations:

```
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
```

[2] 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) ]
```

D.4.6 Set Operations

[1] Set Operator Signatures:

```
value
   19 ∈: A × A-infset → Bool
   20 ∉: A × A-infset → Bool
   21 ∪: A-infset × A-infset → A-infset
   22 ∪: (A-infset)-infset → A-infset
   23 ∩: A-infset × A-infset → A-infset
   24 ∩: (A-infset)-infset → A-infset
   25 \: A-infset × A-infset → A-infset
   26 ⊂: A-infset × A-infset → Bool
   27 ⊆: A-infset × A-infset → Bool
   28 =: A-infset × A-infset → Bool
   29 ≠: A-infset × A-infset → Bool
   30 card: A-infset ~→ Nat
```

[2] Set Examples:

```
examples
   a ∈ {a,b,c}
   a ∉ {}, a ∉ {b,c}
   {a,b,c} ∪ {a,b,d,e} = {a,b,c,d,e}
   ∪{{a},{a,b},{a,d}} = {a,b,d}
   {a,b,c} ∩ {c,d,e} = {c}
   ∩{{a},{a,b},{a,d}} = {a}
   {a,b,c} \ {c,d} = {a,b}
   {a,b} ⊂ {a,b,c}
   {a,b,c} ⊆ {a,b,c}
   {a,b,c} = {a,b,c}
   {a,b,c} ≠ {a,b}
   card {} = 0, card {a,b,c} = 3
```

[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.

[4] Set Operator Definitions: The operations can be defined as follows ($\equiv$ is the definition symbol):

value

$s' \cup s'' \equiv \{ a \mid a{:}A \bullet a \in s' \vee a \in s'' \}$
$s' \cap s'' \equiv \{ a \mid a{:}A \bullet a \in s' \wedge a \in s'' \}$
$s' \setminus s'' \equiv \{ a \mid a{:}A \bullet a \in s' \wedge a \notin s'' \}$
$s' \subseteq s'' \equiv \forall\ a{:}A \bullet a \in s' \Rightarrow a \in s''$
$s' \subset s'' \equiv s' \subseteq s'' \wedge \exists\ a{:}A \bullet a \in s'' \wedge a \notin s'$
$s' = s'' \equiv \forall\ a{:}A \bullet a \in s' \equiv a \in s'' \equiv s \subseteq s' \wedge s' \subseteq s$
$s' \neq s'' \equiv s' \cap s'' \neq \{\}$
card s $\equiv$
  if s $= \{\}$ **then** 0 **else**
  let a:A $\bullet$ a $\in$ s **in** 1 + **card** (s $\setminus$ {a}) **end end**
  pre s /∗ is a finite set ∗/
card s $\equiv$ **chaos** /∗ tests for infinity of s ∗/

D.4.7 Cartesian Operations

type
A, B, C
g0: G0 = A × B × C
g1: G1 = (A × B × C)
g2: G2 = (A × B) × C
g3: G3 = A × (B × C)

value
va:A, vb:B, vc:C, vd:D
(va,vb,vc):G0,
(va,vb,vc):G1
((va,vb),vc):G2
(va3,(vb3,vc3)):G3

decomposition expressions
let (a1,b1,c1) = g0,
(a1′,b1′,c1′) = g1 **in** .. **end**
let ((a2,b2),c2) = g2 **in** .. **end**
let (a3,(b3,c3)) = g3 **in** .. **end**

D.4.8 List Operations

[1] List Operator Signatures:

value
hd: $A^\omega \xrightarrow{\sim} A$
tl: $A^\omega \xrightarrow{\sim} A^\omega$
len: $A^\omega \xrightarrow{\sim}$ **Nat**
inds: $A^\omega \rightarrow$ **Nat-infset**
elems: $A^\omega \rightarrow$ A-**infset**
.(.): $A^\omega \times$ **Nat** $\xrightarrow{\sim}$ A
⁀: A* ≠ Aω A×ω Aω A→ω BoBbol

[2] 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⟩

[3] Informal Explication:

- **hd**: Head gives the first element in a nonempty list.
- **tl**: Tail gives the remaining list of a nonempty list when Head is removed.
- **len**: Length gives the number of elements in a finite list.
- **inds**: Indices give the set of indices from 1 to the length of a nonempty list. For empty lists, this set is the empty set as well.
- **elems**: Elements gives the possibly infinite set of all distinct elements in a list.

- $\ell(i)$: Indexing with a natural number, i larger than 0, into a list ℓ having a number of elements larger than or equal to i, gives the ith element of the list.
- $\frown$: Concatenates two operand lists into one. The elements of the left operand list are followed by the elements of the right. The order with respect to each list is maintained.
- $=$: The equal operator expresses that the two operand lists are identical.
- $\neq$: The nonequal operator expresses that the two operand lists are *not* identical.

The operations can also be defined as follows:

[4] 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″)
```

D.4.9 Map Operations

[1] Map Operator Signatures and Map Operation Examples:

value

m(a): M → A $\stackrel{\sim}{\to}$ 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 $\stackrel{m}{\to}$ B) × (B $\stackrel{m}{\to}$ C) → (A $\stackrel{m}{\to}$ C) [composition]
  [a↦b,a′↦b′] ° [b↦c,b′↦c′,b″↦c″] = [a↦c,a′↦c′]

[2] Map Operation Explication:

- $m(a)$: Application gives the element that a maps to in the map m.
- **dom**: Domain/Definition Set gives the set of values which *maps to* in a map.
- **rng**: Range/Image Set gives the set of values which *are mapped to* in a map.
- †: Override/Extend. When applied to two operand maps, it gives the map which is like an override of the left operand map by all or some "pairings" of the right operand map.
- ∪: Merge. When applied to two operand maps, it gives a merge of these maps.
- \: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements that are not in the right operand set.
- /: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements of the right operand set.

- =: The equal operator expresses that the two operand maps are identical.
- ≠: The nonequal operator expresses that the two operand maps are *not* identical.
- °: Composition. When applied to two operand maps, it gives the map from definition set elements of the left operand map, m_1, to the range elements of the right operand map, m_2, such that if a is in the definition set of m_1 and maps into b, and if b is in the definition set of m_2 and maps into c, then a, in the composition, maps into c.

[3] 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
```

D.5 λ-Calculus + Functions

D.5.1 The λ-Calculus Syntax

```
type /∗ A BNF Syntax: ∗/
    ⟨L⟩ ::= ⟨V⟩ | ⟨F⟩ | ⟨A⟩ | ( ⟨A⟩ )
    ⟨V⟩ ::= /∗ variables, i.e. identifiers ∗/
    ⟨F⟩ ::= λ⟨V⟩ • ⟨L⟩
    ⟨A⟩ ::= ( ⟨L⟩⟨L⟩ )
value /∗ Examples ∗/
    ⟨L⟩: e, f, a, ...
    ⟨V⟩: x, ...
    ⟨F⟩: λ x • e, ...
    ⟨A⟩: f a, (f a), f(a), (f)(a), ...
```

D.5.2 Free and Bound Variables

Let x, y be variable names and e, f be λ-expressions.

- $\langle$V$\rangle$: Variable x is free in x.
- $\langle$F$\rangle$: x is free in $\lambda y \bullet e$ if $x \neq y$ and x is free in e.
- $\langle$A$\rangle$: x is free in $f(e)$ if it is free in either f or e (i.e., also in both).

D.5.3 Substitution

In RSL, the following rules for substitution apply:

- **subst**([N/x]x) ≡ N;
- **subst**([N/x]a) ≡ a,

  for all variables a≠ x;
- **subst**([N/x](P Q)) ≡ (**subst**([N/x]P) **subst**([N/x]Q));
- **subst**([N/x]($\lambda x \bullet P$)) ≡ λ y•P;
- **subst**([N/x](λ y•P)) ≡ $\lambda y \bullet$ **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)).

D.5.4 α-Renaming and β-Reduction

- α-renaming: λx•M

  If x, y are distinct variables then replacing x by y in λx•M results in λy•**subst**([y/x]M). We can rename the formal parameter of a λ-function expression provided that no free variables of its body M thereby become bound.
- β-reduction: (λx•M)(N)

  All free occurrences of x in M are replaced by the expression N provided that no free variables of N thereby become bound in the result. (λx•M)(N) ≡ **subst**([N/x]M)

D.5.5 Function Signatures

For sorts we may want to postulate some functions:

type
  A, B, C
value
  obs_B: A → B,
  obs_C: A → C,
  gen_A: B×C → A

D.5.6 Function Definitions

Functions can be defined explicitly:

value
f: Arguments → Result
f(args) ≡ DValueExpr

g: Arguments $\stackrel{\sim}{\rightarrow}$ 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 $\stackrel{\sim}{\rightarrow}$ Result
g(args) **as** result
pre P2(args)
post P3(args,result)

The symbol $\stackrel{\sim}{\rightarrow}$ indicates that the function is partial and thus not defined for all arguments. Partial functions should be assisted by preconditions stating the criteria for arguments to be meaningful to the function.

D.6 Other Applicative Expressions

D.6.1 Simple let Expressions

Simple (i.e., nonrecursive) **let** expressions:

let a = $\mathcal{E}_d$ **in** $\mathcal{E}_b$(a) **end**

is an "expanded" form of:

$(\lambda a.\mathcal{E}_b(a))(\mathcal{E}_d)$

D.6.2 Recursive let Expressions

Recursive **let** expressions are written as:

let f = λa:A • E(f) **in** B(f,a) **end**

is "the same" as:

let f = **Y**F **in** B(f,a) **end**

where:

F ≡ λg•λa•(E(g)) and YF = F(YF)

D.6.3 Predicative let Expressions

Predicative **let** expressions:

let a:A • $\mathcal{P}$(a) **in** $\mathcal{B}$(a) **end**

express the selection of a value a of type A which satisfies a predicate $\mathcal{P}$(a) for evaluation in the body $\mathcal{B}$(a).

D.6.4 Pattern and "Wild Card" let Expressions

Patterns and *wild cards* can be used:

let {a} ∪ s = set **in** ... **end**
let {a,_} ∪ s = set **in** ... **end**

let (a,b,...,c) = cart **in** ... **end**
let (a,_,...,c) = cart **in** ... **end**

let ⟨a⟩⌢ℓ = list **in** ... **end**
let ⟨a,_,b⟩⌢ℓ = list **in** ... **end**

let [a↦b] ∪ m = map **in** ... **end**
let [a↦b,_] ∪ m = map **in** ... **end**

D.6.5 Conditionals

Various kinds of conditional expressions are offered by RSL:

```
if b_expr then c_expr else a_expr
end

if b_expr then c_expr end ≡ /* same as: */
   if b_expr then c_expr else skip end

if b_expr_1 then c_expr_1
elsif b_expr_2 then c_expr_2
elsif b_expr_3 then c_expr_3
...
```

```
elsif b_expr_n then c_expr_n end

case expr of
   choice_pattern_1 → expr_1,
   choice_pattern_2 → expr_2,
   ...
   choice_pattern_n_or_wild_card → expr_n
end
```

D.6.6 Operator/Operand Expressions

```
⟨Expr⟩ ::=
          ⟨Prefix_Op⟩ ⟨Expr⟩
        | ⟨Expr⟩ ⟨Infix_Op⟩ ⟨Expr⟩
        | ⟨Expr⟩ ⟨Suffix_Op⟩
        | ...
⟨Prefix_Op⟩ ::=
        − | ∼ | ∪ | ∩ | card | len | inds | elems | hd | tl | dom | rng
⟨Infix_Op⟩ ::=
          = | ≠ | ≡ | + | − | ∗ | ↑ | / | < | ≤ | ≥ | > | ∧ | ∨ | ⇒
        | ∈ | ∉ | ∪ | ∩ | \ | ⊂ | ⊆ | ⊇ | ⊃ | ⌢ | † | °
⟨Suffix_Op⟩ ::= !
```

D.7 Imperative Constructs

D.7.1 Statements and State Changes

Often, following the RAISE method, software development starts with highly abstract-applicative constructs which, through stages of refinements, are turned into concrete and imperative constructs. Imperative constructs are thus inevitable in RSL.

```
   Unit
value
   stmt: Unit → Unit
   stmt()
```

- Statements accept no arguments.
- Statement execution changes the state (of declared variables).
- **Unit** → **Unit** designates a function from states to states.
- Statements, stmt, denote state-to-state changing functions.
- Writing () as "only" arguments to a function "means" that () is an argument of type **Unit**.

D.7.2 Variables and Assignment

0. **variable** v:Type := expression
1. v := expr

D.7.3 Statement Sequences and skip

Sequencing is expressed using the ';' operator. **skip** is the empty statement having no value or side-effect.

2. **skip**
3. stm_1;stm_2;...;stm_n

D.7.4 Imperative Conditionals

4. **if** expr **then** stm_c **else** stm_a **end**
5. **case** e **of**: p_1→S_1(p_1),...,p_n→S_n(p_n) **end**

D.7.5 Iterative Conditionals

6. **while** expr **do** stm **end**
7. **do** stmt **until** expr **end**

D.7.6 Iterative Sequencing

8. **for** e **in** list_expr • P(b) **do** S(b) **end**

D.8 Process Constructs

D.8.1 Process Channels

Let A and B stand for two types of (channel) messages and i:KIdx for channel array indexes, then:

channel c:A
channel { k[i]:B • i: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).

D.8.2 Process Composition

Let P and Q stand for names of process functions, i.e., of functions which express willingness to engage in input and/or output events, thereby communicating over declared channels. Let P() and Q stand for process expressions, then:

P ‖ Q Parallel composition
P [] Q Nondeterministic external choice (either/or)
P ⨅ Q Nondeterministic internal choice (either/or)
P ‡ Q Interlock parallel composition

express the parallel (‖) of two processes, or the nondeterministic choice between two processes: either external ([]) or internal (⨅). The interlock (‡) composition expresses that the two processes are forced to communicate only with one another, until one of them terminates.

D.8.3 Input/Output Events

Let c, k[i] and e designate channels of type A and B, then:

c ?, k[i] ? Input
c ! e, k[i] ! e Output

expresses the willingness of a process to engage in an event that "reads" an input, respectively "writes" an output.

D.8.4 Process Definitions

The below signatures are just examples. They emphasise that process functions must somehow express, in their signature, via which channels they wish to engage in input and output events.

```
value
   P: Unit → in c out k[i]
   Unit
   Q: i:KIdx →  out c in k[i] Unit

   P() ≡ ... c ? ... k[i] ! e ...
   Q(i) ≡ ... k[i] ? ... c ! e ...
```

The process function definitions (i.e., their bodies) express possible events.

D.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.[42]

[42] For schemes, classes and objects we refer to [6, Chap. 10]