

Dines Bjørner

Domain Science & Engineering

A Foundation for Software Development

DRAFT

Monday, December 5, 2016: 07:21 am

A Compendium of Six Papers

Fredsvej 11, DK-2840 Holte, Denmark

Technical University of Denmark

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

This compendium was put together 2nd half of November 2016
It was first released Monday, December 5, 2016: 07:21 am

Kari; Charlotte and Nikolaj; Camilla, Marianne, Katrine, Caroline and Jakob

the full meaning of life

Preface

The Triptych Dogma

In order to *specify* **software**,
we must understand its requirements.

In order to *prescribe* **requirements**,
we must understand the **domain**,
so we must *describe* it.

General

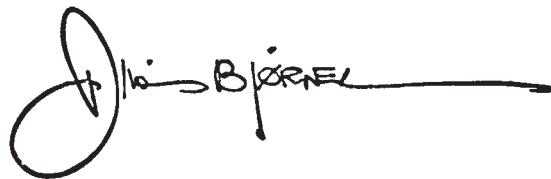
The thesis of this collection of papers is that domain engineering is a viable, yes, we would claim, necessary initial phase of software development. I mean this rather seriously: How can one think of implementing **software**, preferably satisfying some **requirements**, without demonstrating that one understands the **domain**? So in this collection of papers I shall explain what domain engineering is, some of the science that goes with it, and how one can 'derive' requirements prescriptions (for computing systems) from domain descriptions. But there is an altogether different reason, also, for presenting these papers: Software houses may not take up the challenge to develop software that satisfies customers expectations, that is, reflects the domain such as these customers know it, and software that is correct with respect to requirements, with proofs of correctness often having to refer to the domain. But computing scientists are shown, in these papers, that domain science and engineering is a field full of interesting problems to be researched. We consider domain descriptions, requirements prescriptions and software design specifications to be mathematical quantities.

A Brief Guide

I have collected six papers in this document:

- Chapter 1: [49, Manifest Domains: Analysis & Description] Pages 3–52
- Chapter 2: [53, Domain Facets: Analysis & Description] Pages 53–79
- Chapter 3: [52, Formal Models of Processes and Prompts] Pages 81–112
- Chapter 4: [50, To Every Manifest Domain Mereology a CSP Expression] Pages 113–132
- Chapter 5: [54, From Domain Descriptions to Requirements Prescriptions] Pages 135–176
- Chapter 6: [48, Domains: Their Simulation, Monitoring and Control] Pages 179–187

We urge the reader to study the **Contents** listing and from there to learn that there is a **Bibliography** common to all six chapters, two example appendices, **An RSL Primer**, a set of indexes into definitions, concepts, examples, analysis and description prompts, and an index of **RSL Symbols**.



Dines Bjørner. December 5, 2016: 07:21 am
Fredsevej 11, DK-2840 Holte, Denmark

Contents

Preface	v
The Triptych Dogma	v
General	v
A Brief Guide	v
<hr/>	
Part I Domains	
<hr/>	
1 Manifest Domains: Analysis & Description	3
Summary	3
1.1 Introduction	3
1.1.1 The TripTych Approach to Software Engineering	4
1.1.2 Method and Methodology	4
Method	4
Discussion	4
Methodology	5
1.1.3 Computer and Computing Science	5
1.1.4 What Is a Manifest Domain?	5
1.1.5 What Is a Domain Description?	6
1.1.6 Towards a Methodology of Manifest Domain Analysis & Description	6
Practicalities of Domain Analysis & Description.	6
The Four Domain Analysis & Description “Players”.	6
An Interactive Domain Analysis & Description Dialogue.	7
Prompts	7
A Domain Analysis & Description Language.	7
The Domain Description Language.	7
Domain Descriptions: Narration & Formalisation	7
1.1.7 One Domain – Many Models?	8
1.1.8 Formal Concept Analysis	8
A Formalisation	8
Types Are Formal Concepts	9
Practicalities	9
Formal Concepts: A Wider Implication	9
1.1.9 Structure of Chapter	10
1.2 Entities	10
1.2.1 General	10
a: Analysis Prompt: is-entity	10
1.2.2 Endurants and Perdurants	10
b: Analysis Prompt: is-endurant	10
c: Analysis Prompt: is-perdurant	11

1.2.3	Discrete and Continuous Endurants	11
	d: Analysis Prompt: is discrete	11
	e: Analysis Prompt: is continuous	11
1.2.4	An Upper Ontology Diagram of Domains	11
1.3	Endurants	11
1.3.1	Parts, Components and Materials	12
	General	12
	Part, Component and Material Analysis Prompts	13
	f: Analysis Prompt: is part	13
	g: Analysis Prompt: is component	13
	h: Analysis Prompt: is material	13
	Atomic and Composite Parts	13
	i: Analysis Prompt: is-atomic	14
	j: Analysis Prompt: is-composite	14
	On Observing Part Sorts and Types	14
	On Discovering Part Sorts	14
	k: Analysis Prompt: observe-parts	14
	Part Sort Observer Functions	15
	1: Description Prompt: 1.observe-part-sorts	15
	On Discovering Concrete Part Types	16
	l: Analysis Prompt: has-concrete-type	16
	2: Description Prompt: observe-part-type	16
	Forms of Part Types	17
	Part Sort and Type Derivation Chains	17
	No Recursive Derivations	17
	Names of Part Sorts and Types	17
	More On Part Sorts and Types	18
	External and Internal Qualities of Parts	19
	Three Categories of Internal Qualities	19
1.3.2	Unique Part Identifiers	19
	3: Description Prompt: observe-unique-identifier	19
1.3.3	Mereology	20
	Part Relations	20
	Part Mereology: Types and Functions	20
	m: Analysis Prompt: has-mereology	20
	4: Description Prompt: 1.observe-mereology	21
	Formulation of Mereologies	22
1.3.4	Part Attributes	22
	Inseparability of Attributes from Parts	22
	Attribute Quality and Attribute Value	23
	Endurant Attributes: Types and Functions	23
	n: Analysis Prompt: attribute-names	23
	The Attribute Value Observer	24
	5: Description Prompt: observe-attributes	24
	Attribute Categories	25
	Access to Attribute Values	26
	Event Values	26
	Shared Attributes	26
1.3.5	Components	27
	o: Analysis Prompt: has-components	27
	6: Description Prompt: observe-component-sorts	27
1.3.6	Materials	28
	p: Analysis Prompt: has-materials	28
	7: Description Prompt: observe-material-sorts	29

	Materials-related Part Attributes	29
	Laws of Material Flows and Leaks	30
1.3.7	“No Junk, No Confusion”	31
1.3.8	Discussion of Endurants	32
1.4	Perdurants	32
1.4.1	States	32
1.4.2	Actions, Events and Behaviours	32
	Time Considerations	33
	Actors	33
	Parts, Attributes and Behaviours	33
1.4.3	Discrete Actions	33
1.4.4	Discrete Events	34
1.4.5	Discrete Behaviours	34
	Channels and Communication	34
	Relations Between Attribute Sharing and Channels	34
1.4.6	Continuous Behaviours	35
1.4.7	Attribute Value Access	35
	Access to Static Attribute Values	36
	Access to External Attribute Values	36
	Access to Controllable Attribute Values	36
	Access to Event Values	36
1.4.8	Perdurant Signatures and Definitions	36
1.4.9	Action Signatures and Definitions	37
1.4.10	Event Signatures and Definitions	38
1.4.11	Discrete Behaviour Signatures and Definitions	38
	Behaviour Signatures	38
	Part Behaviours:	39
	Behaviour Definitions	39
	Process Schema I: Abstract <code>is_composite(p)</code>	39
	Process Schema II: Concrete <code>is_composite(p)</code>	40
	Process Schema III: <code>is_atomic(p)</code>	40
	Process Schema IV: Core Process (I)	41
	Process Schema V: Core Process (II)	41
1.4.12	Concurrency: Communication and Synchronisation	43
1.4.13	Summary and Discussion of Perdurants	43
	Summary	44
	Discussion	44
1.5	Closing	44
1.5.1	Analysis & Description Calculi for Other Domains	44
1.5.2	On Domain Description Languages	44
1.5.3	Comparison to Other Work	45
	Background: The TripTych Domain Ontology	45
	General	45
	0: Ontology Science & Engineering:	45
	1: Knowledge Engineering:	46
	Specific	47
	2: Database Analysis:	47
	3: Domain Analysis:	47
	4: Domain Specific Languages:	48
	5: Feature-oriented Domain Analysis (FODA):	48
	6: Software Product Line Engineering:	48
	7: Problem Frames:	48
	8: Domain Specific Software Architectures (DSSA):	49
	9: Domain Driven Design (DDD):	49

	10: Unified Modeling Language (UML):	49
	11: Requirements Engineering:	50
	Summary of Comparisons	50
1.5.4	Open Problems	50
1.5.5	Tony Hoare's Summary on 'Domain Modeling'	51
1.5.6	Beauty Is Our Business	51
1.6	Bibliographical Notes	51
2	Domain Facets: Analysis & Description	53
	Summary	53
2.1	Introduction	53
2.1.1	Facets of Domains	53
2.1.2	Structure of Paper	54
2.2	Intrinsics	54
2.2.1	Conceptual Analysis	54
2.2.2	Requirements	57
2.2.3	On Modeling Intrinsics	57
2.3	Support Technologies	57
2.3.1	Conceptual Analysis	57
2.3.2	Requirements	60
2.3.3	On Modeling Support Technologies	60
2.4	Rules & Regulations	61
2.4.1	Conceptual Analysis	61
2.4.2	Requirements	62
2.4.3	On Modeling Rules and Regulations	63
2.5	Scripts	63
2.5.1	Conceptual Analysis	63
2.5.2	Requirements	65
2.5.3	On Modeling Scripts	65
2.6	License Languages	65
2.6.1	Conceptual Analysis	65
	The Settings	65
	On Licenses	66
	Permissions and Obligations	66
2.6.2	The Pragmatics	66
	Digital Media	66
	Operations on Digital Works	66
	License Agreement and Obligation	67
	Two Assumptions	67
	Protection of the Artistic Electronic Works	67
	Health-care	67
	Patients and Patient Medical Records	67
	Medical Staff	67
	Professional Health Care	67
	Government Documents	68
	Documents	68
	Document Attributes	68
	Actor Attributes and Licenses	68
	Document Tracing	68
	Transportation	68
	A Synopsis	68
	A Pragmatics and Semantics Analysis	69
	Contracted Operations, An Overview	69
2.6.3	Schematic Rendition of License Language Constructs	69

	Licensing	69
	Licensors and Licensees	69
	Digital Media	70
	Heath-care	70
	Documents	70
	Transport	70
	Actors and Actions	70
2.6.4	Requirements	71
2.6.5	On Modeling License Languages	72
2.7	Management & Organisation	72
2.7.1	Conceptual Analysis	72
2.7.2	Requirements	76
2.7.3	On Modeling Management and Organisation	76
2.8	Human Behaviour	76
2.8.1	Conceptual Analysis	76
2.8.2	Requirements	77
2.8.3	On Modeling Human Behaviour	78
2.9	Conclusion	78
2.9.1	Completion	78
2.9.2	Integrating Formal Descriptions	78
2.9.3	The Impossibility of Describing Any Domain Completely	78
2.9.4	Rôles for Domain Descriptions	79
2.9.5	Grand Challenges of Informatics	79
2.10	Bibliographical Notes	79
3	Manifest Domains: Formal Models of Processes and Prompts	81
	Summary	81
3.1	Introduction	81
3.1.1	The Triptych Approach to Software Development	81
3.1.2	Method and Methodology	82
3.1.3	Related Work	82
3.1.4	Structure of Paper	83
3.2	Domain Analysis and Description	83
3.2.1	General	83
3.2.2	Entities	83
	a: Analysis Prompt: is-entity	83
3.2.3	Endurants and Perdurants	84
	b: Analysis Prompt: is-endurant	84
	c: Analysis Prompt: is-perdurant	84
3.2.4	Discrete and Continuous Endurants	85
	d: Analysis Prompt: is discrete	85
	e: Analysis Prompt: is continuous	85
3.2.5	Parts, Components and Materials	85
	General	85
	Part, Component and Material Prompts	85
	f: Analysis Prompt: is part	85
	g: Analysis Prompt: is component	85
	h: Analysis Prompt: is material	85
3.2.6	Atomic and Composite Parts	86
	i: Analysis Prompt: is-atomic	86
	j: Analysis Prompt: is-composite	86
3.2.7	On Observing Part Sorts	86
	Part Sort Observer Functions	86
	1: Description Prompt: 3.observe-part-sorts	86

	On Discovering Concrete Part Types	87
	k: Analysis Prompt: has-concrete-type	87
	2: Description Prompt: observe-concrete-type	87
	External and Internal Qualities of Parts	87
3.2.8	Unique Part Identifiers	87
	3: Description Prompt: 3.observe-unique-identifier	87
3.2.9	Mereology	88
	Part Mereology: Types and Functions	88
	l: Analysis Prompt: has-mereology	88
	4: Description Prompt: observe-mereology	88
3.2.10	Part, Material and Component Attributes	89
	5: Description Prompt: 3.observe-attributes	89
3.2.11	Components	89
	m: Analysis Prompt: has-components	89
	6: Description Prompt: observe-part-components	89
3.2.12	Materials	90
	Part Materials	90
	n: Analysis Prompt: has-materials	90
	7: Description Prompt: observe-part-material-sorts	90
	Material Parts	91
	o: Analysis Prompt: has-parts	91
	8: Description Prompt: observe-material-part-sorts	91
3.2.13	Components and Materials	91
3.2.14	Discussion	91
3.3	Syntax and Semantics	91
3.3.1	Form and Content	91
3.3.2	Syntactic and Semantic Types	92
3.3.3	Names and Denotations	92
3.4	A Model of the Domain Analysis & Description Process	92
3.4.1	Introduction	92
	A Summary of Prompts	92
	Preliminaries	93
	Initialising the Domain Analysis & Description Process	93
3.4.2	A Model of the Analysis & Description Process	93
	A Process State	93
	A Technicality	94
	Analysis & Description of Endurants	94
3.4.3	Discussion of The Process Model	96
	Termination	96
	Axioms and Proof Obligations	96
	Order of Analysis & Description: A Meaning of ' \oplus '	96
	Laws of Description Prompts	97
3.5	A Domain Analyser's & Describer's Domain Image	97
3.6	Domain Types	98
3.6.1	Syntactic Types: Parts, Materials and Components	98
	Syntax of Part, Material and Component Sort Names	98
	An Abstract Syntax of Domain Endurants	98
	Quality Types	99
	Well-formed Syntactic Types	99
	Well-formed Definitions	99
	No Recursive Definitions	100
3.6.2	Semantic Types: Parts, Materials and Components	101
	Part, Material and Component Values	101
	Quality Values	101

	Type Checking	102
3.7	From Syntax to Semantics and Back Again!	102
3.7.1	The Analysis & Description Prompt Arguments	102
3.7.2	Some Auxiliary Maps: Syntax to Semantics and Semantics to Syntax	102
3.7.3	M: A Meaning of Type Names	103
	Preliminaries	103
	Atomic Parts	103
	Abstract Composite Parts	104
	Concrete Composite Parts	104
	Materials	105
	Components	105
3.7.4	The ι Description Function	105
	Discussion	106
3.8	A Formal Description of a Meaning of Prompts	106
3.8.1	On Function Overloading	106
3.8.2	The Analysis Prompts	106
	is_entity	106
	is_endurant	107
	is_discrete	107
	is_part	107
	is_material [\equiv is_continuous]	107
	is_component	107
	is_atomic	107
	is_composite	108
	has_concrete_type	108
	has_mereology	108
	has_materials	108
	has_components	108
	has_parts	108
3.8.3	The Description Prompts	108
	A Description State	109
	observe_part_sorts	109
	observe_concrete_type	109
	observe_unique_identifier	110
	observe_mereology	110
	observe_part_attributes	110
	observe_part_material_sort	110
	observe_component_sort	111
	observe_material_part_sort	111
3.8.4	Discussion of The Prompt Model	111
3.9	Conclusion	111
3.9.1	What Has Been Achieved?	112
3.9.2	Are the Models Valid?	112
3.9.3	Future Work	112
4	To Every Manifest Domain Mereology a CSP Expression	113
	Summary	113
4.1	Introduction	113
4.1.1	Computing Science Mereology	113
4.1.2	From Domains via Requirements to Software	114
4.1.3	Domains: Science and Engineering	114
4.1.4	Contributions of This Chapter	115
4.1.5	Structure of This Chapter	115
4.2	Our Concept of Mereology	115

4.2.1	Informal Characterisation	115
4.2.2	Six Examples	116
	Air Traffic	116
	Buildings	116
	Financial Service Industry	117
	Machine Assemblies	118
	Oil Industry	118
	“The” Overall Assembly	118
	A Concretised Composite parts	119
	Railway Nets	119
	Discussion	120
4.3	An Abstract, Syntactic Model of Mereologies	120
4.3.1	Parts and Subparts	120
4.3.2	No “Infinitely” Embedded Parts	121
4.3.3	Unique Identifications	122
4.3.4	Attributes	123
	Attribute Names and Values	123
	Attribute Categories	123
4.3.5	Mereology	124
4.3.6	The Model	125
4.4	Some Part Relations	125
4.4.1	‘Immediately Within’	125
4.4.2	‘Transitive Within’	125
4.4.3	‘Adjacency’	126
4.4.4	Transitive ‘Adjacency’	126
4.5	An Axiom System	126
4.5.1	Parts and Attributes	126
	\mathcal{P} The Part Sort	126
	\mathcal{A} The Attribute Sort	126
4.5.2	The Axioms	126
	\mathbb{P} Part-hood	127
	\mathbb{PP} Proper Part-hood	127
	\mathbb{O} Overlap	127
	\mathbb{U} Underlap	127
	\mathbb{OX} Over-cross	127
	\mathbb{UX} Under-cross	127
	\mathbb{PO} Proper Overlap	127
4.5.3	Satisfaction	127
	A Proof Sketch	128
4.6	A Semantic CSP Model of Mereology	128
4.6.1	Parts \simeq Processes	128
4.6.2	Channels	128
4.6.3	Compilation	129
4.6.4	Discussion	130
	General	130
	Specific	130
4.7	Concluding Remarks	130
4.7.1	Relation to Other Work	130
4.7.2	What Has Been Achieved?	132
4.7.3	Future Work	132

Part II Requirements

5	From Domain Descriptions to Requirements Prescriptions	135
	Summary	135
5.1	Introduction	136
5.1.1	The Triptych Dogma of Software Development	136
5.1.2	Software As Mathematical Objects	136
5.1.3	The Contribution of This Paper	136
5.1.4	Some Comments on the chapter Content	136
5.1.5	Structure of Paper	137
5.2	An Example Domain: Transport	137
5.2.1	Endurants	137
5.2.2	Domain, Net, Fleet and Monitor	137
	Hubs and Links	138
	Unique Identifiers	138
	Mereology	139
	Attributes, I	140
5.2.3	Perdurants	143
	Hub Insertion Action	143
	Link Disappearance Event	143
	Road Traffic	143
	Global Values:	143
	Channels:	144
	Behaviour Signatures:	144
	The Road Traffic System Behaviour:	144
5.2.4	Domain Facets	146
5.3	Requirements	146
5.3.1	The Three Phases of Requirements Engineering	146
5.3.2	Order of Presentation of Requirements Prescriptions	146
5.3.3	Design Requirements and Design Assumptions	147
5.3.4	Derived Requirements	147
5.4	Domain Requirements	147
5.4.1	Domain Projection	148
	Domain Projection — Narrative	148
	Domain Projection — Formalisation	148
	Discussion	151
5.4.2	Domain Instantiation	152
	Domain Instantiation	152
	Domain Instantiation — Abstraction	155
	Discussion	155
5.4.3	Domain Determination	156
	Domain Determination: Example	156
	Discussion	157
5.4.4	Domain Extension	157
	Endurant Extensions	158
	[a] Vehicle Extension:	158
	[b] Road Pricing Calculator: Basic Sort and Unique Identifier:	159
	[c] Vehicle to Road Pricing Calculator Channel:	159
	[d] Toll-gate Sorts and Dynamic Types:	159
	[e] Toll-gate to Calculator Channels:	161
	[f] Road Pricing Calculator Attributes:	161
	[g] “Total” System State:	162
	[h] “Total” System Behaviour:	162
	Discussion	164
5.4.5	Requirements Fitting	164
5.4.6	Discussion	165

5.5	Interface and Derived Requirements	165
5.5.1	Interface Requirements	166
	Shared Phenomena	166
	Environment–Machine Interface:	166
	Shared Endurants	167
	Data Initialisation:	167
	Data Refreshment:	169
	Shared Perdurants	169
5.5.2	Derived Requirements	170
	Derived Actions	171
	Derived Events	172
	No Derived Behaviours	172
5.5.3	Discussion	173
	Derived Requirements	173
	Introspective Requirements	173
5.6	Machine Requirements	173
5.7	Conclusion	173
5.7.1	What has been Achieved ?	174
5.7.2	Present Shortcomings and Research Challenges	174
5.7.3	Comparison to “Classical” Requirements Engineering:	174
	[120, Deriving Specifications for Systems That Are	
	Connected to the Physical World]	174
	[79, Goal-directed Requirements Acquisition]	175

Part III Software

6	Domains: Their Simulation, Monitoring and Control	179
	A Divertimento of Ideas and Suggestions	179
	Model Driven Software Engineering and Software Product Lines	179
6.1	Introduction	179
6.2	Domain Descriptions	180
6.3	Interpretations	180
6.3.1	What Is a Domain-based Demo?	180
6.3.2	Simulations	181
6.3.3	Monitoring & Control	183
6.3.4	Machine Development	184
6.3.5	Verifiable Software Development	185
6.4	Conclusion	186

Part IV Closing

7	Conclusion	191
7.1	What Have We Achieved ?	191
7.2	Domain Science & Engineering	191
8	Bibliography	193

Part V Domain Descriptions

A	Credit Card Systems	205
	Summary	205
A.1	Introduction	205
A.2	Endurants	205
	A.2.1 Credit Card Systems	205
	A.2.2 Credit Cards	207
	A.2.3 Banks	207
	A.2.4 Shops	208
A.3	Perdurants	208
	A.3.1 Behaviours	208
	A.3.2 Channels	209
	A.3.3 Behaviour Interactions	209
	A.3.4 Credit Card	210
	A.3.5 Banks	211
	A.3.6 Shops	212
A.4	Discussion	213
B	Weather Information Systems	215
	Summary	215
B.1	On Weather Information Systems	215
	B.1.1 On a Base Terminology	215
	B.1.2 Some Illustrations	216
	Weather Stations	216
	Weather Forecasts	216
	Forecast Consumers	216
B.2	Major Parts of a Weather Information System	216
B.3	Endurants	217
	B.3.1 Parts and Materials	217
	B.3.2 Unique Identifiers	218
	B.3.3 Mereologies	219
	B.3.4 Attributes	219
	Clock, Time and Time-intervals	219
	Locations	220
	Weather	220
	Weather Stations	221
	Weather Data Interpreter	221
	Weather Forecasts	221
	Weather Forecast Consumer	221
B.4	Perdurants	222
	B.4.1 A WIS Context	222
	B.4.2 Channels	222
	B.4.3 WIS Behaviours	222
	B.4.4 Clock	223
	B.4.5 Weather Station	223
	B.4.6 Weather Data Interpreter	224
	collect_wd	224
	calculate_wf	224
	disseminate_wf	225
	B.4.7 Weather Forecast Consumer	225
B.5	Conclusion	226
	B.5.1 Reference to Similar Work	226
	B.5.2 What Have We Achieved ?	226
	B.5.3 What Needs to be Done Next ?	226
	B.5.4 Acknowledgements	226

Part VI Miscellaneous

C	RSL: The RAISE Specification Language – A Primer	229
C.1	Type Expressions	229
C.1.1	Atomic Types	229
C.1.2	Composite Types	229
	Concrete Composite Types	229
	Sorts and Observer Functions	230
C.2	Type Definitions	230
C.2.1	Concrete Types	230
C.2.2	Subtypes	231
C.2.3	Sorts — Abstract Types	231
C.3	The RSL Predicate Calculus	231
C.3.1	Propositional Expressions	231
C.3.2	Simple Predicate Expressions	231
C.3.3	Quantified Expressions	232
C.4	Concrete RSL Types: Values and Operations	232
C.4.1	Arithmetic	232
C.4.2	Set Expressions	232
	Set Enumerations	232
	Set Comprehension	232
C.4.3	Cartesian Expressions	232
	Cartesian Enumerations	232
C.4.4	List Expressions	233
	List Enumerations	233
	List Comprehension	233
C.4.5	Map Expressions	233
	Map Enumerations	233
	Map Comprehension	233
C.4.6	Set Operations	234
	Set Operator Signatures	234
	Set Examples	234
	Informal Explication	234
	Set Operator Definitions	235
C.4.7	Cartesian Operations	235
C.4.8	List Operations	235
	List Operator Signatures	235
	List Operation Examples	235
	Informal Explication	236
	List Operator Definitions	236
C.4.9	Map Operations	236
	Map Operator Signatures and Map Operation Examples	236
	Map Operation Explication	237
	Map Operation Redefinitions	237
C.5	λ-Calculus + Functions	238
C.5.1	The λ-Calculus Syntax	238
C.5.2	Free and Bound Variables	238
C.5.3	Substitution	238
C.5.4	α-Renaming and β-Reduction	238
C.5.5	Function Signatures	239
C.5.6	Function Definitions	239
C.6	Other Applicative Expressions	239
C.6.1	Simple let Expressions	239

C.6.2	Recursive let Expressions	240
C.6.3	Predicative let Expressions	240
C.6.4	Pattern and “Wild Card” let Expressions	240
C.6.5	Conditionals	240
C.6.6	Operator/Operand Expressions	241
C.7	Imperative Constructs	241
C.7.1	Statements and State Changes	241
C.7.2	Variables and Assignment	241
C.7.3	Statement Sequences and skip	241
C.7.4	Imperative Conditionals	242
C.7.5	Iterative Conditionals	242
C.7.6	Iterative Sequencing	242
C.8	Process Constructs	242
C.8.1	Process Channels	242
C.8.2	Process Composition	242
C.8.3	Input/Output Events	242
C.8.4	Process Definitions	243
C.9	Simple RSL Specifications	243
D	Indexes	245
D.1	Definitions	245
D.2	Concepts	249
D.3	Examples	257
D.4	Analysis Prompts	259
D.5	Description Prompts	259
D.6	RSL Symbols	259

Domains

Manifest Domains: Analysis & Description

Summary

We show¹ that manifest domains, an understanding of which are a prerequisite for software requirements prescriptions, can be precisely described: narrated and formalised. We show that such manifest domains can be understood as a collection of *endurant*, that is, basically spatial entities: parts, components and materials, and *perdurant*, that is, basically temporal entities: actions, events and behaviours. We show that parts can be modeled in terms of external qualities whether: atomic or composite parts, having internal qualities: unique identifications, mereologies, which model relations between parts, and attributes. We show that the manifest domain analysis endeavour can be supported by a calculus of manifest domain analysis prompts: `is_entity`, `is_endurant`, `is_perdurant`, `is_part`, `is_component`, `is_material`, `is_atomic`, `is_composite`, `is_stationary`, etcetera; and show how the manifest domain description endeavour can be supported by a calculus of manifest domain description prompts: `observe_part_sorts`, `observe_part_type`, `observe_components`, `observe_materials`, `observe_unique_identifier`, `observe_mereology`, `observe_attributes`. We show how to model attributes, essentially following Michael Jackson, [115], but with a twist: The attribute model introduces the attribute analysis prompts `is_static_attribute`, `is_dynamic_attribute`, `is_inert_attribute`, `is_reactive_attribute`, `is_active_attribute`, `is_autonomous_attribute`, `is_biddable_attribute` and `is_programmable_attribute`. The twist suggests ways of modeling “access” to the values of these kinds of attributes: the static attributes by simply “*copying*” them, once, the reactive and programmable attributes by “*carrying*” them as function parameters whose values are kept always updated, and the remaining, the `external_attributes`, by inquiring, when needed, as to their value, as if they were always offered on CSP-like channels [111]. We show how to model essential aspects of perdurants in terms of their signatures based on the concepts of endurants. And we show how one can “compile” descriptions of endurant parts into descriptions of perdurant behaviours. We do not show prompt calculi for perdurants. The above contributions express a method with principles, techniques and tools for constructing domain descriptions. It is important to realise that we do not wish to nor claim that the method can describe all that it is interesting to know about domains.

1.1 Introduction

The broader subject of this compendium is that of software development. The narrower subject of this chapter is that of manifest domain engineering. We shall see software development in the context of the TripTych approach (next section). The contribution of this compendium is twofold: the propagation of manifest domain engineering as a first phase of the development of a large class of software — and a set

¹ This chapter is based on [49].

of principles, techniques and tools for the engineering of the analysis & descriptions of manifest domains. These principles, techniques and tools are embodied in a set of analysis and description prompts. We claim that this embodiment — in the form of prompts — is novel.

1.1.1 The TripTych Approach to Software Engineering

We suggest a TripTych view of software engineering: *before hardware and software systems can be designed and coded we must have a reasonable grasp of “its” requirements; before requirements can be prescribed we must have a reasonable grasp of “the underlying” domain.* To us, therefore, software engineering contains the three sub-disciplines:

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

This paper contributes, we claim, to a methodology for domain analysis &² domain description. **Chapter 5, From Domain Descriptions to Requirements Prescriptions, [54]** shows how to “refine” domain descriptions into requirements prescriptions, and **Chapter 6, Domains: Their Simulation, Monitoring and Control, [48]** indicates more general relations between domain descriptions and domain demos, domain simulators and more general domain specific software.

The concept of **systems engineering** arises naturally in the TripTych approach. First: *domains can be claimed to be systems.* Secondly: *requirements are usually not restricted to software, but encompasses all the human and technological “assists” that must be considered.* Other than that we do not wish to consider domain analysis & description principles, techniques and tools specific to “systems engineering”.

1.1.2 Method and Methodology

Method

By a **method** we shall understand a “structured” set of principles for selecting and applying a number of techniques and tools for analysing problems and synthesizing solutions for a given domain³

The ‘structuring’ amounts, in this treatise on domain analysis & description, to the techniques and tools being related to a set of domain analysis & description “prompts”, “issued by the method”, prompting the domain engineer, hence carried out by the domain analyser & describer⁴ — conditional upon the result of other prompts.

Discussion

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

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

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

And the **main tools** of the TripTych domain analysis and description approach are the analysis and description prompts and the description language, here the Raise Specification Language RSL [96].

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

² When, as here, we write $A \& B$ we mean $A \& B$ to be one subject.

³ Definitions and examples are delimited by ☉ symbols.

⁴ We shall thus use the term domain engineer to cover both the analyser & the describer.

Methodology

By **methodology** we shall understand the study and knowledge about one or more methods⁵ ☉

1.1.3 Computer and Computing Science

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

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

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

1.1.4 What Is a Manifest Domain?

By ‘**domain**’ we mean the same as ‘problem domain’ [120]. We offer a number of complementary delimitations of what we mean by a manifest domain. But first some examples, “by name”!

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

A **manifest domain** is a human- and artifact-assisted arrangement of enduring, that is spatially “stable”, and perduring, that is temporally “fleeting” entities. Enduring entities are either parts or components or materials. Perduring entities are either actions or events or behaviours ☉

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

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

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

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

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

We shall in the rest of this paper just write ‘domain’ instead of ‘manifest domain’.

⁵ Please note our distinction between method and methodology. We often find the two, to us, separate terms used interchangeably.

1.1.5 What Is a Domain Description ?

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

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

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

By a **domain description** we shall thus understand a text which describes the entities of the domain: whether enduring or perdurant, and when enduring whether discrete or continuous, atomic or composite; or when perdurant whether actions, events or behaviours. as well as the qualities of these entities. So the task of the domain analyser cum describer is clear: There is a domain: right in front of our very eyes, and it is expected that that domain be described.

1.1.6 Towards a Methodology of Manifest Domain Analysis & Description

Practicalities of Domain Analysis & Description.

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

The Four Domain Analysis & Description “Players”.

We can say that there are four ‘players’ at work here. (i) the domain, (ii) the domain analyser & describer, (iii) the domain analysis & description method, and (iv) the evolving domain analysis & description (document). (i) The domain is there. The domain analyser & describer cannot change the domain. Analysing & describing the domain does not change it⁶. During the analysis & description process the domain can be considered inert. (It changes with the installation of such software as has been developed from the requirements developed from the domain description.) In the physical sense the domain will usually contain entities that are static (i.e., constant), and entities that are dynamic (i.e., variable). (ii) The domain analyser & domain describer is a human, preferably a scientist/engineer

Note 1.1. At the present time domain analysis appears to be partly an artistic, partly a scientific endeavour. Until such a time when domain analysis & description principles, techniques and tools have matured it will remain so., well-educated and trained in domain science & engineering. The domain analyser & describer observes the domain, analyses it according to a method and thereby produces a domain description. (iii) As a concept the method is here considered “fixed”. By ‘fixed’ we mean that its principles, techniques and tools do not change during a domain analysis & description. The domain analyser & describer may very well apply these principles, techniques and tools more-or-less haphazardly during domain analysis & description, flaunting the method, but the method remains invariant. The method, however, may vary from one domain analysis & description (project) to another domain analysis & description (project). Domain analysers & describers, may, for example, have become wiser from a project to the next. (iv) Finally there is the evolving *domain analysis & description*. That description is a text, usually both informal and formal.

⁶ Observing domains, such as we are trying to encircle the concept of domain, is not like observing the physical world at the level of subatomic particles. The experimental physicists’ instruments of observation change what is being observed.

Applying a *domain description prompt* to the domain yields an *additional domain description text* which is added to the thus evolving *domain description*. One may speculate of the rôle of the “input” domain description. Does it change? Does it help determine the additional domain description text? Etcetera. Without loss of generality we can assume that the “input” domain description is changed⁷ and that it helps determine the added text.

Of course, analysis & description is a trial-and-error, iterative process. During a sequence of analyses, that is, analysis prompts, the analyser “discovers” either more pleasing abstractions or that earlier analyses or descriptions were wrong, or that an entity either need be abstracted or made less abstract. So they are corrected.

An Interactive Domain Analysis & Description Dialogue.

We see domain analysis & description as a process involving the above-mentioned four ‘players’, that is, as a dialogue between the domain analyser & describer and the domain, where the dialogue is guided by the method and the result is the description. We see the method as a ‘player’ which issues prompts: alternating between: “analyse this” (analysis prompts) and “describe that” (synthesis or, rather, description prompts).

Prompts

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

A Domain Analysis & Description Language.

The ‘Domain Analysis & Description Language’ thus consists of a number of meta-functions, the prompts. The meta-functions have names (say `is_endurant`) and types, but have no formal definition. They are not computable. They are “performed” by the domain analysers & describers. These meta-functions are systematically introduced and informally explained in Sects. 1.2, 1.3 and 1.4.

The Domain Description Language.

The ‘Domain Description Language’ is RSL [96], the RAISE Specification Language [97]. With suitable, simple adjustments it could also be either of Alloy [114], Event B [1], VDM-SL [61, 62, 89] or Z [183]. We have chosen RSL because of its simple provision for defining sorts, expressing axioms, and postulating observers over sorts.

Domain Descriptions: Narration & Formalisation

Descriptions *must* be readable and *must* be mathematically precise.⁸ For that reason we decompose domain description fragments into clearly identified⁹ “pairs” of narrative texts and formal texts.

⁷ for example being “stylistically” revised.

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

⁹ The “clear identification” is here achieved by narrative text item and corresponding formula line numbers.

1.1.7 One Domain – Many Models ?

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

Example 6 . One Domain – Three Models: In this paper we shall bring many examples from a domain containing automobiles. (i) One domain model may focus on roads and vehicles, with roads being modeled in terms of atomic hubs (road intersections) and atomic links (road sections between immediately neighbouring hubs), and with automobiles being modeled in terms of atomic vehicles. (ii) Another domain model considers hubs of the former model as being composite, consisting, in addition to the “bare” hub, also of a signaling part — with automobiles remaining atomic vehicles, (iii) A third model focuses on vehicles, now as composite parts consisting of composite and atomic sub-parts such as they are relevant in the assembly-line manufacturing of cars¹⁰ □

1.1.8 Formal Concept Analysis

Domain analysis involves that of concept analysis. As soon as we have identified an entity for analysis we have identified a concept. The entity is usually a spatio-temporal, i.e., a physical thing. Once we speak of it, it becomes a concept. Instead of examining just one entity the domain analyser shall examine many entities. Instead of describing one entity the domain describer shall describe a class of entities. Ganter & Wille’s [95] addresses this issue.

A Formalisation

This section is a transcription of Ganter & Wille’s [95] *Formal Concept Analysis, Mathematical Foundations*, the 1999 edition, Pages 17–18.

Some Notation: By \mathcal{E} we shall understand the type of entities; by \mathbb{E} we shall understand a phenomenon of type \mathcal{E} ; by \mathcal{Q} we shall understand the type of qualities; by \mathbb{Q} we shall understand a quality of type \mathcal{Q} ; by \mathcal{E} -set we shall understand the type of sets of entities; by \mathbb{ES} we shall understand a set of entities of type \mathcal{E} -set; by \mathcal{Q} -set we shall understand the type of sets of qualities; and by \mathbb{QS} we shall understand a set of qualities of type \mathcal{Q} -set.

Definition 1.2. Formal Context: A **formal context** $\mathbb{K} := (\mathbb{ES}, \mathbb{I}, \mathbb{QS})$ consists of two sets; \mathbb{ES} of entities and \mathbb{QS} of qualities, and a relation \mathbb{I} between \mathbb{E} and \mathbb{Q} ■

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

Example enduring entities are a specific vehicle, another specific vehicle, etcetera; a specific street segment (link), another street segment, etcetera; a specific road intersection (hub), another specific road intersection,

¹⁰ The road nets of the first two models can be considered a zeroth model.

etcetera, a monitor. Example enduring entity qualities are (a vehicle) has mobility, (a vehicle) has velocity (≥ 0), (a vehicle) has acceleration, etcetera; (a link) has length (> 0), (a link) has location, (a link) has traffic state, etcetera.

Definition 1.3. Qualities Common to a Set of Entities: For any subset, $sES \subseteq ES$, of entities we can define \mathcal{DQ} for “derive[d] set of qualities”.

$$\begin{aligned} \mathcal{DQ} : \mathcal{E}\text{-set} &\rightarrow (\mathcal{E}\text{-set} \times \mathcal{I} \times \mathcal{Q}\text{-set}) \rightarrow \mathcal{Q}\text{-set} \\ \mathcal{DQ}(sES)(ES, I, QS) &\equiv \{Q \mid Q:\mathcal{Q}, E:\mathcal{E} \cdot E \in sES \wedge E \cdot I \cdot Q\} \\ \text{pre: } sES &\subseteq ES \end{aligned}$$

The above expresses: “the set of qualities common to entities in sES ” ■

Definition 1.4. Entities Common to a Set of Qualities: For any subset, $sQS \subseteq QS$, of qualities we can define \mathcal{DE} for “derive[d] set of entities”.

$$\begin{aligned} \mathcal{DE} : \mathcal{Q}\text{-set} &\rightarrow (\mathcal{E}\text{-set} \times \mathcal{I} \times \mathcal{Q}\text{-set}) \rightarrow \mathcal{E}\text{-set} \\ \mathcal{DE}(sQS)(ES, I, QS) &\equiv \{E \mid E:\mathcal{E}, Q:\mathcal{Q} \cdot Q \in sQS \wedge E \cdot I \cdot Q\}, \\ \text{pre: } sQS &\subseteq QS \end{aligned}$$

The above expresses: “the set of entities which have all qualities in sQS ” ■

Definition 1.5. Formal Concept: A **formal concept** of a context \mathbb{K} is a pair:

- (sQ, sE) where
 - ∞ $\mathcal{DQ}(sE)(E, I, Q) = sQ$ and
 - ∞ $\mathcal{DE}(sQ)(E, I, Q) = sE$;
- sQ is called the **intent** of \mathbb{K} and sE is called the **extent** of \mathbb{K} ■

Types Are Formal Concepts

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

Practicalities

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

Formal Concepts: A Wider Implication

The formal concepts of a domain form Galois Connections [95]. We gladly admit that this fact is one of the reasons why we emphasise formal concept analysis. At the same time we must admit that this paper does not do justice to this fact. We have experimented with the analysis & description of a number of domains, and have noticed such Galois connections, but it is, for us, too early to report on this. Thus we invite the reader to study this aspect of domain analysis.

1.1.9 Structure of Chapter

Sections 1.2–1.4 are the main sections of this chapter. They cover the analysis and description of endurants and perdurants. Section 1.2 introduce the concepts of entities, endurant entities and perdurant entities. Section 1.3 introduces the external qualities of parts, components and materials, and the internal qualities of unique part identifiers, part mereologies and part attributes. Section 1.4 complements Sect. 1.3. It covers analysis and description of perdurants. We consider the “compilation”, Sect. 1.4.11, of part descriptions, i.e., endurants, into behaviour descriptions to be a separate contribution. Section 1.5 concludes the chapter.

1.2 Entities

1.2.1 General

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

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

- *is_entity* — where *is_entity*(θ) holds if θ is an entity ◇¹¹

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

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

1.2.2 Endurants and Perdurants

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

That is, endurants “reside” in space. Endurants are, in the words of Whitehead [180], continuants.

Example 7 . Traffic System Endurants: Examples of traffic system endurants are: traffic system, road nets, fleets of vehicles, sets of hubs (i.e., street intersections), sets of links (i.e., street segments [between hubs]), and individual hubs, links and vehicles □

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

That is, perdurants “reside” in space and time. Perdurants are, in the words of Whitehead [180], occurrents.

Example 8 . Traffic System Perdurants: Examples of road net perdurants are: *insertion* and *removal* of hubs or links (actions), *disappearance* of links (events), vehicles *entering* or *leaving* the road net (actions), vehicles *crashing* (events) and *road traffic* (behaviour) □

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

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

¹¹ Analysis prompt definitions and description prompt definitions and schemes are delimited by ◇.

is_entity is a prerequisite prompt for *is_endurant* ◇

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

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

is_entity is a prerequisite prompt for *is_perdurant* ◇

In the words of Whitehead [180] — as communicated by Sowa [170, Page 70] — an endurant has stable qualities that enable its various appearances at different times to be recognised as the same individual; a perdurant is in a state of flux that prevents it from being recognised by a stable set of qualities.

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

1.2.3 Discrete and Continuous Endurants

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

Example 9 . Discrete Endurants: Examples of discrete endurants are a road net, a link, a hub, a vehicle, a traffic signal, etcetera □

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

Example 10 . Continuous Endurants: Examples of continuous endurants are water, oil, gas, sand, grain, etcetera □

Continuity shall here not be understood in the sense of mathematics. Our definition of ‘continuity’ focused on *prolonged, without interruption, in an unbroken series or pattern*. In that sense materials and components shall be seen as ‘continuous’,

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

- *is_discrete* — e is discrete if *is_discrete*(e) holds ◇

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

- *is_continuous* — e is continuous if *is_continuous*(e) holds ◇

1.2.4 An Upper Ontology Diagram of Domains

Figure 1.1 on the following page shows a so-called upper ontology for manifest domains. So far we have covered only a fraction of this ontology, as noted. By ontologies we shall here understand “*formal representations of a set of concepts within a domain and the relationships between those concepts*”. In Sect. 1.5.3 we shall review relations between our approach to modeling domains and that of many related modeling approaches, including the so-called ontology approach based on AI-models.

1.3 Endurants

This section brings a comprehensive treatment of the analysis and description of endurants.

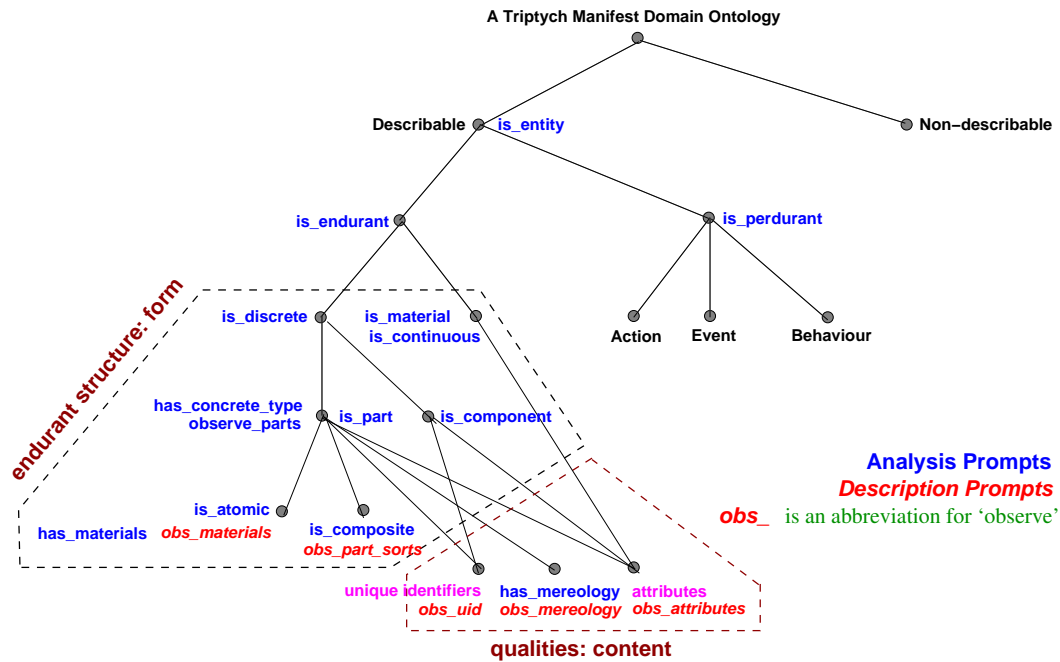


Fig. 1.1. An Upper Ontology for Domains

1.3.1 Parts, Components and Materials

General

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

We shall define the terms ‘unique identification’, ‘mereology’, and ‘attributes’ shortly.

Example 11 . Parts: Example 7 on Page 10 illustrated, and examples 15 on the facing page and 16 on Page 14 shall illustrate parts ☐

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

Example 12 . Components: Examples of components are: chairs, tables, sofas and book cases in a living room, letters, newspapers, and small packages in a mail box, machine assembly units on a conveyor belt, boxes in containers of a container vessel, etcetera ☐

“At the Discretion of the Domain Engineer”: We emphasise the following analysis and description aspects: (a) The domain is full of observable phenomena. It is the decision of the domain analyser cum describer whether to analyse and describe some such phenomena, that is, whether to include them in a domain model. (b) The borderline between an endurant being (considered) discrete or being (considered) continuous is fuzzy. It is the decision of the domain analyser cum describer whether to model an endurant as discrete or continuous. (c) The borderline between a discrete endurant being (considered) a part or being (considered) a component is fuzzy. It is the decision of the domain analyser cum describer whether to model a discrete endurant as a part or as a component. (d) In Sect. 1.4.11 we shall show how to “compile” parts into processes. A factor, therefore, in determining whether to model a discrete endurant as a part or as a component is whether we may consider a discrete endurant as also representing a process.

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

Example 13 . Materials: Examples of material endurants are: air of an air conditioning system, grain of a silo, gravel of a barge, oil (or gas) of a pipeline, sewage of a waste disposal system, and water of a hydro-electric power plant. \square

Example 14 . Parts Containing Materials: Pipeline units are here considered discrete, i.e., parts. Pipeline units serve to convey material \square

Part, Component and Material Analysis Prompts

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

- *is_part* — e is a part if *is_part*(e) holds \diamond

We remind the reader that the outcome of *is_part*(e) is very much dependent on the domain engineer's intention with the domain description, cf. Sect. 1.3.1 on the facing page.

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

- *is_component* — e is a component if *is_component*(e) holds \diamond

We remind the reader that the outcome of *is_component*(e) is very much dependent on the domain engineer's intention with the domain description, cf. Sect. 1.3.1 on the preceding page.

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

- *is_material* — e is a material if *is_material*(e) holds \diamond

We remind the reader that the outcome of *is_material*(e) is very much dependent on the domain engineer's intention with the domain description, cf. Sect. 1.3.1 on the facing page.

Atomic and Composite Parts

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

Definition 9 Atomic Part: **Atomic parts** are those which, in a given context, are deemed to not consist of meaningful, separately observable proper sub-parts \odot

A **sub-part** is a part \odot

Example 15 . Atomic Parts: Examples of atomic parts of the above mentioned domains are: aircraft¹² (of air traffic), demand/deposit accounts (of banks), containers (of container lines), documents (of document systems), hubs, links and vehicles (of road traffic), patients, medical staff and beds (of hospitals), pipes, valves and pumps (of pipeline systems), and rail units and locomotives (of railway systems) \square

Definition 10 Composite Part: **Composite parts** are those which, in a given context, are deemed to indeed consist of meaningful, separately observable proper sub-parts \odot

¹² Aircraft from the point of view of airport management are atomic. From the point of view of aircraft manufacturers they are composite.

Example 16 . Composite Parts: Examples of composite parts of the above mentioned domains are: airports and air lanes (of air traffic), banks (of a financial service industry), container vessels (of container lines), dossiers of documents (of document systems), routes (of road nets), medical wards (of hospitals), pipelines (of pipeline systems), and trains, rail lines and train stations (of railway systems). \square

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

- $is_atomic(p)$: p is an atomic endurant if $is_atomic(p)$ holds \diamond

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

- $is_composite(p)$: p is a composite endurant if $is_composite(p)$ holds \diamond

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

Whither Atomic or Composite: If we are analysing & describing vehicles in the context of a road net, cf. Example 7 on Page 10, then we have chosen to abstract vehicles as atomic; if, on the other hand, we are analysing & describing vehicles in the context of an automobile maintenance garage then we might very well choose to abstract vehicles as composite — the sub-parts being the object of diagnosis by the auto mechanics.

On Observing Part Sorts and Types

We use the term ‘sort’ when we wish to speak of an abstract type [164], that is, a type for which we do not wish to express a model¹³. We shall use the term ‘type’ to cover both abstract types and concrete types.

On Discovering Part Sorts

Recall from Sect. 1.1.8 on Page 9 that we “equate” a formal concept with a type (i.e., a sort). Thus, to us, a part sort is a set of all those entities which all have exactly the same qualities. Our aim now is to present the basic principles that let the domain analyser decide on part sorts. We observe parts one-by-one. (α) Our analysis of parts concludes when we have “lifted” our examination of a particular part instance to the conclusion that it is of a given sort, that is, reflects a formal concept.

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

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

- $observe_parts(p)$

directs the domain analyser to observe the sub-parts of p \diamond

Let us say the sub-parts of p are: $\{p_1, p_2, \dots, p_m\}$. (β) The analyser analyses, for each of these parts, p_{i_k} , which formal concept, i.e., sort, it belongs to; let us say that it is of sort P_k ; thus the sub-parts of p are of sorts $\{P_1, P_2, \dots, P_m\}$. Some P_k may be atomic sorts, some may be composite sorts.

The domain analyser continues to examine a finite number of other composite parts: $\{p_j, p_\ell, \dots, p_n\}$. It is then “discovered”, that is, decided, that they all consists of the same number of sub-parts $\{p_{i_1}, p_{i_2}, \dots, p_{i_m}\}$, $\{p_{j_1}, p_{j_2}, \dots, p_{j_m}\}$, $\{p_{\ell_1}, p_{\ell_2}, \dots, p_{\ell_m}\}$, ..., $\{p_{n_1}, p_{n_2}, \dots, p_{n_m}\}$, of the same, respective, part sorts. (γ) It is therefore concluded, that is, decided, that $\{p_i, p_j, p_\ell, \dots, p_n\}$ are all of the same part sort P with observable part sub-sorts $\{P_1, P_2, \dots, P_m\}$.

Above we have *type-font-highlighted* three sentences: (α, β, γ). When you analyse what they “pre-scribe” you will see that they entail a “depth-first search” for part sorts. The β sentence says it rather directly: “The analyser analyses, for each of these parts, p_k , which formal concept, i.e., part sort it belongs to.” To do this analysis in a proper way, the analyser must (“recursively”) analyse the parts “down” to their atomicity, and from the atomic parts decide on their part sort, and work (“recurse”) their way “back”, through possibly intermediate composite parts, to the p_k s. Of course, when the analyser starts by examining atomic parts then the analysis “recursion” is not necessary; as it is never necessary when the analyser proceeds “bottom-up”: analysing only such composite parts whose sub-parts have already been analysed

¹³ for example, in terms of the concrete types: sets, Cartesians, lists, maps, or other.

Part Sort Observer Functions

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

Domain Description Prompt 1 . *observe_part_sorts*: *If $\text{is_composite}(p)$ holds, then the analyser “applies” the domain description prompt*

- *observe_part_sorts(p)*

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

1. <i>observe_part_sorts</i> schema	
Narration:	
[s]	... narrative text on sorts ...
[o]	... narrative text on sort observers ...
[i]	... narrative text on sort recognisers ...
[p]	... narrative text on proof obligations ...
Formalisation:	
type	
[s]	$P,$
[s]	$P_i \ [1 \leq i \leq m]$ comment: $P_i \ [1 \leq i \leq m]$ abbreviates P_1, P_2, \dots, P_m
value	
[o]	obs_part_ $P_i: P \rightarrow P_i \ [1 \leq i \leq m]$
[i]	is_ $P_i: (P_1 P_2 \dots P_m) \rightarrow \mathbf{Bool} \ [1 \leq i \leq m]$
proof obligation [Disjointness of part sorts]	
[p]	$\forall p: (P_1 P_2 \dots P_m) \bullet$
[p]	$\wedge \{ \mathbf{is_}P_i(p) \equiv \wedge \{ \sim \mathbf{is_}P_j(p) \mid j \in \{1..m\} \setminus \{i\} \} \mid i \in \{1..m\} \}$

is_composite is a **prerequisite prompt** of *observe_part_sorts* \triangle

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

Example 17 . Composite and Atomic Part Sorts of Transportation: The following example illustrates the multiple use of the *observe_part_sorts* function: first to $\delta:\Delta$, a specific transport domain, Item 1, then to an $n:N$, the net of that domain, Item 2, and then to an $f:F$, the fleet of that domain, Item 3.

- 1 A transportation domain is viewed as composed from a net (of hubs and links), a fleet (of vehicles) and a monitor.
- 2 A transportation net is here seen as composed from a collection of hubs and a collection of links.
- 3 A fleet is here seen as a collection of vehicles.

The monitor is considered an atomic part.

type

1 Δ, N, F, M

value

1 **obs_part_** $N: \Delta \rightarrow N, \mathbf{obs_part_}F: \Delta \rightarrow F, \mathbf{obs_part_}M: \Delta \rightarrow M$

type

2 HS, LS

value

2 **obs_part_HS**: $N \rightarrow HS$, **obs_part_LS**: $N \rightarrow LS$

type

3 **VS**

value

3 **obs_part_VS**: $F \rightarrow VS$

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

On Discovering Concrete Part Types

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

- *has_concrete_type*(p).

is_discrete is a **prerequisite prompt** of *has_concrete_type* \diamond

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

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

- *observe_concrete_type*(p)¹⁴

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

2. observe_concrete_type schema

Narration:

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

Formalisation:

type

- [t₁] $S_1, S_2, \dots, S_m, \dots, S_n,$
- [t₂] $T = \mathcal{E}(S_1, S_2, \dots, S_n)$

value

- [o] **obs_part_T**: $P \rightarrow T$

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

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

Example 18. **Concrete Part Types of Transportation**: We continue Example 17 on the previous page:

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

¹⁴ *has_concrete_type* is a **prerequisite prompt** of *observe_concrete_type*.

- 5 Hubs and links are, until further analysis, part sorts.
- 6 A collection of vehicles is here seen as a set of vehicles.
- 7 Vehicles are, until further analysis, part sorts.

type

- 4 $H_s = \text{H-set}, L_s = \text{L-set}$
- 5 H, L
- 6 $V_s = \text{V-set}$
- 7 V

value

- 4 $\text{obs_part_Hs}: HS \rightarrow H_s, \text{obs_part_Ls}: LS \rightarrow L_s$
- 6 $\text{obs_part_Vs}: VS \rightarrow V_s \square$

Forms of Part Types

Usually it is wise to restrict the part type definitions, $T_i = \mathcal{E}_i(Q, R, \dots, S)$, to simple type expressions. $T = A\text{-set}$ or $T = A^*$ or $T = ID \rightarrow_m A$ or $T = A_t | B_t | \dots | C_t$ where ID is a sort of unique identifiers, $T = A_t | B_t | \dots | C_t$ defines the disjoint types $A_t = \text{mk}A_t(s:A_s)$, $B_t = \text{mk}B_t(s:B_s)$, ..., $C_t = \text{mk}C_t(s:C_s)$, and where A, A_s, B_s, \dots, C_s are sorts. Instead of $A_t = \text{mk}A_t(a:A_s)$, etc., we may write $A_t::A_s$ etc.

Part Sort and Type Derivation Chains

Let P be a composite sort. Let P_1, P_2, \dots, P_m be the part sorts “discovered” by means of $\text{observe_part_sorts}(p)$ where $p:P$. We say that P_1, P_2, \dots, P_m are (immediately) **derived** from P . If P_k is derived from P_j and P_j is derived from P_i , then, by transitivity, P_k is **derived** from P_i .

No Recursive Derivations

We “mandate” that if P_k is derived from P_j then there can be no P derived from P_j such that P is P_j , that is, P_j cannot be derived from P_j .

That is, we do not allow recursive domain sorts.

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

Names of Part Sorts and Types

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

Example 19 . Container Line Sorts: Our example is that of a container line with container vessels and container terminal ports.

- 8 A container line contains a number of container vessels
and a number of container terminal ports,
as well as other parts.
- 9 A container vessel contains a container stowage area, etc.
- 10 A container terminal port contains a container stowage area, etc.
- 11 A container stowage areas contains a set of uniquely identified container bays.
- 12 A container bay contains a set of uniquely identified container rows.

- 13 A container row contains a set of uniquely identified container stacks.
 14 A container stack contains a stack, i.e., a first-in, last-out sequence of containers.
 15 Containers are further undefined.

After a some slight editing we get:

type CL VS, VI, V, Vs = VI \rightarrow_{h} V, PS, PI, P, Ps = PI \rightarrow_{h} P value obs_part_VS: CL \rightarrow VS obs_part_Vs: VS \rightarrow Vs obs_part_PS: CL \rightarrow PS obs_part_Ps: CTPS \rightarrow CTPs type CSA value obs_part_CSA: V \rightarrow CSA obs_part_CSA: P \rightarrow CSA	type BAYS, BI, BAY, Bays=BI \rightarrow_{h} BAY ROWS, RI, ROW, Rows=RI \rightarrow_{h} ROW STKS, SI, STK, Stks=SI \rightarrow_{h} STK C value obs_part_BAYS: CSA \rightarrow BAYS, obs_part_Bays: BAYS \rightarrow Bays obs_part_ROWS: BAY \rightarrow ROWS, obs_part_Rows: ROWS \rightarrow Rows obs_part_STKS: ROW \rightarrow STKS, obs_part_Stks: STKS \rightarrow Stks obs_part_Stk: STK \rightarrow C*
--	---

Note that `observe_part_sorts(v:V)` and `observe_part_sorts(p:P)` both yield CSA \square

More On Part Sorts and Types

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

Example 1.6. We comment on Example 17, Page 15: Parts of type Δ and N are composed from three, respectively two abstract sub-parts of distinct types \square

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

Example 1.7. We comment on Example 17: Parts of type N, F, HS, LS and VS are all composite \square

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

Example 1.8. We comment on Example 17: Parts of type N are composed from three sub-parts \square

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

Example 1.9. We comment on Example 17: Parts of type HS, LS and VS are composed from an indefinite numbers of hubs, links and vehicles, respectively \square

Example 20 . Pipeline Parts:

- 16 A pipeline consists of an indefinite number of pipeline units.
 17 A pipeline units is either a well, or a pipe, or a pump, or a valve, or a fork, or a join, or a sink.
 18 All these unit sorts are atomic and disjoint.

type
 16 PL, U, We, Pi, Pu, Va, Fo, Jo, Si
 16 Well, Pipe, Pump, Valv, Fork, Join, Sink
value
 16 obs_part_Us: PL \rightarrow U-set
type
 17 U == We | Pi | Pu | Va | Fo | Jo | Si
 18 We::Well, Pi::Pipe, Pu::Pump, Va::Valv, Fo::Fork, Jo::Join, Si::Sink \square

External and Internal Qualities of Parts

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

Three Categories of Internal Qualities

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

1.3.2 Unique Part Identifiers

We introduce a notion of unique identification of parts. We assume (i) that all parts, p , of any domain P , have unique identifiers, (ii) that unique identifiers (of parts $p:P$) are abstract values (of the unique identifier sort PI of parts $p:P$), (iii) such that distinct part sorts, P_i and P_j , have distinctly named unique identifier sorts, say PI_i and PI_j , (iv) that all $\pi_i:PI_i$ and $\pi_j:PI_j$ are distinct, and (v) that the observer function `uid_P` applied to p yields the unique identifier, say $\pi:PI$, of p .

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

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

- *observe_unique_identifier*

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

3. *observe_unique_identifier* schema

Narration:

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

Formalisation:

- type**
- [s] PI
- value**
- [u] `uid_P`: $P \rightarrow PI$
- axiom**
- [a] \mathcal{U}

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

Example 21 . Unique Transportation Net Part Identifiers: We continue Example 17 on Page 15.

19 Links and hubs have unique identifiers

20 and unique identifier observers.

type

19 LI, HI

value

20 uid_LI: $L \rightarrow LI$

20 uid_HI: $H \rightarrow HI$

axiom [Well-formedness of Links, L, and Hubs, H]

19 $\forall l, l': L \cdot \text{uid_LI}(l) = \text{uid_LI}(l') \Rightarrow l = l'$,

19 $\forall h, h': H \cdot \text{uid_HI}(h) = \text{uid_HI}(h') \Rightarrow h = h' \quad \square$

Axiom 19, although expressed for links and hubs of road nets, applies in general: Two parts with the same unique part identifiers are indeed one and the same part.

1.3.3 Mereology

Mereology is the study and knowledge of parts and part relations. Mereology, as a logical/philosophical discipline, can perhaps best be attributed to the Polish mathematician/logician Stanisław Leśniewski [68, 43].

Part Relations

Which are the relations that can be relevant for part-hood? We give some examples. Two otherwise distinct parts may share attribute values.¹⁵

Example 22 . Shared Timetable Mereology (I): Two or more distinct public transport busses may “run” according to the (identically) same, thus “shared”, bus time table (cf. Example 32 on Page 27) \square

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

Example 23 . Topological Connectedness Mereology: (i) two rail units may be connected (i.e., adjacent); (ii) a road link may be connected to two road hubs; (iii) a road hub may be connected to zero or more road links; (iv) distinct vehicles of a road net may be monitored by one and the same road pricing sub-system \square

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

Part Mereology: Types and Functions

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

- *has_mereology*

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

We can define a **mereology type** as a type \mathcal{E} xpression over unique [part] identifier types. We generalise to unique [part] identifiers over a definite collection of part sorts, P_1, P_2, \dots, P_n , where the parts $p_1:P_1, p_2:P_2, \dots, p_n:P_n$ are not necessarily (immediate) sub-parts of some part $p:P$.

type

PI_1, PI_2, \dots, PI_n

$MT = \mathcal{E}(PI_1, PI_2, \dots, PI_n),$

¹⁵ For the concept of attribute value see Sect. 1.3.4 on Page 23.

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

- *observe_mereology*

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

4. observe_mereology schema

Narration:

[t] ... narrative text on mereology type ...

[m] ... narrative text on mereology observer ...

[a] ... narrative text on mereology type constraints ...

Formalisation:

type

[t] $MT^{16} = \mathcal{E}(PI1, PI2, \dots, PI_m)$

value

[m] **obs_mereo_P**: $P \rightarrow MT$

axiom [Well-formedness of Domain Mereologies]

[a] $\mathcal{A}(MT)$

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

Example 24 . Road Net Part Mereologies: We continue Example 17 on Page 15 and Example 21 on Page 19.

- 21 Links are connected to exactly two distinct hubs.
- 22 Hubs are connected to zero or more links.
- 23 For a given net the link and hub identifiers of the mereology of hubs and links must be those of links and hubs, respectively, of the net.

type

21 $LM' = HI\text{-set}, LM = \{ |his:HI\text{-set} \cdot card(his)=2 | \}$

22 $HM = LI\text{-set}$

value

21 **obs_mereo_L**: $L \rightarrow LM$

22 **obs_mereo_H**: $H \rightarrow HM$

axiom [Well-formedness of Road Nets, N]

23 $\forall n:N, l:L, h:H \cdot$

23 $l \in \text{obs_part_Ls}(\text{obs_part_LS}(n))$

23 $\wedge h \in \text{obs_part_Hs}(\text{obs_part_HS}(n))$

23 $\Rightarrow \text{obs_mereo_L}(l) \subseteq \cup \{ \text{uid_H}(h) \mid h \in \text{obs_part_Hs}(\text{obs_part_HS}(n)) \}$

23 $\wedge \text{obs_mereo_H}(h) \subseteq \cup \{ \text{uid_L}(l) \mid l \in \text{obs_part_Ls}(\text{obs_part_LS}(n)) \}$ \square

Example 25 . Pipeline Parts Mereology: We continue Example 20 on Page 18. Pipeline units serve to conduct fluid or gaseous material. The flow of these occur in only one direction: from so-called input to so-called output.

- 24 Wells have exactly one connection to an output unit.

¹⁶ *MT* will be used several times in Sect. 1.4.11.

- 25 Pipes, pumps and valves have exactly one connection from an input unit and one connection to an output unit.
 26 Forks have exactly one connection from an input unit and exactly two connections to distinct output units.
 27 Joins have exactly two connections from distinct input units and one connection to an output unit.
 28 Sinks have exactly one connection from an input unit.
 29 Thus we model the mereology of a pipeline unit as a pair of disjoint sets of unique pipeline unit identifiers.

type

29 $UM' = (UI_set \times UI_set)$

29 $UM = \{ \{ (iuis, ouis) : UM' \cdot iuis \cap ouis = \{ \} \} \}$

value

29 **obs_mereo_U**: UM

axiom [Well-formedness of Pipeline Systems, PLS (0)]

$\forall pl: PL, u: U \cdot u \in \mathbf{obs_part_Us}(pl) \Rightarrow$

let $(iuis, ouis) = \mathbf{obs_mereo_U}(u)$ **in**

case $(\mathbf{card} \ iuis, \mathbf{card} \ ouis)$ **of**

24 $(0, 1) \rightarrow \mathbf{is_We}(u),$

25 $(1, 1) \rightarrow \mathbf{is_Pi}(u) \vee \mathbf{is_Pu}(u) \vee \mathbf{is_Va}(u),$

26 $(1, 2) \rightarrow \mathbf{is_Fo}(u),$

27 $(2, 1) \rightarrow \mathbf{is_Jo}(u),$

28 $(1, 0) \rightarrow \mathbf{is_Si}(u), _ \rightarrow \mathbf{false}$

end end

Example 38 on Page 30 (axiom Page 30) and Example 39 on Page 31 (axiom Page 31) illustrates the need to constrain the sets of enduring entities denoted by definitions of part sort, unique identifier and mereology attribute definitions \square

Formulation of Mereologies

The `observe_mereology` domain descriptor, Page 21, may give the impression that the mereo type MT can be described “at the point of issue” of the `observe_mereology` prompt. Since the MT type expression may, in general, depend on any part sort the mereo type MT can, for some domains, “first” be described when all part sorts have been dealt with. **Chapter 3, Domain Analysis and Description – Formal Models of Processes and Prompts, [52]** presents a model of one form of evaluation of the TripTych analysis and description prompts.

1.3.4 Part Attributes

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

Inseparability of Attributes from Parts

Parts are typically recognised because of their spatial form and are otherwise characterised by their intangible, but measurable attributes. That is, whereas endurants, whether discrete (as are parts and components) or continuous (as are materials), are physical, tangible, in the sense of being spatial [or being abstractions, i.e., concepts, of spatial endurants], attributes are intangible: cannot normally be touched¹⁷, or seen¹⁸, but

¹⁷ One can see the red colour of a wall, but one touches the wall.

¹⁸ One cannot see electric current, and one may touch an electric wire, but only if it conducts high voltage can one know that it is indeed an electric wire.

can be objectively measured¹⁹. Thus, in our quest for describing domains where humans play an active rôle, we rule out subjective “attributes”: feelings, sentiments, moods. Thus we shall abstain, in our domain science also from matters of aesthetics. We learned from Sect. 1.1.8 that a formal concept, that is, a type, consists of all the entities which all have the same qualities. Thus removing a quality from an entity makes no sense: the entity of that type either becomes an entity of another type or ceases to exist (i.e., becomes a non-entity)!

Attribute Quality and Attribute Value

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

Example 26 . Attribute Propositions and Other Values: A particular street segment (i.e., a link), say ℓ , satisfies the proposition (attribute) `has_length`, and may then have value `length 90 meter` for that attribute. Another link satisfies the same proposition but has another value; and yet another link satisfies the same proposition and may have the same value. That is: all links satisfies `has_length` and has some value for that attribute. A particular road transport domain, δ , has three immediate sub-parts: `net`, n , `fleet`, f , and `monitor` m ; typically `nets` has `net_name` and `has_net_owner` proposition attributes with, for example, `US Interstate Highway System` respectively `US Department of Transportation` as values for those attributes. There may be other aspects of the net value n \square

Endurant Attributes: Types and Functions

Let us recall that attributes cover qualities other than unique identifiers and mereology. Let us then consider that parts have one or more attributes. These attributes are qualities which help characterise “what it means” to be a part. Note that we expect every part to have at least one attribute.

Example 27 . Atomic Part Attributes: Examples of attributes of atomic parts such as a human are: *name, gender, birth-date, birth-place, nationality, height, weight, eye colour, hair colour*, etc. Examples of attributes of transport net links are: *length, location, 1 or 2-way link, link condition*, etc. \square

Example 28 . Composite Part Attributes: Examples of attributes of composite parts such as a road net are: *owner, public or private net, free-way or toll road, a map of the net*, etc. Examples of attributes of a group of people could be: *statistic distributions of gender, age, income, education, nationality, religion*, etc. \square

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

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

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

η is a type operator. Applied to a type A it yields is name²⁰ \diamond

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

¹⁹ That is, we restrict our domain analysis with respect to attributes to such quantities which are observable, say by mechanical, electrical or chemical instruments. Once objective measurements can be made of human feelings, beauty, and other, we may wish to include these “attributes” in our domain descriptions.

²⁰ Normally, in non-formula texts, type A is referred to by ηA . In formulas A denote a type, that is, a set of entities. Hence, when we wish to emphasize that we speak of the name of that type we use ηA . But often we omit the distinction

The Attribute Value Observer

The “built-in” description language operator

- **attr_A**

applies to parts, $p:P$, where $\eta A \in \text{attribute_names}(p)$. It yields the value of attribute A of p .

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

- *observe_attributes.*

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

5. *observe_attributes* schema

Narration:

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

Formalisation:

type

- [t] $A_i \ [1 \leq i \leq n]$

value

- [o] **attr_{A_i}**: $P \rightarrow A_i \ [1 \leq i \leq n]$
- [i] **is_{A_i}**: $(A_1 | A_2 | \dots | A_n) \rightarrow \text{Bool} \ [1 \leq i \leq n]$

proof obligation [Disjointness of Attribute Types]

- [p] $\forall \delta: \Delta$
- [p] **let** P be any part sort **in** [the Δ domain description]
- [p] **let** $a: (A_1 | A_2 | \dots | A_n)$ **in** **is_{A_i}**(a) \neq **is_{A_j}**(a) **end end** [$i \neq j, 1 \leq i, j \leq n$]

The **type** (or rather sort) definitions: A_1, A_2, \dots, A_n , inform us that the domain analyser has decided to focus on the distinctly named A_1, A_2, \dots, A_n attributes.²¹ And the **value** clauses **attr_{A₁}**: $P \rightarrow A_1$, **attr_{A₂}**: $P \rightarrow A_2$, ..., **attr_{A_n}**: $P \rightarrow A_n$ are then “automatically” given: if a part, $p:P$, has an attribute A_i then there is postulated, “by definition” [eureka] an attribute observer function **attr_{A_i}**: $P \rightarrow A_i$ etcetera \triangle

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

- **has_attribute_{A₁}**(p), **has_attribute_{A₂}**(p), ..., and **has_attribute_{A_n}**(p)

holds. Thus the observer functions **attr_{A₁}**, **attr_{A₂}**, ..., **attr_{A_n}** can be applied to p in P and yield attribute values $a_1:A_1, a_2:A_2, \dots, a_n:A_n$ respectively.

Example 29 . Road Hub Attributes: After some analysis a domain analyser may arrive at some interesting hub attributes:

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

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

32 such that

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

b and the state is in the state space.

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

²¹ The attribute type names are not like type names of, for example, a programming language. Instead they are chosen by the domain analyser to reflect on domain phenomena. Cf. Example 27 on the preceding page and Example 28.

```

type
30    $H\Sigma = (LI \times LI)\text{-set}$ 
31    $H\Omega = H\Sigma\text{-set}$ 
value
30    $\text{attr\_H}\Sigma : H \rightarrow H\Sigma$ 
31    $\text{attr\_H}\Omega : H \rightarrow H\Omega$ 
axiom [Well-formedness of Hub States,  $H\Sigma$ ]
32    $\forall h:H \bullet \text{let } h\sigma = \text{attr\_H}\Sigma(h) \text{ in}$ 
32a     $\{li,li' \mid li,li':LI \bullet (li,li') \in h\sigma\} \subseteq \text{obs\_mereo\_H}(h)$ 
32b     $\wedge h\sigma \in \text{attr\_H}\Omega(h)$ 
32   end  $\square$ 

```

Attribute Categories

One can suggest a hierarchy of part attribute categories: static or dynamic values — and within the dynamic value category: inert values or reactive values or active values — and within the dynamic active value category: autonomous values or biddable values or programmable values. We now review these attribute value types. The review is based on [115, M.A. Jackson]. **Part attributes** are either constant or varying, i.e., **static** or **dynamic** attributes. By a **static attribute**, $a:A$, $\text{is_static_attribute}(a)$, we shall understand an attribute whose values are constants, i.e., cannot change. By a **dynamic attribute**, $a:A$, $\text{is_dynamic_attribute}(a)$, we shall understand an attribute whose values are variable, i.e., can change. **Dynamic attributes** are either inert, reactive or active attributes. By an **inert attribute**, $a:A$, $\text{is_inert_attribute}(a)$, we shall understand a dynamic attribute whose values only change as the result of external stimuli where these stimuli prescribe properties of these new values. By a **reactive attribute**, $a:A$, $\text{is_reactive_attribute}(a)$, we shall understand a dynamic attribute whose values, if they vary, change value in response to the change of other attribute values. By an **active attribute**, $a:A$, $\text{is_active_attribute}(a)$, we shall understand a dynamic attribute whose values change (also) of its own volition. **Active attributes** are either autonomous, biddable or programmable attributes. By an **autonomous attribute**, $a:A$, $\text{is_autonomous_attribute}(a)$, we shall understand a dynamic active attribute whose values change value only “on their own volition”. The values of an autonomous attributes are a “law unto themselves and their surroundings”. By a **biddable attribute**, $a:A$, $\text{is_biddable_attribute}(a)$, (of a part) we shall understand a dynamic active attribute whose values are prescribed but may fail to be observed as such. By a **programmable attribute**, $a:A$, $\text{is_programmable_attribute}(a)$, we shall understand a dynamic active attribute whose values can be prescribed.

Example 30 . Static and Dynamic Attributes: Link lengths can be considered **static**. Buses (i.e., vehicles) have a *timetable* attribute which is **inert**, i.e., can change, only when the bus company decides so. The weather can be considered **autonomous**. Pipeline valve units include the two attributes of *valve opening* (open, close) and *internal flow* (measured, say gallons per second). The valve opening attribute is of the **biddable** attribute category. The flow attribute is **reactive** (flow changes with valve opening/closing). Hub states (red, yellow, green) can be considered **biddable**: one can “try” set the signals but the electro-mechanics may fail. Bus companies **program** their own timetables, i.e., bus company timetables are **programmable** — are computers \square

External Attributes: By an **external attribute** we shall understand a dynamic attribute which is not a biddable or a programmable attribute \odot . The idea of external attributes is this: They are the attributes whose values are set by factors “outside” the part of which they are an attribute. In contrast, the programmable (and biddable) attributes have their values deterministically (non-deterministically) set by the part [behaviour] of which they are an attribute.

Controllable Attributes: By a **controllable attribute** we shall understand either a biddable or a programmable attribute \odot .

Figure 1.2 on the next page captures an attribute value ontology.

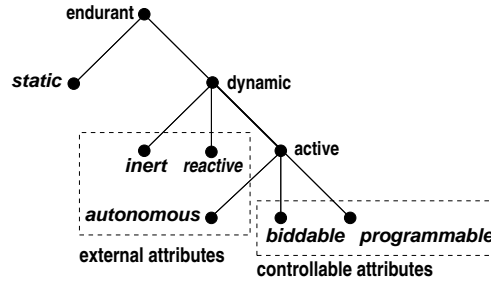


Fig. 1.2. Attribute Value Ontology

Access to Attribute Values

In an action, event or a behaviour description (Sect. 1.4.9) static values of parts, p , (say of type A) can be “copied”, $\text{attr}_A(p)$, and still retain their (static) value. But, for action, event or behaviour descriptions, external dynamic values of parts, p , cannot be “copied”, but $\text{attr}_A(p)$ must be “performed” every time they are needed. That is: static values require at most one domain access, whereas external attribute values require repeated domain accesses. We shall return to the issue of attribute value access in Sect. 1.4.7.

Event Values

Among the external attribute values we observe a new kind of value: the **event values**. We may optionally ascribe ordinarily typed, say A , values, $a:A$, with event attributes. By an **event attribute** we shall understand an attribute whose values are either “nil” ([f]or “absent”), or are some more definite value ($a:A$) \odot . Event values *occur* instantaneously. They can be thought of as the raising of a signal followed immediately by the lowering of that signal.

Example 31 . Event Attributes: (i) The passing of a vehicle past a tollgate is an event. It occurs at a usually unpredictable time. It otherwise “carries” no specific value. (ii) The identification of a vehicle by a tollgate sensor is an event. It occurs at a usually unpredictable time. It specifically “carries” a vehicle identifier value \square

Event attributes are not to be confused with event perdurants. External attributes are either event attributes or are not. More on access to event attribute values in Sect. 1.4.7 on Page 36.

Shared Attributes

Normally part attributes of different part sorts are distinctly named. If, however, $\text{observe_attributes}(p_i:P_i)$ and $\text{observe_attributes}(p_j:P_j)$, for any two distinct part sorts, P_i and P_j , of a domain, “discovers” identically named attributes, say A , then we say that parts $p_i:P_i$ and $p_j:P_j$ **share** attribute A . that is, that $a:\text{attr}_A(p_i)$ (and $a':\text{attr}_A(p_j)$) is a **shared attribute** (with $a=a'$ always (\square) holding).

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

Example 1.10. Shared Attributes. Examples of shared attributes: (i) Bus timetable attributes have the same value as the fleet timetable attribute – cf. Example 32 below. (ii) A link incident upon or emanating from a hub shares the connection between that link and the hub as an attribute. (iii) Two pipeline units²², p_i with unique identifier π_i , and p_j with unique identifier π_j , that are connected, such that an outlet marked π_j of p_i “feeds into” inlet marked π_i of p_j , are said to share the connection (modeled by, e.g., $\{(\pi_i, \pi_j)\}$) \square

²² See Example 25 on Page 21

Example 32 . Shared Timetables: The fleet and vehicles of Example 17 on Page 15 and Example 18 on Page 16 is that of a bus company.

- 34 From the fleet and from the vehicles we observe unique identifiers.
- 35 Every bus mereology records the same one unique fleet identifier.
- 36 The fleet mereology records the set of all unique bus identifiers.
- 37 A bus timetable is a shared fleet and bus attribute.

type

34 FI, VI, BT

value

34 uid_F: $F \rightarrow FI$

34 uid_V: $V \rightarrow VI$

35 obs_mereo_F: $F \rightarrow VI\text{-set}$ [cf. Sect. 1.3.3 on Page 20]

36 obs_mereo_V: $V \rightarrow FI$

37 attr_BT: $(F|V) \rightarrow BT$

axiom

$\square \forall f:F \Rightarrow$

$\forall v:V \cdot v \in \text{obs_part_Vs}(\text{obs_part_VC}(f)) \cdot \text{attr_BT}(f) = \text{attr_BT}(v) \quad \square$

The simple identical attribute name-sharing first outlined above may be generalised. If P_i and P_j are two distinct types of a domain, then if an attribute, A, of P_i is to be shared with an attribute, B, of P_j , attribute B must be expressed in terms of A.

1.3.5 Components

We refer to Sect. 1.3.1 on Page 12 for a first coverage of the concept of components: definition and examples. Components are discrete endurants which the domain analyser & describer has chosen to not endow with **internal qualities**.

Example 33 . Parts and Components: We observe components as associated with atomic parts: The contents, that is, the collection of zero, one or more boxes, of a container are the components of the container part. Conveyor belts transport machine assembly units and these are thus considered the components of the conveyor belt \square

We now complement the `observe_part_sorts` (of Sect. 1.3.1). We assume, without loss of generality, that only atomic parts may contain components. Let $p:P$ be some atomic part.

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

- `has_components(p)`

yields **true** if atomic part p may contain zero, one or more components otherwise false \diamond

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

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

- `observe_component_sorts(p)`

yields the component sorts and component sort observer domain description text according to the following schema: – whether or not the actual part p contains any components:

6. `observe_component_sorts_P` schema

Narration:

- [s] ... narrative text on component sorts ...
- [o] ... narrative text on component observers ...

[i] ... narrative text on component sort recognisers ...
 [p] ... narrative text on component sort proof obligations ...

Formalisation:**type**

[s] $K1, K2, \dots, Kn$
 [s] $K = K1 | K2 | \dots | Kn$
 [s] $KS = K\text{-set}$

value

[o] **components**: $P \rightarrow KS$
 [i] **is** $_K$: $(K1 | K2 | \dots | Kn) \rightarrow \mathbf{Bool} [1 \leq i \leq n]$

Proof Obligation: [Disjointness of Component Sorts]

[p] $\forall k_i: (K1 | K2 | \dots | Kn) \bullet$
 [p] $\wedge \{ \mathbf{is}_K(k_i) \equiv \wedge \{ \sim \mathbf{is}_K(k_j) | j \in \{1..m\} \setminus \{i\} \} | i \in \{1..m\} \}$

The K_i are all distinct \triangle

Example 34 . Container Components: We continue Example 19 on Page 17.

38 When we apply `obs_component_sorts_C` to any container $c:C$ we obtain
 a a type clause stating the sorts of the various components, $ck:CK$, of a container,
 b a union type clause over these component sorts, and
 c the component observer function signature.

type

38a $CK1, CK2, \dots, CKn$
 38b $CKS = (CK1 | CK2 | \dots | CKn)\text{-set}$

value

38c **obs_comp_CKS**: $C \rightarrow CKS$ \square

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

1.3.6 Materials

We refer to Sect. 1.3.1 on Page 12 for a first coverage of the concept of materials. Continuous endurants (i.e., **materials**) are entities, m , which satisfy:

- $\mathbf{is_material}(m) \equiv \mathbf{is_endurant}(m) \wedge \mathbf{is_continuous}(m)$

Example 35 . Parts and Materials: We observe materials as associated with atomic parts: Thus liquid or gaseous materials are observed in pipeline units \square

We shall in this paper not cover the case of parts being immersed in materials²³. We assume, without loss of generality, that only atomic parts may contain materials. Let $p:P$ be some atomic part.

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

- $\mathbf{has_materials}(p)$

yields **true** if the atomic part $p:P$ potentially may contain materials otherwise false \diamond

²³ Most such cases have the material play a minor, an abstract rôle with respect to the immersed parts. That is, we presently leave it to hydro- and aerodynamics to domain analyse those cases.

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

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

- *observe_material_sorts(e)*

yields the material sorts and material sort observers domain description text according to the following schema: whether or not part p actually contains materials:

7. observe_material_sorts schema

Narration:

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

Formalisation:

type

- [s] M_1, M_2, \dots, M_n
- [s] $M = M_1 \mid M_2 \mid \dots \mid M_n$
- [s] $MS = M\text{-set}$

value

- [o] **obs_mat** $_M$: $P \rightarrow M$ $[1 \leq i \leq n]$
- [o] **materials**: $P \rightarrow MS$
- [i] **is** $_M$: $M \rightarrow \text{Bool}$ $[1 \leq i \leq n]$

proof obligation [Disjointness of Material Sorts]

- [p] $\forall m_i:M \cdot \bigwedge \{ \text{is}_M(m_i) \equiv \bigwedge \{ \sim \text{is}_M(m_j) \mid j \in \{1..m\} \setminus \{i\} \} \mid i \in \{1..m\} \}$

The M_i are all distinct \triangle

Example 36 . Pipeline Material: We continue Example 20 on Page 18 and Example 25 on Page 21.

39 When we apply **obs_material_sorts** $_U$ to any unit $u:U$ we obtain

- a a type clause stating the material sort LoG for some further undefined liquid or gaseous material,
- and
- b a material observer function signature.

type

39a **LoG**

value

39b **obs_mat** $_{LoG}$: $U \rightarrow LoG$

has_materials(u) is a prerequisite for **obs_mat** $_{LoG}$ (u) \square

Materials-related Part Attributes

It seems that the “interplay” between parts and materials is an area where domain analysis in the sense of this compendium is relevant.

Example 37 . Pipeline Material Flow: We continue Examples 20, 25 and 36. Let us postulate a[n attribute] sort Flow. We now wish to examine the flow of liquid (or gaseous) material in pipeline units. We use two types

40 **type** F, L.

Productive flow, F , and wasteful leak, L , is measured, for example, in terms of volume of material per second. We then postulate the following unit attributes “measured” at the point of in- or out-flow or in the interior of a unit.

- 41 current flow of material into a unit input connector,
- 42 maximum flow of material into a unit input connector while maintaining laminar flow,
- 43 current flow of material out of a unit output connector,
- 44 maximum flow of material out of a unit output connector while maintaining laminar flow,
- 45 current leak of material at a unit input connector,
- 46 maximum guaranteed leak of material at a unit input connector,
- 47 current leak of material at a unit input connector,
- 48 maximum guaranteed leak of material at a unit input connector,
- 49 current leak of material from “within” a unit, and
- 50 maximum guaranteed leak of material from “within” a unit.

type

40 F, L

value

41 $\text{attr_cur_iF}: U \rightarrow UI \rightarrow F$

42 $\text{attr_max_iF}: U \rightarrow UI \rightarrow F$

43 $\text{attr_cur_oF}: U \rightarrow UI \rightarrow F$

44 $\text{attr_max_oF}: U \rightarrow UI \rightarrow F$

45 $\text{attr_cur_iL}: U \rightarrow UI \rightarrow L$

46 $\text{attr_max_iL}: U \rightarrow UI \rightarrow L$

47 $\text{attr_cur_oL}: U \rightarrow UI \rightarrow L$

48 $\text{attr_max_oL}: U \rightarrow UI \rightarrow L$

49 $\text{attr_cur_L}: U \rightarrow L$

50 $\text{attr_max_L}: U \rightarrow L$

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 may be considered either reactive or biddable attributes \square

Laws of Material 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 modeling. That is in contrast to incorporating such notions of flows and leaks in requirements modeling where one has to show implementability. Modeling flows and leaks is important to the modeling of materials-based domains.

Example 38 . Pipelines: Intra Unit Flow and Leak Law:

- 51 For every unit of a pipeline system, except the well and the sink units, the following law apply.
- 52 The flows into a unit equal
 - a the leak at the inputs
 - b plus the leak within the unit
 - c plus the flows out of the unit
 - d plus the leaks at the outputs.

axiom [Well-formedness of Pipeline Systems, PLS (1)]

51 $\forall \text{pls:PLS}, b:B \setminus \text{We} \setminus \text{Si}, u:U \cdot$

51 $b \in \text{obs_part_Bs}(\text{pls}) \wedge u = \text{obs_part_U}(b) \Rightarrow$

51 **let** $(\text{iuis}, \text{ouis}) = \text{obs_mereo_U}(u)$ **in**

52 $\text{sum_cur_iF}(u)(\text{iuis}) =$

52a $\text{sum_cur_iL}(u)(\text{iuis})$

52b $\oplus \text{attr_cur_L}(u)$

52c $\oplus \text{sum_cur_oF}(u)(\text{ouis})$

52d $\oplus \text{sum_cur_oL}(u)(\text{ouis})$

51 **end**

- 53 The sum_cur_iF (cf. Item 52) sums current input flows over all input connectors.

- 54 The sum_cur_iL (cf. Item 52a) sums current input leaks over all input connectors.
 55 The sum_cur_oF (cf. Item 52c) sums current output flows over all output connectors.
 56 The sum_cur_oL (cf. Item 52d) sums current output leaks over all output connectors.

```

53  $\text{sum\_cur\_iF}: U \rightarrow \text{UI-set} \rightarrow F$ 
53  $\text{sum\_cur\_iF}(u)(iuis) \equiv \oplus \{ \text{attr\_cur\_iF}(u)(ui) \mid ui:U \bullet ui \in iuis \}$ 
54  $\text{sum\_cur\_iL}: U \rightarrow \text{UI-set} \rightarrow L$ 
54  $\text{sum\_cur\_iL}(u)(iuis) \equiv \oplus \{ \text{attr\_cur\_iL}(u)(ui) \mid ui:U \bullet ui \in iuis \}$ 
55  $\text{sum\_cur\_oF}: U \rightarrow \text{UI-set} \rightarrow F$ 
55  $\text{sum\_cur\_oF}(u)(ouis) \equiv \oplus \{ \text{attr\_cur\_oF}(u)(ui) \mid ui:U \bullet ui \in ouis \}$ 
56  $\text{sum\_cur\_oL}: U \rightarrow \text{UI-set} \rightarrow L$ 
56  $\text{sum\_cur\_oL}(u)(ouis) \equiv \oplus \{ \text{attr\_cur\_oL}(u)(ui) \mid ui:U \bullet ui \in ouis \}$ 
 $\oplus: (F|L) \times (F|L) \rightarrow F$ 

```

where \oplus is both an infix and a distributed-fix function which adds flows and or leaks \square

Example 39 . Pipelines: Inter Unit Flow and Leak Law:

- 57 For every pair of connected units of a pipeline system the following law apply:
 a the flow out of a unit directed at another unit minus the leak at that output connector
 b equals the flow into that other unit at the connector from the given unit plus the leak at that connector.

axiom [Well-formedness of Pipeline Systems, PLS (2)]

```

57  $\forall \text{pls:PLS}, b, b': B, u, u': U \bullet$ 
57  $\{b, b'\} \subseteq \text{obs\_part\_Bs}(\text{pls}) \wedge b \neq b' \wedge u' = \text{obs\_part\_U}(b')$ 
57  $\wedge \text{let } (iuis, ouis) = \text{obs\_mereo\_U}(u), (iuis', ouis') = \text{obs\_mereo\_U}(u'),$ 
57  $ui = \text{uid\_U}(u), ui' = \text{uid\_U}(u') \text{ in}$ 
57  $ui \in iuis \wedge ui' \in ouis' \Rightarrow$ 
57a  $\text{attr\_cur\_oF}(u')(ui') - \text{attr\_leak\_oF}(u')(ui')$ 
57b  $= \text{attr\_cur\_iF}(u)(ui) + \text{attr\_leak\_iF}(u)(ui)$ 
57 end
57 comment:  $b'$  precedes  $b$   $\square$ 

```

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.

1.3.7 “No Junk, No Confusion”

Domain descriptions are, as we have already shown, formulated, both informally and formally, by means of abstract types, that is, by sorts for which no concrete models are usually given. Sorts are made to denote possibly empty, possibly infinite, rarely singleton, sets of entities on the basis of the qualities defined for these sorts, whether external or internal. By **junk** we shall understand that the domain description unintentionally denotes undesired entities. By **confusion** we shall understand that the domain description unintentionally have two or more identifications of the same entity or type. The question is *can we formulate a [formal] domain description such that it does not denote junk or confusion?* The short answer to this is no! So, since one naturally wishes “no junk, no confusion” what does one do? The answer to that is *one proceeds with great care!* To avoid junk we have stated a number of sort well-formedness axioms, for example:²⁴

- Page 20 for *wf* links and hubs,
- Page 21 for *wf* road net mereologies,
- Page 21 for *wf* pipeline mereologies,
- Page 25 for *wf* hub states,
- Page 30 for *wf* pipeline systems,
- Page 31 for *wf* pipeline systems,

²⁴ Let *wf* abbreviate *well-formed*.

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

- Page 15 for *Disjointness of Part Sorts*,
- Page 24 for *Disjointness of Attribute Types* and
- Page 29 for *Disjointness of Material Sorts*.

1.3.8 Discussion of Endurants

In Sect. 1.3.1 on Page 14 a “depth-first” search for part sorts was hinted at, but only in the sequence of examples, as given. That sequence of examples essentially expressed that we discover domains epistemologically²⁵ but understand them ontologically.²⁶ The Danish philosopher Søren Kierkegaard (1813–1855) expressed it this way: *Life is lived forwards, but is understood backwards*. The presentation of the of the **domain analysis prompts** and the **domain description prompts** results in domain descriptions which are ontological. The “depth-first” search recognizes the epistemological nature of bringing about understanding. This “depth-first” search that ends with the analysis of atomic part sorts can be guided, i.e., hastened (shortened), by postulating composite sorts that “correspond” to vernacular nouns: everyday nouns that stand for classes of endurants.

1.4 Perdurants

We shall not present a set of **domain analysis prompts** and a set of **domain description prompts** leading to description language, i.e., RSL texts describing perdurant entities. The reason for giving this albeit cursory overview of perdurants is that we can justify our detailed study of endurants, their part and sub parts, their unique identifiers, mereology and attributes. This justification is manifested (i) in expressing the types of signatures, (ii) in basing behaviours on parts, (iii) in basing the for need for CSP-oriented inter-behaviour communications on shared part attributes, (iv) in indexing behaviours as are parts, i.e., on unique identifiers, and (v) in directing inter-behaviour communications across channel arrays indexed as per the mereology of the part behaviours. These are all notions related to endurants and are now justified by their use in describing perdurants. Perdurants can perhaps best be explained in terms of a notion of state and a notion of time. We shall, in this paper, not detail notions of time, but refer to [110, 86, 64, 175].

1.4.1 States

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

Example 40 . States: A road hub can be a state, cf. Hub State, $H\Sigma$, Example 29 on Page 24. A road net can be a state – since its hubs can be. Container stowage areas, CSA, Example 19 on Page 17, of container vessels and container terminal ports can be states as containers can be removed from and put on top of container stacks. Pipeline pipes can be states as they potentially carry material. Conveyor belts can be states as they may carry components ◻

1.4.2 Actions, Events and Behaviours

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

²⁵ **Epistemology:** the theory of knowledge, especially with regard to its methods, validity, and scope. Epistemology is the investigation of what distinguishes justified belief from opinion.

²⁶ **Ontology:** the branch of metaphysics dealing with the nature of being.

Time Considerations

We shall, without loss of generality, assume that actions and events are atomic and that behaviours are composite. Atomic perdurants may “occur” during some time interval, but we omit consideration of and concern for what actually goes on during such an interval. Composite perdurants can be analysed into “constituent” actions, events and “sub-behaviours”. We shall also omit consideration of temporal properties of behaviours. Instead we shall refer to two seminal monographs: *Specifying Systems* [124, Leslie Lamport] and *Duration Calculus: A Formal Approach to Real-Time Systems* [187, Zhou ChaoChen and Michael Reichhardt Hansen] (and [26, Chapter 15]). For a seminal book on “time in computing” we refer to the eclectic [91, Mandrioli et al., 2012]. And for seminal book on time at the epistemology level we refer to [175, J. van Benthem, 1991].

Actors

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

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

Example 41 . Actors: We refer to the road transport and the pipeline systems examples of earlier. The fleet, each vehicle and the road management of the *Transportation System* of Examples 17 on Page 15 and 32 on Page 27 can be considered actors; so can the net and its links and hubs. The pipeline monitor and each pipeline unit of the *Pipeline System*, Example 20 on Page 18 and Examples 20 on Page 18 and 25 on Page 21 will be considered actors □

Parts, Attributes and Behaviours

Example 41 focused on what shall soon become a major relation within domains: that of parts being also considered actors, or more specifically, being also considered to be behaviours.

Example 42 . Parts, Attributes and Behaviours: Consider the term ‘train’²⁷. It has several possible “meanings”. (i) the train as a part, viz., as standing on a train station platform; (ii) the train as listed in a timetable (an attribute of a transport system part), (iii) the train as a behaviour: speeding down the rail track □

1.4.3 Discrete Actions

Definition 13 Discrete Action: By a **discrete action** [181, Wilson and Shpall] we shall understand a foreseeable thing which deliberately potentially changes a well-formed state, in one step, usually into another, still well-formed state, and for which an actor can be made responsible ◉

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

Example 43 . Road Net Actions: Examples of *Road Net* actions initiated by the net actor are: insertion of hubs, insertion of links, removal of hubs, removal of links, setting of hub states. Examples of *Traffic System* actions initiated by vehicle actors are: moving a vehicle along a link, stopping a vehicle, starting a vehicle, moving a vehicle from a link to a hub and moving a vehicle from a hub to a link □

²⁷ This example is due to Paul Lindgreen, a Danish computer scientist. It dates from the late 1970s.

1.4.4 Discrete Events

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

Events can be characterised by a pair of (before and after) states, a predicate over these and, optionally, a time or time interval. The notion of event continues to puzzle philosophers [82, 155, 134, 80, 103, 12, 122, 70, 147, 69]. We note, in particular, [80, 12, 122].

Example 44 . Road Net and Road Traffic Events: Some road net events are: “disappearance” of a hub or a link, failure of a hub state to change properly when so requested, and occurrence of a hub state leading traffic into “wrong-way” links. Some road traffic events are: the crashing of one or more vehicles (whatever ‘crashing’ means), a car moving in the wrong direction of a one-way link, and the clogging of a hub with too many vehicles ☐

1.4.5 Discrete Behaviours

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

Example 45 . Behaviours: (i) Road Nets: A sequence of hub and link insertions and removals, link disappearances, etc. (ii) Road Traffic: A sequence of movements of vehicles along links, entering, circling and leaving hubs, crashing of vehicles, etc. (iii) Pipelines: A sequence of pipeline pump and valve openings and closings, and failures to do so (events), etc. (iv) Container Vessels and Ports: Concurrent sequences of movements (by cranes) of containers from vessel to port (unloading), with sequences of movements (by cranes) from port to vessel (loading), with dropping of containers by cranes, etcetera ☐

Channels and Communication

Behaviours sometimes synchronise and usually communicate. We use the CSP [111] notation (adopted by RSL) to introduce and model behaviour communication. Communication is abstracted as the sending ($ch!m$) and receipt ($ch?$) of messages, $m:M$, over channels, ch .

type M
channel $ch:M$

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

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

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

Relations Between Attribute Sharing and Channels

We shall now interpret the syntactic notion of attribute sharing with the semantic notion of channels. This is in line with the above-hinted interpretation of parts with behaviours, and, as we shall soon see, part attributes with behaviour states. Thus, for every pair of parts, $p_{ik}:P_i$ and $p_{j\ell}:P_j$, of distinct sorts, P_i and P_j which share attribute values in A we are going to associate a channel. If there is only one pair of parts, $p_{ik}:P_i$ and $p_{j\ell}:P_j$, of these sorts, then we associate just a simple channel, say $attr_A_ch_{P_i,P_j}$, with the shared attribute.

channel $attr_A_ch_{P_i,P_j}:A$.

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

channel {attr_A_ch[$\{\pi_i, \pi_j\}$]:A | $\pi_i = uid_P_i(p_i) \wedge \pi_j \in uids$ }.

The above can be extended in two ways: From channel matrices to channel tensors, etc., hence the term channel ‘array’. And from simple shared attributes to “embedded sharing”.

We say that P and Q enjoy **embedded attribute sharing** when the following is the case: Part sort P has attribute type A, and part sort Q, different from P, has attribute type B where B is defined in terms of A. For cases where P and Q enjoy embedded attribute sharing the mereology of parts $p:P$ will include $uid_Q(q)$ and the mereology of parts $q:Q$ will include $uid_P(p)$.

Example 46 . Bus System Channels: We extend Examples 17 on Page 15 and 32 on Page 27. We consider the fleet and the vehicles to be behaviours.

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

59 the fleet and

60 the vehicles.

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

value

58 $\delta:\Delta,$

59 $f:F = \mathbf{obs_part_F}(\delta),$

60 $vs:V\text{-set} = \mathbf{obs_part_Vs}(\mathbf{obs_part_VC}(\mathbf{obs_part_F}(\delta)))$

channel

61 {attr_BT_ch[$\{uid_F(f), uid_V(v)\}$] | $v:V \cdot v \in vs$ }:BT \square

1.4.6 Continuous Behaviours

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

Example 47 . Flow in Pipelines: We refer to Examples 25, 36, 37, 38 and 39. Let us assume that oil is the (only) material of the pipeline units. Let us assume that there is a sufficient volume of oil in the pipeline units leading up to a pump. Let us assume that the pipeline units leading from the pump (especially valves and pumps) are all open for oil flow. Whether or not that oil is flowing, if the pump is pumping (with a sufficient head²⁸) then there will be oil flowing from the pump outlet into adjacent pipeline units \square

To describe the flow of material (say in pipelines) requires knowledge about a number of material attributes — not all of which have been covered in the above-mentioned examples. To express flows one resorts to the mathematics of fluid-dynamics using such second order differential equations as first derived by Bernoulli (1700–1782) and Navier–Stokes (1785–1836 and 1819–1903). There is, as yet, no notation that can serve to integrate formal descriptions (like those of Alloy, B, The B Method, RSL, VDM or Z) with first, let alone second order differential equations. But some progress has been made [129, 186] since [179].

1.4.7 Attribute Value Access

We refer to paragraph “Access to Attribute Values” in Section 1.3.4 Page 26. We distinguish between four kinds of attributes: the static attributes which are those whose values are fixed, i.e., does not change, the programmable attributes or biddable attributes, i.e., the controllable attributes, which are those dynamic values are exclusively set by part processes, and the remaining dynamic attributes which here, technically speaking, are seen as separate external processes. The event attributes are those external attributes whose value occur for an instant of time.

²⁸ The **pump head** is the linear vertical measurement of the maximum height a specific pump can deliver a liquid to the pump outlet.

Access to Static Attribute Values

The **static attributes** can be “copied”, $\text{attr_A}(p)$, and retain their values.

Access to External Attribute Values

By the **external attributes**, to repeat, we shall understand the inert, the autonomous and the reactive attributes ☺

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

63 Each external attribute, A , is seen as an individual behaviour, each “accessible” by means of unique channel, attr_A_ch .

64 External attribute values are then the value, a , of, i.e., accessed by the input, $\text{attr_A_ch}?$.

62 **value** $\xi A = \{\eta A \mid A \text{ is any external attribute name}\}$

63 **channel** $\{\text{attr_A_ch}:A \mid \eta A \in \xi A\}$

64 **value** $a = \text{attr_A_ch}?$

We shall omit the η prefix in actual descriptions. The choice of representing external attribute values as CSP processes²⁹ is a technical one.

Access to Controllable Attribute Values

The controllable attributes are treated as function arguments. This is a technical choice. It is motivated as follows. We find that these values are a function of other part attribute values, including at least one controllable attribute value, and that the values are set (i.e., updated) by part behaviours. That is, to each part, whether atomic or composite, we associate a behaviour. That behaviour is (to be) described as we describe functions. These functions (normally) “go on forever”. Therefore these functions are described basically by a “tail” recursive definition:

value $f: \text{Arg} \rightarrow \text{Arg}; f(a) \equiv (\dots \text{let } a' = \mathcal{F}(\dots)(a) \text{ in } f(a') \text{ end})$

where \mathcal{F} is some expression based on values defined within the function definition body of f and on f ’s “input” argument a , and where a can be seen as a controllable attribute.

Access to Event Values

We refer to Sect. 1.3.4 on Page 26. Event values reflect a stage change in a part behaviour. We therefore model events as messages communicated over a channel, attr_A_ch , that is, $\text{attr_A_ch}!a$, where A is the event attribute, i.e., message type. Thus fulfillment of $\text{attr_A_ch}?$ expresses both that the event has taken place and its value, if relevant. Example 52 on Page 42 illustrates the concept of event attributes and event values.

1.4.8 Perdurant Signatures and Definitions

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

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

Definition 17 Function Type Expression: By a **function type expression** we shall understand a pair of type expressions, separated by a function type constructor either \rightarrow (total) or \leadsto (partial) function ☺

The type expressions are part sort or type, or material sort or type, or component sort or type, or attribute type names, but may, occasionally be expressions over respective type names involving **-set**, \times , $*$, \rightarrow and $|$ type constructors.

²⁹ — not to be confused with domain behaviours

1.4.9 Action Signatures and Definitions

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

action: $VAL \rightarrow \Sigma \xrightarrow{\sim} \Sigma$
 action(v)(σ) **as** σ'
 pre: $\mathcal{P}(v, \sigma)$
 post: $\mathcal{Q}(v, \sigma, \sigma')$

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

Example 48 . Insert Hub Action Formalisation: We formalise aspects of the above-mentioned hub action:

65 Insertion of a hub requires
 66 that no hub exists in the net with the unique identifier of the inserted hub,
 67 and then results in an updated net with that hub.

value

65 insert_H: $H \rightarrow N \xrightarrow{\sim} N$
 65 insert_H(h)(n) **as** n'
 66 **pre:** $\sim \exists h': H \cdot h' \in \text{obs_part_Hs}(\text{obs_part_HS}(n)) \cdot \text{uid_H}(h) = \text{uid_H}(h')$
 67 **post:** $\text{obs_part_Hs}(\text{obs_part_HS}(n')) = \text{obs_part_Hs}(\text{obs_part_HS}(n)) \cup \{h\}$ \square

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

Perdurant (action) analysis thus proceeds as follows: identifying relevant actions, assigning names to these, delineating the “smallest” relevant state³⁰, ascribing signatures to action functions, and determining action pre-conditions and action post-conditions. Of these, ascribing signatures is the most crucial: In the process of determining the action signature one oftentimes discovers that part or component or material attributes have been left (“so far”) “undiscovered”.

Example 48 showed example of a signature with only a part argument. Example 49 shows examples of signatures whose arguments are parts and unique identifiers, or parts, unique identifiers and attribute values.

Example 49 . Some Function Signatures: Inserting a link between two identified hubs in a net:

value insert_L: $L \times (HI \times HI) \rightarrow N \xrightarrow{\sim} N$

Removing a hub and removing a link:

value remove_H: $HI \rightarrow N \xrightarrow{\sim} N$
 remove_L: $LI \rightarrow N \xrightarrow{\sim} N$

Changing a hub state.

value change_H Σ : $HI \times H\Sigma \rightarrow N \xrightarrow{\sim} N$ \square

³⁰ By “smallest” we mean: containing the fewest number of parts. Experience shows that the domain analyser cum describer should strive for identifying the smallest state.

1.4.10 Event Signatures and Definitions

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

value

```

event:  $\Sigma \times \Sigma \xrightarrow{\sim} \mathbf{Bool}$ 
event( $\sigma, \sigma'$ ) as tf
  pre:  $P(\sigma)$ 
  post: tf =  $Q(\sigma, \sigma')$ 

```

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

Example 50 . Link Disappearance Formalisation: We formalise aspects of the above-mentioned link disappearance event:

```

68 The result net is not well-formed.
69 For a link to disappear there must be at least one link in the net;
70 and such a link may disappear such that
71 it together with the resulting net makes up for the “original” net.

```

value

```

68 link_diss_event:  $N \times N' \xrightarrow{\sim} \mathbf{Bool}$ 
68 link_diss_event( $n, n'$ ) as tf
69   pre:  $\mathbf{obs\_part\_Ls}(\mathbf{obs\_part\_LS}(n)) \neq \{\}$ 
70   post: tf =  $\exists l: L \cdot l \in \mathbf{obs\_part\_Ls}(\mathbf{obs\_part\_LS}(n)) \Rightarrow$ 
71          $l \notin \mathbf{obs\_part\_Ls}(\mathbf{obs\_part\_LS}(n'))$ 
71          $\wedge n' \cup \{l\} = \mathbf{obs\_part\_Ls}(\mathbf{obs\_part\_LS}(n))$   $\square$ 

```

1.4.11 Discrete Behaviour Signatures and Definitions

Behaviour Signatures

We shall only cover behaviour signatures when expressed in RSL/CSP [96]. The behaviour functions are now called processes. That a behaviour function is a never-ending function, i.e., a process, is “revealed” in the function signature by the “trailing” **Unit**:

behaviour: $\dots \rightarrow \dots \mathbf{Unit}$

That a process takes no argument is “revealed” by a “leading” **Unit**:

behaviour: $\mathbf{Unit} \rightarrow \dots$

That a process accepts channel, viz.: ch , inputs, including accesses an external attribute A , is “revealed” in the function signature as follows:

behaviour: $\dots \rightarrow \mathbf{in} \ ch \ \dots$, resp. $\mathbf{in} \ attr_A _ch$

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

behaviour: $\dots \rightarrow \mathbf{out} \ ch \ \dots$

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

behaviour: ARG \rightarrow ...

where ARG can be any type expression:

T, T \rightarrow T, T \rightarrow T \rightarrow T, etcetera

where T is any type expression.

Part Behaviours:

We can, without loss of generality, associate with each part a behaviour; parts which share attributes (and are therefore referred to in some parts’ mereology), can communicate (their “sharing”) via channels.

Processes are named, and part process names have indexes, namely the unique part identifier: $\pi:\Pi$. The p be the part and let $part_\pi$ be the name of the process associated with part p . The process named $part_\pi$ shall have the process name $part_\pi$ mean the following. Let $part_\pi(args) \equiv \mathcal{B}$ be the definition of process $part_\pi$. Occurrences of π in the definition body \mathcal{B} shall be considered bound to the π of the process name $part_\pi$. Thus, if the process named $part_i$ has π bound to i both in the process name $part_\pi$ and in the body \mathcal{B} .

The process evolves around a state, or, rather, a set of values: its possibly changing mereology, $mt:MT^{31}$, the possible components and materials of the part, and the attributes of the part. A behaviour signature is therefore:

behaviour $_{\pi:\Pi}$: $me:MT \times sa:SA \rightarrow ca:CA \rightarrow \mathbf{in\ ichns}(ea:EA) \mathbf{in,out\ iochs}(me) \mathbf{Unit}$

where (i) $\pi:\Pi$ is the unique identifier of part p , i.e., $\pi = \mathbf{uid_P}(p)$, (ii) $me:ME$ is the mereology of part p , $me = \mathbf{obs_mereo_P}(p)$, (iii) $sa:SA$ lists the static attribute values of the part, (iv) $ca:CA$ lists the controllable and attribute values of the part, (v) $\mathbf{ichns}(ea:EA)$ refer to the external attribute input channels, and where (vi) $\mathbf{iochs}(me)$ are the input/output channels serving the attributes shared between the part p and the parts designated in its mereology me , cf. Sect. 1.4.7. We focus, for a little while, on the expression of $sa:SA$, $ea:EA$ and $ca:CA$, that is, on the concrete types of SA, EA and CA.

$\mathcal{S}_{\mathcal{A}}(p)$: $sa:SA$ lists the static value types, (svT_1, \dots, svT_s) , where s is the number of static attributes of parts $p:P$.

$\mathcal{E}_{\mathcal{A}}(p)$: $ea:EA$ lists the external attribute value channels of parts $p:P$ in the behaviour signature and as input channels, \mathbf{ichns} , see 9 lines above.

$\mathcal{C}_{\mathcal{A}}(p)$: $ca:CA$ lists the controllable value expression types of parts $p:P$. A **controllable attribute value expression** is an expression involving one or more attribute value expressions of the type of the biddable or programmable attribute \odot

Behaviour Definitions

Let P be a composite sort defined in terms of sub-sorts P_1, P_2, \dots, P_n . The process definition compiled from $p:P$, is composed from a process description, $\mathcal{M}cP_{\mathbf{uid_P}(p)}$, relying on and handling the unique identifier, mereology and attributes of part p operating in parallel with processes p_1, p_2, \dots, p_n where p_1 is compiled from $p_1:P_1$, p_2 is compiled from $p_2:P_2$, ..., and p_n is compiled from $p_n:P_n$. The domain description “compilation” schematic below “formalises” the above.

Process Schema I: Abstract $\mathbf{is_composite}(p)$

value

compile_process: $P \rightarrow \mathbf{RSL_Text}$

compile_process(p) \equiv

$\mathcal{M}P_{\mathbf{uid_P}(p)}(\mathbf{obs_mereo_P}(p), \mathcal{S}_{\mathcal{A}}(p))(\mathcal{C}_{\mathcal{A}}(p))$
 $\parallel \text{compile_process}(\mathbf{obs_part_P}_1(p))$

³¹ For MT see footnote 16 on Page 21.

```

|| compile_process(obs_part_P2(p))
|| ...
|| compile_process(obs_part_Pn(p))

```

The text macros: $\mathcal{S}_{\mathcal{A}}$ and $\mathcal{C}_{\mathcal{A}}$ were informally explained above. Part sorts P_1, P_2, \dots, P_n are obtained from the `observe_part_sorts` prompt, Page 15.

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

Process Schema II: Concrete `is_composite(p)`

```

type
  Qs = Q-set
value
  qs:Q-set = obs_part_Qs(p)
  compile_process: P → RSL-Text
  compile_process(p) ≡
     $\mathcal{M}P_{\mathbf{uid}_P(p)}(\mathbf{obs\_mereo}_P(p), \mathcal{S}_{\mathcal{A}}(p))(\mathcal{C}_{\mathcal{A}}(p))$ 
    || || {compile_process(q) | q:Q•q ∈ qs}

```

Process Schema III: `is_atomic(p)`

```

value
  compile_process: P → RSL-Text
  compile_process(p) ≡
     $\mathcal{M}P_{\mathbf{uid}_P(p)}(\mathbf{obs\_mereo}_P(p), \mathcal{S}_{\mathcal{A}}(p))(\mathcal{C}_{\mathcal{A}}(p))$ 

```

Example 51 . Bus Timetable Coordination: We refer to Examples 17 on Page 15, 18 on Page 16, 32 on Page 27 and 46 on Page 35.

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

73 The fleet process is compiled as per Process Schema II (Page 40).

The definitions of the fleet and vehicle processes are simplified so as to emphasize the master/slave, programmable/inert relations between these processes.

```

type
  Δ, F, VS      [Example 17 on Page 15]
  V, Vs=V-set   [Example 18 on Page 16]
  FI, VI, BT     [Example 32 on Page 27]
value
72  δ:Δ,
72  f:F = obs_part_F(δ),
72  fi:FI = uid_F(f)
72  vs:V-set = obs_part_Vs(obs_part_VS(f))
axiom
72  ∀ v:V•v ∈ vs ⇒ □ attr_BT(f) = attr_BT(v) [Example 32 on Page 27]
value

```

```

73   fleetfi: BT → out attr_BT_ch Unit
73   fleetfi(bt) ≡  $\mathcal{M}F_{fi}(bt) \parallel \parallel \{ \text{vehicle}_{uid_{\mathcal{V}}(v)}() \mid v:V \bullet v \in vs \}$ 

73   vehiclevi: Unit → in attr_BT_ch Unit
73   vehiclevi ≡  $\mathcal{M}V_{vi}(\text{attr\_BT\_ch}) ; \text{vehicle}_{vi}()$ 

```

The fleet process \mathcal{M}_F is a “never-ending” processes:

```

value
 $\mathcal{M}F_{fi}$ : BT → out attr_BT_ch Unit
 $\mathcal{M}F_{fi}(bt) \equiv \text{let } bt' = \mathcal{F}_{fi}(bt) \text{ in } \mathcal{M}F_{fi}(bt') \text{ end}$ 

```

Function \mathcal{F}_{fi} is a simple action. The expression of actual synchronisation and communication between the fleet and the vehicle processes is contained in \mathcal{F}_{fi} .

```

value
 $\mathcal{F}_{fi}$ : bt:BT → out attr_BT_ch BT
 $\mathcal{F}_{fi}(bt) \equiv (\text{let } bt' = f_{fi}(bt)(\dots) \text{ in } bt' \text{ end}) \parallel (\text{attr\_BT\_ch} ! bt ; bt)$ 
 $f_{fi}$ : BT → ... → BT

```

The auxiliary function f_{fi} “embodies” the programmable nature of the timetable attribute \square

Please note a master part’s programmable attribute can be reflected in two ways: as a programmable attribute and as an output channel to the behaviour specification of slave parts. This is illustrated, in Example 51 where the fleet behaviour has programmable attribute BT and output channel attr_BT_ch to vehicle behaviours.

Process Schema IV: Core Process (I)

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

```

 $\mathcal{M}P_{\pi:\Pi}$ : me:MT × sa:SA → ca:CA → in ichns(ea:EA) in,out iochs(me) Unit
 $\mathcal{M}P_{\pi:\Pi}(me,sa)(ca) \equiv \text{let } (me',ca') = \mathcal{F}_{\pi:\Pi}(me,sa)(ca) \text{ in } \mathcal{M}P_{\pi:\Pi}(me',sa)(ca') \text{ end}$ 

 $\mathcal{F}_{\pi:\Pi}$ : me:MT × sa:SA → CA → in ichns(ea:EA) in,out iochs(me) → MT × CA

```

\mathcal{F}_{π} potentially communicates with all those part processes (of the whole domain) with which it shares attributes, that is, has connectors. \mathcal{F}_{π} is expected to contain input/output clauses referencing the channels of the in ... out ... part of their signatures. These clauses enable the sharing of attributes. \mathcal{F}_{π} also contains expressions, attr_A_ch ?, to external attributes.

We present a rough sketch of \mathcal{F}_{π} . The \mathcal{F}_{π} action non-deterministically internal choice chooses between

- either [1,2,3,4]
 - ∞ [1] accepting input from
 - ∞ [4] a suitable (“offering”) part process,
 - ∞ [2] optionally offering a reply, and
 - ∞ [3] finally delivering an updated state;
- or [5,6,7,8]
 - ∞ [5] finding a suitable “order” (val)
 - ∞ [8] to a suitable (“inquiring”) behaviour (π'),
 - ∞ [6] offering that value (on channel ch[π'])
 - ∞ [7] and then delivering an updated state;
- or [9] doing own work resulting in an updated state.

Process Schema V: Core Process (II)

```

value
 $\mathcal{F}_{\pi}$ : me:MT × sa:SA → ca:CA → in ichns(ea:EA) in,out iochs(me) MT × CA
 $\mathcal{F}_{\pi}(me,sa)(ca) \equiv$ 
[1]  $\square \{ \text{let } val = \text{ch}[\pi'] ? \text{ in}$ 

```

```

[2]      ( ch[ $\pi'$ ] ! in_reply(val)(me,sa)(ca) [] skip ) ;
[3]      in_update(val)(me,sa)(ca) end
[4]      |  $\pi'$ :  $\Pi \cdot \pi' \in \mathcal{E}(\pi, me)$  }
[5]      [] [] { let val = await_reply( $\pi'$ )(me,sa)(ca) in
[6]      ch[ $\pi'$ ] ! val ;
[7]      out_update(val)(me,sa)(ca) end
[8]      |  $\pi'$ :  $\Pi \cdot \pi' \in \mathcal{E}(\pi, me)$  }
[9]      [] (me,own_work(sa)(ca))

channels ch[ $\pi'$ ] are defined in in ichns(ea:EA) in,out iochs(me)

in_reply: VAL  $\rightarrow$  SA $\times$ EA  $\rightarrow$  CA  $\rightarrow$  in ichns(ea:EA) in,out iochs(me) VAL
in_update: VAL  $\rightarrow$  MT $\times$ SA  $\rightarrow$  CA  $\rightarrow$  in,out iochs(me) MT $\times$ CA
await_reply:  $\Pi \rightarrow$  MT $\times$ SA  $\rightarrow$  CA  $\rightarrow$  in,out iochs(me) VAL
out_update: VAL  $\rightarrow$  MT $\times$ SA  $\rightarrow$  CA  $\rightarrow$  in,out iochs(me) MT $\times$ CA
own_work: SA $\times$ EA  $\rightarrow$  CA  $\rightarrow$  in,out iochs(me) CA

```

We leave these auxiliary functions and VAL undefined.

Example 52. Tollgates: Part and Behaviour: Our example is disconnected from that of a larger example of road pricing. Figure 1.3 abstracts essential features of a tollgate.

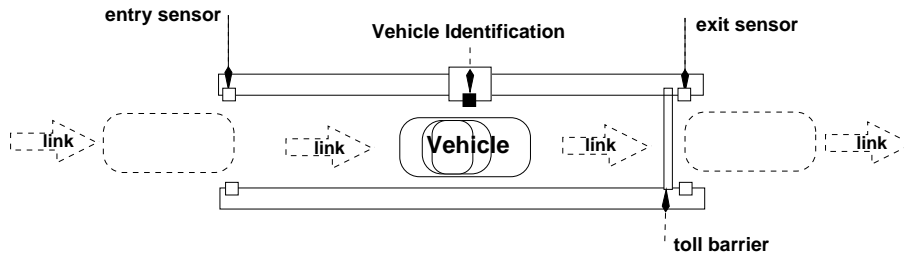


Fig. 1.3. A tollgate

74 A tollgate is a composite part. It consists of

75 an entry sensor ⁽³²⁾, a vehicle identity sensor ⁽³³⁾, a barrier ⁽³⁴⁾, and an exit sensor ⁽³⁵⁾.

76 The sensors function as follows:

- a When a vehicle first starts passing the entry sensor then it sends an appropriate (event) message to the tollgate.
- b When a vehicle's identity is recognised by the identity sensor then it sends an appropriate (event) message to the tollgate.
- c When a vehicle ends passing the exit sensor then it sends an appropriate (event) link message to the tollgate.

77 We therefore model these sensors as shared dynamic event attributes.

- a For the sensors these are master attributes.
- b For the tollgate they are slave attributes.
- c In all three cases they are therefore modeled as channels.

78 A vehicle passing the gate

³² ES

³³ IS

³⁴ B

³⁵ XS

- a first “triggers” the entry sensor (“Enter”),
- b which results in the lowering (“Lower”) of the barrier,
- c then the vehicle identity sensor (“vi:VI”),
- d with the tollgate “mysteriously”³⁶ handling that identity, and, simultaneously
- e raising (“Raise”) the barrier, and
- f finally the output sensor (“Exit”) is triggered as the vehicle leaves the tollgate,
- g and the barrier is lowered.

79 whereupon the tollgate resumes being a tollgate.

80 TGI is the type unique tollgate identifiers.

Instead of one tollgate we may think of a number of tollgates: Each with their unique identifier — together with a finite set of two or more such identifiers, **tgis:TGI-set**.

```

type
74   TG
75   ES, IS, B, XS
78a  En = {"Enter"}
78b  Ba = {"Lower", "Raise"}
78c  Id = VI
78e  Ex = {"Exit"}
80   TGI

value
75   obs_part_ES: TG → ES
75   obs_part_IS: TG → IS
75   obs_part_B: TG → B
75   obs_part_XS: TG → XS
80   uid_TGI: TG → TGI
78a  attr_Enter: TG|ES → {"Enter"}
78c  attr_Identity: TG|IS → VI
78e  attr_Exit: TG|XS → {"Exit"}

channel
78   {attr_En_ch[tgi]|tgi:TGI•tgi∈tgis}: En
78   {attr_Id_ch[tgi]|tgi:TGI•tgi∈tgis}: VI
78   {attr_Ba_ch[tgi]|tgi:TGI•tgi∈tgis}: BA
78   {attr_Ex_ch[tgi]|tgi:TGI•tgi∈tgis}: Ex

value
78   gatetgi:TGI: Unit →
78   in attr_En_ch[tgi], attr_Id_ch[tgi], attr_Ex_ch[tgi]
78   out attr_Ba_ch[tgi] Unit
78   gatetgi:TGI() ≡
78a   attr_En_ch[tgi] ? ;
78b   attr_Ba_ch[tgi] ! "Lower" ;
78c   let vi = attr_Id_ch[tgi] ? in
78d   ( handle(vi) ||
78e   attr_Ba_ch[tgi] ! "Raise" ) ;
78f   attr_Ex_ch[tgi] ? ;
78g   attr_Ba[tgi] ! "Lower" ;
79   gatetgi:TGI() end

```

The enter, identity and exit events are slave attributes of the tollgate part and master attributes of respectively the entry sensor, the vehicle identity sensor, and the exit sensor sub-parts. We do not define the behaviours of these sub-parts. We only assume that they each issue appropriate **attr_A_ch ! output** messages where A is either Enter, Identity, or Exit and where event values en:Enter and ex:Exit are ignored □

1.4.12 Concurrency: Communication and Synchronisation

Process Schemas I, II and IV (Pages 39, 40 and 41), reveal that two or more parts, which temporally coexist (i.e., at the same time), imply a notion of concurrency. Process Schema IV, through the RSL/CSP language expressions **ch!v** and **ch?**, indicates the notions of communication and synchronisation. Other than this we shall not cover these crucial notion related to parallelism.

1.4.13 Summary and Discussion of Perdurants

The most significant contribution of Sect. 1.4 has been to show that for every domain description there exists a normal form behaviour — here expressed in terms of a CSP process expression.

³⁶ ... that is, passes vi on to the road pricing monitor — where we omit showing relevant channels.

Summary

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

Discussion

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

1.5 Closing

In Sect. 1.1.1 we emphasised that in order to develop software the designers *must have a reasonable grasp of the “underlying” domain*. That means that when we design software, its requirements, to us, must be based on such a “grasp”, that is, that the domain description must cover that “underlying” domain. We are not claiming that the domain descriptions (for software development) must cover more than the “underlying” domain. But what that “underlying” domain then is, is an open question which we do not speculate on in this paper. Domain descriptions are not “cast in stone!” It is to be expected that domains are researched and their descriptions are developed as research projects — typically in universities. It is also to be expected that several domain descriptions coexist “simultaneously”, that they may converge, that some whither away, are rejected, and that new descriptions are developed “on top of”, that is, on the basis of existing ones, which they replace, descriptions that enlarge on, or restrict previous descriptions. It is finally to be expected that when requirements are to be “derived” from a domain description, see, for example, [54], that the requirements cum domain engineers redevelop a projected domain description having some existing domain descriptions “at hand”.

1.5.1 Analysis & Description Calculi for Other Domains

The analysis and description calculus of this paper appears suitable for manifest domains. For other domains other calculi may be necessary. There is the introvert, composite domain(s) of systems software: operating systems, compilers, database management systems, Internet-related software, etcetera. The classical computer science and software engineering disciplines related to these components of systems software appears to have provided the necessary analysis and description “calculi.” There is the domain of financial systems software accounting & bookkeeping, banking systems, insurance, financial instruments handling (stocks, etc.), etcetera. Etcetera. For each domain characterisable by a distinct set of analysis & description calculus prompts such calculi must be identified.

1.5.2 On Domain Description Languages

We have in this paper expressed the domain descriptions in the RAISE [97] specification language RSL [96]. With what is thought of as minor changes, one can reformulate these domain description texts in either of Alloy [114] or The B-Method [1] or VDM [61, 62, 89] or Z [183]. One could also express domain descriptions algebraically, for example in CafeOBJ [94, 93]. The analysis and the description prompts remain the same. The description prompts now lead to Alloy, B-Method, VDM, Z or CafeOBJ texts. We did not go into much detail with respect to perdurants. For all the very many domain descriptions, covered elsewhere, RSL (with its CSP sub-language) suffices. It is favoured here because of its integrated CSP sub-language which both facilitates the ‘compilation’ of part descriptions into “the dynamics” of parts in terms of CSP processes, and the modeling of external attributes in terms of CSP process input channels. But there are cases, not documented in this paper, where, [60], we have conjoined our RSL domain descriptions with descriptions in Petri Nets [156] or MSC [113] (Message Sequence Charts) or StateCharts [105].

1.5.3 Comparison to Other Work

Background: The Triptych Domain Ontology

We shall now compare the approach of this paper to a number of techniques and tools that are somehow related — if only by the term ‘domain’ ! Common to all the “other” approaches is that none of them presents a prompt calculus that help the domain analyser elicit a, or the, domain description. Figure 1.1 on Page 12 shows the tree-like structuring of what modern day AI researchers cum ontologists would call *an upper ontology*.

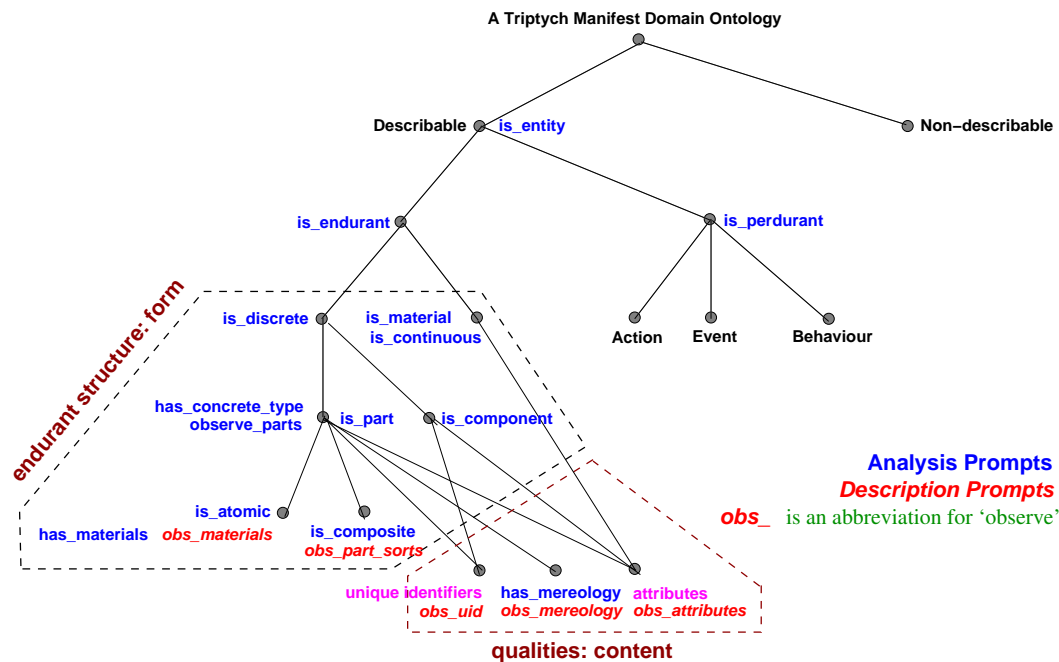


Fig. 1.4. The Upper Ontology of Triptych Manifest Domains

General

Two related approaches to structuring domain understanding will be reviewed.

0: Ontology Science & Engineering:

Ontologies are “*formal representations of a set of concepts within a domain and the relationships between those concepts*” — expressed usually in some logic. Ontology engineering [14] construct ontologies. Ontology science appears to mainly study structures of ontologies, especially so-called upper ontology structures, and these studies “waver” between philosophy and information science³⁷. Internet published ontologies usually consists of thousands of logical expressions. These are represented in some, for example, low-level mechanisable form so that they can be interchanged between ontology research groups and processed by various tools. There does not seem to be a concern for “deriving” such ontologies into requirements for software. Usually ontology presentations either start with the presentation of, or makes reference to its reliance on, an upper ontology. The term ‘ontology’ has been much used in connection with automating the design of various aspects WWW applications [178]. Description Logic [8] has been proposed as a language for the Semantic Web [9].

³⁷ We take the liberty of regarding information science as part of computer science, cf. Page 5.

The interplay between endurants and perdurants is studied in [17]. That study investigates axiom systems for two ontologies. One for endurants (SPAN), another for perdurants (SNAP). No examples of descriptions of specific domains are, however, given, and thus no specific techniques nor tools are given, method components which could help the engineer in constructing specific domain descriptions. [17] is therefore only relevant to the current paper insofar as it justifies our emphasis on endurant versus perdurant entities. The interplay between endurant and perdurant entities and their qualities is studied in [119]. In our study the term quality is made specific and covers the ideas of external and internal qualities, cf. Sect. 1.3.1 on Page 19. External qualities focus on whether endurant or perdurant, whether part, component or material, whether action, event or behaviour, whether atomic or composite part, etcetera. Internal qualities focus on unique identifiers (of parts), the mereology (of parts), and the attributes (of parts, components and materials), that is, of endurants. In [119] the relationship between universals (types), particulars (values of types) and qualities is not “restricted” as in the TripTych domain analysis, but is axiomatically interwoven in an almost “recursive” manner. Values [of types (‘quantities’ [of ‘qualities’])] are, for example, seen as sub-ordinated types; this is an ontological distinction that we do not make. The concern of [119] is also the relations between qualities and both endurant and perdurant entities, where we have yet to focus on “qualities”, other than signatures, of perdurants. [119] investigates the quality/quantity issue wrt. endurance/perdurance and poses the questions: [b] are non-persisting quality instances enduring, perduring or neither? and [c] are persisting quality instances enduring, perduring or neither? and arrives, after some analysis of the endurance/perdurance concepts, at the answers: [b'] non-persisting quality instances are neither enduring nor perduring particulars (i.e., entities), and [c'] persisting quality instances are enduring particulars. Answer [b'] justifies our separating enduring and perduring entities into two disjoint, but jointly “exhaustive” ontologies. The more general study of [119] is therefore really not relevant to our prompt calculi, in which we do not speculate on more abstract, conceptual qualities, but settle on external endurant qualities, on the unique identifier, mereology and attribute qualities of endurants, and the simple relations between endurants and perdurants, specifically in the relations between signatures of actions, events and behaviours and the endurant sorts, and especially the relation between parts and behaviours as outlined in Sect. 1.4.11. That is, the TripTych approach to ontology, i.e., its domain concept, is not only model-theoretic, but, we risk to say, radically different. The concerns of TripTych domain science & engineering is based on that of algorithmic engineering. The domains to which we are applying our analysis & description tools and techniques are spatio-temporal, that is, can be observed, physically; this is in contrast to such conceptual domains as various branches of mathematics, physics, biology, etcetera. Domain science & engineering is not aimed at letting the computer solve problems based on the knowledge it may have stored. Instead it builds models based on knowledge of, but not “in” the domain. The TripTych form of domain science & engineering differs from conventional ontological engineering in the following, essential ways: The TripTych domain descriptions rely essentially on a “built-in” upper ontology: types, abstract as well as model-oriented (i.e., concrete) and actions, events and behaviours. Domain science & engineering is not, to a first degree, concerned with modalities, and hence do not focus on the modeling of knowledge and belief, necessity and possibility, i.e., alethic modalities, epistemic modality (certainty), promise and obligation (deontic modalities), etcetera.

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

The next section overlaps with the present section.

1: Knowledge Engineering:

The concept of knowledge has occupied philosophers since Plato. No common agreement on what ‘knowledge’ is has been reached. From [128, 6, 136, 172] we may learn that *knowledge is a familiarity with someone or something; it can include facts, information, descriptions, or skills acquired through experience or education; it can refer to the theoretical or practical understanding of a subject; knowledge is produced by socio-cognitive aggregates (mainly humans) and is structured according to our understanding of how human reasoning and logic works.* The seminal reference here is [84]. The aim of knowledge engineering was formulated, in 1983, by an originator of the concept, Edward A. Feigenbaum [87]: knowledge engineering is an engineering discipline that involves integrating knowledge into computer systems

in order to solve complex problems normally requiring a high level of human expertise. *Knowledge engineering* focus on continually building up (acquire) large, shared data bases (i.e., knowledge bases), their continued maintenance, testing the validity of the stored ‘knowledge’, continued experiments with respect to knowledge representation, etcetera. *Knowledge engineering* can, perhaps, best be understood in contrast to algorithmic engineering: In the latter we seek more-or-less conventional, usually imperative programming language expressions of algorithms *whose algorithmic structure embodies the knowledge required to solve the problem being solved by the algorithm*. The former seeks to solve problems based on an interpreter inferring possible solutions from logical data. This logical data has three parts: *a collection that “mimics” the semantics of, say, the imperative programming language, a collection that formulates the problem, and a collection that constitutes the knowledge particular to the problem*. We refer to [63]. Domain science & engineering is not aimed at letting the computer solve problems based on the knowledge it may have stored. Instead it builds models based on knowledge of the domain.

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

Specific

2: Database Analysis:

There are different, however related “schools of database analysis”. DSD: the Bachman (or data structure) diagram model [10]; RDM: the relational data model [77]; and ER: entity set relationship model [72] “schools”. DSD and ER aim at graphically specifying database structures. Codd’s RDM simplifies the data models of DSD and ER while offering two kinds of languages with which to operate on RDM databases: SQL and Relational Algebra. All three “schools” are focused more on data modeling for databases than on domain modeling both enduring and perdurant entities.

3: Domain Analysis:

Domain analysis, or product line analysis (see below), as it was then conceived in the early 1980s by James Neighbors [142], is the analysis of related software systems in a domain to find their common and variable parts. This form of domain analysis turns matters “upside-down”: it is the set of software “systems” (or packages) that is subject to some form of inquiry, albeit having some domain in mind, in order to find common features of the software that can be said to represent a named domain.

In this section we shall mainly be comparing the TripTych approach to domain analysis to that of Reubén Prieto-Díaz’s approach [150, 151, 152]. Firstly, our understanding of domain analysis basically coincides with Prieto-Díaz’s. Secondly, in, for example, [150], Prieto-Díaz’s domain analysis is focused on the very important stages that precede the kind of domain modeling that we have described: major concerns are *selection of what appears to be similar, but specific entities, identification of common features, abstraction of entities and classification*. *Selection* and *identification* is assumed in our approach, but we suggest to follow the ideas of Prieto-Díaz. *Abstraction* (from values to types and signatures) and *classification* into parts, materials, actions, events and behaviours is what we have focused on. All-in-all we find Prieto-Díaz’s work very relevant to our work: relating to it by providing guidance to pre-modeling steps, thereby emphasising issues that are necessarily informal, yet difficult to get started on by most software engineers. Where we might differ is on the following: although Prieto-Díaz does mention a need for domain specific languages, he does not show examples of domain descriptions in such DSLs. We, of course, basically use mathematics as the DSL. In our approach we do not consider requirements, let alone software components, as do Prieto-Díaz, but we find that that is not an important issue.

4: Domain Specific Languages:

Martin Fowler³⁸ defines a *Domain-specific language* (DSL) as a *computer programming language of limited expressiveness focused on a particular domain* [90]. Other references are [135, 171]. Common to [171, 135, 90] is that they define a domain in terms of classes of software packages; that they never really “derive” the DSL from a description of the domain; and that they certainly do not describe the domain in terms of that DSL, for example, by formalising the DSL. In [108] a domain specific language for railway tracks is the basis for verification of the monitoring and control of train traffic on these tracks. Specifications in that domain specific language, DSL, manifested by track layout drawings and signal interlocking tables, are translated into SystemC [99]. [108] thus takes one very specific DSL and shows how to (informally) translate their “programs”, which are not “directly executable”, and hence does not satisfy Fowler’s definition of DSLs, into executable programs. [108] is a great paper, but it is not solving our problem, that of systematically describing any manifest domain. [108] does, however, point a way to search for — say graphical — DSLs and the possible translation of their programs into executable ones.

5: Feature-oriented Domain Analysis (FODA):

Feature oriented domain analysis (FODA) is a domain analysis method which introduced feature modeling to domain engineering. FODA was developed in 1990 following several U.S. Government research projects. Its concepts have been regarded as “critically advancing software engineering and software reuse.” The US Government-supported report [121] states: “*FODA is a necessary first step*” for software reuse. To the extent that TripTych domain engineering with its subsequent requirements engineering indeed encourages reuse at all levels: domain descriptions and requirements prescription, we can only agree. Another source on FODA is [78]. Since FODA “leans” quite heavily on ‘Software Product Line Engineering’ our remarks in that section, next, apply equally well here.

6: Software Product Line Engineering:

Software product line engineering, earlier known as domain engineering, is the entire process of *reusing domain knowledge* in the production of new software systems. Key concerns of software product line engineering are reuse, the building of repositories of reusable software components, and domain specific languages with which to more-or-less automatically build software based on reusable software components. These are not the primary concerns of TripTych *domain science & engineering*. But they do become concerns as we move from domain descriptions to requirements prescriptions. But it strongly seems that software product line engineering is not really focused on the concerns of domain description — such as is TripTych domain engineering. It seems that software product line engineering is primarily based, as is, for example, FODA: Feature-oriented Domain Analysis, on analysing features of software systems. Our [38] puts the ideas of software product lines and model-oriented software development in the context of the TripTych approach.

7: Problem Frames:

The concept of problem frames is covered in [116]. Jackson’s prescription for software development focus on the “triple development” of descriptions of the problem world, the requirements and the machine (i.e., the hardware and software) to be built. Here domain analysis means the same as for us: the problem world analysis. In the problem frame approach the software developer plays three, that is, all the TripTych rôles: domain engineer, requirements engineer and software engineer, “all at the same time”, iterating between these rôles repeatedly. So, perhaps belabouring the point, domain engineering is done only to the extent needed by the prescription of requirements and the *design* of software. These, really are minor points. But in “restricting” oneself to consider only those aspects of the domain which are mandated by the requirements prescription and software design one is considering a potentially smaller fragment [117] of the domain than is suggested by the TripTych approach. At the same time one is, however, sure to consider aspects of the domain that might have been overlooked when pursuing domain description development in the “more general” TripTych approach.

³⁸ <http://martinfowler.com/dsl.html>

8: Domain Specific Software Architectures (DSSA):

It seems that the concept of DSSA was formulated by a group of ARPA³⁹ project “seekers” who also performed a year long study (from around early-mid 1990s); key members of the DSSA project were Will Tracz, Bob Balzer, Rick Hayes-Roth and Richard Platek [174]. The [174] definition of domain engineering is “*the process of creating a DSSA: domain analysis and domain modeling followed by creating a software architecture and populating it with software components.*” This definition is basically followed also by [137, 167, 133]. Defined and pursued this way, DSSA appears, notably in these latter references, to start with the analysis of software components, “per domain”, to identify commonalities within application software, and to then base the idea of software architecture on these findings. Thus DSSA turns matter “upside-down” with respect to TripTych requirements development by starting with software components, assuming that these satisfy some requirements, and then suggesting domain specific software built using these components. This is not what we are doing: we suggest, **Chapter 5, From Domain Descriptions to Requirements Prescriptions, [54]**, that requirements can be “derived” systematically from, and formally related back to domain descriptions without, in principle, considering software components, whether already existing, or being subsequently developed. Of course, given a domain description it is obvious that one can develop, from it, any number of requirements prescriptions and that these may strongly hint at shared, (to be) implemented software components; but it may also, as well, be the case that two or more requirements prescriptions “derived” from the same domain description may share no software components whatsoever! It seems to this author that had the DSSA promoters based their studies and practice on also using formal specifications, at all levels of their study and practice, then some very interesting insights might have arisen.

9: Domain Driven Design (DDD):

Domain-driven design (DDD)⁴⁰ “*is an approach to developing software for complex needs by deeply connecting the implementation to an evolving model of the core business concepts; the premise of domain-driven design is the following: placing the project’s primary focus on the core domain and domain logic; basing complex designs on a model; initiating a creative collaboration between technical and domain experts to iteratively cut ever closer to the conceptual heart of the problem.*”⁴¹ We have studied some of the DDD literature, mostly only accessible on the Internet, but see also [109], and find that it really does not contribute to new insight into domains such as we see them: it is just “plain, good old software engineering cooked up with a new jargon.

10: Unified Modeling Language (UML):

Three books representative of UML are [65, 162, 118]. The term domain analysis appears numerous times in these books, yet there is no clear, definitive understanding of whether it, the domain, stands for entities in the domain such as we understand it, or whether it is wrought up, as in several of the ‘approaches’ treated in this section, to wit, in items [3–5, 7–9] with either software design (as it most often is), or requirements prescription. Certainly, in UML, in [65, 162, 118] as well as in most published papers claiming “adherence” to UML, that domain analysis usually is manifested in some UML text which “models” some requirements facet. Nothing is necessarily wrong with that, but it is therefore not really the TripTych form of domain analysis with its concepts of abstract representations of enduring and perdurants, with its distinctions between domain and requirements, and with its possibility of “deriving” requirements prescriptions from domain descriptions. The UML notion of class diagrams is worth relating to our structuring of the domain. Class diagrams appear to be inspired by [10, Bachman, 1969] and [72, Chen, 1976]. It seems that (i) each part sort — as well as other than part sorts — deserves a class diagram (box); and (ii) that (assignable) attributes — as well as other non-part types — are written into the diagram box. Class diagram boxes are line-connected with annotations where some annotations are as per the mereology of the part type and the connected part types and others are not part related. The class diagrams are said to be object-oriented but

³⁹ ARPA: The US DoD Advanced Research Projects Agency

⁴⁰ Eric Evans: <http://www.domaindrivendesign.org/>

⁴¹ http://en.wikipedia.org/wiki/Domain-driven_design

it is not clear how objects relate to parts as many are rather implementation-oriented quantities. All this needs looking into a bit more, for those who care.

11: Requirements Engineering:

There are in-numerous books and published papers on requirements engineering. A seminal one is [177]. I, myself, find [125] full of very useful, non-trivial insight. [81] is seminal in that it brings a number of early contributions and views on requirements engineering. Conventional text books, notably [146, 149, 169] all have their “mandatory”, yet conventional coverage of requirements engineering. None of them “derive” requirements from domain descriptions, yes, OK, from domains, but since their description is not mandated it is unclear what “the domain” is. Most of them repeatedly refer to domain analysis but since a written record of that domain analysis is not mandated it is unclear what “domain analysis” really amounts to. Axel van Laamsweerde’s book [177] is remarkable. Although also it does not mandate descriptions of domains it is quite precise as to the relationships between domains and requirements. Besides, it has a fine treatment of the distinction between goals and requirements, also formally. Most of the advices given in [125] can beneficially be followed also in TripTych requirements development. Neither [177] nor [125] preempts TripTych requirements development.

Summary of Comparisons

We find that there are two kinds of relevant comparisons: the concept of ontology, its science more than its engineering, and the *Problem Frame* work of Michael A. Jackson. The ontology work, as commented upon in Item [1] (Pages 45–46), is partly relevant to our work: There are at least two issues: Different classes of domains may need distinct upper ontologies. Section 1.5.1 suggests that there may be different upper ontologies for non-manifest domains such as *financial systems*, etcetera. This seems to warrant at least a comparative study. We have assumed, cf. Sect. 1.3.4, that attributes cannot be separated from parts. [119, Johansson 2005] develops the notion that *persisting quality instances are enduring particulars*. The issue need further clarification.

Of all the other “comparison” items ([2]–[12]) basically only Jackson’s *problem frames* (Item [8]) and [108] (Item [5]) really take the same view of domains and, in essence, basically maintain similar relations between requirements prescription and domain description. So potential sources of, we should claim, mutual inspiration ought be found in one-another’s work — with, for example, [100, 117, 108], and the present document, being a good starting point.

But none of the referenced works make the distinction between discrete endurants (parts) and their qualities, with their further distinctions between *unique identifiers*, *mereology* and *attributes*. And none of them makes the distinction between *parts*, *components* and *materials*. Therefore our contribution can include the mapping of parts into behaviours interacting as per the part mereologies as highlighted in the process schemas of Sect. 1.4.11 Pages 39–42.

1.5.4 Open Problems

The present paper has outlined a great number of principles, techniques and tools of domain analysis & description. They give rise, now, to the investigation of further principles, techniques and tools as well as underlying theories. We list some of these “to do” items: (1) *a mathematical model of prompts*; (2) *a sharpened definition of “what is a domain”*; (3) *laws of description prompts*; (4) *an understanding of domain facets*; (5) *a prompt calculus for perdurants*; (6) *commensurate discrete and continuous models* [179, 186]; (7) *a study of the interplay between parts, materials and components*; (8) *a closer study of external attributes and their variety of access forms and of biddable attributes*; and (9) *specific domain theories*; etcetera.

1.5.5 Tony Hoare's Summary on 'Domain Modeling'

In a 2006 e-mail, in response, undoubtedly to my steadfast, perhaps conceived as stubborn insistence, on domain engineering, Tony Hoare summed up his reaction to domain engineering as follows, and I quote⁴²:

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

- 1 The models describe all aspects of the real world that are relevant for any good software design in the area. They describe possible places to define the system boundary for any particular project.*
- 2 They make explicit the preconditions about the real world that have to be made in any embedded software design, especially one that is going to be formally proved.*
- 3 They describe the whole range of possible designs for the software, and the whole range of technologies available for its realisation.*
- 4 They provide a framework for a full analysis of requirements, which is wholly independent of the technology of implementation.*
- 5 They enumerate and analyse the decisions that must be taken earlier or later in any design project, and identify those that are independent and those that conflict. Late discovery of feature interactions can be avoided.”*

All of these issues are covered, to some extent, in [26, Part IV]. Tony Hoare's list pertains to a wider range than just the Manifest Domains treated in this paper.

1.5.6 Beauty Is Our Business

*It's life that matters, nothing but life –
the process of discovering, the everlasting and perpetual process,
not the discovery itself, at all.*⁴³

I find that quote appropriate in the following, albeit rather mundane, sense: It is the process of analysing and describing a domain that exhilarates me: that causes me to feel very happy and excited. There is beauty [88, E.W. Dijkstra Festschrift] not only in the result but also in the process.

1.6 Bibliographical Notes

Web page www.imm.dtu.dk/~dibj/domains/ lists the published papers and reports mentioned below. I have thought about domain engineering for more than 25 years. But serious, focused writing only started to appear since 2006: [26, Part IV] — with [23, 20] being exceptions: [28] suggests a number of domain science and engineering research topics; **Chapter 2, Domain Facets: Analysis & Description, [53]** covers the concept of domain facets; [59] explores compositionality and Galois connections. **Chapter 5, From Domain Descriptions to Requirements Prescriptions, [54]** shows how to systematically, but, of course, not automatically, “derive” requirements prescriptions from domain descriptions; [36] takes the triptych software development as a basis for outlining principles for believable software management; [31, 43] presents a model for Stanisław Leśniewski's [68] concept of mereology; [34, 37] present an extensive example and is otherwise a precursor for the present chapter; **Chapter 3, Domain Analysis and Description – Formal Models of Processes and Prompts, [52]** presents a formal model (i.e., an operational semantics) of the process of “discovering” domains and of their prompts; **Chapter 6, Domains: Their Simulation, Monitoring and Control, [38]** presents, based on the TripTych view of software development as ideally proceeding from domain description via requirements prescription to software design, concepts such as software demos and simulators; [40] analyses the TripTych, especially its domain engineering approach, with respect to Maslow's⁴⁴ and Peterson's and Seligman's⁴⁵ notions of humanity:

⁴² E-Mail to Dines Bjørner, July 19, 2006

⁴³ Fyodor Dostoyevsky, *The Idiot*, 1868, Part 3, Sect. V

⁴⁴ *Theory of Human Motivation*. Psychological Review 50(4) (1943):370-96; and *Motivation and Personality*, Third Edition, Harper and Row Publishers, 1954.

⁴⁵ *Character strengths and virtues: A handbook and classification*. Oxford University Press, 2004

how can computing relate to notions of humanity; the first part of [45] is a precursor for the present paper with its second part presenting a first formal model of the elicitation process of analysis and description based on the prompts more definitively presented in the current paper; and [46] focus on domain safety criticality. The present paper basically replaces the domain analysis and description section of all of the above reference — including [26, Part IV, 2006].

Domain Facets: Analysis & Description

Summary

This chapter¹ is a continuation of **Chapter 1, Manifest Domains: Analysis & Description, [49]** and is a precursor for **Chapter 5, [From Domain Descriptions to Requirements Prescriptions, [54]**. Where Chap. 1 covered a method for analysing and describing the intrinsics of manifest domains, the present paper covers principles and techniques for describing domain facets — not covered in Chap. 1. Where Chap. 5 covers some basic principles and techniques for structuring requirements analysis and prescription, the present paper hints at requirements that can be derived from domain facets. By a domain facet we shall understand one amongst a finite set of generic ways of analysing a domain: a view of the domain, such that the different facets cover conceptually different views, and such that these views together cover the domain. We shall outline the following domain facets: *intrinsic*s, *support technologies*, *rules & regulations*, *scripts*, *license languages*, *management & organisation*, and *human behaviour*. The present paper is a substantial reformulation and extension of [33] on the background of Chap. 1 [49].

2.1 Introduction

In **Chapter 1, Manifest Domains: Analysis & Description, [49]** we outlined a method for analysing &² describing domains. By a **method** we shall understand a set of principles, techniques and tools for analysing and constructing (synthesizing) an artifact, as here a description \odot ³ By a **domain** we shall understand a potentially infinite set of endurants and a usually finite set of perdurants (actions, events and behaviours) [the latter map endurants into endurants] such that these entities are observable in the world and can be described \odot In this book we cover domain analysis & description principles and techniques not covered in [49]. That paper focused on manifest domains. Here we, on one side, go “outside” the realm of manifest domains, and, on the other side, cover, what we shall refer to as, facets, not covered in [49].

2.1.1 Facets of Domains

By a **domain facet** we shall understand *one amongst a finite set of generic ways of analysing a domain: a view of the domain, such that the different facets cover conceptually different views, and such that these views together cover the domain* \odot Now, the definition of what a domain facet is can seem vague. It cannot be otherwise. The definition is sharpened by the definitions of the specific facets. You can say, that

¹ This chapter is based on [53].

² We use the ampersand (logogram), &, in the following sense: Let A and B be two concepts. By A and B we mean to refer to these two concepts. With $A\&B$ we mean to refer to a composite concept “containing” elements of both A and B .

³ The \odot symbol delimits a definition.

the definition of domain facet is the “sum” of the definitions of these specific facets. The specific facets – so far⁴ – are: intrinsics (Sect. 2.2), support technology (Sect. 2.3), rules & regulations (Sect. 2.4), scripts (Sect. 2.5), license languages (Sect. 2.6), management & organisation (Sect. 2.7) and human behaviour (Sect. 2.8). Of these, the rules & regulations, scripts and license languages are closely related. Vagueness may “pop up”, here and there, in the delineation of facets. It is necessarily so. We are not in a domain of computer science, let alone mathematics, where we can just define ourselves precisely out of any vagueness problems. We are in the domain of (usually) really world facts. And these are often hard to encircle.

2.1.2 Structure of Paper

The structure of the paper follows the seven specific facets, as listed above. Each section, 2.2.–2.8., starts by a definition of the specific facet, Then follows an analysis of the abstract concepts involved usually with one or more examples – with these examples making up most of the section. We then “speculate” on derivable requirements thus relating the present paper to [54]. We close each of the sections, 2.2.–2.8., with some comments on how to model the specific facet of that section.

• • •

Examples 1–22 of sections 2.2.–2.8. present quite a variety. In that, they reflect the wide spectrum of facets.

• • •

More generally, domains can be characterised by intrinsically being enduring, or function, or event, or behaviour intensive. Software support for activities in such domains then typically amount to database systems, computation-bound systems, real-time embedded systems, respectively distributed process monitoring and control systems. Other than this brief discourse we shall not cover the “intensity”-aspect of domains in this paper.

2.2 Intrinsics

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

2.2.1 Conceptual Analysis

The principles and techniques of domain analysis & description, as unfolded in [49], focused on and resulted in descriptions of the intrinsics of domains. They did so in focusing the analysis (and hence the description) on the basic endurants and their related perdurants, that is, on those parts that most readily present themselves for observation, analysis & description.

Example. 1 Railway Net Intrinsics: We narrate and formalise three railway net intrinsics.

From the view of *potential train passengers* a railway net consists of lines, l:L, with names, ln:Ln, stations, s:S, with names sn:Sn, and trains, tn:TN, with names tnm:Tnm. A line connects exactly two distinct stations.

scheme N0 =

```
class
  type
    N, L, S, Sn, Ln, TN, Tnm
  value
    obs_Ls: N → L-set, obs_Ss: N → S-set
```

⁴ We write: ‘so far’ in order to “announce”, or hint that there may be other specific facets. The one listed are the ones we have been able to “isolate”, to identify, in the most recent 10-12 years.

```

    obs_Ln: L → Ln, obs_Sn: S → Sn
    obs_Sns: L → Sn-set, obs_Lns: S → Ln-set
axiom
    ...
end

```

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

From the view of *actual train passengers* a railway net — in addition to the above — allows for several lines between any pair of stations and, within stations, provides for one or more platform tracks, tr:Tr, with names, trn:Trn, from which to embark on or alight from a train.

scheme N1 = extend N0 with

```

class
    type
        Tr, Trn
    value
        obs_Tr: S → Tr-set, obs_Trn: Tr → Trn
    axiom
        ...
end

```

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

From the view of *train operating staff* a railway net — in addition to the above — has lines and stations consisting of suitably connected rail units. A rail unit is either a simple (i.e., linear, straight) unit, or is a switch unit, or is a simple crossover unit, or is a switchable crossover unit, etc. Simple units have two connectors. Switch units have three connectors. Simple and switchable crossover units have four connectors. A path, p:P, (through a unit) is a pair of connectors of that unit. A state, $\sigma : \Sigma$, of a unit is the set of paths, in the direction of which a train may travel. A (current) state may be empty: The unit is closed for traffic. A unit can be in any one of a number of states of its state space, $\omega : \Omega$.

scheme N2 = extend N1 with

```

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

```

Unit and connector types have been added as have concrete types for paths, unit states, unit state spaces and related observer functions, including unit state and unit state space observers. □

Different stakeholder perspectives, not only of intrinsic, as here, but of any facet, lead to a number of different models. The name of a phenomenon of one perspective, that is, of one model, may coincide

with the name of a “similar” phenomenon of another perspective, that is, of another model, and so on. If the intention is that the “same” names cover comparable phenomena, then the developer must state the comparison relation.

Example. 2 Intrinsic of Switches: The intrinsic attribute of a rail switch is that it can take on a number of states. A simple switch (${}^c_l Y_c^{c/}$) has three connectors: $\{c, c_l, c/\}$. c is the connector of the common rail from which one can either “go straight” c_l , or “fork” $c/$ (Fig. 2.1). So we have that a possible state space of such a switch could be ω_{gs} :

$$\begin{aligned} & \{\{\}, \\ & \{(c, c_l)\}, \{(c_l, c)\}, \{(c, c_l), (c_l, c)\}, \\ & \{(c, c/)\}, \{(c/, c)\}, \{(c, c/), (c/, c)\}, \{(c/, c), (c_l, c)\}, \\ & \{(c, c_l), (c_l, c), (c/, c)\}, \{(c, c/), (c/, c), (c_l, c)\}, \{(c/, c), (c, c_l)\}, \{(c, c/), (c_l, c)\}\} \end{aligned}$$

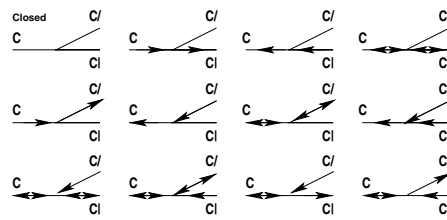


Fig. 2.1. Possible states of a rail switch

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

Example. 3 An Intrinsic of Documents: Think of documents, written, by hand, or typed “onto” a computer text processing system. One way of considering such documents is as follows. First we abstract from the syntax that such a document, or set of more-or-less related documents, or just documents, may have: whether they are letters, with sender and receive addressees, dates written, sent and/or received, opening and closing paragraphs, etc., etc.; or they are books, technical, scientific, novels, or otherwise, or they are application forms, tax returns, patient medical records, or otherwise. Then we focus on the operations that one may perform on documents: their creation, editing, reading, copying, authorisation, “transfer”⁵, “freezing”⁶, and shredding. Finally we consider documents as manifest parts, cf. [49]. Parts, so documents have unique identifications, in this case, changeable mereology, and a number of attributes. The mereology of a document, d , reflects those other documents upon which a document is based, i.e., refers to, and/or refers to d . Among the attributes of a document we can think of (i) a trace of what has happened to a document, i.e., a trace of all the operations performed on “that” document, since and including creation — with that trace, for example, consisting of time-stamped triples of the essence of the operations, the “actor” of the operation (i.e., the operator), and possibly some abstraction of the locale of the document when operated upon; (ii) a synopsis of what the document text “is all about”, (iii) and some “rendition” of the document text. □

This view of documents, whether “implementable” or “implemented” or not, is at the basis of our view of license languages (for *digital media*, *health-care* (patient medical record), *documents*, and *transport* (contracts) as that facet is covered in Sect. 2.6.

⁵ to other editors, readers, etc.

⁶ i.e., prevention of future operations

2.2.2 Requirements

[54] illustrated requirements “derived” from the intrinsics of a road transport system – as outlined in [49]. So this paper has little to add to the subject of requirements “derived” from intrinsics.

2.2.3 On Modeling Intrinsics

[49] outlined basic principles, techniques and tools for modeling the intrinsics of manifest domains. Modeling the domain intrinsics can often be expressed in property-oriented specification languages (like CafeOBJ [92]), model-oriented specification languages (like Alloy [114], B [1], VDM-SL [61, 62, 89], RSL [96], or Z [183]), event-based languages (like Petri nets or [156] or CSP [111]), respectively in process-based specification languages (like MSCs [113], LSCs [106], Statecharts [105], or CSP [111]). An area not well-developed is that of modeling continuous domain phenomena like the dynamics of automobile, train and aircraft movements, flow in pipelines, etc. We refer to [144].

2.3 Support Technologies

- By a domain **support technology** we shall understand ways and means of implementing certain observed phenomena or certain conceived concepts ☉

The “ways and means” may be in the form of “soft technologies”: human manpower, see, however, Sect. 2.8, or in the form of “hard” technologies: electro-mechanics, etc. The term ‘implementing’ is crucial. It is here used in the sense that, $\psi\tau$, which is an ‘implementation’ of a *endurant* or *perdurant*, ϕ , is an extension of ϕ , with ϕ being an abstraction of $\psi\tau$. We strive for the extensions to be proof theoretic conservative extensions [132].

2.3.1 Conceptual Analysis

There are [always] basically two approaches the task of analysing & describing the support technology facets of a domain. One either stumbles over it, or one tries to tackle the issue systematically. The “stumbling” approach occurs when one, in the midst of analysing & describing a domain realises that one is tackling something that satisfies the definition of a support technology facet. In the systematic approach to the analysis & description of the support technology facets of a domain one usually starts with a basically intrinsics facet-oriented domain description. We then suggest that the domain engineer “inquires” of ever *endurant* and *perdurant* whether it is an intrinsic entity or, perhaps a support technology.

Example. 4 Railway Support Technology: We give a rough sketch description of possible rail unit switch technologies.

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

(ii) With the advent of reasonably reliable mechanics, pulleys and levers⁷ and steel wires, switches were made to change state by means of “throwing” levers in a cabin tower located centrally at the station (with the lever then connected through wires etc., to the actual switch).

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

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

It must be stressed that Example 4 is just a rough sketch. In a proper narrative description the software (cum domain) engineer must describe, in detail, the subsystem of electronics, electro-mechanics and the human operator interface (buttons, lights, sounds, etc.). An aspect of supporting technology includes recording the state-behaviour in response to external stimuli. We give an example.

⁷ <https://en.wikipedia.org/wiki/Pulley> and <http://en.wikipedia.org/wiki/Lever>

Example. 5 Probabilistic Rail Switch Unit State Transitions: Figure 2.2 indicates a way of formalising this aspect of a supporting technology. Figure 2.2 intends to model the probabilistic (erroneous and correct) behaviour of a switch when subjected to settings (to switched (s) state) and re-settings (to direct (d) state). A switch may go to the switched state from the direct state when subjected to a switch setting s with probability psd . □

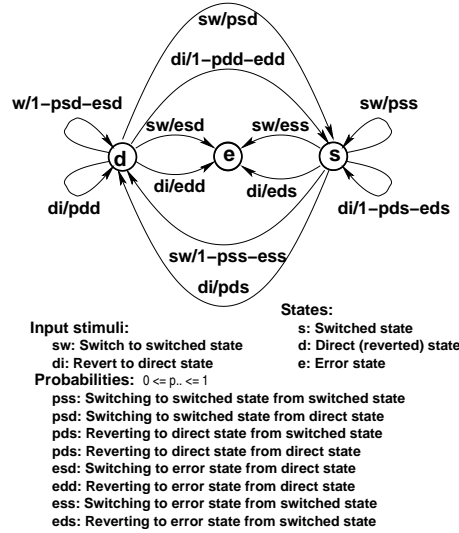


Fig. 2.2. Probabilistic state switching

Example. 6 Traffic Signals: We continue Examples 17, 18, 25 and 33 of [49]. This example should, however, be understandable without reference to [49]. A traffic signal represents a technology in support of visualising hub states (transport net road intersection signaling states) and in effecting state changes.

- 81 A traffic signal, $ts:TS$, is considered a part with observable hub states and hub state spaces. Hub states and hub state spaces are programmable, respectively static attributes of traffic signals.
- 82 A hub state space, $h\omega$, is a set of hub states such that each current hub state is in that hubs' hub state space.
- 83 A hub state, $h\sigma$, is now modeled as a set of hub triples.
- 84 Each hub triple has a link identifier l_i ("coming from"), a colour (red, yellow or green), and another link identifier l_j ("going to").
- 85 Signaling is now a sequence of one or more pairs of next hub states and time intervals, $ti:TI$, for example: $\langle (h\sigma_1, ti_1), (h\sigma_2, ti_2), \dots, (h\sigma_{n-1}, ti_{n-1}), (h\sigma_n, ti_n) \rangle, n > 0$. The idea of a signaling is to first change the designated hub to state $h\sigma_1$, then wait ti_1 time units, then set the designated hub to state $h\sigma_2$, then wait ti_2 time units, etcetera, ending with final state σ_n and a (supposedly) long time interval ti_n before any decisions are to be made as to another signaling. The set of hub states $\{h\sigma_1, h\sigma_2, \dots, h\sigma_{n-1}\}$ of $\langle (h\sigma_1, ti_1), (h\sigma_2, ti_2), \dots, (h\sigma_{n-1}, ti_{n-1}), (h\sigma_n, ti_n) \rangle, n > 0$, is called the set of intermediate states. Their purpose is to secure an orderly phase out of green via yellow to red and phase in of red via yellow to green in some order for the various directions. We leave it to the reader to devise proper well-formedness conditions for signaling sequences as they depend on the hub topology.
- 86 A street signal (a semaphore) is now abstracted as a map from pairs of hub states to signaling sequences. The idea is that given a hub one can observe its semaphore, and given the state, $h\sigma$ (not in the above set), of the hub "to be signaled" and the state $h\sigma_n$ into which that hub is to be signal-led "one looks up" under that pair in the semaphore and obtains the desired signaling.

type

```

81 TS  $\equiv$  H, H $\Sigma$ , H $\Omega$ 
value
82 obs_H $\Sigma$ : H, TS  $\rightarrow$  H $\Sigma$ 
82 obs_H $\Omega$ : H, TS  $\rightarrow$  H $\Omega$ 
type
83 H $\Sigma$  = Htriple-set
83 H $\Omega$  = H $\Sigma$ -set
414 Htriple = LI  $\times$  Colour  $\times$  LI
axiom
82  $\forall$  ts:TS  $\cdot$  obs_H $\Sigma$ (ts)  $\in$  obs_H $\Omega$ (ts)
type
414 Colour == red | yellow | green
85 Signaling = (H $\Sigma$   $\times$  TI)*
85 TI
86 Semaphore = (H $\Sigma$   $\times$  H $\Sigma$ )  $\rightarrow_m$  Signalling
value
86 obs_Semaphore: TS  $\rightarrow$  Semaphore

```

87 Based on [49] we treat hubs as processes with hub state spaces and semaphores as static attributes and hub states as programmable attributes. We ignore other attributes and input/outputs.

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

```

value
87. hub: HI  $\times$  LI-set  $\times$  (H $\Omega$   $\times$  Semaphore)  $\rightarrow$  H $\Sigma$  in ... out ... Unit
87. hub(hi, lis, (h $\omega$ , sema))(h $\sigma$ )  $\equiv$ 
87. ...
88.  $\sqcap$  let h $\sigma'$ :HI  $\cdot$  ... in hub(hi, lis, (h $\omega$ , sema))(signaling(h $\sigma$ , h $\sigma'$ )) end
87. ...
87. pre: {h $\sigma$ , h $\sigma'$ }  $\subseteq$  h $\omega$ 

```

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

89 Given two traffic signal, i.e., hub states, $h\sigma_{init}$ and $h\sigma_{end}$, where $h\sigma_{init}$ designates a present hub state and $h\sigma_{end}$ designates a desired next hub state after signaling.

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

```

value
89 signaling: (H $\Sigma$   $\times$  H $\Sigma$ )  $\times$  Semaphore  $\rightarrow$  H $\Sigma$   $\rightarrow$  H $\Sigma$ 
90 signaling(h $\sigma_{init}$ , h $\sigma_{end}$ , sema)(h $\sigma$ )  $\equiv$  let sg = sema(h $\sigma_{init}$ , h $\sigma_{end}$ ) in signal_sequence(sg)(h $\sigma$ ) end
90 pre h $\sigma_{init}$  = h $\sigma$   $\wedge$  (h $\sigma_{init}$ , h $\sigma_{end}$ )  $\in$  dom sema

```

If a desired hub state change fails (i.e., does not meet the **pre**-condition, or for other reasons (e.g., failure of technology)), then we do not define the outcome of signaling.

```

90 signal_sequence( $\langle \rangle$ )(h $\sigma$ )  $\equiv$  h $\sigma$ 
90 signal_sequence( $\langle (h\sigma', ti) \rangle^{\wedge} sg$ )(h $\sigma$ )  $\equiv$  wait(ti); signal_sequence(sg)(h $\sigma'$ )

```

We omit expression of a number of well-formedness conditions, e.g., that the *htriple* link identifiers are those of the corresponding mereology (*lis*), etcetera. The design of the semaphore, for a single hub or for a net of connected hubs has many similarities with the design of interlocking tables for railway tracks [108]. □

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

Example. 7 Railway Optical Gates: Train traffic ($itf:iTF$), intrinsically, is a total function over some time interval, from time ($t:T$) to continuously positioned ($p:P$) trains ($tn:TN$). Conventional optical gates sample, at regular intervals, the intrinsic train traffic. The result is a sampled traffic ($stf:sTF$). Hence the collection of all optical gates, for any given railway, is a partial function from intrinsic to sampled train traffics (stf). We need to express quality criteria that any optical gate technology should satisfy — relative to a necessary and sufficient description of a closeness predicate. The following axiom does that:

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

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

type

```
T, TN
P = U*
NetTraffic == net:N trf:(TN  $\rightarrow$  P)
iTF = T  $\rightarrow$  NetTraffic
sTF = T  $\rightarrow$  NetTraffic
oG = iTF  $\rightarrow$  sTF
```

value

```
close: NetTraffic  $\times$  TN  $\times$  NetTraffic  $\rightarrow$  Bool
```

axiom

```
 $\forall itt:iTF, og:OG \bullet \text{let } stt = og(itt) \text{ in}$ 
 $\forall t:T \bullet t \in \text{dom } stt \Rightarrow$ 
 $\forall Tn:TN \bullet tn \in \text{dom } trf(itt(t))$ 
 $\Rightarrow tn \in \text{dom } trf(stt(t)) \wedge \text{close}(itt(t), tn, stt(t)) \text{ end}$ 
```

Check-ability is an issue of testing the optical gates when delivered for conformance to the closeness predicate, i.e., to the axiom. □

2.3.2 Requirements

Section 4.4 [Extension] of [54] illustrates a possible toll-gate, whose behaviour exemplifies a support technology. So do pumps of a pipe-line system such as illustrated in Examples 24, 29 and 42–44 in [49]. A pump of a pipe-line system gives rise to several forms of support technologies: from the Egyptian Shadoof [irrigation] pumps, and the Hellenic Archimedian screw pumps, via the 11th century Su Song pumps of China⁸, and the hydraulic “technologies” of Moorish Spain⁹ to the centrifugal and gear pumps of the early industrial age, etcetera. The techniques – to mention those that have influenced this author – of [187, 120, 143, 108] appears to apply well to the modeling of support technology requirements.

2.3.3 On Modeling Support Technologies

Support technologies in their relation to the domain in which they reside typically reflect real-time embeddedness. As such the techniques and languages for modeling support technologies resemble those for modeling event and process intensity, while temporal notions are brought into focus. Hence typical modeling notations include event-based languages (like Petri nets [156] or CSP [111]), respectively process-based specification languages (like MSCs, [113], LSCs [106], Statecharts [105], or CSP [111]), as well as temporal languages (like the Duration Calculus and [187] and Temporal Logic of Actions, TLA+) [124]).

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

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

2.4 Rules &¹⁰ Regulations

- By a **domain rule** we shall understand some text (in the domain) which prescribes how people or equipment are expected to behave when dispatching their duties, respectively when performing their functions ◉
- By a **domain regulation** we shall understand some text (in the domain) which prescribes what remedial actions are to be taken when it is decided that a rule has not been followed according to its intention ◉

The domain rules & regulations need or may not be explicitly present, i.e., written down. They may be part of the “folklore”, i.e., tacitly assumed and understood.

2.4.1 Conceptual Analysis

Example. 8 Trains at Stations:

- Rule: In China the arrival and departure of trains at, respectively from, railway stations is subject to the following rule:
In any three-minute interval at most one train may either arrive to or depart from a railway station.
- Regulation: *If it is discovered that the above rule is not obeyed*, then there is some regulation which prescribes administrative or legal management and/or staff action, as well as some correction to the railway traffic.

□

Example. 9 Trains Along Lines:

- Rule: In many countries railway lines (between stations) are segmented into blocks or sectors. The purpose is to stipulate that if two or more trains are moving along the line, then:
There must be at least one free sector (i.e., without a train) between any two trains along a line.
- Regulation: *If it is discovered that the above rule is not obeyed*, then there is some regulation which prescribes administrative or legal management and/or staff action, as well as some correction to the railway traffic.

□

At a meta-level, i.e., explaining the general framework for describing the syntax and semantics of the human-oriented domain languages for expressing rules and regulations, we can say the following: There are, abstractly speaking, usually three kinds of languages involved wrt. (i.e., when expressing) rules and regulations (respectively when invoking actions that are subject to rules and regulations). Two languages, Rules and Reg, exist for describing rules, respectively regulations; and one, Stimulus, exists for describing the form of the [always current] domain action stimuli. A syntactic stimulus, sy_sti , denotes a function, $se_sti:STI: \Theta \rightarrow \Theta$, from any configuration to a next configuration, where configurations are those of the system being subjected to stimulations. A syntactic rule, $sy_rul:Rule$, stands for, i.e., has as its semantics, its meaning, $rul:RUL$, a predicate over current and next configurations, $(\Theta \times \Theta) \rightarrow \mathbf{Bool}$, where these next configurations have been brought about, i.e., caused, by the stimuli. These stimuli express: If the predicate holds then the stimulus will result in a valid next configuration.

type

Stimulus, Rule, Θ
 $STI = \Theta \rightarrow \Theta$
 $RUL = (\Theta \times \Theta) \rightarrow \mathbf{Bool}$

¹⁰ The concept unifier ‘&’ expresses that $A \& B$ designates one concept, not two: A and B .

value

meaning: Stimulus \rightarrow STI
 meaning: Rule \rightarrow RUL
 valid: Stimulus \times Rule $\rightarrow \Theta \rightarrow \mathbf{Bool}$
 valid(sy_sti,sy_rul)(θ) \equiv meaning(sy_rul)(θ , (meaning(sy_sti))(θ))

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

type

Reg
 Rul_and_Reg = Rule \times Reg
 REG = Pre_REG \times Act_REG
 Pre_REG = $\Theta \times \Theta \rightarrow \mathbf{Bool}$
 Act_REG = $\Theta \rightarrow \Theta$

value

interpret: Reg \rightarrow REG

The idea is now the following: Any action (i.e., event) of the system, i.e., the application of any stimulus, may be an action (i.e., event) in accordance with the rules, or it may not. Rules therefore express whether stimuli are valid or not in the current configuration. And regulations therefore express whether they should be applied, and, if so, with what effort. More specifically, there is usually, in any current system configuration, given a set of pairs of rules and regulations. Let (sy_rul,sy_reg) be any such pair. Let sy_sti be any possible stimulus. And let θ be the current configuration. Let the stimulus, sy_sti, applied in that configuration result in a next configuration, θ' , where $\theta' = (\text{meaning}(\text{sy_sti}))(\theta)$. Let θ' violate the rule, $\sim\text{valid}(\text{sy_sti},\text{sy_rul})(\theta)$, then if predicate part, pre_reg, of the meaning of the regulation, sy_reg, holds in that violating next configuration, pre_reg(θ , (meaning(sy_sti))(θ)), then the action part, act_reg, of the meaning of the regulation, sy_reg, must be applied, act_reg(θ), to remedy the situation.

axiom

$\forall (\text{sy_rul},\text{sy_reg}): \text{Rul_and_Reg} \cdot$
 let se_rul = meaning(sy_rul),
 (pre_reg,act_reg) = meaning(sy_reg) **in**
 $\forall \text{sy_sti}:\text{Stimulus}, \theta:\Theta \cdot$
 $\sim\text{valid}(\text{sy_sti},\text{se_rul})(\theta)$
 $\Rightarrow \text{pre_reg}(\theta, (\text{meaning}(\text{sy_sti}))(\theta))$
 $\Rightarrow \exists n\theta:\Theta \cdot \text{act_reg}(\theta)=n\theta \wedge \text{se_rul}(\theta,n\theta)$
end

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

2.4.2 Requirements

Implementation of rules & regulations implies monitoring and partially controlling the states symbolised by Θ in Sect. 2.4.1. Thus some partial implementation of Θ must be required; as must some monitoring of states $\theta:\Theta$ and implementation of the predicates *meaning*, *valid*, *interpret*, *pre_reg* and action(s) *act_reg*. The emerging requirements follow very much in the line of support technology requirements.

2.4.3 On Modeling Rules and Regulations

Usually rules (as well as regulations) are expressed in terms of domain entities, including those grouped into “the state”, functions, events, and behaviours. Thus the full spectrum of model-ing techniques and notations may be needed. Since rules usually express properties one often uses some combination of axioms and wellformedness predicates. Properties sometimes include temporality and hence temporal notations (like Duration Calculus or Temporal Logic of Actions) are used. And since regulations usually express state (restoration) changes one often uses state changing notations (such as found in Allard [114], B or event-B [1], RSL [96], VDM-SL [61, 62, 89], and Z [183]). In some cases it may be relevant to model using some constraint satisfaction notation [3] or some Fuzzy Logic notations [176].

2.5 Scripts

- By a **domain script** we shall understand the structured, almost, if not outright, formally expressed, wording of a procedure on how to proceed, one that has legally binding power, that is, which may be contested in a court of law \odot

2.5.1 Conceptual Analysis

Rules & regulations are usually expressed, even when informally so, as predicates. Scripts, in their procedural form, are like instructions, as for an algorithm.

Example. 10 A Casually Described Bank Script: Our formulation amounts to just a (casual) rough sketch. It is followed by a series of four large examples. Each of these elaborate on the theme of (bank) scripts. The problem area is that of how repayments of mortgage loans are to be calculated. At any one time a mortgage loan has a balance, a most recent previous date of repayment, an interest rate and a handling fee. When a repayment occurs, then the following calculations shall take place: (i) the interest on the balance of the loan since the most recent repayment, (ii) the handling fee, normally considered fixed, (iii) the effective repayment — being the difference between the repayment and the sum of the interest and the handling fee — and the new balance, being the difference between the old balance and the effective repayment. We assume repayments to occur from a designated account, say a demand/deposit account. We assume that bank to have designated fee and interest income accounts. (i) The interest is subtracted from the mortgage holder’s demand/deposit account and added to the bank’s interest (income) account. (ii) The handling fee is subtracted from the mortgage holder’s demand/deposit account and added to the bank’s fee (income) account. (iii) The effective repayment is subtracted from the mortgage holder’s demand/deposit account and also from the mortgage balance. Finally, one must also describe deviations such as overdue repayments, too large, or too small repayments, and so on. \square

Example. 11 A Formally Described Bank Script: First we must informally and formally define the bank state: There are clients ($c:C$), account numbers ($a:A$), mortgage numbers ($m:M$), account yields ($ay:AY$) and mortgage interest rates ($mi:MI$). The bank registers, by client, all accounts ($\rho:A_Register$) and all mortgages ($\mu:M_Register$). To each account number there is a balance ($\alpha:Accounts$). To each mortgage number there is a loan ($\ell:Loans$). To each loan is attached the last date that interest was paid on the loan.

value

$r, r': \text{Real axiom ...}$

type

C, A, M, Date

$AY' = \text{Real}, AY = \{ | ay:AY' \cdot 0 < ay \leq r | \}$

$MI' = \text{Real}, MI = \{ | mi:MI' \cdot 0 < mi \leq r' | \}$

$Bank' = A_Register \times Accounts \times M_Register \times Loans$

$Bank = \{ | \beta:Bank' \cdot wf_Bank(\beta) | \}$

$A_Register = C \xrightarrow{\#} A\text{-set}$

$Accounts = A \xrightarrow{\#} \text{Balance}$

```

M_Register = C  $\rightarrow_m$  M-set
Loans = M  $\rightarrow_m$  (Loan  $\times$  Date)
Loan, Balance = P
P = Nat

```

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

```

value
  ay:AY, mi:MI
  wf_Bank: Bank  $\rightarrow$  Bool
  wf_Bank( $\rho, \alpha, \mu, \ell$ )  $\equiv \cup \text{rng } \rho = \text{dom } \alpha \wedge \cup \text{rng } \mu = \text{dom } \ell$ 
axiom
  ay < mi [  $\wedge \dots$  ]

```

We — perhaps too rigidly — assume that mortgage interest rates are higher than demand/deposit account interest rates: $\text{ay} < \text{mi}$. Operations on banks are denoted by the commands of the bank script language. First the syntax:

```

type
  Cmd = OpA | CloA | Dep | Wdr | OpM | CloM | Pay
  OpA == mkOA(c:C)
  CloA == mkCA(c:C, a:A)
  Dep == mkD(c:C, a:A, p:P)
  Wdr == mkW(c:C, a:A, p:P)
  OpM == mkOM(c:C, p:P)
  Pay == mkPM(c:C, a:A, m:M, p:P, d:Date)
  CloM == mkCM(c:C, m:M, p:P)
  Reply = A | M | P | OkNok
  OkNok == ok | notok
value
  period: Date  $\times$  Date  $\rightarrow$  Days [for calculating interest]
  before: Date  $\times$  Date  $\rightarrow$  Bool [first date is earlier than last date]

```

And then the semantics:

```

int_Cmd(mkPM(c, a, m, p, d))( $\rho, \alpha, \mu, \ell$ )  $\equiv$ 
  let (b, d') =  $\ell(m)$  in
    if  $\alpha(a) \geq p$ 
      then
        let i = interest(mi, b, period(d, d')),
               $\ell' = \ell \upharpoonright [m \mapsto \ell(m) - (p - i)]$ ,
               $\alpha' = \alpha \upharpoonright [a \mapsto \alpha(a) - p, a_i \mapsto \alpha(a_i) + i]$  in
          (( $\rho, \alpha', \mu, \ell'$ ), ok) end
      else
          (( $\rho, \alpha', \mu, \ell$ ), nok)
      end end
  pre  $c \in \text{dom } \mu \wedge a \in \text{dom } \alpha \wedge m \in \mu(c)$ 
  post before(d, d')

```

interest: MI \times Loan \times Days \rightarrow P

□

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

2.5.2 Requirements

Script requirements call for the possibly interactive computerisation of algorithms, that is, for rather classical computing problems. But sometimes these scripts can be expressed, computably, in the form of programs in a domain specific language. As an example we refer to [76]. [76] illustrates how the design of pension and life insurance products, and their administration, reserve calculations, and audit, can be based on a common formal notation. The notation is human-readable and machine-processable, and specialised to the actuarial domain, achieving great expressive power combined with ease of use and safety. More specifically (a) product definitions based on standard actuarial models, including arbitrary continuous-time Markov and semi-Markov models, with cyclic transitions permitted; (b) calculation descriptions for reserves and other quantities of interest, based on differential equations; and (c) administration rules.

2.5.3 On Modeling Scripts

Scripts (as are licenses) are like programs (respectively like prescriptions program executions). Hence the full variety of techniques and notations for modeling programming (or specification) languages apply [13, 102, 158, 165, 173, 182]. [25, Chaps. 6–9] cover pragmatics, semantics and syntax techniques for defining functional, imperative and concurrent programming languages.

2.6 License Languages

License: a right or permission
granted in accordance with law
by a competent authority
to engage in some business or occupation,
to do some act, or to engage in some transaction
which but for such license would be unlawful ☺

Merriam Webster Online [136]

2.6.1 Conceptual Analysis

The Settings

A special form of scripts are increasingly appearing in some domains, notably the domain of electronic, or digital media. Here licenses express that a licensor, o , permits a licensee, u , to *render* (i.e., play) works of proprietary nature CD ROM-like music, DVD-like movies, etc. while obligating the licensee to pay the licensor on behalf of the owners of these, usually artistic works. Classical digital rights license languages, [15, 5, 73, 74, 75, 112, 71, 101, 104, 130, 140, 138, 131, 123, 163, 154, 153, 2, 141] applied to the electronic “downloading”, payment and rendering (playing) of artistic works (for example music, literature readings and movies). In this we generalise such applications languages and we extend the concept of licensing to also cover work authorisation (work commitment and promises) in health care, public government and schedule transport. The digital works for these new application domains are patient medical records, public government documents and bus/train/aircraft transport contracts. Digital rights licensing for artistic works seeks to safeguard against piracy and to ensure proper payments for the rights to render these works. Health care and public government license languages seek to ensure transparent and professional (accurate and timely) health care, respectively ‘good governance’. Transport contract languages seeks to ensure timely and reliable transport services by an evolving set of transport companies. Proper mathematical definition of licensing languages seeks to ensure smooth and correct computerised management of licenses and contracts.

On Licenses

The concepts of licenses and licensing express relations between (i) *actors* (licensors (the authority) and licensees), (ii) *entities* (artistic works, hospital patients, public administration, citizen documents) and bus transport contracts and (iii) *functions* (on entities), and as performed by actors. By issuing a license to a licensee, a licensor wishes to express and enforce certain permissions and obligations: which functions on which entities the licensee is allowed (is licensed, is permitted) to perform. In this we shall consider four kinds of entities: (i) digital recordings of artistic and intellectual nature: music, movies, readings (“audio books”), and the like, (ii) patients in a hospital as represented also by their patient medical records, (iii) documents related to public government, and (iv) transport vehicles, time tables and transport nets (of a buses, trains and aircraft).

Permissions and Obligations

The *permissions* and *obligations* issues are, (1) for the owner (agent) of some intellectual property to be paid (an *obligation*) by users when they perform *permitted* operations (rendering, copying, editing, sub-licensing) on their works; (2) for the patient to be professionally treated — by medical staff who are basically *obliged* to try to cure the patient; (3) for public administrators and citizens to enjoy good governance: transparency in law making (national parliaments and local prefectures and city councils), in law enforcement (i.e., the daily administration of laws), and law interpretation (the judiciary) — by agents who are basically *obliged* to produce certain documents while being *permitted* to consult (i.e., read, perhaps copy) other documents; and (4) for bus passengers to enjoy reliable bus schedules — offered by bus transport companies on contract to, say public transport authorities and on sub-contract to other such bus transport companies where these transport companies are *obliged* to honour a contracted schedule.

2.6.2 The Pragmatics

By pragmatics we understand the study and practice of the factors that govern our choice of language in social interaction and the effects of our choice on others.

In this section we shall rough-sketch-describe pragmatic aspects of the four domains of (1) production, distribution and consumption of artistic works, (2) the hospitalisation of patient, i.e., hospital health care, (3) the handling of law-based document in public government and (4) the operational management of schedule transport vehicles. The emphasis is on the pragmatics of the terms, i.e., the language used in these four domains.

Digital Media

Example. 12 Digital Media: The intrinsic entities of the performing arts are the artistic works: drama or opera performances, music performances, readings of poems, short stories, novels, or jokes, movies, documentaries, newsreels, etc. We shall limit our span to the scope of electronic renditions of these artistic works: videos, CDs or other. In this paper we shall not touch upon the technical issues of “downloading”(whether “streaming” or copying, or other). That and other issues should be analysed in [185].

Operations on Digital Works

For a consumer to be able to enjoy these works that consumer must (normally first) usually “buy a ticket” to their performances. The consumer, i.e., the theatre, opera, concert, etc., “goer” (usually) cannot copy the performance (e.g., “tape it”), let alone edit such copies of performances. In the context of electronic, i.e., digital renditions of these performances the above “cannots” take on a new meaning. The consumer may copy digital recordings, may edit these, and may further pass on such copies or editions to others. To do so, while protecting the rights of the producers (owners, performers), the consumer requests permission to

have the digital works transferred (“downloaded”) from the owner/producer to the consumer, so that the consumer can render (“play”) these works on own rendering devices (CD, DVD, etc., players), possibly can copy all or parts of them, then possibly can edit all or parts of the copies, and, finally, possibly can further license these “edited” versions to other consumers subject to payments to “original” licensor.

License Agreement and Obligation

To be able to obtain these permissions the user agrees with the wording of some license and pays for the rights to operate on the digital works.

Two Assumptions

Two, related assumptions underlie the pragmatics of the electronics of the artistic works. The first assumption is that the format, the electronic representation of the artistic works is proprietary, that is, that the producer still owns that format. Either the format is publicly known or it is not, that is, it is somehow “secret”. In either case we “derive” the second assumption (from the fulfillment of the first). The second assumption is that the consumer is not allowed to, or cannot operate¹¹ on the works by own means (software, machines). The second assumption implies that acceptance of a license results in the consumer receiving software that supports the consumer in performing all operations on licensed works, their copies and edited versions: rendering, copying, editing and sub-licensing.

Protection of the Artistic Electronic Works

The issue now is: how to protect the intellectual property (i.e., artistic) and financial (exploitation) rights of the owners of the possibly rendered, copied and edited works, both when, and when not further distributed.

□

Health-care

Example. 13 Health-care: Citizens go to hospitals in order to be treated for some calamity (disease or other), and by doing so these citizens become patients. At hospitals patients, in a sense, issue a request to be treated with the aim of full or partial restitution. This request is directed at medical staff, that is, the patient authorises medical staff to perform a set of actions upon the patient. One could claim, as we shall, that the patient issues a license.

Patients and Patient Medical Records

So patients and their attendant patient medical records (PMRs) are the main entities, the “works” of this domain. We shall treat them synonymously: PMRs as surrogates for patients. Typical actions on patients — and hence on PMRs — involve admitting patients, interviewing patients, analysing patients, diagnosing patients, planning treatment for patients, actually treating patients, and, under normal circumstance, to finally release patients.

Medical Staff

Medical staff may request (‘refer’ to) other medical staff to perform some of these actions. One can conceive of describing action sequences (and ‘referrals’) in the form of hospitalisation (not treatment) plans. We shall call such scripts for licenses.

Professional Health Care

The issue is now, given that we record these licenses, their being issued and being honoured, whether the handling of patients at hospitals follow, or does not follow properly issued licenses. □

¹¹ render, copy and edit

Government Documents

Example. 14 Documents: By public government we shall, following Charles de Secondat, baron de Montesquieu (1689–1755)¹², understand a composition of three powers: the law-making (legislative), the law-enforcing and the law-interpreting parts of public government. Typically national parliament and local (province and city) councils are part of law-making government. Law-enforcing government is called the executive (the administration). And law-interpreting government is called the judiciary [system] (including lawyers etc.).

Documents

A crucial means of expressing public administration is through *documents*.¹³ We shall therefore provide a brief domain analysis of a concept of documents. (This document domain description also applies to patient medical records and, by some “light” interpretation, also to artistic works — insofar as they also are documents.) Documents are *created*, *edited* and *read*; and documents can be *copied*, *distributed*, the subject of *calculations* (interpretations) and be *shared* and *shredded*.

Document Attributes

With documents one can associate, as attributes of documents, the *actors* who created, edited, read, copied, distributed (and to whom distributed), shared, performed calculations and shredded documents. With these operations on documents, and hence as attributes of documents one can, again conceptually, associate the *location* and *time* of these operations.

Actor Attributes and Licenses

With actors (whether agents of public government or citizens) one can associate the *authority* (i.e., the *rights*) these actors have with respect to performing actions on documents. We now intend to express these *authorisations as licenses*.

Document Tracing

An issue of public government is whether citizens and agents of public government act in accordance with the laws — with actions and laws reflected in documents such that the action documents enables a trace from the actions to the laws “governing” these actions. We shall therefore assume that every document can be traced back to its law-origin as well as to all the documents any one document-creation or -editing was based on. □

Transportation

Example. 15 Passenger and Goods Transport:

A Synopsis

Contracts obligate transport companies to deliver bus traffic according to a timetable. The timetable is part of the contract. A contractor may sub-contract (other) transport companies to deliver bus traffic according to timetables that are sub-parts of their own timetable. Contractors are either public transport authorities or contracted transport companies. Contracted transport companies may cancel a subset of bus rides provided the total amount of cancellations per 24 hours for each bus line does not exceed a contracted upper limit. The cancellation rights are spelled out in the contract. A sub-contractor cannot increase a contracted upper limit for cancellations above what the sub-contractor was told (in its contract) by its contractor. Etcetera.

¹² *De l'esprit des lois* (*The Spirit of the Laws*), published 1748

¹³ Documents are, for the case of public government to be the “equivalent” of artistic works.

A Pragmatics and Semantics Analysis

The “works” of the bus transport contracts are two: the timetables and, implicitly, the designated (and obligated) bus traffic. A bus timetable appears to define one or more bus lines, with each bus line giving rise to one or more bus rides. Nothing is (otherwise) said about regularity of bus rides. It appears that bus ride cancellations must be reported back to the contractor. And we assume that cancellations by a sub-contractor is further reported back also to the sub-contractor’s contractor. Hence eventually that the public transport authority is notified. Nothing is said, in the contracts, such as we shall model them, about passenger fees for bus rides nor of percentages of profits (i.e., royalties) to be paid back from a sub-contractor to the contractor. So we shall not bother, in this example, about transport costs nor transport subsidies. But will leave that necessary aspect as an exercise. The opposite of cancellations appears to be ‘insertion’ of extra bus rides, that is, bus rides not listed in the time table, but, perhaps, mandated by special events¹⁴ We assume that such insertions must also be reported back to the contractor. We assume concepts of acceptable and unacceptable bus ride delays. Details of delay acceptability may be given in contracts, but we ignore further descriptions of delay acceptability, but assume that unacceptable bus ride delays are also to be (iteratively) reported back to contractors. We finally assume that sub-contractors cannot (otherwise) change timetables. (A timetable change can only occur after, or at, the expiration of a license.) Thus we find that contracts have definite period of validity. (Expired contracts may be replaced by new contracts, possibly with new timetables.)

Contracted Operations, An Overview

The actions that may be granted by a contractor according to a contract are: (i) *start*: to commence, i.e., to start, a bus ride (obligated); (ii) *end*: to conclude a bus ride (obligated); (iii) *cancel*: to cancel a bus ride (allowed, with restrictions); (iv) *insert*: to insert a bus ride; and (v) *subcontract*: to sub-contract part or all of a contract. □

2.6.3 Schematic Rendition of License Language Constructs

There are basically two aspects to licensing languages: (i) the [actual] licensing [and sub-licensing], in the form of licenses, ℓ , by licensors, o , of permissions and thereby implied obligations, and (ii) the carrying-out of these obligations in the form of licensee, u , actions. We shall in this paper treat licensors and licensees on par, that is, some os are also us and vice versa. And we shall think of licenses as not necessarily material entities (e.g., paper documents), but allow licenses to be tacitly established (understood).

Licensing

The granting of a license ℓ by a licensor o , to a set of licensees $u_{u_1}, u_{u_2}, \dots, u_{u_u}$ in which ℓ expresses that these may perform actions $a_{a_1}, a_{a_2}, \dots, a_{a_a}$ on work items $e_{e_1}, e_{e_2}, \dots, e_{e_e}$ can be schematised:

ℓ : **licensor** o **contracts** licensees $\{u_{u_1}, u_{u_2}, \dots, u_{u_u}\}$
 to perform actions $\{a_{a_1}, a_{a_2}, \dots, a_{a_a}\}$ **on work items** $\{e_{e_1}, e_{e_2}, \dots, e_{e_e}\}$
 allowing sub-licensing of actions $\{a_{a_i}, a_{a_j}, \dots, a_{a_k}\}$ **to** $\{u_{u_x}, u_{u_y}, \dots, u_{u_z}\}$

The two sets of action designators, $das : \{a_{a_1}, a_{a_2}, \dots, a_{a_a}\}$ and $sas : \{a_{a_x}, a_{a_y}, \dots, a_{a_z}\}$ need not relate. **Sub-licensing**: Line 3 of the above schema, ℓ , expresses that licensees $u_{u_1}, u_{u_2}, \dots, u_{u_u}$, may act as licensors and (thereby sub-)license ℓ to licensees $us : \{u_{u_x}, u_{u_y}, \dots, u_{u_z}\}$, distinct from $sus : \{u_{u_1}, u_{u_2}, \dots, u_{u_u}\}$, that is, $us \cap sus = \{\}$. **Variants**: One can easily “cook up” any number of variations of the above license schema. **Revoke Licenses**: We do not show expressions for revoking part or all of a previously granted license.

Licensors and Licensees

Example. 16 Licensors and Licensees:

¹⁴ Special events: breakdown (that is, cancellations) of other bus rides, sports event (soccer matches), etc.

Digital Media

: For digital media the original licensors are the original producers of music, film, etc. The “original” licensees are you and me ! Thereafter some of us may become licensors, etc.

Health-care

: For health-care the original licensors are, say in Denmark, the Danish governments’ National Board of Health¹⁵; and the “original” licensees are the national hospitals. These then sub-license their medical clinics (rheumatology, cancer, urology, gynecology, orthopedics, neurology, etc.) which again sub-licenses their medical staff (doctors, nurses, etc.). A medical doctor may, as is the case in Denmark for certain actions, not [necessarily] perform these but may sub-license their execution to nurses, etc.

Documents

: For government documents the original licensor are the (i) heads of parliament, regional and local governments, (ii) government (prime minister) and the heads of respective ministries, respectively the regional and local agencies and administrations. The “original” licensees are (i’) the members of parliament, regional and local councils charged with drafting laws, rules and regulations, (ii’) the ministry, respectively the regional and local agency department heads. These (the ‘s) then become licensors when licensing their staff to handle specific documents.

Transport

: For scheduled passenger (etc.) transportation the original licensors are the state, regional and/or local transport authorities. The “original” licensees are the public and private transport firms. These latter then become licensors licensing drivers to handle specific transport lines and/or vehicles. \square

Actors and Actions

Example. 17 Actors and Actions:

Digital Media: w refers to a digital “work” with w' designating a newly created one; s_i refers to a sector of some work. **render** $w(s_i, s_j, \dots, s_k)$: sectors s_i, s_j, \dots, s_k of work w are rendered (played, visualised) in that order. $w' := \text{copy } w(s_i, s_j, \dots, s_k)$: sectors s_i, s_j, \dots, s_k of work w are copied and becomes work w' . $w' := \text{edit } w \text{ with } \mathcal{E}(w_\alpha(s_a, s_b, \dots, s_c), \dots, w_\gamma(s_p, s_q, \dots, s_r))$: work w is edited while [also] incorporating references to or excerpts from [other] works $w_\alpha(s_a, s_b, \dots, s_c), \dots, w_\gamma(s_p, s_q, \dots, s_r)$. **read** w : work w is read, i.e., information about work w is somehow displayed. ℓ : **licensor m contracts licensees** $\{u_{u_1}, u_{u_2}, \dots, u_{u_n}\}$ **to perform actions** $\{\text{RENDER, COPY, EDIT, READ}\}$ **on work items** $\{w_{i_1}, w_{i_2}, \dots, w_{i_w}\}$. Etcetera: other forms of actions can be thought of.

Health-care: Actors are here limited to the patients and the medical staff. We refer to Fig. 2.3 on the next page. It shows an archetypal hospitalisation plan and identifies a number of actions; π designates patients, t designates treatment (medication, surgery, ...). Actions are performed by medical staff, say h , with h being an implicit argument of the actions. **interview** π : a PMR with name, age, family relations, addresses, etc., is established for patient π . **admit** π : the PMR records the anamnese (medical history) for patient π . **establish analysis plan** π : the PMR records which analyses (blood tests, ECG, blood pressure, etc.) are to be carried out. **analyse** π : the PMR records the results of the analyses referred to previously. **diagnose** π : medical staff h diagnoses, based on the analyses most recently performed. **plan treatment for** π : medical staff h sets up a treatment plan for patient π based on the diagnosis most recently performed. **treat** π **wrt.** t : medical staff h performs treatment t on patient π , observes “reaction” and records this in the PMR. Predicate “actions”: **more analysis** $\pi ?$, **more treatment** $\pi ?$ and **more diagnosis** $\pi ?$. **release** π : either the patient dies or is declared ready to be sent ‘home’. ℓ : **licensor o contracts medical staff** $\{m_{m_1}, m_{m_2}, \dots, m_{m_m}\}$ **to perform actions** $\{\text{INTERVIEW, ADMIT, PLAN ANALYSIS, ANALYSE, DIAGNOSE, PLAN TREATMENT, TREAT, RELEASE}\}$ **on patients** $\{\pi_{p_1}, \pi_{p_2}, \dots, \pi_{p_p}\}$. Etcetera: other forms of actions can be thought of.

¹⁵ In the UK: the NHS, etc.

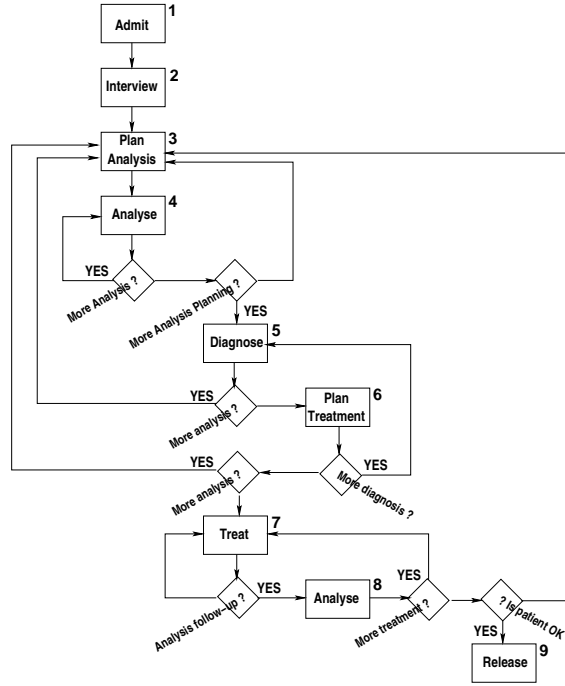


Fig. 2.3. An example single-illness non-fatal hospitalisation plan. States: $\{1,2,3,4,5,6,7,8,9\}$

Documents: d refer to documents with d' designating new documents. $d' := \text{create based on } d_x, d_y, \dots, d_z$: A new document, named d' , is created, with no information “contents”, but referring to existing documents d_x, d_y, \dots, d_z . **edit d with \mathcal{E} based on $d_{n\alpha}, d_\beta, \dots, d_\gamma$** : document d is edited with \mathcal{E} being the editing function and \mathcal{E}^{-1} being its “undo” inverse. **read d** : document d is being read. $d' := \text{copy } d$: document d is copied into a new document named d' . **freeze d** : document d can, from now on, only be read. **shred d** : document d is shredded. That is, no more actions can be performed on d . ℓ : **licensor o contracts civil service staff $\{c_{c1}, c_{c2}, \dots, c_{cc}\}$ to perform actions $\{\text{CREATE, EDIT, READ, COPY, FREEZE, SHRED}\}$ on documents $\{d_{d1}, d_{d2}, \dots, d_{dd}\}$** . Etcetera: other forms of actions can be thought of.

Transport: We restrict, without loss of generality, to bus transport. There is a timetable, tt . It records bus lines, l , and specific instances of bus rides, b . **start bus ride l, b at time t** : Bus line l is recorded in tt and its departure in tt is recorded as τ . Starting that bus ride at t means that the start is either on time, i.e., $t=\tau$, or the start is delayed $\delta_d : \tau-t$ or advanced $\delta_a : t-\tau$ where δ_d and δ_a are expected to be small intervals. All this is to be reported, in due time, to the contractor. **end bus ride l, b at time t** : Ending bus ride l, b at time t means that it is either ended on time, or earlier, or delayed. This is to be reported, in due time, to the contractor. **cancel bus ride l, b at time t** : t must be earlier than the scheduled departure of bus ride l, b . **insert an extra bus l, b' at time t** : t must be the same time as the scheduled departure of bus ride l, b with b' being a “marked” version of b . ℓ : **licensor o contracts transport staff $\{b_{b1}, b_{b2}, \dots, b_{bb}\}$ to perform actions $\{\text{START, END, CANCEL, INSERT}\}$ on work items $\{e_{e1}, e_{e2}, \dots, e_{ee}\}$** . Etcetera: other forms of actions can be thought of. \square

2.6.4 Requirements

Requirements for license language implementation basically amounts to requirements for three aspects. (i) The design of the license language, its abstract and concrete syntax, its interpreter, and its interfaces to distributed licensor and licensee behaviours; (ii) the requirements for a distributed system of licensor and licensee behaviours; and (iii) the monitoring and partial control of the states of licensor and licensee behaviours. The structuring of these distributed licensor and licensee behaviours differ from slightly to somewhat, but not that significant in the four license languages examples. Basically the licensor and licensee behaviours form a set of behaviours. Basically everyone can communicate with everyone. For the

case of digital media licensee behaviours communicate back to licensor behaviours whenever a properly licensed action is performed – resulting in the transfer of funds from licensees to licensors. For the case of health care some central authority is expected to validate the granting of licenses and appear to be bound by medical training. For the case of documents such checks appear to be bound by predetermined authorisation rules. For the case of transport one can perhaps speak of more rigid management & organisation dependencies as licenses are traditionally transferred between independent authorities and companies.

2.6.5 On Modeling License Languages

Licensors are expected to maintain a state which records all the licenses it has issued. Whenever a licensee “reports back” (the begin and/or the end) of the performance of a granted action, this is recorded in its state. Sometimes these granted actions are subject to fees. The licensor therefore calculates outstanding fees — etc. Licensees are expected to maintain a state which records all the licenses it has accepted. Whenever an action is to be performed the licensee records this and checks that it is permitted to perform this action. In many cases the licensee is expected to “report back”, both the beginning and the end of performance of that action, to the licensor. A typical technique of modeling licensors, licensees and patients, i.e., their PMRs, is to model them as (never ending) processes, a la CSP [111] with input/output, channels, communications between licensors, licensees and PMRs. Their states are modeled as programmable attributes.

2.7 Management &¹⁶ Organisation

- By **domain management** we shall understand such people (such decisions) (i) who (which) determine, formulate and thus set standards (cf. rules and regulations, Sect. 2.4) concerning strategic, tactical and operational decisions; (ii) who ensure that these decisions are passed on to (lower) levels of management and to floor staff; (iii) who make sure that such orders, as they were, are indeed carried out; (iv) who handle undesirable deviations in the carrying out of these orders cum decisions; and (v) who “backstop” complaints from lower management levels and from “floor” staff ☉
- By **domain organisation** we shall understand (vi) the structuring of management and non-management staff “overseeable” into clusters with “tight” and “meaningful” relations; (vii) the allocation of strategic, tactical and operational concerns to within management and non-management staff clusters; and hence (viii) the “lines of command”: who does what, and who reports to whom, administratively and functionally ☉

The ‘&’ is justified from the interrelations of items (i–viii).

2.7.1 Conceptual Analysis

We first bring some examples.

Example. 18 Train Monitoring, I: In China, as an example, till the early 1990s, rescheduling of trains occurs at stations and involves telephone negotiations with neighbouring stations (“up and down the lines”). Such rescheduling negotiations, by phone, imply reasonably strict management and organisation (M&O). This kind of M&O reflects the geographical layout of the rail net. □

Example. 19 Railway Management and Organisation: Train Monitoring, II: We single out a rather special case of railway management and organisation. Certain (lowest-level operational and station-located) supervisors are responsible for the day-to-day timely progress of trains within a station and along its incoming and outgoing lines, and according to given timetables. These supervisors and their immediate (middle-level) managers (see below for regional managers) set guidelines (for local station and incoming and outgoing lines) for the monitoring of train traffic, and for controlling trains that are either ahead of or

¹⁶ See footnote 10 on Page 61.

behind their schedules. By an incoming and an outgoing line we mean part of a line between two stations, the remaining part being handled by neighbouring station management. Once it has been decided, by such a manager, that a train is not following its schedule, based on information monitored by non-management staff, then that manager directs that staff: (i) to suggest a new schedule for the train in question, as well as for possibly affected other trains, (ii) to negotiate the new schedule with appropriate neighbouring stations, until a proper reschedule can be decided upon, by the managers at respective stations, (iii) and to enact that new schedule.¹⁷ A (middle-level operations) manager for regional traffic, i.e., train traffic involving several stations and lines, resolves possible disputes and conflicts. \square

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

People staff enterprises, the components of infrastructures with which we are concerned, i.e., for which we develop software. The larger these enterprises — these infrastructure components — the more need there is for management and organisation. The role of management is roughly, for our purposes, twofold: first, to perform strategic, tactical and operational work, to set strategic, tactical and operational policies — and to see to it that they are followed. The role of management is, second, to react to adverse conditions, that is, to unforeseen situations, and to decide how they should be handled, i.e., conflict resolution. Policy setting should help non-management staff operate normal situations — those for which no management interference is thus needed. And management “backstops” problems: management takes these problems off the shoulders of non-management staff. To help management and staff know who’s in charge wrt. policy setting and problem handling, a clear conception of the overall organisation is needed. Organisation defines lines of communication within management and staff, and between these. Whenever management and staff has to turn to others for assistance they usually, in a reasonably well-functioning enterprise, follow the command line: the paths of organigrams — the usually hierarchical box and arrow/line diagrams.

The *management and organisation* model of a domain is a partial specification; hence all the usual abstraction and modeling principles, techniques and tools apply. More specifically, management is a set of predicate functions, or of observer and generator functions. These either parametrise other, the operations functions, that is, determine their behaviour, or yield results that become arguments to these other functions. Organisation is thus a set of constraints on communication behaviours. Hierarchical, rather than linear, and matrix structured organisations can also be modeled as sets (of recursively invoked sets) of equations.

To relate classical organigrams to formal descriptions we first show such an organigram (Fig. 2.4), and then we show schematic processes which — for a rather simple scenario — model managers and the managed! Based on such a diagram, and modeling only one neighbouring group of a manager and the staff working for that manager we get a system in which one manager, mgr, and many staff, stf, coexist or work concurrently, i.e., in parallel. The mgr operates in a context and a state modeled by ψ . Each staff, stf(i) operates in a context and a state modeled by $s\sigma(i)$.

type

Msg, Ψ , Σ , Sx
 $S\Sigma = Sx \rightarrow_{\#} \Sigma$

channel

$\{ ms[i]:Msg \mid i:Sx \}$

value

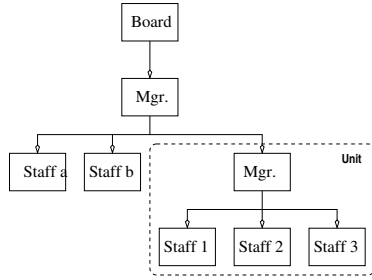
$s\sigma:S\Sigma$, $\psi:\Psi$

$sys: Unit \rightarrow Unit$

$sys() \equiv \parallel \{ stf(i)(s\sigma(i)) \mid i:Sx \} \parallel mgr(\psi)$

¹⁷ That enactment may possibly imply the movement of several trains incident upon several stations: the one at which the manager is located, as well as possibly at neighbouring stations.

A Hierarchical Organisation



A Matrix Organisation

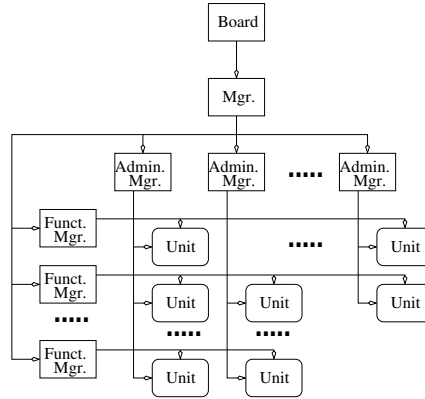


Fig. 2.4. Organisational structures

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

$$\begin{aligned} & \text{mgr: } \Psi \rightarrow \text{in,out } \{ \text{ms}[i] | i:Sx \} \text{ Unit} \\ & \text{mgr}(\psi) \equiv \\ (1) & \text{ let } (\psi', m) = \text{m_out}(\psi) \text{ in } \square \{ \text{ms}[i]!m | i:Sx \}; \text{mgr}(\psi') \text{ end} \\ & \square \\ (2) & \text{ let } \psi' = \square \{ \text{let } m = \text{ms}[i]? \text{ in } \text{m_in}(i, m)(\psi) \text{ end} | i:Sx \} \text{ in } \text{mgr}(\psi') \text{ end} \end{aligned}$$

$$\begin{aligned} & \text{m_out: } \Psi \rightarrow \Psi \times \text{MSG}, \\ & \text{m_in: } Sx \times \text{MSG} \rightarrow \Psi \rightarrow \Psi \end{aligned}$$

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

$$\begin{aligned} & \text{stf: } i:Sx \rightarrow \Sigma \rightarrow \text{in,out } \text{ms}[i] \text{ Unit} \\ & \text{stf}(i)(\sigma) \equiv \\ (1) & \text{ let } m = \text{ms}[i]? \text{ in } \text{stf}(i)(\text{st_in}(m)(\sigma)) \text{ end} \\ & \square \\ (2) & \text{ let } (\sigma', m) = \text{st_out}(\sigma) \text{ in } \text{ms}[i]!m; \text{stf}(i)(\sigma') \text{ end} \end{aligned}$$

$$\begin{aligned} & \text{st_in: } \text{MSG} \rightarrow \Sigma \rightarrow \Sigma, \\ & \text{st_out: } \Sigma \rightarrow \Sigma \times \text{MSG} \end{aligned}$$

Both manager and staff processes recurse (i.e., iterate) over possibly changing states. The management process non-deterministically, internal choice, “alternates” between “broadcast”-issuing orders to staff and receiving individual messages from staff. Staff processes likewise non-deterministically, internal choice, alternate between receiving orders from management and issuing individual messages to management. The conceptual example also illustrates modeling stakeholder behaviours as interacting (here CSP-like) processes.

Example. 20 Strategic, Tactical and Operations Management: We think of (i) strategic, (ii) tactic, and (iii) operational managers as well as (iv) supervisors, (v) team leaders and the rest of the (vi) staff (i.e., workers) of a domain enterprise as functions. Each category of staff, i.e., each function, works in state and updates that state according to schedules and resource allocations — which are considered part of the state. To make the description simple we do not detail the state other than saying that each category works on an “instantaneous copy” of “the” state. Now think of six staff category activities, strategic managers, tactical managers, operational managers, supervisors, team leaders and workers as six simultaneous sets of actions. Each function defines a step of collective (i.e., group) (strategic, tactical, operational) management, supervisor, team leader and worker work. Each step is considered “atomic”. Now think of an enterprise as the “repeated” step-wise simultaneous performance of these category activities. Six “next” states arise. These are, in the reality of the domain, ameliorated, that is reconciled into one state. however with the next iteration, i.e., step, of work having each category apply its work to a reconciled version of the state resulting from that category’s previously yielded state and the mediated “global” state. Caveat: The below is not a mathematically proper definition. It suggests one !

type

0. $\Sigma, \Sigma_s, \Sigma_t, \Sigma_o, \Sigma_u, \Sigma_e, \Sigma_w$

value

1. str, tac, opr, sup, tea, wrk: $\Sigma_i \rightarrow \Sigma_i$
2. stra, tact, oper, supr, team, work: $\Sigma \rightarrow (\Sigma_{x_1} \times \Sigma_{x_2} \times \Sigma_{x_3} \times \Sigma_{x_4} \times \Sigma_{x_5}) \rightarrow \Sigma$
3. objective: $(\Sigma_s \times \Sigma_t \times \Sigma_o \times \Sigma_u \times \Sigma_e \times \Sigma_w) \rightarrow \mathbf{Bool}$
3. enterprise, ameliorate: $(\Sigma_s \times \Sigma_t \times \Sigma_o \times \Sigma_u \times \Sigma_e \times \Sigma_w) \rightarrow \Sigma$
4. enterprise: $(\sigma_s, \sigma_t, \sigma_u, \sigma_e, \sigma_w) \equiv$
6. **let** $\sigma'_s = \text{stra}(\text{str}(\sigma_s))(\sigma'_t, \sigma'_o, \sigma'_u, \sigma'_e, \sigma'_w),$
7. $\sigma'_t = \text{tact}(\text{tac}(\sigma_t))(\sigma'_s, \sigma'_o, \sigma'_u, \sigma'_e, \sigma'_w),$
8. $\sigma'_o = \text{oper}(\text{opr}(\sigma_o))(\sigma'_s, \sigma'_t, \sigma'_u, \sigma'_e, \sigma'_w),$
9. $\sigma'_u = \text{supr}(\text{sup}(\sigma_u))(\sigma'_s, \sigma'_t, \sigma'_o, \sigma'_e, \sigma'_w),$
10. $\sigma'_e = \text{team}(\text{tea}(\sigma_e))(\sigma'_s, \sigma'_t, \sigma'_o, \sigma'_u, \sigma'_w),$
11. $\sigma'_w = \text{work}(\text{wrk}(\sigma_w))(\sigma'_s, \sigma'_t, \sigma'_o, \sigma'_u, \sigma'_e) \text{ in}$
12. **if** $\text{objective}(\sigma'_s, \sigma'_t, \sigma'_o, \sigma'_u, \sigma'_e, \sigma'_w)$
13. **then** $\text{ameliorate}(\sigma'_s, \sigma'_t, \sigma'_o, \sigma'_u, \sigma'_e, \sigma'_w)$
14. **else** $\text{enterprise}(\sigma'_s, \sigma'_t, \sigma'_o, \sigma'_u, \sigma'_e, \sigma'_w)$
15. **end end**

0. Σ is a further undefined and unexplained enterprise state space. The various enterprise players view this state in their own way.
1. Six staff group operations, str, tac, opr, sup, tea and wrk, each act in the enterprise state such as conceived by respective groups to effect a resulting enterprise state such as achieved by respective groups.
2. Six staff group state amelioration functions, ame_s, ame_t, ame_o, ame_u, ame_e and ame_w, each apply to the resulting enterprise states such as achieved by respective groups to yield a result state such as achieved by that group.
3. An overall objective function tests whether a state summary reflects that the objectives of the enterprise has been achieved or not.
4. The enterprise function applies to the tuple of six group-biased (i.e., ameliorated) states. Initially these may all be the same state. The result is an ameliorated state.
5. An iteration, that is, a step of enterprise activities, lines 5.–13. proceeds as follows:
6. strategic management operates
 - in its state space, $\sigma_s : \Sigma$;
 - effects a next (un-ameliorated strategic management) state σ'_s ;
 - and ameliorates this latter state in the context of all the other player’s ameliorated result states.
- 7.–11. The same actions take place, simultaneously for the other players: tac, opr, sup, tea and wrk.
12. A test, *has objectives been met*, is made on the six ameliorated states.
13. If test is successful, then the enterprise terminates in an ameliorated state.

14. Otherwise the enterprise recurses, that is, “repeats” itself in new states.

The above “function” definition is suggestive. It suggests that a solution to the fix-point 6-tuple of equations over “intermediate” states, σ'_x , where x is any of s, t, o, u, e, w , is achievable by iteration over just these 6 equations. \square

2.7.2 Requirements

Top-level, including strategic management tends to not be amenable to “automation”. Increasingly tactical management tends to “divide” time between “bush-fire, stop-gap” actions – hardly automatable and formulating, initiating and monitoring main operations. The initiation and monitoring of tactical actions appear amenable to partial automation. Operational management – with its reliance on rules & regulations, scripts and licenses – is where computer monitoring and partial control has reaped the richest harvests.

2.7.3 On Modeling Management and Organisation

Management and organisation basically spans entity, function, event and behaviour intensities and thus typically require the full spectrum of modeling techniques and notations — summarised in Sect. 2.2.3.

2.8 Human Behaviour

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

2.8.1 Conceptual Analysis

To model human behaviour “smacks” like modeling human actors, the psychology of humans, etc. ! We shall not attempt to model the psychological side of humans — for the simple reason that we neither know how to do that nor whether it can at all be done. Instead we shall be focusing on the effects on non-human manifest entities of human behaviour.

Example. 21 Banking — or Programming — Staff Behaviour: Let us assume a bank clerk, “in ye olde” days, when calculating, say mortgage repayments (cf. Example 10). We would characterise such a clerk as being *diligent*, etc., if that person carefully follows the mortgage calculation rules, and checks and double-checks that calculations “tally up”, or lets others do so. We would characterise a clerk as being *sloppy* if that person occasionally forgets the checks alluded to above. We would characterise a clerk as being *delinquent* if that person systematically forgets these checks. And we would call such a person a *criminal* if that person intentionally miscalculates in such a way that the bank (and/or the mortgage client) is cheated out of funds which, instead, may be diverted to the cheater. Let us, instead of a bank clerk, assume a software programmer charged with implementing an automatic routine for effecting mortgage repayments (cf. Example 11). We would characterise the programmer as being *diligent* if that person carefully follows the mortgage calculation rules, and throughout the development verifies and tests that the calculations are correct with respect to the rules. We would characterise the programmer as being *sloppy* if that person forgets certain checks and tests when otherwise correcting the computing program under development. We would characterise the programmer as being *delinquent* if that person systematically forgets these checks and tests. And we would characterise the programmer as being a *criminal* if that person intentionally provides a program which miscalculates the mortgage interest, etc., in such a way that the bank (and/or the mortgage client) is cheated out of funds. \square

Example. 22 A Human Behaviour Mortgage Calculation: Example 11 gave a semantics to the mortgage calculation request (i.e., command) as would a diligent bank clerk be expected to perform it. To express, that is, to model, how sloppy, delinquent, or outright criminal persons (staff?) could behave we must modify the $\text{int_Cmd}(\text{mkPM}(c, a, m, p, d'))(\rho, \alpha, \mu, \ell)$ definition.


```

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

```

$q: P \times P \rightarrow \text{Bool}$
 $f_1, f_2, f_3, f_4, f_{\text{"staff"}}: P \rightarrow P$ [typically: $f_{\text{"staff"}} = \lambda p.p$]

□

The predicate q and the functions f_1, f_2, f_3, f_4 and $f_{\text{"staff"}}$ of Example 22 are deliberately left undefined. They are being defined by the “staffer” when performing (incl., programming) the mortgage calculation routine. The point of Example 22 is that one must first define the mortgage calculation script precisely as one would like to see the diligent staff (programmer) to perform (incl., correctly program) it before one can “pinpoint” all the places where lack of diligence may “set in”. The invocations of q, f_1, f_2, f_3, f_4 and $f_{\text{"staff"}}$ designate those places. The point of Example 22 is also that we must first domain-define, “to the best of our ability” all the places where human behaviour may play other than a desirable role. If we cannot, then we cannot claim that some requirements aim at countering undesirable human behaviour.

Commensurate with the above, humans interpret rules and regulations differently, and, for some humans, not always consistently — in the sense of repeatedly applying the same interpretations. Our final specification pattern is therefore:

```

type
  Action = Θ → Θ-infset
value
  hum_int: Rule → Θ → RUL-infset
  action: Stimulus → Θ → Θ
  hum_beha: Stimulus × Rules → Action → Θ → Θ-infset
  hum_beha(sy_sti,sy_rul)(α)(θ) as θset
  post
    θset = α(θ) ∧ action(sy_sti)(θ) ∈ θset
    ∧ ∀ θ': Θ • θ' ∈ θset ⇒
      ∃ se_rul: RUL • se_rul ∈ hum_int(sy_rul)(θ) ⇒ se_rul(θ, θ')

```

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

2.8.2 Requirements

Requirements in relation to the human behaviour facet is not requirements about software that “replaces” human behaviour. Such requirements were hinted at in Sects. 2.5.2–2.7.2. Human behaviour facet requirements are about software that checks human behaviour; that it remains diligent; that it does not transgress

into sloppy, delinquent, let alone criminal behaviour. When transgressions are discovered, appropriate remedial actions may be prescribed.

2.8.3 On Modeling Human Behaviour

To model human behaviour is, “initially”, much like modeling management and organisation. But only ‘initially’. The most significant human behaviour modeling aspect is then that of modeling non-determinism and looseness, even ambiguity. So a specification language which allows specifying non-determinism and looseness (like CafeOBJ [92] and RSL [96]) is to be preferred. To prescribe requirements is to prescribe the monitoring of the human input at the computer interface.

2.9 Conclusion

We have introduced the scientific and engineering concept of domain theories and domain engineering; and we have brought but a mere sample of the principles, techniques and tools that can be used in creating domain descriptions.

2.9.1 Completion

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

2.9.2 Integrating Formal Descriptions

We have seen that to model the full spectrum of domain facets one needs not one, but several specification languages. No single specification language suffices. It seems highly unlikely and it appears not to be desirable to obtain a single, “universal” specification language capable of “equally” elegantly, suitably abstractly modeling all aspects of a domain. Hence one must conclude that the full modeling of domains shall deploy several formal notations – including plain, good old mathematics in all its forms. The issues are then the following which combinations of notations to select, and how to make sure that the combined specification denotes something meaningful. The ongoing series of “Integrating Formal Methods” conferences [4] is a good source for techniques, compositions and meanings.

2.9.3 The Impossibility of Describing Any Domain Completely

Domain descriptions are, by necessity, abstractions. One can never hope for any notion of complete domain descriptions. The situation is no better for domains such as we define them than for physics. Physicists strive to understand the manifest world around us – the world that was there before humans started creating “their domains”. The physicists describe the physical world “in bits and pieces” such that large collections of these pieces “fit together”, that is, are based on some commonly accepted laws and in some commonly agreed mathematics. Similarly for such domains as will be the subject of domain science & engineering such as we cover that subject in **Chapter 1**, [49] and **Chapter 6**, [54]. Individual such domain descriptions will be emphasising some clusters of facets, others will be emphasising other aspects.

2.9.4 Rôles for Domain Descriptions

We can distinguish between a spectrum of rôles for domain descriptions. Some of the issues brought forward below may have been touched upon in Chaps. 1 and 5 [49, 54].

Alternative Domain Descriptions: It may very well be meaningful to avail oneself of a variety of domain models (i.e., descriptions) for any one domain, that is, for what we may consider basically one and the same domain. In control theory (a science) and automation (an engineering) we develop specific descriptions, usually on the form of a set of differential equations, for any one control problem. The basis for the control problem is typically the science of mechanics. This science has many renditions (i.e., interpretations). For the control problem, say that of keeping a missile carried by a train wagon, erect during train movement and/or windy conditions, one may then develop a “self-contained” description of the problem based on some mechanics theory presentation. Similarly for domains. One may refer to an existing domain description. But one may re-develop a textually “smaller” domain description for any one given, i.e., specific problem.

Domain Science: A domain description designates a domain theory. That is, a bundle of propositions, lemmas and theorems that are either rather explicit or can be proven from the description. So a domain description is the basis for a theory as well as for the discovery of domain laws, that is, for a domain science. We have sciences of physics (incl. chemistry), biology, etc. Perhaps it is about time to have proper sciences, to the extent one can have such sciences for human-made domains.

Business Process Re-engineering: Some domains manifest serious amounts of human actions and interactions. These may be found to not be efficient to a degree that one might so desire. A given domain description may therefore be a basis for suggesting other *management & organisation* structures, and/or *rules & regulations* than present ones. Yes, even making explicit *scripts* or a *license language* which have hitherto been tacitly understood – without necessarily computerising any support for such a *script* or *license language*. The given and the resulting domain descriptions may then be the basis for operations research models that may show desired or acceptable efficiency improvements.

Software Development: Chapter 5 [54] shows one approach to requirements prescription. Domain analysis & description, i.e., domain engineering, is here seen as an initial phase, with requirements prescription engineering being a second phase, and software design being a third phase. We see domain engineering as indispensable, that is, an absolute must, for software development. Chapter 6 [38, *Domains: Their Simulation, Monitoring and Control*] further illustrates how domain engineering is a base for the development of domain simulators, demos, monitors and controllers.

2.9.5 Grand Challenges of Informatics¹⁹

To establish a reasonably trustworthy and believable theory of a domain, say the transportation, or just the railway domain, may take years, possibly 10–15! Similarly for domains such as the financial service industry, the market (of consumers and producers, retailers, wholesaler, distribution cum supply chain), health care, and so forth. The current author urges younger scientists to get going! It is about time.

2.10 Bibliographical Notes

To create domain descriptions, or requirements prescriptions, or software designs, properly, at least such as this author sees it, is a joy to behold. The beauty of carefully selected and balanced abstractions, their interplay with other such, the relations between phases, stages and steps, and many more conceptual constructions make software engineering possibly the most challenging intellectual pursuit today. For this and more consult [24, 25, 26].

¹⁹ In the early-to-mid 2000s there were a rush of research foundations and scientists enumerating “*Grand Challenges of Informatics*”

Manifest Domains: Formal Models of Processes and Prompts

Summary

Chapter 1, Manifest Domains: Analysis & Description, [49] introduced a method for analysing and describing manifest domains. In this chapter¹ we shall formalise the calculus of this method. The formalisation has two aspects: the formalisation of the process of sequencing the prompts of the calculus, and the formalisation of the individual prompts.

3.1 Introduction

The presentation of a calculus for analysing and describing manifest domains, introduced in **Chapter 1, Manifest Domains: Analysis & Description, [49]** and summarised in Sect. 3.2, was and is necessarily informal. The human process of “extracting” a description of a domain, based on analysis, “wavers” between the domain, as it is revealed to our senses, and therefore necessarily informal, and its recorded description, which we present in two forms, an informal narrative and a formalisation. In the present paper we shall provide a formal, operational semantics formalisation of the analysis and description calculus. There are two aspects to the semantics of the analysis and description calculus. There is the formal explanation of the process of applying the analysis and description prompts, in particular the practical meaning² of the results of applying the analysis prompts, and there is the formal explanation of the meaning of the results of applying the description prompts. The former (i.e., the practical meaning of the results of applying the analysis prompts) amounts to a model of the process whereby the domain analyser cum describer navigates “across” the domain, alternating between applying sequences of one or more analysis prompts and applying description prompts. The latter (formal explanation of the meaning of the results of applying the description prompts) amounts to a model of the domain (as it evolves in the mind of the analyser cum describer³), the meaning of the evolving description, and thereby the relation between the two.

3.1.1 The Triptych Approach to Software Development

Before software can be designed and coded one must have firm understanding of its requirements. Before requirements can be prescribed one must have a clear grasp of the application domain.

¹ This chapter is based on [52].

² in contrast to a formal mathematical meaning

³ By ‘domain analyser cum describer’ we mean a group of one or more professionals, well-educated and trained in the domain analysis & description techniques outlined in, for example, [49], and where these professionals work closely together. By ‘working closely together’ we mean that they, together, day-by-day work on each their sections of a common domain description document which they “buddy check”, say every morning, then discuss, as a group, also every day, and then revise and further extend, likewise every day. By “buddy checking” we mean that group member \mathcal{A} reviews group member \mathcal{B} ’s most recent sections – and where this reviewing alternates regularly: \mathcal{A} may first review \mathcal{B} ’s work, then \mathcal{C} ’s, etcetera.

We shall, occasionally refer to the ‘domain analyser cum describer’ as the ‘domain engineer’.

Definition 18 The Triptych Approach to Software Development: By a **trptych software development** we shall understand a development which, in principle, starts with either studying an existing or developing a new domain description, then proceeds to systematically deriving a requirements prescription from the domain description, and finally designs and codes the software from the requirements prescription

⊙

3.1.2 Method and Methodology

Definition 19 Method: By a **method** we shall understand a set of **principles** for selecting and applying a number of **techniques** and **tools** for **analysing** and **synthesizing** an artifact ⊙

Definition 20 Methodology: By **methodology** we shall understand the study and knowledge of one or more methods ⊙

Definition 21 Formal Method: By **formal method** we shall understand a method some or most of whose techniques and tools can be understood mathematically ⊙

Definition 22 Formal Software Development: By a **formal software development method** we shall understand a formal method where domain descriptions, requirements prescriptions and software designs are expressed in mathematically founded specification languages with the possibility of proving properties of these specifications, of steps and stages of development (refinements within domain descriptions, requirements prescriptions, software designs and between these) — properties such as correctness of software designs with respect to requirements, and satisfaction of user expectations (from software) with respect to domains ⊙

This paper deals with some of the triptych method principles and techniques for developments of domain descriptions. The paper puts forward a formal explanation of some of that method.

3.1.3 Related Work

To this author's knowledge there are not many papers, other than the author's own, [49, 53, 54, 48] and the present paper, which proposes a calculus of analysis and description prompts for capturing a domain, let alone, as this paper tries, to formalise aspects of this calculus.

There is, however a “school of software engineering”, “anchored” in the 1987 publication: [145, Leon Osterweil]. As the title of that paper reveals: “*Software Processes Are Software Too*” the emphasis is on considering the software development process as prescribable by a software program. That is not what we are aiming at. We are aiming at an abstract and formal description of a large class of domain analysis & description processes *in terms of possible development calculi*. And in such a way that one can reason about such processes. The Osterweil paper suggests that any particular software development can be described by a program, and, if we wish to reason about the software development process we must reason over that program, but there is no requirement that the “software process programs” be expressed in a language with a proof system.⁴ In contrast we can reason over the properties of the development calculi as well as over the resulting description.

There is another “school of programming”, one that more closely adheres to the use of a calculus [11, 139]. The calculus here is a set of refinement rules, a *Refinement Calculus*⁵, that “drives” the developer from a specification to an executable program. Again, that is not what we are doing here. The proposed calculi of analysis and of description prompts [49] “drives” the domain engineer in developing a domain description. That description may then be ‘refined’ using a refinement calculus.

⁴ The **RAISE** Specification Language [97] does have a proof system.

⁵ Ralph-Johan Back appears to be the first to have proposed the idea of refinement calculi, cf. his 1978 PhD thesis *On the Correctness of Refinement Steps in Program Development*, [http://users.abo.fi/backrj/index.php?page=Refinement calculus all.html&menu=3](http://users.abo.fi/backrj/index.php?page=Refinement%20calculus.all.html&menu=3).

3.1.4 Structure of Paper

Section 3.2 provides a terse summary of the analysis & description of endurants. It is without examples. For such we refer to [49, Sects. 2.–3., Pages 7–29.]. Section 3.3 is informal. It discusses issues of syntax and semantics. The reason we bring this short section is that the current paper turns “things upside/down”: from semantics we extract syntax ! From the real entities of actual domains we extract domain descriptions. Section 3.4 presents a pseudo-formal operational semantics explication of the process of proceeding through iterated sequences of analysis prompts to description prompts. The formal meaning of these prompts are given in Sect. 3.8. But first we must “prepare the ground”: The meaning of the analysis and description prompts is given in terms of some formal “context” in which the domain engineer works. Section 3.5 discusses this notion of “image” — an informal aspect of the ‘context’. It is a brief discussion. Section 3.6 presents the formal aspect of the ‘context’: perceived abstract syntaxes of the ontology of domain endurants and of endurant values. Section 3.7 Discusses, in a sense, the mental processes – *from syntax to semantics and back again* ! – that the domain engineer appears to undergo while analysing (the semantic) domain entities and synthesizing (the syntactic) domain descriptions. Section 3.8 presents the analysis and description prompts meanings. It represents a high point of this paper. It so-to-speak justifies the whole “exercise” ! Section 3.9 concludes the paper. We summarize what we have “achieved”. And we discuss whether this “achievement” is a valid one !

3.2 Domain Analysis and Description

In the rest of this paper we shall consider entities in the context of their being manifest (i.e., spatio-temporal). The restrictions of what we cover with respect to [49, *Manifest Domains: Analysis & Description*] are: we do not cover perdurants, only endurants, and within endurants we do not cover update mereology, update attributes and shared attributes. These omissions do not affect the main aim of this paper, namely that of presenting a plausible example of how one might wish to operationally formalise the notions of the analysis & description process and of the analysis & description prompts. The presentation is very terse. We refer to [49] for details. Appendices A–B gives “full” examples of “smallish” domain descriptions.

3.2.1 General

In [49] we developed an ontology for structuring and a prompt calculus analysing and describing domains. Figure 3.1 on the next page captures the ontology structure. It is thus a slight simplification of the ‘upper ontology’ figure given in [49] in that it omits the component ontology. The rest of this section will summarise the calculus. We refer to [49] for examples.

To the nodes of the upper ontology of Fig. 3.1 on the following page we have affixed some names. Names beginning with a capital stand for sub-ontologies. Names starting with a slanted *obs_* stand for description prompts. Other names (starting with an *is_* or a *has_* or other) stand for analysis prompts.⁶

3.2.2 Entities

Definition 23 Entity: By an **entity** we shall understand a **phenomenon**, i.e., something that can be observed, i.e., be seen or touched by humans, or that can be conceived as an abstraction of an entity. We further demand that an entity can be objectively described \odot ⁷

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

- *is_entity* — where *is_entity*(θ) holds if θ is an entity \diamond ⁸

⁶ In a coloured version of this document the description prompts are coloured red and the analysis prompts are coloured blue.

⁷ Definitions and examples are delimited by \odot respectively \square

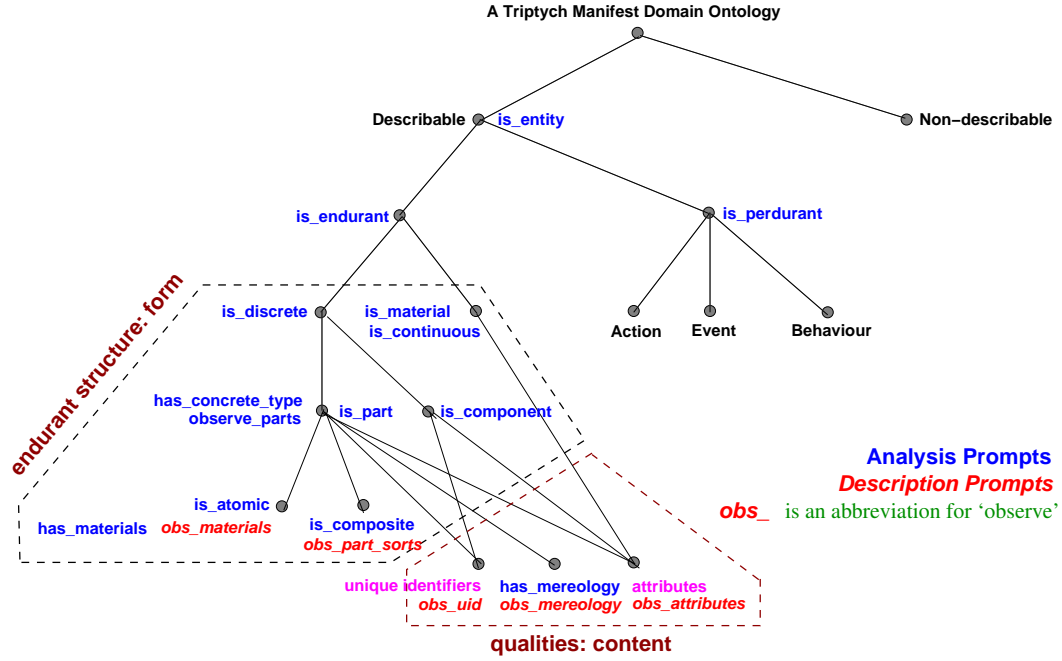


Fig. 3.1. An Upper Ontology for Domains

Although “reasonably” precise, the definition of the concept of **entity** is still not precise enough for us to formalise it. In Sect. 3.8.2 we attempt a series of formalisations of the analysis prompts. This is done on the background of some formalisation (Sect. 3.6) of the ontology being unfolded in this section (i.e., Sect. 3.2). A formalisation that covers the notion of phenomena and entities is not offered.

3.2.3 Endurants and Perdurants

Definition 24 Endurant: By an **endurant** we shall understand an entity that can be observed or conceived and described as a “complete thing” at no matter which given snapshot of time. Were we to “freeze” time we would still be able to observe the entire endurant ☉

Definition 25 Perdurant: By a **perdurant** we shall understand an entity for which only a fragment exists if we look at or touch them at any given snapshot in time, that is, where we to freeze time we would only see or touch a fragment of the perdurant ☉

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

- *is_endurant* — e is an endurant if *is_endurant*(e)⁹ holds.

is_entity is a **prerequisite prompt** for *is_endurant* ◇

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

- *is_perdurant* — e is a perdurant if *is_perdurant*(e)¹⁰ holds.

is_entity is a **prerequisite prompt** for *is_perdurant* ◇

⁸ **Analysis** prompt definitions and **description** prompt definitions and schemes are delimited by ◇ respectively ☉.

⁹ We formalise *is_endurant* in Sect. 3.8.2 on Page 107.

¹⁰ Since we do not cover perdurants in this paper we shall also refrain from trying to formalise this prompt.

3.2.4 Discrete and Continuous Endurants

Definition 26 Discrete Endurant: By a **discrete endurant** we shall understand an endurant which is separate, individual or distinct in form or concept ☺

Definition 27 Continuous Endurant: By a **continuous endurant** we shall understand an endurant which is prolonged, without interruption, in an unbroken series or pattern ☺

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

- $is_discrete — e$ is discrete if $is_discrete(e)^{11}$ holds ◇

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

- $is_continuous — e$ is continuous if $is_continuous(e)^{12}$ holds ◇

3.2.5 Parts, Components and Materials

General

Definition 28 Part: By a **part** we shall understand a discrete endurant which the domain engineer chooses to endow with **internal qualities** such as unique identification, mereology, and one or more attributes ☺

Definition 29 Component: By a **component** we shall understand a discrete endurant which the domain engineer chooses to not endow with **internal qualities** such as unique identification, mereology, and, even perhaps no attributes ☺

Definition 30 Material: By a **material** we shall understand a continuous endurant ☺

Part, Component and Material Prompts

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

- $is_part — e$ is a part if $is_part(e)^{13}$ holds ◇

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

- $is_component — e$ is a component if $is_component(e)^{14}$ holds ◇

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

- $is_material — e$ is a material if $is_material(e)^{15}$ holds ◇

There is no difference between $is_continuous$ and $is_material$, that is $is_continuous \equiv is_material$. We shall henceforth use $is_material$.

¹¹ We formalise $is_discrete$ in Sect. 3.8.2 on Page 107.

¹² We formalise $is_continuous$ in Sect. 3.8.2 on Page 107.

¹³ We formalise is_part in Sect. 3.8.2 on Page 107.

¹⁴ We formalise $is_component$ in Sect. 3.8.2 on Page 107.

¹⁵ We formalise $is_material$ in Sect. 3.8.2 on Page 107.

3.2.6 Atomic and Composite Parts

Definition 31 Atomic Part: Atomic parts are those which, in a given context, are deemed to not consist of meaningful, separately observable proper sub-parts ☹

A sub-part is a part ☹

Definition 32 Composite Part: Composite parts are those which, in a given context, are deemed to indeed consist of meaningful, separately observable proper sub-parts ☹

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

- $is_atomic(p)$: p is an atomic endurant if $is_atomic(p)^{16}$ holds ◇

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

- $is_composite(p)$: p is a composite endurant if $is_composite(p)^{17}$ holds ◇

3.2.7 On Observing Part Sorts

Part Sort Observer Functions

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

- $observe_part_sorts(p)^{18}$

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

8. $observe_part_sorts(p:P)$ schema

Narration:

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

Formalisation:

type

- [s] P_1, P_2, \dots, P_n

value

- [o] $obs_part_P_i: P \rightarrow P_i [1 \leq i \leq m]$

proof obligation [Disjointness of part sorts]

- [p] \mathcal{D}

\mathcal{D} is some predicate over P_1, P_2, \dots, P_n . It expresses their disjointedness. $is_composite$ is a prerequisite prompt of $observe_part_sorts$ △

¹⁶ We formalise is_atomic in Sect. 3.8.2 on Page 107.

¹⁷ We formalise $is_composite$ in Sect. 3.8.2 on Page 108.

¹⁸ We formalise $observe_part_sorts$ in Sect. 3.8.3 on Page 109.

On Discovering Concrete Part Types

Analysis Prompt 11 . *has_concrete_type*: *The domain analyser may decide that it is expedient, i.e., pragmatically sound, to render a part sort, P , whether atomic or composite, as a concrete type, T . That decision is prompted by the holding of the domain analysis prompt:*

- *has_concrete_type(p).*¹⁹

is_discrete is a **prerequisite prompt** of *has_concrete_type* ◇

Many possibilities offer themselves to model a concrete type as: either a set of abstract sorts, or a list of abstract sorts, or any compound of such sorts. Without loss of generality we suggest, as concrete type, as set of sorts. We have modeled many domains. So far, only the set concrete type has been needed.

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

- *observe_concrete_type(p)*²⁰

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

9. *observe_concrete_type($p:P$)* schema

Narration:

[t_1] ... narrative text on types ...
 [t_2] ... narrative text on types ...
 [o] ... narrative text on type observers ...

Formalisation:

type
 [t_1] Q
 [t_2] $T = Q\text{-set}$
value
 [o] $\text{obs_part_}T: P \rightarrow T$

Q may be any part sort; *has_concrete_type* is a **prerequisite prompt** of *observe_part_type* △

External and Internal Qualities of Parts

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

3.2.8 Unique Part Identifiers

We assume that all parts and components have unique identifiers. It may be, however, that we do not always need to define such a part or component identifier.

Domain Description Prompt 3 . *observe_unique_identifier*: *We can, however, always apply the domain description prompt:*

- *observe_unique_identifier(pk)*²¹

¹⁹ We formalise *has_concrete_type* in Sect. 3.8.2 on Page 108.

²⁰ We formalise *observe_concrete_type* in Sect. 3.8.3 on Page 109.

²¹ We formalise *observe_unique_identifier* in Sect. 3.8.3 on Page 110.

to parts, $p:P$, or components, k , resulting in the analyser writing down the unique identifier type and observer domain description text according to the following schema:

10. observe_unique_identifier(pk: (P|K)) schema

Narration:

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

Formalisation:

type

- [s] PI, KI

value

- [u] **uid_P**: $P \rightarrow PI$
- [u] **uid_K**: $K \rightarrow KI$

axiom

- [a] \mathcal{U}

\mathcal{U} is a predicate over part sorts and unique part identifier sorts, respectively component sorts and unique component identifiers. The unique part (component) identifier sort, PI (KI), is unique \triangle

3.2.9 Mereology

Part Mereology: Types and Functions

Analysis Prompt 12 . has_mereology: To discover necessary, sufficient and pleasing “mereology-hoods” the analyser can be said to endow a truth value **true** to the **domain analysis prompt**:

- *has_mereology*.²²

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

- *observe_mereology*(p)²³

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

11. observe_mereology(p:P) schema

Narration:

- [t] ... narrative text on mereology type ...
- [m] ... narrative text on mereology observer ...
- [a] ... narrative text on mereology type constraints ...

Formalisation:

type

- [t] $MT = \mathcal{E}(PI1, PI2, \dots, PIm)$

value

- [m] **obs_mereo_P**: $P \rightarrow MT$

axiom [Well-formedness of Domain Mereologies]

- [a] \mathcal{A}

²² We formalise *has_mereology* in Sect. 3.8.2 on Page 108.

²³ We formalise *observe_mereology* in Sect. 3.8.3 on Page 110.

MT is a type expression over unique part identifiers. \mathcal{A} is some predicate over unique part identifiers. The Pl_i are unique part identifier types \triangle

3.2.10 Part, Material and Component Attributes

Domain Description Prompt 5 . *observe_attributes*: The domain analyser experiments, thinks and reflects about attributes of endurants (parts $p:P$, components, $k:K$, or materials, $m:M$). That process is initiated by the **domain description prompt**:

- *observe_part_attributes*(e).²⁴

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

12. *observe_part_attributes*($e:(P|K|M)$) schema

Narration:

- [t] ... narrative text on attribute sorts ...
- [o] ... narrative text on attribute sort observers ...
- [p] ... narrative text on attribute sort proof obligations ...

Formalisation:

- type**
- [t] A_1, A_2, \dots, A_n
- value**
- [o] $\text{attr_}A_i:(P|K|M) \rightarrow A_i \ [1 \leq i \leq n]$
- proof obligation** [Disjointness of Attribute Types]
- [p] \mathcal{A}

The **type** (or rather sort) definitions: A_1, A_2, \dots, A_n inform us that the domain analyser has decided to focus on the distinctly named A_1, A_2, \dots, A_n attributes.²⁵ \mathcal{A} is a predicate over attribute types A_1, A_2, \dots, A_n . It expresses their Disjointness \triangle

3.2.11 Components

We now complement the *observe_part_sorts* (of Sect. 3.2.7). We assume, without loss of generality, that only atomic parts may contain components. Let $p:P$ be some atomic part.

Analysis Prompt 13 . *has_components*: The **domain analysis prompt**:

- *has_components*(p)²⁶

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

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

- *observe_component_sort*(p)²⁷

²⁴ We formalise *observe_attributes* in Sect. 3.8.3 on Page 110.

²⁵ The attribute type names are not like type names of, for example, a programming language. Instead they are chosen by the domain analyser to reflect on domain phenomena.

²⁶ We formalise *has_components* in Sect. 3.8.2 on Page 108.

²⁷ We formalise *observe_component_sort* in Sect. 3.8.3 on Page 111.

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

13. observe_component_sort(p:P) schema

Narration:

- [s] ... narrative text on component sort ...
- [o] ... narrative text on component sort observer ...

Formalisation:

type

- [s] K

value

- [o] **obs_comps**: $P \rightarrow K\text{-set}$

Components have unique identifiers and attributes, but no mereology \triangle

3.2.12 Materials

Only atomic parts may contain materials and materials may contain [atomic] parts.

Part Materials

Let $p:P$ be some atomic part.

Analysis Prompt 14 . has_material: The domain analysis prompt:

- $has_material(p)^{28}$

yields **true** if the atomic part $p:P$ potentially contains a material otherwise false \diamond

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

- $observe_material_sorts(p)^{29}$

yields the part material sort and material observer domain description text according to the following schema:

14. observe_material_sorts(p:P) schema

Narration:

- [s] ... narrative text on material sort ...
- [o] ... narrative text on material sort observer ...

Formalisation:

type

- [s] M

value

- [o] **obs_mat_M**: $P \rightarrow M$

\triangle

²⁸ We formalise `has_materials` in Sect. 3.8.2 on Page 108.

²⁹ We formalise `observe_material_sorts` in Sect. 3.8.3 on Page 110.

Material Parts

Materials may contain parts. We assume that such parts are always atomic and always of the same sort.

Example: Pipe parts usually contain oil material. And that oil material may contain pigs which are parts whose purpose it is to clean and inspect (i.e., maintain) pipes \square

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

- *has_parts(m)*³⁰

yields **true** if material $m:M$ potentially contains parts otherwise false \diamond

Domain Description Prompt 8 . *observe_material_sorts*: The domain description prompt:

- *observe_material_sort(e)*³¹

yields the material part sorts and material part observers domain description text according to the following schema:

15. *observe_material_sorts(m:M)* schema

Narration:

- [s] ... narrative text on material part sort ...
- [o] ... narrative text on material part sort observer ...

Formalisation:

- type**
- [s] mP
- value**
- [o] **obs_mat_mP**: $M \rightarrow mP$



3.2.13 Components and Materials

Experimental evidence³² appears to justify the following “limitations”: only atomic parts may contain either at most one material, and always of the same sort, or a set of zero, one or more components, all of the same sort; but not both; materials need not be characterised by unique identifiers; and components and materials need not be endowed with mereologies.

3.2.14 Discussion

We have covered the analysis and description calculi for endurants. We omit covering analysis and description techniques and tools for perdurants.

3.3 Syntax and Semantics

3.3.1 Form and Content

Section 3.2 appears to be expressed in the syntax of the **Raise** [97] **Specification Language**, **RSL** [96]. But it only “appears” so. When, in the “conventional” use of **RSL**, we apply meaning functions, we apply them to syntactic quantities. In Sect. 3.2 the “meaning” functions are the analysis, a.–o., and description, [1]–[8], prompts:

³⁰ We formalise *has_parts* in Sect. 3.8.2 on Page 108.

³¹ We formalise *observe_material_part_sort* in Sect. 3.8.3 on Page 111.

³² — in the form of more than 20 medium-to-large scale domain models

is_ atomic, 14, 86	is_ enduring, 10, 84
is_ component, 13, 85	is_ entity, 10, 83
is_ composite, 14, 86	is_ material, 13, 85
is_ continuous, 11, 85	is_ part, 13, 85
is_ discrete, 11, 85	is_ perdurant, 11, 84

and

observe_ attributes, 24, 89	observe_ mereology, 21, 88
observe_ component_ sorts, 27, 89	observe_ part_ sorts, 15, 86
observe_ concrete_ type, 16, 87	observe_ unique_ identifier, 19, 87
observe_ material_ sorts, 29, 90, 91	

The quantities that these prompts are “applied to” are semantic ones, in effect, they are the “ultimate” semantic quantities that we deal with: *the real*, i.e., *actual domain* entities ! The quantities that these prompts “yield” are syntactic ones ! That is, we have “turned matters inside/out”. From semantics we “extract” syntax. The arguments of the above-listed 23 prompts are domain entities, i.e., in principle, in-formalisable things. Their types, typically listed as P , denote possibly infinite classes, \mathcal{P} , of domain entities. When we write P we thus mean \mathcal{P} .

3.3.2 Syntactic and Semantic Types

When we, classically, define a programming language, we first present its syntax, then its semantics. The latter is presented as two – or three – possibly interwoven texts: the static semantics, i.e., the well-formedness of programs, the dynamic semantics, i.e., the mathematical meaning of programs — with a corresponding proof system being the “third text”. We shall briefly comment on the ideas of static and dynamic semantics. In designing a programming language, and therefore also in narrating and formalising it, one is well advised in deciding first on the semantic types, then on the syntactic ones. With describing [f.ex., manifest] domains, matters are the other way around: The semantic domains are given in the form of the endurants and perdurants; and the syntactic domains are given in the form that we, the humans of the domain, mention in our speech acts [166, 7]. That is, from a study of actual life domains, we extract the essentials that speech acts deal with when these speech acts are concerned with performing or talking about entities in some actual world.

3.3.3 Names and Denotations

Above, we may have been somewhat cavalier with the use of names for sorts and names for their meaning. Being so, i.e., “cavalier”, is, unfortunately a “standard” practice. And we shall, regrettably, continue to be cavalier, i.e., “loose” in our use of names of syntactic “things” and names for the denotation of these syntactic “things”. The context of these uses usually makes it clear which use we refer to: a syntactic use or a semantic one. As from Sect. 3.6 we shall be more careful distinguishing clearly between the names of sorts and the values of sorts, i.e., between syntax and semantics.

3.4 A Model of the Domain Analysis & Description Process

3.4.1 Introduction

A Summary of Prompts

In Sect. 3.3.1 we listed the two classes of prompts: the domain [endurant] analysis prompts: and the domain [endurant] description prompts: These prompts are “imposed” upon the domain by the domain analyser cum describer. They are “figuratively” applied to the domain. Their orderly, sequenced application follows the method hinted at in the previous section, detailed in [49, *Manifest Domains: Analysis & Description*]. This process of application of prompts will be expressed in a pseudo-formal notation in this section. The notation looks formal but since we have not formalised these prompts it is only pseudo-formal. We formalise these prompts in Sect. 3.8.

Preliminaries

Let P be a sort, that is, a collection of endurants. By P we shall understand both a syntactic quantity: the name of P , and a semantic quantity, the type (of all endurant values of type) P . By $\iota p:P$ we shall understand a semantic quantity: an (arbitrarily selected) endurant in P . To guide our analysis & description process we decompose it into steps. Each step “handles” a part sort $p:P$ or a material sort $m:M$ or a component sort $k:K$. Steps handling discovery of composite part sorts generates a set of part sort names $P_1, P_2, \dots, P_n:PNm$. Steps handling discovery of atomic part sorts may generate a material sort name, $m:MNm$, or component sort name, $k:KNm$. The part, material and component sort names are put in a reservoir for *sorts to be inspected*. Once handled, the sort name is removed from that reservoir. Handling of material sorts besides discovering their attributes may involve the discovery of further part sorts — which we assume to be atomic. Each domain description prompt results in domain specification text (here we show only the formal texts, not the narrative texts) being deposited in the domain description reservoir, a global variable τ . We do not formalise this text. Clauses of the form `observe_XXX(p)`, where `XXX` ranges over `part_sorts`, `concrete_type`, `unique_identifier`, `mereology`, `part_attributes`, `part_component_sorts`, `part_material_sorts`, and `material_part_sorts`, stand for “text” generating functions. They are defined in Sect. 3.8.3.

Initialising the Domain Analysis & Description Process

We remind the reader that we are dealing only with endurant domain entities. The domain analysis approach covered in Sect. 3.2 was based on decomposing an understanding of a domain from the “overall domain” into its components, and these, if not atomic, into their sub-domains. So we need to initialise the domain analysis & description process by selecting (or choosing) the domain Δ . Here is how we think of that “initialisation” process. The domain analyser & describer spends some time focusing on the domain, maybe at the “white board”³³, rambling, perhaps in an un-structured manner, across its domain, Δ , and its sub-domains. Informally jotting down more-or-less final sort names, building, in the domain analyser & describer’s mind an image of that domain. After some time doing this the domain analyser & describer is ready. An image of the domain includes the or a domain endurant, $\delta:\Delta$. Let Δnm be the name of the sort Δ . That name may be either a part sort name, or a material sort name, or a component sort name.

3.4.2 A Model of the Analysis & Description Process

A Process State

91 Let Nm denote either a part or a material or a component sort name.

92 A global variable αps will accumulate all the sort names being discovered.

93 A global variable vps will hold names of sorts that have been “discovered”, but have yet to be analysed & described.

type

91. $Nm = PNm \mid MNm \mid KNm$

variable

92. $\alpha ps := [\Delta nm]$ **type** $Nm\text{-set}$

93. $vps := [\Delta nm]$ **type** $Nm\text{-set}$

We shall explain the use of [...]s and operations on the above variables in Sect. 3.4.3 on Page 96. Each iteration of the “root” function, `analyse_and_describe_endurant_sort(Nm, nt:nm)`, as we shall call it, involves the selection of a sort (value) (which is that of either a part sort or a material sort) with this sort (value) then being removed.

94 The selection occurs from the global state component vps (hence: ()) and changes that state (hence **Unit**).

³³ Here ‘white board’ is a conceptual notion. It could be physical, it could be yellow “post-it” stickers, or it could be an electronic conference “gadget”.

value

94. `sel_and_rem_Nm: Unit → Nm`

94. `sel_and_rem_Nm() ≡ let nm:Nm • nm ∈ vps in vps := vps \ {nm} ; nm end; pre: vps ≠ {}`

A Technicality

95 The main analysis & description functions of the next sections, except the “root” function, are all expressed in terms of a pair, (nm,val):NmVAL, of a sort name and an endurant value of that sort.

type

95. `NmVAL = (PNm×PVAL) | (MNm×MVAL) | (KNm×KVAL)`

Analysis & Description of Endurants

96 To analyse and describe endurants means to first
 97 examine those endurants which have yet to be so analysed and described
 98 by selecting (and removing from vps) a yet un-examined sort nm;
 99 then analyse and describe an endurant entity (ι :nm) of that sort — this analysis, when applied to composite parts, leads to the insertion of zero³⁴ or more sort names³⁵.

As was indicated in Sect. 3.2, the mereology of a part, if it has one, may involve unique identifiers of any part sort, hence must be done after all such part sort unique identifiers have been identified. Similarly for attributes which also may involve unique identifiers,

100 then, if it has a mereology,
 101 to analyse and describe the mereology of each part sort,
 102 and finally to analyse and describe the attributes of each sort.

value

96. `analyse_and_describe_endurants: Unit → Unit`

96. `analyse_and_describe_endurants() ≡`

97. `while ~is_empty(vps) do`

98. `let nm = sel_and_rem_Nm() in`

99. `analyse_and_describe_endurant_sort(nm, ι :nm) end end ;`

100. `for all nm:PNm • nm ∈ α ps do if has_mereology(nm, ι :nm)36`

101. `then observe_mereology(nm, ι :nm)37 end end`

102. `for all nm:Nm • nm ∈ α ps do observe_attributes(nm, ι :nm)38 end`

The ι :nm of Items 99, 100, 101 and 102 are crucial. The domain analyser is focused on (part or material or component) sort nm and is “directed” (by those items) to choose (select) an endurant (a part or a material or component) ι :nm of that sort.

103 To analyse and describe an endurant

105 If it instead is a material, then to analyse and describe it as a material.

104 is to find out whether it is a part. If so then it is to analyse and describe it.

106 If it instead is a component, then to analyse and describe it as a component.

³⁴ If the sub-parts of ι :nm are all either atomic and have no materials or components or have already been analysed, then no new sort names are added to the repository vps).

³⁵ These new sort names are then “picked-up” for sort analysis &c. in a next iteration of the while loop.

³⁶ We formalise `has_mereology` in Sect. 3.8.2 on Page 108.

³⁷ We formalise `observe_mereology` in Sect. 3.8.3 on Page 110.

³⁸ We formalise `observe_attributes` in Sect. 3.8.3 on Page 110.

value

103. analyse_and_describe_endurant_sort: NmVAL \rightarrow **Unit**
 103. analyse_and_describe_endurant_sort(nm,val) \equiv
 104. **is_part**(nm,val)³⁹ \rightarrow ⁴⁰ analyse_and_describe_part_sorts(nm,val),
 105. **is_material**(nm,val)⁴¹ \rightarrow **observe_material_part_sort**(nm,val)⁴²,
 106. **is_component**(nm,val)⁴³ \rightarrow **observe_component_sort**(nm,val)⁴⁴

107 The analysis and description of a part 110 If composite it is analysed and described as
 108 first describe its unique identifier. such.
 109 If the part is atomic it is analysed and described
 as such; 111 Part p must be discrete.

value

107. analyse_and_describe_part_sorts: NmVAL \rightarrow **Unit**
 107. analyse_and_describe_part_sorts(nm,val) \equiv
 108. **observe_unique_identifier**(nm,val)⁴⁵;
 109. **is_atomic**(nm,val)⁴⁶ \rightarrow analyse_and_describe_atomic_part(nm,val),
 110. **is_composite**(nm,val)⁴⁷ \rightarrow analyse_and_describe_composite_parts(nm,val)
 111. **pre: is_discrete**(nm,val)⁴⁸

112 To analyse and describe an atomic part is to inquire whether
 a it embodies materials, then we analyse and describe these;
 b and if it further has components, then we describe their sorts.

value

112. analyse_and_describe_atomic_part: NmVAL \rightarrow **Unit**
 112. analyse_and_describe_atomic_part(nm,val) \equiv
 112a. **if has_material**(nm,val)⁴⁹ **then observe_part_material_sort**(nm,val)⁵⁰ **end** ;
 112b. **if has_components**(nm,val)⁵¹ **then observe_part_component_sort**(nm,val)⁵² **end**

113 To analyse and describe a composite endurant of sort nm (and value val)
 114 is to analyse if the sort has a concrete type
 115 then we analyse and describe that concrete sort type
 116 else we analyse and describe the abstract sort.

⁴⁴ We formalise **is_part** in Sect. 3.8.2 on Page 107.

⁴⁴ The conditional clause: $\text{cond}_1 \rightarrow \text{clau}_1, \text{cond}_2 \rightarrow \text{clau}_2, \dots, \text{cond}_n \rightarrow \text{clau}_n$
 is same as **if** cond_1 **then** clau_1 **else if** cond_2 **then** clau_2 **else ... if** cond_n **then** clau_n **end end ... end** .

⁴⁴ We formalise **is_material** in Sect. 3.8.2 on Page 107.

⁴⁴ We formalise **observe_material_part_sort** in Sect. 3.8.3 on Page 111.

⁴⁴ We formalise **is_component** in Sect. 3.8.2 on Page 107.

⁴⁴ We formalise **observe_component_sort** in Sect. 3.8.3 on Page 111.

⁴⁸ We formalise **observe_unique_identifier** in Sect. 3.8.3 on Page 110.

⁴⁸ We formalise **is_atomic** in Sect. 3.8.2 on Page 107.

⁴⁸ We formalise **is_composite** in Sect. 3.8.2 on Page 108.

⁴⁸ We formalise **is_discrete** in Sect. 3.8.2 on Page 107.

⁵² We formalise **has_material** in Sect. 3.8.2 on Page 108.

⁵² We formalise **observe_part_material_sort** in Sect. 3.8.3 on Page 110.

⁵² We formalise **has_components** in Sect. 3.8.2 on Page 108.

⁵² We formalise **observe_part_component_sort** in Sect. 3.8.3 on Page 111.

```

value
113. analyse_and_describe_composite_endurant: NmVAL  $\rightarrow$  Unit
113. analyse_and_describe_composite_endurant(nm,val)  $\equiv$ 
114.   if has_concrete_type(nm,val)53
115.     then observe_concrete_type(nm,val)54
116.     else observe_abstract_sorts(nm,val)55
114.   end
113.   pre is_composite(nm,val)56

```

We do not associate materials or components with composite parts.

3.4.3 Discussion of The Process Model

The above model lacks a formal understanding of the individual prompts as listed in Sect. 3.4.1; such an understanding is attempted in Sect. 3.8.

Termination

The sort name reservoir **vps** is “reduced” by one name in each iteration of the **while** loop of the analyse_and_describe_endurants, cf. Item 98 on Page 94, and is augmented by new part, material and component sort names in some iterations of that loop. We assume that (manifest) domains are finite, hence there are only a finite number of domain sorts. It remains to (formally) prove that the analysis & description process terminates.

Axioms and Proof Obligations

We have omitted, from Sect. 3.2, treatment of axioms concerning well-formedness of parts, materials and attributes and proof obligations concerning disjointedness of observed part and material sorts and attribute types. [49] exemplifies axioms and sketches some proof obligations.

Order of Analysis & Description: A Meaning of ‘ \oplus ’

The variables α ps, vps and τ can be defined to hold either sets or lists. The operator \oplus can be thought of as either set union (\cup and $[...] \equiv \{...\}$) — in which case the domain description text in τ is a set of domain description texts — or as list concatenation ($\hat{\ }^{\ } and $[...] \equiv \langle...\rangle$) of domain description texts. The list operator $\ell_1 \oplus \ell_2$ now has at least two interpretations: either $\ell_1 \hat{\ }^{\ } \ell_2$ or $\ell_2 \hat{\ }^{\ } \ell_1$. Thus, in the case of lists, the \oplus , i.e., $\hat{\ }^{\ }$, does not (suffix or prefix) append ℓ_2 elements already in ℓ_1 . The sel_and_rem_Nm function on Page 94 applies to the set interpretation. A list interpretation is:$

```

value
98. sel_and_rem_Nm: Unit  $\rightarrow$  Nm
98. sel_and_rem_Nm()  $\equiv$  let nm = hd v ps in v ps := tl v ps; nm end; pre: v ps  $\neq \langle \rangle$ 

```

In the first case ($\ell_1 \hat{\ }^{\ } \ell_2$) the analysis and description process proceeds from the root, breadth first, In the second case ($\ell_2 \hat{\ }^{\ } \ell_1$) the analysis and description process proceeds from the root, depth first. .

⁵³ We formalise has_concrete_type in Sect. 3.8.2 on Page 108.

⁵³ We formalise observe_concrete_type in Sect. 3.8.3 on Page 109.

⁵³ We formalise observe_part_sorts in Sect. 3.8.3 on Page 109.

⁵³ We formalise is_composite in Sect. 3.8.2 on Page 108.

Laws of Description Prompts

The domain ‘method’ outlined in the previous section suggests that many different orders of analysis & description may be possible. But are they? That is, will they all result in “similar” descriptions? If, for example, \mathcal{D}_a and \mathcal{D}_b are two domain description prompts where \mathcal{D}_a and \mathcal{D}_b can be pursued in any order will that yield the same description? And what do we mean by ‘can be pursued in any order’, and ‘same description’? Let us assume that sort P decomposes into sorts P_a and P_b (etcetera). Let us assume that the domain description prompt \mathcal{D}_a is related to the description of P_a and \mathcal{D}_b to P_b . Here we would expect \mathcal{D}_a and \mathcal{D}_b to commute, that is $\mathcal{D}_a; \mathcal{D}_b$ yields same result as does $\mathcal{D}_b; \mathcal{D}_a$. In [37] we made an early exploration of such laws of domain description prompts. To answer these questions we need a reasonably precise model of domain prompts. We attempt such a model in Sect. 3.8. But we do not prove theorems.

3.5 A Domain Analyser's & Descriptor's Domain Image

Assumptions: We assume that the domain analysers cum descriptors are well educated and well trained in the domain analysis & description techniques such as laid out in [49]. This assumption entails that the domain analysis & description development process is structured in sequences of alternating (one or more) analysis prompts and description prompts. We refer to Footnote 3 (Page 81) as well as to the discussion, “Towards a methodology of manifest domain analysis & description” of [49, Sect. 1.6]. We further assume that the domain analysers cum descriptors makes repeated attempts to analyse & describe a domain. We assume, further, that it is “the same domain” that is being analysed & described – two, three or more times, “all-over”, before commitment is made to attempt a – hopefully – final analysis & description⁵⁴, from “scratch”, that is, having “thrown away”, previous drafts⁵⁵. We then make the further assumption, as this iterative analysis & description process proceeds, from iteration i to $i + 1$, that each and all members of the analysis & description group are forming, in their minds (i.e., brains) an “image” of the domain being analysed. As iterations proceed one can then say that what is being analysed & described increasingly becomes this ‘image’ as much as it is being the domain — which we assume is not changing across iterations. The iterated descriptions are now postulated to converge: a “final” iteration “differs” only “immaterially.” from the description of the “previous” iteration.

• • •

The Domain Engineer's Image of Domains: In the opening (‘Assumptions’) of this section, i.e., above, we hinted at “an image”, in the minds of the domain analysers & descriptors, of the domain being researched and for which a description document is being engineered. In this paragraph we shall analyse what we mean by such a image. Since the analysis & description techniques are based on applying the analysis and description prompts (reviewed in Sect. 3.2) we can assume that the image somehow relates to the ‘ontology’ of the domain entities, whether endurants or perdurants, such as graphed in Fig. 3.1. Rather than further investigating (i.e., analysing / arguing) the form of this, until now, vague notion, we simply conjecture that the image is that of an ‘**abstract syntax of domain types**’.

• • •

The Iterative Nature of The Description Process: Assume that the domain engineers are analysing & describing a particular endurant; that is, as we shall understand it, are examining a given endurant node in the domain description tree! The **domain description tree** is defined by the facts that composite parts have sub-parts which may again be composite (tree branches), ending with atomic parts (the leaves of the tree) but not “circularly”, i.e. recursively ☹

To make this claim: *the domain analysers cum descriptors are examining a given endurant node in the domain description tree* amounts to saying that *the domain engineers have in their mind a reasonably “stable” “picture” of a domain in terms of a domain description tree.*

⁵⁴ – and if that otherwise planned, final analysis & description is not satisfactory, then yet one more iteration is taken.

⁵⁵ It may be useful, though, to keep a list of the names of all the endurant parts and their attribute names, should the group members accidentally forget such endurants and attributes: at least, if they do not appear in later document iterations, then it can be considered a deliberate omission.

We need explain this assumption. In this assumption there is “buried” an understanding that the domain analysers cum describers during the — what we can call “the final” — domain analysis & description process, that leads to a “deliverable” domain description, are not investigating the domain to be described for the first time. That is, we certainly assume that any “final” domain analysis & description process has been preceded by a number of iterations of “trial” domain analysis & description processes.

Hopefully this iteration of experimental domain analysis & description processes converges. Each iteration leads to some domain description, that is, some domain description tree. A first iteration is thus based on a rather incomplete domain description tree which, however, “quickly” emerges into a less incomplete one in that first iteration. When the domain engineers decide that a “final” iteration seems possible then a “final” description emerges. If acceptable, OK, otherwise yet an “final” iteration must be performed. Common to all iterations is that the domain analysers cum describers have in mind some more-or-less “complete” domain description tree and apply the prompts introduced in Sect. 3.4.

3.6 Domain Types

There are two kinds of types associated with domains: the syntactic types of endurant descriptions, and the semantic types of endurant values.

3.6.1 Syntactic Types: Parts, Materials and Components

In this section we outline an ‘**abstract syntax of domain types**’. In Sect. 3.6.1 we introduce the concept of sort names. Then, in Sects. 3.6.1–3.6.1, we describe the syntax of part, material and component types. Finally, in Sects. 3.6.1–3.6.1, we analyse this syntax with respect to a number of well-formedness criteria.

Syntax of Part, Material and Component Sort Names

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

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

- a part sort names,
- b material sort names and
- c component sort names,

117 N

118a $PN_m :: mkPN_m(N)$

118b $MN_m :: mkMN_m(N)$

118c $KN_m :: mkKN_m(N)$

An Abstract Syntax of Domain Endurants

119 We think of the types of parts, materials and components to be a map from their type names to respective type expressions.

120 Thus part types map part sort names into part types;

121 material types map material sort names into material types; and

122 component types map components sort names into component types.

123 Thus we can speak of endurant types to be either part types or material types or component types.

124 A part type expression is either an atomic part type expression or is a composite part type expression or is a concrete composite part type expression.

125 An atomic part type expression consists of a type expression for the qualities of the atomic

- part and, optionally, a material type name or a component type name (cf. Sect. 3.2.13).
- 126 An abstract composite part type expression consists of a type expression for the qualities of the composite part and a finite set of one or more part type names.
- 127 A concrete composite part type expression consists of a type expression for the qualities of the part and a part sort name standing for a set of parts of that sort.
- 128 A material part type expression consists of a type expression for the qualities of the material and an optional part type name.
- 129 We omit consideration of component types.

Endurants: Syntactic Types

- 119 $\text{TypDef} = \text{PTypes} \cup \text{MTypes} \cup \text{KTypes}$
- 120 $\text{PTypes} = \text{PNm} \rightarrow_m \text{PaTyp}$
- 121 $\text{MTypes} = \text{MNm} \rightarrow_m \text{MaTyp}$
- 122 $\text{KTypes} = \text{KNm} \rightarrow_m \text{KoTyp}$
- 123 $\text{ENDType} = \text{PaTyp} \mid \text{MaTyp} \mid \text{KoTyp}$
- 124 $\text{PaTyp} == \text{AtPaTyp} \mid \text{AbsCoPaTyp} \mid \text{ConCoPaTyp}$
- 125 $\text{AtPaTyp} :: \text{mkAtPaTyp}(s_{\text{qs}}:\text{PQ}, s_{\text{omkn}}:(\{ \mid \text{"nil"} \} \mid \text{MNn} \mid \text{KNm}))$
- 126 $\text{AbsCoPaTyp} :: \text{mkAbsCoPaTyp}(s_{\text{qs}}:\text{PQ}, s_{\text{pns}}:\text{PNm-set})$
- 126 **axiom** $\forall \text{mkAbsCoPaTyp}(pq, pns):\text{AbsCoPaTyp} \cdot pns \neq \{ \}$
- 127 $\text{ConCoPaTyp} :: \text{mkConCoPaTyp}(s_{\text{qs}}:\text{PQ}, s_{\text{p}}:\text{PNm})$
- 128 $\text{MaTyp} :: \text{mkMaTyp}(s_{\text{qs}}:\text{MQ}, s_{\text{opn}}:(\{ \mid \text{"nil"} \} \mid \text{PNm}))$
- 129 $\text{KoTyp} :: \text{mkKoTyp}(s_{\text{qs}}:\text{KQ})$

Quality Types

- 130 There are three aspects to part qualities: the type of the part unique identifiers, the type of the part mereology, and the name and type of attributes.
- 131 The type unique part identifiers is a not further defined atomic quantity.
- 132 A part mereology is either "nil" or it is an expression over part unique identifiers, where such expressions are those of either simple unique identifier tokens, or of set, or otherwise over simple unique identifier tokens, or ..., etc.
- 133 The type of attributes pairs distinct attribute names with attribute types —
- 134 both of which we presently leave further undefined.
- 135 Material attributes is the only aspect to material qualities.
- 136 Components have unique identifiers. Component attribute types are left undefined.

Qualities: Syntactic Types

- 130 $\text{PQ} = s_{\text{ui}}:\text{UI} \times s_{\text{me}}:\text{ME} \times s_{\text{attrs}}:\text{ATRS}$
- 131 UI
- 132 $\text{ME} == \text{"nil"} \mid \text{mkUI}(s_{\text{ui}}:\text{UI}) \mid \text{mkUIset}(s_{\text{uil}}:\text{UI}) \mid \dots$
- 133 $\text{ATRS} = \text{ANm} \rightarrow_m \text{ATyp}$
- 134 ANm, ATyp
- 135 $\text{MQ} = s_{\text{attrs}}:\text{ATRS}$
- 136 $\text{KQ} = s_{\text{uid}}:\text{UI} \times s_{\text{attrs}}:\text{ATRS}$

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

Well-formed Syntactic Types

Well-formed Definitions

137 We need define an auxiliary function, `names`, which, given an `endurant` type expression, yields the sort names that are referenced immediately by that type.

- a If the `endurant` type expression is that of an atomic part type then the sort name is that of its optional component sort.
- b If an abstract composite part type then the sort names of its parts.

c If a concrete composite part type then the sort name is that of the sort of its set of parts.

d If a material type then sort name is that of the sort of its optional parts.

e Component sorts have no references to other sorts.

value

137. `names: TypDef → (PNm|MNm|KNm) → (PNm|MNm|KNm)-set`

137. `names(td)(n) ≡`

137. `∪ { ns | ns:(PNm|MNm|KNm)-set •`

137. `case td(n) of`

137a. `mkAtPaTyp(⟦, n′) → ns={n′},`

137b. `mkAbsCoPaTyp(⟦, ns′) → ns=ns′,`

137c. `mkConCoPaTyp(⟦, pn) → ns={pn},`

137d. `mkMaTyp(⟦, n′) → ns={n′},`

137e. `mkKoTyp(⟦) → ns={}`

137. `end }`

138 Endurant sort names being referenced in part types, `PaTyp`, in material types, `MaTyp`, and in component types, `KoTyp`, of the `typedef:TypDef` definition, *must be defined in* the defining set, **dom** `typedef`, of the `typedef:TypDef` definition.

value

138. `wf_TypDef_1: TypDef → Bool`

138. `wf_TypDef_1(td) ≡ ∀ n:(PNm|MNm|CNm) • n ∈ dom td ⇒ names(td)(n) ⊆ dom td`

Perhaps Item 138. should be sharpened:

139 from “*must be defined in*” [138.] to “*must be equal to*”:

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

No Recursive Definitions

140 Type definitions must not define types recursively.

- a A type definition, `typedef:TypDef`, defines, typically composite part sorts, named, say, n , in terms of other part (material and component) types. This is captured in the

- `mncs` (Item 125),
- `pns` (Item 126),
- `p` (Item 127) and
- `pns` (Item 128),

selectable elements of respective type definitions. These elements identify type names of materials and components, parts, a part, and parts, respectively. None of these names may be n .

- b The identified type names may further identify type definitions none of whose selected type names may be n .
- c And so forth.

value

140. `wf_TypDef_2: TypDef → Bool`

140. `wf_TypDef_2(typdef) ≡ ∀ n:(PNm|MNm) • n ∈ dom typdef ⇒ n ∉ type_names(typdef)(n)`

140a. $\text{type_names} : \text{TypDef} \rightarrow (\text{PNm}|\text{MNm}) \rightarrow (\text{PNm}|\text{MNm})\text{-set}$
 140a. $\text{type_names}(\text{typdef})(\text{nm}) \equiv$
 140b. $\text{let } \text{ns} = \text{names}(\text{typdef})(\text{nm}) \cup \{ \text{names}(\text{typdef})(n) \mid n : (\text{PNm}|\text{MNm}) \cdot n \in \text{ns} \} \text{ in}$
 140c. $\text{nm} \notin \text{ns} \text{ end}$

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

3.6.2 Semantic Types: Parts, Materials and Components

Part, Material and Component Values

We define the values corresponding to the type definitions of Items 117.–136, structured as per type definition Item 123 on Page 98.

- | | |
|--|---|
| <p>141 An endurant value is either a part value, a material values or a component value.</p> <p>142 A part value is either the value of an atomic part, or of an abstract composite part, or of a concrete composite part.</p> <p>143 A atomic part value has a part quality value and, optionally, either a material or a possibly empty set of component values (cf. Sect. 3.2.13).</p> <p>144 An abstract composite part value has a part quality value and of at least (hence the axiom) of</p> | <p>145 one or more (distinct part type) part values.</p> <p>146 A concrete composite part value has a part quality value and a set of part values.</p> <p>147 A material value has a material quality value (of material attributes) and a (usually empty) finite set of part values.</p> <p>148 A component value has a component quality value (of a unique identifier and component attributes).</p> |
|--|---|

Endurant Values: Semantic Types

141 $\text{ENDVAL} = \text{PVAL} \mid \text{MVAL} \mid \text{KVAL}$
 142 $\text{PVAL} == \text{AtPaVAL} \mid \text{AbsCoPVAL} \mid \text{ConCoPVAL}$
 143 $\text{AtPaVAL} :: \text{mkAtPaVAL}(s_qval : \text{PQVAL}, s_omkvals : (\{ \mid \text{"nil"} \} \mid \text{MVAL} \mid \text{KVAL}\text{-set}))$
 144 $\text{AbsCoPVAL} :: \text{mkAbsCoPaVAL}(s_qval : \text{PQVAL}, s_pvals : (\text{PNm} \rightarrow_m \text{PVAL}))$
 145 **axiom** $\forall \text{mkAbsCoPaVAL}(pqs, ppm) : \text{AbsCoPVAL} \cdot ppm \neq []$
 146 $\text{ConCoPVAL} :: \text{mkConCoPaVAL}(s_qval : \text{PQVAL}, s_pvals : \text{PVAL}\text{-set})$
 147 $\text{MVAL} :: \text{mkMaVAL}(s_qval : \text{MQVAL}, s_pvals : \text{PVAL}\text{-set})$
 148 $\text{KVAL} :: \text{mkKoVAL}(s_qval : \text{KQVAL})$

Quality Values

- | | |
|---|--|
| <p>149 A part quality value consists of three qualities:</p> <p>150 a unique identifier type name, resp. value, which are both further undefined (atomic value) tokens;</p> <p>151 a mereology expression, resp. value, which is either a single unique identifier (type, resp.) value, or a set of such unique identifier (types, resp.) values, or ...; and</p> <p>152 an aggregate of attribute values, modeled here as a map from attribute type names to attribute values.</p> | <p>153 In this paper we leave attribute type names and attribute values further undefined.</p> <p>154 A material quality value consists just of an aggregate of attribute values, modeled here as a map from attribute type names to attribute values.</p> <p>155 A component quality value consists of a pair: a unique identifier value and an aggregate of attribute values, modeled here as a map from attribute type names to attribute values.</p> |
|---|--|

Qualities: Semantic Types

```

149   PQVAL = UIVAL × MEVAL × ATTRVALS
150   UIVAL
151   MEVAL == mkUIVAL(s_ui:UIVAL)|mkUIVALset(s_uis:UIVAL-set)|...
152   ATTRVALS = ANm  $\rightarrow_{\mathfrak{M}}$  AVAL
153   ANm, AVAL
154   MQVAL = ATTRVALS
155   KQVAL = UIVAL × ATTRVALS

```

We have left to define the values of attributes. For each part and material attribute value we assume a finite set of values. And for each unique identifier type (i.e., for each UI) we likewise assume a finite set of unique identifiers of that type. The value sets may be large. These assumptions help secure that the set of part, material and component values are also finite.

Type Checking

For part, material and component qualities we postulate an overloaded, simple type checking function, `type_of`, that applies to unique identifier values, `uiv:UIVAL`, and yield their unique identifier type name, `ui:UI`, to mereology values, `mev:MEVAL`, and yield their mereology expression, `me:ME`, and to attribute values, `AVAL` and `ATTRSVAL`, and yield their types: `ATyp`, respectively $(ANm \rightarrow_{\mathfrak{M}} AVAL) \rightarrow (ANm \rightarrow_{\mathfrak{M}} ATyp)$. Since we have let undefined both the syntactic type of attributes types, `ATyp`, and the semantic type of attribute values, `AVAL`, we shall leave `type_of` further unspecified.

value `type_of`: $(UIVAL \rightarrow UI) | (MEVAL \rightarrow ME) | (AVAL \rightarrow ATyp) | ((ANm \rightarrow_{\mathfrak{M}} AVAL) \rightarrow (ANm \rightarrow_{\mathfrak{M}} ATyp))$

The definition of the syntactic type of attributes types, `ATyp`, and the semantic type of attribute values, `AVAL`, is a simple exercise in a first-year programming language semantics course.

3.7 From Syntax to Semantics and Back Again !

The two syntaxes of the previous section: that of the syntactic domains, formula Items 117–136 (Pages 98–99), and that of the semantic domains, formula Items 141–155 (Pages 101–101), are not the syntaxes of domain descriptions, but of some aspects common to all domain descriptions developed according to the calculi of this paper. The syntactic domain formulas underlie (“are common to”, i.e., “abstracts”) aspects of all domain descriptions. The semantic domain formulas underlay (“are common to”, i.e., “abstracts”) aspects of the meaning of all domain descriptions. These two syntaxes, hence, are, so-to-speak, in the minds of the domain engineer (i.e., the analyser cum describer) while analysing the domain.

3.7.1 The Analysis & Description Prompt Arguments

The domain engineer analyse & describe endurants on the basis of a sort name i.e., a piece of syntax, `nm:Nm`, and an endurant value, i.e. a “piece” of semantics, `val:VAL`, that is, the arguments, $(nm, t:nm)$, of the analysis and description prompts of Sect. 3.4. Those two quantities are what the domain engineer are “operating” with, i.e., are handling: One is tangible, i.e. can be noted (i.e., “scribbled down”), the other is “in the mind” of the analysers cum describers. We can relate the two in terms of the two syntaxes, the syntactic types, and the meaning of the semantic types. But first some “preliminaries”.

3.7.2 Some Auxiliary Maps: Syntax to Semantics and Semantics to Syntax

We define two kinds of map types:

156 `Nm_to_ENDVALS` are maps from endurant sort names to respective sets of all corresponding endurant values of, and

157 `ENDVAL_to_Nm` are maps from endurant values to respective sort names.

type

156. $Nm_to_ENDVALS = (PNm \rightarrow PVAL\text{-}set) \cup (MNm \rightarrow MVAL\text{-}set) \cup (KNm \rightarrow KVAL\text{-}set)$
 157. $ENDVAL_to_Nm = (PVAL \rightarrow PNm) \cup (MVAL \rightarrow MNm) \cup (KVAL \rightarrow KNm)$

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

- 158 a function, $typval$, from type definitions, $typdef: TypDef$ to $Nm_to_ENDVALS$, and
 159 a function $valtyp$, from $Nm_to_ENDVALS$, to $ENDVAL_to_Nm$.

value

158. $typval: TypDef \rightarrow Nm_to_ENDVALS$
 159. $valtyp: Nm_to_ENDVALS \rightarrow ENDVAL_to_Nm$

160 The $typval$ function is defined in terms of a meaning function M (let $\rho: ENV$ abbreviate $Nm_to_ENDVALS$):

160. $M: (PaTyp \rightarrow ENV \rightarrow PVAL\text{-}set) \mid (MaTyp \rightarrow ENV \rightarrow MVAL\text{-}set) \mid (KoTyp \rightarrow ENV \rightarrow KVAL\text{-}set)$
 158. $typval(td) \equiv \text{let } \rho = [n \mapsto M(td(n))(\rho)]n: (PNm \mid MNm \mid KNm) \cdot n \in \text{dom } td \text{ in } \rho \text{ end}$
 159. $valtyp(\rho) \equiv [v \mapsto n: (PNm \mid MNm \mid CNm), v: (PVAL \mid MVAL \mid KVAL) \cdot n \in \text{dom } \rho \wedge v \in \rho(n)]$

The environment, ρ , of $typval$, Item 158, is the least fix point of the recursive equation

- 158. $\text{let } \rho = [n \mapsto M(td(n))(\rho)]n: (PNm \mid MNm \mid CNm) \cdot n \in \text{dom } td \text{ in } \dots$

The M function is defined next.

3.7.3 M: A Meaning of Type Names**Preliminaries**

The $typval$ function provides for a homomorphic image from $TypDef$ to $TypNm_to_VALS$. So, the narrative below, describes, item-by-item, this image. We refer to formula Items 158 and 160. The definition of M is decomposed into five sub-definitions, one for each kind of endurant type:

- Atomic parts: $mkAtPaTyp(s_qs: (UI \times ME \times ATRS), s_omkn: (\{ \text{"nil"} \} \mid MNm \mid KNm))$, Items 161;
- Abstract composite parts: $mkAbsCoPaTyp(s_qs: PQ, s_pns: PNm\text{-}set)$, 162 on the next page;
- Concrete composite parts: $mkConCoPaTyp(s_qs: PQ, s_p: PNm)$, Items 163 on the following page;
- Materials: $mkMaTyp(s_qs: MQ, s_opn: (\{ \text{"nil"} \} \mid PNm))$, Items 164 on Page 105; and
- Components: $mkKoTyp(s_qs: KQ)$, Items 165 on Page 105.

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

Atomic Parts

- 161 The meaning of an atomic part type expression,
 Item 125. $mkAtPaTyp((ui, me, attrs), omkn)$
 in $mkAtPaTyp(s_qs: PQ, s_omkn: (\{ \text{"nil"} \} \mid MNm \mid KNm))$,
 is the set of all atomic part values,
 Items 143., 149., 152. $mkAtPaVAL((uiv, mev, attrvals), omkval)$
 in $mkAtPaVAL(s_qval: (UIVAL \times MEVAL \times (ANm \rightarrow AVAL)),$
 $s_omkvals: (\{ \text{"nil"} \} \mid MVAL \mid KVAL\text{-}set))$.
 a uiv is a value in $UIVAL$ of type ui ,
 b mev is a value in $MEVAL$ of type me ,
 c $attrvals$ is a value in $(ANm \rightarrow AVAL)$ of type $(ANm \rightarrow ATyp)$, and
 d $omkvals$ is a value in $(\{ \text{"nil"} \} \mid MVAL \mid KVAL\text{-}set)$:
 i either 'nil',

- ii or one material value of type MNm,
- iii or a possibly empty set of component values, each of type KNm.

161. $M: \text{mkAtPaTyp}((\text{UI} \times \text{ME} \times (\text{ANm} \rightarrow_{\text{M}} \text{ATyp})) \times (\{ \text{"nil"} \} | \text{MVAL} | \text{KVAL-set})) \rightarrow \text{ENV} \xrightarrow{\sim} \text{PVAL-set}$
 161. $M(\text{mkAtPaTyp}((\text{ui}, \text{me}, \text{attrs}), \text{omkn}))(\rho) \equiv$
 161. $\{ \text{mkATPaVAL}((\text{uiv}, \text{mev}, \text{attrval}), \text{omkvals}) \mid$
 161a. $\text{uiv:UIVAL} \cdot \text{type_of}(\text{uiv}) = \text{ui},$
 161b. $\text{mev:MEVAL} \cdot \text{type_of}(\text{mev}) = \text{me},$
 161c. $\text{attrval:}(\text{ANm} \rightarrow_{\text{M}} \text{AVAL}) \cdot \text{type_of}(\text{attrval}) = \text{attrs},$
 161d. $\text{omkvals: case omkn of}$
 161(d)i. $\text{"nil"} \rightarrow \text{"nil"},$
 161(d)ii. $\text{mkMNn}(_) \rightarrow \text{mval:MVAL} \cdot \text{type_of}(\text{mval}) = \text{omkn},$
 161(d)iii. $\text{mkKNm}(_) \rightarrow \text{kvals:KVAL-set} \cdot \text{kvals} \subseteq \{ \text{kv} \mid \text{kv:KVAL} \cdot \text{type_of}(\text{kval}) = \text{omkn} \}$
 161d. $\text{end} \}$

Formula terms 161a–161(d)iii express that any applicable uiv is combined with any applicable mev is combined with any applicable attrval is combined with any applicable omkvals.

Abstract Composite Parts

162 The meaning of an abstract composite part type expression,
 Item 126. $\text{mkAbsCoPaTyp}((\text{ui}, \text{me}, \text{attrs}), \text{pns})$
 in $\text{mkAbsCoPaTyp}(\text{s_qs:PQ}, \text{s_pns:PNm-set}),$
 is the set of all abstract, composite part values,
 Items 144., 149., 152., $\text{mkAbsCoPaVAL}((\text{uiv}, \text{mev}, \text{attrvals}), \text{pvals})$
 in $\text{mkAbsCoPaVAL}(\text{s_qval:}(\text{UIVAL} \times \text{MEVAL} \times (\text{ANm} \rightarrow_{\text{M}} \text{AVAL})), \text{s_pvals:}(\text{PNm} \rightarrow_{\text{M}} \text{PVAL})).$
 a uiv is a value in UIVAL of type ui: UI,
 b mev is a value in MEVAL of type me: ME,
 c attrvals is a value in $(\text{ANm} \rightarrow_{\text{M}} \text{AVAL})$ of type $(\text{ANm} \rightarrow_{\text{M}} \text{ATyp})$, and
 d pvals is a map of part values in $(\text{PNm} \rightarrow_{\text{M}} \text{PVAL})$, one for each name, pn:PNm, in pns such that
 these part values are of the type defined for pn.

162. $M: \text{mkAbsCoPaTyp}((\text{UI} \times \text{ME} \times (\text{ANm} \rightarrow_{\text{M}} \text{ATyp})), \text{PNm-set}) \rightarrow \text{ENV} \xrightarrow{\sim} \text{PVAL-set}$
 162. $M(\text{mkAbsCoPaTyp}((\text{ui}, \text{me}, \text{attrs}), \text{pns}))(\rho) \equiv$
 162. $\{ \text{mkAbsCoPaVAL}((\text{uiv}, \text{mev}, \text{attrvals}), \text{pvals}) \mid$
 162a. $\text{uiv:UIVAL} \cdot \text{type_of}(\text{uiv}) = \text{ui}$
 162b. $\text{mev:MEVAL} \cdot \text{type_of}(\text{mev}) = \text{me},$
 162c. $\text{attrvals:}(\text{ANm} \rightarrow_{\text{M}} \text{ATyp}) \cdot \text{type_of}(\text{attrval}) = \text{attrs},$
 162d. $\text{pvals:}(\text{PNm} \rightarrow_{\text{M}} \text{PVAL}) \cdot \text{pvals} \in \{ [\text{pn} \mapsto \text{pval} \mid \text{pn:PNm}, \text{pval:PVAL} \cdot \text{pn} \in \text{pns} \wedge \text{pval} \in \rho(\text{pn})] \}$

Concrete Composite Parts

163 The meaning of a concrete composite part type expression, Item 127.
 $\text{mkConCoPaTyp}((\text{ui}, \text{me}, \text{attrs}), \text{pn})$
 in $\text{mkConCoPaTyp}(\text{s_qs:}(\text{UI} \times \text{ME} \times (\text{ANm} \rightarrow_{\text{M}} \text{ATyp})), \text{s_pn:PNm}),$
 is the set of all concrete, composite set part values,
 Item 146. $\text{mkConCoPaVAL}((\text{uiv}, \text{mev}, \text{attrvals}), \text{pvals})$
 in $\text{mkConCoPaVAL}(\text{s_qval:}(\text{UIVAL} \times \text{MEVAL} \times (\text{ANm} \rightarrow_{\text{M}} \text{AVAL})), \text{s_pvals:PVAL-set}).$
 a uiv is a value in UIVAL of type ui,
 b mev is a value in MEVAL of type me,
 c attrvals is a value in $(\text{ANm} \rightarrow_{\text{M}} \text{AVAL})$ of type attrs, and
 d pvals is a[ny] value in PVAL-set where each part value in pvals is of the type defined for pn.

163. $M: \text{mkConCoPaTyp}((\text{UI} \times \text{ME} \times (\text{ANm} \rightarrow_{\text{M}} \text{ATyp})) \times \text{PNm}) \rightarrow \text{ENV} \xrightarrow{\sim} \text{PVAL-set}$
 163. $M(\text{mkConCoPaTyp}((\text{ui}, \text{me}, \text{attrs}), \text{pn}))(\rho) \equiv$
 163. $\{ \text{mkConCoPaVAL}((\text{uiv}, \text{mev}, \text{attrvals}), \text{pvals}) \mid$
 163a. $\text{uiv:UIVAL} \cdot \text{type_of}(\text{uiv}) = \text{ui},$
 163b. $\text{mev:MEVAL} \cdot \text{type_of}(\text{mev}) = \text{me},$
 163c. $\text{attrvals:}(\text{ANm} \rightarrow_{\text{M}} \text{AVAL}) \cdot \text{type_of}(\text{attrvals}) = \text{attrs},$
 163d. $\text{pvals:PVAL-set} \cdot \text{pvals} \subseteq \rho(\text{pn}) \}$

Materials

164 The meaning of a material type, 128.,
 expression $\text{mkMaTyp}(\text{mq}, \text{pn})$ in $\text{mkMaTyp}(\text{s_qs:MQ}, \text{s_pn:PNm})$
 is the set of values $\text{mkMaVAL}(\text{mqval}, \text{ps})$
 in $\text{mkMaVAL}(\text{s_qval:MQVAL}, \text{s_pvals:PVAL-set})$ such that
 a mqval in MQVAL is of type mq , and
 b ps is a set of part values all of type pn .

164. $M: \text{mkMaTyp}(\text{s_mq:}(\text{ANm} \rightarrow_{\text{M}} \text{ATyp}), \text{s_pn:PNm}) \rightarrow \text{ENV} \xrightarrow{\sim} \text{MVAL-set}$
 164. $M(\text{mq}, \text{pn})(\rho) \equiv$
 164. $\{ \text{mkMVAL}(\text{mqval}, \text{ps}) \mid$
 164a. $\text{mqval:MVAL} \cdot \text{type_of}(\text{mqval}) = \text{mq},$
 164b. $\text{ps:PVAL-set} \cdot \text{ps} \subseteq \rho(\text{pn}) \}$

Components

165 The meaning of a component type, 129., expression $\text{mkKoType}(\text{ui}, \text{attrs})$
 in $\text{mkKoTyp}(\text{s_qs:}(\text{s_uid:UI} \times \text{s_attrs:ATRS}))$ is the set of values, 128., $\text{mkKQVAL}(\text{uiv}, \text{attrvals})$
 in, 148, $\text{mkKoVAL}(\text{s_qval:}(\text{uiv}, \text{attrvals}))$.
 a uiv is in UIVAL of type ui , and
 b attrvals is in ATTRSVAL of type attrs .

165. $M: \text{mkKoTyp}(\text{UI} \times \text{ATRS}) \rightarrow \text{ENV} \rightarrow \text{KVAL-set}$
 165. $M(\text{mkKoType}(\text{ui}, \text{attrs}))(\rho) \equiv$
 165. $\{ \text{mkKoVAL}(\text{uiv}, \text{attrvals}) \mid$
 165a. $\text{uiv:UIVAL} \cdot \text{type_of}(\text{uiv}) = \text{ui},$
 165b. $\text{attrvals:ATTRSVAL} \cdot \text{type_of}(\text{attrvals}) = \text{attrs} \}$

3.7.4 The ι Description Function

We can now define the meaning of the syntactic clause:

- $\iota \text{Nm:Nm}$

166 $\iota \text{Nm:Nm}$ “chooses” an arbitrary value from amongst the values of sort Nm :

value

166. $\iota \text{ nm:Nm} \equiv \text{iota}(\text{nm})$
 166. $\text{iota: Nm} \rightarrow \text{TypDef} \rightarrow \text{VAL}$
 166. $\text{iota}(\text{nm})(\text{td}) \equiv \text{let val:}(\text{PVAL}|\text{MVAL}|\text{KVAL}) \cdot \text{val} \in (\text{typval}(\text{td}))(\text{nm}) \text{ in val end}$

Discussion

From the above two functions, **typval** and **valtyp**, and the type definition “table” `td:TypDef` and “argument value” `val:PVAL|MVAL|KVAL`, we can form some expressions. One can understand these expressions as, for example reflecting the following analysis situations:

- **typval**(`td`): From the type definitions we form a map, by means of function **typval**, from sort names to the set of all values of respective sorts: `Nm_to_ENDVALS`.
That is, whenever we, in the following, as part of some formula, write **typval**(`td`), then we mean to express that the domain engineer forms those associations, in her mind, from sort names to usually very large, non-trivial sets of endurant values.
- **valtyp**(**typval**(`td`)): The domain analyser cum describer “inverts”, again in his mind, the **typval**(`td`) into a simple map, `ENDVAL_to_Nm`, from single endurant values to their sort names.
- (**valtyp**(**typval**(`td`)))(`val`): The domain engineer now “applies”, in her mind, the simple map (above) to an endurant value and obtains its sort name `nm:Nm`.
- `td`((**valtyp**(**typval**(`td`)))(`val`)): The domain analyser cum describer then applies the type definition “table” `td:TypDef` to the sort name `nm:Nm` and obtains, in his mind, the corresponding type definition, `PaTyp|MaTyp|KoTyp`.

We leave it to the reader to otherwise get familiarised with these expressions.

3.8 A Formal Description of a Meaning of Prompts

3.8.1 On Function Overloading

In Sect. 3.4 the analysis and description prompt invocations were expressed as

- `is_XXX(e)`, `has_YYY(e)` and `observe_ZZZ(e)`

where `XXX`, `YYY`, and `ZZZ` were appropriate entity sorts and `e` were appropriate endurants (parts, components and materials). The function invocations, `is_XXX(e)`, etcetera, takes place in the context of a type definition, `td:TypDef`, that is, instead of `is_XXX(e)`, etc. we get

- `is_XXX(e)(td)`, `has_YYY(e)(td)` and `observe_ZZZ(e)(td)`.

We say that the functions `is_XXX`, etc., are “lifted”.

3.8.2 The Analysis Prompts

The analysis is expressed in terms of the analysis prompts:

<code>is_ atomic, 14, 86</code>	<code>is_ endurant, 10, 84</code>
<code>is_ component, 13, 85</code>	<code>is_ entity, 10, 83</code>
<code>is_ composite, 14, 86</code>	<code>is_ material, 13, 85</code>
<code>is_ continuous, 11, 85</code>	<code>is_ part, 13, 85</code>
<code>is_ discrete, 11, 85</code>	<code>is_ perdurant, 11, 84</code>

The analysis takes place in the context of a type definition “image”, `td:TypDef`, in the minds of the domain engineers.

is_entity

The `is_entity` predicate is meta-linguistic, that is, we cannot model it on the basis of the type systems given in Sect. 3.6. So we shall just have to accept that.

is_endurant

See analysis prompt definition 2 on Page 84 and Formula Item 104 on Page 94.

value

$\text{is_endurant: Nm} \times \text{VAL} \rightarrow \text{TypDef} \xrightarrow{\sim} \mathbf{Bool}$
 $\text{is_endurant}(_, \text{val})(\text{td}) \equiv \text{val} \in \mathbf{dom} \text{ valtyp}(\text{typval}(\text{td})); \mathbf{pre: VAL} \text{ is any value type}$

is_discrete

See analysis prompt definition 4 on Page 85 and Formula Item 111 on Page 95.

value

$\text{is_discrete: NmVAL} \rightarrow \text{TypDef} \xrightarrow{\sim} \mathbf{Bool}$
 $\text{is_discrete}(_, \text{val})(\text{td}) \equiv (\text{is_PaTyp} | \text{is_CoTyp})(\text{td}((\text{valtyp}(\text{typval}(\text{td}))) (\text{val})))$

is_part

See analysis prompt definition 6 on Page 85 and Formula Item 104 on Page 94.

value

$\text{is_part: NmVAL} \rightarrow \text{TypDef} \xrightarrow{\sim} \mathbf{Bool}$
 $\text{is_part}(_, \text{val})(\text{td}) \equiv \text{is_PaTyp}(\text{td}((\text{valtyp}(\text{typval}(\text{td}))) (\text{val})))$

is_material [≡ is_continuous]

See analysis prompt definition 8 on Page 85 and Formula Item 105 on Page 94.

We remind the reader that $\text{is_continuous} \equiv \text{is_material}$.

value

$\text{is_material: NmVAL} \rightarrow \text{TypDef} \xrightarrow{\sim} \mathbf{Bool}$
 $\text{is_material}(_, \text{val})(\text{td}) \equiv \text{is_MaTyp}(\text{td}((\text{valtyp}(\text{typval}(\text{td}))) (\text{val})))$

is_component

See analysis prompt definition 7 on Page 85 and Formula Item 106 on Page 94.

value

$\text{is_component: NmVAL} \rightarrow \text{TypDef} \xrightarrow{\sim} \mathbf{Bool}$
 $\text{is_component}(_, \text{val})(\text{td}) \equiv \text{is_CoTyp}(\text{td}((\text{valtyp}(\text{typval}(\text{td}))) (\text{val})))$

is_atomic

See analysis prompt definition 9 on Page 86 and Formula Item 109 on Page 95.

value

$\text{is_atomic: NmVAL} \rightarrow \text{TypDef} \xrightarrow{\sim} \mathbf{Bool}$
 $\text{is_atomic}(_, \text{val})(\text{td}) \equiv \text{is_AtPaTyp}(\text{td}((\text{valtyp}(\text{typval}(\text{td}))) ()))$

is_composite

See analysis prompt definition 10 on Page 86 and Formula Item 110 on Page 95.

value

$\text{is_composite: NmVAL} \rightarrow \text{TypDef} \xrightarrow{\sim} \mathbf{Bool}$
 $\text{is_composite}(_, \text{val})(\text{td}) \equiv (\text{is_AbsCoPaTyp} | \text{is_ConCoPaTyp})(\text{td}((\text{valtyp}(\text{typval}(\text{td}))) (\text{val})))$

has_concrete_type

See analysis prompt definition 11 on Page 87 and Formula Item 114 on Page 95.

value

$\text{has_concrete_type: NmVAL} \rightarrow \text{TypDef} \xrightarrow{\sim} \mathbf{Bool}$
 $\text{has_concrete_type}(_, \text{val})(\text{td}) \equiv \text{is_ConCoPaTyp}(\text{td}((\text{valtyp}(\text{typval}(\text{td}))) (\text{val})))$

has_mereology

See analysis prompt definition 12 on Page 88 and Formula Item 100 on Page 94.

value

$\text{has_mereology: NmVAL} \rightarrow \text{TypDef} \xrightarrow{\sim} \mathbf{Bool}$
 $\text{has_mereology}(_, \text{val})(\text{td}) \equiv \text{s_me}(\text{td}((\text{valtyp}(\text{typval}(\text{td}))) (\text{val}))) \neq \text{"nil"}$

has_materials

See analysis prompt definition 14 on Page 90 and Formula Item 112a on Page 95.

value

$\text{has_material: NmVAL} \rightarrow \text{TypDef} \xrightarrow{\sim} \mathbf{Bool}$
 $\text{has_material}(_, \text{val})(\text{td}) \equiv \text{is_MNM}(\text{s_omkn}(\text{td}((\text{valtyp}(\text{typval}(\text{td}))) (\text{val}))))$
pre: $\text{is_AtPaTyp}(\text{td}((\text{valtyp}(\text{typval}(\text{td}))) (\text{val})))$

has_components

See analysis prompt definition 13 on Page 89 and Formula Item 112b on Page 95.

value

$\text{has_components: NmVAL} \rightarrow \text{TypDef} \xrightarrow{\sim} \mathbf{Bool}$
 $\text{has_components}(_, \text{val})(\text{td}) \equiv \text{is_KNM}(\text{s_omkn}(\text{td}((\text{valtyp}(\text{typval}(\text{td}))) (\text{val}))))$
pre: $\text{is_AtPaTyp}(\text{td}((\text{valtyp}(\text{typval}(\text{td}))) (\text{val})))$

has_parts

See description prompt definition 15 on Page 91.

value

$\text{has_parts: NmVAL} \rightarrow \text{TypDef} \xrightarrow{\sim} \mathbf{Bool}$
 $\text{has_parts}(_, \text{val})(\text{td}) \equiv \text{is_PNM}(\text{s_opn}(\text{td}((\text{valtyp}(\text{typval}(\text{td}))) (\text{val}))))$
pre: $\text{is_MaTyp}(\text{td}((\text{valtyp}(\text{typval}(\text{td}))) (\text{val})))$

3.8.3 The Description Prompts

These are the domain description prompts to be defined:

observe_ attributes, 24, 89	observe_ mereology, 21, 88
observe_ component_ sorts, 27, 89	observe_ part_ sorts, 15, 86
observe_ concrete_ type, 16, 87	observe_ unique_ identifier, 19, 87
observe_ material_ sorts, 29, 90, 91	

A Description State

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

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

variable

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

We shall explain the use of [...]s and the operations of \setminus and \oplus on the above variables in Sect. 3.4.3 on Page 96.

observe_part_sorts

See description prompt definition 1 on Page 86 and Formula Item 116 on Page 95.

value

```

observe_part_sorts: NmVAL  $\rightarrow$  TypDef  $\rightarrow$  Unit
observe_part_sorts(nm,val)(td)  $\equiv$ 
  let mkAbsCoPaTyp( $\_, \{P_1, P_2, \dots, P_n\}$ ) = td((valtyp(typval(td)))(val)) in
     $\tau := \tau \oplus [ \text{" type } P_1, P_2, \dots, P_n;$ 
      value
        obs_part_  $P_1$  nm  $\rightarrow$   $P_1$ 
        obs_part_  $P_2$  nm  $\rightarrow$   $P_2$ 
        ...,
        obs_part_  $P_n$  nm  $\rightarrow$   $P_n$ ;
      proof obligation
         $\mathcal{D}; "$  ]
    ||  $vps := vps \oplus ([P_1, P_2, \dots, P_n] \setminus \alpha ps)$ 
    ||  $\alpha ps := \alpha ps \oplus [P_1, P_2, \dots, P_n]$ 
  end
  pre: is_AbsCoPaTyp(td((valtyp(typval(td)))(val)))

```

\mathcal{D} is a predicate expressing the disjointness of part sorts P_1, P_2, \dots, P_n

observe_concrete_type

See description prompt definition 2 on Page 87 and Formula Item 115 on Page 95.

value

```

observe_concrete_type: NmVAL  $\rightarrow$  TypDef  $\rightarrow$  Unit
observe_concrete_type(nm,val)(td)  $\equiv$ 
  let mkConCoPaTyp( $\_, P$ ) = td((valtyp(typval(td)))(val)) in
     $\tau := \tau \oplus [ \text{" type } T = P\text{-set} ; \text{value obs\_part\_} T: nm \rightarrow T; "$  ]
    ||  $vps := vps \oplus ([P] \setminus \alpha ps)$ 
    ||  $\alpha ps := \alpha ps \oplus [P]$ 
  end
  pre: is_ConCoPaTyp(td((valtyp(typval(td)))(val)))

```

observe_unique_identifier

See description prompt definition 3 on Page 87 and Formula Item 108 on Page 95.

value

```
observe_unique_identifier: P → TypDef → Unit
observe_unique_identifier(nm,val)(td) ≡
  τ := τ ⊕ [ " type PI ; value uid_PI: nm → PI ; axiom ℳ ; " ]
```

ℳ is a predicate expression over unique identifiers.

observe_mereology

See description prompt definition 4 on Page 88 and Formula Item 101 on Page 94.

value

```
observe_mereology: NmVAL → TypDef → Unit
observe_mereology(nm,val)(td) ≡
  τ := τ ⊕ [ " type MT = ℳ (PI1,PI2,...,PIn) ;
    value obs_mereo_P: nm → MT ;
    axiom ℳℰ ; " ]
  pre: has_mereology(nm,val)(td) 56
```

ℳ(PI1,PI2,...,PI_n) is a type expression over unique part identifiers. ℳℰ is a predicate expression over unique part identifiers.

observe_part_attributes

See description prompt definition 5 on Page 89 and Formula Item 102 on Page 94.

value

```
observe_part_attributes: NmVAL → TypDef → Unit
observe_part_attributes(nm,val)(td) ≡
  let {A1,A2,...,Aa} = dom s_attrs(s_qs(val)) in
  τ := τ ⊕ [ " type A1, A2, ..., Aa
    value attr_A1: nm→Ai
    attr_A2: nm→A1
    ...
    attr_Aa: nm→Ai
    proof obligation [Disjointness of Attribute Types]
    ℳ ; " ]
  end
```

ℳ is a predicate over attribute types A₁, A₂, ..., A_a.

observe_part_material_sort

See description prompt definition 7 on Page 90 and Formula Item 112a on Page 95.

value

```
observe_part_material_sort: NmVAL → TypDef → Unit
observe_part_material_sort(nm,val)(td) ≡
  let M = s_pns(td((valtyp(typval(td)))(val))) in
```

⁵⁶ See analysis prompt definition 12 on Page 88

```

 $\tau := \tau \oplus [ \text{type } M ; \text{value obs\_mat\_M: nm} \rightarrow M ]$ 
 $\parallel vps := vps \oplus ([M] \setminus \alpha ps)$ 
 $\parallel \alpha ps := \alpha ps \oplus [M]$ 
end
pre: is_AtPaVAL(val)  $\wedge$  is_MNm(s_pns(td((valtyp(typval(td)))(val))))

```

observe_component_sort

See description prompt definition 6 on Page 89 and Formula Item 112b on Page 95.

value

```

observe_component_sort: NmVAL  $\rightarrow$  TypDef  $\rightarrow$  Unit
observe_component_sort(nm,val)(td)  $\equiv$ 
  let K = s_omkn(td((valtyp(typval(td)))(val))) in
   $\tau := \tau \oplus [ \text{type } K ; \text{value obs\_comps: nm} \rightarrow K\text{-set}; ]$ 
   $\parallel vps := vps \oplus ([K] \setminus \alpha ps)$ 
   $\parallel \alpha ps := \alpha ps \oplus [K]$ 
end
pre: is_AtPaTyp(td((valtyp(typval(td)))(val)))  $\wedge$  has_components(nm,val)

```

observe_material_part_sort

See description prompt definition 8 on Page 91 and Formula Item 106 on Page 94.

value

```

observe_material_part_sort: NmVAL  $\rightarrow$  TypDef  $\rightarrow$  Unit
observe_material_part_sort(nm,val)(td)  $\equiv$ 
  let P = s_pns(td((valtyp(typval(td)))(val))) in
   $\tau := \tau \oplus [ \text{type } P ; \text{value obs\_part\_P: nm} \rightarrow P ]$ 
   $\parallel vps := vps \oplus ([P] \setminus \alpha ps)$ 
   $\parallel \alpha ps := \alpha ps \oplus [P]$ 
end
pre is_MaTyp(td((valtyp(typval(td)))(val)))  $\wedge$  is_PNm(s_pns(td((valtyp(typval(td)))(val))))

```

3.8.4 Discussion of The Prompt Model

The prompt model of this section is formulated so as to reflect a “wavering”, of the domain engineer, between syntactic and semantic reflections. The syntactic reflections are represented by the syntactic arguments of the sort names, nm, and the type definitions, td. The semantic reflections are represented by the semantic argument of values, val. When we, in the various prompt definitions, use the expression $td((valtyp(typval(td)))(val))$ we mean to model that the domain analyser cum describer reflects semantically: “viewing”, as it were, the endurant. We could, as well, have written $td(nm)$ — reflecting a syntactic reference to the (emerging) type model in the mind of the domain engineer.

3.9 Conclusion

It is time to summarise, conclude and look forward.

3.9.1 What Has Been Achieved ?

[49] proposed a set of domain analysis & description prompts – and Sect. 3.2. summarised that language. Sections 3.4. and 3.8. proposed an operational semantics for the process of selecting and applying prompts, respectively a more abstract meaning of of these prompts, the latter based on some notions of an “image” of perceived abstract types of syntactic and of semantic structures of the perceived domain. These notions were discussed in Sects. 3.5. and 3.6. To the best of our knowledge this is the first time a reasonably precise notion of ‘method’ with a similarly reasonably precise notion of a calculi of tools has been backed up formal definitions.

3.9.2 Are the Models Valid ?

Are the formal descriptions of the process of selecting and applying the analysis & description prompts, Sect. 3.4., and the meaning of these prompts, Sect. 3.8., modeling this process and these meanings realistically ? To that we can only answer the following: The process model is definitely modeling plausible processes. We discuss interpretations of the analysis & description order that this process model imposes in Sect. 3.4.3. There might be other orders, but the ones suggested in Sect. 3.4. can be said to be “orderly” and reflects empirical observations. The model of the meaning of prompts, Sect. 3.8., is more of an hypothesis. This model refers to “images” that the domain engineer is claimed to have in her mind. It must necessarily be a valid model, perhaps one of several valid models. We have speculated, over many years, over the existence of other models. But this is the most reasonable to us.

3.9.3 Future Work

We have hinted at possible ‘laws of description prompts’ in Sect. 3.4.3. Whether the process and prompt models (Sects. 3.4. and 3.8.) are sufficient to express, let alone prove such laws is an open question. If the models are sufficient, then they certainly are valid.

To Every Manifest Domain Mereology a CSP Expression

In memory of Douglas T. Ross 1929–2007¹

Summary

We give an abstract model² of parts and part-hood relations, of Stanisław Leśniewski, of software application domains such as the financial service industry, railway systems, road transport systems, health care, oil pipelines, secure [IT] systems, etcetera. We relate this model to axiom systems for mereology [68], showing satisfiability, and show that for every mereology there corresponds a class of Communicating Sequential Processes [111], that is: a λ -expression.

4.1 Introduction

The term ‘mereology’ is accredited to the Polish mathematician, philosopher and logician Stanisław Leśniewski (1886–1939) who “was a nominalist: he rejected axiomatic set theory and devised three formal systems, *Protothetic*, *Ontology*, and *Mereology* as a concrete alternative to set theory”. In this contribution I shall be concerned with only certain aspects of mereology, namely those that appears most immediately relevant to domain science (a relatively new part of current computer science). Our knowledge of ‘mereology’ has been through studying, amongst others, [68, 126].

4.1.1 Computing Science Mereology

“Mereology (from the Greek *μερος* ‘part’) is the theory of parthood relations: of the relations of part to whole and the relations of part to part within a whole”³. In this contribution we restrict ‘parts’ to be those that, firstly, are spatially distinguishable, then, secondly, while “being based” on such spatially distinguishable parts, are conceptually related. We use the term part in exactly the meaning that term was endowed in Chapter 1. The relation: “being based”, shall be made clear in this chapter.

Accordingly two parts, p_x and p_y , (of a same “whole”) are either “adjacent”, or are “embedded within” one another as loosely indicated in Fig. 4.1 on the next page.

‘Adjacent’ parts are direct parts of a same third part, p_z , i.e., p_x and p_y are “embedded within” p_z ; or one (p_x) or the other (p_y) or both (p_x and p_y) are parts of a same third part, p'_z “embedded within” p_z ; etcetera; as loosely indicated in Fig. 4.2 on the following page. or one is “embedded within” the other — etc. as loosely indicated in Fig. 4.2 on the next page.

Parts, whether adjacent or embedded within one another, can share properties. For adjacent parts this sharing seems, in the literature, to be diagrammatically expressed by letting the part rectangles “intersect”. Usually properties are not spatial hence ‘intersection’ seems confusing. We refer to Fig. 4.3 on the following page.

¹ See the big paragraph first in Sect. 4.7.1 on Page 130.

² This chapter is based on [50]. That paper is a complete rewrite of [44]. The rewritten sections are marked with vertical margin bars, as here !

³ Achille Varzi: Mereology, <http://plato.stanford.edu/entries/mereology/> 2009 and [68]

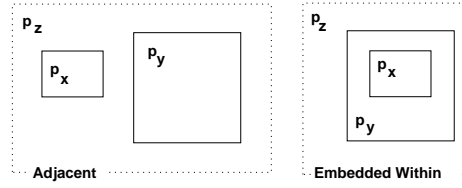


Fig. 4.1. ‘Adjacent’ and “Embedded Within” parts

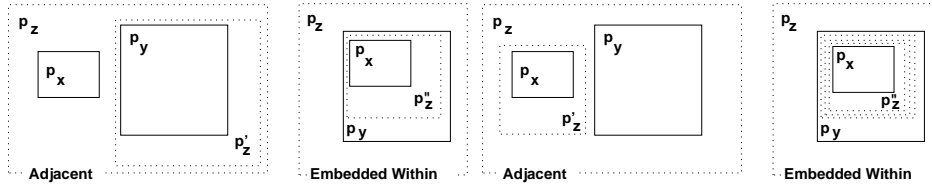


Fig. 4.2. ‘Adjacent’ and “Embedded Within” parts

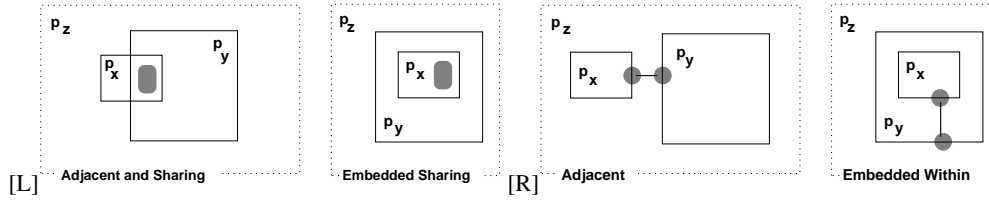


Fig. 4.3. Two models, [L,R], of parts sharing properties

Instead of depicting parts sharing properties as in Fig. 4.3[L]left, where dashed rounded edge rectangles stands for ‘sharing’, we shall (eventually) show parts sharing properties as in Fig. 4.3[R]right where $\bullet\text{---}\bullet$ connections connect those parts.

4.1.2 From Domains via Requirements to Software

One reason for our interest in mereology is that we find that concept relevant to the modeling of domains. A derived reason is that we find the modeling of domains relevant to the development of software. Conventionally a first phase of software development is that of requirements engineering. To us domain engineering is (also) a prerequisite for requirements engineering [29, 58]. Thus to properly **design** Software we need to **understand** its or their **Requirements**; and to properly **prescribe** **Requirements** one must **understand** its **Domain**. To **argue** correctness of **Software** with respect to **Requirements** one must usually **make assumptions** about the **Domain**: $\mathbb{D}, \mathbb{S} \models \mathbb{R}$. Thus **description** of **Domains** become an indispensable part of **Software** development.

4.1.3 Domains: Science and Engineering

Domain Science is the study and knowledge of domains. **Domain Engineering** is the practice of “**walking the bridge**” from domain science to domain descriptions: to **create domain descriptions** on the background of scientific knowledge of domains, the specific domain “at hand”, or domains in general; and to **study domain descriptions** with a view to broaden and deepen scientific results about domain descriptions. This contribution is based on the engineering and study of many descriptions, of air traffic, banking, commerce (the consumer/retailer/wholesaler/producer supply chain), container lines, health care, logistics, pipelines, railway systems, secure [IT] systems, stock exchanges, etcetera.

4.1.4 Contributions of This Chapter

A general contribution is that of providing elements of a domain science. Three specific contributions are those of (i) giving a model that satisfies published formal, axiomatic characterisations of mereology; (ii) showing that to every (such modeled) mereology there corresponds a CSP [111] program; and (iii) suggesting complementing **syntactic** and **semantic** theories of mereology.

4.1.5 Structure of This Chapter

We briefly overview the structure of this contribution. First, on Sect. 4.2, **we loosely characterise how we look at mereologies: “what they are to us !”**. Then, in Sect. 4.3, **we give an abstract, model-oriented specification of a class of mereologies** in the form of composite parts and composite and atomic subparts and their possible connections. The abstract model as well as the axiom system (Sect. 4.5) focuses on the **syntax of mereologies**. Following that, in Sect. 4.5 **we indicate how the model of Sect. 4.3 satisfies the axiom system of that section**. In preparation for Sect. 4.6 **presents characterisations of attributes of parts, whether atomic or composite**. Finally Sect. 4.6 presents **a semantic model of mereologies**, one of a wide variety of such possible models. This one emphasize the possibility of considering parts and subparts as processes and hence a mereology as a system of processes. Section 4.7 concludes with some remarks on what we have achieved.

4.2 Our Concept of Mereology

4.2.1 Informal Characterisation

Mereology, to us, is the study and knowledge about how physical and conceptual parts relate and what it means for a part to be related to another part: *being disjoint, being adjacent, being neighbours, being contained properly within, being properly overlapped with*, etcetera. By physical parts we mean such spatial individuals which can be pointed to. **Examples:** *a road net (consisting of street segments and street intersections); a street segment (between two intersections); a street intersection; a road (of sequentially neighbouring street segments of the same name) a vehicle; and a platoon (of sequentially neighbouring vehicles).*

By a conceptual part we mean an abstraction with no physical extent, which is either present or not. **Examples:** *a bus timetable (not as a piece or booklet of paper, or as an electronic device, but) as an image in the minds of potential bus passengers; and routes of a pipeline, that is, neighbouring sequences of pipes, valves, pumps, forks and joins, for example referred to in discourse: “the gas flows through “such-and-such” a route”*. The tricky thing here is that a route may be thought of as being both a concept or being a physical part — in which case one ought give them different names: a planned route and an actual road, for example.

The mereological notion of subpart, that is: *contained within* can be illustrated by **examples:** *the intersections and street segments are subparts of the road net; vehicles are subparts of a platoon; and pipes, valves, pumps, forks and joins are subparts of pipelines*. The mereological notion of adjacency can be illustrated by **examples.** We consider the various controls of an air traffic system, cf. Fig. 4.4 on the next page, as well as its aircraft, as adjacent within the air traffic system; the pipes, valves, forks, joins and pumps of a pipeline, cf. Fig. 4.9 on Page 119, as adjacent within the pipeline system; two or more banks of a banking system, cf. Fig. 4.6 on Page 117, as being adjacent. The mereo-topological notion of neighbouring can be illustrated by **examples:** *Some adjacent pipes of a pipeline are neighbouring (connected) to other pipes or valves or pumps or forks or joins, etcetera; two immediately adjacent vehicles of a platoon are neighbouring*. The mereological notion of proper overlap can be illustrated by **examples** some of which are of a general kind: *two routes of a pipelines may overlap; and two conceptual bus timetables may overlap with some, but not all bus line entries being the same; and some of really reflect adjacency: two adjacent pipe overlap in their connection, a wall between two rooms overlap each of these rooms — that is, the rooms overlap each other “in the wall”*.

4.2.2 Six Examples

We shall, in Sect. 4.3, present a model that is claimed to abstract essential mereological properties of air traffic, buildings and their installations, machine assemblies, financial service industry, the oil industry and oil pipelines, and railway nets.

Air Traffic

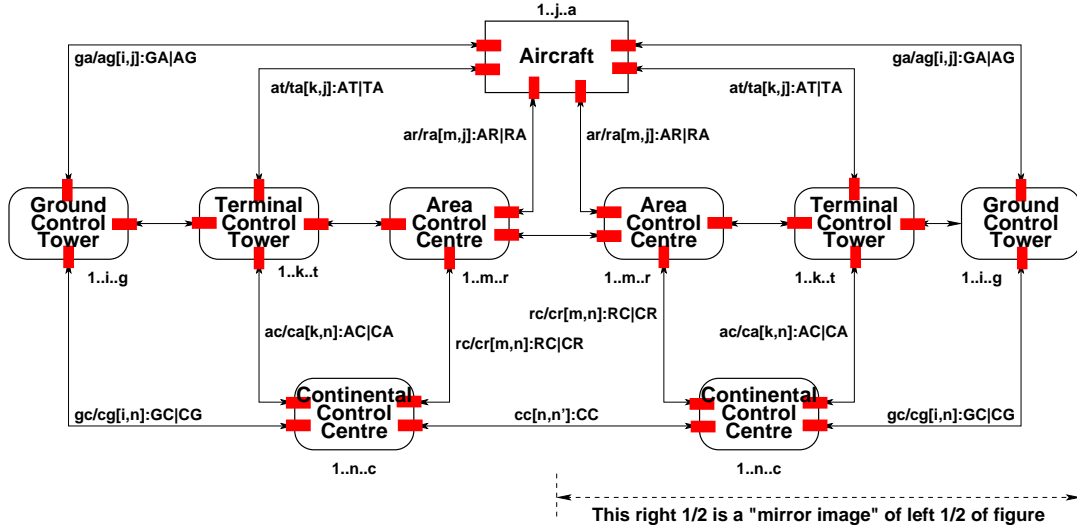


Fig. 4.4. A schematic air traffic system

Figure 4.4 shows nine adjacent (9) boxes and eighteen adjacent (18) lines. Boxes and lines are parts. The line parts “neighbours” the box parts they “connect”. Individually boxes and lines represent adjacent parts of the composite air traffic “whole”. The rounded corner boxes denote buildings. The sharp corner box denote an aircraft. Lines denote radio telecommunication. The “overlap” between neighbouring line and box parts are indicated by “connectors”. Connectors are shown as small filled, narrow, either horizontal or vertical “filled” rectangle⁴ at both ends of the double-headed-arrows lines, overlapping both the line arrows and the boxes. The index ranges shown attached to, i.e., labeling each unit, shall indicate that there are a multiple of the “single” (thus representative) box or line unit shown. These index annotations are what makes the diagram of Fig. 4.4 schematic. Notice that the ‘box’ parts are fixed installations and that the double-headed arrows designate the ether where radio waves may propagate. We could, for example, assume that each such line is characterised by a combination of location and (possibly encrypted) radio communication frequency. That would allow us to consider all lines for not overlapping. And if they were overlapping, then that must have been a decision of the air traffic system.

Buildings

Figure 4.5 on the next page shows a building plan — as a composite part. The building consists of two buildings, A and H. The buildings A and H are neighbours, i.e., shares a common wall. Building A has rooms B, C, D and E, Building H has rooms I, J and K; Rooms L and M are within K. Rooms F and G are within C.

The thick lines labeled N, O, P, Q, R, S, and T models either electric cabling, water supply, air conditioning, or some such “flow” of gases or liquids.

⁴ There are 38 such rectangles in Fig. 4.4.

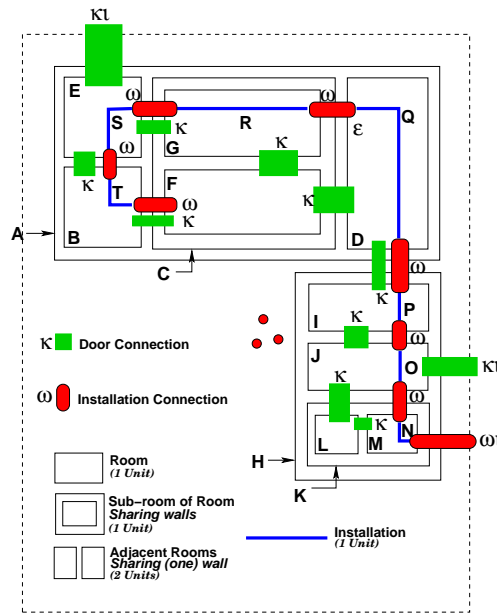


Fig. 4.5. A building plan with installation

Connection κ_{10} provides means of a connection between an environment, shown by dashed lines, and B or J, i.e. “models”, for example, a door. Connections κ provides “access” between neighbouring rooms. Note that ‘neighbouring’ is a transitive relation. Connection ω_{10} allows electricity (or water, or oil) to be conducted between an environment and a room. Connection ω allows electricity (or water, or oil) to be conducted through a wall. Etcetera.

Thus “the whole” consists of A and B. Immediate subparts of A are B, C, D and E. Immediate subparts of C are G and F. Etcetera.

Financial Service Industry

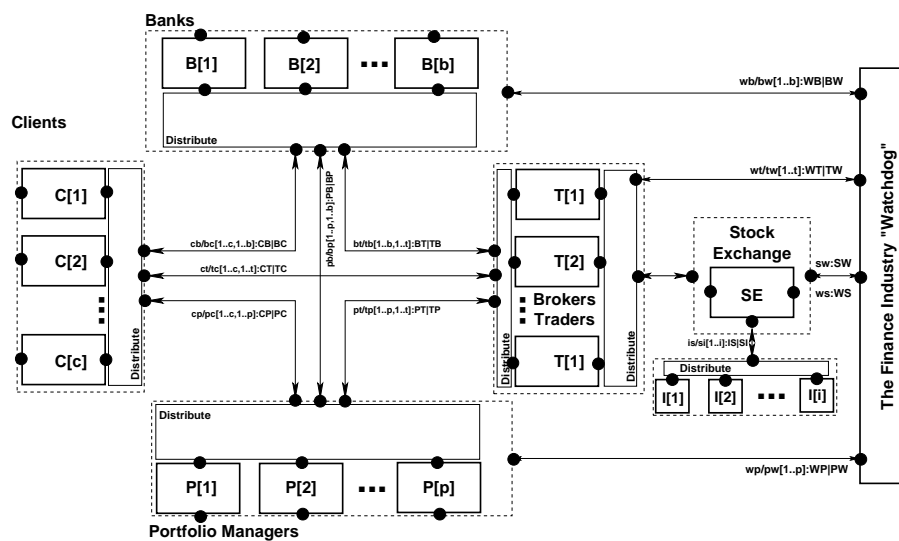


Fig. 4.6. A Financial Service Industry

Figure 4.6 on the preceding page is rather rough-sketchy! It shows seven (7) larger boxes [6 of which are shown by dashed lines], six [6] thin lined “distribution” boxes, and twelve (12) double-headed lines. Boxes and lines are parts. (We do not described what is meant by “distribution”.) Where double-headed lines touch upon (dashed) boxes we have connections. Six (6) of the boxes, the dashed line boxes, are composite parts, five (5) of them consisting of a variable number of atomic parts; five (5) are here shown as having three atomic parts each with bullets “between” them to designate “variability”. Clients, not shown, access the outermost (and hence the “innermost” boxes, but the latter is not shown) through connections, shown by bullets, •.

Machine Assemblies

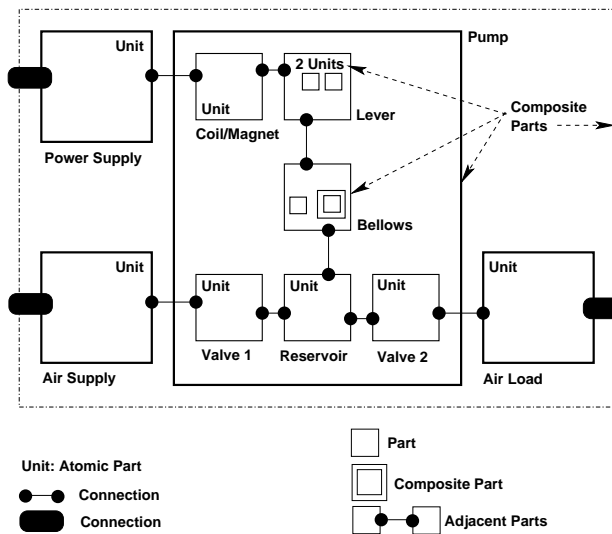


Fig. 4.7. An air pump, i.e., a physical mechanical system

Figure 4.7 shows a machine assembly. Square boxes show composite and atomic parts. Black circles or ovals show connections. The full, i.e., the level 0, composite part consists of four immediate parts and three internal and three external connections. The Pump is an assembly of six (6) immediate parts, five (5) internal connections and three (3) external connectors. Etcetera. Some connections afford “transmission” of electrical power. Other connections convey torque. Two connections convey input air, respectively output air.

Oil Industry

“The” Overall Assembly

Figure 4.8 on the facing page shows a composite part consisting of fourteen (14) composite parts, left-to-right: one oil field, a crude oil pipeline system, two refineries and one, say, gasoline distribution network, two seaports, an ocean (with oil and ethanol tankers and their sea lanes), three (more) seaports, and three, say gasoline and ethanol distribution networks.

Between all of the neighbouring composite parts there are connections, and from some of these composite parts there are connections (to an external environment). The crude oil pipeline system composite part will be concretised next.

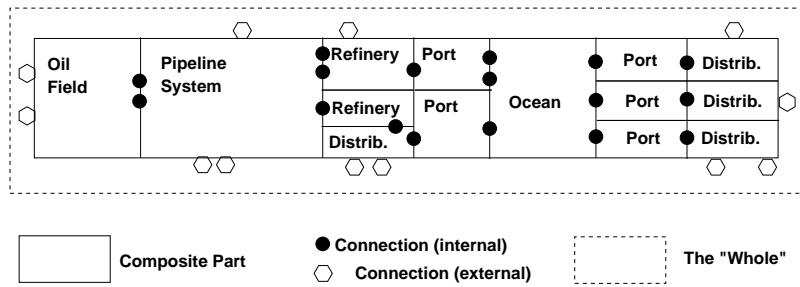


Fig. 4.8. A Schematic of an Oil Industry

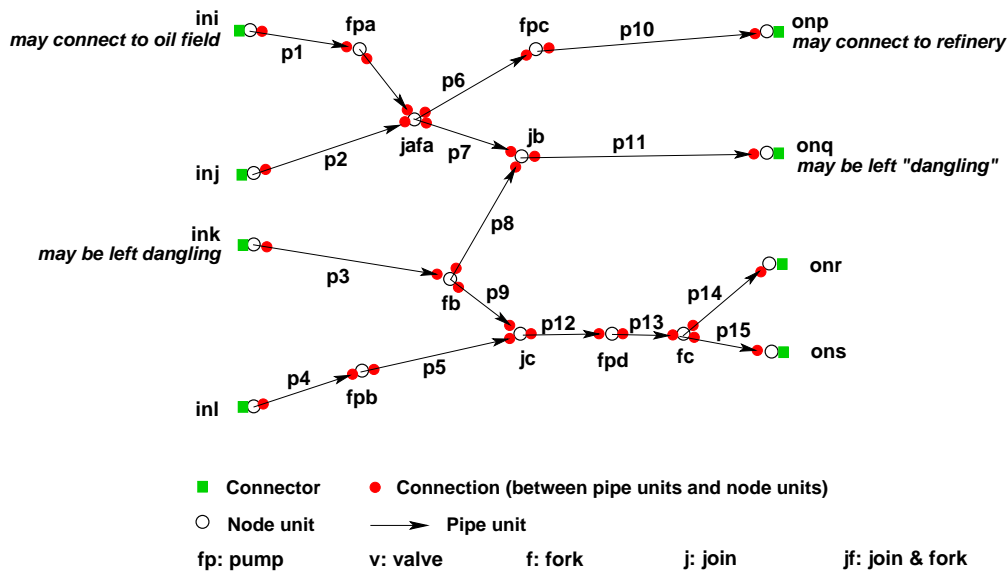


Fig. 4.9. A Pipeline System

A Concretised Composite parts

Figure 4.9 shows a pipeline system. It consists of 32 atomic parts: fifteen (15) pipe units (shown as directed arrows and labeled p1–p15), four (4) input node units (shown as small circles, \circ , and labeled ini – $in\ell$), four (4) flow pump units (shown as small circles, \circ , and labeled fpa – fpd), five (5) valve units (shown as small circles, \circ , and labeled vx – vw), three (3) join units (shown as small circles, \circ , and labeled jb – jc), two (2) fork units (shown as small circles, \circ , and labeled fb – fc), one (1) combined join & fork unit (shown as small circles, \circ , and labeled $jafa$), and four (4) output node units (shown as small circles, \circ , and labeled onp – ons).

In this example the routes through the pipeline system start with node units and end with node units, alternates between node units and pipe units, and are connected as shown by fully filled-out dark coloured disc connections. Input and output nodes have input, respectively output connections, one each, and shown as lighter coloured connections.

Railway Nets

Figure 4.10 on the following page diagrams four rail units, each with two, three or four connectors shown as narrow, somewhat “longish” rectangles. Multiple instances of these rail units can be assembled (i.e., composed) by their connectors as shown on Fig. 4.10 on the next page into proper rail nets.

Figure 4.10 on the following page diagrams an example of a proper rail net. It is assembled from the kind of units shown in Fig. 4.10. In Fig. 4.10 consider just the four dashed boxes: The dashed boxes are assembly

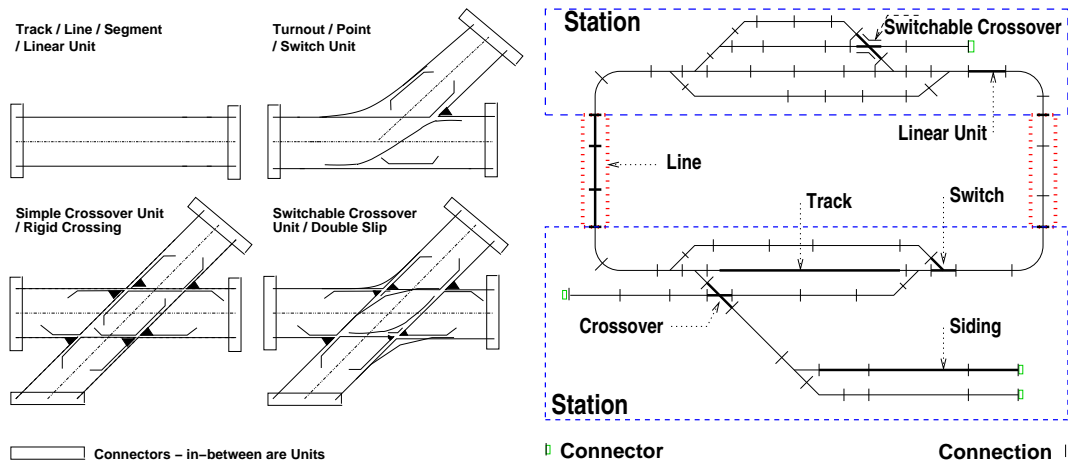


Fig. 4.10. To the left: Four rail units.

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

units. Two designate stations, two designate lines (tracks) between stations. We refer to to the caption four line text of Fig. 4.10 for more “statistics”. We could have chosen to show, instead, for each of the four “dangling” connectors, a composition of a connection, a special “end block” rail unit and a connector.

Discussion

We have brought these examples only to indicate the issues of a “whole” and atomic and composite parts, adjacency, within, neighbour and overlap relations, and the ideas of attributes and connections. We shall make the notion of ‘connection’ more precise in the next section. [184] gives URLs to a number of domain models illustrating a great variety of mereologies.

4.3 An Abstract, Syntactic Model of Mereologies

4.3.1 Parts and Subparts

168 We distinguish between **atomic** and **composite parts**.

169 Atomic parts do not contain separately distinguishable parts.

170 Composite parts contain at least one separately distinguishable part.

type

168. $P == AP \mid CP$

169. $AP :: mkA(...)$

170. $CP :: mkC(..., s_{sps}:P\text{-set})$

It is the domain analyser who decides what constitutes “the whole”, that is, how parts relate to one another, what constitutes parts, and whether a part is atomic or composite. We refer to the proper parts of a composite part as subparts. Figure 4.11 on the facing page illustrates composite and atomic parts. The *slanted sans serif* uppercase identifiers of Fig. 4.11 $A1, A2, A3, A4, A5, A6$ and $C1, C2, C3$ are meta-linguistic, that is. they stand for the parts they “decorate”; they are not identifiers of “our system”.

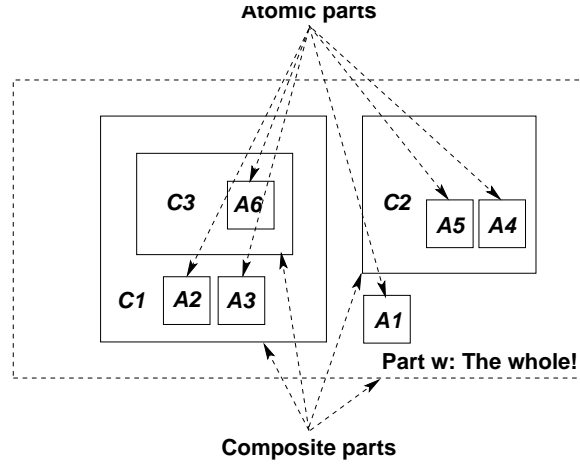


Fig. 4.11. Atomic and composite parts

4.3.2 No “Infinitely” Embedded Parts

The above syntax, Items 168–170, does not prevent composite parts, p , to contain composite parts, p' , “ad-infinitum! But we do not wish such “recursively” contained parts!

171 To express the property that parts are finite we introduce a notion of part derivation.

172 The part derivation of an atomic part is the empty set.

173 The part derivation of a composite part, p , $\text{mkC}(pq, ps)$ where pq is that composite part’s quality, is the set ps of subparts of p .

value

171. $\text{pt_der}: P \rightarrow P\text{-set}$

172. $\text{pt_der}(\text{mkA}(pq)) \equiv \{\}$

173. $\text{pt_der}(\text{mkC}(pq, ps)) \equiv ps$

174 We can also express the part derivation, $\text{pt_der}(ps)$ of a set, ps , of parts.

175 If the set is empty then $\text{pt_der}(\{\})$ is the empty set, $\{\}$.

176 Let $\text{mkA}(pq)$ be an element of ps , then $\text{pt_der}(\{\text{mkA}(pq)\} \cup ps')$ is ps' .

177 Let $\text{mkC}(pq, ps')$ be an element of ps , then $\text{pt_der}(ps' \cup ps)$ is ps' .

174. $\text{pt_der}: P\text{-set} \rightarrow P\text{-set}$

175. $\text{pt_der}(\{\}) \equiv \{\}$

176. $\text{pt_der}(\{\text{mkA}(pq)\} \cup ps) \equiv ps$

177. $\text{pt_der}(\{\text{mkC}(pq, ps')\} \cup ps) \equiv ps' \cup ps$

178 Therefore, to express that a part is finite we postulate

179 a natural number, n , such that a notion of iterated part set derivations lead to an empty set.

180 An iterated part set derivation takes a set of parts and part set derive that set repeatedly, n times.

181 If the result is an empty set, then part p was finite.

value

178. $\text{no_infinite_parts}: P \rightarrow \text{Bool}$

179. $\text{no_infinite_parts}(p) \equiv \exists n: \text{Nat} \cdot \text{it_pt_der}(\{p\})(n) = \{\}$

180. $\text{it_pt_der}: P\text{-set} \rightarrow \text{Nat} \rightarrow P\text{-set}$

181. $\text{it_pt_der}(ps)(n) \equiv \text{let } ps' = \text{pt_der}(ps) \text{ in if } n=1 \text{ then } ps' \text{ else } \text{it_pt_der}(ps')(n-1) \text{ end end}$

4.3.3 Unique Identifications

Each physical part can be uniquely distinguished for example by an abstraction of its properties at a time of origin. In consequence we also endow conceptual parts with unique identifications.

182 In order to refer to specific parts we endow all parts, whether atomic or composite, with **unique identifications**.

183 We postulate functions which observe these **unique identifications**, whether as parts in general or as atomic or composite parts in particular.

184 such that any to parts which are distinct have **unique identifications**.

type

182. UI

value

183. uid_UI: $P \rightarrow UI$

axiom

184. $\forall p, p': P \cdot p \neq p' \Rightarrow uid_UI(p) \neq uid_UI(p')$

A model for uid_UI can be given. Presupposing subsequent material (on attributes and mereology) — “lumped” into part qualities, $pq:PQ$, we augment definitions of atomic and composite parts:

type

169. AP :: mkA($s_pq:(s_uid:UI,...)$)

170. CP :: mkC($s_pq:(s_uid:UI,...), s_sps:P\text{-set}$)

value

183. $uid_UI(mkA((ui,...))) \equiv ui$

183. $uid_UI(mkC((ui,...)),...) \equiv ui$

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

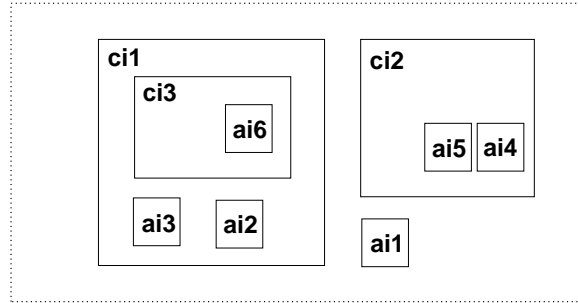


Fig. 4.12. ai_j : atomic part identifiers, ci_k : composite part identifiers

No two parts have the same unique identifier.

185 We define an auxiliary function, no_pts_uis, which applies to a[ny] part, p , and yields a pair: the number of subparts of the part argument, and the set of unique identifiers of parts within p .

186 no_pts_uis is defined in terms of yet an auxiliary function, sum_no_pts_uis.

value

185. no_pts_uis: $P \rightarrow (\text{Nat} \times UI\text{-set}) \rightarrow (\text{Nat} \times UI\text{-set})$

185. $no_pts_uis(mkA(ui,...))(n,uis) \equiv (n+1,uis \cup \{ui\})$

185. $no_pts_uis(mkC((ui,...),ps))(n,uis) \equiv \text{let } (n',uis') = \text{sum_no_pts_uis}(ps) \text{ in } (n+n',uis \cup uis') \text{ end}$

185. **pre:** no_infinite_parts(p)

186. sum_no_pts_uis: $P\text{-set} \rightarrow (\text{Nat} \times UI\text{-set}) \rightarrow (\text{Nat} \times UI\text{-set})$

```

186. sum_no_pts_uis(ps)(n,uis)  $\equiv$ 
186.   case ps of
186.     {}  $\rightarrow$  (n,uis),
186.     {mkA(ui,...)}  $\cup$  ps'  $\rightarrow$  sum_no_pts_uis(ps')(n+1,uis  $\cup$  {ui}),
186.     {mkC((ui,...),ps')}  $\cup$  ps''  $\rightarrow$ 
186.       let (n'',uis'') = sum_no_pts_uis(ps')(1,{ui}) in sum_no_pts_uis(ps'')(n+n'',uis  $\cup$  uis'') end
186.   end
186. pre:  $\forall p:P \cdot p \in ps \Rightarrow \text{no\_infinite\_parts}(p)$ 

```

187 That no two parts have the same unique identifier can now be expressed by demanding that the number of parts equals the number of unique identifiers.

axiom

187. $\forall p:P \cdot \text{let } (n,uis) = \text{no_pts_uis}(0,\{\}) \text{ in } n = \text{card } uis \text{ end}$

4.3.4 Attributes

Attribute Names and Values

188 Parts have sets of named attribute values, $\text{attrs}:\text{ATTRS}$.

189 One can observe attributes from parts.

190 Two distinct parts may share attributes:

- a For some (one or more) attribute name that is among the attribute names of both parts,
- b it is always the case that the corresponding attribute values are identical.

type

188. $\text{ANm}, \text{AVAL}, \text{ATTRS} = \text{ANm} \rightarrow \text{AVAL}$

value

189. $\text{attr_ATTRS}: P \rightarrow \text{ATTRS}$

190. $\text{share}: P \times P \rightarrow \text{Bool}$

190. $\text{share}(p,p') \equiv$

190. $p \neq p' \wedge \sim \text{trans_adj}(p,p') \wedge$

190a. $\exists \text{anm}:\text{ANm} \cdot \text{anm} \in \text{dom } \text{attr_ATTRS}(p) \cap \text{dom } \text{attr_ATTRS}(p') \Rightarrow$

190b. $\square (\text{attr_ATTRS}(p))(\text{anm}) = (\text{attr_ATTRS}(p'))(\text{anm})$

The function trans_adj is defined in Sect. 4.4.4 on Page 126.

Attribute Categories

One can suggest a hierarchy of part attribute categories: static or dynamic values — and within the dynamic value category: inert values or reactive values or active values — and within the dynamic active value category: autonomous values or biddable values or programmable values. By a **static attribute**, $a:A$, $\text{is_static_attribute}(a)$, we shall understand an attribute whose values are constants, i.e., cannot change. By a **dynamic attribute**, $a:A$, $\text{is_dynamic_attribute}(a)$, we shall understand an attribute whose values are variable, i.e., can change. By an **inert attribute**, $a:A$, $\text{is_inert_attribute}(a)$, we shall understand a dynamic attribute whose values only change as the result of external stimuli where these stimuli prescribe properties of these new values. By a **reactive attribute**, $a:A$, $\text{is_reactive_attribute}(a)$, we shall understand a dynamic attribute whose values, if they vary, change value in response to the change of other attribute values. By an **active attribute**, $a:A$, $\text{is_active_attribute}(a)$, we shall understand a dynamic attribute whose values change (also) of its own volition. By an **autonomous attribute**, $a:A$, $\text{is_autonomous_attribute}(a)$, we shall understand a dynamic active attribute whose values change value only “on their own volition”. The values of an autonomous attributes are a “law unto themselves and their

surroundings”. By a **biddable attribute**, $a:A$, $\text{is_biddable_attribute}(a)$, (of a part) we shall understand a dynamic active attribute whose values are prescribed but may fail to be observed as such. By a **programmable attribute**, $a:A$, $\text{is_programmable_attribute}(a)$, we shall understand a dynamic active attribute whose values can be prescribed. By an **external attribute** we mean inert, reactive, active or autonomous attribute. By a **controllable attribute** we mean a biddable or programmable attribute.

We define some auxiliary functions:

- 191 \mathcal{S}_A applies to attrs:ATTRS and yields a grouping $(sa_1, sa_2, \dots, sa_{n_s})^5$, of **static** attribute values.
 192 \mathcal{C}_A applies to attrs:ATTRS and yields a grouping $(ca_1, ca_2, \dots, ca_{n_c})^6$ of **controllable** attribute values.
 193 \mathcal{E}_A applies to attrs:ATTRS and yields a set, $\{eA_1, eA_2, \dots, eA_{n_e}\}^7$ of **external** attribute names.

type

$SA, CA = \text{AVAL}^*$

$EA = \text{ANm-st}$

value

191. $\mathcal{S}_A: \text{ATTRS} \rightarrow SA$

192. $\mathcal{C}_A: \text{ATTRS} \rightarrow CA$

193. $\mathcal{E}_A: \text{ATTRS} \rightarrow EA$

The attribute names of static, controllable and external attributes do not overlap and together make up the attribute names of attrs .

4.3.5 Mereology

In order to illustrate other than the within and adjacency part relations we introduce the notion of mereology. Figure 4.13 illustrates a mereology between parts. A specific mereology-relation is, visually, a $\bullet \text{---} \bullet$ line that connects two distinct parts.

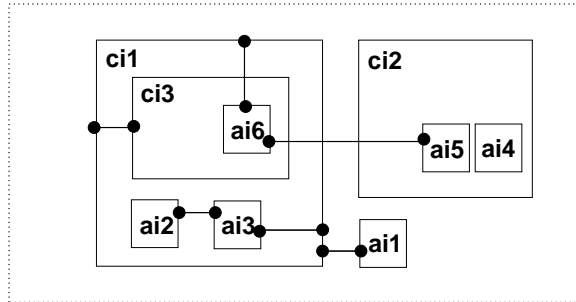


Fig. 4.13. Mereology: Relations between Parts

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

type

194. $ME = \text{UI-set}$

We may refer to the connectors by the two element sets of the unique identifiers of the parts they connect. For **example** with respect to Fig. 4.13:

⁵ – where $\{sa_1, sa_2, \dots, sa_{n_s}\} \subseteq \text{rng attrs}$

⁶ – where $\{ca_1, ca_2, \dots, ca_{n_c}\} \subseteq \text{rng attrs}$

⁷ – where $\{eA_1, eA_2, \dots, eA_{n_e}\} \subseteq \text{dom attrs}$

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

4.3.6 The Model

195 The “whole” is a part.

196 A part value has a part sort name and is either the value of an atomic part or of an abstract composite part.

197 A atomic part value has a part quality value.

198 An abstract composite part value has a part quality value and a set of at least of one or more part values.

199 A part quality value consists of a unique identifier, a mereology, and a set of one or more attribute named attribute values.

```

195  W = P
196  P = AP | CP
197  AP :: mkA(s_pq:PQ)
198  CP :: mkC(s_pq:PQ,s_ps:P-set)
199  PQ = UI×ME×(ANm  $\rightarrow$  AVAL)

```

We now assume that parts are not “recursively infinite”, and that all parts have unique identifiers

4.4 Some Part Relations

4.4.1 ‘Immediately Within’

200 One part, p , is said to be *immediately within*, $\text{imm_within}(p, p')$, another part, if p' is a composite part and p is observable in p' .

value

```

200.  imm_within: P × P → Bool
200.  imm_within(p, p') ≡
200.    case p' of
200.      (__, mkA(__, ps)) → p ∈ ps,
200.      (__, mkC(__, ps)) → p ∈ ps,
200.      _ → false
200.  end

```

4.4.2 ‘Transitive Within’

We can generalise the ‘immediate within’ property.

201 A part, p , is transitively within a part p' , $\text{trans_within}(p, p')$,
 a either if p , is immediately within p'
 b or
 c if there exists a (proper) composite part p'' of p' such that $\text{trans_within}(p'', p)$.

value

```

201.  trans_wihin: P × P → Bool
201.  trans_within(p, p') ≡
201a.    imm_within(p, p')
201b.    ∨
201c.    case p' of
201c.      (__, mkC(__, ps)) → p ∈ ps ∧
201c.        ∃ p'':P• p'' ∈ ps ∧ trans_within(p'', p),
201c.      _ → false
201.  end

```

4.4.3 'Adjacency'

202 Two parts, p, p' , are said to be *immediately adjacent*, $\text{imm_adj}(p, p')(c)$, to one another, in a composite part c , such that p and p' are distinct and observable in c .

value

202. $\text{imm_adj}: P \times P \rightarrow P \rightarrow \mathbf{Bool}$
 202. $\text{imm_adj}(p, p')(\text{mkA}(_, ps)) \equiv p \neq p' \wedge \{p, p'\} \subseteq ps$
 202. $\text{imm_adj}(p, p')(\text{mkC}(_, ps)) \equiv p \neq p' \wedge \{p, p'\} \subseteq ps$
 202. $\text{imm_adj}(p, p')(\text{mkA}(_)) \equiv \mathbf{false}$

4.4.4 Transitive 'Adjacency'

We can generalise the immediate 'adjacent' property.

- 203 Two parts, p', p'' , of a composite part, p , are $\text{trans_adj}(p', p'')$ in p
 a either if $\text{imm_adj}(p', p'')(p)$,
 b or if there are two p''' and p'''' such that
 i p''' and p'''' are immediately adjacent parts of p and
 ii p is equal to p''' or p''' is properly within p and p' is equal to p'''' or p'''' is properly within p'

We leave the formalisation to the reader.

4.5 An Axiom System

Classical axiom systems for mereology focus on just one sort of "things", namely \mathcal{P} arts. Leśniewski had in mind, when setting up his mereology to have it supplant set theory. So parts could be composite and consisting of other, the sub-parts — some of which would be atomic; just as sets could consist of elements which were sets — some of which would be empty.

4.5.1 Parts and Attributes

In our axiom system for mereology we shall avail ourselves of two sorts: \mathcal{P} arts, and \mathcal{A} tttributes.⁸

- type \mathcal{P}, \mathcal{A}

\mathcal{A} tttributes are associated with \mathcal{P} arts. We do not say very much about attributes: We think of attributes of parts to form possibly empty sets. So we postulate a primitive predicate, \in , relating \mathcal{P} arts and \mathcal{A} tttributes.

- $\in: \mathcal{A} \times \mathcal{P} \rightarrow \mathbf{Bool}$.

4.5.2 The Axioms

The axiom system to be developed in this section is a variant of that in [68]. We introduce the following relations between parts:

$\text{part_of}: P: \mathcal{P} \times \mathcal{P} \rightarrow \mathbf{Bool}$	Page 127
$\text{proper_part_of}: PP: \mathcal{P} \times \mathcal{P} \rightarrow \mathbf{Bool}$	Page 127
$\text{overlap}: O: \mathcal{P} \times \mathcal{P} \rightarrow \mathbf{Bool}$	Page 127
$\text{underlap}: U: \mathcal{P} \times \mathcal{P} \rightarrow \mathbf{Bool}$	Page 127
$\text{over_crossing}: OX: \mathcal{P} \times \mathcal{P} \rightarrow \mathbf{Bool}$	Page 127
$\text{under_crossing}: UX: \mathcal{P} \times \mathcal{P} \rightarrow \mathbf{Bool}$	Page 127
$\text{proper_overlap}: PO: \mathcal{P} \times \mathcal{P} \rightarrow \mathbf{Bool}$	Page 127
$\text{proper_underlap}: PU: \mathcal{P} \times \mathcal{P} \rightarrow \mathbf{Bool}$	Page 127

⁸ Identifiers P and A stand for model-oriented types (parts and atomic parts), whereas identifiers \mathcal{P} and \mathcal{A} stand for property-oriented types (parts and attributes).

Let \mathbb{P} denote **part-hood**; p_x is part of p_y , is then expressed as $\mathbb{P}(p_x, p_y)$.⁹ (4.1) Part p_x is part of itself (reflexivity). (4.2) If a part p_x is part p_y and, vice versa, part p_y is part of p_x , then $p_x = p_y$ (anti-symmetry). (4.3) If a part p_x is part of p_y and part p_y is part of p_z , then p_x is part of p_z (transitivity).

$$\forall p_x : \mathcal{P} \bullet \mathbb{P}(p_x, p_x) \quad (4.1)$$

$$\forall p_x, p_y : \mathcal{P} \bullet (\mathbb{P}(p_x, p_y) \wedge \mathbb{P}(p_y, p_x)) \Rightarrow p_x = p_y \quad (4.2)$$

$$\forall p_x, p_y, p_z : \mathcal{P} \bullet (\mathbb{P}(p_x, p_y) \wedge \mathbb{P}(p_y, p_z)) \Rightarrow \mathbb{P}(p_x, p_z) \quad (4.3)$$

Let \mathbb{PP} denote **proper part-hood**. p_x is a proper part of p_y is then expressed as $\mathbb{PP}(p_x, p_y)$. \mathbb{PP} can be defined in terms of \mathbb{P} . $\mathbb{PP}(p_x, p_y)$ holds if p_x is part of p_y , but p_y is not part of p_x .

$$\mathbb{PP}(p_x, p_y) \triangleq \mathbb{P}(p_x, p_y) \wedge \neg \mathbb{P}(p_y, p_x) \quad (4.4)$$

Overlap, \mathbb{O} , expresses a relation between parts. Two parts are said to overlap if they have “something” in common. In classical mereology that ‘something’ is parts. To us parts are spatial entities and these cannot “overlap”. Instead they can ‘share’ attributes.

$$\mathbb{O}(p_x, p_y) \triangleq \exists a : \mathcal{A} \bullet a \in p_x \wedge a \in p_y \quad (4.5)$$

Underlap, \mathbb{U} , expresses a relation between parts. Two parts are said to underlap if there exists a part p_z of which p_x is a part and of which p_y is a part.

$$\mathbb{U}(p_x, p_y) \triangleq \exists p_z : \mathcal{P} \bullet \mathbb{P}(p_x, p_z) \wedge \mathbb{P}(p_y, p_z) \quad (4.6)$$

Think of the underlap p_z as an “umbrella” which both p_x and p_y are “under”.

Over-cross, \mathbb{OX} , p_x and p_y are said to over-cross if p_x and p_y overlap and p_x is not part of p_y .

$$\mathbb{OX}(p_x, p_y) \triangleq \mathbb{O}(p_x, p_y) \wedge \neg \mathbb{P}(p_x, p_y) \quad (4.7)$$

Under-cross, \mathbb{UX} , p_x and p_y are said to under cross if p_x and p_y underlap and p_y is not part of p_x .

$$\mathbb{UX}(p_x, p_y) \triangleq \mathbb{U}(p_x, p_y) \wedge \neg \mathbb{P}(p_y, p_x) \quad (4.8)$$

Proper Overlap, \mathbb{PO} , expresses a relation between parts. p_x and p_y are said to properly overlap if p_x and p_y over-cross and if p_y and p_x over-cross.

$$\mathbb{PO}(p_x, p_y) \triangleq \mathbb{OX}(p_x, p_y) \wedge \mathbb{OX}(p_y, p_x) \quad (4.9)$$

Proper Underlap, \mathbb{PU} , p_x and p_y are said to properly underlap if p_x and p_y under-cross and p_x and p_y under-cross.

$$\mathbb{PU}(p_x, p_y) \triangleq \mathbb{UX}(p_x, p_y) \wedge \mathbb{UX}(p_y, p_x) \quad (4.10)$$

4.5.3 Satisfaction

We shall sketch a proof that the *model* of the previous section, Sect. 4.3, *satisfies* is a model for the *axioms* of this section. To that end we first define the notions of *interpretation*, *satisfiability*, *validity* and *model*. **Interpretation**: By an interpretation of a predicate we mean an assignment of a truth value to the predicate where the assignment may entail an assignment of values, in general, to the terms of the predicate. **Satisfiability**: By the satisfiability of a predicate we mean that the predicate is true for some interpretation. **Valid**: By the validity of a predicate we mean that the predicate is true for all interpretations. **Model**: By a model of a predicate we mean an interpretation for which the predicate holds.

⁹ Our notation now is not RSL but a conventional first-order predicate logic notation.

A Proof Sketch

We assign

- 204 P as the meaning of \mathcal{P}
- 205 ATR as the meaning of \mathcal{A} ,
- 206 imm_within as the meaning of \mathbb{P} ,
- 207 trans_within as the meaning of \mathbb{PP} ,
- 208 \in : ATTR \times ATTRS-set \rightarrow Bool as the meaning of \in : $\mathcal{A} \times \mathcal{P} \rightarrow \mathbf{Bool}$ and
- 209 sharing as the meaning of \mathbb{O} .

With the above assignments it is now easy to prove that the other axiom-operators U, PO, PU, OX and UX can be modeled by means of imm_within, within, ATTR \times ATTRS-set \rightarrow Bool and sharing.

4.6 A Semantic CSP Model of Mereology

The model of Sect. 4.3 can be said to be an abstract model-oriented definition of the syntax of mereology. Similarly the axiom system of Sect. 4.5 can be said to be an abstract property-oriented definition of the syntax of mereology.

We show that to every mereology there corresponds a program of cooperating sequential processes CSP. We assume that the reader has practical knowledge of Hoare's CSP [111].

4.6.1 Parts \simeq Processes

The model of mereology presented in Sect. 4.3 focused on (i) parts, (ii) unique identifiers and (iii) mereology. To parts we associate CSP processes. Part processes are indexed by the unique part identifiers. The mereology reveals the structure of CSP channels between CSP processes.

4.6.2 Channels

We define a general notion of a vector of channels. One vector element for each "pair" of distinct unique identifiers. Vector indices are set of two distinct unique identifiers.

- 210 Let w be the "whole" (i.e., a part).
- 211 Let uis be the set of all unique identifiers of the "whole".
- 212 Let M be the type of messages sent over channels.
- 213 Channels provide means for processes to synchronise and communicate.

value

- 210. w:P
- 211. uis = **let** ($_, uis'$)=no_parts_uis(w) **in** uis' **end**

type

- 212. M

channel

- 213. $\{ch[\{ui, ui'\}]: M \mid ui, ui': U \bullet ui \neq ui' \wedge \{ui, ui'\} \subseteq uis\}$

- 214 We also define channels for access to external attribute values.

Without loss of generality we do so for all possible parts and all possible attributes.

channel

- 214. $\{xch[ui, an]: AVAL \mid ui: U \bullet ui \in uis, an: AN_m\}$

4.6.3 Compilation

We now show how to compile “real-life, actual” parts into **RSL-Text**. That is, turning “semantics” into syntax !

value

```
compile_P: P → RSL-Text
compile_P(mkA(ui,me,attrs)) ≡ `  $\mathcal{M}_a(ui,me,attrs)$  '
compile_P(mkC((ui,me,attrs),{p1,p2,...,pn})) ≡
  `  $\mathcal{M}_c(ui,me,attrs) \parallel \text{compile\_process}(p_1) \parallel \text{compile\_process}(p_2) \parallel \dots \parallel \text{compile\_process}(p_n)$  '

```

The ‘core’ processes \mathcal{M}_a and \mathcal{M}_c relate to *atomic* and *composite* parts. They are defined, schematically, below as just \mathcal{M} .

value

```
 $\mathcal{M}$ : ui:UI × me:ME × attrs:ATTRS → ca:  $\mathcal{C}_a(attrs)$  → RSL-Text
 $\mathcal{M}(ui,me,attrs)(ca) \equiv$  ` let (me',ca') =  $\mathcal{F}(ui,me,attrs)(ca)$  in  $\mathcal{M}(ui,me',attrs)(ca')$  end '
value '
 $\mathcal{F}$ : ui:UI × me:ME × attrs:ATTRS → ca:CA → in in_chs(ui,attrs) in, out in_out_chs(ui,me) → ME × CA '

```

Recall (Page 124) that $\mathcal{C}_a(attrs)$ is a grouping, $(ca_1, ca_2, \dots, ca_{n_c})$, of controlled attribute values.

215 The `in_chs` function applies to a set of uniquely named attributes and yields some **RSL-Text**, in the form of **input** channel declarations, one for each external attribute.

```
215. in_chs: ui:UI × attrs:ATTRS → RSL-Text
215. in_chs(ui,attrs) ≡ ` in { xch[ui,xai] | xai:ANm • xai ∈  $\mathcal{C}_a(attrs)$  } '

```

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

```
216. in_out_chs: ui:UI × me:ME → RSL-Text
216. in_out_chs(ui,me) ≡ ` in,out { xch[ui,ui'] | ui:UI • ui' ∈ me } '

```

\mathcal{F} is an action: it returns a possibly updated mereology and possibly updated controlled attribute values. We present a rough sketch of \mathcal{F} . The \mathcal{F} action non-deterministically internal choice chooses between

- either [1,2,3,4]
 - ⊗ [1] accepting input from
 - ⊗ [4] a suitable (“offering”) part process,
 - ⊗ [2] optionally offering a reply;
 - ⊗ [3] leading to an updated state;
- or [3,4]
 - ⊗ [5] finding a suitable “order” (val)
 - ⊗ [8] to a suitable (“inquiring”) behaviour,
 - ⊗ [6] offering that value,
 - ⊗ [7] leading to an updated state;
- or [9] doing own work leading to a new state.

value

```
 $\mathcal{F}(ui,me,attrs)(ca) \equiv$ 
[1]   □ { let val = ch[{ui,ui'}]? in
[2]     (ch[{ui,ui'}]!in_reply(val,(ui,me,attrs))(ca)) ;
[3]     in_update(val,(ui,me,attrs))(ca) end
[4]   | ui':UI • ui' ∈ me }
[5]   □ □ { let val = await_reply(ui',me,attrs)(ca) in
[6]     ch[{ui,ui'}]!val ;
[7]     out_update(val,(ui,me,attrs))(ca) end
[8]   | ui':UI • ui' ∈ me }
[9]   □ (me,own_work(ui,attrs))(ca)

```

channels $ch[ui, ui']$ are defined in **in** $in_chs(ea:EA)$ **in,out** $in_out_chs(me:ME)$

$in_reply: VAL \times (ui:UI \times me:ME \times attrs:ATTRS) \rightarrow ca:CA \rightarrow$
 $\quad \quad \quad \mathbf{in} \ in_chs(attrs) \ \mathbf{in,out} \ in_out_chs(ui,me) \rightarrow VAL$

$in_update: VAL \times (ui:UI \times me:ME \times attrs:ATTRS) \rightarrow ca:CA \rightarrow$
 $\quad \quad \quad \mathbf{in,out} \ in_out_chs(ui,me) \rightarrow ME \times CA$

$await_reply: (ui:UI, me:ME) \rightarrow ca:CA \rightarrow \mathbf{in,out} \ in_out_chs(ui,me:ME) \rightarrow VAL$

$out_update: (VAL \times (ui:UI \times me:ME <> attrs:ATTRS)) \rightarrow ca:CA \rightarrow$
 $\quad \quad \quad \mathbf{in,out} \ in_out_chs(ui,me) \rightarrow ME \times CA$

$own_work: (ui:UI \times attrs:ATTRS) \rightarrow CA \rightarrow \mathbf{in,out} \ in_out_chs(ui,me) \ CA$

4.6.4 Discussion

General

A little more meaning has been added to the notions of parts and their mereology. The within and adjacent to relations between parts (composite and atomic) reflect a phenomenological world of geometry, and the mereological relation between parts reflect both physical and conceptual world understandings: physical world in that, for example, radio waves cross geometric “boundaries”, and conceptual world in that ontological classifications typically reflect lattice orderings where *overlaps* likewise cross geometric “boundaries”.

Specific

The notion of parts is far more general than that of, for example, Sect. 3.6.2 on Page 101. We have been able to treat Stanisław Leśniewski’s notion of mereology solely based on parts, that is, their semantic values, without introducing the notion of the syntax of parts. Our compilation functions are (thus) far more general than need (for example as needed in Sect. 3.6.2 on Page 101).

4.7 Concluding Remarks

4.7.1 Relation to Other Work

The present contribution has been conceived in the following context.

My first awareness of the concept of ‘mereology’ was from listening to many presentations by **Douglas T. Ross** (1929–2007) at IFIP working group WG 2.3 meetings over the years 1980–1999. In [161] Douglas T. Ross and John E. Ward reports on the 1958–1967 MIT project for *computer-aided design (CAD) for numerically controlled production*.¹⁰ Pages 13–17 of [161] reflects on issues bordering to and behind the concerns of mereology. Ross’ thinking is clearly seen in the following text: “... *our consideration of fundamentals begins not with design or problem-solving or programming or even mathematics, but with philosophy (in the old-fashioned meaning of the word) – we begin by establishing a “world-view”. We have repeatedly emphasized that there is no way to bound or delimit the potential areas of application of our system, and that we must be prepared to cope with any conceivable problem. Whether the system will assist in any way in the solution of a given problem is quite another matter, ..., but in order to have a firm and uniform foundation, we must have a uniform philosophical basis upon which to approach any given problem. This “world-view” must provide a working framework and methodology in terms of which any aspect of our awareness of the world may be viewed. It must be capable of expressing the utmost in reality, giving expression to unending layers of ever-finer and*

¹⁰ Doug is said to have coined the term and the abbreviation CAD [159].

more concrete detail, but at the same time abstract chimerical¹¹ visions bordering on unreality must fall within the same scheme. “Above all, the world-view itself must be concrete and workable, for it will form the basis for all involvement of the computer in the problem-solving process, as well as establishing a viewpoint for approaching the unknown human component of the problem-solving team.” Yes, indeed, the philosophical disciplines of ontology, epistemology and mereology, amongst others, ought be standard curricula items in the computer science and software engineering studies, or better: domain engineers cum software system designers ought be imbued by the wisdom of those disciplines as was Doug. “... in the summer of 1960 we coined the word *plex* to serve as a generic term for these philosophical ruminations. “*Plex*” derives from the word *plexus*, “An interwoven combination of parts in a structure”, (Webster). ... The purpose of a ‘**modeling plex**’ is to represent completely and in its entirety a “thing”, whether it is concrete or abstract, physical or conceptual. A ‘*modeling plex*’ is a trinity with three primary aspects, all of which must be present. If any one is missing a complete representation or modeling is impossible. The three aspects of *plex* are **data, structure, and algorithm**. ... ” which “... is concerned with the behavioral characteristics of the *plex* model – the interpretive rules for making meaningful the data and structural aspects of the *plex*, for assembling specific instances of the *plex*, and for interrelating the *plex* with other *plexes* and operators on *plexes*. Specification of the algorithmic aspect removes the ambiguity of meaning and interpretation of the data structure and provides a complete representation of the thing being modeled.” In the terminology of the current paper a *plex* is a part (whether composite or atomic), the data are the properties (of that part), the structure is the mereology (of that part) and the algorithm is the process (for that part). Thus Ross was, perhaps, a first instigator (around 1960) of object-orientedness. A first, “top of the iceberg” account of the mereology-ideas that Doug had then can be found in the much later (1976) three page note [160]. Doug not only ‘invented’ CAD but was also the father of AED (Algol Extended for Design), the Automatically Programmed Tool (APT) language, SADT (Structured Analysis and Design Technique) and helped develop SADT into the IDEF0 method for the Air Force’s Integrated Computer-Aided Manufacturing (ICAM) program’s IDEF suite of analysis and design methods. Douglas T. Ross went on for many years thereafter, to deepen and expand his ideas of relations between mereology and the programming language concept of type at the IFIP WG2.3 working group meetings. He did so in the, to some, enigmatic, but always fascinating style you find on Page 63 of [160].

In [127] **Henry S. Leonard** and **Henry Nelson Goodman**: *A Calculus of Individuals and Its Uses* present the American Pragmatist version of Leśniewski’s mereology. It is based on a single primitive: *discreet*. The idea of the calculus of individuals is, as in Leśniewski’s mereology, to avoid having to deal with the empty sets while relying on explicit reference to classes (or parts).

[68] **R. Casati** and **A. Varzi**: *Parts and Places: the structures of spatial representation* has been the major source for this paper’s understanding of mereology. Although our motivation was not the spatial or topological mereology, [168], and although the present paper does not utilize any of these concepts’ axiomatisation in [68, 168] it is best to say that it has benefited much from these publications.

Domain descriptions, besides mereological notions, also depend, in their successful form, on FCA: Formal Concept Analysis. Here a main inspiration has been drawn, since the mid 1990s from **B. Ganter** and **R. Wille**’s *Formal Concept Analysis — Mathematical Foundations* [95]. *The approach takes as input a matrix specifying a set of objects and the properties thereof, called attributes, and finds both all the “natural” clusters of attributes and all the “natural” clusters of objects in the input data, where a “natural” object cluster is the set of all objects that share a common subset of attributes, and a “natural” property cluster is the set of all attributes shared by one of the natural object clusters. Natural property clusters correspond one-for-one with natural object clusters, and a concept is a pair containing both a natural property cluster and its corresponding natural object cluster. The family of these concepts obeys the mathematical axioms defining a lattice, a Galois connection*). Thus the choice of adjacent and embedded (‘within’) parts and their connections is determined after serious formal concept analysis. In [59] we present a ‘concept analysis’ approach to domain description, where the present paper presents the mereological approach.

The present paper is based on [31] of which it is an extensive revision and extension.

¹¹ Chimerical: existing only as the product of unchecked imagination: fantastically visionary or improbable

4.7.2 What Has Been Achieved ?

We have given a model-oriented specification of mereology. We have indicated that the model satisfies a widely known axiom system for mereology. We have suggested that (perhaps most) work on mereology amounts to syntactic studies. So we have suggested one of a large number of possible, schematic semantics of mereology. And we have shown that to every mereology there corresponds a set of communicating sequential process (CSP).

4.7.3 Future Work

- I hereby offer collaboration with someone, say a young PhD student, to furnish a formal proof instead of the sketch outline in Sect. 4.5.3 on Page 128.
- We need to characterise, in a proper way, the class of CSP programs for which there corresponds a mereology. Are you game ?
- One could also wish for an extensive editing and publication of Doug Ross' surviving notes.

Requirements

From Domain Descriptions to Requirements Prescriptions

Summary

Chapter 1, Manifest Domains: Analysis & Description, [49] introduced a method for analysing and describing manifest domains. In this chapter¹ we show how to systematically, but, of course, not automatically, “derive” initial requirements prescriptions from domain descriptions. There are, as we see it, three kinds of requirements: (i) domain requirements, (ii) interface requirements and (iii) machine requirements. The machine is the hardware and software to be developed from the requirements. (i) **Domain requirements** are those requirements which can be expressed solely using technical terms of the domain. (ii) **Interface requirements** are those requirements which can be expressed using technical terms of both the domain and the machine. (iii) **Machine requirements** are those requirements which can be expressed solely using technical terms of the machine. We show principles, techniques and tools for “deriving” domain requirements. The domain requirements development focus on (i.1) projection, (i.2) instantiation, (i.3) determination, (i.4) extension and (i.5) fitting. These domain-to-requirements operators can be described briefly: (i.1) projection removes such descriptions which are to be omitted for consideration in the requirements, (i.2) instantiation mandates specific mereologies, (i.3) determination specifies less non-determinism, (i.4) extension extends the evolving requirements prescription with further domain description aspects and (i.5) fitting resolves “loose ends” as they may have emerged during the domain-to-requirements operations. We briefly review principles, techniques and tools for “deriving” interface requirements based on sharing domain (ii.1) endurants, and (ii.2) perdurants (i.e., actions, events and behaviours) with their machine correspondants. The unfolding of interface requirements lead to a number of machine concepts in terms of which the interface requirements are expressed. These machine concepts, both hardware and software, make possible the expression of a set of — what we shall call — **derived requirements**. The paper explores this concept briefly. We do not cover machine requirements in this paper. The reason is that we find, cf. [26, Sect. 19.6], that when the individual machine requirements are expressed then references to domain phenomena are, in fact, abstract references, that is, they do not refer to the semantics of what they name. This paper claims only to structure the quest for requirements conception. Instead of “discovering” requirements ‘ab initio’, for example, through interviews with stakeholders, we suggest to “derive” the requirements based on domain descriptions. Instead of letting the individual requirements arise out of initial stakeholder interviews, we suggest to structure these (i) around the structures of domain descriptions, and (ii) around the structures emerging from domain, interface and machine requirements. We shall refer to the requirements emerging from (i+ii) as the initial requirements. To these we add the derived requirements merging from interview with stakeholders: We are strongly of the opinion that the techniques and tools of, for example, [79, 116, 187, 120, 143, 177] can be smoothly integrated with those of this paper. We think that there is some clarification to be gained. We claim that our approach contributes to a restructuring of the field of requirements engineering and its very many diverse concerns, a structuring that is logically motivated and is based on viewing software specifications as mathematical objects.

¹ This chapter is based on [54].

5.1 Introduction

Chapter 1, Manifest Domains: Analysis & Description, [49] introduced a method for analysing and describing manifest domains. In this chapter we show how to systematically, but, of course, not automatically, “derive” requirements prescriptions from domain descriptions.

5.1.1 The Triptych Dogma of Software Development

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

5.1.2 Software As Mathematical Objects

Our base view is that computer programs are mathematical objects. That is, the text that makes up a computer program can be reasoned about. This view entails that computer program specifications can be reasoned about. And that the requirements prescriptions upon which these specifications are based can be reasoned about. This base view entails, therefore, that specifications, whether software design specifications, or requirements prescriptions, or domain descriptions, must [also] be formal specifications. This is in contrast to considering software design specifications being artifacts of sociological, or even of psychological “nature”.

5.1.3 The Contribution of This Paper

We claim that the present paper content contributes to our understanding and practice of software engineering as follows: (1) it shows how the new phase of engineering, domain engineering, as introduced in [49], forms a prerequisite for requirements engineering; (2) it endows the “classical” form of requirements engineering with a structured set of development stages and steps: (a) first a domain requirements stage, (b) to be followed by an interface requirements stages, and (c) to be concluded by a machine requirements stage; (3) it further structures and gives a reasonably precise contents to the stage of domain requirements: (i) first a projection step, (ii) then an instantiation step, (iii) then a determination step, (iv) then an extension step, and (v) finally a fitting step — with these five steps possibly being iterated; and (4) it also structures and gives a reasonably precise contents to the stage of interface requirements based on a notion of shared entities. Each of the steps (i–v) open for the possibility of simplifications. Steps (a–c) and (i–v), we claim, are new. They reflect a serious contribution, we claim, to a logical structuring of the field of requirements engineering and its very many otherwise seemingly diverse concerns.

5.1.4 Some Comments on the chapter Content

By **methodology** we understand the study and knowledge of one or more methods ² By a **method** understand the study and knowledge of the principles, techniques and tools for constructing some artifact, here (primarily) software ³ This paper is, perhaps, unusual in the following respects: (i) It is a methodology paper, hence there are no “neat” theories about development, no succinctly expressed propositions, lemmas nor theorems, and hence no proofs³. (ii) As a consequence the paper is borne by many, and by extensive examples. (iii) The examples of this paper are all focused on a generic road transport net. (iv) To reasonably fully exemplify the requirements approach, illustrating how our method copes with a seeming complexity of interrelated method aspects, the full example of this paper embodies very many description and prescription elements: hundreds of concepts (types, axioms, functions). (v) This methodology paper covers a “grand”

² The [⊙] marks the end of definitions.

³ — where these proofs would be about the development theories. The example development of requirements do imply properties, but formulation and proof of these do not constitute specifically new contributions — so are left out.

area of software engineering: Many textbooks and papers are written on *Requirements Engineering*. We postulate, in contrast to all such books (and papers), that requirements engineering should be founded on domain engineering. Hence we must, somehow, show that our approach relates to major elements of what the *Requirements Engineering* books put forward. (vi) As a result, this paper is long.

5.1.5 Structure of Paper

The structure of the paper is as follows: Section 5.2 provides a fair-sized, hence realistic example. Sections 5.3–5.5 covers our approach to requirements development. Section 5.3 overviews the issue of ‘requirements’; relates our approach (i.e., Sects. 5.4–5.5) to *systems*, *user and external equipment* and *functional requirements*; and Sect. 5.3 also introduces the concepts of the *machine* to be requirements prescribed, the *domain*, the *interface* and the *machine requirements*. Section 5.4 covers the *domain requirements* stages of *projection* (Sect. 5.4.1), *instantiation* (Sect. 5.4.2), *determination* (Sect. 5.4.3), *extension* (Sect. 5.4.4) and *fitting* (Sect. 5.4.5). Section 5.5 covers key features of *interface requirements*: shared phenomena (Sect. 5.5.1), shared endurants (Sect. 5.5.1) and shared actions, shared events and shared behaviours (Sect. 5.5.1). Section 5.5.1 further introduces the notion of derived requirements. Section 5.7 concludes the paper.

5.2 An Example Domain: Transport

In order to exemplify the various stages and steps of requirements development we first bring a domain description example. The example follows the steps of an idealised domain description. First we describe the endurants, then we describe the perdurants. Endurant description initially focus on the composite and atomic parts. Then on their “internal” qualities: unique identifications, mereologies, and attributes. The descriptions alternate between enumerated, i.e., labeled narrative sentences and correspondingly “numbered” formalisations. The narrative labels cum formula numbers will be referred to, frequently in the various steps of domain requirements development.

5.2.1 Endurants

Since we have chosen a manifest domain, that is, a domain whose endurants can be pointed at, seen, touched, we shall follow the analysis & description process as outlined in [49] and formalised in [45]. That is, we first identify, analyse and describe (manifest) parts, composite and atomic, abstract (Sect. 5.2.2) or concrete (Sect. 5.2.2). Then we identify, analyse and describe their unique identifiers (Sect. 5.2.2), mereologies (Sect. 5.2.2), and attributes (Sects. 5.2.2–5.2.2).

The example fragments will be presented in a small type-font.

5.2.2 Domain, Net, Fleet and Monitor

Applying `observe_part_sorts` [49, Sect. 3.1.6] to a transport domain $\delta:\Delta$ yields the following. The root domain, Δ , is that of a composite traffic system (217a.) with a road net, (217b.) with a fleet of vehicles and (217c.) of whose individual position on the road net we can speak, that is, monitor.⁴

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

type

217 Δ
 217a N
 217b F

⁴ The monitor can be thought of, i.e., conceptualised. It is not necessarily a physically manifest phenomenon.

217c M
value
 217a **obs_part_N**: $\Delta \rightarrow N$
 217b **obs_part_F**: $\Delta \rightarrow F$
 217c **obs_part_M**: $\Delta \rightarrow M$

Applying `observe_part_sorts` [49, Sect. 3.1.6] to a net, $n:N$, yields the following.

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

type
 218a HA
 218b LA
value
 218a **obs_part_HA**: $N \rightarrow HA$
 218b **obs_part_LA**: $N \rightarrow LA$

Hubs and Links

Applying `observe_part_types` [49, Sect. 3.1.7] to hub and link aggregates yields the following.

219 Hub aggregates are sets of hubs.
 220 Link aggregates are sets of links.
 221 Fleets are set of vehicles.

type
 219 H, HS = H-set
 220 L, LS = L-set
 221 V, VS = V-set
value
 219 **obs_part_HS**: $HA \rightarrow HS$
 220 **obs_part_LS**: $LA \rightarrow LS$
 221 **obs_part_VS**: $F \rightarrow VS$

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

value
 222a **links**: $\Delta \rightarrow L\text{-set}$
 222a **links**(δ) \equiv **obs_part_LS**(**obs_part_LA**(**obs_part_N**(δ)))
 222b **hubs**: $\Delta \rightarrow H\text{-set}$
 222b **hubs**(δ) \equiv **obs_part_HS**(**obs_part_HA**(**obs_part_N**(δ)))

Unique Identifiers

Applying `observe_unique_identifier` [49, Sect. 3.2] to the observed parts yields the following.

223 Nets, hub and link aggregates, hubs and links, fleets, vehicles and the monitor all
 a have unique identifiers
 b such that all such are distinct, and
 c with corresponding observers.

```

type
223a NI, HAI, LAI, HI, LI, FI, VI, MI
value
223c uid_NI: N → NI
223c uid_HAI: HA → HAI
223c uid_LAI: LA → LAI
223c uid_HI: H → HI
223c uid_LI: L → LI
223c uid_FI: F → FI
223c uid_VI: V → VI
223c uid_MI: M → MI
axiom
223b NI ∩ HAI = ∅, NI ∩ LAI = ∅, NI ∩ HI = ∅, etc.

```

where axiom 223b. is expressed semi-formally, in mathematics. We introduce some auxiliary functions:

224 xtr_lis extracts all link identifiers of a traffic system.
 225 xtr_his extracts all hub identifiers of a traffic system.
 226 Given an appropriate link identifier and a net get_link ‘retrieves’ the designated link.
 227 Given an appropriate hub identifier and a net get_hub ‘retrieves’ the designated hub.

```

value
224 xtr_lis: Δ → LI-set
224 xtr_lis(δ) ≡
224   let ls = links(δ) in {uid_LI(l) | l:L • l ∈ ls} end
225 xtr_his: Δ → HI-set
225 xtr_his(δ) ≡
225   let hs = hubs(δ) in {uid_HI(h) | h:H • h ∈ hs} end
226 get_link: LI → Δ → L
226 get_link(li)(δ) ≡
226   let ls = links(δ) in
226     let l:L • l ∈ ls ∧ li=uid_LI(l) in l end end
226   pre: li ∈ xtr_lis(δ)
227 get_hub: HI → Δ → H
227 get_hub(hi)(δ) ≡
227   let hs = hubs(δ) in
227     let h:H • h ∈ hs ∧ hi=uid_HI(h) in h end end
227   pre: hi ∈ xtr_his(δ)

```

Mereology

We cover the mereologies of all part sorts introduced so far. We decide that nets, hub aggregates, link aggregates and fleets have no mereologies of interest. Applying `observe_mereology` [49, Sect. 3.3.2] to hubs, links, vehicles and the monitor yields the following.

228 Hub mereologies reflect that they are connected to zero, one or more links.
 229 Link mereologies reflect that they are connected to exactly two distinct hubs.
 230 Vehicle mereologies reflect that they are connected to the monitor.
 231 The monitor mereology reflects that it is connected to all vehicles.
 232 For all hubs of any net it must be the case that their mereology designates links of that net.
 233 For all links of any net it must be the case that their mereologies designates hubs of that net.
 234 For all transport domains it must be the case that
 a the mereology of vehicles of that system designates the monitor of that system, and that
 b the mereology of the monitor of that system designates vehicles of that system.

```

value
228 obs_mereo_H: H → LI-set
229 obs_mereo_L: L → HI-set
axiom

```

```

229  $\forall l:L \cdot \text{card } \text{obs\_mereo\_L}(l)=2$ 
value
230  $\text{obs\_mereo\_V}: V \rightarrow M$ 
231  $\text{obs\_mereo\_M}: M \rightarrow VI\text{-set}$ 
axiom
232  $\forall \delta:\Delta, \text{hs}:HS \cdot \text{hs}=\text{hubs}(\delta), \text{ls}:LS \cdot \text{ls}=\text{links}(\delta) \cdot$ 
232  $\forall h:H \cdot h \in \text{hs} \cdot \text{obs\_mereo\_H}(h) \subseteq \text{xtr\_lis}(\delta) \wedge$ 
233  $\forall l:L \cdot l \in \text{ls} \cdot \text{obs\_mereo\_L}(l) \subseteq \text{xtr\_his}(\delta) \wedge$ 
234a let  $f:F \cdot f=\text{obs\_part\_F}(\delta) \Rightarrow$ 
234a let  $m:M \cdot m=\text{obs\_part\_M}(\delta),$ 
234a  $\text{vs}:VS \cdot \text{vs}=\text{obs\_part\_VS}(f)$  in
234a  $\forall v:V \cdot v \in \text{vs} \Rightarrow \text{uid\_V}(v) \in \text{obs\_mereo\_M}(m)$ 
234b  $\wedge \text{obs\_mereo\_M}(m) = \{\text{uid\_V}(v) | v:V \cdot v \in \text{vs}\}$ 
234b end end

```

Attributes, I

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

- **Hubs:** *locations*⁵ are considered static, *hub states* and *hub state spaces* are considered programmable;
- **Links:** *lengths* and *locations* are considered static, *link states* and *link state spaces* are considered programmable;
- **Vehicles:** *manufacturer name*, *engine type* (whether diesel, gasoline or electric) and *engine power* (kW/horse power) are considered static; *velocity* and *acceleration* may be considered reactive (i.e., a function of gas pedal position, etc.), *global position* (informed via a GNSS: Global Navigation Satellite System) and *local position* (calculated from a global position) are considered biddable

Applying `observe_attributes` [49, Sect. 3.4.3] to hubs, links, vehicles and the monitor yields the following.

First hubs.

- 235 Hubs
- a have geodetic locations, `GeoH`,
 - b have *hub states* which are sets of pairs of identifiers of links connected to the hub⁶,
 - c and have *hub state spaces* which are sets of hub states⁷.
- 236 For every net,
- a link identifiers of a hub state must designate links of that net.
 - b Every hub state of a net must be in the hub state space of that hub.
- 237 We introduce an auxiliary function: `xtr_lis` extracts all link identifiers of a hub state.

```

type
235a GeoH
235b  $H\Sigma = (LI \times LI)\text{-set}$ 
235c  $H\Omega = H\Sigma\text{-set}$ 
value
235a  $\text{attr\_GeoH}: H \rightarrow \text{GeoH}$ 
235b  $\text{attr\_H}\Sigma: H \rightarrow H\Sigma$ 
235c  $\text{attr\_H}\Omega: H \rightarrow H\Omega$ 
axiom
236  $\forall \delta:\Delta,$ 
236 let  $\text{hs} = \text{hubs}(\delta)$  in
236  $\forall h:H \cdot h \in \text{hs} \cdot$ 
236a  $\text{xtr\_lis}(h) \subseteq \text{xtr\_lis}(\delta)$ 
236b  $\wedge \text{attr\_}\Sigma(h) \in \text{attr\_}\Omega(h)$ 
236 end
value
237  $\text{xtr\_lis}: H \rightarrow LI\text{-set}$ 
237  $\text{xtr\_lis}(h) \equiv$ 
237  $\{li | li:LI, (li', li''): LI \times LI \cdot (li', li'') \in \text{attr\_H}\Sigma(h) \wedge li \in \{li', li''\}\}$ 

```

⁵ By location we mean a geodetic position.

⁶ A hub state “signals” which input-to-output link connections are open for traffic.

⁷ A hub state space indicates which hub states a hub may attain over time.

Then links.

238 Links have lengths.

239 Links have geodetic location.

240 Links have states and state spaces:

- a States modeled here as pairs, (hi', hi'') , of identifiers the hubs with which the links are connected and indicating directions (from hub h' to hub h'' .) A link state can thus have 0, 1, 2, 3 or 4 such pairs.
- b State spaces are the set of all the link states that a link may enjoy.

type

238 LEN

239 GeoL

240a $L\Sigma = (HI \times HI)\text{-set}$

240b $L\Omega = L\Sigma\text{-set}$

value

238 **attr**_LEN: $L \rightarrow \text{LEN}$

239 **attr**_GeoL: $L \rightarrow \text{GeoL}$

240a **attr**_LΣ: $L \rightarrow L\Sigma$

240b **attr**_LΩ: $L \rightarrow L\Omega$

axiom

240 $\forall n:N \bullet$

240 **let** ls = xtr-links(n), hs = xtr_hubs(n) **in**

240 $\forall l:L \bullet l \in \text{ls} \Rightarrow$

240a **let** $\text{ls} = \text{attr_L}\Sigma(l)$ **in**

240a $0 \leq \text{card } \text{ls} \leq 4$

240a $\wedge \forall (hi', hi''):(HI \times HI) \bullet (hi', hi'') \in \text{ls}$

240a $\Rightarrow \{hi', hi''\} = \text{obs_mereo_L}(l)$

240b $\wedge \text{attr_L}\Sigma(l) \in \text{attr_L}\Omega(l)$

240 **end end**

Then vehicles.

241 Every vehicle of a traffic system has a position which is either ‘on a link’ or ‘at a hub’.

- a An ‘on a link’ position has four elements: a unique link identifier which must designate a link of that traffic system and a pair of unique hub identifiers which must be those of the mereology of that link.
- b The ‘on a link’ position real is the fraction, thus properly between 0 (zero) and 1 (one) of the length from the first identified hub “down the link” to the second identifier hub.
- c An ‘at a hub’ position has three elements: a unique hub identifier and a pair of unique link identifiers — which must be in the hub state.

type

241 VPos = onL | atH

241a onL :: LI HI HI R

241b R = **Real** **axiom** $\forall r:R \bullet 0 \leq r \leq 1$

241c atH :: HI LI LI

value

241 **attr**_VPos: $V \rightarrow \text{VPos}$

axiom

241a $\forall n:N, \text{onL}(li, fhi, thi, r): \text{VPos} \bullet$

241a $\exists l:L \bullet l \in \text{obs_part_LS}(\text{obs_part_N}(n))$

241a $\Rightarrow li = \text{uid_L}(l) \wedge \{fhi, thi\} = \text{obs_mereo_L}(l),$

241c $\forall n:N, \text{atH}(hi, fli, tli): \text{VPos} \bullet$

241c $\exists h:H \bullet h \in \text{obs_part_HS}(\text{obs_part_N}(n))$

241c $\Rightarrow hi = \text{uid_H}(h) \wedge \{fli, tli\} \in \text{attr_L}\Sigma(h)$

242 We introduce an auxiliary function distribute.

- a distribute takes a net and a set of vehicles and
- b generates a map from vehicles to distinct vehicle positions on the net.
- c We sketch a “formal” distribute function, but, for simplicity we omit the technical details that secures distinctness — and leave that to an axiom !

243 We define two auxiliary functions:

- a `xtr_links` extracts all links of a net and
- b `xtr_hub` extracts all hubs of a net.

type

242b `MAP = VI \rightarrow_m VPos`

axiom

242b $\forall \text{map:MAP} \cdot \text{card dom map} = \text{card rng map}$

value

242 `distribute: VS \rightarrow N \rightarrow MAP`

242 `distribute(vs)(n) \equiv`

242a `let (hs,ls) = (xtr_hubs(n),xtr_links(n)) in`

242a `let vps = {onL(uidL(l),fhi,thi,r) |`

242a `l:L \cdot l \in ls \wedge {fhi,thi}`

242a `\subseteq obs_mereo_L(l) \wedge 0 \leq r \leq 1}`

242a `\cup {atH(uidH(h),fli,tli)|`

242a `h:H \cdot h \in hs \wedge {fli,tli}`

242a `\subseteq obs_mereo_H(h)} in`

242b `[uidV(v) \mapsto vp | v:V, vp:VPos \cdot v \in vs \wedge vp \in vps]`

242 `end end`

243a `xtr_links: N \rightarrow L-set`

243a `xtr_links(n) \equiv`

243a `obs_part_LS(obs_part_LA(n))`

243b `xtr_hubs: N \rightarrow H-set`

243a `xtr_hubs(n) \equiv`

243a `obs_part_H(obs_part_HA Δ (n))`

And finally monitors. We consider only one monitor attribute.

244 The monitor has a vehicle traffic attribute.

- a For every vehicle of the road transport system the vehicle traffic attribute records a possibly empty list of time marked vehicle positions.
- b These vehicle positions are alternate sequences of ‘on link’ and ‘at hub’ positions
 - i such that any sub-sequence of ‘on link’ positions record the same link identifier, the same pair of ‘to’ and ‘from’ hub identifiers and increasing fractions,
 - ii such that any sub-segment of ‘at hub’ positions are identical,
 - iii such that vehicle transition from a link to a hub is commensurate with the link and hub mereologies, and
 - iv such that vehicle transition from a hub to a link is commensurate with the hub and link mereologies.

type

244 `Traffic = VI \rightarrow_m (T \times VPos)*`

value

244 `attr_Traffic: M \rightarrow Traffic`

axiom

244b $\forall \delta:\Delta \cdot$

244b `let m = obs_part_M(δ) in`

244b `let tf = attr_Traffic(m) in`

244b `dom tf \subseteq xtr_vis(δ) \wedge`

244b $\forall \text{vi:VI} \cdot \text{vi} \in \text{dom tf} \cdot$

244b `let tr = tf(vi) in`

244b $\forall i,i+1:\text{Nat} \cdot \{i,i+1\} \subseteq \text{dom tr} \cdot$

244b `let (t,vp)=tr(i),(t',vp')=tr(i+1) in`

244b `t < t'`

244(b)i \wedge `case (vp,vp') of`

244(b)i `(onL(li,fhi,thi,r),onL(li',fhi',thi',r'))`

244(b)i $\rightarrow \text{li}=\text{li}' \wedge \text{fhi}=\text{fhi}' \wedge \text{thi}=\text{thi}' \wedge r \leq r' \wedge \text{li} \in \text{xtr_lis}(\delta) \wedge \{\text{fhi,thi}\} = \text{obs_mereo_L}(\text{get_link}(\text{li})(\delta)),$

244(b)ii `(atH(hi,fli,tli),atH(hi',fli',tli'))`

244(b)ii $\rightarrow \text{hi}=\text{hi}' \wedge \text{fli}=\text{fli}' \wedge \text{tli}=\text{tli}' \wedge \text{hi} \in \text{xtr_his}(\delta) \wedge \{\text{fli,tli}\} = \text{obs_mereo_H}(\text{get_hub}(\text{hi})(\delta)),$

```

244(b)iii      (onL(li,fhi,thi,1),atH(hi,fli,tli))
244(b)iii      → li=fli∧thi=hi ∧ {li,tli} ⊆ xtr_lis(δ) ∧ {fhi,thi}=obs_mereo_L(get_link(li)(δ))
244(b)iii      ∧ hi ∈ xtr_his(δ) ∧ (fli,tli) ∈ obs_mereo_H(get_hub(hi)(δ)),
244(b)iv       (atH(hi,fli,tli),onL(li',fhi',thi',0))
244(b)iv       → etcetera,
244b          — → false
244b          end end end end end

```

5.2.3 Perdurants

Our presentation of example perdurants is not as systematic as that of example endurants. Give the simple basis of endurants covered above there is now a huge variety of perdurants, so we just select one example from each of the three classes of perdurants (as outline in [49]): a simple hub insertion *action* (Sect. 5.2.3), a simple link disappearance *event* (Sect. 5.2.3) and a not quite so simple *behaviour*, that of road traffic (Sect. 5.2.3).

Hub Insertion Action

<p>245 Initially inserted hubs, h, are characterised</p> <p style="padding-left: 20px;">a by their unique identifier which not one of any hub in the net, n, into which the hub is being inserted,</p> <p style="padding-left: 20px;">b by a mereology, $\{\}$, of zero link identifiers, and</p> <p style="padding-left: 20px;">c by — whatever — attributes, $attrs$, are needed.</p> <p>246 The result of such a hub insertion is a net, n',</p> <p style="padding-left: 20px;">a whose links are those of n, and</p> <p style="padding-left: 20px;">b whose hubs are those of n augmented with h.</p>	<p>value</p> <p>245 insert_hub: $H \rightarrow N \rightarrow N$</p> <p>246 insert_hub($h$)($n$) as n'</p> <p>245a pre: $uid_H(h) \notin xtr_his(n)$</p> <p>245b \wedge $obs_mereo_H = \{\}$</p> <p>245c \wedge ...</p> <p>246a post: $obs_part_Ls(n) = obs_part_Ls(n')$</p> <p>246b $\wedge obs_part_Hs(n) \cup \{h\} = obs_part_Hs(n')$</p>
---	---

Link Disappearance Event

We formalise aspects of the link disappearance event:

<p>247 The result net, $n':N'$, is not well-formed.</p> <p>248 For a link to disappear there must be at least one link in the net;</p> <p>249 and such a link may disappear such that</p> <p>250 it together with the resulting net makes up for the “original” net.</p>	<p>value</p> <p>247 link_diss_event: $N \times N' \times \mathbf{Bool}$</p> <p>247 link_diss_event(n,n') as tf</p> <p>248 pre: $obs_part_Ls(obs_part_LS(n)) \neq \{\}$</p> <p>249 post: $\exists l:L.l \in obs_part_Ls(obs_part_LS(n)) \Rightarrow$</p> <p>250 $l \notin obs_part_Ls(obs_part_LS(n'))$</p> <p>250 $\wedge n' \cup \{l\} = obs_part_Ls(obs_part_LS(n))$</p>
---	--

Road Traffic

The analysis & description of the road traffic behaviour is composed (i) from the description of the global values of nets, links and hubs, vehicles, monitor, a clock, and an initial distribution, *map*, of vehicles, “across” the net; (ii) from the description of channels between vehicles and the monitor; (iii) from the description of behaviour signatures, that is, those of the overall road traffic system, the vehicles, and the monitor; and (iv) from the description of the individual behaviours, that is, the overall road traffic system, *rts*, the individual vehicles, *veh*, and the monitor, *mon*.

Global Values:

There is given some globally observable parts.

251 besides the domain, $\delta:\Delta$,
 252 a net, $n:N$,
 253 a set of vehicles, $vs:V\text{-set}$,
 254 a monitor, $m:M$, and

255 a clock, clock, behaviour.

256 From the net and vehicles we generate an initial distribution of positions of vehicles.

The $n:N$, $vs:V\text{-set}$ and $m:M$ are observable from any road traffic system domain δ .

value

251 $\delta:\Delta$
 252 $n:N = \text{obs_part_N}(\delta)$,
 252 $ls:L\text{-set} = \text{links}(\delta)$, $hs:H\text{-set} = \text{hubs}(\delta)$,
 252 $lis:LI\text{-set} = \text{xtr_lis}(\delta)$, $his:HI\text{-set} = \text{xtr_his}(\delta)$
 253 $va:VS = \text{obs_part_VS}(\text{obs_part_F}(\delta))$,
 253 $vs:Vs\text{-set} = \text{obs_part_Vs}(va)$,

253 $vis:VI\text{-set} = \{\text{uid_VI}(v) \mid v:V \bullet v \in vs\}$,
 254 $m:\text{obs_part_M}(\delta)$,
 254 $mi = \text{uid_MI}(m)$,
 254 $ma:\text{attributes}(m)$
 255 $\text{clock}: \mathbb{T} \rightarrow \text{out } \{\text{clk_ch}[vi \mid vi:VI \bullet vi \in vis]\} \text{ Unit}$
 256 $vm:\text{MAP} \bullet \text{vpos_map} = \text{distribute}(vs)(n)$;

Channels:

257 We additionally declare a set of vehicle-to-monitor-channels indexed

a by the unique identifiers of vehicles
 b and the (single) monitor identifier.⁸

and communicating vehicle positions.

channel

257 $\{v_m_ch[vi,mi] \mid vi:VI \bullet vi \in vis\}:VPos$

Behaviour Signatures:

258 The road traffic system behaviour, rts , takes no arguments (hence the first **Unit**)⁹; and “behaves”, that is, continues forever (hence the last **Unit**).

259 The vehicle behaviour

a is indexed by the unique identifier, $\text{uid_V}(v):VI$,
 b the vehicle mereology, in this case the single monitor identifier $mi:MI$,
 c the vehicle attributes, $\text{obs_attribs}(v)$
 d and — factoring out one of the vehicle attributes — the current vehicle position.

e The vehicle behaviour offers communication to the monitor behaviour (on channel $vm_ch[vi]$); and behaves “forever”.

260 The monitor behaviour takes

a the monitor identifier,
 b the monitor mereology,
 c the monitor attributes,
 d and — factoring out one of the vehicle attributes — the discrete road traffic, $\text{drtf}:dRTF$, being repeatedly “updated” as the result of **input** communications from (all) vehicles;
 e the behaviour otherwise behaves forever.

value

258 $rts: \text{Unit} \rightarrow \text{Unit}$
 259 $\text{veh}_{vi:VI}: mi:MI \rightarrow vp:VPos \rightarrow \text{out } vm_ch[vi,mi] \text{ Unit}$
 260 $\text{mon}_{mi:MI}: vis:VI\text{-set} \rightarrow RTF \rightarrow \text{in } \{v_m_ch[vi,mi] \mid vi:VI \bullet vi \in vis\}, \text{clk_ch} \text{ Unit}$

The Road Traffic System Behaviour:

261 Thus we shall consider our **road traffic system**, rts , as

a the concurrent behaviour of a number of vehicles and, to “observe”, or, as we shall call it, to monitor their movements,
 b the monitor behaviour.

value

261 $rts() =$
 261a $\parallel \{\text{veh}_{\text{uid_VI}(v)}(mi)(vm(\text{uid_VI}(v))) \mid v:V \bullet v \in vs\}$
 261b $\parallel \text{mon}_{mi}(vis)([vi \mapsto \langle \rangle \mid vi:VI \bullet vi \in vis])$

⁸ Technically speaking: we could omit the monitor identifier.

⁹ The **Unit** designator is an RSL technicality.

where, wrt, the monitor, we dispense with the mereology and the attribute state arguments and instead just have a monitor traffic argument which records the discrete road traffic, MAP, initially set to “empty” traces ($\langle \rangle$, of so far “no road traffic”).

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

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

- a Either the vehicle remains at that hub informing the monitor of its position,
- b or, internally non-deterministically,
 - i moves onto a link, tli, whose “next” hub, identified by thi, is obtained from the mereology of the link identified by tli;
 - ii informs the monitor, on channel vm[vi,mi], that it is now at the very beginning (0) of the link identified by tli, whereupon the vehicle resumes the vehicle behaviour positioned at the very beginning of that link,

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

```

262 vehvi(mi)(vp:atH(hi,fli,tli)) ≡
262a   v_m_ch[vi,mi]!vp ; vehvi(mi)(vp)
262b   □
262(b)i   let {hi',thi}=obs_mereo_L(get_link(tli)(n)) in
262(b)i   assert: hi'=hi
262(b)ii  v_m_ch[vi,mi]!onL(tli,hi,thi,0) ;
262(b)ii  vehvi(mi)(onL(tli,hi,thi,0)) end
262c   □ stop

```

263 We describe here an abstraction of the vehicle behaviour **on** a Link (li). Either

- a the vehicle remains at that link position informing the monitor of its position,
- b or, internally non-deterministically, if the vehicle’s position on the link has not yet reached the hub,
 - i then the vehicle moves an arbitrary increment ℓ_ε (less than or equal to the distance to the hub) along the link informing the monitor of this, or
 - ii else,
 - 1 while obtaining a “next link” from the mereology of the hub (where that next link could very well be the same as the link the vehicle is about to leave),
 - 2 the vehicle informs the monitor that it is now at the hub identified by thi,

whereupon the vehicle resumes the vehicle behaviour positioned at that hub.

c or, internally non-deterministically, the vehicle “disappears — off the radar” !

```

263 vehvi(mi)(vp:onL(li,fhi,thi,r)) ≡
263a   v_m_ch[vi,mi]!vp ; vehvi(mi,va)(vp)
263b   □ if r + ℓε ≤ 1
263(b)i   then
263(b)i   v_m_ch[vi,mi]!onL(li,fhi,thi,r+ℓε) ;
263(b)i   vehvi(mi)(onL(li,fhi,thi,r+ℓε))
263(b)ii  else
263(b)ii1  let li':L|li' ∈ obs_mereo_H(get_hub(thi)(n)) in
263(b)ii2  v_m_ch[vi,mi]!atH(li,thi,li') ;
263(b)ii2  vehvi(mi)(atH(li,thi,li')) end end
263c   □ stop

```

The Monitor Behaviour

264 The monitor behaviour evolves around

- a the monitor identifier,
- b the monitor mereology,
- c and the attributes, ma:ATTR
- d — where we have factored out as a separate arguments — a table of traces of time-stamped vehicle positions,
- e while accepting messages
 - i about time
 - ii and about vehicle positions
- f and otherwise progressing “in[de]finitely”.

265 Either the monitor “does own work”

266 or, internally non-deterministically accepts messages from vehicles.

a A vehicle position message, vp, may arrive from the vehicle identified by vi.

b That message is appended to that vehicle’s movement trace – prefixed by time (obtained from the time channel),

c whereupon the monitor resumes its behaviour —

d where the communicating vehicles range over all identified vehicles.

```

264 monmi(vis)(trf) ≡
265   monmi(vis)(trf)
266   □
266a   □ {let tvp = (clk_ch?,v_m_ch[vi,mi]?) in

```

```

266b   let trf' = trf † [vi ↦ trf(vi)ˆ<tvp>] in
266c   monmi(vis)(trf')
266d   end end | vi:Vl • vi ∈ vis}

```

We are about to complete a long, i.e., a 6.3 page example (!). We can now comment on the full example: The domain, $\delta : \Delta$ is a manifest part. The road net, $n : N$ is also a manifest part. The fleet, $f : F$, of vehicles, $vs : VS$, likewise, is a manifest part. But the monitor, $m : M$, is a concept. One does not have to think of it as a manifest “observer”. The vehicles are on — or off — the road (i.e., links and hubs). We know that from a few observations and generalise to all vehicles. They either move or stand still. We also, similarly, know that. Vehicles move. Yes, we know that. Based on all these repeated observations and generalisations we introduce the concept of vehicle traffic. Unless positioned high above a road net — and with good binoculars — a single person cannot really observe the traffic. There are simply too many links, hubs, vehicles, vehicle positions and times. Thus we conclude that, even in a richly manifest domain, we can also “speak of”, that is, describe concepts over manifest phenomena, including time !

5.2.4 Domain Facets

The example of this section, i.e., Sect. 5.2, focuses on the domain facet [33, 2008] of (i) intrinsics. It does not reflect the other domain facets: (ii) domain support technologies, (iii) domain rules, regulations & scripts, (iv) organisation & management, and (v) human behaviour. The requirements examples, i.e., the rest of this paper, thus builds only on the domain intrinsics. This means that we shall not be able to cover principles, technique and tools for the prescription of such important requirements that handle failures of support technology or humans. We shall, however point out where we think such, for example, fault tolerance requirements prescriptions “fit in” and refer to relevant publications for their handling.

5.3 Requirements

This and the next three sections, Sects. 5.4.–5.5., are the main sections of this paper. Section 5.4. is the most detailed and systematic section. It covers the *domain requirements* operations of projection, instantiation, determination, extension and, less detailed, fitting. Section 5.5. surveys the *interface requirements* issues of *shared phenomena*: shared endurants, shared actions, shared events and shared behaviour, and “completes” the exemplification of the detailed *domain extension* of our requirements into a *road pricing system*. Section 5.5. also covers the notion of derived requirements.

5.3.1 The Three Phases of Requirements Engineering

There are, as we see it, three kinds of design assumptions and requirements: (i) domain requirements, (ii) interface requirements and (iii) machine requirements. (i) **Domain requirements** are those requirements which can be expressed solely using terms of the domain ⊙ (ii) **Interface requirements** are those requirements which can be expressed only using technical terms of both the domain and the machine ⊙ (iii) **Machine requirements** are those requirements which, in principle, can be expressed solely using terms of the machine ⊙

Definition 33 Verification Paradigm: Some preliminary designations: let \mathcal{D} designate the the domain description; let \mathcal{R} designate the requirements prescription, and let \mathcal{S} designate the system design. Now $\mathcal{D}, \mathcal{S} \models \mathcal{R}$ shall be read: it must be verified that the \mathcal{S} ystem design satisfies the \mathcal{R} equirements prescription in the context of the \mathcal{D} omain description ⊙

The “in the context of \mathcal{D} ...” term means that proofs of \mathcal{S} oftware design correctness with respect to \mathcal{R} equirements will often have to refer to \mathcal{D} omain requirements assumptions. We refer to [100, Gunter, Jackson and Zave, 2000] for an analysis of a varieties of forms in which \models relate to variants of \mathcal{D} , \mathcal{R} and \mathcal{S} .

5.3.2 Order of Presentation of Requirements Prescriptions

The domain requirements development stage — as we shall see — can be sub-staged into: projection, instantiation, determination, extension and fitting. The interface requirements development stage —

can be sub-staged into *shared: endurant, action, event and behaviour* developments, where “sharedness” pertains to phenomena shared between, i.e., “present” in, both the domain (concretely, manifestly) and the machine (abstractly, conceptually). These development stages need not be pursued in the order of the three stages and their sub-stages. We emphasize that one thing is the stages and steps of development, as for example these: projection, instantiation, determination, extension, fitting, shared endurants, shared actions, shared events, shared behaviours, etcetera, another thing is the requirements prescription that results from these development stages and steps. The further software development, after and on the basis of the requirements prescription starts only when all stages and steps of the requirements prescription have been fully developed. The domain engineer is now free to rearrange the final prescription, irrespective of the order in which the various sections were developed, in such a way as to give a most pleasing, pedagogic and cohesive reading (i.e., presentation). From such a requirements prescription one can therefore not necessarily see in which order the various sections of the prescription were developed.

5.3.3 Design Requirements and Design Assumptions

A crucial distinction is between design requirements and design assumptions. The **design requirements** are those requirements for which the system designer **has to** implement hardware or software in order to satisfy system user expectations ☉ The **design assumptions** are those requirements for which the system designer **does not** have to implement hardware or software, but whose properties the designed hardware, respectively software relies on for proper functioning ☉

Example 5.1. . Road Pricing System — Design Requirements: The design requirements for the road pricing calculator of this paper are for the design (ii) of that part of the vehicle software which interfaces the GNSS receiver and the road pricing calculator (cf. Items 345–348), (iii) of that part of the toll-gate software which interfaces the toll-gate and the road pricing calculator (cf. Items 353–355) and (i) of the road pricing calculator (cf. Items 384–397) ☐

Example 5.2. . Road Pricing System — Design Assumptions: The design assumptions for the road pricing calculator include: (i) that *vehicles* behave as prescribed in Items 344–348, (ii) that the GNSS regularly offers vehicles correct information as to their global position (cf. Item 345), (iii) that *toll-gates* behave as prescribed in Items 350–355, and (iv) that the *road net* is formed and well-formed as defined in Examples 5.7–5.9 ☐

Example 5.3. . Toll-Gate System — Design Requirements: The design requirements for the toll-gate system of this paper are for the design of software for the toll-gate and its interfaces to the road pricing system, i.e., Items 349–350 ☉

Example 5.4. . Toll-Gate System — Design Assumptions: The design assumptions for the toll-gate system include (i) that the vehicles behave as per Items 344–348, and (ii) that the road pricing calculator behave as per Items 384–397 ☉

5.3.4 Derived Requirements

In building up the domain, interface and machine requirements a number of machine concepts are introduced. These machine concepts enable the expression of additional requirements. It is these we refer to as derived requirements. Techniques and tools espoused in such classical publications as [79, 116, 187, 125, 177] can in those cases be used to advantage.

5.4 Domain Requirements

Domain requirements primarily express the assumptions that a design must rely upon in order that that design can be verified. Although domain requirements firstly express assumptions it appears that the software designer is well-advised in also implementing, as data structures and procedures, the endurants, respectively perdurants expressed in the domain requirements prescriptions. Whereas domain endurants are “real-life” phenomena they are now, in domain requirements prescriptions, abstract concepts (to be represented by a machine).

Definition 34 Domain Requirements Prescription: A **domain requirements prescription** is that subset of the requirements prescription whose technical terms are defined in a domain description \odot

To determine a relevant subset all we need is collaboration with requirements, cum domain stake-holders. Experimental evidence, in the form of example developments of requirements prescriptions from domain descriptions, appears to show that one can formulate techniques for such developments around a few domain-description-to-requirements-prescription operations. We suggest these: projection, instantiation, determination, extension and fitting. In Sect. 5.3.2 we mentioned that the order in which one performs these domain-description-to-domain-requirements-prescription operations is not necessarily the order in which we have listed them here, but, with notable exceptions, one is well-served in starting out requirements development by following this order.

5.4.1 Domain Projection

Definition 35 Domain Projection: By a **domain projection** we mean a subset of the domain description, one which projects out all those *endurants*: parts, materials and components, as well as *perdurants*: actions, events and behaviours that the stake-holders do not wish represented or relied upon by the machine \odot

The resulting document is a partial domain requirements prescription. In determining an appropriate subset the requirements engineer must secure that the final “projection prescription” is complete and consistent — that is, that there are no “dangling references”, i.e., that all entities and their internal properties that are referred to are all properly defined.

Domain Projection — Narrative

We now start on a series of examples that illustrate domain requirements development.

Example 5.5. . Domain Requirements. Projection: A Narrative Sketch: We require that the road pricing system shall [at most] relate to the following domain entities – and only to these¹⁰: the net, its links and hubs, and their properties (unique identifiers, mereologies and some attributes), the vehicles, as endurants, and the general vehicle behaviours, as perdurants. We treat projection together with a concept of simplification. The example simplifications are vehicle positions and, related to the simpler vehicle position, vehicle behaviours. To prescribe and formalise this we copy the domain description. From that domain description we remove all mention of the hub insertion action, the link disappearance event, and the monitor \square

As a result we obtain $\Delta_{\mathcal{P}}$, the projected version of the domain requirements prescription¹¹.

Domain Projection — Formalisation

The requirements prescription hinges, crucially, not only on a systematic narrative of all the projected, instantiated, determined, extended and fitted specifications, but also on their formalisation. In the formal domain projection example we, regrettably, omit the narrative texts. In bringing the formal texts we keep the item numbering from Sect. 5.2, where you can find the associated narrative texts.

Example 5.6. . Domain Requirements — Projection: Main Sorts

type	
217	$\Delta_{\mathcal{P}}$
217a	$N_{\mathcal{P}}$
217b	$F_{\mathcal{P}}$
value	

¹⁰ By ‘relate to ... these’ we mean that the required system does not rely on domain phenomena that have been “projected away”.

¹¹ Restrictions of the net to the toll road nets, hinted at earlier, will follow in the next domain requirements steps.

217a **obs_part_N** $_{\mathcal{D}}$: $\Delta_{\mathcal{D}} \rightarrow N_{\mathcal{D}}$
 217b **obs_part_F** $_{\mathcal{D}}$: $\Delta_{\mathcal{D}} \rightarrow F_{\mathcal{D}}$
type
 218a $HA_{\mathcal{D}}$
 218b $LA_{\mathcal{D}}$
value
 218a **obs_part_HA**: $N_{\mathcal{D}} \rightarrow HA$
 218b **obs_part_LA**: $N_{\mathcal{D}} \rightarrow LA$

Concrete Types

type
 219 $H_{\mathcal{D}}, HS_{\mathcal{D}} = H_{\mathcal{D}}\text{-set}$
 220 $L_{\mathcal{D}}, LS_{\mathcal{D}} = L_{\mathcal{D}}\text{-set}$
 221 $V_{\mathcal{D}}, VS_{\mathcal{D}} = V_{\mathcal{D}}\text{-set}$
value
 219 **obs_part_HS** $_{\mathcal{D}}$: $HA_{\mathcal{D}} \rightarrow HS_{\mathcal{D}}$
 220 **obs_part_LS** $_{\mathcal{D}}$: $LA_{\mathcal{D}} \rightarrow LS_{\mathcal{D}}$
 221 **obs_part_VS** $_{\mathcal{D}}$: $F_{\mathcal{D}} \rightarrow VS_{\mathcal{D}}$
 222a **links**: $\Delta_{\mathcal{D}} \rightarrow L\text{-set}$
 222a **links**($\delta_{\mathcal{D}}$) \equiv **obs_part_LS** $_{\mathcal{D}}$ (**obs_part_LA** $_{\mathcal{D}}$ ($\delta_{\mathcal{D}}$))
 222b **hubs**: $\Delta_{\mathcal{D}} \rightarrow H\text{-set}$
 222b **hubs**($\delta_{\mathcal{D}}$) \equiv **obs_part_HS** $_{\mathcal{D}}$ (**obs_part_HA** $_{\mathcal{D}}$ ($\delta_{\mathcal{D}}$))

Unique Identifiers

type
 223a HI, LI, VI, MI
value
 223c **uid_HI**: $H_{\mathcal{D}} \rightarrow HI$
 223c **uid_LI**: $L_{\mathcal{D}} \rightarrow LI$
 223c **uid_VI**: $V_{\mathcal{D}} \rightarrow VI$
 223c **uid_MI**: $M_{\mathcal{D}} \rightarrow MI$
axiom
 223b $HI \cap LI = \emptyset, HI \cap VI = \emptyset, HI \cap MI = \emptyset,$
 223b $LI \cap VI = \emptyset, LI \cap MI = \emptyset, VI \cap MI = \emptyset$

Mereology

value
 228 **obs_mereo_H** $_{\mathcal{D}}$: $H_{\mathcal{D}} \rightarrow LI\text{-set}$
 229 **obs_mereo_L** $_{\mathcal{D}}$: $L_{\mathcal{D}} \rightarrow HI\text{-set}$
 229 **axiom** $\forall l:L_{\mathcal{D}} \bullet \text{card obs_mereo_L}_{\mathcal{D}}(l)=2$
 230 **obs_mereo_V** $_{\mathcal{D}}$: $V_{\mathcal{D}} \rightarrow MI$
 231 **obs_mereo_M** $_{\mathcal{D}}$: $M_{\mathcal{D}} \rightarrow VI\text{-set}$
axiom
 232 $\forall \delta_{\mathcal{D}}:\Delta_{\mathcal{D}}, hs:HS \bullet hs = \text{hubs}(\delta), ls:LS \bullet ls = \text{links}(\delta_{\mathcal{D}}) \Rightarrow$
 232 $\forall h:H_{\mathcal{D}} \bullet h \in hs \Rightarrow$
 232 **obs_mereo_H** $_{\mathcal{D}}$ (h) $\subseteq \text{xtr_his}(\delta_{\mathcal{D}}) \wedge$
 233 $\forall l:L_{\mathcal{D}} \bullet l \in ls \bullet$
 232 **obs_mereo_L** $_{\mathcal{D}}$ (l) $\subseteq \text{xtr_lis}(\delta_{\mathcal{D}}) \wedge$
 234a **let** $f:F_{\mathcal{D}} \bullet f = \text{obs_part_F}_{\mathcal{D}}(\delta_{\mathcal{D}}) \Rightarrow$
 234a $vs:VS_{\mathcal{D}} \bullet vs = \text{obs_part_VS}_{\mathcal{D}}(f)$ **in**
 234a $\forall v:V_{\mathcal{D}} \bullet v \in vs \Rightarrow$
 234a **uid_V** $_{\mathcal{D}}$ (v) $\in \text{obs_mereo_M}_{\mathcal{D}}(m) \wedge$
 234b **obs_mereo_M** $_{\mathcal{D}}$ (m)
 234b $= \{\text{uid_V}_{\mathcal{D}}(v) \mid v:V \bullet v \in vs\}$
 234b **end**

Attributes: We project attributes of hubs, links and vehicles.

First **hubs**:

```

type
235a GeoH
235b  $H\Sigma_{\mathcal{D}} = (LI \times LI)\text{-set}$ 
235c  $H\Omega_{\mathcal{D}} = H\Sigma_{\mathcal{D}}\text{-set}$ 
value
235b  $\text{attr\_H}\Sigma_{\mathcal{D}}: H_{\mathcal{D}} \rightarrow H\Sigma_{\mathcal{D}}$ 
235c  $\text{attr\_H}\Omega_{\mathcal{D}}: H_{\mathcal{D}} \rightarrow H\Omega_{\mathcal{D}}$ 
axiom
236  $\forall \delta_{\mathcal{D}}: \Delta_{\mathcal{D}},$ 
236   let  $hs = \text{hubs}(\delta_{\mathcal{D}})$  in
236    $\forall h: H_{\mathcal{D}} \bullet h \in hs \bullet$ 
236a      $\text{xtr\_lis}(h) \subseteq \text{xtr\_lis}(\delta_{\mathcal{D}})$ 
236b      $\wedge \text{attr\_}\Sigma_{\mathcal{D}}(h) \in \text{attr\_}\Omega_{\mathcal{D}}(h)$ 
236   end

```

Then **links**:

```

type
239 GeoL
240a  $L\Sigma_{\mathcal{D}} = (HI \times HI)\text{-set}$ 
240b  $L\Omega_{\mathcal{D}} = L\Sigma_{\mathcal{D}}\text{-set}$ 
value
239  $\text{attr\_GeoL}: L \rightarrow \text{GeoL}$ 
240a  $\text{attr\_L}\Sigma_{\mathcal{D}}: L_{\mathcal{D}} \rightarrow L\Sigma_{\mathcal{D}}$ 
240b  $\text{attr\_L}\Omega_{\mathcal{D}}: L_{\mathcal{D}} \rightarrow L\Omega_{\mathcal{D}}$ 
axiom

```

240a– 240b on Page 141.

Finally **vehicles**: For ‘road pricing’ we need vehicle positions. But, for “technical reasons”, we must abstain from the detailed description given in Items 241–241c¹² We therefore simplify vehicle positions.

267 A simplified vehicle position designates
 a either a link
 b or a hub,

```

type
267 SVPos = SonL | SatH
267a SonL :: LI
267b SatH :: HI
axiom
241a'  $\forall n: N, \text{SonL}(li): \text{SVPos} \bullet$ 
241a'    $\exists l: L \bullet l \in \text{obs\_part\_LS}(\text{obs\_part\_N}(n)) \Rightarrow li = \text{uid\_L}(l)$ 
241c'  $\forall n: N, \text{SatH}(hi): \text{SVPos} \bullet$ 
241c'    $\exists h: H \bullet h \in \text{obs\_part\_HS}(\text{obs\_part\_N}(n)) \Rightarrow hi = \text{uid\_H}(h)$ 

```

Global Values

```

value
251  $\delta_{\mathcal{D}}: \Delta_{\mathcal{D}},$ 
252  $n: N_{\mathcal{D}} = \text{obs\_part\_N}_{\mathcal{D}}(\delta_{\mathcal{D}}),$ 
252  $ls: L_{\mathcal{D}}\text{-set} = \text{links}(\delta_{\mathcal{D}}),$ 
252  $hs: H_{\mathcal{D}}\text{-set} = \text{hubs}(\delta_{\mathcal{D}}),$ 
252  $lis: LI\text{-set} = \text{xtr\_lis}(\delta_{\mathcal{D}}),$ 
252  $his: HI\text{-set} = \text{xtr\_his}(\delta_{\mathcal{D}})$ 

```

Behaviour Signatures: We omit the monitor behaviour.

268 We leave the vehicle behaviours’ attribute argument undefined.

¹² The ‘technical reasons’ are that we assume that the GNSS cannot provide us with direction of vehicle movement and therefore we cannot, using only the GNSS provide the details of ‘offset’ along a link (*onL*) nor the “from/to link” at a hub (*atH*).

```

type
268  ATTR
value
258  trs℘: Unit → Unit
259  veh℘: VI×MI×ATTR → ... Unit

```

The System Behaviour: We omit the monitor behaviour.

```

value
261a  trs℘() = || { veh℘(uid_VI(v), obs_mereo_V(v), _) | v:V℘•v ∈ vs }

```

The Vehicle Behaviour: Given the simplification of vehicle positions we *simplify* the vehicle behaviour given in Items 262–263

```

262'  vehvi(mi)(vp:SatH(hi)) ≡
262a'      v_m_ch[vi,mi]!SatH(hi) ; vehvi(mi)(SatH(hi))
262(b)i'    [] let li:L!li ∈ obs_mereo_H(get_hub(hi)(n)) in
262(b)ii'   v_m_ch[vi,mi]!SonL(li) ; vehvi(mi)(SonL(li)) end
262c'      [] stop

263'  vehvi(mi)(vp:SonL(li)) ≡
263a'      v_m_ch[vi,mi]!SonL(li) ; vehvi(mi)(SonL(li))
263(b)ii1'  [] let hi:H!hi ∈ obs_mereo_L(get_link(li)(n)) in
263(b)ii2'  v_m_ch[vi,mi]!SatH(hi) ; vehvi(mi)(atH(hi)) end
263c'      [] stop

```

We can simplify Items 262'–263c' further.

```

269  vehvi(mi)(vp) ≡
270      v_m_ch[vi,mi]!vp ; vehvi(mi)(vp)
271      [] case vp of
271          SatH(hi) →
272              let li:L!li ∈ obs_mereo_H(get_hub(hi)(n)) in
273                  v_m_ch[vi,mi]!SonL(li) ; vehvi(mi)(SonL(li)) end,
271          SonL(li) →
274              let hi:H!hi ∈ obs_mereo_L(get_link(li)(n)) in
275                  v_m_ch[vi,mi]!SatH(hi) ; vehvi(mi)(atH(hi)) end end
276      [] stop

```

269 This line coalesces Items 262' and 263'.
 270 Coalescing Items 262a' and 263'.
 271 Captures the distinct parameters of Items 262' and 263'.
 272 Item 262(b)i'.
 273 Item 262(b)ii'.
 274 Item 263(b)ii1'.
 275 Item 263(b)ii2'.
 276 Coalescing Items 262c' and 263c'.

The above vehicle behaviour definition will be transformed (i.e., further “refined”) in Sect. 5.5.1’s Example 5.15; cf. Items 344– 348 on Page 163 □

Discussion

Domain projection can also be achieved by developing a “completely new” domain description — typically on the basis of one or more existing domain description(s) — where that “new” description now takes the rôle of being the project domain requirements.

5.4.2 Domain Instantiation

Definition 36 Domain Instantiation: By **domain instantiation** we mean a **refinement** of the partial domain requirements prescription (resulting from the projection step) in which the refinements aim at rendering the *endurants*: parts, materials and components, as well as the *perdurants*: actions, events and behaviours of the domain requirements prescription more concrete, more specific \odot Instantiations usually render these concepts less general.

Properties that hold of the projected domain shall also hold of the (therefrom) instantiated domain.

Refinement of endurants can be expressed (i) either in the form of concrete types, (ii) or of further “delineating” axioms over sorts, (iii) or of a combination of concretisation and axioms. We shall exemplify the third possibility. Example 5.7 express requirements that the road net (on which the road-pricing system is to be based) must satisfy. Refinement of perdurants will not be illustrated (other than the simplification of the *vehicle* projected behaviour).

Domain Instantiation

Example 5.7. . Domain Requirements. Instantiation Road Net: We now require that there is, as before, a road net, $n_{\mathcal{J}}:N_{\mathcal{J}}$, which can be understood as consisting of two, “connected sub-nets”. A toll-road net, $trn_{\mathcal{J}}:TRN_{\mathcal{J}}$, cf. Fig. 5.1, and an ordinary road net, n_o . The two are connected as follows: The toll-road net, $trn_{\mathcal{J}}$, borders some toll-road plazas, in Fig. 5.1 shown by white filled circles (i.e., hubs). These toll-road plaza hubs are proper hubs of the ‘ordinary’ road net, n_o .

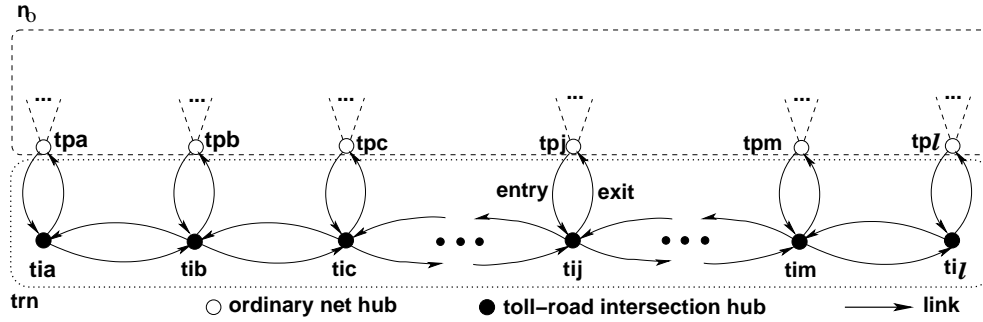


Fig. 5.1. A simple, linear toll-road net trn . tp_j : toll plaza j , ti_j : toll road intersection j .

Upper dashed sub-figure hint at an ordinary road net n_o .

Lower dotted sub-figure hint at a toll-road net trn .

Dash-dotted (---) “V”-images above tp_j s hint at links to remaining “parts” of n_o .

277 The instantiated domain, $\delta_{\mathcal{J}}:\Delta_{\mathcal{J}}$ has just the net, $n_{\mathcal{J}}:N_{\mathcal{J}}$ being instantiated.

278 The road net consists of two “sub-nets”

a an “ordinary” road net, $n_o:N_{\mathcal{J}}$ and

b a toll-road net proper, $trn:TRN_{\mathcal{J}}$ —

c “connected” by an interface $hil:HIL$:

i That interface consists of a number of toll-road plazas (i.e., hubs), modeled as a list of hub identifiers, $hil:HI^*$.

ii The toll-road plaza interface to the toll-road net, $trn:TRN_{\mathcal{J}}$ ¹³, has each plaza, $hil[i]$, connected to a pair of toll-road links: an entry and an exit link: $(l_e:L, l_x:L)$.

iii The toll-road plaza interface to the ‘ordinary’ net, $n_o:N_{\mathcal{J}}$, has each plaza, i.e., the hub designated by the hub identifier $hil[i]$, connected to one or more ordinary net links, $\{l_{i_1}, l_{i_2}, \dots, l_{i_k}\}$.

278b The toll-road net, $trn:TRN_{\mathcal{J}}$, consists of three collections (modeled as lists) of links and hubs:

¹³ We (sometimes) omit the subscript \mathcal{J} when it should be clear from the context what we mean.

- i a list of pairs of toll-road entry/exit links: $\langle (l_{e_1}, l_{x_1}), \dots, (l_{e_\ell}, l_{x_\ell}) \rangle$,
 - ii a list of toll-road intersection hubs: $\langle h_{i_1}, h_{i_2}, \dots, h_{i_\ell} \rangle$, and
 - iii a list of pairs of main toll-road (“up” and “down”) links: $\langle (m_{l_{i_{1u}}}, m_{l_{i_{1d}}}), (m_{l_{i_{2u}}}, m_{l_{i_{2d}}}), \dots, (m_{l_{i_{\ell u}}}, m_{l_{i_{\ell d}}}) \rangle$.
- d The three lists have commensurate lengths (ℓ).

ℓ is the number of toll plazas, hence also the number of toll-road intersection hubs and therefore a number one larger than the number of pairs of main toll-road (“up” and “down”) links

type

```

277  $\Delta_{\mathcal{J}}$ 
278  $N_{\mathcal{J}} = N_{\mathcal{J}'} \times \text{HIL} \times \text{TRN}$ 
278a  $N_{\mathcal{J}'}$ 
278b  $\text{TRN}_{\mathcal{J}} = (L \times L)^* \times H^* \times (L \times L)^*$ 
278c  $\text{HIL} = H^*$ 

```

axiom

```

278d  $\forall n_{\mathcal{J}} : N_{\mathcal{J}} \bullet$ 
278d   let  $(n_{\Delta}, \text{hil}, (\text{exll}, \text{hl}, \text{lll})) = n_{\mathcal{J}}$  in
278d   len  $\text{hil} = \text{len } \text{exll} = \text{len } \text{hl} = \text{len } \text{lll} + 1$ 
278d   end

```

We have named the “ordinary” net sort (primed) $N_{\mathcal{J}'}$. It is “almost” like (unprimed) $N_{\mathcal{J}}$ — except that the interface hubs are also connected to the toll-road net entry and exit links.

The partial concretisation of the net sorts, $N_{\mathcal{J}'}$ into $N_{\mathcal{J}}$ requires some additional well-formedness conditions to be satisfied.

279 The toll-road intersection hubs all¹⁴ have distinct identifiers.

```

279  $\text{wf\_dist\_toll\_road\_isect\_hub\_ids} : H^* \rightarrow \text{Bool}$ 
279  $\text{wf\_dist\_toll\_road\_isect\_hub\_ids}(\text{hl}) \equiv$ 
279   len  $\text{hl} = \text{card } \text{xtr\_his}(\text{hl})$ 

```

280 The toll-road links all have distinct identifiers.

```

280  $\text{wf\_dist\_toll\_road\_u\_d\_link\_ids} : (L \times L)^* \rightarrow \text{Bool}$ 
280  $\text{wf\_dist\_toll\_road\_u\_d\_link\_ids}(\text{lll}) \equiv$ 
280    $2 \times \text{len } \text{lll} = \text{card } \text{xtr\_lis}(\text{lll})$ 

```

281 The toll-road entry/exit links all have distinct identifiers.

```

281  $\text{wf\_dist\_e\_x\_link\_ids} : (L \times L)^* \rightarrow \text{Bool}$ 
281  $\text{wf\_dist\_e\_x\_link\_ids}(\text{exll}) \equiv$ 
281    $2 \times \text{len } \text{exll} = \text{card } \text{xtr\_lis}(\text{exll})$ 

```

282 Proper net links must not designate toll-road intersection hubs.

```

282  $\text{wf\_isold\_toll\_road\_isect\_hubs} : H^* \times H^* \rightarrow N_{\mathcal{J}} \rightarrow \text{Bool}$ 
282  $\text{wf\_isold\_toll\_road\_isect\_hubs}(\text{hil}, \text{hl})(n_{\mathcal{J}}) \equiv$ 
282   let  $\text{ls} = \text{xtr\_links}(n_{\mathcal{J}})$  in
282   let  $\text{his} = \cup \{ \text{obs\_mereo\_L}(l) \mid l : L \bullet l \in \text{ls} \}$  in
282    $\text{his} \cap \text{xtr\_his}(\text{hl}) = \{ \}$  end end

```

283 The plaza hub identifiers must designate hubs of the ‘ordinary’ net.

```

283  $\text{wf\_p\_hubs\_pt\_of\_ord\_net} : H^* \rightarrow N'_{\Delta} \rightarrow \text{Bool}$ 
283  $\text{wf\_p\_hubs\_pt\_of\_ord\_net}(\text{hil})(n'_{\Delta}) \equiv$ 
283   elems  $\text{hil} \subseteq \text{xtr\_his}(n'_{\Delta})$ 

```

¹⁴ A ‘must’ can be inserted in front of all ‘all’s,

284 The plaza hub mereologies must each,
 a besides identifying at least one hub of the ordinary net,
 b also identify the two entry/exit links with which they are supposed to be connected.

```

284 wf_p_hub_intf:  $N'_\Delta \rightarrow \mathbf{Bool}$ 
284 wf_p_hub_intf( $n_o, hil, (exll, \_, \_)$ )  $\equiv$ 
284    $\forall i: \mathbf{Nat} \cdot i \in \mathbf{inds} \text{ exll} \Rightarrow$ 
284     let  $h = \text{get\_H}(hil(i))(n'_\Delta)$  in
284     let  $lis = \mathbf{obs\_mereo\_H}(h)$  in
284     let  $lis' = lis \setminus \text{xtr\_lis}(n')$  in
284      $lis' = \text{xtr\_lis}(\text{exll}(i))$  end end end

```

285 The mereology of each toll-road intersection hub must identify
 a the entry/exit links
 b and exactly the toll-road ‘up’ and ‘down’ links
 c with which they are supposed to be connected.

```

285 wf_toll_road_isect_hub_iface:  $N_{\mathcal{J}} \rightarrow \mathbf{Bool}$ 
285 wf_toll_road_isect_hub_iface( $\_, \_, (exll, hl, lll)$ )  $\equiv$ 
285    $\forall i: \mathbf{Nat} \cdot i \in \mathbf{inds} \text{ hl} \Rightarrow$ 
285      $\mathbf{obs\_mereo\_H}(hl(i)) =$ 
285a      $\text{xtr\_lis}(\text{exll}(i)) \cup$ 
285     case  $i$  of
285b        $1 \rightarrow \text{xtr\_lis}(lll(1)),$ 
285b       len  $hl \rightarrow \text{xtr\_lis}(lll(\mathbf{len} \text{ hl} - 1))$ 
285b        $\_ \rightarrow \text{xtr\_lis}(lll(i)) \cup \text{xtr\_lis}(lll(i - 1))$ 
285     end

```

286 The mereology of the entry/exit links must identify exactly the
 a interface hubs and the
 b toll-road intersection hubs
 c with which they are supposed to be connected.

```

286 wf_exll:  $(L \times L)^* \times H^* \times H^* \rightarrow \mathbf{Bool}$ 
286 wf_exll( $\text{exll}, hil, hl$ )  $\equiv$ 
286    $\forall i: \mathbf{Nat} \cdot i \in \mathbf{len} \text{ exll}$ 
286     let  $(hi, (el, xl), h) = (hil(i), \text{exll}(i), hl(i))$  in
286      $\mathbf{obs\_mereo\_L}(el) = \mathbf{obs\_mereo\_L}(xl)$ 
286      $= \{hi\} \cup \{\mathbf{uid\_H}(h)\}$  end
286   pre:  $\mathbf{len} \text{ eell} = \mathbf{len} \text{ hil} = \mathbf{len} \text{ hl}$ 

```

287 The mereology of the toll-road ‘up’ and ‘down’ links must
 a identify exactly the toll-road intersection hubs
 b with which they are supposed to be connected.

```

287 wf_u_d_links:  $(L \times L)^* \times H^* \rightarrow \mathbf{Bool}$ 
287 wf_u_d_links( $lll, hl$ )  $\equiv$ 
287    $\forall i: \mathbf{Nat} \cdot i \in \mathbf{inds} \text{ lll} \Rightarrow$ 
287     let  $(ul, dl) = lll(i)$  in
287      $\mathbf{obs\_mereo\_L}(ul) = \mathbf{obs\_mereo\_L}(dl) =$ 
287a      $\mathbf{uid\_H}(hl(i)) \cup \mathbf{uid\_H}(hl(i + 1))$  end
287   pre:  $\mathbf{len} \text{ lll} = \mathbf{len} \text{ hl} + 1$ 

```

We have used some additional auxiliary functions:

```

xtr_his:  $H^* \rightarrow HI\text{-set}$ 
xtr_his(hl)  $\equiv \{\text{uid\_HI}(h) \mid h:H \cdot h \in \text{elems hl}\}$ 
xtr_lis:  $(L \times L) \rightarrow LI\text{-set}$ 
xtr_lis(l', l'')  $\equiv \{\text{uid\_LI}(l')\} \cup \{\text{uid\_LI}(l'')\}$ 
xtr_lis:  $(L \times L)^* \rightarrow LI\text{-set}$ 
xtr_lis(III)  $\equiv$ 
 $\cup \{xtr\_lis(l', l'') \mid (l', l''):(L \times L)^* \cdot (l', l'') \in \text{elems III}\}$ 

```

288 The well-formedness of instantiated nets is now the conjunction of the individual well-formedness predicates above.

```

288 wf_instantiated_net:  $N_{\mathcal{J}} \rightarrow \mathbf{Bool}$ 
288 wf_instantiated_net(n'_Δ, hil, (exll, hl, III))
279   wf_dist_toll_road_isect_hub_ids(hl)
280   ∧ wf_dist_toll_road_u_d_link_ids(III)
281   ∧ wf_dist_e_e_link_ids(exll)
282   ∧ wf_isolated_toll_road_isect_hubs(hil, hl)(n')
283   ∧ wf_p_hubs_pt_of_ord_net(hil)(n')
284   ∧ wf_p_hub_interf(n'_Δ, hil, (exll, __, __))
285   ∧ wf_toll_road_isect_hub_iface(__, __, (exll, hl, III))
286   ∧ wf_exll(exll, hil, hl)
287   ∧ wf_u_d_links(III, hl)

```

Domain Instantiation — Abstraction

Example 5.8. . **Domain Requirements. Instantiation Road Net, Abstraction:** Domain instantiation has refined an abstract definition of net sorts, $n_{\mathcal{D}}:N_{\mathcal{D}}$, into a partially concrete definition of nets, $n_{\mathcal{J}}:N_{\mathcal{J}}$. We need to show the refinement relation:

- $\text{abstraction}(n_{\mathcal{J}}) = n_{\mathcal{D}}$.

value

```

289 abstraction:  $N_{\mathcal{J}} \rightarrow N_{\mathcal{D}}$ 
290 abstraction(n'_Δ, hil, (exll, hl, III))  $\equiv$ 
291   let  $n_{\mathcal{D}}:N_{\mathcal{D}}$  •
291     let  $hs = \text{obs\_part\_HS}_{\mathcal{D}}(\text{obs\_part\_HA}_{\mathcal{D}}(n'_{\mathcal{D}}))$ ,
291      $ls = \text{obs\_part\_LS}_{\mathcal{D}}(\text{obs\_part\_LA}_{\mathcal{D}}(n'_{\mathcal{D}}))$ ,
291      $ths = \text{elems hl}$ ,
291      $eells = \text{xtr\_links}(eell)$ ,  $llls = \text{xtr\_links}(III)$  in
292      $hs \cup ths = \text{obs\_part\_HS}_{\mathcal{D}}(\text{obs\_part\_HA}_{\mathcal{D}}(n_{\mathcal{D}}))$ 
293     ∧  $ls \cup eells \cup llls = \text{obs\_part\_LS}_{\mathcal{D}}(\text{obs\_part\_LA}_{\mathcal{D}}(n_{\mathcal{D}}))$ 
294    $n_{\mathcal{D}}$  end end

```

289 The abstraction function takes a concrete net, $n_{\mathcal{J}}:N_{\mathcal{J}}$, and yields an abstract net, $n_{\mathcal{D}}:N_{\mathcal{D}}$.

290 The abstraction function doubly decomposes its argument into constituent lists and sub-lists.

291 There is postulated an abstract net, $n_{\mathcal{D}}:N_{\mathcal{D}}$, such that

292 the hubs of the concrete net and toll-road equals those of the abstract net, and

293 the links of the concrete net and toll-road equals those of the abstract net.

294 And that abstract net, $n_{\mathcal{D}}:N_{\mathcal{D}}$, is postulated to be an abstraction of the concrete net.

Discussion

Domain descriptions, such as illustrated in [49, *Manifest Domains: Analysis & Description*] and in this paper, model families of concrete, i.e., specifically occurring domains. Domain instantiation, as exemplified in this section (i.e., Sect. 5.4.2), “narrow down” these families. Domain instantiation, such as it is defined, cf. Definition 36 on Page 152, allows the requirements engineer to instantiate to a concrete instance of a very specific domain, that, for example, of the toll-road between *Bolzano Nord* and *Trento Sud* in Italy (i.e., $n=7$)¹⁵.

¹⁵ Here we disregard the fact that this toll-road does not start/end in neither *Bolzano Nord* nor *Trento Sud*.

5.4.3 Domain Determination

Definition 37 Determination: By **domain determination** we mean a refinement of the partial domain requirements prescription, resulting from the instantiation step, in which the refinements aim at rendering the *endurants*: parts, materials and components, as well as the *perdurants*: functions, events and behaviours of the partial domain requirements prescription less non-determinate, more determinate \odot

Determinations usually render these concepts less general. That is, the value space of endurants that are made more determinate is “smaller”, contains fewer values, as compared to the endurants before determination has been “applied”.

Domain Determination: Example

We show an example of ‘domain determination’. It is expressed solely in terms of axioms over the concrete toll-road net type.

Example 5.9. . Domain Requirements. Determination Toll-roads: We focus only on the toll-road net. We single out only two ‘determinations’:

All Toll-road Links are One-way Links

295 *The entry/exit and toll-road links*

- a are always all one way links,
- b as indicated by the arrows of Fig. 5.1 on Page 152,
- c such that each pair allows traffic in opposite directions.

```

295 opposite_traffics:  $(L \times L)^* \times (L \times L)^* \rightarrow \mathbf{Bool}$ 
295 opposite_traffics(exl, ill)  $\equiv$ 
295  $\forall (lt, lf): (L \times L) \bullet (lt, lf) \in \mathbf{elems} \text{ exl} \wedge \text{ill} \Rightarrow$ 
295a let  $(lt\sigma, lf\sigma) = (\mathbf{attr\_L}\Sigma(lt), \mathbf{attr\_L}\Sigma(lf))$  in
295a'.  $\mathbf{attr\_L}\Omega(lt) = \{lt\sigma\} \wedge \mathbf{attr\_L}\Omega(lf) = \{lf\sigma\}$ 
295a''.  $\wedge \mathbf{card} \text{ } lt\sigma = 1 = \mathbf{card} \text{ } lf\sigma$ 
295  $\wedge \mathbf{let} (\{(hi, hi')\}, \{(hi'', hi''')\}) = (lt\sigma, lf\sigma)$  in
295c  $hi = hi''' \wedge hi' = hi''$ 
295 end end
```

Predicates 295a'. and 295a''. express the same property.

All Toll-road Hubs are Free-flow

296 *The hub state spaces* are singleton sets of the toll-road hub states which always allow exactly these (and only these) crossings:

- a from *entry* links back to the paired *exit* links,
- b from *entry* links to emanating *toll-road* links,
- c from incident *toll-road* links to *exit* links, and
- d from incident *toll-road* link to emanating *toll-road* links.

```

296 free_flow_toll_road_hubs:  $(L \times L)^* \times (L \times L)^* \rightarrow \mathbf{Bool}$ 
296 free_flow_toll_road_hubs(exl, ill)  $\equiv$ 
296  $\forall i: \mathbf{Nat} \bullet i \in \mathbf{inds} \text{ } hl \Rightarrow$ 
296  $\mathbf{attr\_H}\Sigma(hl(i)) =$ 
296a  $h\sigma\_ex\_ls(exl(i))$ 
296b  $\cup h\sigma\_et\_ls(exl(i), (i, ill))$ 
296c  $\cup h\sigma\_tx\_ls(exl(i), (i, ill))$ 
296d  $\cup h\sigma\_tt\_ls(i, ill)$ 
```

296a: from *entry* links back to the paired *exit* links:

```

296a  $h\sigma\_ex\_ls: (L \times L) \rightarrow L\Sigma$ 
296a  $h\sigma\_ex\_ls(e, x) \equiv \{(\mathbf{uid\_LI}(e), \mathbf{uid\_LI}(x))\}$ 
```


296b: from entry links to emanating toll-road links:

```

296b  hσetJs: (L×L)×(Nat×(em:L×in:L)*)→LΣ
296b  hσetJs((e,⊔),(i,ll)) ≡
296b      case i of
296b          2      → {(uidLI(e),uidLI(em(ll(1))))},
296b          len ll+1 → {(uidLI(e),uidLI(em(ll(len ll))))},
296b          —      → {(uidLI(e),uidLI(em(ll(i-1))))},
296b                      (uidLI(e),uidLI(em(ll(i))))}
296b      end

```

The *em* and *in* in the toll-road link list (em:L×in:L)^{*} designate selectors for *emanating*, respectively *incident* links.

296c: from incident toll-road links to exit links:

```

296c  hσtxJs: (L×L)×(Nat×(em:L×in:L)*)→LΣ
296c  hσtxJs((⊔,x),(i,ll)) ≡
296c      case i of
296c          2      → {(uidLI(in(ll(1))),uidLI(x))},
296c          len ll+1 → {(uidLI(in(ll(len ll))),uidLI(x))},
296c          —      → {(uidLI(in(ll(i-1))),uidLI(x))},
296c                      (uidLI(in(ll(i))),uidLI(x))}
296c      end

```

296d: from incident toll-road link to emanating toll-road links:

```

296d  hσttJs: Nat×(em:L×in:L)*→LΣ
296d  hσttJs(i,ll) ≡
296d      case i of
296d          2      → {(uidLI(in(ll(1))),uidLI(em(ll(1))))},
296d          len ll+1 → {(uidLI(in(ll(len ll))),uidLI(em(ll(len ll))))},
296d          —      → {(uidLI(in(ll(i-1))),uidLI(em(ll(i-1))))},
296d                      (uidLI(in(ll(i))),uidLI(em(ll(i))))}
296d      end

```

The example above illustrated ‘domain determination’ with respect to endurants. Typically “endurant determination” is expressed in terms of axioms that limit state spaces — where “endurant instantiation” typically “limited” the mereology of endurants: how parts are related to one another. We shall not exemplify domain determination with respect to perdurants.

Discussion

The borderline between instantiation and determination is fuzzy. Whether, as an example, fixing the number of toll-road intersection hubs to a constant value, e.g., $n=7$, is instantiation or determination, is really a matter of choice !

5.4.4 Domain Extension

Definition 38 Extension: By **domain extension** we understand the introduction of endurants (see Sect. 5.4.4) and perdurants (see Sect. 5.5.2) that were not feasible in the original domain, but for which, with computing and communication, and with new, emerging technologies, for example, sensors, actuators and satellites, there is the possibility of feasible implementations, hence the requirements, that what is introduced becomes part of the unfolding requirements prescription ☺

Endurant Extensions

Definition 39 Endurant Extension: By an **endurant extension** we understand the introduction of one or more endurants into the projected, instantiated and determined domain \mathcal{D}_R resulting in domain \mathcal{D}_R' , such that these form a conservative extension of the theory, $\mathcal{T}_{\mathcal{D}_R}$ denoted by the domain requirements \mathcal{D}_R (i.e., “before” the extension), that is: every theorem of $\mathcal{T}_{\mathcal{D}_R}$ is still a theorem of $\mathcal{T}_{\mathcal{D}_R}'$.

Usually domain extensions involve one or more of the already introduced sorts. In Example 5.10 we introduce (i.e., “extend”) vehicles with GPSS-like sensors, and introduce toll-gates with entry sensors, vehicle identification sensors, gate actuators and exit sensors. Finally road pricing calculators are introduced.

Example 5.10. . Domain Requirements — Endurant Extension: We present the extensions in several steps. Some of them will be developed in this section. Development of the remaining will be deferred to Sect. 5.5.1. The reason for this deferment is that those last steps are examples of interface requirements. The initial extension-development steps are: [a] vehicle extension, [b] sort and unique identifiers of road price calculators, [c] vehicle to road pricing calculator channel, [d] sorts and dynamic attributes of toll-gates, [e] road pricing calculator attributes, [f] “total” system state, and [g] the overall system behaviour. This decomposition establishes system interfaces in “small, easy steps”.

[a] Vehicle Extension:

297 There is a domain, $\delta_\mathcal{E}:\Delta_\mathcal{E}$, which contains
 298 a fleet, $f_\mathcal{E}:F_\mathcal{E}$, that is,
 299 a set, $vs_\mathcal{E}:VS_\mathcal{E}$, of
 300 extended vehicles, $v_\mathcal{E}:V_\mathcal{E}$ — their extension amounting to
 301 a dynamic reactive attribute, whose value, $ti_gpos:TiGPos$, at any time, reflects that vehicle's *time-stamped global position*.¹⁶
 302 The vehicle's GNSS receiver calculates, loc_pos , its local position, $lpos:LPos$, based on these signals.
 303 Vehicles access these external attributes via the external attribute channel, $attr_TiGPos_ch$.

type

297 $\Delta_\mathcal{E}$
 298 $F_\mathcal{E}$
 299 $VS_\mathcal{E} = V_\mathcal{E}\text{-set}$
 300 $V_\mathcal{E}$
 301 $TiGPos = \mathbb{T} \times GPos$
 302 $GPos, LPos$

value

297 $\delta_\mathcal{E}:\Delta_\mathcal{E}$
 298 $obs_part_F_\mathcal{E}: \Delta_\mathcal{E} \rightarrow F_\mathcal{E}$
 298 $f = obs_part_F_\mathcal{E}(\delta_\mathcal{E})$
 299 $obs_part_VS_\mathcal{E}: F_\mathcal{E} \rightarrow VS_\mathcal{E}$
 299 $vs = obs_part_VS_\mathcal{E}(f)$
 299 $vis = xtr_vis(vs)$
 301 $attr_TiGPos_ch[vi]?$
 302 $loc_pos: GPos \rightarrow LPos$

channel

302 $\{attr_TiGPos_ch[vi]|vi:VI \bullet vi \in vis\}:TiGPos$

We define two auxiliary functions,

304 xtr_vs , which given a domain, or a fleet, extracts its set of vehicles, and
 305 xtr_vis which given a set of vehicles generates their unique identifiers.

¹⁶ We refer to literature on GNSS, *global navigation satellite systems*. The simple vehicle position, $vp:SVPos$, is determined from three to four time-stamped signals received from a like number of GNSS satellites [83].

value

```

304 xtr_vs: ( $\Delta_{\mathcal{E}} | F_{\mathcal{E}} | VS_{\mathcal{E}}$ )  $\rightarrow$   $V_{\mathcal{E}}\text{-set}$ 
304 xtr_vs(arg)  $\equiv$ 
304   is_ $\Delta_{\mathcal{E}}$ (arg)  $\rightarrow$  obs_part_VS $_{\mathcal{E}}$ (obs_part_F $_{\mathcal{E}}$ (arg)),
304   is_F $_{\mathcal{E}}$ (arg)  $\rightarrow$  obs_part_VS $_{\mathcal{E}}$ (arg),
304   is_VS $_{\mathcal{E}}$ (arg)  $\rightarrow$  arg
305 xtr_vis: ( $\Delta_{\mathcal{E}} | F_{\mathcal{E}} | VS_{\mathcal{E}}$ )  $\rightarrow$   $VI\text{-set}$ 
305 xtr_vis(arg)  $\equiv$  {uid_VI(v) | v  $\in$  xtr_vs(arg)}

```

[b] Road Pricing Calculator: Basic Sort and Unique Identifier:

306 The domain $\delta_{\mathcal{E}}: \Delta_{\mathcal{E}}$, also contains a pricing calculator, $c: C_{\delta_{\mathcal{E}}}$, with unique identifier $ci: CI$.

type

```
306 C, CI
```

value

```

306 obs_part_C:  $\Delta_{\mathcal{E}} \rightarrow C$ 
306 uid_CI:  $C \rightarrow CI$ 
306 c = obs_part_C( $\delta_{\mathcal{E}}$ )
306 ci = uid_CI(c)

```

[c] Vehicle to Road Pricing Calculator Channel:

307 Vehicles can, on their own volition, offer the timed local position, $viti_lpos: VITiLPos$
 308 to the pricing calculator, $c: C_{\delta_{\mathcal{E}}}$ along a vehicles-to-calculator channel, v_c_ch .

type

```
307  $VITiLPos = VI \times (\mathbb{T} \times LPos)$ 
```

channel

```
308 {v_c_ch[vi,ci] | vi: VI, ci: CI • vi  $\in$  vis  $\wedge$  ci = uid_C(c)} :  $VITiLPos$ 
```

[d] Toll-gate Sorts and Dynamic Types:

We extend the domain with toll-gates for vehicles entering and exiting the toll-road entry and exit links. Figure 5.2 illustrates the idea of gates.

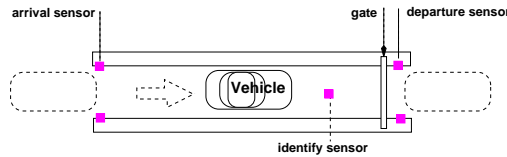


Fig. 5.2. A toll plaza gate

Figure 5.2 is intended to illustrate a vehicle entering (or exiting) a toll-road arrival link. The toll-gate is equipped with three sensors: an arrival sensor, a vehicle identification sensor and an departure sensor. The arrival sensor serves to prepare the vehicle identification sensor. The departure sensor serves to prepare the gate for closing when a vehicle has passed. The vehicle identify sensor identifies the vehicle and “delivers” a pair: the current time and the vehicle identifier. Once the vehicle identification sensor has identified a vehicle the gate opens and a message is sent to the road pricing calculator as to the passing vehicle’s identity and the identity of the link associated with the toll-gate (see Items 325- 326 on Page 161).

309 The domain contains the extended net, $n: N_{\mathcal{E}}$,

310 with the net extension amounting to the toll-road net, $\text{TRN}_{\mathcal{E}}$, that is, the instantiated toll-road net, $\text{trn}:\text{TRN}_{\mathcal{J}}$, is extended, into $\text{trn}:\text{TRN}_{\mathcal{E}}$, with entry, $\text{eg}:\text{EG}$, and exit, $\text{xg}:\text{XG}$, toll-gates.

From entry- and exit-gates we can observe

311 their unique identifier and
 312 their mereology: pairs of entry-, respectively exit link and calculator unique identifiers; further
 313 a pair of gate entry and exit sensors modeled as external attribute channels, $(\text{ges}:\text{ES}, \text{gls}:\text{XS})$, and
 314 a time-stamped vehicle identity sensor modeled as external attribute channels.

type

309 $\text{N}_{\mathcal{E}}$
 310 $\text{TRN}_{\mathcal{E}} = (\text{EG} \times \text{XG})^* \times \text{TRN}_{\mathcal{J}}$
 311 GI

value

309 $\text{obs_part_N}_{\mathcal{E}}: \Delta_{\mathcal{E}} \rightarrow \text{N}_{\mathcal{E}}$
 310 $\text{obs_part_TRN}_{\mathcal{E}}: \text{N}_{\mathcal{E}} \rightarrow \text{TRN}_{\mathcal{E}}$
 311 $\text{uid_G}: (\text{EG} \times \text{XG}) \rightarrow \text{GI}$
 312 $\text{obs_mereo_G}: (\text{EG} \times \text{XG}) \rightarrow (\text{LI} \times \text{CI})$
 310 $\text{trn}:\text{TRN}_{\mathcal{E}} = \text{obs_part_TRN}_{\mathcal{E}}(\delta_{\mathcal{E}})$

channel

313 $\{\text{attr_entry_ch}[\text{gi}] | \text{gi}:\text{GI} \times \text{tr_eGlds}(\text{trn})\}$ "enter"
 313 $\{\text{attr_exit_ch}[\text{gi}] | \text{gi}:\text{GI} \times \text{tr_xGlds}(\text{trn})\}$ "exit"
 314 $\{\text{attr_identity_ch}[\text{gi}] | \text{gi}:\text{GI} \times \text{tr_Glds}(\text{trn})\}$ TIVI

type

314 $\text{TIVI} = \mathbb{T} \times \text{VI}$

We define some **auxiliary functions** over toll-road nets, $\text{trn}:\text{TRN}_{\mathcal{E}}$:

315 tr_eGl extracts the list of entry gates,
 316 tr_xGl extracts the list of exit gates,
 317 tr_eGlds extracts the set of entry gate identifiers,
 318 tr_xGlds extracts the set of exit gate identifiers,
 319 tr_Gs extracts the set of all gates, and
 320 tr_Glds extracts the set of all gate identifiers.

value

315 $\text{tr_eGl}: \text{TRN}_{\mathcal{E}} \rightarrow \text{EG}^*$
 315 $\text{tr_eGl}(\text{pgl}, _) \equiv \{\text{eg} | (\text{eg}, \text{xg}): (\text{EG}, \text{XG}) \bullet (\text{eg}, \text{xg}) \in \text{elems pgl}\}$
 316 $\text{tr_xGl}: \text{TRN}_{\mathcal{E}} \rightarrow \text{XG}^*$
 316 $\text{tr_xGl}(\text{pgl}, _) \equiv \{\text{xg} | (\text{eg}, \text{xg}): (\text{EG}, \text{XG}) \bullet (\text{eg}, \text{xg}) \in \text{elems pgl}\}$
 317 $\text{tr_eGlds}: \text{TRN}_{\mathcal{E}} \rightarrow \text{GI-set}$
 317 $\text{tr_eGlds}(\text{pgl}, _) \equiv \{\text{uid_GI}(\text{g}) | \text{g}:\text{EG} \bullet \text{g} \in \text{tr_eGs}(\text{pgl}, _)\}$
 318 $\text{tr_xGlds}: \text{TRN}_{\mathcal{E}} \rightarrow \text{GI-set}$
 318 $\text{tr_xGlds}(\text{pgl}, _) \equiv \{\text{uid_GI}(\text{g}) | \text{g}:\text{EG} \bullet \text{g} \in \text{tr_xGs}(\text{pgl}, _)\}$
 319 $\text{tr_Gs}: \text{TRN}_{\mathcal{E}} \rightarrow \text{G-set}$
 319 $\text{tr_Gs}(\text{pgl}, _) \equiv \text{tr_eGs}(\text{pgl}, _) \cup \text{tr_xGs}(\text{pgl}, _)$
 320 $\text{tr_Glds}: \text{TRN}_{\mathcal{E}} \rightarrow \text{GI-set}$
 320 $\text{tr_Glds}(\text{pgl}, _) \equiv \text{tr_eGlds}(\text{pgl}, _) \cup \text{tr_xGlds}(\text{pgl}, _)$

321 A **well-formedness condition** expresses

- a that there are as many entry end exit gate pairs as there are toll-plazas,
- b that all gates are uniquely identified, and
- c that each entry [exit] gate is paired with an entry [exit] link and has that link's unique identifier as one element of its mereology, the other elements being the calculator identifier and the vehicle identifiers.

The well-formedness relies on awareness of

322 the unique identifier, $\text{ci}:\text{CI}$, of the road pricing calculator, $\text{c}:\text{C}$, and
 323 the unique identifiers, $\text{vis}:\text{VI-set}$, of the fleet vehicles.

axiom

321 $\forall n:\text{N}_{\mathcal{E}_3}, \text{trn}:\text{TRN}_{\mathcal{E}_3} \bullet$
 321 $\text{let } (\text{exgl}, (\text{exl}, \text{hl}, \text{lll})) = \text{obs_part_TRN}_{\mathcal{E}_3}(n) \text{ in}$
 321a $\text{len exgl} = \text{len exl} = \text{len hl} = \text{len lll} + 1$

```

321b  ∧ card xtr_Glds(exgl) = 2 * len exgl
321c  ∧ ∀ i:Nat·i ∈ inds exgl•
321c    let ((eg,xg),(el,xl)) = (exgl(i),exl(i)) in
321c    obs_mereo_G(eg) = (uid_U(el),ci,vis)
321c  ∧ obs_mereo_G(xg) = (uid_U(xl),ci,vis)
321    end end

```

[e] Toll-gate to Calculator Channels:

324 We distinguish between entry and exit gates.
 325 Toll road entry and exit gates offers the road pricing calculator a pair: whether it is an entry or an exit gates, and
 pair of the passing vehicle's identity and the time-stamped identity of the link associated with the toll-gate
 326 to the road pricing calculator via a (gate to calculator) channel.

```

type
324  EE = "entry"|"exit"
325  EEViTiLI = EE × (VI × (T × SonL))
channel
326  {g_c_ch[gi,ci]|gi:GI•gi ∈ gis}:EETiViLI

```

[f] Road Pricing Calculator Attributes:

327 The road pricing attributes include a programmable traffic map, trm:TRM, which, for each vehicle inside the
 toll-road net, records a chronologically ordered list of each vehicle's timed position, (τ ,lpos), and
 328 a static (total) road location function, vplf:VPLF. The vehicle position location function, vplf:VPLF, which,
 given a local position, lpos:LPos, yields either the simple vehicle position, svpos:SVPos, designated by the GNSS-
 provided position, or yields the response that the provided position is off the toll-road net The vplf:VPLF function
 is constructed, construct_vplf,
 329 from awareness, of a geodetic road map, GRM, of the topology of the extended net, n_ℓ:N_ℓ, including the mere-
 ology and the geodetic attributes of links and hubs.

```

type
327  TRM = VI →h (T × SVPos)*
328  VPLF = GRM → LPos → (SVPos | "off_N")
329  GRM
value
327  attr_TRM: Cℓ → TRM
328  attr_VPLF: Cℓ → VPLF

```

The geodetic road map maps geodetic locations into hub and link identifiers.

239 Geodetic link locations represent the set of point locations of a link.

235a Geodetic hub locations represent the set of point locations of a hub.

330 A geodetic road map maps geodetic link locations into link identifiers and geodetic hub locations into hub identi-
 fiers.

331 We sketch the construction, geo_GRM, of geodetic road maps.

```

type
330  GRM = (GeoL →h LI) ∪ (GeoH →h HI)
value
331  geo_GRM: N → GRM
331  geo_GRM(n) ≡
331    let ls = xtr_links(n), hs = xtr_hubs(n) in
331    [attr_GeoL(l) → uid_LI(l) | l:L•l ∈ ls]
331    ∪
331    [attr_GeoH(h) → uid_HI(h) | h:H•h ∈ hs] end

```

332 The `vplf:VPLF` function obtains a simple vehicle position, `svpos`, from a geodetic road map, `grm:GRM`, and a local position, `lpos`:

value

```
332 obtain_SVPos: GRM → LPos → SVPos
332 obtain_SVPos(grm)(lpos) as svpos
332 post: case svpos of
332     SatH(hi) → within(lpos, grm(hi)),
332     SonL(li) → within(lpos, grm(li)),
332     "off_N" → true end
```

where *within* is a predicate which holds if its first argument, a local position calculated from a GNSS-generated global position, falls within the point set representation of the geodetic locations of a link or a hub. The design of the *obtain_SVPos* represents an interesting challenge.

[g] “Total” System State:

Global values:

```
333 There is a given domain,  $\delta_{\mathcal{E}}:\Delta_{\mathcal{E}}$ ;
334 there is the net,  $n_{\mathcal{E}}:\mathbf{N}_{\mathcal{E}}$ , of that domain;
335 there is toll-road net,  $trn_{\mathcal{E}}:\mathbf{TRN}_{\mathcal{E}}$ , of that net;
336 there is a set,  $egs_{\mathcal{E}}:\mathbf{EG-set}$ , of entry gates;
337 there is a set,  $xgs_{\mathcal{E}}:\mathbf{XG-set}$ , of exit gates;
338 there is a set,  $gis_{\mathcal{E}}:\mathbf{GI-set}$ , of gate identifiers;
339 there is a set,  $vs_{\mathcal{E}}:\mathbf{V-set}$ , of vehicles;
340 there is a set,  $vis_{\mathcal{E}}:\mathbf{VI-set}$ , of vehicle identifiers;
341 there is the road-pricing calculator,  $c_{\mathcal{E}}:\mathbf{C}_{\mathcal{E}}$  and
342 there is its unique identifier,  $ci_{\mathcal{E}}:\mathbf{CI}$ .
```

value

```
333  $\delta_{\mathcal{E}}:\Delta_{\mathcal{E}}$ 
334  $n_{\mathcal{E}}:\mathbf{N}_{\mathcal{E}} = \mathbf{obs\_part\_N}_{\mathcal{E}}(\delta_{\mathcal{E}})$ 
335  $trn_{\mathcal{E}}:\mathbf{TRN}_{\mathcal{E}} = \mathbf{obs\_part\_TRN}_{\mathcal{E}}(n_{\mathcal{E}})$ 
336  $egs_{\mathcal{E}}:\mathbf{EG-set} = \mathbf{xtr\_egs}(trn_{\mathcal{E}})$ 
337  $xgs_{\mathcal{E}}:\mathbf{XG-set} = \mathbf{xtr\_xgs}(trn_{\mathcal{E}})$ 
338  $gis_{\mathcal{E}}:\mathbf{GI-set} = \mathbf{xtr\_gis}(trn_{\mathcal{E}})$ 
339  $vs_{\mathcal{E}}:\mathbf{V-set} = \mathbf{obs\_part\_VS}(\mathbf{obs\_part\_F}_{\mathcal{E}}(\delta_{\mathcal{E}}))$ 
340  $vis_{\mathcal{E}}:\mathbf{VI-set} = \{\mathbf{uid\_VI}(v_{\mathcal{E}}) \mid v_{\mathcal{E}}:\mathbf{V-set} \wedge v_{\mathcal{E}} \in vs_{\mathcal{E}}\}$ 
341  $c_{\mathcal{E}}:\mathbf{C}_{\mathcal{E}} = \mathbf{obs\_part\_C}_{\mathcal{E}}(\delta_{\mathcal{E}})$ 
342  $ci_{\mathcal{E}}:\mathbf{CI} = \mathbf{uid\_CI}(c_{\mathcal{E}})$ 
```

In the following we shall omit the cumbersome \mathcal{E} subscripts.

[h] “Total” System Behaviour:

The signature and definition of the system behaviour is sketched as are the signatures of the vehicle, toll-gate and road pricing calculator. We shall model the behaviour of the road pricing system as follows: we shall not model behaviours nets, hubs and links; thus we shall model only the behaviour of vehicles, *veh*, the behaviour of toll-gates, *gate*, and the behaviour of the road-pricing calculator, *calc*. The behaviours of vehicles and toll-gates are presented here. But the behaviour of the road-pricing calculator is “deferred” till Sect. 5.5.1 since it reflects an interface requirements.

```
343 The road pricing system behaviour, sys, is expressed as
    a the parallel,  $\parallel$ , (distributed) composition of the behaviours of all vehicles,
    b with the parallel composition of the parallel (likewise distributed) composition of the behaviours of all entry
      gates,
    c with the parallel composition of the parallel (likewise distributed) composition of the behaviours of all exit
      gates,
    d with the parallel composition of the behaviour of the road-pricing calculator,
```

```

value
343 sys: Unit → Unit
343 sys() ≡
343a   || {vehuid,V(v)(obs_mereo_V(v))|v:V•v ∈ vs}
343b   || || {gateuid,EG(eg)(obs_mereo_G(eg),"entry")|eg:EG•eg ∈ egs}
343c   || || {gateuid,XG(xg)(obs_mereo_G(xg),"exit")|xg:XG•xg ∈ xgs}
343d   ||   calcuid,C(c)(vis,gis)(rlf)(trm)

344 vehvi: (ci:CI×gis:GI-set) → in attr_TiGPos[vi] out v_c_ch[vi,ci] Unit
350 gategi: (ci:CI×VI-set×LI)×ee:EE →
350   in attr_entry_ch[gi,ci],attr_id_ch[gi,ci],attr_exit_ch[gi,ci]
350   out attr_barrier_ch[gi],g_c_ch[gi,ci] Unit
384 calcci: (vis:VI-set×gis:GI-set)×VPLF→TRM→
384   in {v_c_ch[vi,ci]|vi:VI•vi ∈ vis},{g_c_ch[gi,ci]|gi:GI•gi ∈ gis} Unit

```

We consider "entry" or "exit" to be a static attribute of toll-gates. The behaviour signatures were determined as per the techniques presented in [49, Sect. 4.1.1 and 4.5.2].

Vehicle Behaviour: We refer to the vehicle behaviour, in the domain, described in Sect. 5.2's The Road Traffic System Behaviour Items 262 and Items 263, Page 145 and, projected, Page 151.

344 Instead of moving around by explicitly expressed internal non-determinism¹⁷ vehicles move around by unstated internal non-determinism and instead receive their current position from the global positioning subsystem.
 345 At each moment the vehicle receives its time-stamped global position, $(\tau, \text{gpos}): \text{TiGPos}$,
 346 from which it calculates the local position, $\text{lpos}: \text{VPos}$
 347 which it then communicates, with its vehicle identification, $(\text{vi}, (\tau, \text{lpos}))$, to the road pricing subsystem —
 348 whereupon it resumes its vehicle behaviour.

```

value
344 vehvi: (ci:CI×gis:GI-set) →
344   in attr_TiGPos_ch[vi] out v_c_ch[vi,ci] Unit
344 vehvi(ci,gis) ≡
345   let  $(\tau, \text{gpos}) = \text{attr\_TiGPos\_ch}[vi]?$  in
346   let  $\text{lpos} = \text{loc\_pos}(\text{gpos})$  in
347   v_c_ch[vi,ci] !  $(\text{vi}, (\tau, \text{lpos}))$  ;
348   vehvi(ci,gis) end end
344 pre vi ∈ vis

```

The *vehicle* signature has $\text{attr_TiGPos_ch}[vi]$ model an external vehicle attribute and $\text{v_c_ch}[vi,ci]$ the embedded attribute sharing [49, Sect. 4.1.1 and 4.5.2] between vehicles (their position) and the price calculator's road map. The above behaviour represents an assumption about the behaviour of vehicles. If we were to design software for the monitoring and control of vehicles then the above vehicle behaviour would have to be refined in order to serve as a proper interface requirements. The refinement would include handling concerns about the drivers' behaviour when entering, passing and exiting toll-gates, about the proper function of the GNSS equipment, and about the safe communication with the road price calculator. The above concerns would already have been addressed in a model of *domain facets* such as *human behaviour*, *technology support*, proper tele-communications *scripts*, etcetera. We refer to [33].

Gate Behaviour: The entry and the exit gates have "vehicle enter", "vehicle exit" and "timed vehicle identification" sensors. The following assumption can now be made: during the time interval between a gate's vehicle "entry" sensor having first sensed a vehicle entering that gate and that gate's "exit" sensor having last sensed that vehicle leaving that gate that gate's vehicle time and "identify" sensor registers the time when the vehicle is entering the gate and that vehicle's unique identification. We sketch the toll-gate behaviour:

349 We parameterise the toll-gate behaviour as either an entry or an exit gate.
 350 Toll-gates operate autonomously and cyclically.
 351 The attr_enter_ch event "triggers" the behaviour specified in formula line Item 352–354 starting with a "Raise" barrier action.
 352 The time-of-passing and the identity of the passing vehicle is sensed by attr_passing_ch channel events.

¹⁷ We refer to Items 262b, 262c on Page 145 and 263b, 263(b)ii, 263c on Page 145

353 Then the road pricing calculator is informed of time-of-passing and of the vehicle identity vi and the link li associated with the gate – and with a "Lower" barrier action.

354 And finally, after that vehicle has left the entry or exit gate the barrier is again "Lower"ed and

355 that toll-gate's behaviour is resumed.

type

349 $EE = \text{"enter"} \mid \text{"exit"}$

value

```

350  $gate_{gi}: (ci:CI \times VI\text{-set} \times LI) \times ee:EE \rightarrow$ 
350   in  $attr\_enter\_ch[gi], attr\_passing\_ch[gi], attr\_leave\_ch[gi]$ 
350   out  $attr\_barrier\_ch[gi], g\_c\_ch[gi,ci]$  Unit
350  $gate_{gi}((ci,vis,li),ee) \equiv$ 
351    $attr\_enter\_ch[gi] ? ; attr\_barrier\_ch[gi] ! \text{"Lower"}$ 
352   let  $(\tau,vi) = attr\_passing\_ch[gi] ?$  in assert  $vi \in vis$ 
353    $(attr\_barrier\_ch[gi] ! \text{"Raise"}$ 
353    $\parallel g\_c\_ch[gi,ci] ! (ee,(vi,(\tau,SonL(li)))) ;$ 
354    $attr\_leave\_ch[gi] ? ; attr\_barrier\_ch[gi] ! \text{"Lower"}$ 
355    $gate_{gi}((ci,vis,li),ee)$ 
350   end
350   pre  $li \in lis$ 
```

The *gate* signature's $attr_enter_ch[gi]$, $attr_passing_ch[gi]$, $attr_barrier_ch[gi]$ and $attr_leave_ch[gi]$ model respective external attributes [49, Sect. 4.1.1 and 4.5.2] (the $attr_barrier_ch[gi]$ models reactive (i.e., output) attribute), while $g_c_ch[gi,ci]$ models the embedded attribute sharing between gates (their identification of vehicle positions) and the calculator road map. The above behaviour represents an assumption about the behaviour of toll-gates. If we were to design software for the monitoring and control of toll-gates then the above gate behaviour would have to be refined in order to serve as a proper interface requirements. The refinement would include handling concerns about the drivers' behaviour when entering, passing and exiting toll-gates, about the proper function of the entry, passing and exit sensors, about the proper function of the gate barrier (opening and closing), and about the safe communication with the road price calculator. The above concerns would already have been addressed in a model of *domain facets* such as *human behaviour*, *technology support*, proper tele-communications *scripts*, etcetera. We refer to [33] \square

We shall define the *calculator* behaviour in Sect. 5.5.1 on Page 169. The reason for this deferral is that it exemplifies interface requirements.

Discussion

The requirements assumptions expressed in the specifications of the vehicle and gate behaviours assume that these behave in an orderly fashion. But they seldom do! The $attr_TiGPos_ch$ sensor may fail. And so may the $attr_enter_ch$, $attr_passing_ch$, and $attr_leave_ch$ sensors and the $attr_barrier_ch$ actuator. These attributes represent support technology facets. They can fail. To secure fault tolerance one must prescribe very carefully what counter-measures are to be taken and/or the safety assumptions. We refer to [187, 120, 143]. They cover three alternative approaches to the handling of fault tolerance. Either of the approaches can be made to fit with our approach. First one can pursue our approach to where we stand now. Then we join the approaches of either of [187, 120, 143]. [120] likewise decompose the requirements prescription as is suggested here.

5.4.5 Requirements Fitting

Often a domain being described "fits" onto, is "adjacent" to, "interacts" in some areas with, another domain: *transportation* with *logistics*, *health-care* with *insurance*, *banking* with *securities trading* and/or *insurance*, and so on. The issue of requirements fitting arises when two or more software development projects are based on what appears to be the same domain. The problem then is to harmonise the two or more software development projects by harmonising, if not too late, their requirements developments.

We thus assume that there are n domain requirements developments, $d_{r_1}, d_{r_2}, \dots, d_{r_n}$, being considered, and that these pertain to the same domain — and can hence be assumed covered by a same domain description.

Definition 40 Requirements Fitting: By **requirements fitting** we mean a harmonisation of $n > 1$ domain requirements that have overlapping (shared) not always consistent parts and which results in n partial domain requirements', $p_{dr_1}, p_{dr_2}, \dots, p_{dr_m}$, and m shared domain requirements, $s_{dr_1}, s_{dr_2}, \dots, s_{dr_m}$, that “fit into” two or more of the partial domain requirements. The above definition pertains to the result of ‘fitting’. The next definition pertains to the act, or process, of ‘fitting’.

Definition 41 Requirements Harmonisation: By **requirements harmonisation** we mean a number of alternative and/or co-ordinated prescription actions, one set for each of the domain requirements actions: Projection, Instantiation, Determination and Extension. They are – we assume n separate software product requirements: Projection: If the n product requirements do not have the same projections, then identify a common projection which they all share, and refer to it as the common projection. Then develop, for each of the n product requirements, if required, a specific projection of the common one. Let there be m such specific projections, $m \leq n$. Instantiation: First instantiate the common projection, if any instantiation is needed. Then for each of the m specific projections instantiate these, if required. Determination: Likewise, if required, “perform” “determination” of the possibly instantiated common projection, and, similarly, if required, “perform” “determination” of the up to m possibly instantiated projections. Extension: Finally “perform extension” likewise: First, if required, of the common projection (etc.), then, if required, on the up m specific projections (etc.). These harmonization developments may possibly interact and may need to be iterated.

By a **partial domain requirements** we mean a domain requirements which is short of (that is, is missing) some prescription parts: text and formula. By a **shared domain requirements** we mean a domain requirements. By **requirements fitting** m shared domain requirements texts, $sdrs$, into n partial domain requirements we mean that there is for each partial domain requirements, pdr_i , an identified, non-empty subset of $sdrs$ (could be all of $sdrs$), $ssdrs_i$, such that textually conjoining $ssdrs_i$ to pdr_i , i.e., $ssdrs_i \oplus pdr_i$ can be claimed to yield the “original” d_{r_i} , that is, $\mathcal{M}(ssdrs_i \oplus pdr_i) \subseteq \mathcal{M}(d_{r_i})$, where \mathcal{M} is a suitable meaning function over prescriptions.

5.4.6 Discussion

Facet-oriented Fittings: An altogether different way of looking at domain requirements may be achieved when also considering domain facets — not covered in neither the example of Sect. 5.2 nor in this section (i.e., Sect. 5.4) nor in the following two sections. We refer to [33].

Example 5.11. . Domain Requirements — Fitting: Example 5.10 hints at three possible sets of interface requirements: (i) for a road pricing [sub-]system, as will be illustrated in Sect. 5.5.1; (ii) for a vehicle monitoring and control [sub-]system, and (iii) for a toll-gate monitoring and control [sub-]system. The vehicle monitoring and control [sub-]system would focus on implementing the vehicle behaviour, see Items 344– 348 on Page 163. The toll-gate monitoring and control [sub-]system would focus on implementing the calculator behaviour, see Items 350– 355 on the preceding page. The fitting amounts to (a) making precise the (narrative and formal) texts that are specific to each of the three (i–iii) separate sub-system requirements are kept separate; (b) ensuring that (meaning-wise) shared texts that have different names for (meaning-wise) identical entities have these names renamed appropriately; (c) that these texts are subject to commensurate and ameliorated further requirements development; etcetera.

5.5 Interface and Derived Requirements

We remind the reader that **interface requirements** can be expressed only using terms from both the domain and the machine. Users are not part of the machine. So no reference can be made to users, such as “the system must be user friendly”, and the like!¹⁸ By **interface requirements** we [also] mean

¹⁸ So how do we cope with the statement: “the system must be user friendly”? We refer to Sect. 5.5.3 on Page 173 for a discussion of this issue.

requirements prescriptions which refines and extends the domain requirements by considering those requirements of the domain requirements whose endurants (parts, materials) and perdurants (actions, events and behaviours) are “shared” between the domain and the machine (being requirements prescribed) ☉ The two interface requirements definitions above go hand-in-hand, i.e., complement one-another.

By **derived requirements** we mean *requirements prescriptions* which are expressed in terms of the machine concepts and facilities introduced by the emerging requirements ☉

5.5.1 Interface Requirements

Shared Phenomena

By **sharing** we mean (a) that *some or all properties* of an **endurant** is represented both in the domain and “inside” the machine, and that their machine representation must at suitable times reflect their state in the domain; and/or (b) that an **action** requires a sequence of several “on-line” interactions between the machine (being requirements prescribed) and the domain, usually a person or another machine; and/or (c) that an **event** arises either in the domain, that is, in the environment of the machine, or in the machine, and need be communicated to the machine, respectively to the environment; and/or (d) that a **behaviour** is manifested both by actions and events of the domain and by actions and events of the machine ☉ So a systematic reading of the domain requirements shall result in an identification of all shared endurants, parts, materials and components; and perdurants actions, events and behaviours. Each such shared phenomenon shall then be individually dealt with: **endurant sharing** shall lead to interface requirements for data initialisation and refreshment as well as for access to endurant attributes; **action sharing** shall lead to interface requirements for interactive dialogues between the machine and its environment; **event sharing** shall lead to interface requirements for how such event are communicated between the environment of the machine and the machine; and **behaviour sharing** shall lead to interface requirements for action and event dialogues between the machine and its environment.

Environment–Machine Interface:

Domain requirements extension, Sect. 5.4.4, usually introduce new endurants into (i.e., ‘extend’ the) domain. Some of these endurants may become elements of the domain requirements. Others are to be projected “away”. Those that are let into the domain requirements either have their endurants represented, somehow, also in the machine, or have (some of) their properties, usually some attributes, accessed by the machine. Similarly for perdurants. Usually the machine representation of shared perdurants access (some of) their properties, usually some attributes. The interface requirements must spell out which domain extensions are shared. Thus domain extensions may necessitate a review of domain projection, instantiations and determination. In general, there may be several of the projection–eliminated parts (etc.) whose dynamic attributes need be accessed in the usual way, i.e., by means of `attr_XYZ_ch` channel communications (where XYZ is a projection–eliminated part attribute).

Example 5.12. . Interface Requirements — Projected Extensions: We refer to Fig. 5.2 on Page 159. We do not represent the GNSS system in the machine: only its “effect”: the ability to record global positions by accessing the GNSS attribute (channel):

channel

303 {attr_TiGPos_ch[vi]|vi:VI•vi ∈ xtr_VIs(vs)}: TiGPos

And we do not really represent the gate nor its sensors and actuator in the machine. But we do give an idealised description of the gate behaviour, see Items 350–355 Instead we represent their dynamic gate attributes:

- (313) the vehicle entry sensors (leftmost ■s),
- (313) the vehicle identity sensor (center ■), and
- (314) the vehicle exit sensors (rightmost ■s)

by channels — we refer to Example 5.10 (Sect. 5.5.1, Page 160):

channel

313 {attr_entry_ch[gi]|gi:GI•xtr_eGlds(trn)} "enter"
 313 {attr_exit_ch[gi]|gi:GI•xtr_xGlds(trn)} "exit"
 314 {attr_identity_ch[gi]|gi:GI•xtr_Glds(trn)} TIVI □

Shared Endurants

Example 5.13. . **Interface Requirements. Shared Endurants:** The main shared endurants are the vehicles, the net (hubs, links, toll-gates) and the price calculator. As domain endurants hubs and links undergo changes, all the time, with respect to the values of several attributes: *length*, *geodetic information*, *names*, *wear and tear* (where-ever applicable), *last/next scheduled maintenance* (where-ever applicable), *state* and *state space*, and many others. Similarly for vehicles: their position, velocity and acceleration, and many other attributes. We then come up with something like hubs and links are to be represented as tuples of relations; each net will be represented by a pair of relations a hubs relation and a links relation; each hub and each link may or will be represented by several tuples; etcetera. In this database modeling effort it must be secured that “standard” operations on nets, hubs and links can be supported by the chosen relational database system

Data Initialisation:

In general, one must prescribe data initialisation, that is provision for an interactive user interface dialogue with a set of proper display screens, one for establishing net, hub or link attributes names and their types, and, for example, two for the input of hub and link attribute values. Interaction prompts may be prescribed: next input, on-line vetting and display of evolving net, etc. These and many other aspects may therefore need prescriptions.

Example 5.14. . **Interface Requirements. Shared Endurant Initialisation:** The domain is that of the road net, $n:N$. By ‘shared road net initialisation’ we mean the “ab initio” establishment, “from scratch”, of a data base recording the properties of all links, $l:L$, and hubs, $h:H$, their unique identifications, **uid**_L(l) and **uid**_H(h), their mereologies, **obs_mereo**_L(l) and **obs_mereo**_H(h), the initial values of all their static and programmable attributes and the access values, that is, channel designations for all other attribute categories.

356 There are r_l and r_h “recorders” recording link, respectively hub properties – with each recorder having a unique identity.
 357 Each recorder is charged with the recording of a set of links or a set of hubs according to some partitioning of all such.
 358 The recorders inform a central data base, **net_db**, of their recordings $(r_i, \text{hol}, (u_j, m_j, \text{attrs}_j))$ where
 359 r_i is the identity of the recorder,
 360 hol is either a **hub** or a **link** literal,
 361 $u_j = \text{uid}_L(l)$ or $\text{uid}_H(h)$ for some link or hub,
 362 $m_j = \text{obs_mereo}_L(l)$ or $\text{obs_mereo}_H(h)$ for that link or hub and
 363 attrs_j are *attributes* for that link or hub — where *attributes* is a function which “records” all respective static and dynamic attributes (left undefined).

type

356 RI

value

356 $rl, rh: \text{NAT}$ **axiom** $rl > 0 \wedge rh > 0$

type

358 $M = RI \times \text{"link"} \times \text{LNK} \mid RI \times \text{"hub"} \times \text{HUB}$

358 $\text{LNK} = LI \times HI\text{-set} \times \text{LATTRS}$

358 $\text{HUB} = HI \times LI\text{-set} \times \text{HATTRS}$

value

357 partitioning: $L\text{-set} \rightarrow \text{Nat} \rightarrow (L\text{-set})^*$

357 $\mid H\text{-set} \rightarrow \text{Nat} \rightarrow (H\text{-set})^*$

357 partitioning(s)(r) as sl

357 **post:** $\text{len } sl = r \wedge \bigcup \text{elems } sl = s$

357 $\wedge \forall si, sj: (L\text{-set} \mid H\text{-set}) \cdot$

357 $si \neq \{\} \wedge sj \neq \{\} \wedge \{si, sj\} \subseteq \text{elems } ss \Rightarrow si \cap sj = \{\}$

364 The $r_l + r_h$ recorder behaviours interact with the one **net_db** behaviour

channel

364 r_db: $RI \times (LNK|HUB)$

value

364 link_rec: $RI \rightarrow L\text{-set} \rightarrow \text{out r_db Unit}$

364 hub_rec: $RI \rightarrow H\text{-set} \rightarrow \text{out r_db Unit}$

364 net_db: $\text{Unit} \rightarrow \text{in r_db Unit}$

365 The data base behaviour, net_db, offers to receive messages from the link and hub recorders.

366 The data base behaviour, net_db, deposits these messages in respective variables.

367 Initially there is a net, $n : N$,

368 from which is observed its links and hubs.

369 These sets are partitioned into r_l , respectively r_h length lists of non-empty links and hubs.

370 The ab-initio data initialisation behaviour, ab_initio_data, is then the parallel composition of link recorder, hub recorder and data base behaviours with link and hub recorder being allotted appropriate link, respectively hub sets.

371 We construct, for technical reasons, as the reader will soon see, disjoint lists of link, respectively hub recorder identities.

value

365 net_db:

variable

366 lnk_db: $(RI \times LNK)\text{-set}$

366 hub_db: $(RI \times HUB)\text{-set}$

value

367 n:N

368 ls:L-set = obs_Ls(obs_LS(n))

368 hs:H-set = obs_Hs(obs_HS(n))

369 lsl:(L-set)* = partitioning(ls)(rl)

369 lhl:(H-set)* = partitioning(hs)(rh)

371 rll:RI* **axiom** len rll = rl = card elems rll

371 rhl:RI* **axiom** len rhl = rh = card elems rhl

370 ab_initio_data: $\text{Unit} \rightarrow \text{Unit}$

370 ab_initio_data() \equiv

370 || {lnk_rec(rll[i])(lsl[i])|i:Nat.1 ≤ i ≤ rl} ||

370 || {hub_rec(rhl[i])(lhl[i])|i:Nat.1 ≤ i ≤ rh} ||

370 || net_db()

372 The link and the hub recorders are near-identical behaviours.

373 They both revolve around an imperatively stated **for all ... do ... end**. The selected link (or hub) is inspected and the “data” for the data base is prepared from

374 the unique identifier,

375 the mereology, and

376 the attributes.

377 These “data” are sent, as a message, prefixed the senders identity, to the data base behaviour.

378 We presently leave the ... unexplained.

value

364 link_rec: $RI \rightarrow L\text{-set} \rightarrow \text{Unit}$

372 link_rec(ri,ls) \equiv

373 **for** $\forall l:L.l \in ls$ **do** uid_L(l)

374 **let** lnk = (uid_L(l),

375 **obs_mereo_L(l),**

376 **attributes(l)) in**

377 rdb ! (ri, "link", lnk);

378 **... end**

373 **end**

```

364 hub_rec: RI × H-set → Unit
372 hub_rec(ri,hs) ≡
373   for ∀ h:H•h ∈ hs do uid_H(h)
374     let hub = (uid_L(h),
375               obs_mereo_H(h),
376               attributes(h)) in
377     rdb ! (ri,"hub",hub);
378   ... end
373 end

```

379 The net_db data base behaviour revolves around a seemingly “never-ending” cyclic process.
 380 Each cycle “starts” with acceptance of some,
 381 either link or hub data.
 382 If link data then it is deposited in the link data base,
 383 if hub data then it is deposited in the hub data base.

```

value
379 net_db() ≡
380   let (ri,hol,data) = r_db ? in
381   case hol of
382     "link" → ... ; lnk_db := lnk_db ∪ (ri,data),
383     "hub"  → ... ; hub_db := hub_db ∪ (ri,data)
381   end end ;
379' ... ;
379 net_db()

```

The above model is an idealisation. It assumes that the link and hub data represent a well-formed net. Included in this well-formedness are the following issues: (a) that all link or hub identifiers are communicated exactly once, (b) that all mereologies refer to defined parts, and (c) that all attribute values lie within an appropriate value range. If we were to cope with possible recording errors then we could, for example, extend the model as follows: (i) when a link or a hub recorder has completed its recording then it increments an initially zero counter (say at formula Item 378); (ii) before the net data base recycles it tests whether all recording sessions has ended and then proceeds to check the data base for well-formedness issues (a–b–c) (say at formula Item 379') □

The above example illustrates the ‘interface’ phenomenon: In the formulas, for example, we show both manifest domain entities, viz., n, l, h etc., and abstract (required) software objects, viz., $(ui, me, attrs)$.

Data Refreshment:

One must also prescribe data refreshment: an interactive user interface dialogue with a set of proper display screens one for selecting the updating of net, of hub or of link attribute names and their types and, for example, two for the respective update of hub and link attribute values. Interaction-prompts may be prescribed: next update, on-line vetting and display of revised net, etc. These and many other aspects may therefore need prescriptions.

Shared Perdurants

We can expect that for every part in the domain that is shared with the machine and for which there is a corresponding behaviour of the domain there might be a corresponding process of the machine. If a projected, instantiated, ‘determined’ and possibly extended domain part is dynamic, then it is definitely a candidate for being shared and having an associated machine process. We now illustrate the concept of shared perdurants via the domain requirements extension example of Sect. 5.4.4, i.e. Example 5.10 Pages 158–164.

Example 5.15. . Interface Requirements — Shared Behaviours: Road Pricing Calculator Behaviour:

```

384 The road-pricing calculator alternates between offering to accept communication from
385 either any vehicle
386 or any toll-gate.

```

```

384 calc: ci:CI × (vis:VI-set × gis:GI-set) → RLF → TRM →
385   in {v_c_ch[ci,vi]|vi:VI•vi ∈ vis},
386   {g_c_ch[ci,gi]|gi:GI•gi ∈ gis} Unit
384 calc(ci,(vis,gis))(rlf)(trm) ≡
385   react_to_vehicles(ci,(vis,gis))(rlf)(trm)
384   []
386   react_to_gates(ci,(vis,gis))(rlf)(trm)
384   pre ci = ciℓ ∧ vis = visℓ ∧ gis = gisℓ

```

The calculator signature's $v_c_ch[ci,vi]$ and $g_c_ch[ci,gi]$ model the embedded attribute sharing between vehicles (their position), respectively gates (their vehicle identification) and the calculator road map [49, Sect. 4.1.1 and 4.5.2].

387 If the communication is from a vehicle inside the toll-road net
 388 then its toll-road net position, vp , is found from the road location function, rlf ,
 389 and the calculator resumes its work with the traffic map, trm , suitably updated,
 390 otherwise the calculator resumes its work with no changes.

```

385 react_to_vehicles(ci,(vis,gis),vplf)(trm) ≡
385   let (vi,(τ,lpos)) = [] {v_c_ch[ci,vi]?|vi:VI•vi ∈ vis} in
387   if vi ∈ dom trm
388   then let vp = vplf(lpos) in
389     calc(ci,(vis,gis),vplf)(trm†[vi→trm^(⟨(τ,vp))]) end
390   else calc(ci,(vis,gis),vplf)(trm) end end

```

391 If the communication is from a gate,
 392 then that gate is either an entry gate or an exit gate;
 393 if it is an entry gate
 394 then the calculator resumes its work with the vehicle (that passed the entry gate) now recorded, afresh, in the traffic map, trm .
 395 Else it is an exit gate and
 396 the calculator concludes that the vehicle has ended its to-be-paid-for journey inside the toll-road net, and hence to be billed;
 397 then the calculator resumes its work with the vehicle now removed from the traffic map, trm .

```

386 react_to_gates(ci,(vis,gis),vplf)(trm) ≡
386   let (ee,(τ,(vi,li))) = [] {g_c_ch[ci,gi]?|gi:GI•gi ∈ gis} in
392   case ee of
393     "Enter" →
394       calc(ci,(vis,gis),vplf)(trm ∪ [vi→⟨(τ,SonL(li))]),
395     "Exit" →
396       billing(vi,trm(vi)^(⟨(τ,SonL(li))));
397       calc(ci,(vis,gis),vplf)(trm \ {vi}) end end

```

The above behaviour is the one for which we are to design software □

5.5.2 Derived Requirements

Definition 42 Derived Perdurant: By a **derived perdurant** we shall understand a perdurant which is not shared with the domain, but which focus on exploiting facilities of the software or hardware of the machine ☉

“Exploiting facilities of the software”, to us, means that requirements, imply the presence, in the machine, of concepts (i.e., hardware and/or software), and that it is these concepts that the **derived requirements** “rely” on. We illustrate all three forms of perdurant extensions: derived actions, derived events and derived behaviours.

Derived Actions

Definition 43 Derived Action: By a **derived action** we shall understand (a) a conceptual action (b) that calculates a usually non-Boolean valued property from, and possibly changes to (c) a machine behaviour state (d) as instigated by some actor ☺

Example 5.16. . Domain Requirements. Derived Action: Tracing Vehicles: The example is based on the *Road Pricing Calculator Behaviour* of Example 5.15 on Page 169. The “external” actor, i.e., a user of the *Road Pricing Calculator* system wishes to trace specific vehicles “cruising” the toll-road. That user (a *Road Pricing Calculator* staff), issues a command to the *Road Pricing Calculator* system, with the identity of a vehicle not already being traced. As a result the *Road Pricing Calculator* system augments a possibly void trace of the timed toll-road positions of vehicles. We augment the definition of the *calculator* definition Items 384–397, Pages 169–170.

398 Traces are modeled by a pair of dynamic attributes:

a as a programmable attribute, $tra:TRA$, of the set of identifiers of vehicles being traced, and

b as a reactive attribute, $vdu:VDU^{19}$, that maps vehicle identifiers into time-stamped sequences of simple vehicle positions, i.e., as a subset of the $trm:TRM$ programmable attribute.

399 The actor-to-calculator *begin* or *end* trace command, $cmd:Cmd$, is modeled as an autonomous dynamic attribute of the *calculator*.

400 The *calculator* signature is furthermore augmented with the three attributes mentioned above.

401 The occurrence and handling of an actor trace command is modeled as a non-deterministic external choice and a *react_to_trace_cmd* behaviour.

402 The reactive attribute value ($attr_vdu_ch?$) is that subset of the traffic map (trm) which records just the time-stamped sequences of simple vehicle positions being traced (tra).

type

398a $TRA = VI\text{-}set$

398b $VDU = TRM$

399 $Cmd = BTr \mid ETr$

399 $BTr :: VI$

399 $ETr :: VI$

value

400 $calc: ci:CI \times (vis:VI\text{-}set \times gis:GI\text{-}set) \rightarrow RLF \rightarrow TRM \rightarrow TRA$

385,386 $\text{in } \{v_c_ch[ci,vi] \mid vi:VI \bullet vi \in vis\},$

385,386 $\{g_c_ch[ci,gi] \mid gi:GI \bullet gi \in gis\},$

401,402 $attr_cmd_ch, attr_vdu_ch \text{ Unit}$

384 $calc(ci,(vis,gis))(rlf)(trm)(tra) \equiv$

385 $\text{react_to_vehicles}(ci,(vis,gis),)(rlf)(trm)(tra)$

386 $\square \text{react_to_gates}(ci,(vis,gis))(rlf)(trm)(tra)$

401 $\square \text{react_to_trace_cmd}(ci,(vis,gis))(rlf)(trm)(tra)$

384 $\text{pre } ci = ci_{\mathcal{E}} \wedge vis = vis_{\mathcal{E}} \wedge gis = gis_{\mathcal{E}}$

402 $\text{axiom } \square attr_vdu_ch[ci]? = trm|tra$

The 401,402 $attr_cmd_ch, attr_vdu_ch$ of the *calculator* signature models the *calculator*’s external *command* and *visual display unit* attributes.

403 The *react_to_trace_cmd* alternative behaviour is either a “Begin” or an “End” request which identifies the affected vehicle.

404 If it is a “Begin” request

405 and the identified vehicle is already being traced then we do not prescribe what to do !

406 Else we resume the calculator behaviour, now recording that vehicle as being traced.

407 If it is an “End” request

408 and the identified vehicle is already being traced then we do not prescribe what to do !

409 Else we resume the calculator behaviour, now recording that vehicle as no longer being traced.

¹⁹ VDU: visual display unit


```

403 react_to_trace_cmd(ci,(vis,gis))(vplf)(trm)(tra) ≡
403   case attr_cmd_ch[ci]? of
404,405,406   mkBTr(vi) → if vi ∈ tra then chaos else calc(ci,(vis,gis))(vplf)(trm)(tra ∪ {vi}) end
407,408,409   mkETr(vi) → if vi ∉ tra then chaos else calc(ci,(vis,gis))(vplf)(trm)(tra \ {vi}) end
403   end

```

The above behaviour, Items 384–409, is the one for which we are to design software \square

Example 5.16 exemplifies an action requirement as per definition 43: (a) the action is conceptual, it has no physical counterpart in the domain; (b) it calculates (402) a visual display (vdu); (c) the vdu value is based on a conceptual notion of traffic road maps (trm), an element of the calculator state; (d) the calculation is triggered by an actor (attr_cmd_ch).

Derived Events

Definition 44 Derived Event: By a **derived event** we shall understand (a) a conceptual event, (b) that calculates a property or some non-Boolean value (c) from a machine behaviour state change \odot

Example 5.17. . Domain Requirements. Derived Event: Current Maximum Flow: The example is based on the *Road Pricing Calculator Behaviour* of Examples 5.16 and 5.15 on Page 169. By “the current maximum flow” we understand a time-stamped natural number, the number representing the highest number of vehicles which at the time-stamped moment cruised or now cruises around the toll-road net. We augment the definition of the *calculator* definition Items 384–409, Pages 169–172.

```

410 We augment the calculator signature with
411 a time-stamped natural number valued dynamic programmable attribute, (t:ℕ,max:Max).
412 Whenever a vehicle enters the toll-road net, through one of its [entry] gates,
    a it is checked whether the resulting number of vehicles recorded in the road traffic map is higher than
      the hitherto maximum recorded number.
    b If so, that programmable attribute has its number element “upped” by one.
    c Otherwise not.
413 No changes are to be made to the react_to_gates behaviour (Items 386–397 Page 170) when a vehicle exits
    the toll-road net.

type
411   MAX = ℕ × NAT
value
400,410   calc: ci:CI × (vis:VI-set × gis:GI-set) → RLF → TRM → TRA → MAX
385,386   in {v_c_ch[ci,vi]|vi:VI•vi ∈ vis}, {g_c_ch[ci,gi]|gi:GI•gi ∈ gis}, attr_cmd_ch,attr_vdu_ch Unit
386   react_to_gates(ci,(vis,gis))(vplf)(trm)(tra)(t,m) ≡
386   let (ee,(τ,(vi,li))) = [] {g_c_ch[ci,gi]|gi:GI•gi ∈ gis} in
392   case ee of
412     "Enter" →
412       calc(ci,(vis,gis))(vplf)(trm ∪ [vi → ((τ,SonL(li)))])(tra)(τ,if card dom trm=m then m+1 else m end),
413     "Exit" →
413       billing(vi,trm(vi)^(τ,SonL(li))); calc(ci,(vis,gis))(vplf)(trm \ {vi})(tra)(t,m) end
392   end

```

The above behaviour, Items 384 on Page 169 through 412c, is the one for which we are to design software \square

Example 5.17 exemplifies a derived event requirement as per Definition 44: (a) the event is conceptual, it has no physical counterpart in the domain; (b) it calculates (412b) the max value based on a conceptual notion of traffic road maps (trm), (c) which is an element of the calculator state.

No Derived Behaviours

There are no derived behaviours. The reason is as follows. Behaviours are associated with parts. A possibly ‘derived behaviour’ would entail the introduction of an ‘associated’ part. And if such a part made sense it should – in all likelihood – already have been either a proper domain part or become a domain extension. If the domain-to-requirements engineer insist on modeling some interface requirements as a process then we consider that a technical matter, a choice of abstraction.

5.5.3 Discussion

Derived Requirements

Formulation of derived actions or derived events usually involves technical terms not only from the domain but typically from such conceptual ‘domains’ as mathematics, economics, engineering or their visualisation. Derived requirements may, for some requirements developments, constitute “sizable” requirements compared to “all the other” requirements. For their analysis and prescription it makes good sense to first having developed “the other” requirements: domain, interface and machine requirements. The treatment of the present paper does not offer special techniques and tools for the conception, &c., of derived requirements. Instead we refer to the seminal works of [79, 125, 177].

Introspective Requirements

Humans, including human users are, in this paper, considered to never be part of the domain for which a requirements prescription is being developed. If it is necessary to involve humans in the domain description or the requirements prescription then their prescription is to reflect assumptions upon whose behaviour the machine rely. It is therefore that we, above, have stated, in passing, that we cannot accept requirements of the kind: “*the machine must be user friendly*”, because, in reality, it means “*the user must rely upon the machine being ‘friendly’*” whatever that may mean. We are not requirements prescribing humans, nor their sentiments !

5.6 Machine Requirements

Other than listing a sizable number of machine requirement facets we shall not cover machine requirements in this paper. The reason for this is as follows. We find, cf. [26, Sect. 19.6], that when the individual machine requirements are expressed then references to domain phenomena are, in fact, abstract references, that is, they do not refer to the semantics of what they name. Hence machine requirements “fall” outside the scope of this paper — with that scope being “*derivation*” of requirements from domain specifications with emphasis on derivation techniques that relate to various aspects of the domain.

(A) There are the technology requirements of (1) performance and (2) dependability. Within dependability requirements there are (a) accessibility, (b) availability, (c) integrity, (d) reliability, (e) safety, (f) security and (g) robustness requirements. A proper treatment of dependability requirements need a careful definition of such terms as *failure*, *error*, *fault*, and, from these *dependability*. (B) And there are the development requirements of (i) process, (ii) maintenance, (iii) platform, (iv) management and (v) documentation requirements. Within maintenance requirements there are (ii.1) adaptive, (ii.2) corrective, (ii.3) perfective, (ii.4) preventive, and (ii.5) extensional requirements. Within platform requirements there are (iii.1) development, (iii.2) execution, (iii.3) maintenance, and (iii.4) demonstration platform requirements. We refer to [26, Sect. 19.6] for an early treatment of machine requirements.

5.7 Conclusion

Conventional requirements engineering considers the domain only rather implicitly. Requirements gathering (‘acquisition’) is not structured by any pre-existing knowledge of the domain, instead it is “structured” by a number of relevant techniques and tools [116, 177, 117] which, when applied, “fragment-by-fragment” “discovers” such elements of the domain that are immediately relevant to the requirements. The present paper turns this requirements prescription process “up-side-down”. Now the process is guided (“steered”, “controlled”) almost exclusively by the domain description which is assumed to be existing before the requirements development starts. In conventional requirements engineering many of the relevant techniques and tools can be said to take into account *sociological* and *psychological* facets of gathering the requirements and *linguistic* facets of expressing these requirements. That is, the focus is rather much on the *process*. In the present paper’s requirements “derivation” from domain descriptions the focus is all the time

on the descriptions and prescriptions, in particular on their formal expressions and the “transformation” of these. That is (descriptions and) prescriptions are considered formal, *mathematical* objects. That is, the focus is rather much on the *objects*.

• • •

We conclude by briefly reviewing what has been achieved, present shortcomings & possible research challenges, and a few words on relations to “classical requirements engineering”.

5.7.1 What has been Achieved ?

We have shown how to systematically “derive” initial aspects of requirements prescriptions from domain descriptions. The stages²⁰ and steps²¹ of this “derivation”²² are new. We claim that current requirements engineering approaches, although they may refer to a or the ‘domain’, are not really ‘serious’ about this: they do not describe the domain, and they do not base their techniques and tools on a reasoned understanding of the domain. In contrast we have identified, we claim, a logically motivated decomposition of requirements into three phases, cf. Footnote 20., of domain requirements into five steps, cf. Footnote 21., and of interface requirements, based on a concept of shared entities, tentatively into (α) shared endurants, (β) shared actions, (γ) shared events, and (δ) shared behaviours (with more research into the (α - δ) techniques needed).

5.7.2 Present Shortcomings and Research Challenges

We see three shortcomings: (1) The “derivation” techniques have yet to consider “extracting” requirements from domain facet descriptions. Only by including domain facet descriptions can we, in “deriving” requirements prescriptions, include failures of, for example, support technologies and humans, in the design of fault-tolerant software. (2) The “derivation” principles, techniques and tools should be given a formal treatment. (3) There is a serious need for relating the approach of the present paper to that of the seminal text book of [177, Axel van Lamsweerde]. [177] is not being “replaced” by the present work. It tackles a different set of problems. We refer to the penultimate paragraph before the **Acknowledgment** closing.

5.7.3 Comparison to “Classical” Requirements Engineering:

Except for a few, represented by two, we are not going to compare the contributions of the present paper with published journal or conference papers on the subject of requirements engineering. The reason for this is the following. The present paper, rather completely, we claim, reformulates requirements engineering, giving it a ‘foundation’, in domain engineering, and then developing requirements engineering from there, viewing requirements prescriptions as “derived” from domain descriptions. We do not see any of the papers, except those reviewed below [120] and [79], referring in any technical sense to ‘domains’ such as we understand them.

[120, Deriving Specifications for Systems That Are Connected to the Physical World]

The paper that comes closest to the present paper in its serious treatment of the [problem] domain as a precursor for requirements development is that of [120, Jones, Hayes & Jackson]. A purpose of [120] (Sect. 1.1, Page 367, last §) is to see “how little can one say” (about the problem domain) when expressing assumptions about requirements. This is seen by [120] (earlier in the same paragraph) as in contrast to our form of domain modeling. [120] reveals assumptions about the domain when expressing rely guarantees in tight conjunction with expressing the guarantee (requirements). That is, analysing and expressing requirements, in [120], goes hand-in-hand with analysing and expressing fragments of the domain. The current paper takes the view that since, as demonstrated in [49], it is possible to model sizable aspects of domains,

²⁰ (a) domain, (b) interface and (c) machine requirements

²¹ For domain requirements: (i) projection, (ii) instantiation, (iii) determination, (iv) extension and (v) fitting; etc.

²² We use double quotation marks: “...” to indicate that the derivation is not automatable.

then it would be interesting to study how one might “derive” — and which — requirements prescriptions from domain descriptions; and having demonstrated that (i.e., the “how much can be derived”) it seems of scientific interest to see how that new start (i.e., starting with a priori given domain descriptions or starting with first developing domain descriptions) can be combined with existing approaches, such as [120]. We do appreciate the “tight coupling” of rely–guarantees of [120]. But perhaps one loses understanding the domain due to its fragmented presentation. If the ‘relies’ are not outright, i.e., textually directly expressed in our domain descriptions, then they obviously must be provable properties of what our domain descriptions express. Our, i.e., the present, paper — with its background in [49, Sect. 4.7] — develops — with a background in [115, M.A. Jackson] — a set of principles and techniques for the access of attributes. The “discovery” of the CM and SG channels of [120] and of the type of their messages, seems, compared to our approach, less systematic. Also, it is not clear how the [120] case study “scales” up to a larger domain. The *sluice gate* of [120] is but part of a large (‘irrigation’) system of reservoirs (water sources), canals, sluice gates and the fields (water sinks) to be irrigated. We obviously would delineate such a larger system and research & develop an appropriate, both informal, a narrative, and formal domain description for such a class of irrigation systems based on assumptions of precipitation and evaporation. Then the users’ requirements, in [120], that the sluice gate, over suitable time intervals, is open 20% of the time and otherwise closed, could now be expressed more pertinently, in terms of the fields being appropriately irrigated.

[79, Goal-directed Requirements Acquisition]

outlines an approach to requirements acquisition that starts with fragments of domain description. The domain description is captured in terms of predicates over *actors*, *actions*, *events*, *entities* and (their) *relations*. Our approach to domain modeling differs from that of [79] as follows: Agents, actions, entities and relations are, in [79], seen as specialisations of a concept of *objects*. The nearest analogy to relations, in [49], as well as in this paper, is the signatures of perdurants. Our ‘agents’ relate to discrete endurants, i.e., parts, and are the behaviours that evolve around these parts: one agent per part! [79] otherwise include describing parts, relations between parts, actions and events much like [49] and this paper does. [79] then introduces a notion of goal. A **goal**, in [79], is defined as “a nonoperational objective to be achieved by the desired system. Nonoperational means that the objective is not formulated in terms of objects and actions “available” to some agent of the system”²³ [79] then goes on to exemplify goals. In this, the current paper, we are not considering goals, also a major theme of [177].²⁴ Typically the expression of goals of [79, 177], are “within” computer & computing science and involve the use of temporal logic.²⁵ “Constraints are operational objectives to be achieved by the desired (i.e., required) system, . . . , formulated in terms of objects and actions “available” to some agents of the system. . . . Goals are made operational through constraints. . . . A constraint operationalising a goal amounts to some abstract “implementation” of this goal” [79]. [79] then goes on to express goals and constraints operationalising these. [79] is a fascinating paper²⁶ as it shows how to build goals and constraints on domain description fragments.

• • •

These papers, [120] and [79], as well as the current paper, together with such seminal monographs as [187, 143, 177], clearly shows that there are many diverse ways in which to achieve precise requirements

²³ We have reservations about this definition: Firstly, it is expressed in terms of some of the “things” it is not! (To us, not a very useful approach.) Secondly, we can imagine goals that are indeed formulated in terms of objects and actions ‘available’ to some agent of the system. For example, wrt. the ongoing library examples of [79], *the system shall automate the borrowing of books*, etcetera. Thirdly, we assume that by “‘available’ to some agent of the system” is meant that these agents, actions, entities, etc., are also required.

²⁴ An example of a goal — for the road pricing system — could be that of *shortening travel times of motorists, reducing gasoline consumption and air pollution, while recouping investments on toll-road construction*. We consider techniques for ensuring the above kind of goals “outside” the realm of computer & computing science but “inside” the realm of operations research (OR) — while securing that the OR models are commensurate with our domain models.

²⁵ In this paper we do not exemplify goals, let alone the use of temporal logic. We cannot exemplify all aspects of domain description and requirements prescription, but, if we were, would then use the temporal logic of [187, The Duration Calculus].

²⁶ — that might, however, warrant a complete rewrite.

prescriptions. The [187, 143] monographs primarily study the $\mathcal{D}, \mathcal{S} \models \mathcal{R}$ specification and proof techniques from the point of view of the specific tools of their specification languages²⁷. Physics, as a natural science, and its many engineering ‘renditions’, are manifested in many separate sub-fields: Electricity, mechanics, statics, fluid dynamics — each with further sub-fields. It seems, to this author, that there is a need to study the [187, 143, 177] approaches and the approach taken in this paper in the light of identifying sub-fields of requirements engineering. The title of the present paper suggests one such sub-field.

²⁷ The Duration Calculus [DC], respectively DC, Timed Automata and Z

Software

Domains: Their Simulation, Monitoring and Control

A Divertimento of Ideas and Suggestions

Model Driven Software Engineering and Software Product Lines

Summary

We sketch¹ some observations of the concepts of domain, requirements and modeling – where abstract interpretations of these models cover both a priori, a posteriori and real-time aspects of the domain as well as 1–1 (i.e., real-time), microscopic and macroscopic simulations, real-time monitoring and real-time monitoring & control of that domain. The reference frame for these concepts are domain models: carefully narrated and formally described domains. We survey more-or-less standard ideas of verifiable software developments and conjecture software product families of demos, simulators, monitors and monitors & controllers – but now these “standard ideas” are recast in the context of core requirements prescriptions being “derived” from domain descriptions.

6.1 Introduction

A background setting for this chapter is the concern for (α) professionally developing the right software, i.e., software which satisfies users expectations, and (ω) software that is right: i.e., software which is correct with respect to user requirements and thus has no “bugs”, no “blue screens”. The present chapter must be seen on the background of a main line of experimental research around the topics of domain science & engineering and requirements engineering and their relation. For details I refer to Chaps. 1–5 [49, 53, 52, 54].

“Confusing Demos”: This author has had the doubtful honour, on his many visits to computer science and software engineering laboratories around the world, to be presented, by his colleagues’ aspiring PhD students, so-called demos of “systems” that they were investigating. There always was a tacit assumption, namely that the audience, i.e., me, knew, a priori, what the domain “behind” the “system” being “demo’ed” was. Certainly, if there was such an understanding, it was brutally demolished by the “demo” presentation. My questions, such as “*what are you demo’ing*” (etcetera) went unanswered. Instead, while we were waiting to see “something interesting” to be displayed on the computer screen one was witnessing frantic, sometimes failed, input of commands and data, “nervous” attempts with “mouse” clickings, etc. – before something intended was displayed. After a, usually 15 minute, grace period, it was time, luckily, to proceed to the next “demo”!

Aims & Objectives: The aims of this chapter is to present (a) some ideas about software that either “demo”, simulate, monitor or monitor & control domains; (b) some ideas about “time scaling”: demo and simulation time versus domain time; and (c) how these kinds of software relate. The (undoubtedly very naïve) objectives of the chapter is also to improve the kind of demo-presentations, alluded to above, so as

¹ This chapter is based on [48].

to ensure that the basis for such demos is crystal clear from the very outset of research & development, i.e., that domains be well-described. The chapter, we think, tackles the issue of so-called ‘model-oriented (or model-based) software development’ from altogether different angles than usually promoted.

An Exploratory Chapter: The chapter is exploratory. There will be no theorems and therefore there will be no proofs. We are presenting what might eventually emerge into (α) a theory of domains, i.e., a domain science [28, 59, 37, 43], and (β) a software development theory of domain engineering versus requirements engineering [36, 29, 30, 34].

The chapter is not a “standard research paper”: it does not compare its claimed achievements with corresponding or related achievements of other researchers – simply because we do not claim “achievements” which have been reasonably well formalised. But we would suggest that you might find some of the ideas of the chapter (in Sect. 6.3) worthwhile. Hence the “divertimento” suffix to the chapter title.

Structure of Chapter: The structure of the chapter is as follows. In Sect. 6.2 we list a number of domain descriptions. In Sect. 6.3 we then outline a series of interpretations of domain descriptions. These arise, when developed in an orderly, professional manner, from requirements prescriptions which are themselves orderly developed from the domain description², cf. [54]. The essence of Sect. 6.3 is (i) the (albeit informal) presentation of such tightly related notions as *demos* (Sect. 6.3.1), *simulators* (Sect. 6.3.2), *monitors* (Sect. 6.3.3) and *monitors & controllers* (Sect. 6.3.3) (these notions can be formalised), and (ii) the conjectures on a product family of domain-based software developments (Sect. 6.3.5). A notion of *script-based simulation* extends demos and is the basis for monitor and controller developments and uses. The scripts used in our examples are related to time, but one can define non-temporal scripts – so the “carrying idea” of Sect. 6.3 extends to a widest variety of software. We claim that Sect. 6.3 thus brings these new ideas: a tightly related software engineering concept of *demo-simulator-monitor-controller* machines, and an extended notion of *reference models for requirements and specifications* [100].

6.2 Domain Descriptions

By a domain description we shall mean a combined narrative, that is, precise, but informal, and a formal description of the application domain **as it is**: no reference to any possible requirements let alone software that is desired for that domain. Thus a requirements prescription is a likewise combined precise, but informal, narrative, and a formal prescription of what we expect from a machine (hardware + software) that is to support endurants, actions, events and behaviours of a possibly business process re-engineered application domain. Requirements expresses a domain **as we would like to be**. We present two example domain descriptions in Part V. We further refer to the published literature for examples: [21, *railways* (2000)], [22, *the ‘market’* (2000)], [30, *public government, IT security, hospitals* (2006) chapters 8–10], [29, *transport nets* (2008)] [34, *pipelines* (2010)]. On the net you may find technical reports covering “larger” domain descriptions. “Older” publications on the concept of domain descriptions are [34, 37, 31, 59, 29, 28, 33] all summarised in Chaps. 1–2 and 5 [49, 53, 54]. Domain descriptions do not necessarily describe computable objects. They relate to the described domain in a way similar to the way in which mathematical descriptions of physical phenomena stand to “the physical world”.

6.3 Interpretations

In this main section of this chapter we present a number of interpretations of rôles of domain descriptions.

6.3.1 What Is a Domain-based Demo?

A *domain-based demo* is a software system which “*present*” endurants and perdurants³: actions, events and behaviours of a domain. The “*presentation*” abstracts these phenomena and their related concepts in various computer generated forms: visual, acoustic, etc.

² We do not show such orderly “derivations” but outline their basics in Sect. 6.3.4.

³ The concepts of ‘endurants’ and ‘perdurants’ were defined in [49].

Examples: There are three main examples. Two are given in Appendix A and Appendix B. The other is summarised below. It is from Chap. 5 on “deriving requirements prescriptions from domain descriptions” [54]. The summary follows. The domain description of Sect. 5.2 of Chap. 5 [54], outlines an abstract concept of transport nets (of hubs [street intersections, train stations, harbours, airports] and links [road segments, rail tracks, shipping lanes, air-lanes]), their development, traffic [of vehicles, trains, ships and aircraft], etc. We shall assume such a transport domain description below. Endurants are, for example, presented as follows: (a) transport nets by two dimensional (2D) road, railway or air traffic maps, (b) hubs and links by highlighting parts of 2D maps and by related photos – and their unique identifiers by labeling hubs and links, (c) routes by highlighting sequences of paths (hubs and links) on a 2D map, (d) buses by photographs and by dots at hubs or on links of a 2D map, and (e) bus timetables by, well, indeed, by showing a 2D bus timetable. Actions are, for example, presented as follows: (f) The insertion or removal of a hub or a link by showing “instantaneous” triplets of “before”, “during” and “after” animation sequences. (g) The start or end of a bus ride by showing flashing animations of the appearance, respectively the flashing disappearance of a bus (dot) at the origin, respectively the destination bus stops. Events are, for example, presented as follows: (h) A mudslide [or fire in a road tunnel, or collapse of a bridge] along a (road) link by showing an animation of part of a (road) map with an instantaneous sequence of (α) the present link, (β) a gap somewhere on the link, (γ) and the appearance of two (“symbolic”) hubs “on either side of the gap”. (i) The congestion of road traffic “grinding to a halt” at, for example, a hub, by showing an animation of part of a (road) map with an instantaneous sequence of the massive accumulation of vehicle dots moving (instantaneously) from two or more links into a hub. Behaviours are, for example, presented as follows: (k) A bus tour: from its start, on time, or “thereabouts”, from its bus stop of origin, via (all) intermediate stops, with or without delays or advances in times of arrivals and departures, to the bus stop of destination (ℓ) The composite behaviour of “all bus tours”, meeting or missing connection times, with sporadic delays, with cancellation of some bus tours, etc. – by showing the sequence of states of all the buses on the net. We say that behaviours ((j)–(ℓ)) are *script-based* in that they (try to) satisfy a bus timetable ((e)).

Towards a Theory of Visualisation and Acoustic Manifestation: The above examples shall serve to highlight the general problem of visualisation and acoustic manifestation. Just as we need sciences of visualising scientific data and of diagrammatic logics, so we need more serious studies of visualisation and acoustic manifestation — so amply, but, this author thinks, inconsistently demonstrated by current uses of interactive computing media.

6.3.2 Simulations

“Simulation is the imitation of some real thing, state of affairs, or process; the act of simulating something generally entails representing certain key characteristics or behaviours of a selected physical or abstract system” [Wikipedia] for the purposes of testing some hypotheses usually stated in terms of the model being simulated and pairs of statistical data and expected outcomes.

Explication of Figure 6.1: Figure 6.1 on the following page attempts to indicate four things: (i) Left top: the rounded edge rectangle labeled “The Domain” alludes to some specific domain (“out there”). (ii) Left middle: the small rounded rectangle labeled “A Domain Description” alludes to some document which narrates and formalises a description of “the domain”. (iii) Left bottom: the medium sized rectangle labeled “Domain Demo based on the Domain Description” (for short “Demo”) alludes to a software system that, in some sense (to be made clear later) “simulates” “The Domain.” (iv) Right: the large rectangle (a) shows a horizontal time axis which basically “divides” that large rectangle into two parts: (b) Above the time axis the “**fat**” rounded edge rectangle alludes to the time-wise behaviour, a *domain trace*, of “The Domain” (i.e., the actual, the real, domain). (c) Below the time axis there are eight “**thin**” rectangles. These are labels S1, S2, S3, S4, S5, S6, S7 and S8. Each of these denote a “run”, i.e., a time-stamped “execution”, a *program trace*, of the “Demo”. Their “relationship” to the time axis is this: their execution takes place in the real time as related to that of “The Domain” behaviour. A *trace* (whether a domain or a program execution trace) is a time-stamped sequence of states: domain states, respectively demo, simulator, monitor and monitor & control states.

From Fig. 6.1 on the next page and the above explication we can conclude that “executions” S4 and S5 each share exactly one time point, t , at which “The Domain” and “The Simulation” “share” time, that

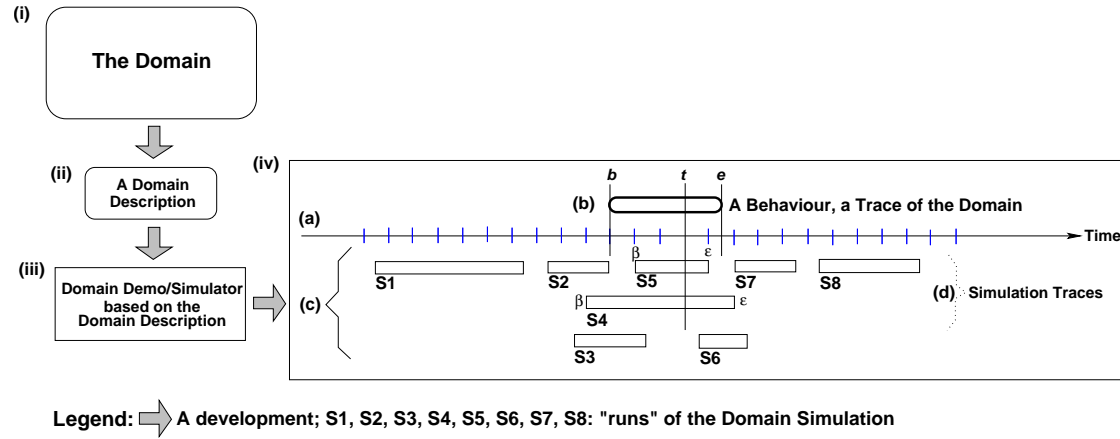


Fig. 6.1. Simulations

is, the time-stamped execution S4 and S5 reflect a "Simulation" state which at time t should reflect (some abstraction of) "The Domain" state. Only if the domain behaviour (i.e., trace) fully "surrounds" that of the simulation trace, or, vice-versa (cf. Fig. 6.1[S4,S5]), is there a "shared" time. Only if the 'begin' and 'end' times of the domain behaviour are identical to the 'start' and 'finish' times of the simulation trace, is there an infinity of shared 1–1 times. Only then do we speak of a real-time simulation. In Fig. 6.2 on the facing page we show "the same" "Domain Behaviour" (three times) and a (1) simulation, a (2) monitoring and a (3) monitoring & control, all of whose 'begin/start' (b/β) and 'end/finish' (e/ϵ) times coincide. In such cases the "Demo/Simulation" takes place in real-time throughout the 'begin...end' interval. Let β and ϵ be the 'start' and 'finish' times of either S4 or S5. Then the relationship between t, β, ϵ, b and e is $\frac{t-b}{e-t} = \frac{t-\beta}{\epsilon-t}$ — which leads to a second degree polynomial in t which can then be solved in the usual, high school manner.

Script-based Simulation: A script-based simulation is the behaviour, i.e., an execution, of, basically, a demo which, step-by-step, follows a script: that is a prescription for highlighting endurants, actions, events and behaviours. Script-based simulations where the script embodies a notion of time, like a bus timetable, and unlike a route, can be thought of as the execution of a demos where "chunks" of demo operations take place in accordance with "chunks"⁴ of script prescriptions. The latter (i.e., the script prescriptions) can be said to represent simulated (i.e., domain) time in contrast to "actual computer" time. The actual times in which the script-based simulation takes place relate to domain times as shown in Simulations S1 to S8 in Fig. 6.1 and in Fig. 6.2(1–3). Traces Fig. 6.2(1–3) and S8 Fig. 6.1 are said to be *real-time*: there is a one-to-one mapping between computer time and domain time. S1 and S4 Fig. 6.1 are said to be *microscopic*: disjoint computer time intervals map into distinct domain times. S2, S3, S5, S6 and S7 are said to be *macroscopic*: disjoint domain time intervals map into distinct computer times. In order to concretise the above "vague" statements let us take the example of simulating bus traffic as based on a bus timetable script. A simulation scenario could be as follows. Initially, not relating to any domain time, the simulation "demos" a net, available buses and a bus timetable. The person(s) who are requesting the simulation are asked to decide on the ratio of the domain time interval to simulation time interval. If the ratio is 1 a real-time simulation has been requested. If the ratio is less than 1 a microscopic simulation has been requested. If the ratio is larger than 1 a macroscopic simulation has been requested. A chosen ratio of, say 48 to 1 means that a 24 hour bus traffic is to be simulated in 30 minutes of elapsed simulation time. Then the person(s) who are requesting the simulation are asked to decide on the starting domain time, say 6:00am, and the domain time interval of simulation, say 4 hours – in which case the simulation of bus traffic from 6am till 10am is to be shown in 5 minutes (300 seconds) of elapsed simulation time. The person(s) who are requesting the simulation are then asked to decide on the "sampling times" or "time intervals": If 'sampling times' 6:00 am, 6:30 am, 7:00 am, 8:00 am, 9:00 am, 9:30 am and 10:00 am are chosen, then the simulation

⁴ We deliberately leave the notion of chunk vague so as to allow as wide an spectrum of simulations.

is stopped at corresponding simulation times: 0 sec., 37.5 sec., 75 sec., 150 sec., 225 sec., 262.5 sec. and 300 sec. The simulation then shows the state of selected endurants and actions at these domain times. If ‘*sampling time interval*’ is chosen and is set to every 5 min., then the simulation shows the state of selected endurants and actions at corresponding domain times. The simulation is resumed when the person(s) who are requesting the simulation so indicates, say by a “resume” icon click. The time interval between adjacent simulation stops and resumptions contribute with 0 time to elapsed simulation time – which in this case was set to 5 minutes. Finally the requestor provides some statistical data such as numbers of potential and actual bus passengers, etc. Then two clocks are started: a domain time clock and a simulation time clock. The simulation proceeds as driven by, in this case, the bus time table. To include “unforeseen” events, such as the wreckage of a bus (which is then unable to complete a bus tour), we allow any number of such events to be randomly scheduled. Actually scheduled events “interrupts” the “programmed” simulation and leads to thus unscheduled stops (and resumptions) where the unscheduled stop now focuses on showing the event.

The Development Arrow: The arrow, \Rightarrow , between a pair of boxes (of Fig. 6.1 on the preceding page) denote a step of development: (i) from the domain box to the domain description box, \Downarrow , it denotes the development of a domain description based on studies and analyses of the domain; (ii) from the domain description box to the domain demo box, \Downarrow , it denotes the development of a software system — where that development assumes an intermediate requirements box which has not been show; (iii) from the domain demo box to either of a simulation traces, \Rightarrow , it denotes the development of a simulator as the related demo software system, again depending on whichever special requirements have been put to the simulator.

6.3.3 Monitoring & Control

Figure 6.2 shows three different kinds of uses of software systems (where (2) [Monitoring] and (3) [Monitoring & Control] represent further) developments from the demo or simulation software system mentioned in Sect. 6.3.1 and Sect. 6.3.2 on the preceding page. We have added some (three) horizontal

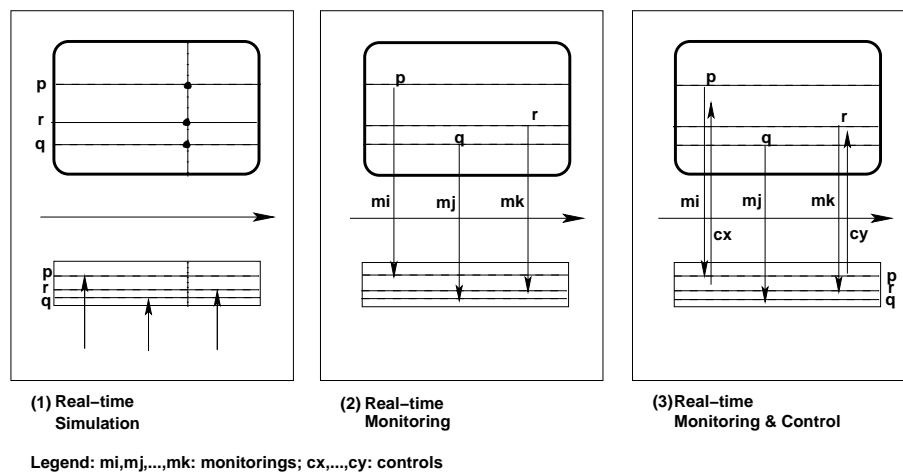


Fig. 6.2. Simulation, Monitoring and Monitoring & Control

and labeled (p, q and r) lines to Fig. 6.2(1,2,3) (with respect to the traces of Fig. 6.1 on the preceding page). They each denote a trace of a endurant, an action or an event, that is, they are traces of values of these phenomena or concepts. A (named) endurant value entails a description of the endurant, whither atomic (‘hub’, ‘link’, ‘bus timetable’) or composite (‘net’, ‘set of hubs’, etc.): of its unique identity, its mereology and a selection of its attributes. A (named) action value could, for example, be the pair of the before and after states of the action and some description of the function (‘insertion of a link’, ‘start of a bus tour’) involved in the action. A (named) event value could, for example, be a pair of the before and

after states of the endurants causing, respectively being effected by the event and some description of the predicate ('mudslide', 'break-down of a bus') involved in the event. A cross section, such as designated by the vertical lines (one for the domain trace, one for the "corresponding" program trace) of Fig. 6.2 on the preceding page(1) denotes a state: a domain, respectively a program state. Figure 6.2(1) attempts to show a real-time demo or simulation for the chosen domain. Figure 6.2(2) purports to show the deployment of real-time software for monitoring (chosen aspects of) the chosen domain. Figure 6.2(3) purports to show the deployment of real-time software for monitoring as well as controlling (chosen aspects of) the chosen domain.

Monitoring: By *domain monitoring* we mean "to be aware of the state of a domain", its endurants, actions, events and behaviour. Domain monitoring is thus a process, typically within a distributed system for collecting and storing state data. In this process "observation" points — i.e., endurants, actions and where events may occur — are identified in the domain, cf. points p, q and r of Fig. 6.2. Sensors are inserted at these points. The "downward" pointing vertical arrows of Figs. 6.2(2–3), from "the domain behaviour" to the "monitoring" and the "monitoring & control" traces express communication of what has been sensed (measured, photographed, etc.) [as directed by and] as input data (etc.) to these monitors. The monitor (being "executed") may store these "sensings" for future analysis.

Control: By *domain control* we mean "the ability to change the value" of endurants and the course of actions and hence behaviours, including prevention of events of the domain. Domain control is thus based on domain monitoring. Actuators are inserted in the domain "at or near" monitoring points or at points related to these, viz. points p and r of Fig. 6.2 on the previous page(3). The "upward" pointing vertical arrows of Fig. 6.2 on the preceding page(3), from the "monitoring & control" traces to the "domain behaviour" express communication, to the domain, of what has been computed by the controller as a proper control reaction in response to the monitoring.

6.3.4 Machine Development

Machines: By a *machine* we shall understand a combination of hardware and software. For demos and simulators the machine is "mostly" software with the hardware typically being graphic display units with tactile instruments. For monitors the "main" machine, besides the hardware and software of demos and simulators, additionally includes *sensors* distributed throughout the domain and the technological machine means of *communicating* monitored signals from the sensors to the "main" machine and the processing of these signals by the main machine. For monitors & controllers the machine, besides the monitor machine, further includes actuators placed in the domain and the machine means of computing and communicating control signals to the actuators.

Requirements Development: Essential parts of Requirements to a Machine can be systematically "derived" from a Domain description. These essential parts are the *domain requirements* and the *interface requirements*. Domain requirements are those requirements which can be expressed, say in narrative form, by mentioning technical terms only of the domain. These technical terms cover only phenomena and concepts (endurants, actions, events and behaviours) of the domain. Some domain requirements are *projected*, *instantiated*, made more *deterministic* and *extended*⁵. We bring examples that are taken from Sect. 2. of [54], cf. Sect. 6.3.1 on Page 181 of this chapter. (a) By *domain projection* we mean a sub-setting of the domain description: parts are left out which the requirements stake-holders, collaborating with the requirements engineer, decide is of no relevance to the requirements. For our example it could be that our domain description had contained models of road net attributes such as "the wear & tear" of road surfaces, the length of links, states of hubs and links (that is, [dis]allowable directions of traffic through hubs and along links), etc. Projection might then omit these attributes. (b) By *domain instantiation* we mean a specialisation of endurants, actions, events and behaviours, refining them from abstract simple entities to more concrete such, etc. For our example it could be that we only model freeways or only model road-pricing nets – or any one or more other aspects. (c) By *domain determination* we mean that of making the domain description cum domain requirements prescription less non-deterministic, i.e., more deterministic (or even the other way around!). For our example it could be that we had domain-described states of street intersections as

⁵ We omit consideration of *fitting*.

not controlled by traffic signals – where the determination is now that of introducing an abstract notion of traffic signals which allow only certain states (of red, yellow and green). (d) By *domain extension* we basically mean that of extending the domain with phenomena and concepts that were not feasible without information technology. For our examples we could extend the domain with bus mounted GPS gadgets that record and communicate (to, say a central bus traffic computer) the more-or-less exact positions of buses – thereby enabling the observation of bus traffic. Interface requirements are those requirements which can be expressed, say in narrative form, by mentioning technical terms both of the domain and of the machine. These technical terms thus cover shared phenomena and concepts, that is, phenomena and concepts of the domain which are, in some sense, also (to be) represented by the machine. Interface requirements represent (i) the initialisation and “on-the-fly” update of machine endurants on the basis of *shared* domain endurants; (ii) the interaction between the machine and the domain while the machine is carrying out a (previous domain) action; (iii) machine responses, if any, to domain events — or domain responses, if any, to machine events cum “outputs”; and (iv) machine monitoring and machine control of domain phenomena. Each of these four (i–iv) interface requirement facets themselves involve projection, instantiation, determination, extension and fitting. Machine requirements are those requirements which can be expressed, say in narrative form, by mentioning technical terms only of the machine. (An example is: visual display units.)

6.3.5 Verifiable Software Development

An Example Set of Conjectures: We illustrate some conjectures.

- (A) From a domain, \mathbb{D} , one can develop a domain description \mathcal{D} . \mathcal{D} cannot be [formally] verified. It can be [informally] validated “against” \mathcal{D} . Individual properties, $\mathcal{P}_{\mathbb{D}}$, of the domain description \mathcal{D} and hence, purportedly, of the domain, \mathbb{D} , can be expressed and possibly proved $\mathcal{D} \models \mathcal{P}_{\mathbb{D}}$ and these may be validated to be properties of \mathcal{D} by observations in (or of) that domain.
- (B) From a domain description, \mathcal{D} , one can develop domain demo requirements, \mathcal{R}_{DEM} , for, and from \mathcal{R}_{DEM} one can develop a domain demo software specification \mathcal{S}_{DEM} such that $\mathcal{D}, \mathcal{S}_{\text{DEM}} \models \mathcal{R}_{\text{DEM}}$. The formula $\mathcal{D}, \mathcal{S} \models \mathcal{R}$ can be read as follows: in order to prove that the Software satisfies the Requirements, assumptions about the Domain must often be made explicit in steps of the proof.
- (C) From a domain description, \mathcal{D} , and a domain demo requirements prescription, \mathcal{R}_{DEM} , one can develop domain simulator requirements, \mathcal{R}_{SIM} , for, and from such a \mathcal{R}_{SIM} , one can develop a domain simulator software specification \mathcal{S}_{SIM} such that $(\mathbb{D}; \mathcal{R}_{\text{DEM}}), \mathcal{S}_{\text{SIM}} \models \mathcal{R}_{\text{SIM}}$. We have “lumped” $(\mathbb{D}; \mathcal{R}_{\text{DEM}})$ as the two constitute the extended domain for which we, in this case of development, suggest the next stage requirements and software development to take place.
- (D) From a domain description, \mathbb{D} , and a domain simulator requirements prescription, \mathcal{R}_{SIM} , one can develop domain monitor requirements, \mathcal{R}_{MON} , for, and from such a \mathcal{R}_{MON} one can develop, a domain monitor software specification \mathcal{S}_{MON} such that $(\mathbb{D}; \mathcal{R}_{\text{SIM}}), \mathcal{S}_{\text{MON}} \models \mathcal{R}_{\text{MON}}$.
- (E) From a domain description, \mathbb{D} , and a domain monitor requirements specification, \mathcal{R}_{MON} , one can develop domain monitor & controller requirements, $\mathcal{R}_{\text{M\&C}}$, for, and from such a $\mathcal{R}_{\text{M\&C}}$ one can develop, a domain monitor & controller software specification $\mathcal{S}_{\text{M\&C}}$ such that $(\mathbb{D}; \mathcal{R}_{\text{MON}}), \mathcal{S}_{\text{M\&C}} \models \mathcal{R}_{\text{M\&C}}$.

Many other such developments can be diagrammed (cf. Fig. 6.3). The one just illustrated leads to four kinds of strongly related software. The arrow between two boxes, from \mathcal{D} to \mathcal{R} , means the same thing as was illustrated in Chap. 2. Two arrows, from boxes \mathcal{D} to \mathcal{R}' , to a box \mathcal{R}'' means that prescription \mathcal{R}' did not suffice: there were requirements, to be expressed in \mathcal{R}'' , which needed to be “derived” from \mathcal{D} – typically also requiring *fitting*, as outlined in Sect. 5.4.5 on Page 164.

Chains of Verifiable Developments: The above illustrated just one chain (A–E) of developments. There are others. All are shown in Fig. 6.3 on the next page. The figure can also be interpreted as prescribing a subset of a possible range of software products [66, 148] for a given domain. One domain may give rise to many different kinds of DEMO machines, SIMulators, MONitors and Mitor & Controllers For each of these there are similarly, “exponentially” many variants of successor machines.

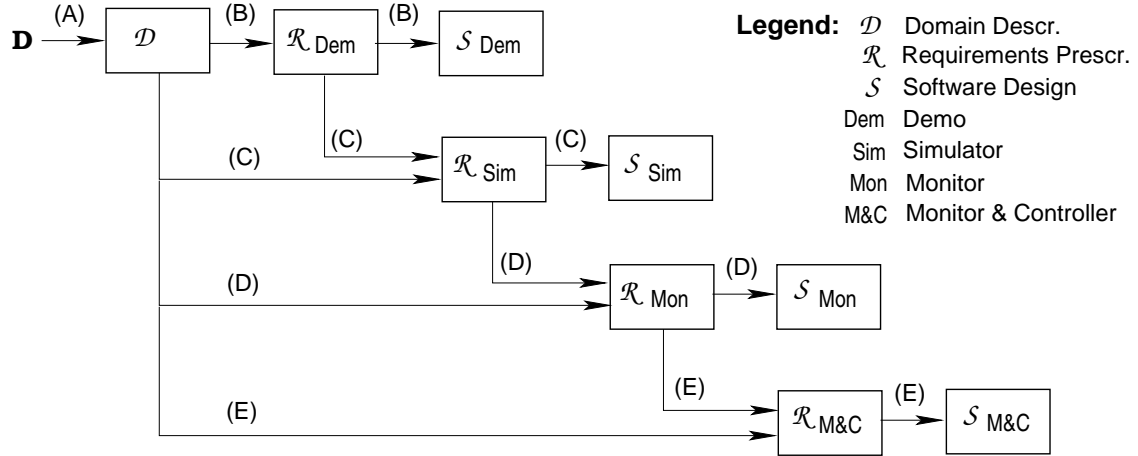


Fig. 6.3. Chains of Verifiable Development

6.4 Conclusion

Our divertimento is almost over. It is time to conclude.

The Correctness Relations: The $\mathbb{D}, \mathbb{M} \models \mathbb{R}$ (‘correctness’ of) development relation appears to have been first indicated in the Computational Logic Inc. Stack [16, 98] and the EU ESPRIT ProCoS [18, 19] projects; [100] presents this same idea with a purpose much like ours, but with more technical discussions.

Domain Engineering: The term ‘domain engineering’ appears to have at least two meanings: the one used here [28, 33] and one [107, 85, 67] emerging out of the Software Engineering Institute at CMU where it is also called *product line engineering*⁶. Our meaning, is, in a sense, more narrow, but then it seems to also be more highly specialised (with detailed description and formalisation principles and techniques). Fig. 6.3 illustrates, in capsule form, what we think is the CMU/SEI meaning.

Model-driven Software Engineering: The relationship between, say Fig. 6.3 and *model-based software development* seems obvious but need be explored. An extensive discussion of the term ‘domain’, as it appears in the software engineering literature is found in [49, Sect. 5.3].

What Have We Achieved: We have characterised a spectrum of strongly domain-related as well as strongly inter-related (cf. Fig. 6.3) software product families: *demos*, *simulators*, *monitors* and *monitor & controllers*. We have indicated varieties of these: simulators based on demos, monitors based on simulators, monitor & controllers based on monitors, in fact any of the latter ones in the software product family list as based on any of the earlier ones. We have sketched temporal relations between simulation traces and domain behaviours: *a priori*, *a posteriori*, *macroscopic* and *microscopic*, and we have identified the real-time cases which lead on to monitors and monitor & controllers.

What Have We Not Achieved — Some Conjectures: We have not characterised the software product family relations other than by the $\mathbb{D}, \mathbb{S} \models \mathbb{R}$ and $(\mathbb{D}; \mathbb{S}_{\text{XYZ}}), \mathbb{S} \models \mathbb{R}$ clauses. That is, we should like to prove conjectured type theoretic inclusion relations like:

$$\wp(\llbracket \mathcal{S}_{\text{mod ext.}} \rrbracket) \supseteq \wp(\llbracket \mathcal{S}'_{\text{mod ext.}} \rrbracket), \quad \wp(\llbracket \mathcal{S}'_{\text{mod ext.}} \rrbracket) \supseteq \wp(\llbracket \mathcal{S}''_{\text{mod ext.}} \rrbracket)$$

where X and Y range appropriately, where $\llbracket \mathcal{S} \rrbracket$ expresses the meaning of \mathcal{S} , where $\wp(\llbracket \mathcal{S} \rrbracket)$ denote the space of all machine meanings and where $\wp(\llbracket \mathcal{S}_{\text{mod ext.}} \rrbracket)$ is intended to denote that space modulo (“free

⁶ http://en.wikipedia.org/wiki/Domain_engineering.

of”) the y facet (here *ext.*, for extension). That is, it is conjectured that the set of more specialised, i.e., n primed, machines of kind x is type theoretically “contained” in the set of m primed (unprimed) x machines ($0 \leq m < n$). There are undoubtedly many such interesting relations between the DEMO, SIMULATOR, MONITOR and MONITOR & CONTROLLER machines, unprimed and primed.

What Should We Do Next: This chapter has the subtitle: *A Divertimento of Ideas and Suggestions*. It is not a proper theoretical chapter. It tries to throw some light on families and varieties of software, i.e., their relations. It focuses, in particular, on so-called DEMO, SIMULATOR, MONITOR and MONITOR & CONTROLLER software and their relation to the “originating” domain, i.e., that in which such software is to serve, and hence that which is being *extended* by such software, cf. the compounded ‘domain’ $(\mathbb{D}; \mathbb{M}_i)$ of in $(\mathbb{D}; \mathbb{M}_i), \mathbb{M}_j \models \mathbb{D}$. These notions should be studied formally. All of these notions: requirements projection, instantiation, determination and extension can be formalised; and the specification language, in the form used here (without CSP processes, [111]) has a formal semantics and a proof system — so the various notions of development, $(\mathbb{D}; \mathbb{M}_i), \mathbb{M}_j \models \mathbb{R}$ and $\wp(\mathbb{M})$ can be formalised.

Closing

Conclusion

7.1 What Have We Achieved ?

We started this volume by claiming, in the Preface (Page v), that

- domain engineering is a viable,
 - ⊗ yes, we would claim,
 - ⊗ necessary initial phase of software development.

We strongly think that the previous chapters have borne out that claim.

We then went on, again in the Preface (Page v), to claim that

- that domain science and engineering
 - ⊗ is a field full of interesting problems
 - ⊗ to be researched.

Again, we strongly think that the previous chapters have borne out also that claim.¹

With this we rest our case !

7.2 Domain Science & Engineering

The title of this compendium is **Domain Science & Engineering**.

- The ‘science’ is covered in Chaps. 3 and 4.
- The ‘engineering’ is covered in remaining chapters.

¹ Several of the individual chapters, towards their end, that is, in Sects. 1.5.4–1.5.5, 2.9.1–2.9.4, 3.9.3, 4.7.3 and 5.7.2, lists some worthwhile research problems.

Bibliography

1. Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings and Modeling in Event-B: System and Software Engineering*. Cambridge University Press, Cambridge, England, 1996 and 2009.
2. Open Mobile Alliance. OMA DRM V2.0 Candidate Enabler. http://www.openmobilealliance.org/-release_program/drm_v2_0.html, Sep 2005.
3. Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, August 2003. ISBN 0521825830.
4. K. Araki et al., editors. *IFM 1999–2013: Integrated Formal Methods*, LNCS Vols. 1945, 2335, 2999, 3771, 4591, 5423, 6496, 7321 and 7940. Springer, 1999–2013.
5. Alapan Arnab and Andrew Hutchison. Fairer Usage Contracts for DRM. In *Proceedings of the Fifth ACM Workshop on Digital Rights Management (DRM'05)*, pages 65–74, Alexandria, Virginia, USA, Nov 2005.
6. Rober Audi. *The Cambridge Dictionary of Philosophy*. Cambridge University Press, The Pitt Building, Trumpington Street, Cambridge CB2 1RP, England, 1995.
7. John Longshaw Austin. *How To Do Things With Words*. Oxford University Press, second edition, 1976.
8. Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, January 2003. 570 pages, 14 tables, 53 figures; ISBN: 0521781760.
9. Franz Baader, Ian Horrocks, and Ulrike Sattler. Description Logics as Ontology Languages for the Semantic Web. In Dieter Hutter and Werner Stephan, editors, *Mechanizing Mathematical Reasoning*, pages 228–248. Springer, Heidelberg, 2005.
10. C. Bachman. Data structure diagrams. *Data Base, Journal of ACM SIGBDP*, 1(2), 1969.
11. Ralph-Johan Back, Abo Akademi, J. von Wright, and F. B. Schneider. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag New York, Inc., 1998.
12. Alain Badiou. *Being and Event*. Continuum, 2005. (L'être et l'événements, Edition du Seuil, 1988).
13. Jaco W. de Bakker. *Control Flow Semantics*. The MIT Press, Cambridge, Mass., USA, 1995.
14. V. Richard Benjamins and Dieter Fensel. The Ontological Engineering Initiative (KA)2. Internet publication + Formal Ontology in Information Systems, University of Amsterdam, SWI, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands and University of Karlsruhe, AIFB, 76128 Karlsruhe, Germany, 1998. <http://www.aifb.uni-karlsruhe.de/WBS/broker/KA2.htm>.
15. Yochai Benkler. Coase's Penguin, or Linux and the Nature of the Firm. *The Yale Law Journal*, 112, 2002.
16. W.R. Bevier, W.A. Hunt Jr., J Strother Moore, and W.D. Young. An approach to system verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989. Special Issue on System Verification.
17. Thomas Bittner, Maureen Donnelly, and Barry Smith. Endurants and Perdurants in Directly Depicting Ontologies. *AI Communications*, 17(4):247–258, December 2004. IOS Press, in [157].
18. Dines Bjørner. A ProCoS Project Description. *Published in two slightly different versions: (1) EATCS Bulletin, October 1989, (2) (Ed. Ivan Plander:) Proceedings: Intl. Conf. on AI & Robotics, Strebske Pleso, Slovakia, Nov. 5-9, 1989, North-Holland, Publ., Dept. of Computer Science, Technical University of Denmark, October 1989.*
19. Dines Bjørner. Trustworthy Computing Systems: The ProCoS Experience. In *14'th ICSE: Intl. Conf. on Software Eng., Melbourne, Australia*, pages 15–34. ACM Press, May 11–15 1992.

20. Dines Bjørner. Michael Jackson's Problem Frames: Domains, Requirements and Design. In Li ShaoYang and Michael Hinchley, editors, *ICFEM'97: International Conference on Formal Engineering Methods*, Los Alamitos, November 12–14 1997. IEEE Computer Society. Final Version¹.
21. Dines Bjørner. Formal Software Techniques in Railway Systems. In Eckehard Schnieder, editor, *9th IFAC Symposium on Control in Transportation Systems*, pages 1–12, Technical University, Braunschweig, Germany, 13–15 June 2000. VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik, VDI-Gesellschaft für Fahrzeug- und Verkehrstechnik. Invited talk.
22. Dines Bjørner. Domain Models of "The Market" — in Preparation for E-Transaction Systems. In *Practical Foundations of Business and System Specifications* (Eds.: Haim Kilov and Ken Baclawski), The Netherlands, December 2002. Kluwer Academic Press. Final draft version².
23. Dines Bjørner. Domain Engineering: A "Radical Innovation" for Systems and Software Engineering ? In *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, Heidelberg, October 7–11 2003. Springer-Verlag. The Zohar Manna International Conference, Taormina, Sicily 29 June – 4 July 2003. ³.
24. Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. .
25. Dines Bjørner. *Software Engineering, Vol. 2: Specification of Systems and Languages*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. Chapters 12–14 are primarily authored by Christian Krog Madsen.
26. Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
27. Dines Bjørner. A Container Line Industry Domain. Techn. report, Fredsvej 11, DK-2840 Holte, Denmark, June 2007. Extensive Draft⁴.
28. Dines Bjørner. Domain Theory: Practice and Theories, Discussion of Possible Research Topics. In *IC-TAC'2007*, volume 4701 of *Lecture Notes in Computer Science* (eds. J.C.P. Woodcock et al.), pages 1–17, Heidelberg, September 2007. Springer.
29. Dines Bjørner. From Domains to Requirements. In *Montanari Festschrift*, volume 5065 of *Lecture Notes in Computer Science* (eds. Pierpaolo Degano, Rocco De Nicola and José Meseguer), pages 1–30, Heidelberg, May 2008. Springer.
30. Dines Bjørner. *Domain Engineering: Technology Management, Research and Engineering*. Research Monograph (# 4); JAIST Press, 1-1, Asahidai, Nomi, Ishikawa 923-1292 Japan, This Research Monograph contains the following main chapters:
 - 1 *On Domains and On Domain Engineering – Prerequisites for Trustworthy Software – A Necessity for Believable Management*, pages 3–38.
 - 2 *Possible Collaborative Domain Projects – A Management Brief*, pages 39–56.
 - 3 *The Rôle of Domain Engineering in Software Development*, pages 57–72.
 - 4 *Verified Software for Ubiquitous Computing – A VSTTE Ubiquitous Computing Project Proposal*, pages 73–106.
 - 5 *The Triptych Process Model – Process Assessment and Improvement*, pages 107–138.
 - 6 *Domains and Problem Frames – The Triptych Dogma and M.A.Jackson's PF Paradigm*, pages 139–175.
 - 7 *Documents – A Rough Sketch Domain Analysis*, pages 179–200.
 - 8 *Public Government – A Rough Sketch Domain Analysis*, pages 201–222.
 - 9 *Towards a Model of IT Security — – The ISO Information Security Code of Practice – An Incomplete Rough Sketch Analysis*, pages 223–282.
 - 10 *Towards a Family of Script Languages – – Licenses and Contracts – An Incomplete Sketch*, pages 283–328.
- 2009.
31. Dines Bjørner. On Mereologies in Computing Science. In *Festschrift: Reflections on the Work of C.A.R. Hoare*, History of Computing (eds. Cliff B. Jones, A.W. Roscoe and Kenneth R. Wood), pages 47–70, London, UK, 2009. Springer.
32. Dines Bjørner. What is Logistics ? A Domain Analysis. Techn. report, Incomplete Draft⁵, Fredsvej 11, DK-2840 Holte, Denmark, June 2009.

¹ <http://www.imm.dtu.dk/db/.pdf>

² <http://www2.imm.dtu.dk/db/themarket.pdf>

³ <http://www2.imm.dtu.dk/db/zohar.pdf>

⁴ <http://www2.imm.dtu.dk/db/container-paper.pdf>

⁵ <http://www2.imm.dtu.dk/db/pipeline.pdf>

33. Dines Bjørner. Domain Engineering. In Paul Boca and Jonathan Bowen, editors, *Formal Methods: State of the Art and New Directions*, Eds. Paul Boca and Jonathan Bowen, pages 1–42, London, UK, 2010. Springer.
34. Dines 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.
35. Dines Bjørner. On Development of Web-based Software: A Divertimento of Ideas and Suggestions. Technical, Technical University of Vienna, August–October 2010. <http://www.imm.dtu.dk/~dibj/wfdftp.pdf>.
36. Dines Bjørner. Believable Software Management. *Encyclopedia of Software Engineering*, 1(1):1–32, 2011.
37. Dines Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics Part II of II: The Science Part*. *Kibernetika i sistemny analiz*, (2):100–120, May 2011.
38. Dines Bjørner. Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions. In *Rainbow of Computer Science, Festschrift for Hermann Maurer on the Occasion of His 70th Anniversary*, Festschrift (eds. C. Calude, G. Rozenberg and A. Saloma), pages 167–183. Springer, Heidelberg, Germany, January 2011.
39. Dines Bjørner. Documents – a Domain Description⁶. Experimental Research Report 2013-3, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013.
40. Dines Bjørner. *Domain Science and Engineering as a Foundation for Computation for Humanity*, chapter 7, pages 159–177. Computational Analysis, Synthesis, and Design of Dynamic Systems. CRC [Francis & Taylor], 2013. (eds.: Justyna Zander and Pieter J. Mosterman).
41. Dines Bjørner. Pipelines – a Domain Description⁷. Experimental Research Report 2013-2, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013.
42. Dines Bjørner. Road Transportation – a Domain Description⁸. Experimental Research Report 2013-4, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013.
43. Dines Bjørner. *A Rôle for Mereology in Domain Science and Engineering*. Synthese Library (eds. Claudio Calosi and Pierluigi Graziani). Springer, Amsterdam, The Netherlands, October 2014.
44. Dines Bjørner. *A Rôle for Mereology in Domain Science and Engineering*. Synthese Library (eds. Claudio Calosi and Pierluigi Graziani). Springer, Amsterdam, The Netherlands, May 2014.
45. Dines Bjørner. Domain Analysis: Endurants – An Analysis & Description Process Model. In Shusaku Iida, José Meseguer, and Kazuhiro Ogata, editors, *Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi*. Springer, May 2014.
46. Dines Bjørner. Domain Engineering – A Basis for Safety Critical Software. Invited Keynote, ASSC2014: Australian System Safety Conference, Melbourne, 26–28 May, December 2014.
47. Dines Bjørner. A Credit Card System: Uppsala Draft. Technical Report: Experimental Research, Fredsvej 11, DK-2840 Holte, Denmark, November 2016. <http://www.imm.dtu.dk/~dibj/2016/credit/accs.pdf>.
48. Dines Bjørner. Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions. *Submitted for consideration to Journal of Logical and Algebraic Methods in Programming*, 2016. <http://www.imm.dtu.dk/~dibj/2016/demos/faoc-demo.pdf>.
49. Dines Bjørner. Manifest Domains: Analysis & Description. *Formal Aspects of Computing*, ...(...):1–51, 2016. DOI 10.1007/s00165-016-0385-z <http://link.springer.com/article/10.1007/s00165-016-0385-z>.
50. Dines Bjørner. To Every Manifest Domain Mereology a CSP Expression. Research Report, Fredsvej 11, DK-2840 Holte, Denmark, December 2016. <http://www.imm.dtu.dk/~dibj/2016/credit/accs.pdf>.
51. Dines Bjørner. Weather Information Systems: Towards a Domain Description. Technical Report: Experimental Research, Fredsvej 11, DK-2840 Holte, Denmark, November 2016. <http://www.imm.dtu.dk/~dibj/2016/wis/wis-p.pdf>.
52. Dines Bjørner. Domain Analysis and Description – Formal Models of Processes and Prompts. *Submitted for consideration to Formal Aspects of Computing*, 2016–2017. <http://www.imm.dtu.dk/~dibj/2016/-process/process-p.pdf>.
53. Dines Bjørner. Domain Facets: Analysis & Description. *Submitted for consideration to Formal Aspects of Computing*, 2016–2017. <http://www.imm.dtu.dk/~dibj/2016/facets/faoc-facets.pdf>.
54. Dines Bjørner. From Domain Descriptions to Requirements Prescriptions – A Different Approach to Requirements Engineering. *Submitted for consideration to Formal Aspects of Computing*, 2016–2017.
55. Dines Bjørner. The Tokyo Stock Exchange Trading Rules. R&D Experiment, Fredsvej 11, DK-2840 Holte, Denmark, January and February, 2010. Version 1, 78 pages: many auxiliary appendices⁹, Version 2, 23 pages: omits many appendices and corrects some errors.¹⁰

⁶ <http://www.imm.dtu.dk/~dibj/doc-p.pdf>

⁷ <http://www.imm.dtu.dk/~dibj/pipe-p.pdf>

⁸ <http://www.imm.dtu.dk/~dibj/road-p.pdf>

⁹ <http://www2.imm.dtu.dk/db/todai/tse-1.pdf>

¹⁰ <http://www2.imm.dtu.dk/db/todai/tse-2.pdf>

56. Dines Bjørner. [57] *Chap. 9: Towards a Model of IT Security — – The ISO Information Security Code of Practice – An Incomplete Rough Sketch Analysis*, pages 223–282. JAIST Press, March 2009.
57. Dines Bjørner. *Domain Engineering: Technology Management, Research and Engineering*. A JAIST Press Research Monograph # 4, 536 pages, March 2009.
58. Dines Bjørner. The Rôle of Domain Engineering in Software Development. Why Current Requirements Engineering Seems Flawed! In *Perspectives of Systems Informatics*, volume 5947 of *Lecture Notes in Computer Science*, pages 2–34, Heidelberg, Wednesday, January 27, 2010. Springer.
59. Dines Bjørner and Asger Eir. Compositionality: Ontology and Mereology of Domains. Some Clarifying Observations in the Context of Software Engineering in July 2008, eds. Martin Steffen, Dennis Dams and Ulrich Hannemann. In *Festschrift for Prof. Willem Paul de Roever Concurrency, Compositionality, and Correctness*, volume 5930 of *Lecture Notes in Computer Science*, pages 22–59, Heidelberg, July 2010. Springer.
60. Dines Bjørner, Chris W. George, Anne Eliabeth Haxthausen, Christian Krog Madsen, Steffen Holmslykke, and Martin Pěnička. "UML"-ising Formal Techniques. In *INT 2004: Third International Workshop on Integration of Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*, pages 423–450. Springer-Verlag, 28 March 2004, ETAPS, Barcelona, Spain. Final Version¹¹.
61. Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.
62. Dines Bjørner and Cliff B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.
63. Dines Bjørner and Jørgen Fischer Nilsson. Algorithmic & Knowledge Based Methods — Do they "Unify" ? In *International Conference on Fifth Generation Computer Systems: FGCS'92*, pages 191–198. ICOT, June 1–5 1992.
64. Wayne D. Blizard. A Formal Theory of Objects, Space and Time. *The Journal of Symbolic Logic*, 55(1):74–89, March 1990.
65. Grady Booch, Jim Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
66. Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. ACM Press/Addison-Wesley, New York, NY, 2000.
67. F. Buschmann, K. Henney, and D.C. Schmidt. *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*. John Wiley & Sons Ltd., England, 2007.
68. R. Casati and A. Varzi. *Parts and Places: the structures of spatial representation*. MIT Press, 1999.
69. Roberto Casati and Achille Varzi. Events. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2010 edition, 2010.
70. Roberto Casati and Achille C. Varzi, editors. *Events*. Ashgate Publishing Group – Dartmouth Publishing Co. Ltd., Wey Court East, Union Road, Farnham, Surrey, GU9 7PT, United Kingdom, 23 March 1996.
71. C.E.C. Digital Rights: Background, Systems, Assessment. Commission of The European Communities, Staff Working Paper, 2002. Brussels, 14.02.2002, SEC(2002) 197.
72. Peter P. Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Trans. Database Syst*, 1(1):9–36, 1976.
73. C. N. Chong, R. J. Corin, J. M. Doumen, S. Etalle, P. H. Hartel, Y. W. Law, and A. Tokmakoff. LicenseScript: a logical language for digital rights management. *Annals of telecommunications special issue on Information systems security*, 2006.
74. C. N. Chong, S. Etalle, and P. H. Hartel. Comparing Logic-based and XML-based Rights Expression Languages. In *Confederated Int. Workshops: On The Move to Meaningful Internet Systems (OTM)*, number 2889 in *LNCS*, pages 779–792, Catania, Sicily, Italy, 2003. Springer.
75. Cheun Ngen Chong, Ricardo Corin, and Sandro Etalle. LicenseScript: A novel digital rights languages and its semantics. In *Proc. of the Third International Conference WEB Delivering of Music (WEDELMU-SIC'03)*, pages 122–129. IEEE Computer Society Press, 2003.
76. David R. Christiansen, Klaus Grue, Henning Niss, Peter Sestoft, and Kristján S. Sigtryggsson. Actulus Modeling Language - An actuarial programming language for life insurance and pensions. Technical Report, http://www.edlund.dk/sites/default/files/Downloads/paper_actulus-modeling-language.pdf, Edlund A/S, Denmark, Bjerregårds Sidevej 4, DK-2500 Valby. (+45) 36 15 06 30. edlund@edlund.dk, <http://www.edlund.dk/en/insights/scientific-papers>. This paper illustrates how the design of pension and life insurance products, and their administration, reserve calculations, and audit, can be based on a common formal notation. The notation is human-readable and machine-processable, and specialised to the actuarial domain, achieving great expressive power combined with ease of use and safety.

¹¹ <http://www.imm.dtu.dk/db/fmuml.pdf>

77. E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.
78. Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, 2000.
79. Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Sci. Comput. Program.*, 20(1-2):3–50, April 1993.
80. Donald Davidson. *Essays on Actions and Events*. Oxford University Press, 1980.
81. Merlin Dorfman and Richard H. Thayer, editors. *Software Requirements Engineering*. IEEE Computer Society Press, 1997.
82. F. Dretske. Can Events Move? *Mind*, 76(479-492), 1967. Reprinted in [70, 1996], pp. 415-428.
83. ESA. Global Navigation Satellite Systems. Web, European Space Agency. http://en.wikipedia.org/wiki/-Satellite_navigation.
84. Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning about Knowledge*. The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts 02142, 1996. 2nd printing.
85. R. Falbo, G. Guizzardi, and K.C. Duarte. An Ontological Approach to Domain Engineering. In *Software Engineering and Knowledge Engineering*, Proceedings of the 14th international conference SEKE'02, pages 351–358, Ischia, Italy, July 15-19 2002. ACM.
86. David John Farmer. *Being in time: The nature of time in light of McTaggart's paradox*. University Press of America, Lanham, Maryland, 1990. 223 pages.
87. Edward A. Feigenbaum and Pamela McCorduck. *The fifth generation*. Addison-Wesley, Reading, MA, USA, 1st ed. edition, 1983.
88. W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, editors. *Beauty is Our Business*, Texts and Monographs in Computer Science, New York, NY, USA, 1990. Springer. A Birthday Salute to Edsger W. Dijkstra.
89. John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. ISBN 0-521-62348-0.
90. Martin Fowler. *Domain Specific Languages*. Signature Series. Addison Wesley, October 20120.
91. Carlo A. Furia, Dino Mandrioli, Angelo Morzenti, and Matteo Rossi. *Modeling Time in Computing*. Monographs in Theoretical Computer Science. Springer, 2012.
92. K. Futatsugi, A.T. Nakagawa, and T. Tamai, editors. *CAFE: An Industrial-Strength Algebraic Formal Method*, Sara Burgerhartstraat 25, P.O. Box 211, NL-1000 AE Amsterdam, The Netherlands, 2000. Elsevier. Proceedings from an April 1998 Symposium, Numazu, Japan.
93. Kokichi Futatsugi, Daniel Gălină, and Kazuhiro Ogata. Principles of proof scores in CafeOBJ. *Theor. Comput. Science*, 464:90–112, 2012.
94. Kokichi Futatsugi and Ataru Nakagawa. An overview of CAFE specification environment - an algebraic approach for creating, verifying, and maintaining formal specifications over networks. In *Proc. of 1st International Conference on Formal Engineering Methods (ICFEM '97), November 12-14, 1997, Hiroshima, JAPAN*, pages 170–182. IEEE, 1997.
95. Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis — Mathematical Foundations*. Springer-Verlag, January 1999.
96. Chris W. George, Peter Haff, Klaus Havelund, Anne Elisabeth Haxthausen, Robert Milne, Claus Bendix Nielsen, Søren Prehn, and Kim Ritter Wagner. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hempstead, England, 1992.
97. Chris W. George, Anne Elisabeth Haxthausen, Steven Hughes, Robert Milne, Søren Prehn, and Jan Storbank Pedersen. *The RAISE Development Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hempstead, England, 1995.
98. Don I. Good and William D. Young. Mathematical Methods for Digital Systems Development. In *VDM '91: Formal Software Development Methods*, pages 406–430. Springer-Verlag, October 1991. Volume 2.
99. T. Grötzer, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer, Dordrecht, 2002.
100. Carl A. Gunter, Elsa L. Gunter, Michael A. Jackson, and Pamela Zave. A Reference Model for Requirements and Specifications. *IEEE Software*, 17(3):37–43, May–June 2000.
101. Carl A. Gunter, Stephen T. Weeks, and Andrew K. Wright. Models and Languages for Digital Rights. In *Proc. of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)*, pages 4034–4038, Maui, Hawaii, USA, January 2001. IEEE Computer Society Press.
102. C.A. Gunther. *Semantics of Programming Languages*. The MIT Press, Cambridge, Mass., USA, 1992.
103. P.M.S. Hacker. Events and Objects in Space and Time. *Mind*, 91:1–19, 1982. reprinted in [70], pp. 429-447.

104. Joseph Y. Halpern and Vicky Weissman. A Formal Foundation for XrML. In *Proc. of the 17th IEEE Computer Security Foundations Workshop (CSFW'04)*, 2004.
105. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
106. David Harel and Rami Marelly. *Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
107. M. Harsu. A Survey on Domain Engineering. Review, Institute of Software Systems, Tampere University of Technology, Finland, December 2002.
108. A.E. Haxthausen, J. Peleska, and S. Kinder. A formal approach for the construction and verification of railway control systems. *Formal Aspects of Computing*, 23:191–219, 2011.
109. Dan Haywood. *Domain-Driven Design Using Naked Objects*. The Pragmatic Bookshelf (an imprint of 'The Pragmatic Programmers, LLC.'). <http://pragprog.com/>, 2009.
110. Martin Heidegger. *Sein und Zeit (Being and Time)*. Oxford University Press, 1927, 1962.
111. Charles Anthony Richard Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985. Published electronically: <http://www.usingcsp.com/csp-book.pdf> (2004).
112. ContentGuard Inc. XrML: Extensible rights Markup Language. <http://www.xrml.org>, 2000.
113. ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992, 1996, 1999.
114. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-10114-9.
115. Michael A. Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. ACM Press. Addison-Wesley, Reading, England, 1995.
116. Michael A. Jackson. *Problem Frames — Analyzing and Structuring Software Development Problems*. ACM Press, Pearson Education. Addison-Wesley, England, 2001.
117. Michael A. Jackson. Program Verification and System Dependability. In Paul Boca and Jonathan Bowen, editors, *Formal Methods: State of the Art and New Directions*, pages 43–78, London, UK, 2010. Springer.
118. Ivar Jacobson, Grady Booch, and Jim Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
119. Ingvar Johansson. Qualities, Quantities, and the Endurant-Perdurant Distinction in Top-Level Ontologies. In K.D. Althoff, A. Dengel, R. Bergmann, M. Nick, and Th. Roth-Berghofer, editors, *Professional Knowledge Management WM 2005*, volume 3782 of *Lecture Notes in Artificial Intelligence*, pages 543–550. Springer, 2005. 3rd Biennial Conference, Kaiserslautern, Germany, April 10-13, 2005, Revised Selected Papers.
120. Cliff B. Jones, Ian Hayes, and Michael A. Jackson. Deriving Specifications for Systems That Are Connected to the Physical World. In Cliff Jones, Zhiming Liu, and James Woodcock, editors, *Formal Methods and Hybrid Real-Time Systems: Essays in Honour of Dines Bjørner and Zhou Chaochen on the Occasion of Their 70th Birthdays*, volume 4700 of *Lecture Notes in Computer Science*, pages 364–390. Springer, 2007.
121. K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. FODA: Feature-Oriented Domain Analysis. Feasibility Study CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, November 1990. <http://www.sei.cmu.edu/library/abstracts/reports/90tr021.cfm>.
122. Jaegwon Kim. *Supervenience and Mind*. Cambridge University Press, 1993.
123. R.H. Koenen, J. Lacy, M. Mackay, and S. Mitchell. The long march to interoperable digital rights management. *Proceedings of the IEEE*, 92(6):883–897, June 2004.
124. Leslie Lamport. *Specifying Systems*. Addison-Wesley, Boston, Mass., USA, 2002.
125. Søren Lauesen. *Software Requirements - Styles and Techniques*. Addison-Wesley, UK, 2002.
126. C. Lejewski. A note on Leśniewski's Axiom System for the Mereological Notion of Ingredient or Element. *Topoi*, 2(1):63–71, June, 1983.
127. Henry S. Leonard and Nelson Goodman. The Calculus of Individuals and its Uses. *Journal of Symbolic Logic*, 5:45–44, 1940.
128. W. Little, H.W. Fowler, J. Coulson, and C.T. Onions. *The Shorter Oxford English Dictionary on Historical Principles*. Clarendon Press, Oxford, England, 1987.
129. Zhiming Liu, J. C. P. Woodcock, and Huibiao Zhu, editors. *Unifying Theories of Programming and Formal Engineering Methods - International Training School on Software Engineering, Held at ICTAC 2013, Shanghai, China, August 26-30, 2013, Advanced Lectures*, volume 8050 of *Lecture Notes in Computer Science*. Springer, 2013.
130. IPR Systems Pty Ltd. Open Digital Rights Language (ODRL). <http://odrl.net>, 2001.
131. Gordon E. Lyon. Information Technology: A Quick-Reference List of Organizations and Standards for Digital Rights Management. NIST Special Publication 500-241, National Institute of Standards and Technology, Technology Administration, U.S. Department of Commerce, Oct 2002.

132. Tom Maibaum. Conservative Extensions, Interpretations Between Theories and All That. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development*, volume 1214 of *LNCS*, pages 40–66, 1997.
133. Neno Medvidovic and Edward Colbert. Domain-Specific Software Architectures (DSSA). Power Point Presentation, found on The Internet, Absolute Software Corp., Inc.: Abs[S/W], 5 March 2004.
134. D.H. Mellor. Things and Causes in Spacetime. *British Journal for the Philosophy of Science*, 31:282–288, 1980.
135. Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, December 2005.
136. Merriam Webster Staff. Online Dictionary: <http://www.m-w.com/home.htm>, 2004. Merriam-Webster, Inc., 47 Federal Street, P.O. Box 281, Springfield, MA 01102, USA.
137. Erik Mettala and Marc H. Graham. The Domain Specific Software Architecture Program. Project Report CMU/SEI-92-SR-009, Software Engineering Institute Carnegie Mellon University Pittsburgh, Pennsylvania 15213, June 1992.
138. S. Michiels, K. Verslype, W. Joosen, and B. De Decker. Towards a Software Architecture for DRM. In *Proceedings of the Fifth ACM Workshop on Digital Rights Management (DRM'05)*, pages 65–74, Alexandria, Virginia, USA, Nov 2005.
139. C. Carroll Morgan. *Programming from Specifications*. International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK, 1990.
140. D. Mulligan and A. Burstein. Implementing copyright limitations in rights expression languages. In *Proc. of 2002 ACM Workshop on Digital Rights Management*, volume 2696 of *Lecture Notes in Computer Science*, pages 137–154. Springer-Verlag, 2002.
141. Deirdre K. Mulligan, John Han, and Aaron J. Burstein. How DRM-Based Content Delivery Systems Disrupt Expectations of “Personal Use”. In *Proc. of The 3rd International Workshop on Digital Rights Management*, pages 77–89, Washington DC, USA, Oct 2003. ACM.
142. James M. Neighbors. The Draco Approach to Constructing Software from Reusable Components. *IEEE Transactions of Software Engineering*, SE-10(5), September 1984.
143. Ernst-Rüdiger Olderog and Henning Dierks. *Real-Time Systems: Formal Specification and Automatic Verification*. Cambridge University Press, UK, 2008.
144. Ernst Rüdiger Olderog, Anders Peter Ravn, and Rafael Wisniewski. Linking Discrete and Continuous Models, Applied to Traffic Maneuvers. In Jonathan Bowen, Michael Hinchey, and Ernst Rüdiger Olderog, editors, *BCS FACS – ProCoS Workshop on Provably Correct Systems*, Lecture Notes in Computer Science. Springer, 2016.
145. Leon Osterweil. Software processes are software too. In *Proceedings of the 9th International Conference on Software Engineering*, ICSE '87, pages 2–13, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.
146. Shari Lawrence Pfleeger. *Software Engineering, Theory and Practice*. Prentice-Hall, 2nd edition, 2001.
147. Chia-Yi Tony Pi. *Mereology in Event Semantics*. Phd, McGill University, Montreal, Canada, August 1999.
148. K. Pohl, G. Bockle, and F. van der Linden. *Software Product Line Engineering*. Springer, Berlin, Heidelberg, New York, 2005.
149. Roger S. Pressman. *Software Engineering, A Practitioner's Approach*. International Edition, Computer Science Series. McGraw-Hill, 5th edition, 1981–2001.
150. Rubén Prieto-Díaz. Domain Analysis for Reusability. In *COMPSAC 87*. ACM Press, 1987.
151. Rubén Prieto-Díaz. Domain analysis: an introduction. *Software Engineering Notes*, 15(2):47–54, 1990.
152. Rubén Prieto-Díaz and Guillermo Arrango. *Domain Analysis and Software Systems Modelling*. IEEE Computer Society Press, 1991.
153. Riccardo Pucella and Vicky Weissman. A Logic for Reasoning about Digital Rights. In *Proc. of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, pages 282–294. IEEE Computer Society Press, 2002.
154. Riccardo Pucella and Vicky Weissman. A Formal Foundation for ODRL. In *Proc. of the Workshop on Issues in the Theory of Security (WIST'04)*, 2004.
155. A. Quinton. Objects and Events. *Mind*, 88:197–214, 1979.
156. Wolfgang Reisig. *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*. Leitfäden der Informatik. Vieweg+Teubner, 1st edition, 15 June 2010. 248 pages; ISBN 978-3-8348-1290-2.
157. Jochen Renz and Hans W. Guesgen, editors. *Spatial and Temporal Reasoning*, volume 14, vol. 4, Journal: AI Communications, Amsterdam, The Netherlands, Special Issue. IOS Press, December 2004.
158. John C. Reynolds. *The Semantics of Programming Languages*. Cambridge University Press, 1999.
159. Douglas T. Ross. Computer-aided design. *Commun. ACM*, 4(5):41–63, 1961.

160. Douglas T. Ross. Toward foundations for the understanding of type. In *Proceedings of the 1976 conference on Data: Abstraction, definition and structure*, pages 63–65, New York, NY, USA, 1976. ACM.
161. Douglas T. Ross and J. E. Ward. Investigations in computer-aided design for numerically controlled production. Final Technical Report ESL-FR-351, , May 1968. 1 December 1959 – 3 May 1967. Electronic Systems Laboratory Electrical Engineering Department, MIT, Cambridge, Massachusetts 02139.
162. Jim Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
163. Pamela Samuelson. Digital rights management {and, or, vs.} the law. *Communications of ACM*, 46(4):41–45, Apr 2003.
164. Donald Sannella and Andrzej Tarlecki. *Foundations of Algebraic Semantics and Formal Software Development*. Monographs in Theoretical Computer Science. Springer, Heidelberg, 2012.
165. David A. Schmidt. *Denotational Semantics: a Methodology for Language Development*. Allyn & Bacon, 1986.
166. John R. Searle. *Speech Act*. CUP, 1969.
167. Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
168. Barry Smith. Mereotopology: A Theory of Parts and Boundaries. *Data and Knowledge Engineering*, 20:287–303, 1996.
169. Ian Sommerville. *Software Engineering*. Pearson, 8th edition, 2006.
170. John F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Brooks/Cole Thompson Learning, August 17, 1999.
171. Diomidis Spinellis. Notable design patterns for domain specific languages. *Journal of Systems and Software*, 56(1):91–99, February 2001.
172. Staff of Encyclopædia Britannica. Encyclopædia Britannica. Merriam Webster/Britannica: Access over the Web: <http://www.eb.com:180/>, 1999.
173. Robert Tennent. *The Semantics of Programming Languages*. Prentice-Hall Intl., 1997.
174. Will Tracz. Domain-specific software architecture (DSSA) frequently asked questions (FAQ). *Software Engineering Notes*, 19(2):52–56, 1994.
175. Johan van Benthem. *The Logic of Time*, volume 156 of *Synthese Library: Studies in Epistemology, Logic, Methodology, and Philosophy of Science (Editor: Jaakko Hintikka)*. Kluwer Academic Publishers, P.O.Box 17, NL 3300 AA Dordrecht, The Netherlands, second edition, 1983, 1991.
176. F. Van der Rhee, H.R. Van Nauta Lemke, and J.G. Dukman. Knowledge based fuzzy control of systems. *IEEE Trans. Autom. Control*, 35(2):148–155, February 1990.
177. Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.
178. H. Wang, J. S. Dong, and J. Sun. Reasoning Support for Semantic Web Ontology Family Languages Using Alloy. *International Journal of Multiagent and Grid Systems*, IOS Press, 2(4):455–471, 2006.
179. Ji Wang, XinYao Yu, and Chao Chen Zhou. Hybrid Refinement. Research Report 20, UNU/IIST, P.O.Box 3058, Macau, 1. April 1994.
180. A.N. Whitehead. *The Concept of Nature*. Cambridge University Press, Cambridge, 1920.
181. George Wilson and Samuel Shpall. Action. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2012 edition, 2012.
182. G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, Cambridge, Mass., USA, 1993.
183. J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.
184. WWW. Domain Descriptions:
 - 1 A Container Line Industry Domain: www2.imm.dtu.dk/~db/container-paper.pdf
 - 2 What is Logistics: www2.imm.dtu.dk/~db/logistics.pdf
 - 3 The “Market”: Consumers, Retailers, Wholesalers, Producers www2.imm.dtu.dk/~db/themarket.-pdf
 - 4 MITS: Models of IT Security. Security Rules & Regulations: www2.imm.dtu.dk/~db/it-security.-pdf
 - 5 A Domain Model of Oil Pipelines: www2.imm.dtu.dk/~db/pipeline.pdf
 - 6 A Railway Systems Domain: <http://euler.fd.cvut.cz/railwaydomain>
 - 7 Transport Systems: www2.imm.dtu.dk/~db/comet/comet1.pdf
 - 8 The Tokyo Stock Exchange www2.imm.dtu.dk/~db/todai/tse-1.pdf and www2.imm.dtu.dk/~db/todai/tse-2.pdf

- 9 *On Development of Web-based Software. A Divertimento of Ideas and Suggestions*: www2.imm.dtu.dk/~db/wfdftp.pdf
 . R&D Experiments, Dines Bjørner, DTU Informatics, Technical University of Denmark, 2007–2010.
185. JianWen Xiang and Dines Bjørner. The Electronic Media Industry: A Domain Analysis and a License Language. Technical note, JAIST, School of Information Science, 1-1, Asahidai, Tatsunokuchi, Nomi, Ishikawa, Japan 923-1292, Summer 2006.
186. Naijun Zhan, Shuling Wang, and Hengjun Zhao. Formal modelling, analysis and verification of hybrid systems. In *ICTAC Training School on Software Engineering*, pages 207–281, 2013. http://dx.doi.org/10.1007/978-3-642-39721-9_5, DBLP, <http://dblp.uni-trier.de>.
187. Chao Chen Zhou and Michael R. Hansen. *Duration Calculus: A Formal Approach to Real-time Systems*. Monographs in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2004.

Domain Descriptions

In this appendix we show two domain descriptions. Both very terse:

- **Credit Card Systems**
- **Weather Information Systems**

Appendix A, Pages 205–213

Appendix B, Pages 215–226

I have posted, on the Internet, a number of other domain descriptions:

- **Documents**
<http://www.imm.dtu.dk/~dibj/doc-p.pdf> [39] 2013.
- **Transport Systems**
<http://www.imm.dtu.dk/~dibj/comet/comet1.pdf> [42] 2010.
- **The Tokyo Stock Exchange**
<http://www.imm.dtu.dk/~dibj/todai/tse-1.pdf>
 and <http://www.imm.dtu.dk/~dibj/todai/tse-2.pdf> [55] 2010.
- **On Development of Web-based Software: A Divertimento**
<http://www.imm.dtu.dk/~dibj/wfdftp.pdf> 2010.
- Pipelines – a Domain Description¹ [41] 2009.
- **What is Logistics ?**
<http://www.imm.dtu.dk/~dibj/logistics.pdf> [32] 2009.
- **A Container Line Industry Domain**
<http://www.imm.dtu.dk/~dibj/container-paper.pdf>, [27] 2007
- **Models of IT Security: Security Rules & Regulations**
<http://www.imm.dtu.dk/~dibj/it-security.pdf> [56] 2006.
- **Railway Systems Descriptions: 1996–2003**
 - ⊗ Chris George, Dines Bjørner and Søren Prehn:
Scheduling and Rescheduling of Trains
<http://www.imm.dtu.dk/~dibj/amore/docs/scheduling.pdf>, 1996
 - ⊗ Dines Bjørner:
A Railway Systems Domain
<http://www.imm.dtu.dk/~dibj/UNU-IIST-railways.pdf>
 An “old” UNU-IIST report, 1997
 - ⊗ Dines Bjørner:
Formal Software Techniques in Railway Systems
<http://www.imm.dtu.dk/~dibj/amore/docs/dines-ifac.pdf>, 2002
 - ⊗ Alben Strupchanska, Martin Penicka and Dines Bjørner:
Railway Staff Rostering
<http://www.imm.dtu.dk/~dibj/amore/docs/albena-amore.pdf>, 2003
 - ⊗ Dines Bjørner:
Dynamics of Railway Nets
<http://www.imm.dtu.dk/~dibj/amore/docs/ifac-dynamics.pdf>, 2003
 - ⊗ Martin Penicka, Alben Strupchanska and Dines Bjørner:
Train Maintenance Routing
<http://www.imm.dtu.dk/~dibj/amore/docs/martin-amore.pdf>, 2003

¹ <http://www.imm.dtu.dk/~dibj/pipeline.pdf>

Credit Card Systems

Summary

This report presents an attempt at a model of a credit card system. *Right-flushed “one-liners” refer to domain description prompts of Chapter 1.* Appendix C presents a primer of RSL, the Raise Specification Language.

A.1 Introduction

We present a domain description of an abstracted credit card system. The narrative part of the description is terse, perhaps a bit too terse. I might “repair” this shortness if told so. A reference is made to my paper: [49, Manifest Domains: Analysis & Description]. That paper can be found on the Internet: <http://www2.compute.dtu.dk/~dibj/2015/faoc/faoc-bjorner.pdf>.

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

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

A.2 Endurants

A.2.1 Credit Card Systems

Sect. 1.3.1 Domain Description Prompt 1 on Page 15: observe_part_sorts

414 Credit card systems, *ccs:CCS*,¹ consists of three kinds of parts:

415 an assembly, *cs:CS*, of credit cards³,

416 an assembly, *bs:BS*, of banks, and

417 an assembly, *ss:SS*, of shops.

¹ The composite part *CS* can be thought of as a credit card company, say VISA². The composite part *BS* can be thought of as a bank society, say BBA: British Banking Association. The composite part *SS* can be thought of as the association of retailers, say bira: British Independent Retailers Association. The model does not prevent “shops” from being airlines, or car rental agencies, or dentists, or consultancy firms. In this case *SS* would be some appropriate association.

³ We “equate” credit cards with their holders.

type

414 CCS

415 CS

416 BS

417 SS

value

415 **obs_part_CS**: CCS \rightarrow CS

416 **obs_part_BS**: CCS \rightarrow BS

417 **obs_part_SS**: CCS \rightarrow SS

Sect. 1.3.1 Domain Description Prompt 2 on Page 16: observe_concrete_type

418 There are credit cards, $c:C$, banks $b:B$, and shops $s:S$.

419 The credit card part, $cs:CS$, abstracts a set, $soc:Cs$, of card.

420 The bank part, $bs:BS$, abstracts a set, $sob:Bs$, of banks.

421 The shop part, $ss:SS$, abstracts a set, $sos:Ss$, of shops.

type

418 C, B, S

419 $Cs = C\text{-set}$

420 $Bs = B\text{-set}$

421 $Ss = S\text{-set}$

value

419 **obs_part_CS**: CS \rightarrow Cs, **obs_part_Cs**: CS \rightarrow Cs

420 **obs_part_BS**: BS \rightarrow Bs, **obs_part_Bs**: BS \rightarrow Bs

421 **obs_part_SS**: SS \rightarrow Ss, **obs_part_Ss**: SS \rightarrow Ss

Sect. 1.3.2 Domain Description Prompt 3 on Page 19: observe_unique_identifier

422 Assemblers of credit cards, banks and shops have unique identifiers, $csi:CSI$, $bsi:BSI$, and $ssi:SSI$.

423 Credit cards, banks and shops have unique identifiers, $ci:CI$, $bi:BI$, and $si:SI$.

424 One can define functions which extract all the

425 unique credit card,

426 bank and

427 shop identifiers from a credit card system.

422 CSI, BSI, SSI

423 CI, BI, SI

value

422 **uid_CS**: CS \rightarrow CSI, **uid_BS**: BS \rightarrow BSI, **uid_SS**: SS \rightarrow SSI,

423 **uid_C**: C \rightarrow CI, **uid_B**: B \rightarrow BI, **uid_S**: S \rightarrow SI,

425 **xtr_CIs**: CCS \rightarrow CI-set

425 $\text{xtr_CIs}(\text{ccs}) \equiv \{\text{uid_C}(c) \mid c:C \bullet c \in \text{obs_part_Cs}(\text{obs_part_CS}(\text{ccs}))\}$

426 **xtr_BIs**: CCS \rightarrow BI-set

426 $\text{xtr_BIs}(\text{ccs}) \equiv \{\text{uid_B}(b) \mid b:B \bullet b \in \text{obs_part_Bs}(\text{obs_part_BS}(\text{ccs}))\}$

427 **xtr_SIs**: CCS \rightarrow SI-set

427 $\text{xtr_SIs}(\text{ccs}) \equiv \{\text{uid_S}(s) \mid s:S \bullet s \in \text{obs_part_Ss}(\text{obs_part_SS}(\text{ccs}))\}$

428 For all credit card systems it is the case that

429 all credit card identifiers are distinct from bank identifiers,

430 all credit card identifiers are distinct from shop identifiers,

431 all shop identifiers are distinct from bank identifiers,

axiom

428 $\forall \text{ccs:CCS} \bullet$

428 **let** $\text{cis} = \text{xtr_CIs}(\text{ccs})$, $\text{bis} = \text{xtr_BIs}(\text{ccs})$, $\text{sis} = \text{xtr_SIs}(\text{ccs})$ **in**

429 $\text{cis} \cap \text{bis} = \{\}$

430 $\text{cis} \cap \text{sis} = \{\}$

431 $\text{sis} \cap \text{bis} = \{\}$ **end**

A.2.2 Credit Cards

Sect. 1.3.3 Domain Description Prompt 4 on Page 21: observe_mereology

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

type

432. $CM = SI\text{-}set \times BI$

value

432. $obs_mereo_CM: C \rightarrow CM$

434 $wf_CM_of_C: CCS \rightarrow Bool$

434 $wf_CM_of_C(ccs) \equiv$

432 $\text{let } bis = xtr_BIs(ccs), sis = xtr_SIs(ccs) \text{ in}$
 432 $\quad \forall c: C \bullet c \in obs_part_Cs(obs_part_CS(ccs)) \Rightarrow$
 432 $\quad \text{let } (ccsis, bi) = obs_mereo_CM(c) \text{ in}$
 435 $\quad ccsis \subseteq sis$
 436 $\quad \wedge bi \in bis$
 432 end end

A.2.3 Banks

Sect. 1.3.2 Domain Description Prompt 3 on Page 19: observe_unique_identifier

Sect. 1.3.3 Domain Description Prompt 4 on Page 21: observe_mereology

Our model of banks is (also) very limited.

437 A bank has a mereology which “connects” it to a subset of all credit cards and a subset of all shops,
 438 and, as attributes:
 439 a cash register, and
 440 a ledger.
 441 The ledger records for every card, by unique credit card identifier,
 442 the current balance: how much money, credit or debit, i.e., plus or minus, that customer is owed, respectively has borrowed from the bank,
 443 the dates-of-issue and -expiry of the credit card, and
 444 the name, address, and other information about the credit card holder.
 445 The wellformedness of the credit card system includes the wellformedness of the banks with respect to the credit cards and shops:
 446 the bank mereology’s
 447 must list a subset of the credit card identifiers and a subset of the shop identifiers.

type

437 $BM = CI\text{-}set \times SI\text{-}set$

439 $CR = Bal$

440 $LG = CI \rightarrow_m (Bal \times DoI \times DoE \times \dots)$

442 $Bal = Int$

value

437 $obs_mereo_B: B \rightarrow BM$

439 $attr_CR: B \rightarrow CR$

440 $attr_LG: B \rightarrow LG$

445 $wf_BM_B: CCS \rightarrow Bool$

445 $wf_BM_B(ccs) \equiv$

445 $\text{let } allcis = xtr_CIs(ccs), allsis = xtr_SIs(ccs) \text{ in}$
 445 $\quad \forall b: B \bullet b \in obs_part_Bs(obs_part_BS(ccs)) \text{ in}$
 446 $\quad \text{let } (cis, sis) = obs_mereo_B(b) \text{ in}$
 447 $\quad cis \subseteq \forall cis \wedge sis \subseteq allsis \text{ end end}$

A.2.4 Shops

Sect. 1.3.3 Domain Description Prompt 4 on Page 21: *observe_mereology*

448 The mereology of a shop is a pair: a unique bank identifiers, and a set of unique credit card identifiers.

449 The mereology of a shop

450 must list a bank of the credit card system,

451 band a subset (or all) of the unique credit identifiers.

We omit treatment of shop attributes.

type

448 $SM = CI\text{-}set \times BI$

value

448 $obs_mereo_S: S \rightarrow SM$

449 $wf_SM_S: CCS \rightarrow Bool$

449 $wf_SM_S(ccs) \equiv$

449 $\text{let } allcis = xtr_CIs(ccs), allbis = xtr_BIs(ccs) \text{ in}$

449 $\forall s:S \bullet s \in obs_part_SS(obs_part_SS(ccs)) \Rightarrow$

449 $\text{let } (cis, bi) \text{ } obs_mereo_S(s) \text{ in}$

450 $bi \in allbis$

451 $\wedge cis \subseteq allcis$

449 **end end**

A.3 Perdurants

A.3.1 Behaviours

Sect. 1.4.11: Process Schema I: Abstract *is_composite(p)*, Page 39Sect. 1.4.11: Process Schema II: Concrete *is_concrete(p)*, Page 40

452 We ignore the behaviours related to the *CCS*, *CS*, *BS* and *SS* parts.

453 We therefore only consider the behaviours related to the *Cs*, *Bs* and *Ss* parts.

454 And we therefore compile the credit card system into the parallel composition of the parallel compositions of all the credit card, *crd*, all the bank, *bnk*, and all the shop, *shp*, behaviours.

value

452 $ccs:CCS$

452 $cs:CS = obs_part_CS(ccs),$

452 $uics:CSI = uid_CS(cs),$

452 $bs:BS = obs_part_BS(ccs),$

452 $uibs:BSI = uid_BS(bs),$

452 $ss:SS = obs_part_SS(ccs),$

452 $uiss:SSI = uid_SS(ss),$

453 $socs:Cs = obs_part_Cs(cs),$

453 $sobs:Bs = obs_part_Bs(bs),$

453 $soss:Ss = obs_part_Ss(ss),$

value

454 $sys: Unit \rightarrow Unit,$

452 $sys() \equiv$

454 $\text{cards}_{uics}(obs_mereo_CS(cs), \dots)$

454 $\parallel \{ \text{crd}_{uid_C(c)}(obs_mereo_C(c)) \mid c:C \bullet c \in soCs \}$

454 $\parallel \text{banks}_{uibs}(obs_mereo_BS(bs), \dots)$

454 $\parallel \{ \text{bnk}_{uid_B(b)}(obs_mereo_B(b)) \mid b:B \bullet b \in sobS \}$

454 $\parallel \text{shops}_{uiss}(obs_mereo_SS(ss), \dots)$

454 $\parallel \{ \text{shp}_{uid_S(s)}(obs_mereo_S(s)) \mid s:S \bullet s \in soss \},$

452 $\text{cards}_{uics}(\dots) \equiv \text{skip},$
 452 $\text{banks}_{uibs}(\dots) \equiv \text{skip},$
 452 $\text{shops}_{uiss}(\dots) \equiv \text{skip}$

axiom $\text{skip} \parallel \text{behaviour}(\dots) \equiv \text{behaviour}(\dots)$

A.3.2 Channels

Sect. 1.4.5: Channels and Communications, Page 34

Sect. 1.4.5: Relations Between Attributes Sharing and Channels, Page 34

455 Credit card behaviours interact with bank (each with one) and many shop behaviours.

456 Shop behaviours interact with bank (each with one) and many credit card behaviours.

457 Bank behaviours interact with many credit card and many shop behaviours.

The inter-behaviour interactions concern:

458 between credit cards and banks: withdrawal requests as to a sufficient, $\text{mk_Wdr}(\text{am})$, balance on the credit card account for buying am:AM amounts of goods or services, with the bank response of either $\text{is_OK}()$ or $\text{is_NOK}()$, or the revoke of a card;

459 between credit cards and shops: the buying, for an amount, am:AM , of goods or services: $\text{mk_Buy}(\text{am})$, or the refund of an amount;

460 between shops and banks: the deposit of an amount, am:AM , in the shops' bank account: $\text{mk_Depost}(\text{ui}, \text{am})$ or the removal of an amount, am:AM , from the shops' bank account: $\text{mk_Removl}(\text{bi}, \text{si}, \text{am})$

channel

455 $\{\text{ch_cb}[\text{ci}, \text{bi}] \mid \text{ci:CI}, \text{bi:BI} \bullet \text{ci} \in \text{cis} \wedge \text{bi} \in \text{bis}\} : \text{CB_Msg}$
 456 $\{\text{ch_cs}[\text{ci}, \text{si}] \mid \text{ci:CI}, \text{si:SI} \bullet \text{ci} \in \text{cis} \wedge \text{si} \in \text{sis}\} : \text{CS_Msg}$
 457 $\{\text{ch_sb}[\text{si}, \text{bi}] \mid \text{si:SI}, \text{bi:BI} \bullet \text{si} \in \text{sis} \wedge \text{bi} \in \text{bis}\} : \text{SB_Msg}$
 458 $\text{CB_Msg} == \text{mk_Wdrw}(\text{am:aM}) \mid \text{is_OK}() \mid \text{is_NOK}() \mid \dots$
 459 $\text{CS_Msg} == \text{mk_Buy}(\text{am:aM}) \mid \text{mk_Ref}(\text{am:aM}) \mid \dots$
 460 $\text{SB_Msg} == \text{Depost} \mid \text{Removl} \mid \dots$
 460 $\text{Depost} == \text{mk_Dep}((\text{ci:CI} \mid \text{si:SI}), \text{am:aM}) \mid$
 460 $\text{Removl} == \text{mk_Rem}(\text{bi:BI}, \text{si:SI}, \text{am:aM})$

A.3.3 Behaviour Interactions

461 The credit card initiates

a buy transactions

i [1.Buy] by enquiring with its bank as to sufficient purchase funds (am:aM);

ii [2.Buy] if NOK then there are presently no further actions; if OK

iii [3.Buy] the credit card requests the purchase from the shop – handing it an appropriate amount;

iv [4.Buy] finally the shop requests its bank to deposit the purchase amount into its bank account.

b refund transactions

i [1.Refund] by requesting such refunds, in the amount of am:aM , from a[ny] shop; whereupon

ii [2.Refund] the shop requests its bank to move the amount am:aM from the shop's bank account

iii [3.Refund] to the credit card's account.

Thus the three sets of behaviours, crd , bnk and shp interact as sketched in Fig. A.1 on the following page.

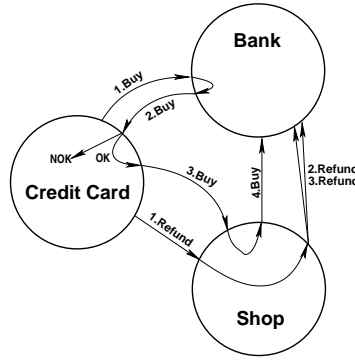


Fig. A.1. Credit Card, Bank and Shop Behaviours

[1.Buy]	Item 467, Pg.210 Item 476, Pg.211	card ch_cb[ci,bi]!mk_Wdrw(am) (shown as ... three lines down) and bank mk_Wdrw(ci,am)=[] {ch_cb[bi,bi]? ci:CI•ci ∈ cis}.
[2.Buy]	Items 469-470, Pg.211 Item 467, Pg.210	bank ch_cb[ci,bi]!lis_[N]OK() and shop (...;ch_cb[ci,bi]?).
[3.Buy]	Item 469, Pg.211 Item 491, Pg.213	card ch_cs[ci,si]!mk_Buy(am) and shop mk_Buy(am)=[] {ch_cs[ci,si]? ci:CI•ci ∈ cis}.
[4.Buy]	Item 492, Pg.213 Item 481, Pg.212	shop ch_sb[si,bi]!mk_Dep(si,am) and bank mk_Dep(si,am)=[] {ch_cs[ci,si]? si:SI•si ∈ sis}.
[1.Refund]	Item 473, Pg.211 Item 492, Pg.213	card ch_cs[ci,si]!mk_Ref((ci,si),am) and shop (si,mk_Ref(ci,am))=[] {si',ch_sb[si,bi]? si,si':SI•{si,si'} ⊆ sis ∧ si=si'}.
[2.Refund]	Item 496, Pg.213 Item 485, Pg.212	shop ch_sb[si,cbi]!mk_Ref(cbi,(ci,si),am and bank (si,mk_Ref(cbi,(ci,am)))=[] {(si',ch_sb[si,bi]? si,si':SI•{si,si'} ⊆ sis ∧ si=si')}.
[3.Refund]	Item 497, Pg.213 Item 486, Pg.212	shop ch_sb[si,sbi]!mk_Wdr(si,am)) end and bank (si,mk_Wdr(ci,am))=[] {(si',ch_sb[si,bi]? si,si':SI•{si,si'} ⊆ sis ∧ si=si')}

A.3.4 Credit Card

Sect. 1.4.11: Process Schema III: *is_atomic(p)*, Page 40

462 The credit card behaviour, *crd*, takes the credit card unique identifier, the credit card mereology, and attribute arguments (omitted). The credit card behaviour, *crd*, accepts inputs from and offers outputs to the bank, *bi*, and any of the shops, *si* ∈ *sis*.

463 The credit card behaviour, *crd*, non-deterministically, internally “cycles” between buying and getting refunds.

value

462 $crd_{ci:CI}: (bi, sis):CM \rightarrow \mathbf{in, out} \text{ ch_cb}[ci, bi], \{ch_cs[ci, si] | si:SI \bullet si \in sis\} \text{ Unit}$

462 $crd_{ci}(bi, sis) \equiv (\text{buy}(ci, (bi, sis)) \sqcap \text{ref}(ci, (bi, sis))) ; crd_{ci}(ci, (bi, sis))$

Sect. 1.4.11: Process Schema IV: Core Processes (I), Page 41

Sect. 1.4.11: Process Schema V: Core Processes (II), Page 41

464 By *am:AM* we mean an amount of money, and by *si:SI* we refer to a shop in which we have selected a number or goods or services (not detailed) costing *am:AM*.

465 The buyer action is simple.

466 The amount for which to buy and the shop from which to buy are selected (arbitrarily).

467 The credit card (holder) withdraws *am:AM* from the bank, if sufficient funds are available⁴.

468 The response from the bank

⁴ First the credit card [holder] requests a withdrawal. If sufficient funds are available, then the withdrawal takes place, otherwise not – and the credit card holder is informed accordingly.

469 is either OK and the credit card [holder] completes the purchase by buying the goods or services offered by the selected shop,
 470 or the response is “not OK”, and the transaction is skipped.

type

464 $AM = \text{Int}$

value

```

465 buy:  $ci:CI \times (bi, sis):CM \rightarrow$ 
465   in, out  $ch\_cb[ci, bi]$  out  $\{ch\_cs[ci, si] | si:SI \cdot si \in sis\}$  Unit
465 buy( $ci, (bi, sis)$ )  $\equiv$ 
466   let  $am:AM \cdot am > 0, si:SI \cdot si \in sis$  in
467   let  $msg = (ch\_cb[ci, bi]!mk\_Wdrw(am); ch\_cb[ci, bi]?)$  in
468   case  $msg$  of
469      $is\_OK() \rightarrow ch\_cs[ci, si]!mk\_Buy(am),$ 
470      $is\_NOK() \rightarrow skip$ 
465   end end end

```

471 The refund action is simple.

472 The credit card [handler] requests a refund $am:AM$

473 from shop $si:SI$.

This request is handled by the shop behaviour’s sub-action *ref*, see lines 489.–498. page 213.

value

```

471 rfu:  $ci:CI \times (bi, sis):CM \rightarrow$  out  $\{ch\_cs[ci, si] | si:SI \cdot si \in sis\}$  Unit
471 rfu( $ci, (bi, sis)$ )  $\equiv$ 
472   let  $am:AM \cdot am > 0, si:SI \cdot si \in sis$  in
473    $ch\_cs[ci, si]!mk\_Ref(bi, (ci, si), am)$ 
471   end

```

A.3.5 Banks

Sect. 1.4.11: Process Schema III: *is_atomic(p)*, Page 40

474 The bank behaviour, *bnk*, takes the bank’s unique identifier, the bank mereology, and the programmable attribute arguments: the ledger and the cash register. The bank behaviour, *bnk*, accepts inputs from and offers outputs to the any of the credit cards, $ci \in cis$, and any of the shops, $si \in sis$.

475 The bank behaviour non-deterministically externally chooses to accept either ‘withdraw’al requests from credit cards or ‘deposit’ requests from shops or ‘refund’ requests from credit cards.

value

```

474  $bnk_{bi:BI}: (cis, sis):BM \rightarrow (LG \times CR) \rightarrow$ 
474   in, out  $\{ch\_cb[ci, bi] | ci:CI \cdot ci \in cis\} \{ch\_sb[si, bi] | si:SI \cdot si \in sis\}$  Unit
474  $bnk_{bi}((cis, sis))(lg: (bal, doi, doe, \dots), cr) \equiv$ 
475    $wdrw(bi, (cis, sis))(lg, cr)$ 
475    $\square$   $depo(bi, (cis, sis))(lg, cr)$ 
475    $\square$   $refu(bi, (cis, sis))(lg, cr)$ 

```

476 The ‘withdraw’ request, *wdrw*, (an action) non-deterministically, externally offers to accept input from a credit card behaviour and marks the only possible form of input from credit cards, $mk_Wdrw(ci, am)$, with the identity of the credit card.

477 If the requested amount (to be withdrawn) is not within balance on the account

478 then we, at present, refrain from defining an outcome (**chaos**), whereupon the bank behaviour is resumed with no changes to the ledger and cash register;

479 otherwise the bank behaviour informs the credit card behaviour that the amount can be withdrawn; whereupon the bank behaviour is resumed notifying a lower balance and ‘withdraws’ the monies from the cash register.

value

```

475 wdrw: bi:BI  $\times$  (cis, sis):BM  $\rightarrow$  (LG  $\times$  CR)  $\rightarrow$  in, out {ch_cb[bi, ci] | ci:CI  $\bullet$  ci  $\in$  cis} Unit
475 wdrw(bi, (cis, sis))(lg, cr)  $\equiv$ 
476   let mk_Wdrw(ci, am) =  $\square$  {ch_cb[ci, bi]? | ci:CI  $\bullet$  ci  $\in$  cis} in
475   let (bal, doi, doe) = lg(ci) in
477   if am > bal
478     then (ch_cb[ci, bi]!is_NOK(); bnkbi(cis, sis)(lg, cr))
479     else (ch_cb[ci, bi]!is_OK(); bnkbi(cis, sis)(lg+[ci  $\rightarrow$  (bal - am, doi, doe)], cr - am)) end
474   end end

```

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

480 The deposit action is invoked, either by a shop behaviour, when a credit card [holder] buy's for a certain amount, am:AM, or requests a refund of that amount. The deposit is made by shop behaviours, either on behalf of themselves, hence am:AM, is to be inserted into the shops' bank account, si:SI, or on behalf of a credit card [i.e., a customer], hence am:AM, is to be inserted into the credit card holder's bank account, si:SI.

481 The message, ch_cs[ci, si]?, received from a credit card behaviour is either concerning a buy [in which case *i* is a ci:CI, hence sale, or a refund order [in which case *i* is a si:SI].

482 In either case, the respective bank account is "upped" by am:AM – and the bank behaviour is resumed.

value

```

480 deposit: bi:BI  $\times$  (cis, sis):BM  $\rightarrow$  (LG  $\times$  CR)  $\rightarrow$ 
481   in, out {ch_sb[bi, si] | si:SI  $\bullet$  si  $\in$  sis} Unit
480 deposit(bi, (cis, sis))(lg, cr)  $\equiv$ 
481   let mk_Dep(si, am) =  $\square$  {ch_cs[ci, si]? | si:SI  $\bullet$  si  $\in$  sis} in
480   let (bal, doi, doe) = lg(si) in
482   bnkbi(cis, sis)(lg+[si  $\rightarrow$  (bal + am, doi, doe)], cr + am)
480   end end

```

483 The refund action

484 non-deterministically externally offers to either

485 non-deterministically externally accept a mk_Ref(ci, am) request from a shop behaviour, si, or

486 non-deterministically externally accept a mk_Wdr(ci, am) request from a shop behaviour, si.

The bank behaviour is then resumed with the

487 credit card's bank balance and cash register incremented by am and the

488 shop' bank balance and cash register decremented by that same amount.

value

```

483 rfu: bi:BI  $\times$  (cis, sis):BM  $\rightarrow$  (LG  $\times$  CR)  $\rightarrow$  in, out {ch_sb[bi, si] | si:SI  $\bullet$  si  $\in$  sis} Unit
483 rfu(bi, (cis, sis))(lg, cr)  $\equiv$ 
485   ((let (si, mk_Ref(cbi, (ci, am))) =  $\square$  {(si', ch_sb[si, bi]) | si, si':SI  $\bullet$  {si, si'}  $\subseteq$  sis  $\wedge$  si = si'}) in
483   let (balc, doic, doec) = lg(ci) in
487   bnkbi(cis, sis)(lg+[ci  $\rightarrow$  (balc + am, doic, doec)], cr + am)
483   end end)
484    $\square$ 
486   ((let (si, mk_Wdr(ci, am)) =  $\square$  {(si', ch_sb[si, bi]) | si, si':SI  $\bullet$  {si, si'}  $\subseteq$  sis  $\wedge$  si = si'}) in
483   let (bals, dois, does) = lg(si) in
488   bnkbi(cis, sis)(lg+[si  $\rightarrow$  (bals - am, dois, does)], cr - am)
483   end end)

```

A.3.6 Shops

Sect. 1.4.11: Process Schema III: *is_atomic(p)*, Page 40

489 The shop behaviour, shp, takes the shop's unique identifier, the shop mereology, etcetera.

490 The shop behaviour non-deterministically, externally
either

491 offers to accept a Buy request from a credit card behaviour,
 492 and instructs the shop's bank to deposit the purchase amount.
 493 whereupon the shop behaviour resumes being a shop behaviour;
 494 or
 495 offers to accept a refund request in this amount, am, from a credit card [holder].
 496 It then proceeds to inform the shop's bank to withdraw the refund from its ledger and cash register,
 497 and the credit card's bank to deposit the refund into its ledger and cash register.
 498 Whereupon the shop behaviour resumes being a shop behaviour.

value

```

489 shpsi:SI: (CI-set × BI) × ... → in,out: {ch_cs[ci,si]|ci:CI•ci ∈ cis}, {ch_sb[si,bi']|bi':BI•bi'isin bis} Unit
489 shpsi((cis,bi),...) ≡
491   (sal(si,(bi,cis),...))
494   []
495   ref(si,(cis,bi),...):

489 sal: SI × (CI-set × BI) × ... → in,out: {cs[ci,si]|ci:CI•ci ∈ cis}, sb[si,bi] Unit
489 sal(si,(cis,bi),...) ≡
491   let mk_Buy(am) = [] {ch_cs[ci,si]?|ci:CI•ci ∈ cis} in
492   ch_sb[si,bi]!mk_Dep(si,am) end ;
493   shpsi((cis,bi),...)

489 ref: SI × (CI-set × BI) × ... → in,out: {ch_cs[ci,si]|ci:CI•ci ∈ cis}, {ch_sb[si,bi']|bi':BI•bi'isin bis} Unit
495 ref(si,(cis,sbi),...) ≡
495   let mk_Ref((ci,cbi,si),am) = [] {ch_cs[ci,si]?|ci:CI•ci ∈ cis} in
496   (ch_sb[si,cbi]!mk_Ref(cbi,(ci,si),am)
497   || ch_sb[si,sbi]!mk_Wdr(si,am)) end ;
498   shpsi((cis,sbi),...)

```

A.4 Discussion

TO BE WRITTEN

B

Weather Information Systems

Summary

This document reports *work in progress*. We show an example domain description. It is developed and presented as outlined in [49]. The domain being described is that of a generic weather information system. Four main endurants (i.e., aspects) of a generic weather information system are those of the weather, weather stations (collecting weather data), weather data interpretation (i.e., meteorological institute[s]), and weather forecast consumers. There are, correspondingly, two kinds of weather information: the weather data, and the weather forecasts. These forms of weather information are acted upon: the weather data interpreter (i.e., a meteorological institute) is gathering weather data; based on such interpretations the meteorological institute is “calculating” weather forecasts; and weather forecast consumers are requesting and further “interpreting” (i.e., rendering) such forecasts. Thus weather data is communicated from weather stations to the weather data interpreter; and weather forecasts are communicated from the weather data interpreter to the weather forecast consumers. It is the dual purpose of this technical report to present a domain description of the essence of generic weather information systems, and to add to the “pile” [55, 35, 41, 39, 42, 48, 47, 51] of technical reports that illustrate the use[fulness] of the principles, techniques and tools of [49].

B.1 On Weather Information Systems

B.1.1 On a Base Terminology

From Wikipedia:

499 **Weather** is the state of the atmosphere, to the degree that it is hot or cold, wet or dry, calm or stormy, clear or cloudy, atmospheric (barometric) pressure: high or low.

500 So weather is characterized by **temperature**, **humidity** (incl. **rain**, **wind** (direction, velocity, center, incl. its possible mobility), **atmospheric pressure**, etcetera.

501 By **weather information** we mean

- either weather data that characterizes the weather as defined above (Item 499),
- or weather forecast, i.e., a prediction of the state of the atmosphere for a given location and time or time interval.

502 Weather data are collected by **weather stations**. We shall here not be concerned with technical means of weather data collection.

503 **Weather forecasts** are used by forecast consumers, anyone: you and me.

504 Weather data interpretation (i.e., **forecasting**) is the science and technology of creating weather forecasts based on **time-** or **time interval-stamped weather data** and **locations**. Weather data interpretation is amongst the charges of meteorological institutes.

505 **Meteorology** is the interdisciplinary scientific study of the atmosphere.

506 An **atmosphere** (from Greek *ατμός* (atmos), meaning “vapour”, and *σφαῖρα* (sphaira), meaning “sphere”) is a layer of gases surrounding a planet or other material body, that is held in place by the gravity of that body.

507 Meteorological institutes work together with the World Meteorological Organization (WMO). Besides weather forecasting, meteorological institutes (and hence WMO) are concerned also with aviation, agricultural, nuclear, maritime, military and environmental meteorology, hydrometeorology and renewable energy.

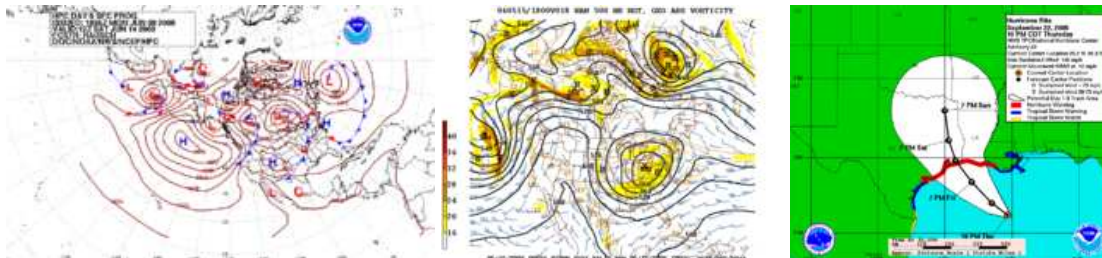
508 Agricultural meteorologists, soil scientists, agricultural hydrologists, and agronomists are persons concerned with studying the effects of weather and climate on plant distribution, crop yield, water-use efficiency, phenology of plant and animal development, and the energy balance of managed and natural ecosystems. Conversely, they are interested in the rôle of vegetation on climate and weather.

B.1.2 Some Illustrations

Weather Stations



Weather Forecasts



Forecast Consumers



B.2 Major Parts of a Weather Information System

We think of the following parts as being of concern in the kind of weather information systems that we shall analyse and describe: Figure B.1 on the next page shows one **weather** (dashed rounded corner all embracing rectangle), one central **weather data interpreter** (cum meteorological institute) seven **weather stations** (rounded corner squares), nineteen **weather forecast consumers**, and one global **clock**. All are distributed, as hinted at, in some geographical space. Figure B.2 on Page 218 shows “an orderly diagram”

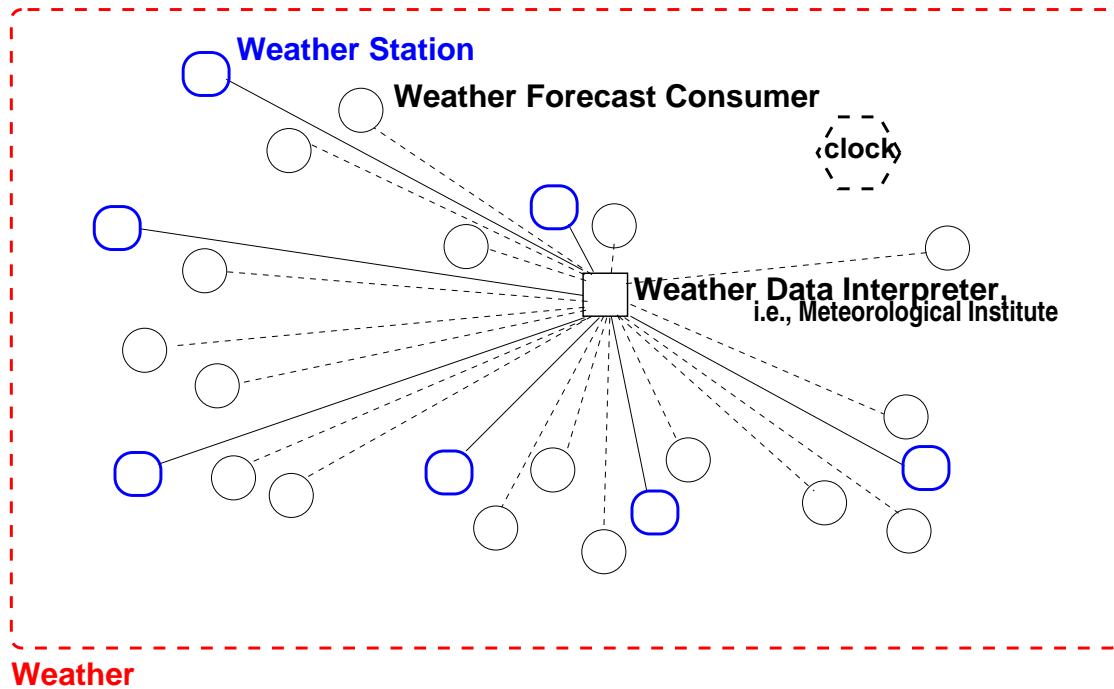


Fig. B.1. A Weather Information System

of “the same” weather information system as Figure B.1. The lines between pairs of the various parts shall indicate means communication between the pairs of (thus) connected parts. Dashed lines “crossing” bundles of these communication lines are labeled ch_{xy} . These labels, ch_{xy} , designated CSP-like channels. An input, by a weather station (wsi), of weather data from the weather (wi), is designated by the CSP expression $ch_{ws}[wi, wsi] ?$. An output, say from the weather data interpreter (wdi) to a weather forecast consumer (fci), of a forecast f , is designated by $ch_{ic}[wdi, fci] ! f$

B.3 Endurants

B.3.1 Parts and Materials

Sect. 1.3.1 Domain Description Prompt 1 on Page 15: observe_part_sorts

509 The WIS domain contains a number of sub-domains:

- a the weather, W , which we consider a material,
- b the weather stations sub-domain, WSS (a composite part),
- c the weather data interpretation sub-domain, $WDIS$ (an atomic part),
- d the weather forecast consumers sub-domain, $WFCS$ (a composite part), and
- e the (“global”) clock (an atomic part).

type

509 WIS
509a W
509b WSS
509c $WDIS$
509d $WFCS$
509e CLK

value

509a obs_material_W: $WIS \rightarrow W$
509b obs_part_WSS: $WIS \rightarrow WSS$

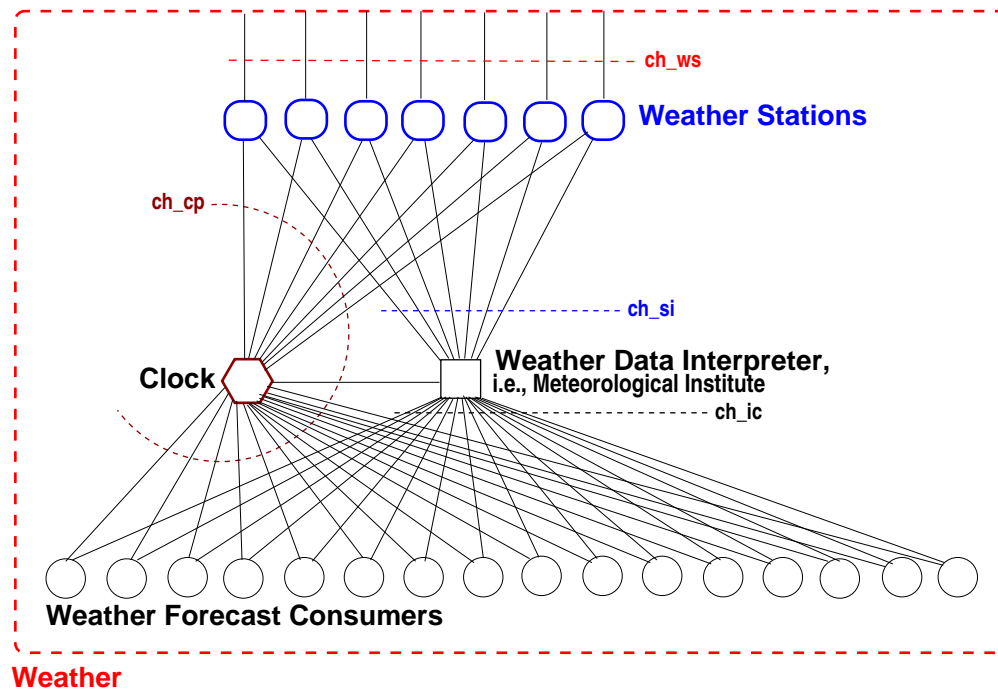


Fig. B.2. A Weather Information System Diagram

509c obs_part_WDIS: WIS \rightarrow WDIS
 509d obs_part_WFCS: WIS \rightarrow WFCS
 509e obs_part_CLK: WIS \rightarrow CLK

Sect. 1.3.1 Domain Description Prompt 2 on Page 16: observe_concrete_type

510 The weather station sub-domain, WSS, consists of a set, WSs,
 511 of atomic weather stations, WS.
 512 The weather forecast consumers sub-domain, WFCS, consists of a set, WFCs,
 513 of atomic weather forecast consumers, WFC.

type

510 WSs = WS-set
 511 WS
 512 WFCs = WFC-set
 513 WFC

value

510 obs_part_WSs: WSS \rightarrow WSs
 512 obs_part_WFCs: WFCS \rightarrow WFCs

B.3.2 Unique Identifiers

We shall consider only atomic parts.

Sect. 1.3.2 Domain Description Prompt 3 on Page 19: observe_unique_identifier

514 Every single weather station has a unique identifier.
 515 The weather data interpretation (i.e., the weather forecast “creator”) has a unique identifier.
 516 Every single weather forecast consumer has a unique identifier.
 517 The global clock has a unique identifier.

type

514 WSI
 515 WDII
 516 WFCI
 517 CLKI

value

514 uid_WSI: WS \rightarrow WSI
 515 uid_WDII: WDIS \rightarrow WDII
 516 uid_WFCI: WFC \rightarrow WFCI
 516 uid_CLKI: CLK \rightarrow CLKI

B.3.3 Mereologies

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

Sect. 1.3.3 Domain Description Prompt 4 on Page 21: observe_mereology

- 518 The mereology of weather stations is the pair of the unique clock identifier and the unique identifier of the weather data interpreter.
 519 The mereology of weather data interpreter is the triple of the unique clock identifier, set of unique identifiers of all the weather stations and the set of unique identifiers of all the weather forecast consumers.
 520 The mereology of weather forecast consumer is the the pair of the unique clock identifier and the unique identifier of the weather data interpreter.
 521 The mereology of the global clock is the triple of the set of all the unique identifiers of weather stations, the unique identifier of the weather data interpreter, and the set of all the unique identifiers of weather forecast consumers.

type

518 WSM = CLKI \times WDII
 519 WDIM = CLKI \times WSI-set \times WFCI-set
 520 WFCM = CLKI \times WDII
 521 CLKM = CLKI \times WDGI-set \times WDII \times WFCI-set

value

518 mereo_WSM: WS \rightarrow WSM
 519 mereo_WDI: WDI \rightarrow WDIM
 520 mereo_WFC: WFC \rightarrow WFCM
 521 mereo_CLK: CLK \rightarrow CLKM

B.3.4 Attributes

Sect. 1.3.4 Domain Description Prompt 5 on Page 24: observe_attributes

Clock, Time and Time-intervals

- 522 The global clock has an autonomous time attribute.
 523 Time values are further undefined, but times are considered absolute in the sense as representing some intervals since “the birth of time”, an example, concrete time could be DECEMBER 5, 2016: 07:21 AM.
 524 Time intervals are further undefined, but time intervals can be considered relative in the sense of representing a quantity elapsed between two times, examples are: 1 day 2 hours and 3 minutes, etc. When a time interval, ti , is specified it is always to be understood to designate the times from now, or from a specified time, t , until the time $t + ti$.
 525 We postulate \oplus , \ominus , and can postulate further “arithmetic” operators, and
 526 we can postulate relational operators.

type

522 TIME
 523 TI

value

522 attr_TIME: CLK \rightarrow TIME
 525 \oplus : TIME \times TI \rightarrow TIME, TI \times TI \rightarrow TI
 525 \ominus : TIME \times TI \rightarrow TIME, TIME \times TIME \rightarrow TI
 526 $=, \neq, <, \leq, \geq, >$: TIME \times TIME \rightarrow Bool, TI \times TI \rightarrow Bool, ...

We do not here define these operations and relations.

Locations

527 Locations are metric, topological spaces and can thus be considered dense spaces of three dimensional points.

528 We can speak of one location properly contained (\subset) within, or contained or equal (\subseteq), or equal ($=$), or not equal (\neq) to another location.

type

527. LOC

value

528. $\subset, \subseteq, =, \neq: \text{LOC} \times \text{LOC} \rightarrow \text{Bool}$

Weather

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

530 From the material weather one can observe its location.

type

529 $W = \text{WP-infset}$

529 WP

value

530 $\text{attr_LOC}: W \rightarrow \text{LOC}$

531 Some meteorological quantities are:

a *Humidity*,

b *Temperature*,

c *Wind* and

d *Barometric pressure*.

532 The weather has an indefinite number of attributes at any one time.

a Humidity distribution, at level (above sea) and by location,

b Temperature distribution, at level (above sea) and by location,

c Wind direction, velocity and mobility of wind center, and by location,

d Barometric pressure, and by location,

e etc., etc.

type

531a Hu

531b Te

531c Wi

531d Ba

532a $\text{HDL} = \text{LOC} \xrightarrow{\text{attr_LOC}} \text{Hu}$

532b $\text{TDL} = \text{LOC} \xrightarrow{\text{attr_LOC}} \text{Te}$

532c $\text{WDL} = \text{LOC} \xrightarrow{\text{attr_LOC}} \text{Wi}$

532d $\text{BPL} = \text{LOC} \xrightarrow{\text{attr_LOC}} \text{Ba}$

532e ...

value

532a $\text{attr_HDL}: W \rightarrow \text{HDL}$

532b $\text{attr_TDL}: W \rightarrow \text{TDL}$

532c $\text{attr_WDL}: W \rightarrow \text{WDL}$

532d $\text{attr_BPL}: W \rightarrow \text{BPL}$

532e ...

Weather Stations

533 Weather stations have static location attributes.

534 Weather stations sample the weather gathering humidity, temperature, wind, barometric pressure, and possibly other data, into time and location stamped weather data.

value

533 attr_LOC: WS \rightarrow LOC

type

534 WD :: mkWD((TIME \times LOC) \times (TDL \times HDL \times WDL \times BPL \times ...))

Weather Data Interpreter

535 There is a programmable attribute: weather data repository, wdr:WDR, of weather data, wd:WD, collected from weather stations.

536 And there is programmable attribute: weather forecast repository, wfr:WFR, of forecasts, wf:WF, disseminate-able to weather forecast consumers.

These repositories are updated when

537 received from the weather stations, respectively when

538 calculated by the weather data interpreter.

type

535 WDR

536 WFR

value

537 update_wdr: TIME \times WD \rightarrow WDR \rightarrow WDR

538 update_wfr: TIME \times WF \rightarrow WFR \rightarrow WFR

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

Weather Forecasts

539 Weather forecasts are weather forecast format-, time- and location-stamped quantities, the latter referred to as wefo:WeFo.

540 There are a definite number ($n \geq 1$) of weather forecast formats.

541 We do not presently define these various weather forecast formats.

542 They are here thought of as being requested, mkWFReq, by weather forecast consumers.

type

539 WF = WFF \times (TIME \times TI) \times LOC \times WeFo

540 WFF = WFF1 | WFF2 | ... | WFF_n

541 WFF1, WFF2, ..., WFF_n

542 WFReq :: mkWFReq(s_wff:WFF,s_ti:(TIME \times TI),s_loc:LOC)

Weather Forecast Consumer

543 There is a programmable attribute, d:D, D for display (!).

544 Displays can be rendered (RND): visualized, tabularised, made audible, translated (between languages and language dialects, ...), etc.

545 A rendered display can be “abstracted back” into its basic form.

546 Any abstracted rendered display is identical to its abstracted form.

type

543 D

544 RND

value

543 attr_D: WFC \rightarrow D

544 rndr_D: RND \times D \rightarrow D

545 abs_D: D \rightarrow D

axiom

546 $\forall d:D, r:RND \bullet \text{abs_D}(\text{rndr}(r,d)) = d$

B.4 Perdurants

B.4.1 A WIS Context

547 We postulate a given system, wis:WIS.

That system is characterized by
 548 a dynamic weather
 549 and its unique identifier,
 550 a set of weather stations
 551 and their unique identifiers,
 552 a single weather data interpreter
 553 and its unique identifier,

554 a set of weather forecast consumers

555 and their unique identifiers, and

556 a single clock

557 and its unique identifier.

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

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

value

547 wis:WIS
 548 w:W = obs_material_W(wis)
 549 wi:WI = uid_WI(w)
 550 wss:WSs = obs_part_WSs(obs_part_WSS(wis))
 551 wsis:WDGI-set = {uid_WSI(ws)|ws:WS•ws ∈ wss}
 552 wdi:WDI = obs_part_WDIS(wis)
 553 wdii:WDII = uid_WDII(wdi)
 554 wfcs:WFCs = obs_part_WFCs(obs_part_WFCS(wis))
 555 wfcis:WFI-set = {uid_WFCI(wfc)|wfc:WFC•wfc ∈ wfcs}
 556 clk:CLK = obs_part_CLK(wis)
 557 clki:CLKI = uid_CLKI(clk)
 558 is:(WI|WSI|WDII|WFCI)-set = {wi} ∪ wsis ∪ {wdii} ∪ wfcis

B.4.2 Channels

Sect. 1.4.5: Channels and Communications, Page 34

Sect. 1.4.5: Relations Between Attributes Sharing and Channels, Page 34

559 Weather stations share weather data, WD, with the weather data interpreter — so there is a set of channels, one each, “connecting” weather stations to the weather data interpreter.

560 The weather data interpreter shares weather forecast requests, WReq, and interpreted weather data (i.e., forecasts), WF, with each and every forecast consumer — so there is a set of channels, one each, “connecting” the weather data interpreter to the interpreted weather data (i.e., forecast) consumers.

561 The clock offers its current time value to each and every part, except the weather, of the WIS system.

channel

559 { ch_si[ws_i,wdii]:WD | ws_i:WSI•ws_i ∈ wsis }
 560 { ch_ic[wdii,fci]:(WReq|WF) | fci:Fci•fci ∈ fcis }
 561 { ch_cp[clki,i]:TIME | i:(WI|CLKI|WSI|WDII|WFCI)•i ∈ is }

B.4.3 WIS Behaviours

Sect. 1.4.11: Process Schema I: Abstract is_composite(p), Page 39

Sect. 1.4.11: Process Schema II: Concrete is_concrete(p), Page 40

562 WIS behaviour, wis_beh, is the

563 parallel composition of all the weather station behaviours, in parallel with the

564 weather data interpreter behaviour, in parallel with the

565 parallel composition of all the weather forecast consumer behaviours, in parallel with the

566 clock behaviour.

value

```

562 wis_beh: Unit → Unit
562 wis_beh() ≡
563   || { ws_beh(uid_WSI(ws),mereo_WS(ws),...) | ws:WS•ws ∈ wss } ||
564   || wdi_beh(uid_WDI(wdi),mereo_WDI(wdi),...)(wd_rep,wf_rep) ||
565   || { wfc_beh(uid_WFC(wfc),mereo_WDG(wfc),...) | wfc:WFC•wfc ∈ wfcs } ||
566   clk_beh(uid_CLKI(clk),mereo_CLK(clk),...)("December 5, 2016: 07:21 am")

```

B.4.4 Clock

Sect. 1.4.11: Process Schema III: is_atomic(p), Page 40

567 The clock behaviour has a programmable attribute, t.
 568 It repeatedly offers its current time to any part of the WIS system.
 It nondeterministically internally “cycles” between
 569 retaining its current time, or
 570 increment that time with a “small” time interval, δ , or
 571 offering the current time to a requesting part.

value

```

567. clk_beh: clki:CLKI × clk_m:CLKM → TIME →
568.   out {ch_cp[clki,i] | i:(WSI|WDI|WFCI)•i ∈ wsis ∪ {wdii} ∪ wfcs } Unit
567. clk_beh(clki,is)(t) ≡
569.   clk_beh(clki,is)(t)
570.   □ clk_beh(clki,is)(t ⊕ δ)
571.   □ ( □ { ch_cp[clki,i] ! t | i:(WSI|WDI|WFCI)•i ∈ is } ; clk_beh(clki,is)(t) )

```

B.4.5 Weather Station

Sect. 1.4.11: Process Schema III: is_atomic(p), Page 40

572 The weather station behaviour communicates with the global clock and the weather data interpreter.
 573 The weather station behaviour simply “cycles” between sampling the weather, reporting its findings to the weather
 data interpreter and resume being that overall behaviour.
 574 The weather station time-stamp “sample” the weather (i.e., meteorological information).
 575 The meteorological information obtained is analysed with respect to temperature (distribution etc.),
 576 humidity (distribution etc.),
 577 wind (distribution etc.),
 578 barometric pressure (distribution etc.), etcetera,
 579 and this is time-stamp and location aggregated (mkWD) and “sent” to the (central ?) weather data interpreter,
 580 whereupon the weather data generator behaviour resumes.

value

```

572 ws_beh: wsi:WSI × (clki,wi,wdii):WDGM × (LOC × ...) →
572   in ch_cp[clki,wsi] out ch_gi[wsi,wdii] Unit
573 ws_beh(wsi,(clki,wi,wdii),(loc,...)) ≡
575   let tdl = attr_TDL(w),
576   hdl = attr_HDL(w),
577   wdl = attr_WDL(w),
578   bpl = attr_BPL(w), ... in
579   ch_gi[wsi,wdii] ! mkWD((ch_cp[clki,wsi] ?,loc),(tdl,hdl,wdl,bpl,...)) end ;
580   wdg_beh(wsi,(clki,wi,wdii),(loc,...))

```

B.4.6 Weather Data Interpreter

Sect. 1.4.11: Process Schema III: *is_atomic(p)*, Page 40

581 The weather data interpreter behaviour communicates with the global clock, all the weather stations and all the weather forecast consumers.

582 The weather data interpreter behaviour non-deterministically internally (\square) chooses to

583 either collect weather data,

584 or calculate some weather forecast,

585 or disseminate a weather forecast.

value

581 wdi_beh: $\text{wdii:WDII} \times (\text{clki, wsis, wfcis}): \text{WDIM} \times \dots \rightarrow (\text{WD_Rep} \times \text{WF_Rep}) \rightarrow$

581 **in** $\text{ch_cp}[\text{clki, wdii}], \{ \text{ch_si}[\text{wsi, wdii}] \mid \text{wsi:WSI} \bullet \text{wsi} \in \text{wsis} \},$

581 **out** $\{ \text{ch_ic}[\text{wdii, wfcis}] \mid \text{wfcis:WFCI} \bullet \text{wfcis} \in \text{wfcis} \}$ **Unit**

581 wdi_beh($\text{wdii}, (\text{clki, wsis, wfcis}), \dots$)(wd_rep, wf_rep) \equiv

583 $\text{collect_wd}(\text{wdii}, (\text{clki, wsis, wfcis}), \dots)(\text{wd_rep, wf_rep})$

582 \square

584 $\text{calculate_wf}(\text{wdii}, (\text{clki, wsis, wfcis}), \dots)(\text{wd_rep, wf_rep})$

582 \square

585 $\text{disseminate_wf}(\text{wdii}, (\text{clki, wsis, wfcis}), \dots)(\text{wd_rep, wf_rep})$

collect_wd

Sect. 1.4.11: Process Schema IV: Core Processes (I), Page 41

Sect. 1.4.11: Process Schema V: Core Processes (II), Page 41

586 The collect weather data behaviour communicates with the global clock and all the weather stations – but “passes-on” the capability to communicate with all of the weather forecast consumers.

587 The collect weather data behaviour

588 non-deterministically externally offers to accept weather data from some weather station,

589 updates the weather data repository with a time-stamped version of that weather data,

590 and resumes being a weather data interpreter behaviour, now with an updated weather data repository.

value

586 collect_wd: $\text{wdii:WDII} \times (\text{clki, wsis, wfcis}): \text{WDIM} \times \dots$

586 $\rightarrow (\text{WD_Rep} \times \text{WF_Rep}) \rightarrow$

586 **in** $\text{ch_cp}[\text{clki, wdii}], \{ \text{ch_si}[\text{wsi, wdii}] \mid \text{wsi:WSI} \bullet \text{wsi} \in \text{wsis} \},$

586 **out** $\{ \text{ch_ic}[\text{wdii, wfcis}] \mid \text{wfcis:WFCI} \bullet \text{wfcis} \in \text{wfcis} \}$ **Unit**

587 collect_wd($\text{wdii}, (\text{clki, wsis, wfcis}), \dots$)(wd_rep, wf_rep) \equiv

588 **let** $((\text{ti, loc}), (\text{hdl, tdl, wdl, bpl}, \dots)) = \square \{ \text{wsi}[\text{wsi, wdii}]? \mid \text{wsi:WSI} \bullet \text{wsi} \in \text{wsis} \}$ **in**

589 **let** $\text{wd_rep}' = \text{update_wdr}(\text{ch_cp}[\text{clki, wdii}], ((\text{ti, loc}), (\text{hdl, tdl, wdl, bpl}, \dots)))(\text{wd_rep})$ **in**

590 $\text{wdi_beh}(\text{wdii}, (\text{clki, wsis, wfcis}), \dots)(\text{wd_rep}', \text{wf_rep})$ **end end**

calculate_wf

Sect. 1.4.11: Process Schema IV: Core Processes (I), Page 41

Sect. 1.4.11: Process Schema V: Core Processes (II), Page 41

591 The calculate forecast behaviour communicates with the global clock – but “passes-on” the capability to communicate with all of weather stations and the weather forecast consumers.

592 The calculate forecast behaviour

593 non-deterministically internally chooses a forecast type from among a indefinite set of such,

594 and a current or “future” time-interval,

595 whereupon it calculates the weather forecast and updates the weather forecast repository,

596 and then resumes being a weather data interpreter behaviour now with the weather forecast repository updated with the calculated forecast.

```

value
591 calculate_wf: wdii:WDII  $\times$  (clki,wsis,wfcis):WDIM  $\times$  ...  $\rightarrow$  (WD_Rep  $\times$  WF_Rep)  $\rightarrow$ 
591   in ch_cp[clki,wdii], { ch_si[wsis,wdii] | wsi:WSI  $\bullet$  wsi  $\in$  wsis },
591   out { ch_ic[wdii,wfci] | wfci:WFCI  $\bullet$  wfci  $\in$  wfcis } Unit
592 calculate_wf(wdii,(clki,wsis,wfcis),...)(wd_rep,wf_rep)  $\equiv$ 
593   let tf:WWF = ft1  $\sqcap$  ft2  $\sqcap$  ...  $\sqcap$  ftn,
594   ti:(TIME  $\times$  TIVAL)  $\bullet$  toti  $\geq$  ch_cp[clki,wdii] ? in
595   let wf_rep' = update_wfr(calc_wf(tf,ti)(wf_rep)) in
596   wdi_beh(wdii,(clki,wsis,wfcis),...)(wd_rep,wf_rep') end end

```

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

```

value
597. calc_wf: WFF  $\times$  (TIME  $\times$  TI)  $\rightarrow$  WFRep  $\rightarrow$  WF
597. calc_wf(tf,ti)(wf_rep)  $\equiv$  ,,

```

disseminate_wf

Sect. 1.4.11: Process Schema IV: Core Processes (I), Page 41

Sect. 1.4.11: Process Schema V: Core Processes (II), Page 41

598 The disseminate weather forecast behaviour communicates with the global clock and all the weather forecast consumers – but “passes-on” the capability to communicate with all of weather stations.

599 The disseminate weather forecast behaviour non-deterministically externally offers to received a weather forecast request from any of the weather forecast consumers, wfci, that request is for a specific format forecast, tf, and either for a specific time or for a time-interval, toti, as well as for a specific location, loc.

600 The disseminate weather forecast behaviour retrieves an appropriate forecast and

601 sends it to the requesting consumer –

602 whereupon the disseminate weather forecast behaviour resumes being a weather data interpreter behaviour

```

value
598 disseminate_wf: wdii:WDII  $\times$  (clki,wsis,wfcis):WDIM  $\times$  ...  $\rightarrow$  (WD_Rep  $\times$  WF_Rep)  $\rightarrow$ 
598   in ch_cp[clki,wdii] in,out { ch_ic[wdii,wfci] | wfci:WFCI  $\bullet$  wfci  $\in$  wfcis } Unit
598 disseminate_wf(wdii,(clki,wsis,wfcis),...)(wd_rep,wf_rep)  $\equiv$ 
599   let mkReqWF((tf,toti,loc),wfci) =  $\sqcap$  { ch_ic[wdii,wfci] ? | wfci:WFCI  $\bullet$  wfci  $\in$  wfcis } in
600   let wf = retr_WF((tf,toti,loc),wf_rep) in
601   ch_ic[wdii,wfci] ! wf ;
602   disseminate_wf(wdii,(clki,wsis,wfcis),...)(wd_rep,wf_rep) end end

```

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

603. retr_WF: (WFF \times (TIME \times TI) \times LOC) \times WFRep \rightarrow WF

603. retr_WF((tf,toti,loc),wf_rep) \equiv ...

We could have included, in our model, the time-stamping of receipt (formula Item 599) of requests, and the time-stamping of delivery of requested forecast in which case we would insert ch_cp[clki,wdii]? at respective points in formula Items 599 and 601.

B.4.7 Weather Forecast Consumer

Sect. 1.4.11: Process Schema IV: Core Processes (I), Page 41

Sect. 1.4.11: Process Schema V: Core Processes (II), Page 41

604 The weather forecast consumer communicates with the global clock and the weather data interpreter.

605 The weather forecast consumer behaviour

606 nondeterministically internally either
 607 selects a suitable weather cast format, tf ,
 608 selects a suitable location, loc' , and
 609 selects, $toti$, a suitable time (past, present or future) or a time interval (that is supposed to start when forecast request is received by the weather data interpreter.
 610 With a suitable formatting of this triple, $mkReqWF(tf, loc', toti)$, the weather forecast consumer behaviour “outputs” a request for a forecast to the weather data interpreter (first “half” of formula Item 609) whereupon it awaits (;) its response (last “half” of formula Item 609) which is a weather forecast, wf ,
 611 whereupon the weather forecast consumer behaviour resumes being that behaviour with it programmable attribute, d , being replaced by the received forecast suitably annotated;
 606 or the weather forecast consumer behaviour
 612 edits a display
 613 and resumes being a weather forecast consumer behaviour with the edited programmable attribute, d' .

value

```

604 wfc_beh: wfc:WFCI  $\times$  (clki,wdii):WFCM  $\times$  (LOC  $\times$  ...)  $\rightarrow$  D  $\rightarrow$ 
604   in ch_cp[clki,wfc],
604   in,out { ch_ic[wdii,wfc] | wfc:WFCI $\cdot$ wfc  $\in$  wfcis } Unit
605 wfc_beh(wfc,(clki,wdii),(loc,...))(d)  $\equiv$ 
607   let tf = tf1  $\sqcap$  tf2  $\sqcap$  ...  $\sqcap$  tfn,
608   loc':LOC  $\cdot$  loc' = loc  $\vee$  loc'  $\neq$  loc,
609   (t,ti):(TIME $\times$ TI)  $\cdot$  ti  $\geq$  0 in
610   let wf = (ch_ic[wdii,wfc] ! mkReqWF(tf,loc',(t,ti))) ; ch_ic[wdii,wfc] ? in
611   wfc_beh(wfc,(clki,wdii),(loc,...))((tf,loc',(t,ti)),wf) end end
606    $\sqcap$ 
612   let d':D { \EQ } rndr\_D(d,{\DOTDOTDOT}) in
613   wfc_beh(wfc,(clki,wdii),(loc,...))(d') end

```

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

B.5 Conclusion

B.5.1 Reference to Similar Work

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

B.5.2 What Have We Achieved ?

TO BE WRITTEN

B.5.3 What Needs to be Done Next ?

TO BE WRITTEN

B.5.4 Acknowledgements

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

Miscellaneous

C

RSL: The RAISE Specification Language – A Primer

C.1 Type Expressions

Type expressions are expressions whose value are types, that is, possibly infinite sets of values (of “that” type).

C.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, -1, 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" |

C.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.

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 the following are type expressions:

- | | |
|--|--|
| [7] A-set | [13] $A \rightarrow B$ |
| [8] A-infset | [14] $A \rightsquigarrow B$ |
| [9] $A \times B \times \dots \times C$ | [15] (A) |
| [10] A^* | [16] $A \mid B \mid \dots \mid C$ |
| [11] A^o | [17] mk_id(sel_a:A,...,sel_b:B) |
| [12] $A \multimap B$ | [18] sel_a:A ... sel_b:B |

The following the meaning of the atomic and the composite type expressions:

- 1 The Boolean type of truth values **false** and **true**.
- 2 The integer type on integers ..., -2, -1, 0, 1, 2,
- 3 The natural number type of positive integer values 0, 1, 2, ...

- 4 The real number type of real values, i.e., values whose numerals can be written as an integer, followed by a period (“.”), followed by a natural number (the fraction).
- 5 The character type of character values “a”, “bb”, ...
- 6 The text type of character string values “aa”, “aaa”, ..., “abc”, ...
- 7 The set type of finite cardinality set values.
- 8 The set type of infinite and finite cardinality set values.
- 9 The Cartesian type of Cartesian values.
- 10 The list type of finite length list values.
- 11 The list type of infinite and finite length list values.
- 12 The map type of finite definition set map values.
- 13 The function type of total function values.
- 14 The function type of partial function values.
- 15 In (A) A is constrained to be:
 - either a Cartesian $B \times C \times \dots \times D$, 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., $(A \rightarrow B)$, or (A^*) -set, or $(A\text{-set})$ list, or $(A|B) \rightarrow (C|D|(E \rightarrow F))$, 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.

Sorts and Observer Functions

type

A, B, C, ..., D

value

obs_B: $A \rightarrow B$, obs_C: $A \rightarrow C$, ..., obs_D: $A \rightarrow D$

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

$A = B \times C \times \dots \times D$

C.2 Type Definitions

C.2.1 Concrete Types

Types can be concrete in which case the structure of the type is specified by type expressions:

type

$A = \text{Type_expr}$

Some schematic type definitions are:

- [19] $\text{Type_name} = \text{Type_expr} \text{ /* without | s or subtypes */}$
- [20] $\text{Type_name} = \text{Type_expr}_1 \mid \text{Type_expr}_2 \mid \dots \mid \text{Type_expr}_n$
- [21] $\text{Type_name} ==$

$$\text{mk_id}_1(\text{s_a1}:\text{Type_name_a1}, \dots, \text{s_ai}:\text{Type_name_ai}) \mid$$

$$\dots \mid$$

$$\text{mk_id}_n(\text{s_z1}:\text{Type_name_z1}, \dots, \text{s_zk}:\text{Type_name_zk})$$
- [22] $\text{Type_name} :: \text{sel_a}:\text{Type_name_a} \dots \text{sel_z}:\text{Type_name_z}$
- [23] $\text{Type_name} = \{ \mid v:\text{Type_name}' \cdot \mathcal{P}(v) \mid \}$

where a form of [20]–[21] 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

```

C.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

$$A = \{ | b:B \bullet \mathcal{P}(b) | \}$$

C.2.3 Sorts — Abstract Types

Types can be (abstract) sorts in which case their structure is not specified:

type

$$A, B, \dots, C$$

C.3 The RSL Predicate Calculus

C.3.1 Propositional Expressions

Let identifiers (or propositional expressions) a, b, \dots, c designate Boolean values (**true** or **false** [or **chaos**]). Then:

false, true

$$a, b, \dots, c \sim a, a \wedge b, a \vee b, a \Rightarrow b, a = b, a \neq b$$

are propositional expressions having Boolean values. $\sim, \wedge, \vee, \Rightarrow, =$ and \neq are Boolean connectives (i.e., operators). They can be read as: *not, and, or, if then* (or *implies*), *equal* and *not equal*.

C.3.2 Simple Predicate Expressions

Let identifiers (or propositional expressions) a, b, \dots, c designate Boolean values, let x, y, \dots, z (or term expressions) designate non-Boolean values and let i, j, \dots, k designate number values, then:

false, true

$$a, b, \dots, c$$

$$\sim a, a \wedge b, a \vee b, a \Rightarrow b, a = b, a \neq b$$

$$x = y, x \neq y,$$

$$i < j, i \leq j, i \geq j, i \neq j, i \geq j, i > j$$

are simple predicate expressions.

C.3.3 Quantified Expressions

Let X, Y, \dots, 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:

$$\begin{aligned} &\forall x:X \cdot \mathcal{P}(x) \\ &\exists y:Y \cdot \mathcal{Q}(y) \\ &\exists ! z:Z \cdot \mathcal{R}(z) \end{aligned}$$

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.

C.4 Concrete RSL Types: Values and Operations

C.4.1 Arithmetic

type

Nat, Int, Real

value

$+, -, *: \text{Nat} \times \text{Nat} \rightarrow \text{Nat} \mid \text{Int} \times \text{Int} \rightarrow \text{Int} \mid \text{Real} \times \text{Real} \rightarrow \text{Real}$
 $/: \text{Nat} \times \text{Nat} \rightarrow \text{Nat} \mid \text{Int} \times \text{Int} \rightarrow \text{Int} \mid \text{Real} \times \text{Real} \rightarrow \text{Real}$
 $<, \leq, =, \neq, \geq, > (\text{Nat} \mid \text{Int} \mid \text{Real}) \rightarrow (\text{Nat} \mid \text{Int} \mid \text{Real})$

C.4.2 Set Expressions

Set Enumerations

Let the below a ’s denote values of type A , then the below designate simple set enumerations:

$$\begin{aligned} &\{\{\}, \{a\}, \{e_1, e_2, \dots, e_n\}, \dots\} \in \mathbf{A\text{-}set} \\ &\{\{\}, \{a\}, \{e_1, e_2, \dots, e_n\}, \dots, \{e_1, e_2, \dots\}\} \in \mathbf{A\text{-}infset} \end{aligned}$$

Set Comprehension

The expression, last line below, to the right of the \equiv , expresses set comprehension. The expression “builds” the set of values satisfying the given predicate. It is abstract in the sense that it does not do so by following a concrete algorithm.

type

A, B

P = A \rightarrow Bool

Q = A \rightarrow B

value

comprehend: A-infset \times P \times Q \rightarrow B-infset
comprehend(s, P, Q) \equiv { Q(a) | a:A \cdot a \in s \wedge P(a) }

C.4.3 Cartesian Expressions

Cartesian Enumerations

Let e range over values of Cartesian types involving A, B, \dots, C , then the below expressions are simple Cartesian enumerations:

type

A, B, ..., C

A \times B \times ... \times C

value

(e1, e2, ..., en)

C.4.4 List Expressions

List Enumerations

Let a range over values of type A , then the below expressions are simple list enumerations:

$$\begin{aligned} \{ \langle \rangle, \langle e \rangle, \dots, \langle e_1, e_2, \dots, e_n \rangle, \dots \} &\in A^* \\ \{ \langle \rangle, \langle e \rangle, \dots, \langle e_1, e_2, \dots, e_n \rangle, \dots, \langle e_1, e_2, \dots, e_n, \dots \rangle, \dots \} &\in A^\omega \\ \langle a_i \dots a_j \rangle \end{aligned}$$

The last line above assumes a_i and a_j to be integer-valued expressions. It then expresses the set of integers from the value of e_i to and including the value of e_j . If the latter is smaller than the former, then the list is empty.

List Comprehension

The last line below expresses list comprehension.

type
 $A, B, P = A \rightarrow \mathbf{Bool}, Q = A \rightarrow B$
value
 comprehend: $A^\omega \times P \times Q \rightarrow B^\omega$
 $\text{comprehend}(l, P, Q) \equiv \langle Q(l(i)) \mid i \text{ in } \langle 1..len\ l \rangle \bullet P(l(i)) \rangle$

C.4.5 Map Expressions

Map Enumerations

Let (possibly indexed) u and v range over values of type T_1 and T_2 , respectively, then the below expressions are simple map enumerations:

type
 T_1, T_2
 $M = T_1 \rightarrow_m T_2$
value
 $u, u_1, u_2, \dots, u_n: T_1, v, v_1, v_2, \dots, v_n: T_2$
 $[], [u \mapsto v], \dots, [u_1 \mapsto v_1, u_2 \mapsto v_2, \dots, u_n \mapsto v_n]$ all $\in M$

Map Comprehension

The last line below expresses map comprehension:

type
 U, V, X, Y
 $M = U \rightarrow_m V$
 $F = U \rightarrow X$
 $G = V \rightarrow Y$
 $P = U \rightarrow \mathbf{Bool}$
value
 comprehend: $M \times F \times G \times P \rightarrow (X \rightarrow_m Y)$
 $\text{comprehend}(m, F, G, P) \equiv [F(u) \mapsto G(m(u)) \mid u: U \bullet u \in \mathbf{dom}\ m \wedge P(u)]$

C.4.6 Set Operations

Set Operator Signatures

value

- 19 \in : $A \times A\text{-infset} \rightarrow \text{Bool}$
- 20 \notin : $A \times A\text{-infset} \rightarrow \text{Bool}$
- 21 \cup : $A\text{-infset} \times A\text{-infset} \rightarrow A\text{-infset}$
- 22 \cup : $(A\text{-infset})\text{-infset} \rightarrow A\text{-infset}$
- 23 \cap : $A\text{-infset} \times A\text{-infset} \rightarrow A\text{-infset}$
- 24 \cap : $(A\text{-infset})\text{-infset} \rightarrow A\text{-infset}$
- 25 \setminus : $A\text{-infset} \times A\text{-infset} \rightarrow A\text{-infset}$
- 26 \subset : $A\text{-infset} \times A\text{-infset} \rightarrow \text{Bool}$
- 27 \subseteq : $A\text{-infset} \times A\text{-infset} \rightarrow \text{Bool}$
- 28 $=$: $A\text{-infset} \times A\text{-infset} \rightarrow \text{Bool}$
- 29 \neq : $A\text{-infset} \times A\text{-infset} \rightarrow \text{Bool}$
- 30 **card**: $A\text{-infset} \rightarrow \text{Nat}$

Set Examples

examples

- $a \in \{a,b,c\}$
- $a \notin \{\}, a \notin \{b,c\}$
- $\{a,b,c\} \cup \{a,b,d,e\} = \{a,b,c,d,e\}$
- $\cup\{\{a\},\{a,bb\},\{a,d\}\} = \{a,b,d\}$
- $\{a,b,c\} \cap \{c,d,e\} = \{c\}$
- $\cap\{\{a\},\{a,bb\},\{a,d\}\} = \{a\}$
- $\{a,b,c\} \setminus \{c,d\} = \{a,bb\}$
- $\{a,bb\} \subset \{a,b,c\}$
- $\{a,b,c\} \subseteq \{a,b,c\}$
- $\{a,b,c\} = \{a,b,c\}$
- $\{a,b,c\} \neq \{a,bb\}$
- card** $\{\} = 0$, **card** $\{a,b,c\} = 3$

Informal Explication

- 19 \in : The membership operator expresses that an element is a member of a set.
- 20 \notin : The nonmembership operator expresses that an element is not a member of a set.
- 21 \cup : The infix union operator. When applied to two sets, the operator gives the set whose members are in either or both of the two operand sets.
- 22 \cup : The distributed prefix union operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.
- 23 \cap : The infix intersection operator. When applied to two sets, the operator gives the set whose members are in both of the two operand sets.
- 24 \cap : The prefix distributed intersection operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.
- 25 \setminus : The set complement (or set subtraction) operator. When applied to two sets, the operator gives the set whose members are those of the left operand set which are not in the right operand set.
- 26 \subseteq : The proper subset operator expresses that all members of the left operand set are also in the right operand set.
- 27 \subset : The proper subset operator expresses that all members of the left operand set are also in the right operand set, and that the two sets are not identical.
- 28 $=$: The equal operator expresses that the two operand sets are identical.
- 29 \neq : The nonequal operator expresses that the two operand sets are *not* identical.
- 30 **card**: The cardinality operator gives the number of elements in a finite set.

Set Operator Definitions

The operations can be defined as follows (\equiv is the definition symbol):

```

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  $s$  */

```

C.4.7 Cartesian Operations

<pre> type A, B, C g0: G0 = A × B × C g1: G1 = (A × B × C) g2: G2 = (A × B) × C g3: G3 = A × (B × C) </pre>	<pre> (va,vb,vc):G1 ((va,vb),vc):G2 (va3,(vb3,vc3)):G3 </pre>
<pre> value va:A, vb:B, vc:C, vd:D (va,vb,vc):G0, </pre>	<pre> 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 </pre>

C.4.8 List Operations

List Operator Signatures

```

value
hd:  $A^\omega \xrightarrow{\sim} A$ 
tl:  $A^\omega \xrightarrow{\sim} A^\omega$ 
len:  $A^\omega \xrightarrow{\sim} \mathbf{Nat}$ 
inds:  $A^\omega \rightarrow \mathbf{Nat-infset}$ 
elems:  $A^\omega \rightarrow \mathbf{A-infset}$ 
 $\langle \cdot \rangle$ :  $A^\omega \times \mathbf{Nat} \xrightarrow{\sim} A$ 
 $\wedge$ :  $A^* \times A^* \times A^* \times A^* \times A^* \rightarrow \mathbf{Bool}$ 

```

List Operation Examples

```

examples
hd  $\langle a1, a2, \dots, am \rangle = a1$ 
tl  $\langle a1, a2, \dots, am \rangle = \langle a2, \dots, am \rangle$ 
len  $\langle a1, a2, \dots, am \rangle = m$ 
inds  $\langle a1, a2, \dots, am \rangle = \{1, 2, \dots, m\}$ 
elems  $\langle a1, a2, \dots, am \rangle = \{a1, a2, \dots, am\}$ 
 $\langle a1, a2, \dots, am \rangle(i) = ai$ 
 $\langle a, b, c \rangle \wedge \langle a, b, d \rangle = \langle a, b, c, a, b, d \rangle$ 
 $\langle a, b, c \rangle = \langle a, b, c \rangle$ 
 $\langle a, b, c \rangle \neq \langle a, b, d \rangle$ 

```

Informal Explication

- **hd**: Head gives the first element in a nonempty list.
- **tl**: Tail gives the remaining list of a nonempty list when Head is removed.
- **len**: Length gives the number of elements in a finite list.
- **inds**: Indices give the set of indices from 1 to the length of a nonempty list. For empty lists, this set is the empty set as well.
- **elems**: Elements gives the possibly infinite set of all distinct elements in a list.
- $\ell(i)$: Indexing with a natural number, i larger than 0, into a list ℓ having a number of elements larger than or equal to i , gives the i th element of the list.
- $\hat{\ }:$ Concatenates two operand lists into one. The elements of the left operand list are followed by the elements of the right. The order with respect to each list is maintained.
- $=$: The equal operator expresses that the two operand lists are identical.
- \neq : The nonequal operator expresses that the two operand lists are *not* identical.

The operations can also be defined as follows:

List Operator Definitions

value

$\text{is_finite_list}: A^{\omega} \rightarrow \text{Bool}$

$\text{len } q \equiv$
 $\text{case is_finite_list}(q) \text{ of}$
 $\text{true} \rightarrow \text{if } q = \langle \rangle \text{ then } 0 \text{ else } 1 + \text{len tl } q \text{ end,}$
 $\text{false} \rightarrow \text{chaos end}$

$\text{inds } q \equiv$
 $\text{case is_finite_list}(q) \text{ of}$
 $\text{true} \rightarrow \{ i \mid i:\text{Nat} \cdot 1 \leq i \leq \text{len } q \},$
 $\text{false} \rightarrow \{ i \mid i:\text{Nat} \cdot i \neq 0 \} \text{ end}$

$\text{elems } q \equiv \{ q(i) \mid i:\text{Nat} \cdot i \in \text{inds } q \}$

$q(i) \equiv$
 $\text{if } i=1$
 then
 $\text{if } q \neq \langle \rangle$
 $\text{then let } a:A, q':Q \cdot q = \langle a \rangle \hat{\ } q' \text{ in } a \text{ end}$
 else chaos end
 $\text{else } q(i-1) \text{ end}$

$fq \hat{\ } iq \equiv$
 $\langle \text{if } 1 \leq i \leq \text{len } fq \text{ then } fq(i) \text{ else } iq(i - \text{len } fq) \text{ end}$
 $\mid i:\text{Nat} \cdot \text{if } \text{len } iq \neq \text{chaos} \text{ then } i \leq \text{len } fq + \text{len } iq \text{ end} \rangle$
 $\text{pre is_finite_list}(fq)$

$iq' = iq'' \equiv$
 $\text{inds } iq' = \text{inds } iq'' \wedge \forall i:\text{Nat} \cdot i \in \text{inds } iq' \Rightarrow iq'(i) = iq''(i)$

$iq' \neq iq'' \equiv \sim(iq' = iq'')$

C.4.9 Map Operations

Map Operator Signatures and Map Operation Examples

value

$$m(a): M \rightarrow A \xrightarrow{\sim} B, m(a) = b$$

$$\text{dom}: M \rightarrow \mathbf{A}\text{-infset} \text{ [domain of map]}$$

$$\text{dom} [a1 \mapsto b1, a2 \mapsto b2, \dots, an \mapsto bn] = \{a1, a2, \dots, an\}$$

$$\text{rng}: M \rightarrow \mathbf{B}\text{-infset} \text{ [range of map]}$$

$$\text{rng} [a1 \mapsto b1, a2 \mapsto b2, \dots, an \mapsto bn] = \{b1, b2, \dots, bn\}$$

$$\dagger: M \times M \rightarrow M \text{ [override extension]}$$

$$[a \mapsto b, a' \mapsto bb', a'' \mapsto bb''] \dagger [a' \mapsto bb'', a'' \mapsto bb'] = [a \mapsto b, a' \mapsto bb'', a'' \mapsto bb']$$

$$\cup: M \times M \rightarrow M \text{ [merge } \cup]$$

$$[a \mapsto b, a' \mapsto bb', a'' \mapsto bb''] \cup [a''' \mapsto bb'''] = [a \mapsto b, a' \mapsto bb', a'' \mapsto bb'', a''' \mapsto bb''']$$

$$\setminus: M \times \mathbf{A}\text{-infset} \rightarrow M \text{ [restriction by]}$$

$$[a \mapsto b, a' \mapsto bb', a'' \mapsto bb''] \setminus \{a\} = [a' \mapsto bb', a'' \mapsto bb'']$$

$$/: M \times \mathbf{A}\text{-infset} \rightarrow M \text{ [restriction to]}$$

$$[a \mapsto b, a' \mapsto bb', a'' \mapsto bb''] / \{a', a''\} = [a \mapsto b]$$

$$=, \neq: M \times M \rightarrow \mathbf{Bool}$$

$$\circ: (A \xrightarrow{m_1} B) \times (B \xrightarrow{m_2} C) \rightarrow (A \xrightarrow{m_1 \circ m_2} C) \text{ [composition]}$$

$$[a \mapsto b, a' \mapsto bb'] \circ [bb \mapsto c, bb' \mapsto c', bb'' \mapsto c''] = [a \mapsto c, a' \mapsto c']$$

Map Operation Explication

- $m(a)$: Application gives the element that a maps to in the map m .
- **dom**: Domain/Definition Set gives the set of values which *maps to* in a map.
- **rng**: Range/Image Set gives the set of values which *are mapped to* in a map.
- \dagger : 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.
- \cup : Merge. When applied to two operand maps, it gives a merge of these maps.
- \setminus : 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.
- \neq : The nonequal operator expresses that the two operand maps are *not* identical.
- \circ : Composition. When applied to two operand maps, it gives the map from definition set elements of the left operand map, m_1 , to the range elements of the right operand map, m_2 , such that if a is in the definition set of m_1 and maps into b , and if b is in the definition set of m_2 and maps into c , then a , in the composition, maps into c .

Map Operation Redefinitions

The map operations can also be defined as follows:

value

$$\text{rng } m \equiv \{ m(a) \mid a:A \bullet a \in \text{dom } m \}$$

$$m1 \dagger m2 \equiv$$

$$[a \mapsto b \mid a:A, b:B \bullet$$

$$a \in \text{dom } m1 \setminus \text{dom } m2 \wedge bb = m1(a) \vee a \in \text{dom } m2 \wedge bb = m2(a)]$$

$$m1 \cup m2 \equiv [a \mapsto b \mid a:A, b:B \bullet$$

$$a \in \text{dom } m1 \wedge bb = m1(a) \vee a \in \text{dom } m2 \wedge bb = m2(a)]$$

$$m \setminus s \equiv [a \mapsto m(a) \mid a:A \bullet a \in \text{dom } m \setminus s]$$

$$m / s \equiv [a \mapsto m(a) \mid a:A \bullet a \in \text{dom } m \cap s]$$

$$m1 = m2 \equiv$$

$$\text{dom } m1 = \text{dom } m2 \wedge \forall a:A \bullet a \in \text{dom } m1 \Rightarrow m1(a) = m2(a)$$

$$m1 \neq m2 \equiv \sim(m1 = m2)$$

$$m^\circ n \equiv$$

$$[a \mapsto c \mid a:A, c:C \bullet a \in \text{dom } m \wedge c = n(m(a))]$$

$$\text{pre rng } m \subseteq \text{dom } n$$

C.5 λ -Calculus + Functions

C.5.1 The λ -Calculus Syntax

type /* A BNF Syntax: */

$$\langle L \rangle ::= \langle V \rangle \mid \langle F \rangle \mid \langle A \rangle \mid (\langle A \rangle)$$

$$\langle V \rangle ::= /* \text{variables, i.e. identifiers} */$$

$$\langle F \rangle ::= \lambda \langle V \rangle \bullet \langle L \rangle$$

$$\langle A \rangle ::= (\langle L \rangle \langle L \rangle)$$

value /* Examples */

$$\langle L \rangle: e, f, a, \dots$$

$$\langle V \rangle: x, \dots$$

$$\langle F \rangle: \lambda x \bullet e, \dots$$

$$\langle A \rangle: f a, (f a), f(a), (f)(a), \dots$$

C.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).

C.5.3 Substitution

In RSL, the following rules for substitution apply:

- **subst** $([N/x]x) \equiv N$;
- **subst** $([N/x]a) \equiv a$,
for all variables $a \neq x$;
- **subst** $([N/x](P Q)) \equiv (\text{subst}([N/x]P) \text{ subst}([N/x]Q))$;
- **subst** $([N/x](\lambda x \bullet P)) \equiv \lambda y \bullet P$;
- **subst** $([N/x](\lambda y \bullet P)) \equiv \lambda y \bullet \text{subst}([N/x]P)$,
if $x \neq y$ and y is not free in N or x is not free in P ;
- **subst** $([N/x](\lambda y \bullet P)) \equiv \lambda z \bullet \text{subst}([N/z] \text{subst}([z/y]P))$,
if $y \neq x$ and y is free in N and x is free in P
(where z is not free in $(N P)$).

C.5.4 α -Renaming and β -Reduction

- α -renaming: $\lambda x \bullet M$
If x, y are distinct variables then replacing x by y in $\lambda x \bullet M$ results in $\lambda y \bullet \text{subst}([y/x]M)$. 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: $(\lambda x \bullet M)(N)$
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. $(\lambda x \bullet M)(N) \equiv \text{subst}([N/x]M)$

C.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: BB × C → A

```

C.5.6 Function Definitions

Functions can be defined explicitly:

```

value
  f: Arguments → Result
  f(args) ≡ DValueExpr

  g: Arguments  $\leadsto$  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  $\leadsto$  Result
  g(args) as result
  pre P2(args)
  post P3(args,result)

```

The symbol \leadsto indicates that the function is partial and thus not defined for all arguments. Partial functions should be assisted by preconditions stating the criteria for arguments to be meaningful to the function.

C.6 Other Applicative Expressions

C.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:

```
 $(\lambda a. \mathcal{E}_b(a))(\mathcal{E}_d)$ 
```

C.6.2 Recursive let Expressions

Recursive **let** expressions are written as:

$$\text{let } f = \lambda a:A \bullet E(f) \text{ in } B(f,a) \text{ end}$$

is “the same” as:

$$\text{let } f = YF \text{ in } B(f,a) \text{ end}$$

where:

$$F \equiv \lambda g \bullet \lambda a \bullet (E(g)) \text{ and } YF = F(YF)$$

C.6.3 Predicative let Expressions

Predicative **let** expressions:

$$\text{let } a:A \bullet \mathcal{P}(a) \text{ in } \mathcal{B}(a) \text{ end}$$

express the selection of a value a of type A which satisfies a predicate $\mathcal{P}(a)$ for evaluation in the body $\mathcal{B}(a)$.

C.6.4 Pattern and “Wild Card” let Expressions

Patterns and *wild cards* can be used:

$$\text{let } \{a\} \cup s = \text{set} \text{ in } \dots \text{ end}$$

$$\text{let } \{a, _ \} \cup s = \text{set} \text{ in } \dots \text{ end}$$

$$\text{let } (a,b,\dots,c) = \text{cart} \text{ in } \dots \text{ end}$$

$$\text{let } (a,_,\dots,c) = \text{cart} \text{ in } \dots \text{ end}$$

$$\text{let } \langle a \rangle^\ell = \text{list} \text{ in } \dots \text{ end}$$

$$\text{let } \langle a,_,bb \rangle^\ell = \text{list} \text{ in } \dots \text{ end}$$

$$\text{let } [a \rightarrow bb] \cup m = \text{map} \text{ in } \dots \text{ end}$$

$$\text{let } [a \rightarrow b, _] \cup m = \text{map} \text{ in } \dots \text{ end}$$

C.6.5 Conditionals

Various kinds of conditional expressions are offered by RSL:

$$\begin{array}{l} \text{if } b_expr \text{ then } c_expr \text{ else } a_expr \\ \text{end} \end{array}$$

$$\begin{array}{l} \text{if } b_expr \text{ then } c_expr \text{ end} \equiv /* \text{ same as: } */ \\ \text{if } b_expr \text{ then } c_expr \text{ else skip end} \end{array}$$

$$\begin{array}{l} \text{if } b_expr_1 \text{ then } c_expr_1 \\ \text{elseif } b_expr_2 \text{ then } c_expr_2 \\ \text{elseif } b_expr_3 \text{ then } c_expr_3 \\ \dots \\ \text{elseif } b_expr_n \text{ then } c_expr_n \text{ end} \end{array}$$

$$\text{case } expr \text{ of}$$

```

choice_pattern_1 → expr_1,
choice_pattern_2 → expr_2,
...
choice_pattern_n_or_wild_card → expr_n
end

```

C.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⟩ ::= !

```

C.7 Imperative Constructs

C.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**.

C.7.2 Variables and Assignment

0. **variable** *v*:Type := expression
1. *v* := expr

C.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*

C.7.4 Imperative Conditionals

4. **if** expr **then** stm_c **else** stm_a **end**
5. **case** e **of**: $p_1 \rightarrow S_1(p_1), \dots, p_n \rightarrow S_n(p_n)$ **end**

C.7.5 Iterative Conditionals

6. **while** expr **do** stm **end**
7. **do** stmt **until** expr **end**

C.7.6 Iterative Sequencing

8. **for** e **in** list_expr • P(b) **do** S(b) **end**

C.8 Process Constructs**C.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).

C.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.

C.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.

C.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

$P: \text{Unit} \rightarrow \text{in } c \text{ out } k[i]$

Unit

$Q: i:KIdx \rightarrow \text{out } c \text{ in } k[i] \text{ Unit}$

$P() \equiv \dots c ? \dots k[i] ! e \dots$

$Q(i) \equiv \dots k[i] ? \dots c ! e \dots$

The process function definitions (i.e., their bodies) express possible events.

C.9 Simple RSL Specifications

Often, we do not want to encapsulate small specifications in schemes, classes, and objects, as is often done in RSL. An RSL specification is simply a sequence of one or more types, values (including functions), variables, channels and axioms:

type

...

variable

...

channel

...

value

...

axiom

...

In practice a full specification repeats the above listings many times, once for each “module” (i.e., aspect, facet, view) of specification. Each of these modules may be “wrapped” into scheme, class or object definitions.¹

¹ For schemes, classes and objects we refer to [25, Chap. 10]

D

Indexes

D.1. Definitions	245
D.2. Concepts	249
D.3. Examples	257
D.4. Analysis Prompts	259
D.5. Description Prompts	259
D.6. RSL Symbols	259

D.1 Definitions

abstract	inert, 25, 123
type, 14	programmable, 25, 124
action	reactive, 25, 123
derived, 171	shared, 26
discrete, 33	sharing
active	embedded, 35
attribute, 25, 123	static, 25, 123
Actor, 33	value expression
actor, 33	controllable, 39
analysis	autonomous
domain	attribute, 25, 123
prompt, 7, 10, 11, 13, 14, 16, 20, 23, 27, 28, 32,	
83–85, 87–91	behaviour
prompt	continuous, 35
domain, 7, 10, 11, 13, 14, 16, 20, 23, 27, 28, 32,	discrete, 34
83–85, 87–91	biddable
assumptions	attribute, 25, 124
design, 147	
Atomic	Component, 12, 85
part, 13, 86	component, 12, 85
Atomic Part, 13, 86	Composite
attribute	part, 13, 86
active, 25, 123	Composite Part, 13, 86
biddable, 25, 124	computer
controllable, 25	science, 5
value expression, 39	computing
dynamic, 25, 123	science, 5
embedded	concept
sharing, 35	formal, 9
event, 26	concrete
external, 25, 36	type, 14
	confusion, 31

- context
 - formal, 8
- continuous
 - behaviour, 35
 - endurant, 11, 85
- Continuous Endurant, 11, 85
- controllable
 - attribute, 25
 - value expression, 39
 - value expression
 - attribute, 39
- derived, 17
 - action, 171
 - event, 172
 - perdurant, 170
 - requirements, 166, 170
- Derived Action, 171
- Derived Event, 172
- Derived Perdurant, 170
- description
 - domain, 6
 - prompt, 7, 15, 16, 19, 21, 24, 27, 29, 32, 87–91
 - tree, 97
 - prompt
 - domain, 7, 15, 16, 19, 21, 24, 27, 29, 32, 87–91
 - text, 6
 - tree
 - domain, 97
- design
 - assumptions, 147
 - requirements, 147
- Determination, 156
- determination
 - domain, 156
- development
 - software
 - triptych, 82
 - triptych
 - software, 82
- discrete
 - action, 33
 - behaviour, 34
 - endurant, 11, 85
- Discrete Action, 33
- Discrete Behaviour, 34
- Discrete Endurant, 11, 85
- Domain
 - Engineering, 114
 - Science, 114
- domain, 5, 53
 - analysis
 - prompt, 7, 10, 11, 13, 14, 16, 20, 23, 27, 28, 32, 83–85, 87–91
 - description, 6
 - prompt, 7, 15, 16, 19, 21, 24, 27, 29, 32, 87–91
 - tree, 97
 - determination, 156
 - extension, 157
 - facet, 53
 - human behaviour, 76
 - instantiation, 152
 - management, 72
 - manifest, 5
 - organisation, 72
 - partial
 - requirement, 165
 - prescription
 - requirements, 148
 - projection, 148
 - prompt
 - analysis, 7, 10, 11, 13, 14, 16, 20, 23, 27, 28, 32, 83–85, 87–91
 - description, 7, 15, 16, 19, 21, 24, 27, 29, 32, 87–91
 - regulation, 61
 - requirement
 - partial, 165
 - shared, 165
 - requirements, 135, 146
 - prescription, 148
 - rule, 61
 - script, 63
 - shared
 - requirement, 165
 - tree
 - description, 97
 - Domain Instantiation, 152
 - Domain Projection, 148
 - Domain Requirements Prescription, 148
 - dynamic
 - attribute, 25, 123
 - embedded
 - attribute
 - sharing, 35
 - sharing
 - attribute, 35
 - Endurant, 10, 84
 - endurant, 10, 84
 - continuous, 11, 85
 - discrete, 11, 85
 - extension, 158
 - Endurant Extension, 158
 - Engineering
 - Domain, 114
 - engineering
 - systems, 4
 - Entity, 10, 83
 - entity, 10, 83, 84
 - Epistemology, 32
 - Event, 34
 - event, 34
 - attribute, 26
 - derived, 172
 - expression
 - function
 - type, 36

- type
 - function, 36
- Extension, 157
- extension
 - domain, 157
 - endurant, 158
- extent, 9
- external
 - attribute, 25, 36
 - part
 - quality, 19, 87
 - quality
 - part, 19, 87
- facet
 - domain, 53
- fitting
 - requirements, 165
- formal
 - concept, 9
 - context, 8
 - method, 82
 - software development, 82
 - software development
 - method, 82
- Formal Method, 82
- Formal Software Development, 82
- function
 - expression
 - type, 36
 - signature, 36
 - type
 - expression, 36
- Function Signature, 36
- Function Type Expression, 36
- goal, 175
- harmonisation
 - requirements, 165
- has_ concrete_ type
 - prerequisite
 - prompt, 16
 - prompt
 - prerequisite, 16
- has_ mereology
 - prerequisite
 - prompt, 21
 - prompt
 - prerequisite, 21
- human behaviour
 - domain, 76
- inert
 - attribute, 25, 123
- instantiation
 - domain, 152
- intent, 9
- interface
 - requirements, 135, 146, 165
- internal
 - part
 - quality, 19, 87
 - quality
 - part, 19, 87
- intrinsic, 54
- is_ composite
 - prerequisite
 - prompt, 15, 86
 - prompt
 - prerequisite, 15, 86
- is_ discrete
 - prerequisite
 - prompt, 14
 - prompt
 - prerequisite, 14
- is_ entity
 - prerequisite
 - prompt, 84
 - prompt
 - prerequisite, 84
- junk, 31
- knowledge, 46
- machine
 - requirements, 135, 146
- management
 - domain, 72
- manifest
 - domain, 5
- Material, 12, 85
- material, 12, 28, 85
- mereology, 20
 - type, 20
- Method, 82
- method, 4, 53, 82, 136
 - formal, 82
 - software development, 82
 - software development
 - formal, 82
- Methodology, 82
- methodology, 5, 82, 136
- observe_ concrete_ type
 - prerequisite
 - prompt, 16
 - prompt
 - prerequisite, 16
- observe_ part_ type
 - prerequisite
 - prompt, 87
 - prompt
 - prerequisite, 87
- Ontology, 32
- organisation
 - domain, 72

- Part, 12, 85
- part, 12, 85
 - Atomic, 13, 86
 - Composite, 13, 86
 - external
 - quality, 19, 87
 - internal
 - quality, 19, 87
 - qualities, 19, 87
 - quality
 - external, 19, 87
 - internal, 19, 87
- partial
 - domain
 - requirement, 165
 - requirement
 - domain, 165
- Perdurant, 10, 84
- perdurant, 10, 84
 - derived, 170
- phenomenon, 10, 83
- prerequisite
 - has_ concrete_ type
 - prompt, 16
 - has_ mereology
 - prompt, 21
 - is_ composite
 - prompt, 15, 86
 - is_ discrete
 - prompt, 14
 - is_ entity
 - prompt, 84
 - observe_ concrete_ type
 - prompt, 16
 - observe_ part_ type
 - prompt, 87
 - prompt, 87
 - has_ concrete_ type, 16
 - has_ mereology, 21
 - is_ composite, 15, 86
 - is_ discrete, 14
 - is_ entity, 10, 11, 84
 - observe_ concrete_ type, 16
 - observe_ part_ type, 87
- prescription
 - domain
 - requirements, 148
 - requirements
 - domain, 148
- programmable
 - attribute, 25, 124
- projection
 - domain, 148
- prompt
 - analysis
 - domain, 7, 10, 11, 13, 14, 16, 20, 23, 27, 28, 32, 83–85, 87–91
 - description
 - domain, 7, 15, 16, 19, 21, 24, 27, 29, 32, 87–91
- domain
 - analysis, 7, 10, 11, 13, 14, 16, 20, 23, 27, 28, 32, 83–85, 87–91
 - description, 7, 15, 16, 19, 21, 24, 27, 29, 32, 87–91
- has_ concrete_ type
 - prerequisite, 16
- has_ mereology
 - prerequisite, 21
- is_ composite
 - prerequisite, 15, 86
- is_ discrete
 - prerequisite, 14
- is_ entity
 - prerequisite, 84
- observe_ concrete_ type
 - prerequisite, 16
- observe_ part_ type
 - prerequisite, 87
- prerequisite, 87
 - has_ concrete_ type, 16
 - has_ mereology, 21
 - is_ composite, 15, 86
 - is_ discrete, 14
 - is_ entity, 84
 - observe_ concrete_ type, 16
 - observe_ part_ type, 87
- qualities
 - part, 19, 87
- quality
 - external
 - part, 19, 87
 - internal
 - part, 19, 87
 - part
 - external, 19, 87
 - internal, 19, 87
- reactive
 - attribute, 25, 123
- regulation
 - domain, 61
- requirement
 - domain
 - partial, 165
 - shared, 165
 - partial
 - domain, 165
 - shared
 - domain, 165
- requirements
 - derived, 166, 170
 - design, 147
 - domain, 135, 146
 - prescription, 148
 - fitting, 165
 - harmonisation, 165

- interface, 135, 146, 165
- machine, 135, 146
- prescription
 - domain, 148
- Requirements Fitting, 165
- Requirements Harmonisation, 165
- rule
 - domain, 61
- Science
 - Domain, 114
- science
 - computer, 5
 - computing, 5
- script
 - domain, 63
- share, 26
- shared
 - attribute, 26
 - domain
 - requirement, 165
 - requirement
 - domain, 165
- sharing, 166
 - attribute
 - embedded, 35
 - embedded
 - attribute, 35
- signature
 - function, 36
- software
 - development
 - triptych, 82
 - triptych
 - development, 82
- software development
 - formal
 - method, 82
 - method
 - formal, 82
- sort, 14
- State, 32
- state, 32
- static
 - attribute, 25, 123
- sub-part, 13, 86
- support
 - technology, 57
- systems
 - engineering, 4
- technology
 - support, 57
- text
 - description, 6
- The Triptych Approach to Software Development, 82
- tree
 - description
 - domain, 97
 - domain
 - description, 97
- triptych
 - development
 - software, 82
 - software
 - development, 82
- type, 14
 - abstract, 14
 - concrete, 14
 - expression
 - function, 36
 - function
 - expression, 36
 - mereology, 20
- value expression
 - attribute
 - controllable, 39
 - controllable
 - attribute, 39
- Verification Paradigm, 146

D.2 Concepts

- [endurant]
 - analysis prompts
 - domain, 92
 - description prompts
 - domain, 92
 - domain
 - analysis prompts, 92
 - description prompts, 92
- abstract
 - value, 19
- abstraction, 10, 57, 83
- access
 - attribute
 - value, 26
 - value
 - attribute, 26
- accessibility, 173
- action, 6, 32, 53, 69
 - shared, 137, 146
- actor, 66
 - attributes, 68
- adaptive, 173
- algorithmic

- engineering, 47
- analyser
 - domain, 4, 6
- analysis
 - domain, 4, 6, 9, 47–50
 - prompt, 7, 10, 11, 13, 14, 16, 20, 23, 27, 28, 32, 83–85, 87–91
 - problem
 - world, 48
 - product line, 47
 - prompt
 - domain, 7, 10, 11, 13, 14, 16, 20, 23, 27, 28, 32, 83–85, 87–91
 - world
 - problem, 48
- analysis prompts
 - [endurant]
 - domain, 92
 - domain
 - [endurant], 92
- architecture
 - software, 49
- assumptions
 - design, 147
- atomic, 6
- attribute, 46
 - access
 - value, 26
 - biddable, 35
 - controllable, 35, 36
 - value expression, 39
 - dynamic, 35
 - embedded
 - sharing, 35, 163, 164, 170
 - event, 26, 35
 - external, 36, 158, 160, 164
 - value, 26, 36
 - programmable, 35
 - shared, 83
 - sharing
 - embedded, 35, 163, 164, 170
 - static, 35, 36
 - update, 83
 - value
 - access, 26
 - external, 26, 36
 - value expression
 - controllable, 39
- autonomous, 36
- availability, 173
- axiom
 - sort
 - well-formedness, 31
 - well-formedness
 - sort, 31
- bases
 - knowledge, 47
- behaviour, 6, 32, 53
 - shared, 137, 146
- biddable
 - attribute, 35
- change
 - state, 35
- class
 - diagram, 49
- common
 - projection, 165
- communication, 43
- component, 83
 - reusable
 - software, 48
 - software, 49
 - reusable, 48
- composite, 6
- composite, 137
- computer
 - program, 136
 - science, 5, 45, 46
- computing
 - science, 5, 46
- conceive, 10, 83
- concept
 - formal, 9
- concurrency, 43
- confusion, 32
- conservative
 - extension, 158
 - proof theoretic, 57
 - proof theoretic
 - extension, 57
- Constraint, 175
- constructor
 - function
 - type, 36
 - type
 - function, 36
- context, 9
- continuant, 10
- continuous, 6
 - time, 35
- control, 62
- controllable
 - attribute, 35, 36
 - value expression, 39
 - value expression
 - attribute, 39
- corrective, 173
- demo
 - domain, 4
- demonstration, 173
- dependability, 173
 - requirements, 173
- derivation
 - part, 121
- derived

- requirements, 135, 137, 146
- describer, 4
 - domain, 4, 6
- description
 - development
 - domain, 48
 - domain, 4, 6, 47–50, 136
 - development, 48
 - facet, 174
 - projected, 44
 - prompt, 7, 15, 16, 19, 21, 24, 27, 29, 32, 87–91
 - tree, 97
 - facet
 - domain, 174
 - projected
 - domain, 44
 - prompt
 - domain, 7, 15, 16, 19, 21, 24, 27, 29, 32, 87–91
 - tree
 - domain, 97
- description prompts
 - [endurant]
 - domain, 92
 - domain
 - [endurant], 92
- descriptions
 - domain, 49
- design
 - assumptions, 147
 - requirements, 147
 - software, 4, 48, 49
 - specification, 136
 - specification
 - software, 136
- determination, 135, 146, 148
- development, 173
 - description
 - domain, 48
 - domain
 - description, 48
 - requirements, 146
 - interface
 - requirements, 146
 - model-oriented
 - software, 48
 - requirements, 49, 50, 173
 - domain, 146
 - interface, 146
 - software
 - model-oriented, 48
 - triptych, 82
 - triptych
 - software, 82
- diagram
 - class, 49
- discrete, 6
- documentation, 173
- domain, 49, 50, 53
 - [endurant]
 - analysis prompts, 92
 - description prompts, 92
 - analyser, 4, 6
 - analysis, 4, 6, 9, 47–50
 - prompt, 7, 10, 11, 13, 14, 16, 20, 23, 27, 28, 32, 83–85, 87–91
 - analysis prompts
 - [endurant], 92
 - demo, 4
 - describer, 4, 6
 - description, 4, 6, 47–50, 136
 - development, 48
 - facet, 174
 - projected, 44
 - prompt, 7, 15, 16, 19, 21, 24, 27, 29, 32, 87–91
 - tree, 97
 - description prompts
 - [endurant], 92
 - descriptions, 49
 - development
 - description, 48
 - requirements, 146
 - engineer, 4, 6, 48
 - engineering, 4, 6, 46, 48, 49, 137, 174
 - extension
 - requirements, 166
 - facet, 53, 54, 146
 - description, 174
 - intrinsic, 54
 - support technology, 57
 - intrinsic, 146
 - intrinsic, 54
 - facet, 54
 - language
 - specific, 47, 48
 - manifest, 53
 - modeling, 30, 47, 49
 - partial
 - requirement, 165
 - prescription
 - requirements, 148
 - problem, 5
 - projected
 - description, 44
 - prompt
 - analysis, 7, 10, 11, 13, 14, 16, 20, 23, 27, 28, 32, 83–85, 87–91
 - description, 7, 15, 16, 19, 21, 24, 27, 29, 32, 87–91
 - requirement
 - partial, 165
 - shared, 165
 - requirements, 135, 146
 - development, 146
 - extension, 166
 - prescription, 148
 - science, 6, 46

- scientist, 6
- semantic, 102
- shared
 - requirement, 165
- simulator, 4
- software
 - specific, 4, 49
- specific
 - language, 47, 48
 - software, 4, 49
- support technology
 - facet, 57
- syntactic, 102
- tree
 - description, 97
- domain requirements
 - partial
 - prescription, 148
 - prescription
 - partial, 148
- dynamic
 - attribute, 35
 - external
 - value, 26
 - value
 - external, 26
- embedded
 - attribute
 - sharing, 35, 163, 164, 170
 - sharing, 35
 - attribute, 35, 163, 164, 170
- endurant, 5, 6, 53, 83
 - shared, 137, 146
- engineer
 - domain, 4, 6, 48
 - requirements, 48
 - software, 6, 48
- engineering
 - algorithmic, 47
 - domain, 4, 6, 46, 48, 49, 137, 174
 - knowledge, 46, 47
 - ontological, 46
 - ontology, 45
 - product line
 - software, 48
 - requirements, 4, 48, 50, 137, 174
 - software, 136
 - product line, 48
- entities, 6
- entry, 161
- entry,, 160
- error, 173
- event, 6, 32, 53
 - attribute, 26, 35
 - shared, 137, 146
- execution, 173
- exit, 161
- exit,, 160
- expression
 - function
 - type, 36
 - type, 36
 - function, 36
- extension, 57, 135, 146, 148
 - conservative, 158
 - proof theoretic, 57
- domain
 - requirements, 166
 - proof theoretic
 - conservative, 57
 - requirements
 - domain, 166
- extensional, 173
- external
 - attribute, 36, 158, 160, 164
 - value, 26, 36
 - dynamic
 - value, 26
 - part
 - quality, 19, 87
 - processes, 35
 - quality
 - part, 19, 87
 - value
 - attribute, 26, 36
 - dynamic, 26
- facet, 53
 - description
 - domain, 174
 - domain, 53, 54, 146
 - description, 174
 - intrinsic, 54
 - support technology, 57
 - intrinsic
 - domain, 54
 - machine
 - requirement, 173
 - requirement
 - machine, 173
 - specific, 54
 - support technology
 - domain, 57
- failure, 173
- fault, 173
- fitting, 135, 146, 148
- formal
 - concept, 9
 - method
 - software development, 82
 - software development
 - method, 82
 - specification, 136
- formal concept analysis, 9
- frame
 - problem, 48
- frames

- problem, 48
- function
 - constructor
 - type, 36
 - expression
 - type, 36
 - name, 36
 - type
 - constructor, 36
 - expression, 36
- goal, 50, 175
- guarantee, 174
 - rely, 174
- hardware, 48
- has_ concrete_ type
 - prerequisite
 - prompt, 16
 - prompt
 - prerequisite, 16
- has_ mereology
 - prerequisite
 - prompt, 21
 - prompt
 - prerequisite, 21
- head, 35
 - pump, 35
- human behaviour, 54
- identifier
 - unique, 19, 46
- imperative
 - language
 - programming, 47
 - programming
 - language, 47
- implementation
 - partial, 62
- inert, 36
- information
 - science, 45, 46
- initial, 135
- instantiation, 135, 146, 148
- intrinsic, 146
 - domain, 146
- integrity, 173
- intensive, 54
- interface
 - development
 - requirements, 146
 - requirements, 135, 146, 158, 164, 166
 - development, 146
- interface
 - requirements, 137
- internal
 - part
 - quality, 19, 87
 - qualities, 12, 19, 22, 27, 85
- quality
 - part, 19, 87
- interval
 - time, 34
- intrinsic, 54
 - domain, 54
 - facet, 54
 - facet
 - domain, 54
- is_ composite
 - prerequisite
 - prompt, 15, 86
 - prompt
 - prerequisite, 15, 86
- is_ discrete
 - prerequisite
 - prompt, 14
 - prompt
 - prerequisite, 14
- is_ entity
 - prerequisite
 - prompt, 84
 - prompt
 - prerequisite, 84
- junk, 31
- knowledge, 46
 - bases, 47
 - engineering, 46, 47
 - representation, 47
- language
 - domain
 - specific, 47, 48
 - imperative
 - programming, 47
 - programming
 - imperative, 47
 - specific
 - domain, 47, 48
- license, 65, 66, 69
- license languages, 54
- licensee, 65, 66, 69
- licensing, 66, 69
- licensor, 65, 66, 69
- machine, 48, 135
 - facet
 - requirement, 173
 - requirement, 173
 - facet, 173
 - requirements, 135, 146, 173
- maintenance, 173
 - requirements, 173
- management, 173
- management & organisation, 54
- manifest
 - domain, 53

- phenomena, 5
- mathematical
 - object, 136
- mereology, 46
 - observer, 20
 - type, 20
 - update, 83
- method, 53
 - formal
 - software development, 82
 - software development
 - formal, 82
- methodology, 4
- model-oriented
 - development
 - software, 48
 - software
 - development, 48
- modeling
 - domain, 30, 47, 49
 - requirements, 30
- monitor, 62
- name
 - function, 36
- non-manifest
 - qualities, 5
- object
 - mathematical, 136
- objective
 - operational, 175
- obligation, 69
 - proof, 32
- observe, 10, 83
- observe_ concrete_ type
 - prerequisite
 - prompt, 16
 - prompt
 - prerequisite, 16
- observe_ part_ type
 - prerequisite
 - prompt, 87
 - prompt
 - prerequisite, 87
- observer
 - mereology, 20
- occurrent, 10
- ontological
 - engineering, 46
- ontology
 - engineering, 45
 - science, 45
 - upper, 45, 46
- operational
 - objective, 175
- operations research, 79
- parallelism, 43
- part, 13, 86
 - derivation, 121
 - external
 - quality, 19, 87
 - internal
 - quality, 19, 87
 - quality
 - external, 19, 87
 - internal, 19, 87
 - sort, 14
- partial
 - domain
 - requirement, 165
 - domain requirements
 - prescription, 148
 - implementation, 62
 - prescription
 - domain requirements, 148
 - requirement
 - domain, 165
- perdurant, 5, 6, 53, 83
- perfective, 173
- performance, 173
- permission, 69
- permit, 65
- phenomena
 - manifest, 5
 - shared, 137
- philosophy, 45
- platform, 173
 - requirements, 173
- prerequisite
 - has_ concrete_ type
 - prompt, 16
 - has_ mereology
 - prompt, 21
 - is_ composite
 - prompt, 15, 86
 - is_ discrete
 - prompt, 14
 - is_ entity
 - prompt, 84
 - observe_ concrete_ type
 - prompt, 16
 - observe_ part_ type
 - prompt, 87
 - prompt
 - has_ concrete_ type, 16
 - has_ mereology, 21
 - is_ composite, 15, 86
 - is_ discrete, 14
 - is_ entity, 84
 - observe_ concrete_ type, 16
 - observe_ part_ type, 87
- prescription
 - domain
 - requirements, 148
 - domain requirements

- partial, 148
- partial
 - domain requirements, 148
 - requirements, 4, 48–50, 136, 174
 - domain, 148
- preventive, 173
- principles, 53
- problem
 - analysis
 - world, 48
 - domain, 5
 - frame, 48
 - frames, 48
 - world, 48
 - analysis, 48
- process, 173
 - schema, 50
- processes
 - external, 35
- product line
 - analysis, 47
 - engineering
 - software, 48
 - software, 48
 - engineering, 48
- program
 - computer, 136
- programmable
 - attribute, 35
- programming
 - imperative
 - language, 47
 - language
 - imperative, 47
- projected
 - description
 - domain, 44
 - domain
 - description, 44
- projection, 135, 146, 148
 - common, 165
 - specific, 165
- prompt, 7
 - analysis
 - domain, 7, 10, 11, 13, 14, 16, 20, 23, 27, 28, 32, 83–85, 87–91
 - description
 - domain, 7, 15, 16, 19, 21, 24, 27, 29, 32, 87–91
 - domain
 - analysis, 7, 10, 11, 13, 14, 16, 20, 23, 27, 28, 32, 83–85, 87–91
 - description, 7, 15, 16, 19, 21, 24, 27, 29, 32, 87–91
 - has_ concrete_ type
 - prerequisite, 16
 - has_ mereology
 - prerequisite, 21
 - is_ composite
 - prerequisite, 15, 86
 - is_ discrete
 - prerequisite, 14
 - is_ entity
 - prerequisite, 84
 - observe_ concrete_ type
 - prerequisite, 16
 - observe_ part_ type
 - prerequisite, 87
 - prerequisite
 - has_ concrete_ type, 16
 - has_ mereology, 21
 - is_ composite, 15, 86
 - is_ discrete, 14
 - is_ entity, 84
 - observe_ concrete_ type, 16
 - observe_ part_ type, 87
- prompts, 4
- proof
 - obligation, 32
- proof theoretic
 - conservative
 - extension, 57
 - extension
 - conservative, 57
- pump
 - head, 35
- qualities, 6
 - internal, 12, 19, 22, 27, 85
 - non-manifest, 5
- quality, 46
 - external
 - part, 19, 87
 - internal
 - part, 19, 87
 - part
 - external, 19, 87
 - internal, 19, 87
- reactive, 36
- reliability, 173
- rely
 - guarantee, 174
- representation
 - knowledge, 47
- requirement
 - domain
 - partial, 165
 - shared, 165
 - facet
 - machine, 173
 - machine, 173
 - facet, 173
 - partial
 - domain, 165
 - shared
 - domain, 165
- requirements, 48–50

- dependability, 173
- derived, 135, 137, 146
- design, 147
- development, 49, 50, 173
 - domain, 146
 - interface, 146
- domain, 135, 146
 - development, 146
 - extension, 166
 - prescription, 148
- engineer, 48
- engineering, 4, 48, 50, 137, 174
- extension
 - domain, 166
- interface, 135, 146, 158, 164, 166
 - development, 146
- interface , 137
- machine, 135, 146, 173
- maintenance, 173
- modeling, 30
- platform, 173
- prescription, 4, 48–50, 136, 174
 - domain, 148
- technology, 173
- reusable
 - component
 - software, 48
 - software
 - component, 48
- reuse, 48
- robustness, 173
- rules & regulations, 54
- safety, 173
- schema
 - process, 50
- science
 - computer, 5, 45, 46
 - computing, 5, 46
 - domain, 6, 46
 - information, 45, 46
 - ontology, 45
- scientist
 - domain, 6
- scripts, 54
- security, 173
- semantic
 - domain, 102
- shared
 - action, 137, 146
 - attribute, 83
 - behaviour, 137, 146
 - domain
 - requirement, 165
 - endurant, 137, 146
 - event, 137, 146
 - phenomena, 137
 - requirement
 - domain, 165
- sharing, 19
 - attribute
 - embedded, 35, 163, 164, 170
 - embedded, 35
 - attribute, 35, 163, 164, 170
- signature, 32, 46
- simplification, 136, 148
- simplify, 150
- simulator
 - domain, 4
- software, 48
 - architecture, 49
 - component, 49
 - reusable, 48
 - design, 4, 48, 49
 - specification, 136
 - development
 - model-oriented, 48
 - triptych, 82
 - domain
 - specific, 4, 49
 - engineer, 6, 48
 - engineering, 136
 - product line, 48
 - model-oriented
 - development, 48
 - product line, 48
 - engineering, 48
 - reusable
 - component, 48
 - specific
 - domain, 4, 49
 - specification
 - design, 136
 - triptych
 - development, 82
- software development
 - formal
 - method, 82
 - method
 - formal, 82
- sort, 9
 - axiom
 - well-formedness, 31
 - part, 14
 - well-formedness
 - axiom, 31
- specific
 - domain
 - language, 47, 48
 - software, 4, 49
 - facet, 54
 - language
 - domain, 47, 48
 - projection, 165
 - software
 - domain, 4, 49
- specification

- design
 - software, 136
- formal, 136
- software
 - design, 136
- state, 32
 - change, 35
- static
 - attribute, 35, 36
 - value, 26
- sub-part, 13, 86
- support
 - technology, 164
- support technology, 54
 - domain
 - facet, 57
 - facet
 - domain, 57
- synchronisation, 43
- syntactic
 - domain, 102
- techniques, 53
- technology
 - requirements, 173
 - support, 164
- time, 32, 34
 - continuous, 35
 - interval, 34
- tools, 53
- tree
 - description
 - domain, 97
 - domain
 - description, 97
- triptych
 - development
 - software, 82
 - software
 - development, 82
- type, 9
 - constructor
 - function, 36
 - expression, 36
 - function, 36
 - function
 - constructor, 36
 - expression, 36
 - mereology, 20
- Unified Modeling Language
 - UML, 49
 - old .UML, 49
- unique
 - identifier, 19, 46
- update
 - attribute, 83
 - mereology, 83
- upper
 - ontology, 45, 46
- value
 - abstract, 19
 - access
 - attribute, 26
 - attribute
 - access, 26
 - external, 26, 36
 - dynamic
 - external, 26
 - external
 - attribute, 26, 36
 - dynamic, 26
 - static, 26
- value expression
 - attribute
 - controllable, 39
 - controllable
 - attribute, 39
- well-formedness
 - axiom
 - sort, 31
 - sort
 - axiom, 31
- world
 - analysis
 - problem, 48
 - problem, 48
 - analysis, 48

D.3 Examples

- 41 Actors, 33
- 27 Atomic Part Attributes, 23
- 15 Atomic Parts, 13
- 26 Attribute Propositions and Other Values, 23
- 45 Behaviours, 34
- 46 Bus System Channels, 35
- 51 Bus Timetable Coordination, 40
- 12 Components, 12
- 28 Composite Part Attributes, 23
- 16 Composite Parts, 14
- 17 Composite and Atomic Part Sorts of Transportation, 15
- 18 Concrete Part Types of Transportation, 16
- 34 Container Components, 28
- 19 Container Line Sorts, 17
- 10 Continuous Endurants, 11
- 9 Discrete Endurants, 11

4 Endurant Entity Qualities, 5
 31 Event Attributes, 26
 47 Flow in Pipelines, 35
 48 Insert Hub Action Formalisation, 37
 50 Link Disappearance Formalisation, 38
 2 Manifest Domain Endurants, 5
 3 Manifest Domain Perdurants, 5
 13 Materials, 13
 1 Names of Manifest Domains, 5
 6 One Domain – Three Models, 8
 14 Parts Containing Materials, 13
 33 Parts and Components, 27
 35 Parts and Materials, 28
 42 Parts, Attributes and Behaviours, 33
 11 Parts, 12
 5 Perdurant Entity Qualities, 5
 37 Pipeline Material Flow, 29
 36 Pipeline Material, 29
 25 Pipeline Parts Mereology, 21
 20 Pipeline Parts, 18
 39 Pipelines: Inter Unit Flow and Leak Law, 31
 38 Pipelines: Intra Unit Flow and Leak Law, 30
 29 Road Hub Attributes, 24
 43 Road Net Actions, 33
 24 Road Net Part Mereologies, 21
 44 Road Net and Road Traffic Events, 34
 1.10 Shared Attributes, 26
 22 Shared Timetable Mereology (I), 20
 32 Shared Timetables, 27
 49 Some Function Signatures, 37
 40 States, 32
 30 Static and Dynamic Attributes, 25
 52 Tollgates: Part and Behaviour, 42
 23 Topological Connectedness Mereology, 20
 7 Traffic System Endurants, 10
 8 Traffic System Perdurants, 10
 21 Unique Transportation Net Part Identifiers, 19

 Actors (#41), 33
 Atomic Part Attributes (#27), 23
 Atomic Parts (#15), 13
 Attribute Propositions and Other Values (#26), 23

 Behaviours (#45), 34
 Bus System Channels (#46), 35
 Bus Timetable Coordination (#51), 40–41

 Components (#12), 12
 Composite and Atomic Part Sorts of Transportation (#17), 15–16
 Composite Part Attributes (#28), 23
 Composite Parts (#16), 14
 Concrete Part Types of Transportation (#18), 16–17
 Container Components (#34), 28
 Container Line Sorts (#19), 17–18
 Continuous Endurants (#10), 11

 Discrete Endurants (#9), 11
 Domain Requirements

Derived Action:
 Tracing Vehicles (#5.16), 171–172
 Derived Event:
 Current Maximum Flow (#5.17), 172
 Determination
 Toll-roads (#5.9), 156–157
 Endurant Extension (#5.10), 158–164
 Fitting (#5.11), 165
 Instantiation
 Road Net (#5.7), 152–155
 Road Net, Abstraction (#5.8), 155
 Projection (#5.6), 148–151
 Projection:
 A Narrative Sketch (#5.5), 148

 Endurant Entity Qualities (#4), 5
 Event Attributes (#31), 26

 Flow in Pipelines (#47), 35

 Insert Hub Action Formalisation (#48), 37
 Interface Requirements
 Projected Extensions (#5.12), 166
 Shared
 Endurant Initialisation (#5.14), 167–169
 Endurants (#5.13), 167
 Shared Behaviours (#5.15), 169–170

 Link Disappearance Formalisation (#50), 38

 Manifest Domain Endurants (#2), 5
 Manifest Domain Perdurants (#3), 5
 Materials (#13), 13

 Names of Manifest Domains (#1), 5

 One Domain – Three Models (#6), 8

 Parts (#11), 12
 Parts and Components (#33), 27
 Parts and Materials (#35), 28
 Parts Containing Materials (#14), 13
 Parts, Attributes and Behaviours (#42), 33
 Perdurant Entity Qualities (#5), 5
 Pipeline Material (#36), 29
 Pipeline Material Flow (#37), 29–30
 Pipeline Parts (#20), 18
 Pipeline Parts Mereology (#25), 21–22
 Pipelines: Inter Unit Flow and Leak Law (#39), 31
 Pipelines: Intra Unit Flow and Leak Law (#38), 30–31

 Road Hub Attributes (#29), 24–25
 Road Net Actions (#43), 33
 Road Net and Road Traffic Events (#44), 34
 Road Net Part Mereologies (#24), 21
 Road Pricing System
 Design Assumptions (#5.2), 147
 Design Requirements (#5.1), 147

 Shared Attributes (#1.10), 26
 Shared Timetable Mereology (I) (#22), 20
 Shared Timetables (#32), 27

Some Function Signatures (#49), 37
 States (#40), 32
 Static and Dynamic Attributes (#30), 25
 Toll-Gate System
 Design Assumptions (#5.4), 147
 Design Requirements (#5.3), 147

Tollgates: Part and Behaviour (#52), 42–43
 Topological Connectedness Mereology (#23), 20
 Traffic System Endurants (#7), 10
 Traffic System Perdurants (#8), 10

Unique Transportation Net Part Identifiers (#21), 19–20

D.4 Analysis Prompts

is_ atomic, 14, 86
 is_ component, 13, 85
 is_ composite, 14, 86
 is_ continuous, 11, 85
 is_ discrete, 11, 85

is_ endurant, 10, 84
 is_ entity, 10, 83
 is_ material, 13, 85
 is_ part, 13, 85
 is_ perdurant, 11, 84

D.5 Description Prompts

observe_ attributes, 24, 89
 observe_ component_ sorts, 27, 89
 observe_ concrete_ type, 16, 87
 observe_ material_ sorts, 29, 90, 91

observe_ mereology, 21, 88
 observe_ part_ sorts, 15, 86
 observe_ unique_ identifier, 19, 87

D.6 RSL Symbols

Arithmetics

..., -2, -1, 0, 1, 2, ..., 229
 $a_i * a_j$, 232
 $a_i + a_j$, 232
 a_i / a_j , 232
 $a_i = a_j$, 231
 $a_i \geq a_j$, 231
 $a_i > a_j$, 231
 $a_i \leq a_j$, 231
 $a_i < a_j$, 231
 $a_i \neq a_j$, 231
 $a_i - a_j$, 232

Cartesians

(e_1, e_2, \dots, e_n) , 232

Chaos

chaos, 235, 236

Clauses

... **elsif** ... , 240
case b_e **of** $pa_1 \rightarrow c_1, \dots, pa_n \rightarrow c_n$ **end** , 240
if b_e **then** c_c **else** c_a **end** , 240

Combinators

let $a:A \bullet P(a)$ **in** c **end** , 240
let $pa = e$ **in** c **end** , 239

Functions

$f(args)$ **as** result, 239
post $P(args, result)$, 239

pre $P(args)$, 239

$f(a)$, 238

$f(args) \equiv expr$, 239

Imperative

case b_e **of** $pa_1 \rightarrow c_1, \dots, pa_n \rightarrow c_n$ **end** , 242
do stmt **until** b_e **end** , 242
for e **in** $list_{expr} \bullet P(b)$ **do** stmt(e) **end** , 242
if b_e **then** c_c **else** c_a **end** , 242
skip , 241
variable $v:Type$ **:=** expression , 241
while b_e **do** stmt **end** , 242
 $f()$, 241
 $stm_1; stm_2; \dots; stm_n$; , 241
 $v := expression$, 241

Lists

$\langle Q(l(i)) | i \text{ in } \langle 1..len \rangle \bullet P(a) \rangle$, 233
 hAB , 233
 $\ell(i)$, 235
 $\langle e_i .. e_j \rangle$, 233
 $\langle e_1, e_2, \dots, e_n \rangle B$, 233
elems ℓ , 235
hd ℓ , 235
inds ℓ , 235
len ℓ , 235
tl ℓ , 235

Logics

$b_i \vee b_j$, 231
 $\forall a:A \bullet P(a)$, 232
 $\exists! a:A \bullet P(a)$, 232
 $\exists a:A \bullet P(a)$, 232
 $\sim b$, 231
false, 229, 231
true, 229, 231
 $a_i = a_j$, 232
 $a_i \geq a_j$, 232
 $a_i > a_j$, 232
 $a_i \leq a_j$, 232
 $a_i < a_j$, 232
 $a_i \neq a_j$, 232
 $b_i \Rightarrow b_j$, 231
 $b_i \wedge b_j$, 231

Maps

$[F(e) \mapsto G(m(e)) \mid e:E \bullet e \in \text{dom } m \wedge P(e)]$, 233
 $[]$, 233
 $[u_1 \mapsto v_1, u_2 \mapsto v_2, \dots, u_n \mapsto v_n]$, 233
 $m_i \setminus m_j$, 237
 $m_i \circ m_j$, 237
 m_i / m_j , 237
dom m , 237
rng m , 237
 $m_i = m_j$, 237
 $m_i \cup m_j$, 237
 $m_i \dagger m_j$, 237
 $m_i \neq m_j$, 237
 $m(e)$, 237

Processes

channel $c:T$, 242
channel $\{k[i]:T \bullet i:\text{KIdx}\}$, 242
 $c!e$, 242
 $c?$, 242
 $k[i]!e$, 242
 $k[i]?$, 242
 $P \parallel Q$, 242
 $P \# Q$, 242
 $P: \text{Unit} \rightarrow \text{in } c \text{ out } k[i] \text{ Unit}$, 243
 $P \parallel Q$, 242

$P \parallel Q$, 242
 $Q: i:\text{KIdx} \rightarrow \text{out } c \text{ in } k[i] \text{ Unit}$, 243

Sets

$\{Q(a) \mid a:A \bullet a \in s \wedge P(a)\}$, 232
 $\{\}$, 232
 $\{e_1, e_2, \dots, e_n\}$, 232
 $\cap \{s_1, s_2, \dots, s_n\}$, 234
 $\cup \{s_1, s_2, \dots, s_n\}$, 234
card s , 234
 $e \in s$, 234
 $e \notin s$, 234
 $s_i = s_j$, 234
 $s_i \cap s_j$, 234
 $s_i \cup s_j$, 234
 $s_i \subset s_j$, 234
 $s_i \subseteq s_j$, 234
 $s_i \neq s_j$, 234
 $s_i \setminus s_j$, 234

Types

$(T_1 \times T_2 \times \dots \times T_n)$, 229
 T^* , 229
 T^ω , 229
 $T_1 \times T_2 \times \dots \times T_n$, 229
Bool, 229
Char, 229
Int, 229
Nat, 229
Real, 229
Text, 229
Unit, 241, 243
 $\text{mk_id}(s_1:T_1, s_2:T_2, \dots, s_n:T_n)$, 229
 $s_1:T_1 \ s_2:T_2 \ \dots \ s_n:T_n$, 229
 $T = \text{Type_Expr}$, 230
 $T_1 \mid T_2 \mid \dots \mid T_1 \mid T_n$, 229
 $T = \{\mid v:T' \bullet P(v)\}$, 230, 231
 $T == TE_1 \mid TE_2 \mid \dots \mid TE_n$, 230
 $T_i \rightsquigarrow T_j$, 229
 $T_i \rightarrow T_j$, 229
T-infset, 229
T-set, 229