

Dines Bjørner

SOFTWARE ENGINEERING

An Unended Quest

August 1, 2008: 09:42

Proposal for a Dr. Techn. Thesis

This document was submitted to
The Technical University of Denmark
on August 4, 2008, dated August 1, 2008

Friday August 1, 2008: Dines Bjorner Dr.techn. Thesis

— to the grateful, fond & loving memory of my parents Else and Ivar
1905–1993, 1907–1971

Preface

This book, together with three recently published volumes¹ primarily authored by the present author, constitutes the documents submitted for the defense of the Danish academic degree of Dr.techn.

The present volume was to have been written during a basically 2006 calendar year sabbatical — spent at JAIST: Japan Advanced Institute of Science and Technology, Ishikawa Prefecture, Japan — on a most kind leave from the Technical University of Denmark. JAIST, however, put such extraordinary and unexpected, and I should, for an old man like I was then, heavy strain on my sabbat time that a major part of the thesis, Part II, could only be completed after leaving JAIST.²

Now you may rightfully wonder: “Why on earth is Dines Bjørner writing a Dr.techn. thesis?”. Well there is a simple answer. Why not? A PhD degree in Denmark is like a PhD in the best countries. A Dr.techn. degree in Denmark is something substantially more. Whereas a PhD degree attests to the holders ability to “do science”, i.e., to study appropriate problems according to best scientific principles, a Danish Dr.techn. degree should attest to something more substantial: To actual and lasting research results.

I think that the three volumes footnoted below bear witness to substantial, actual and lasting research results on software engineering. And I will argue this point in the present volume.

Usually a Danish Dr.techn. degree is attempted in the younger years, and I reached the mandatory, i.e., the formal retirement age of 70 on October 4, 2007. Usually a Dr.techn. degree ought be a prerequisite for a chair at the

¹*Software Engineering*, Vol. 1: *Abstraction and Modelling*, Vol. 2: *Specification of Systems and Languages*, Vol. 3: *Domains, Requirements and Software Design*, Texts in Theoretical Computer Science, the EATCS Series. Springer 2006

²In deference to JAIST it should, however, be mentioned that some of the work then (instead) done at JAIST (now) appears as some of the appendices to this report and that the work in general, in the minds of my colleagues at JAIST, supports the current thesis !

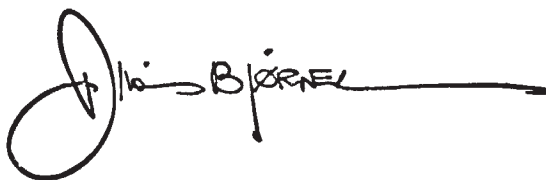
VIII

Technical University of Denmark. But I obtained my chair when I was 38 and had not yet had the possibility, in industry, to do the research necessary for such a thesis. And to submit a Dr.techn. thesis while already a professor was — in my days — not a proposition: “One just did not do that! Imagine if it was rejected!” Now this thesis may possibly be rejected, but I have retired!

Meanwhile, in the 31 years since I took possession of my chair, I have been blessed with health and energy (and time also at home (evenings and weekends) and during vacations and holidays³) to do a lot of thinking and reflection over the topic of software engineering in the context of formal techniques. This thinking and reflection has been done in an environment of many industry-oriented development projects where significant fragments of the contributions claimed in this volume were first tried out in ‘actual-life’ and on large-scale software developments. My university during most of these years, the Technical University of Denmark, has been my main source of academic joy for most of those years.

• • •

The current work has been underway now for almost 30 months. The state of this work is incomplete. There are some loose ends in some of the non-DrTechn appendices. They have been left dangling. One could go on forever. So is the case also for the main text and the DrTechn appendices.

A handwritten signature in black ink, reading "Dines Bjørner". The signature is stylized, with a large loop for the 'D' and a long horizontal stroke at the end.

Fredsvej 11, Holte, Denmark; August 1, 2008

³The time factor is important: My lovely wife of 42+ years provided for that!

Acknowledgements

In [87] I listed the numerous colleagues, most of whom I have worked with, and all of whom have over the years greatly influenced my thoughts and actions. I single out here a few: (the late) Cai Kindberg, Jean Paul Jacob, Gerald M. Weinberg, Peter Lucas, Gene Amdahl, (the late) John W. Backus, Lotfi Zadeh, (the late) E.F. (Ted) Codd, Cliff B. Jones, (the late) Hans Bekič, Heinz Zemanek, Dana Scott, (the late) Andrei Petrovich Ershov, Hans Langmaack, Andrzej Blikle, Neil D. Jones, Jørgen Fischer Nilsson, David Harel, Bo Stig Hansen, (the late) Søren Prehn, Sir Tony Hoare, Mícheál Mac an Airchinnigh, Michael Jackson, Zhou ChaoChen, Chris George, Jim Woodcock, Kokichi Futatsugi, (the late) Joseph A. Goguen, Larry Druffel and Wolfgang Reisig — listed more-or-less chronologically.

I wish in particular to acknowledge my deepest thanks and gratitude to (the late) Søren Prehn and to Chris George — for more than a quarter century of inspiration. Thanks to Lotfi Zadeh for first mentoring me. Emotional thanks to (the late) Cai Kindberg for first hiring me and for keeping me relatively straight since !

I also express my gratitude to the members of IFIP Working Groups WG 2.2 and WG 2.3 (not already mentioned above). The meetings of these working groups, with their “free for all” *topics for discussion sessions and debates*, have helped me sharpen and focus on what these volumes are about: Jean-Raymond Abrial, Jaco W. de Bakker, Manfred Broy, (the late) Ole-Johan Dahl, (the late) Edsger W. Dijkstra, Leslie Lamport, Zohar Manna, John McCarthy, Bertrand Meyer, Peter D. Mosses, Ernst-Rüdiger Olderog, Amir Pnueli, John Reynolds, Willem-Paul de Roever and Wlad Turski — listed alphabetically.

Over the years my many students have contributed to help me solidify the methodological principles and techniques of [87–89]. They are too numerous to mention. Well, some have or will be acknowledged on these pages!

From the writing phase of [87–89] invaluable thanks goes to my former students, Christian Krog Madsen (who wrote Chaps. 12–14 of Vol. 2), Steffen Holmslykke (who wrote Sect. 10.3 of Vol. 2), Martin Pěnička (who basically

wrote Sects. 12.3.4, 14.4.1 and 14.4.2 of Vol. 2), Yang Shaofa (who did the formalisation work of Vol. 2's Sect. 13.6), to Kirsten Mark Hansen (who so kindly allowed me, in Vol. 3's Sect. 19.6.5) to bring a crucial part (on Fault Tree Analysis) from her fine PhD Thesis, and to Hugh Anderson and Stefan Andrei. Most recently I have enjoyed collaborating with my former PhD student Dr. Asger Eir [103].

General acknowledgements related to my career in the last 30 years in Denmark are due to Prof. Christian Gram. Christian and I founded the Dansk Datamatik Center (DDC) whose 10+ year existence, 1979–1989, was a golden age of formal techniques in Denmark. Christian is a person whom I treasure very much.

Special acknowledgement is due to my colleague at the Technical University of Denmark (DTU) Hans Bruun. Without his exploratory engineering work on the static semantics of both the CHILL [174] and the Ada [121] programming languages DDC basically would have had no basis to start and, in its first may years, to thrive. I could have wished that Hans would have written several papers, perhaps a Dr.Techn. thesis, on his ground-breaking work.

Last but not least: the writing of this thesis would not have been possible without the financial support of my university (DTU) and the very kind permission to award me a year (2006) of sabbatical (at JAIST) to pursue that work. Well, a significant part of the current work has had to be done after my retirement and the size of my state pension, after 34 years at DTU, is nothing to acknowledge with any gratitude!

Contents

Dedication	V
Preface	VII
Acknowledgements	IX

Part I THE THESIS

1 The Thesis	3
1.1 On Software Engineering	3
1.2 The Three Volume Book: Software Engineering	4
1.3 Textbook Versus Research Monograph	4
1.4 The Thesis and Its Issues	4
1.4.1 On Method and Methodology	8
1.4.2 On Semiotics	8
1.4.3 On Computer and Computing Science	9
1.4.4 On Mathematics	10
1.4.5 On Entities, Operations, Events and Behaviour	10
1.4.6 On Phenomena and Concepts and Their Description	11
1.4.7 On Abstraction and Modelling	12
1.4.8 On Informal and Formal Specifications	13
1.4.9 On The Triptych of Software Engineering	14
1.4.10 On Documentation	15
1.4.11 On Omissions	15
Verification	15
Management	16
1.5 The Real Thesis	16
1.6 Treatment	17

Part II THE TEN THESIS ISSUES

2	On Method and Methodology	21
2.1	Background	22
2.1.1	Definitions of The Concept of 'Method'	22
	Discussion of 'Method' Definitions	22
	'Method' Conclusion	22
2.1.2	Why Emphasize the Concept of 'Method'?	23
2.2	Coverage in Book	23
2.3	Critique	25
2.3.1	On a Multitude of Principles, Techniques and Tools	25
2.3.2	On an Ontology of Principles, Techniques and Tools	25
2.4	Conclusion	26
3	On Semiotics	27
3.1	Mathematical Semiotics	28
3.2	Why the Semiotics Triplet Emphasis?	28
3.3	Coverage in Book	28
3.3.1	Places of Enunciation	28
3.3.2	Places of Deployment	29
	An Aside:	29
3.4	Critique	30
3.4.1	The Semiotics Claim	30
3.4.2	The Adequacy Claim	30
3.5	Conclusion	30
4	On The Computer and Computing Sciences	33
4.1	The Sciences of Computers and Computing	33
4.2	Coverage in Book	34
4.3	Critique	35
4.3.1	Scant Treatment of Computer Science	35
4.3.2	Software Engineering Based on Computing Science	36
4.3.3	Missing Computing Science Issues	36
4.4	A Diatribe	37
4.5	Conclusion	39
5	Mathematics	41
5.1	Why Mathematics?	41
5.2	Mathematical Notations	43
5.2.1	Writing Mathematics	43
5.2.2	Mathematical Abstractions	43
5.3	Which Mathematics?	43
5.3.1	Two Kinds of Mathematics	43
	Mathematical Underpinnings	44
	Software Engineering Mathematics	44
5.3.2	Specification Mathematics — which is included	44
5.3.3	Specification Mathematics — which is not included	47
5.4	Coverage in Book	48
5.5	Critique	48
5.5.1	Foundations	48
5.5.2	Continuity	48

5.5.3	Models of “Other” Mathematics	49
5.5.4	Possible “Waverings”	50
5.6	Conclusion	50
6	Simple Entities, Operations, Events and Behaviours	51
6.1	Introduction	51
6.1.1	Entities	52
6.1.2	Operations	53
6.1.3	States	53
6.1.4	Actions	53
6.1.5	Events	54
6.1.6	Behaviours and Processes	54
6.1.7	An Improved “Story”	54
6.1.8	Overview of Entities	55
6.1.9	Specific	55
6.1.10	Entities and Properties as an Ontological Base	56
6.2	Coverage in Book	56
6.2.1	Simple Entities	57
	Atomic and Composite Entities	57
	Atomic Entities	57
	Composite Entities	57
	Attributes, Types and Values	57
	Entities, Sub-entities and Mereology	58
	On “Data” Modelling	58
6.2.2	Functions	58
6.2.3	Events and Behaviours	58
6.3	Critique	59
6.4	Conclusion	59
7	Descriptions of Phenomena and Concepts	61
7.1	The Issues	61
7.1.1	Phenomena and Concepts	61
7.1.2	Descriptions	61
	Software Design Specification	62
	Requirements Prescription	62
	Domain Description	63
	What Can Be Described?	63
7.1.3	On Russel’s Theory of Descriptions	64
7.1.4	Conclusion	64
7.1.5	Caveats: Narratives and Annotations	64
7.2	What Are Phenomena and Concepts?	65
7.2.1	Phenomena	65
7.2.2	Concepts	65
7.3	Coverage in Book	65
7.4	Critique	66
7.4.1	Lack of Consistent Use of Narrative and Annotations	66
7.4.2	Critique of Coverage of Concept of Descriptions	66
7.4.3	Critique of Coverage of Concept of Phenomena and Concepts	66

7.4.4	Critique of Coverage of Descriptions, Phenomena and Concepts	66
7.4.5	Towards a Philosophy of Descriptions and Phenomena (<i>etc.</i>)	67
7.5	Conclusion	67
8	Abstraction and Modelling	69
8.1	The Issues	69
8.1.1	Abstraction	69
8.1.2	Models and Modelling	70
8.2	Coverage in Book	70
8.2.1	Abstraction	70
8.2.2	Property- Versus Model-Oriented Abstraction	71
8.2.3	Composite Modelling Patterns	71
8.3	Critique	72
8.4	Conclusion	72
9	On Informal and Formal Specifications	73
9.1	The Issues	73
9.1.1	The Communication of Descriptions <i>etc.</i>	73
9.1.2	Developing Descriptions, Prescriptions and Designs	74
9.2	Coverage in Book	74
9.3	Critique	74
9.4	Some Sociological Aspects of Formal Specifications	75
9.5	Conclusion	76
10	On The Triptych of Software Engineering	79
10.1	The Triptych	79
10.2	Domain Engineering	80
10.2.1	The Issues	80
10.2.2	Coverage in Book	81
10.2.3	Critique	81
	[1] Domain Facets	81
	[2] Domain Management & Organisation	81
	[3] Domain Scripts	82
	[4] Domain Verification, Model Checking and Formal Testing	82
	[5] Domain Theory	83
10.3	Requirements Engineering	83
10.3.1	The Issues	83
10.3.2	Coverage in Book	84
10.3.3	Critique	84
	[1] Domain Requirements	84
	[2] Interface Requirements	84
	[3] Requirements Verification, Model Checking and Formal Testing	85
10.4	Software Design	85
10.4.1	The Issues	86
10.4.2	Coverage in Book	86
10.4.3	Critique	86
	[1] Current Fashions	86

	[2] Software Verification, Model Checking and Formal Testing	87
	[3] Transition to Program Texts	87
10.5	Specific Software Development Models	88
10.6	Characterisations of SE and our Coverage of SE	88
10.7	Conclusion	89
11	Documentation	91
11.1	The Issue	91
11.2	Coverage in Book	92
11.3	Critique	92
11.3.1	Synopsis: Vol. 3, Sect. 2.4.4 (Page 63)	92
11.3.2	Implicit/Derivative Goals: Vol. 3, Sect. 2.4.6 (Page 65)	93
11.3.3	Tools	93
11.4	Conclusion	93
<hr/>		
Part III SOME NON-ISSUES		
<hr/>		
12	On Omissions	97
12.1	Main Omissions	97
12.1.1	Verification	97
	General	97
	On "Heroic Efforts"	98
	A Remedy	98
12.1.2	Management	99
12.2	Other Omissions	99
12.2.1	Refinement Calculi	100
12.2.2	UML	100
12.2.3	$\exists\forall$: Intentional Programming	100
	Discussion	101
12.2.4	Extreme Programming (XP)	102
12.2.5	Web Programming	102
12.3	Conclusion	103
<hr/>		
Part IV CONCLUSION		
<hr/>		
13	The Thesis Reviewed	107
	An Essay: Reflections	107
13.1	Reminder: Thesis Documents	108
13.2	What Is Being Claimed ?	108
13.2.1	A Contribution of Methodology, Didactics and Pedagogics	108
	[0] Meta Contribution	108
13.2.2	Main Methodology Contributions	109
	[1] The Triptych Paradigm	109
	[1.1] Domain Engineering	109
	[1.2] Domain Facets	109
	[1.3] Domain to Domain Requirements	109
	Operations	109

	[1.4] Domain to Interface Requirements	
	Operations	109
	[1.5] Machine Requirements	109
13.2.3	Main Supporting Methodology Contributions	110
	[2] Simple Entities, Operations, Events and Behaviours	110
	[3] Informal and Formal Descriptions	110
	[4] Phenomena and Concepts	110
	[5] Method Principles, Techniques and Tools	110
	[6] Semiotics: Pragmatics, Semantics and Syntax	110
	[7] Documentation	110
	[8] Abstraction and Modelling	111
13.2.4	Ancillary Methodology Contributions	111
	[9] Mathematics	111
	[10] Computer versus Computing Science	111
13.3	Does 'This' Constitute A Dr.Techn. Proposal ?	111
13.3.1	An Answer	111
13.3.2	Some Justification	112
13.3.3	Some More Justification	113
13.3.4	Some of My Publications	114
13.4	An Unending Quest	120

Part V DR. TECHN. APPENDICES

A	Domain Engineering	123
A.1	Introduction	124
A.1.1	Application cum Business Domains	124
A.1.2	Physics, Domains and Engineering	125
A.2	Domain Engineering: The Engineering Dogma	125
A.3	Entities, Functions, Events and Behaviours	126
A.3.1	Entities	126
	Atomic Entities	126
	Attributes — Types and Values:	127
	Composite Entities	127
	States	128
A.3.2	Functions	128
	Function Signatures	129
	Function Descriptions	129
A.3.3	Events	129
A.3.4	Behaviours	130
A.3.5	Discussion	131
A.4	Domain Facets	131
A.4.1	Intrinsics	132
	Railway Net Intrinsics:	132
	Conceptual Versus Actual Intrinsics	135
	On Modelling Intrinsics	135
A.4.2	Support Technologies	135
	Railway Optical Gate Technology Requirements:	137
	On Modelling Support Technologies	138

A.4.3	Management and Organisation	138
	Conceptual Analysis, First Part	139
	Methodological Consequences	140
	Conceptual Analysis, Second Part	140
	Conceptual Model of a Manager-Staff Relation, I:	140
	Conceptual Model of a Manager-Staff Relation, II:	141
	Conceptual Model of a Manager-Staff Relation, III:	141
	On Modelling Management and Organisation	142
A.4.4	Rules and Regulations	142
	A Meta-characterisation of Rules and Regulations	143
	On Modelling Rules and Regulations	144
A.4.5	Scripts and Licensing Languages	145
	Licensing Languages	147
	On Modelling Scripts	147
A.4.6	Human Behaviour	147
	A Meta-characterisation of Human Behaviour	149
	On Modelling Human Behaviour	149
A.4.7	Completion	149
A.4.8	Integrating Formal Descriptions	150
A.5	From Domain Models to Requirements	150
A.5.1	Domain Requirements	151
A.5.2	Interface Requirements	151
A.5.3	Machine Requirements	151
A.6	Why Domain Engineering?	151
A.6.1	Two Reasons for Domain Engineering	151
A.6.2	An Engineering Reason for Domain Modelling	152
A.6.3	On a Science of Domains	152
	Domain Theories	152
	A Scientific Reason for Domain Modelling	153
A.7	Conclusion	153
A.7.1	Summary	153
A.7.2	Grand Challenges of Informatics	153
A.7.3	Acknowledgements	153
A.8	Bibliographical Notes	154
B	Compositionality: Ontology and Mereology of Domains	155
B.1	A Prologue Example	155
	Narrative	156
	Formalisation	156
	Narrative	156
	Formalisation	157
	Closing Remarks	157
B.2	Introduction	157
B.2.1	Domain Engineering	157
	The Domain Engineering Dogma	157
	The Software Development Triptych	158
	Domain Descriptions	158

XVIII Contents

	B.2.2	Compositionality	158
	B.2.3	Ontology	159
	B.2.4	Mereology	159
	B.2.5	Paper Outline	160
B.3		An Ontology Aspect of Description Structures	160
	B.3.1	Simple Entities	161
	B.3.2	Operations	161
		“First Class” Entities	162
	B.3.3	Events	162
		Time and Time Stamps	162
		Definition: Event	163
		Definition: Same Event	163
		Definition: Event Designator:	164
		“First Class” Entities	164
	B.3.4	Behaviours	164
		“First Class” Entities	165
	B.3.5	First-class Entities	165
		Operations as Entities	165
		Actions as Entities	165
		Events as Entities	165
		Behaviours as Entities	166
	B.3.6	The Ontology: Entities and Properties	166
B.4		Simple Atomic and Composite Entities	166
	B.4.1	Simple Attributes — Types and Values	166
	B.4.2	Atomic Entities	167
	B.4.3	Composite Entities	167
	B.4.4	Discussion	168
		Attributes	168
		Compositions	168
B.5		Atomic and Composite Operations	168
	B.5.1	Signatures — Names and Types	169
	B.5.2	Atomic Operations	169
		Example Atomic Operations	169
	B.5.3	Composite Operations	169
		Example Composite Operations	170
		The Composition Homomorphism	170
B.6		Atomic and Composite Events	171
	B.6.1	Atomic Events	171
	B.6.2	Definitions: Atomic and Composite Events	172
	B.6.3	Composite Events	172
		Synchronising Events	172
		Sub-Events	172
		Sequential Events	172
		Embedded Events	173
		Event Clusters	173
B.7		Atomic and Composite Behaviours	174
	B.7.1	Modelling Actions and Events	174
	B.7.2	Atomic Behaviours	174
	B.7.3	Composite Behaviours	174

	Simple Traces	174
	Simple Behaviours	174
	Consecutive Behaviours	175
	Concurrent Behaviours	175
	Communicating Behaviours	175
	Joined Behaviours	176
	Forked Behaviours	176
B.7.4	General Behaviours	177
B.7.5	A Model of Behaviours	177
B.8	Mereology and Compositionality Concluded	177
B.8.1	The Mereology Axioms	177
B.8.2	Composite Simple Entities	178
	Mereology	178
	Compositionality	179
B.8.3	Composite Operations	181
	Mereology	181
	Compositionality	181
B.8.4	Composite Events	181
	Mereology	181
	Compositionality	182
B.8.5	Composite Behaviours	182
	Mereology	182
	Compositionality	183
B.9	Galois Connections	183
B.9.1	Definition	184
B.9.2	Concept Formation in Formal Concept Analysis [FCA]	185
B.9.3	Classification of Railway Networks	186
B.9.4	Relating Domain Concepts Intensionally	186
B.9.5	Further Examples	186
B.9.6	Generalisation	187
B.9.7	Galois Connections and Ontology	188
	Composite Entities	189
	Composite Operations	189
	Composite Events	190
	Composite Behaviours	191
B.9.8	Galois Connections Concluded	192
B.10	Conclusion	192
B.10.1	Ontology	192
B.10.2	Mereology	193
B.10.3	Research Issues	193
	Compositionality	193
	Mereology	193
	Ontology	193
	Galois Connections	194
B.10.4	Acknowledgement	194
B.11	Bibliographical Notes	194
B.12	Two Axiom Systems for Mereology	194
B.12.1	Parts and Places	194
B.12.2	A Calculus of Individuals Based on 'Connection'	196

C	Domain Theory: Practice and Theories: A Discussion of Possible Research Topics	199
C.1	Introduction	199
C.1.1	A Preamble	199
C.1.2	On Originality	199
C.1.3	Structure of Paper	200
C.2	Domain Engineering: A Dogma and its Consequences	200
C.2.1	The Dogma	200
C.2.2	The Consequences	200
C.2.3	The Triptych Verification	201
C.2.4	Full Scale Development: A First Suggested Research Topic	201
C.2.5	Examples of Domains	201
	The Examples	201
	Some Remarks	203
C.2.6	Domains: Suggested Research Topics	203
C.3	Domain Facets	204
C.3.1	Stages of Domain Development	204
C.3.2	The Facets	205
C.3.3	Intrinsics	205
C.3.4	Support Technology	205
	Sampling Behaviour of Support Technologies	205
	Probabilistic cum Statistical Behaviour of Support Technologies	206
	Support Technology Quality Control, a Sketch	206
	Support Technologies: Suggested Research Topics	207
C.3.5	Management and Organisation	207
	An Abstraction of Management Functions	207
	Management and Organisation: Suggested Research Topics	208
C.3.6	Rules and Regulations	209
	Abstraction of Rules and Regulations	209
	Quality Control of Rules and Regulations	209
	Rules and Regulations Suggested Research Topic:	210
C.3.7	Human Behaviour	210
	Abstraction of Human Behaviour	210
	Human Behaviour Suggested Research Topics:	211
C.3.8	Domain Modelling: Suggested Research Topic	211
C.4	Domains: Miscellaneous Issues	211
C.4.1	Domain Theories	211
	Example Theorem of Railway Domain Theory	212
	Why Domain Theories ?	212
	Domain Theories: Suggested Research Topics:	212
C.4.2	Domain Descriptions and Requirements Prescriptions	212
	From Domains to Requirements	212
	Domain and Interface Requirements: Suggested Research Topics	213
	Machine Requirements	213
C.4.3	Requirements-Specific Domain Software Development	
	Models	213
	Software "Intensities"	214

	“Abstract” Developments	214
	Requirements-Specific Devt. Models: Suggested Research Topics	214
C.4.4	On Two Reasons for Domain Modelling	214
	An Engineering Reason for Domain Modelling	215
	A Science Reason for Domain Modelling	215
	Domains Versus Requirements-Specific Development Models	215
C.5	Conclusion	216
C.5.1	What Has Been Achieved ?	216
C.5.2	What Needs to Be Achieved ?	216
	Domain Theories: Grand Challenge Research Topics	216
C.5.3	Acknowledgements	216
D	From Domains to Requirements: A Rigorous Approach	217
D.1	Introduction	218
D.2	The Triptych Principle of Software Engineering	218
D.3	Domain Engineering	219
D.3.1	Stages of Domain Engineering	219
D.3.2	First Example of a Domain Description	219
	Rough Sketching — Business Processes	219
	Narrative — Entities	220
	Formalisation — Entities	220
	Narrative — Operations	220
	Formalisation — Operations	221
	Narrative — Events	222
	Formalisation — Events	222
	Narrative — Behaviours	223
	Formalisation — Behaviours	223
D.3.3	Domain Modelling: Describing Facets	223
	Domain Intrinsic	224
	A Transportation Intrinsic — Narrative	224
	A Transportation Intrinsic — Formalisation	224
	Domain Support Technologies	225
	A Transportation Support Technology Facet — Narrative, 1	225
	A Transportation Support Technology Facet — Formalisation, 1	225
	A Transportation Support Technology Facet — Narrative, 2	226
	A Transportation Support Technology Facet — Formalisation, 2	226
	A Transportation Support Technology Facet — Narrative, 3	226
	A Transportation Support Technology Facet — Formalisation, 3	226
	Domain Management & Organisation	227
	A Transportation Management & Organisation Facet — Narrative	227

	A Transportation Management & Organisation	
	Facet — Formalisation	228
	Domain Rules & Regulations	228
	Domain Rules	228
	Domain Regulations	228
	A Transportation Rules & Regulations Facet —	
	Narrative	229
	A Transportation Rules & Regulations Facet —	
	Formalisation	229
	Domain Scripts	229
	A Transportation Script Facet — Narrative . . .	230
	A Transportation Script Facet — Formalisation .	230
	Domain Human Behaviour	230
	Transportation Human Behaviour Facets —	
	Narrative	230
	Transportation Human Behaviour Facets —	
	Formalisation	231
D.3.4	Discussion	231
D.4	Requirements Engineering	231
	The Example Requirements	231
D.4.1	Stages of Requirements Engineering	232
D.4.2	Business Process Re-engineering	232
	Re-engineering Domain Entities	232
	Re-engineering Domain Operations	233
	Re-engineering Domain Events	233
	Re-engineering Domain Behaviours	233
D.4.3	Domain Requirements Prescription	233
	Domain Projection	234
	Domain Projection — Narrative	234
	Domain Projection — Formalisation	234
	Domain Instantiation	234
	Domain Instantiation — Narrative	234
	Domain Instantiation — Formalisation, Toll	
	Way Net	235
	Domain Instantiation — Formalisation,	
	Well-formedness	236
	Domain Determination	236
	Domain Determination — Narrative	237
	Domain Determination — Formalisation	237
	Domain Extension	238
	Domain Extension — Narrative	238
	Domain Extension — Formalisation	239
	Domain Extension — Formalisation of Support	
	Technology	239
	Requirements Fitting	239
	Requirements Fitting Procedure — A Sketch . . .	240
	Requirements Fitting — Narrative	240
	Requirements Fitting — Formalisation	241
	Domain Requirements Consolidation	241

D.4.4	Interface Requirements Prescription	241
	Shared Entities	242
	Data Initialisation	242
	Data Refreshment	242
	Shared Operations	243
	Interactive Operation Execution	243
	Shared Events	243
	Shared Behaviours	243
D.5	Discussion	243
D.5.1	An 'Odyssey'	243
D.5.2	Claims of Contribution	244
D.5.3	Comparison to Other Work	244
D.5.4	A Critique	244
D.6	Acknowledgments	245
E	Believable Software Management	247
E.1	Introduction	247
E.2	Background	249
E.2.1	General	249
E.2.2	Characterisations	249
	Software	249
	Software Engineering	250
	Further Characterisations	251
	Domain	251
	Domain Description	251
	Domain Engineering	251
	Machine	252
	Requirements	252
	Requirements Prescription	252
	Requirements Engineering	252
	Software	252
	Software Design	252
	Software Development	252
	Software Engineering	252
	Software House	253
	Resources	253
	Management	253
	Strategic Management	254
	Tactical Management	254
	Operations Management	254
	Software Management	255
	Method and Methodology	255
	Method	255
	Methodology	255
E.2.3	Discussion	255
E.3	On Software Development Processes	256
E.3.1	Processes, Process Specifications and Process Models	256
E.3.2	Software Development Process Descriptions	256
	Domain Engineering	257

	Requirements Engineering	259
	Software Design	261
E.3.3	Documents	262
E.3.4	Formal Techniques	262
E.3.5	Software Development Graphs	263
E.3.6	The Process Model Graph	264
	Process Model Graph Construction	264
	Graphs and Graph Traversals (Traces)	264
	Process Models and Processes	265
	Incomplete and Extraneous Processes	265
	Process Iterations	266
E.3.7	Classical Process Models	266
E.4	CMM: The Capability Maturity Model	266
E.4.1	Level 1, Initial	267
E.4.2	Level 2, Repeatable	267
E.4.3	Level 3, Defined	268
E.4.4	Level 4, Managed	268
E.4.5	Level 5, Optimising	269
E.5	Software Management	269
E.5.1	Software Project Management	269
	Summary of Software Development Project Processes	270
	The Creative Aspects of Software Project Management	270
	The Monitoring and Control Aspects of Software Project	
	Management	272
	Syntactic and Semantic Process Compliance.	273
	A "Base 0" for TripTych Developments.	274
	Pragmatics.	274
	Resource Planning.	275
	Monitoring & Controlling Resource Usage.	276
	From Informal to Formal Development.	276
	Informal Development,	277
	Systematic, Rigorous and Formal Development.	277
	Staff Qualification.	278
	Tools.	278
	Tool Qualification.	279
	Review of Process Assessment and Process Improvement	
	Issues	279
E.5.2	Software Product Management	280
	Summary of Requirements Development Stages	281
	Implied Domain-Specific Software Product Possibilities	281
	"Middle-layer" and Systems Software Products	282
	The Creative Aspects of Software Product Management	283
	Software Evolution	283
E.6	Believable Software Managers and Management	284
E.6.1	Project Believability \equiv Project Quality	284
E.6.2	Product Believability \equiv Project Quality	285
E.6.3	Believable Software Managers	286
E.6.4	Believable Software Management	286
E.7	Conclusion	286

E.8	Bibliographical Notes	286
E.9	Software Development Documents	287
E.9.1	Domain Engineering Documents	287
E.9.2	Requirements Engineering Documents	287
E.9.3	Software Design Engineering Documents	289
F	An Example Domain Model: Intrinsic	291
F.1	A Transport Example	292
F.2	An Essence of 'Transport'	292
F.3	Business Processes	292
F.4	Entities	292
F.4.1	Basic Entities	292
F.4.2	Further Entity Properties	295
F.4.3	Entity Projections	296
F.5	Operations	297
F.5.1	Syntax	297
F.5.2	Semantics	299
F.6	Events	305
F.6.1	Some General Comments	305
F.6.2	Transport Event Examples	305
F.6.3	Banking Event Examples	306
F.7	Some Fundamental Modelling Concepts	306
F.7.1	Time and Time Intervals	306
F.7.2	Vehicles and Hub and Link Positions	308
F.8	Behaviours	309
F.8.1	Traffic as a Behaviour	309
F.8.2	A Net Behaviour	311
F.9	Traffic Events	312
F.10	Discussion	312

Part VI MISCELLANEOUS APPENDICES

G	Documents	315
G.1	Introduction	315
G.1.1	Aims and Objectives	315
	Aims	316
	Objectives	316
G.1.2	Domain Engineering	316
G.1.3	Related Work	316
G.1.4	A Caveat	317
G.1.5	Structure of Paper	317
G.2	Documents: A Domain Analysis	317
G.2.1	Basics	317
	Analysis: Basic Entities	317
	Narrative: Basic Entities	318
	Formalisation: Basic Entities	318
	Narrative: Basic Functions	319
	Formalisation: Basic Functions	319

	Formalisation: The Editing Functions	319
	Analysis: Document Identifiers	320
	Formalisation: Document Identifiers	320
	Analysis: Document Kinds	321
	Analysis: Document History	321
	Narrative: Document Kinds	322
	Formalisation: Document Kinds	322
	Narrative: Document History	323
	Formalisation: Document History	323
G.2.2	Locations, Time and Agents	324
	Locations	324
	Analysis: Points	324
	Narration: Locations	325
	Formalisation: Locations	325
	Time and Time Intervals	326
	Analysis: Time	326
	Formalisation: Time	326
	Comments on the Axiomatisation: Time	327
	Analysis: Time Intervals	327
	Formalisation: Time and Time Intervals	327
	Time and Space	328
	Analysis: Time and Space	328
	Annotations: Time and Space	328
	Discussion of the Blizzard Model of Space/Time	329
	Agents	329
	Analysis: Agents	329
	Narrative: Agents	329
	Formalisation: Agents	330
G.2.3	Spaces of Documents and Agents	330
	Narrative: Spaces of Documents and Agents	330
	Formalisation: Spaces of Documents and Agents	331
	Formalisation: Types of Document Operations	331
G.2.4	Dynamic System of Documents	331
G.2.5	Document Traces	332
	Note:	332
	First: Extension of Create and Edit Operations	333
	Then: Definition of Traces	333
G.2.6	Summary	333
G.3	From Domain Models to Requirements	334
G.3.1	Domain Requirements	334
G.3.2	Interface Requirements	334
G.3.3	Machine Requirements	334
G.4	Why Domain Engineering?	335
G.4.1	Two Reasons for Domain Theories	335
G.4.2	A Utilitarian, an Engineering Reason	335
G.4.3	A Scientific Reason	336
G.5	Conclusion	336
G.5.1	What Have We Shown?	336
G.5.2	What Have We Not Shown?	336

	Non-determinism	336
	Two Differences between Domains and Requirements	337
G.5.3	Shortcomings	337
G.5.4	What Needs Be Done?	337
H	Three License Languages	339
H.1	Introduction	340
H.1.1	What Kind of Science Is This?	340
H.1.2	What Kind of Contributions?	341
H.2	Pragmatics of The Three License Languages	341
H.2.1	The Performing Arts: Producers and Consumers	341
	Operations on Digital Works	341
	Past versus Future	342
	License Agreement and Obligation	342
	The Artistic Electronic Works: Two Assumptions	342
	Protection of the Artistic Electronic Works	342
	An Artistic Digital Rights License Language	342
H.2.2	Hospital Health Care: Patients and Medical Staff	343
	Hospital Health Care: Patients and Patient Medical Records	343
	Hospital Health Care: Medical Staff	343
	Professional Health Care	343
	A Hospital Health Care License Language	343
H.2.3	Public Government and the Citizens	343
	The Three Branches of Government	343
	Documents	344
	Document Attributes	344
	Actor Attributes and Licenses	345
	Document Tracing	345
	A Public Administration Document License Language	345
H.3	Semantic Intents of Licensor and Licensee Actions	345
H.3.1	Overview	345
H.3.2	Licenses and Actions	346
	Licenses	346
	Actions	346
	Two Languages	346
H.3.3	Sub-licensing, Scheme I	346
H.3.4	Sub-licensing, Scheme II	347
H.3.5	Multiple Licenses	348
H.4	Syntax and Informal Semantics	348
H.4.1	A General Artistic License Language	348
	The Syntax	348
	Licenses	349
	Syntax Annotations	349
	Informal Semantics	350
	Actions	351
	Annotations and Informal Semantics:	351
H.4.2	A Hospital Health Care License Language	352
	A Notion of License Execution States	352
	The Syntax	353

	Licenses	354
	Actions	354
	Informal Semantics	355
H.4.3	A Public Administration License Language	355
	The Syntax: Licenses	355
	The Form of Licenses	356
	Licensed Operations	356
	Syntax & Informal Semantics Annotations:	
	Licenses	356
	The Syntax: Actions	358
	Preliminary Remarks	359
	Syntax & Informal Semantics Annotations:	
	Actions	359
H.4.4	Discussion	360
	Comparisons	360
	Differences	360
H.5	Formal Semantics	361
H.5.1	A Model of Common Aspects	361
	Actors: Behaviours and Processes	361
	System States	361
	System Processes	361
	Actor Processes	362
	Functions	363
	Auxiliary Function Signatures	363
	Elaboration Function Signature	364
	Discussion of Auxiliary Functions	364
	Schema Definitions of Elaboration Functions ...	364
H.5.2	A Model of The General Artistic License Language	365
	A Licensor/Licensee State	365
	Auxiliary Functions	365
	Elaboration Functions	365
H.5.3	A Model of The Hospital Health Care License Language ...	365
	A Patient [Medical Record] State	365
	A Medical Staff State	365
	Auxiliary Functions	365
	Elaboration Functions	365
H.5.4	A Model of The Public Administration License Language ...	365
	A Public Administrator State	365
	Auxiliary Functions	366
	Elaboration Functions	366
H.5.5	Discussion	366
H.6	Conclusion	366
H.6.1	Summary: What Have We Achieved?	366
H.6.2	Open Issues	366
	An Open Issue: Rôle of Modal Logics	366
	Another Open Issue: Fair Use	366
H.6.3	Acknowledgements	367
H.7	The Gunter et al. Model — And its Reformulation	367
H.7.1	Digital Rights Licensing	367

	What is Digital Rights?	367
	Realities by Users and Licenses Issued by Owners	367
	Digital Rights Management (DRM)	367
	Structure of Presentation	367
H.7.2	Transcript of the Gunter/Weeks/Wright Paper	368
	Actions, Events, Realities and Licenses	368
H.7.3	Standard Licenses	370
	Up Front Licenses	370
	Flat Rate Licenses	370
	Per Use Licenses	371
	Up to Expiry Date Licenses	371
	Non-cancelable Multi-use Licenses	372
	Cancelable Multi-use Licenses	372
H.7.4	A License Language	373
	Syntax	373
	Examples	373
	Semantics	373
H.7.5	An RSL Model	374
	Actions, Events, Realities and Licences	374
H.7.6	Standard Licences	375
	Syntax	375
	Semantics	376
H.7.7	A License Language	379
H.7.8	End of "Gunter" Paper	379
I	Management and Organisation	381
I.1	Management and Organisation	382
I.1.1	Management	382
	Management Issues	382
	Basic Functions of Management	382
	Formation of Business Policy	383
	Implementation of Policies and Strategies	383
	Development of Policies and Strategies	383
	Management Levels	384
	Resources	384
	Resource Conversion	384
	Strategic Management	384
	Tactical Management	385
	Operational Management	385
	Supervisors and Team Leaders	386
	Description of 'Management'	386
	Review of Support Examples	387
	The Enterprise Function	387
	The Enterprise Processes	388
I.1.2	Organisation	388
I.2	A Simple, Functional Description of Management	388
I.2.1	A Base Narrative	389
I.2.2	A Formalisation	390
I.2.3	A Discussion of The Formal Model	390

	A Re-Narration	390
	On The Environment &c.	391
	On Intra-communication	391
	On Recursive Next-state Definitions	391
	Summary	392
I.3	A Simple, Process Description of Management	392
I.3.1	An Enterprise System	392
I.3.2	States and The System Composition	392
I.3.3	Channels and Messages	393
I.3.4	Process Signatures	393
I.3.5	The Shared State Process	394
I.3.6	Staff Processes	394
I.3.7	A Generic Staff Behaviour	394
	A Diagrammatic Rendition	395
	Auxiliary Functions	395
	Assumptions	397
I.3.8	Management Operations	398
	Focus on Management	398
	Own and Global States	398
	State Classification	398
	Transport System States	399
I.3.9	The Overall Managed System	399
I.3.10	Discussion	400
	Management Operations	400
	Managed States	400
I.4	Discussion of First Two Management Models	400
I.4.1	Generic Management Models	400
I.4.2	Management as Scripts	400
I.5	Transport Enterprise Organisation	401
I.5.1	Transport Organisations	401
I.5.2	Analysis	401
I.5.3	Modelling Concepts	402
	Net Kinds	402
	Enterprise Kinds	403
	Staff Kinds	403
	Staff Kind Constraints	403
	Narrative	403
	Formalisation	404
	Hierarchical Staff Structures	404
	Matrix Staff Structures	404
	Net and Enterprise Kind Constraints	404
	Narrative	404
	Formalisation	404
I.5.4	Net Signaling	405
	Narrative	405
	Formalisation	406
I.6	Discussion	406

J	Bayesian Networks	407
J.1	Introduction	407
J.1.1	Aims & Objectives	408
	Aims	408
	Objectives	408
J.1.2	Motivation	408
J.1.3	Structure of Document	408
J.2	Bayesian Networks	409
J.2.1	Informal Introduction to Bayesian Networks	409
J.2.2	Representations of Bayesian Networks	409
	Pictures: Network Graphs	409
	Bayesian Tables	409
	Formalisation	410
	An Example	411
K	On Russel's' Theory of Descriptions	413
K.1	Denoting Sentences	413
K.2	Definite Descriptions	413
K.2.1	Co-referring Expressions	414
K.2.2	Non-referring Expressions	414
K.3	Definite Descriptions	414
K.4	Indefinite Descriptions	416
L	Repository	417
L.1	Definitions	417
L.1.1	'Method'	418
L.1.2	'Principle', 'Technique' and 'Tool'	420
L.2	Method Indexes	421
L.2.1	Volume 1	421
L.2.2	Volume 2	422
L.2.3	Volume 3	424
M	List of Papers on Verification	431
N	Biography	433
O	References	437
	Last page	453

THE THESIS

The thesis evolves around the following claimed contribution to a methodology for the general field of software engineering:

- that *domain engineering* — for which a set of novel principles and techniques are claimed — is an indispensable part of software; Chap. 10
- that requirements engineering — for which a set of novel principles and techniques are likewise claimed — must *be based on a carefully developed and documented domain models*; Chap. 10
- that descriptions, prescriptions and specifications, both informally, in the form of narratives, and formally, in the form of mathematical statements, be built up around a *description ontology of simple entities, functions, events and behaviours*; Chap. 7
Chap. 6
- that descriptions (etc.) initially be in the form of *abstract models of phenomena and concepts* in which careful, separate consideration has been paid to the *pragmatics, semantics and syntax* of these, and Chap. 8
Chap. 3
- that *documentation* be extensively provided for all aspects of development. Chap. 11

Complementing the above claimed contributions to the methodology of developing large scale software systems is the emphasis on

- clearly enunciated *methodology principles, techniques and tools*; Chap. 2
- a clearly understood *separation of issues of computer versus issues of computing sciences*; Chap. 4
- a clear mandate that *the software engineer is comfortably “at home” with basic mathematical notions* (sets, lambda-Calculus, algebra and mathematical logic); and Chap. 5
- that *the software engineer is competent in formulating abstract models both informally and formally*. Chap. 9

The Thesis

1.1 On Software Engineering

The focus of our study is software engineering — trying to contribute an answer to the question: “what is software engineering”. Hence we need first delineate what we mean by that term before we study it — or, rather, as a result of practicing and studying the field, we came up with the delineation found in [87–89]. That three volume book claims to be about software engineering. We must therefore,

- not only delineate the concept of ‘software engineering’ properly, but we must argue
- that these volumes do indeed cover substantial aspects of that software engineering concept in a proper manner,
- and that our coverage contributes a number of new ideas to software engineering.

Characterisations of the ‘software engineering’ concept are found in

- Vol. 1 on Pages V–VI, XIII, and 4–7, and in
- Vol. 3 on Pages 3–6.

Pages 3–5 of Vol. 3 brings characterisations by authors of popular textbooks as well as by some recognised researchers of software engineering.

Section 10.6 (of the present ‘Thesis’) will review these characterisations critically, and will similarly (i.e., critically) review whether these and our delineations (Vol. 1, Pages V–VI, XIII, and 4–7 as well as Vol. 3, Page 6) are indeed properly handled by [87–89]. Section 12.1.1 will critically review our “grand” omission of verification and Sect. 12.1.2 will critically review our omission of software engineering management, whether of development or of products.

1.2 The Three Volume Book: Software Engineering

For the sake of good order we list proper references to the three particular volumes that are at the basis of this thesis:

- [87]: *Software Engineering*, Vol. 1: Abstraction and Modelling, *Texts in Theoretical Computer Science, the EATCS Series*. Springer. ISBN 3-540-21149-7. xxxix + 711 pages. Published Dec. 9, 2005.
- [88]: *Software Engineering*, Vol. 2: Specification of Systems and Languages, *Texts in Theoretical Computer Science, the EATCS Series*. Springer. ISBN 3-540-21150-0. xxiv + 779 pages. Published Feb. 9, 2006. Chapters 12–14 of this volume are primarily authored by Christian Krog Madsen.
- [89]: *Software Engineering*, Vol. 3: Domains, Requirements and Software Design, *Texts in Theoretical Computer Science, the EATCS Series*. Springer. ISBN 3-540-21151-9. xxx + 733 pages. Published March 9, 2006..

A total of lxxxxiii + 2223 = 2316 pages.

1.3 Textbook Versus Research Monograph

The three volumes, [87–89], that are at the basis of this thesis, were written in order to summarise more than 30 years of research. The outward form of [87–89] is that of a textbook. The principles and techniques propagated by these texts reflect those many years of research. It is the purpose of the present thesis document to claim and to show so. Hence the view taken in this thesis is that the three volumes [87–89], as a whole, also constitutes a research monograph.

This is not a shift of convenience or of opportunism. This is merely a reflection on two issues. (1) We are putting forward a thesis for the defense of the degree of Dr.Techn., hence a thesis that has engineering orientation and goals of engineering. And (2) our scientific discipline is that of computing science (CS), not computer science (CS). These two concepts, CS&CS (!), are treated in Sect. 1.4.3 and in Chap. 4.

1.4 The Thesis and Its Issues

So what, then, is the thesis of the three volumes and, hence of the present document ? We express this thesis in two parts:

The Thesis Base: The basis for our thesis is that in order to conduct professional, responsible software development the software engineer must possess a number of diverse skills and apply these in a methodological manner.

These skills, we claim, are hinted at by the following terms — listed in no particular order of priority:

- ⊕ mathematics,
- ⊕ computer and computing science,
- ⊕ abstraction and modelling,
- ⊖ refinement calculi, formal testing and verification,
- ⊕ semiotics: pragmatics, semantics and syntax,
- ⊕ the ontological notion of specifying concepts,
- ⊕ the triptych notion of software engineering,
- ⊖ knowledge science & engineering,
- ⊕ documentation and
- ⊖ software project and product management

The concluding part of the thesis formulation now follows:

The Thesis Implementation: The thesis is now that I have contributed in two ways to secure the base conjecture:

1. I have provided an overall, pragmatic and logically coherent framework for software engineering in the form of the triptych concept — where that framework is made up from the constituent issues listed below, and
2. I have contributed to several of that frameworks' constituent disciplines ("the issues").

The three circled minus (\ominus) items above are not seriously covered in [87–89] and will not be covered in this document, but two of them ((i) refinement calculi, formal testing and verification, and (ii) software project and product management) will be discussed in Chap. 12. The 'software project and product management' topic is, however, now covered in Appendix E.

In the following subsections I highlight issues related to the above 'thesis' components. In subsequent chapters these issues are then subjected to a closer analysis. In summary the issues center around:

1. Method and Methodology Sect. 1.4.1 and Chap. 2
 The contribution here is that no (method cum software engineering) textbooks, and hence no claimed methods have heretofore spelled out the principles, techniques and tools of their method and that I do so in a rather detailed, consistent and reasonably complete manner. (I also claim to have provided, in Vol.1, Sect. 1.5 and in Vol.3, Chap. 3, a proper, comprehensive definition of the 'method' and 'methodology' concepts.)
2. Semiotics: Pragmatics, Semantics and Syntax Sect. 1.4.2 and Chap. 3
 The contribution here is that no (method cum software engineering) textbooks, and hence no claimed methods have heretofore

covered semiotics issues and shown their relevance to software engineering, and that I do so in a rather consistent and reasonably complete manner.

3. Computer and Computing Science Sect. 1.4.3 and Chap. 4

The contribution here is that no (method cum software engineering) textbooks, and hence no claimed methods have heretofore spelled out the distinction, such as we make it, between computer and computing science, nor have they emphasised the need for the practicing software engineer to possess a firm, but not necessarily deep base in computer science and a reasonably firmer base in computing science — such as I provide it in [87–89].

4. Mathematics Sect. 1.4.4 and Chap. 5

The (minor) contribution here is that only few (method) textbooks, and hence few claimed methods have heretofore bothered to make sure that a mathematics basis is provided — such as I do.

5. Entities, Functions, Events and Behaviour Sect. 1.4.5 and Chap. 6

This is a part of the ontology of specifying concepts.

The not insignificant contribution here is that I provide a simple description ontology where most (method cum software engineering) textbooks do not even bother to cover description principles and techniques.

Chapter 6 also remedies what we now consider to be an inadequate treatment of the entities, functions, events and behaviour ontology. The remedy amounts to entities and properties being the base and functions, events and behaviours being derived from that base. Thus properties (which in my previous treatments were, incorrectly, we now think, seen as attributes of type and value) are seen as adjunct, that is, inseparable from entities.

6. Phenomena, Concepts and Descriptions Sect. 1.4.6 and Chap. 7

This is another part of the ontology of specifying concepts.

The not insignificant contribution here is that I further elaborate on the simple description ontology where most (method cum software engineering) textbooks hardly cover these issues. Jackson's [206] is the sole exception.

7. Abstraction and Modelling Sect. 1.4.7 and Chap. 8

Although also a part of the ontology of specifying concepts we treat this issue separately.

The contribution here is that very few (method cum software engineering) textbooks, and hence few claimed methods have focused so intensely on abstraction in connection with modelling and that I do so in a rather consistent and reasonably complete manner — while suggesting new ways of conceptually structuring abstract models (hierarchy and composition, denotation and computation, configurations: contexts and states, temporality and

spatiality, etc. — issues covered in Vol. 2). Included in the abstraction and modelling issue is that I show how to “UML-ise” formal techniques rather than formalise ‘UML’ (modularity, Petri nets, message and live sequence charts, and statecharts — issues also covered in Vol. 2).

8. Informal and Formal Specifications Sect. 1.4.8, Chap. 9
This is yet another part of the ontology of specifying concepts.

The (minor) contribution here is that I very much emphasise the need for both informal and formal specifications — a need which is not recognised by (method cum software engineering) textbooks, and hence by few claimed methods. I also emphasise the interplay between informal and formal specifications. This is practised and “preached” by programming methodologists, but rarely subject to a more systematic treatment — as here.

9. The Triptych of Software Engineering Sect. 1.4.9, Chap. 10

The contribution here (a major one) is that no (method cum software engineering) textbook, and hence no claimed methods have heretofore introduced or covered the notion of domain engineering and the reliance of requirements engineering on domain models. I do so in Vol. 3 and rather extensively so.

The books by Jackson [206,207] do indeed cover a number of facets of what Jackson call ‘the environment’, but not with rigour and ‘consequentiality’ of this thesis.

10. Documentation Sect. 1.4.10, Chap. 11
This is our final part of the ontology of specifying concepts.

The contribution here is that no (method cum software engineering) textbook, and hence no claimed methods have heretofore spelled out nor, really, emphasised the pivotal importance of documentation let alone the various types and kinds of documents that need be produced as an integral part of software development, and that I do so in a rather consistent and reasonably complete manner.

11. Omissions Sect. 1.4.11, Chap. 12

There are three disciplines with which a professional software engineer must be reasonably familiar but which are neither treated here nor in [87–89]. Those are the disciplines of

- formal testing, model checking and verification,
- knowledge science & engineering and
- philosophy of science.

Formal testing, model checking and verification is omitted simply because we are ourselves not deeply enough knowledgeable about this field and because

we find that we cannot formulate succinct principles and techniques and recommend such tools that will let the software engineer perform these tasks short of heroism. See Sect. 1.4.11 and Chap. 12 for more.

With respect to philosophy, some familiarity with the issues of the sub-disciplines of

- epistemology (the study of knowledge and justified belief),
- mereology (theory of parts and wholes),
- ontology (specification of a conceptualization),

to us, appear relevant to the practicing software engineer as well as the computing scientist who wishes to contribute to the foundations of software engineering. *We shall, however, not bring the issue of philosophy in connection with computing science and software engineering up in this thesis as it has not been brought up in [87–89].*

1.4.1 On Method and Methodology

The concepts of method and methodology and the constituent concepts of method principles, techniques and tools are covered in

- Vol. 1, Sect. 1.5, Pages 31–33, and
- Vol. 3, Chap. 3, Pages 93–101.

In Chap. 2 we define what we mean by a method, by methodology and by attendant method principles, techniques and tools. And then we claim to have systematically applied these concepts by highlighting, throughout all three volumes of [87–89] the enunciated principles, techniques and tools.

The thesis is that without a proper understanding and grasp of the concepts of method, methodology, principles, techniques and tools, the software engineer is unnecessarily and detrimentally constrained in achieving good designs. We claim that Vols. 1–3 [87–89] provide the basis for obtaining such an understanding and acquiring such a grasp.

Section 2.3 on page 25 critically reviews whether we have achieved what we desired.

1.4.2 On Semiotics

The concept of semiotics and the constituent concepts of pragmatics, semantics and syntax are covered in

- Vol. 1, Sect. 1.6.2, Pages 34–48, and
- Vol. 2, Part IV, Chaps. 6–9, Pages 143–235.

In Chap. 3 we define what we mean by semiotics and the constituent concepts of pragmatics, semantics and syntax. And then we claim to have systematically applied these concepts by highlighting, throughout all three volumes of [87–89] the enunciated distinction between pragmatics, semantics and syntax.

The thesis is that without a proper understanding and grasp of the concepts of semiotics, pragmatics, semantics and syntax, the software engineer is unnecessarily and detrimentally constrained in achieving good designs. We claim that Vols. 1–3 [87–89] provide the basis for obtaining such an understanding and acquiring such a grasp.

Section 3.4 on page 30 critically reviews whether we have achieved what we desired.

1.4.3 On Computer and Computing Science

The distinction between computer and computing science is first highlighted in

- Vol. 1, Sect. 1.1, Pages 3–4.

In Chap. 4 we define what we mean by computer and computing science. And then we claim to have systematically emphasised, throughout all three volumes of [87–89] the concept of computing science.

The thesis is that without a proper understanding and grasp of the concepts of computer science and computing science (including the importance of their distinction), the software engineer is unnecessarily and detrimentally constrained in achieving good designs. We claim that Vols. 1–3 [87–89] provide the basis for obtaining such an understanding and acquiring such a grasp.

Section 4.3 on page 35 critically reviews whether we have achieved what we desired.

• • •

Thus we critically review the various places in Vols. 1–3 where we, in contrast to conventional treatments, cover computational models (i.e., models usually covered in some “classico-mathematical” style) from the semiotic and formal specification points of view. We feel compelled to make some rather negative observations on the use of the “classico-mathematical”, i.e., computer science, style rather than the, to us, far more convincing and hence appropriate computing science style of formal language cum system specification. Thus, except for some sections of Vol. 2’s Chaps. 14, the computing science view is the view which dominates [87–89].

1.4.4 On Mathematics

Basic aspects of discrete mathematics are covered in

- Vol. 1 Chaps. 2 to 4 and 6 to 9: Numbers, sets, Cartesians¹, functions, λ -calculus, algebras and mathematical logic.

In Sect. 5.1 (starting Page 41) we motivate the need for the use of mathematics in software engineering, in particular discrete mathematics. Section 5.3 (starting Page 43) then outlines the rôle of a number of disciplines within discrete mathematics. We then outline the use of mathematics in the book, Sect. 5.4, before reviewing that use critically, in Sect. 5.5.

The thesis is that without a proper understanding and grasp of the notions of discrete mathematics, the software engineer is unnecessarily and detrimentally constrained in achieving good designs. We claim that Vol. 1 [87] provides the basis for obtaining such an understanding and acquiring such a grasp.

We do not critically review our use and propagation of mathematics in [87–89].

1.4.5 On Entities, Operations, Events and Behaviour

The notions of simple simple entities, functions, events and behaviour are covered

- concretely in Vols. 1–2,
- and more conceptually in Vol. 3, Chap. 5
- as well as further exemplified in the rest of Vol. 3.

Simple entities (in domains and requirements) eventually, and typically, “end up” as data “inside” computers. (ii) Operations eventually, and typically end up as procedures (“methods”, code) “inside” computers. (iii) Events reflect interaction with the environment of computers (or interactions between computing processes). And (iv) behaviours (strands of actions and events, with actions being invocations of functions) reflect computing (and data communication network) processes.

The thesis is that without a proper understanding and grasp of the notions of simple entities, operations, events and behaviour, the software engineer is unnecessarily and detrimentally constrained in achieving good designs. We claim that Vols. 1–3 [87–89] provide the basis for obtaining such an understanding and acquiring such a grasp.

¹The use of the term ‘Cartesians’ may appear unfamiliar to some readers. By a Cartesian, after the French mathematician René Descartes (1596–1650), we understand a fixed grouping, a structure, of entities. In some programming languages Cartesians are called records.

Chapter 6 will review the concepts of simple entities, operations, events and behaviour (Sect. 6.2) and their deployment in the book (Sect. 6.2).

We repeat what was first mentioned in Item 6 on page 6:

Chapter 6 remedies what we now consider to be an inadequate treatment of the simple entities, operations, events and behaviour ontology. The remedy amounts to simple entities and properties being the base and operations, events and behaviours being derived from that base. Thus properties (which in our previous treatments were, incorrectly, we now think, seen as attributes of type and value) are seen as adjunct, that is, inseparable from entities.

Section 6.3 on page 59 critically reviews the various places in Vols. 1–3 whether we have achieved what we desired.

1.4.6 On Phenomena and Concepts and Their Description

The notions of phenomena and concepts and their description are covered

- systematically in Vols. 1–2 and
- conceptually in Vol. 3, Chap. 5
- and further illustrated in the rest of Vol. 3.

Phenomena is what is physically manifest. They are specific instances of what can be seen, heard, smelled, tasted, felt or otherwise physically measured.² Concepts “lift” such physical instances to classes of phenomena of “same kind”. From concepts, by further “lifting”, one can form further concepts. Phenomena and concepts are what we model: in domain descriptions, in requirements prescriptions and in software designs.

The thesis is that without a proper understanding and some grasp of the notions of phenomena and concepts the software engineer is unnecessarily and detrimentally constrained in achieving good designs. But the thesis goes further: From the modelling of phenomena and concepts in domain descriptions via the modelling of these notions in requirements prescriptions and in software designs there must be a clear line. Coding or code optimisation which “blurs” the original domain phenomena and concepts are here conjectured to be “an evil”. We claim that Vol. 3 [89] provides the basis for obtaining and proper understanding and acquiring such a grasp.

Section 7.4 on page 66 critically reviews the various places in Vols. 1–3 where we emphasize the distinctions between phenomena and concepts.

• • •

The concepts of descriptions, prescriptions and specifications are covered in

²The objectivity of what human senses can register is a problem. We shall not cover this problem in [87–89]. The issue borders to philosophy: “what there exists”.

- Vol. 3, Chaps. 5–7.

The term specification is the general term. The term description is used for specifications of domains and the term prescription is used for the specifications of requirements. Thus we reserve the use of the term specification for that of software design.

Descriptions, prescriptions and specifications can either be expressed informally or formally, or both. We shall advocate the latter.

The thesis is that without a proper understanding and grasp of the concepts of descriptions, prescriptions and specifications, the software engineer is unnecessarily and detrimentally constrained in achieving good designs. We claim that Vol. 3 [89] provide the basis for obtaining such an understanding and acquiring such a grasp.

Section 7.4 on page 66 also critically reviews the various places in Vols. 1–3 where we advance these complementary forms of expression: informal and formal.

1.4.7 On Abstraction and Modelling

The concepts of abstraction and of property- and model-oriented abstractions are first covered in Vol. 1, Chap. 12:

- Abstraction: Sect. 12.1, Pages 232–235, and
- Property-oriented Modelling: Sects. 12.2–.3, Pages 235–250 respectively
- Model-oriented Modelling: Sects. 12.3–.4, Pages 241–254.

and then applied throughout the three volumes [87–89].

Usually formal specification languages are either typically property-oriented, like `CafeOBJ` [147, 148, 161, 162] and `Cas1` [24, 142, 241, 242], or are typically model-oriented, like `B` [1, 131], `VDM-SL` [111, 112, 157, 158] and `Z` [186, 187, 283, 284, 297]. The formal specification language `RSL`, however, permits expressing both properties and models — depending on the extent to which sorts and axioms are used, or not used, respectively.

That is why we have chosen the `RAISE` Specification Language, `RSL`, as our main tool for formalisation.

The thesis is that without a proper understanding and grasp of the concepts of property-orientedness and model-orientedness, the software engineer is unnecessarily and detrimentally constrained in achieving good designs. We claim that Vols. 1–3 [87–89] provide the basis for obtaining such an understanding and acquiring such a grasp.

Section 8.3 on page 72 critically reviews the various places in Vols. 1–3 where we alternate between property- versus model-oriented abstractions, i.e., models.

• • •

There is an altogether different dimension to the development and presentation of abstract models. Besides the conventional choice amongst set-, Cartesian-, list-, map- and function-oriented model-oriented models (covered in Vol. 1, Chaps. 11-17), we claim that there is additionally the possibility of emphasising hierarchy or composition, denotation or computation, different configuration (context and state) styles, and different space/time aspects of phenomena and concepts being modelled. Vol. 2 Chaps. 2-5 covers these aspects.

The thesis is that without a proper understanding and grasp of the concepts of specific model-oriented choices, the software engineer is unnecessarily and detrimentally constrained in achieving good designs. We claim that Vols. 1-2 [87,88] provide the basis for obtaining such an understanding and acquiring such a grasp.

Section 8.3 on page 72 also critically reviews the various places in Vols. 1-3 where we choose between such model-oriented abstractions.

1.4.8 On Informal and Formal Specifications

The concepts of informal (rough sketch, terminology, narrative) and formal descriptions, prescriptions and specifications are specifically covered in

- Vol. 3, Chap. 2, Sect. 2.5, Pages 70-84.

By informal expression we mean a precise expression in some natural, but possibly professional, that is, domain specific language — possibly “mingled” with “easy-to-grasp” diagrams, drawings, pictures. By a natural language we mean such as for example English.

By formal expression we mean an expression in a formal, possibly diagrammatic language, and at least a language that has a clear formal syntax and formal semantics and possibly a proof system.

The thesis is that without a proper understanding of the concepts of informal and formal expressions and of the joint necessity of informal and formal expressions the software engineer is unnecessarily and detrimentally constrained in achieving good designs. We claim that Vols. 1-3 [87-89] provide the basis for obtaining such an understanding and acquiring such a grasp.

We claim, moreover, that this is a first textbook on software engineering which systematically show the application of formal techniques in all phases, stages and steps of software development.

The *sans serif slanted text* of the above indented claim shall be seen on the background that some “prominent” textbooks on software engineering may indeed mention ‘formal methods’, “tucked away” in some isolated chapter, but that these same textbooks do not take the formal techniques and tools

that they refer to serious — as there are no examples of the use elsewhere in the textbook !

Section 9.3 on page 74 critically reviews the various places in Vols. 1–3 where we advance these complementary forms of expression: informal and formal.

1.4.9 On The Triptych of Software Engineering

The concept ‘the triptych of software engineering’, with its emphasis on the initial phase, domain engineering, is covered in

- Vol. 1, Chap. 1, Sect. 1.2, Pages 7–13
- Vol. 3, Chaps. 1, 8, 17 and 26.

The dogma behind this triptych is as follows:

- Before software can be designed (i.e., specified)
 - ★ we must understand its requirements.
- And before we can prescribe requirements
 - ★ we must understand the domain.

The dogma implies:

- A domain description is a description of the application domain, the universe of discourse, free from any reference to the requirements (for software) that may be about to be established and certainly, and, even more so, without any reference to that software.

We claim that a major contribution of [89] is that of “prefixing” conventional software development with an initial phase of domain engineering. Conventionally it is claimed that requirements engineering is a first technical phase of software development. With [89] we claim to have shown that that is insufficient: that a serious and laborious, but beautiful prior phase of domain engineering is indispensable.

Appendix D presents a capsule introduction to domain and requirements engineering. Rather than studying [89, Parts IV and V] we recommend studying instead Appendix D.

The thesis is that without a proper understanding of the concept of the domain, requirements and software design triptych and of the joint necessity of the domain, requirements and software design triptych, the software engineer is unnecessarily and detrimentally constrained in achieving good designs. We claim that Vol. 3 [89] provides the basis for obtaining such an understanding and acquiring such a grasp.

Section 10.4.3 on page 86 (of this thesis) will critically review the triptych concept.

We will, throughout Chap. 10, point out which aspects of [89, Parts IV and V] we believe are made more clear by Appendix D [95, to appear].

1.4.10 On Documentation

The concept of documentation is — and the principles, techniques and tools of creating and handling documents are — covered in

- Vol. 3, Chap. 2.

By a document we mean a text, informal or formal text, on paper or readable (say electronically) as would be a paper document. Domain descriptions, requirements prescriptions, software designs, their informative introduction or their informal or formal analysis are here represented as documents. In the triptych approach to software development we set great store by documentation.

The thesis is that without a proper understanding and grasp of the concepts of proper informative, specification and analytic documentation, the software engineer is unnecessarily and detrimentally constrained in achieving good designs. We claim that Vol. 3 [89] provide the basis for obtaining such an understanding and acquiring such a grasp.

Section 11.3 on page 92 critically reviews our treatment of ‘documentation’.

1.4.11 On Omissions

Chapter 12, “On Omissions”, will critically review three “omissions” from [87–89]. They are:

- verification and Sect. 12.1.1
- management. Sect. 12.1.2

We shall argue in the above-referenced sections why these three issues have not been given a serious treatment in [87–89].

Verification

Verification is used here in the broadest sense of:

- testing (informal and formal),
- model checking and
- formal verification (i.e., proofs) of theorems

about descriptions, prescriptions and specifications or about implementation properties.

I decided as from 2003 to omit treating verification for the following reasons:

- The state-of-the-art of testing, especially formal testing and formal verification was, and I believe still is not settled, that is, not stable: new concepts, new tools are emerging at almost regular intervals, many enough that any treatment would soon be outdated.
- Even with the better of these formal testing and formal verification techniques and tools it still, so I think, requires an heroic effort of the part of any practicing, well-educated, including formal testing and formal verification-trained software engineer to carry through formal tests and formal verifications for other than trivial developments.

It is a subsidiary aim of [87–89] to cover only those areas of (formal) software engineering where there is at least some guarantee that the material that is put forward can be learned by a fair percentage of today’s BSc and MSc students and can be deployed by them without heroic efforts.

Management

Management is used here in the broadest sense of:

- software product management: the study and decisions concerning which software products
 - ★ to develop
 - ★ market and
 - ★ service; and of
- software development management:
 - ★ the planning and resourcing of (future) development, and
 - ★ the initiation, monitoring and control of (actual, ongoing) development.

In Appendix E, ‘Believable Software Management’ we cover essential parts of software development management and overview some issues of software product management. We consider Appendix E to be part of this Dr. Techn. thesis.

1.5 The Real Thesis

The sum total of the conjectures, the claims, the sub-theses, made above and expanded upon in Chaps. 2–12, is this:

No heroism: To do proper, professional software development shall not need heroisms by the software engineers.

What are now heroic achievements must become routine !³

I claim that Vols. 1–3 [87–89] provide a basis for making software development less dependent on “super-programmer” heroes !

³This sentence is lifted from a 9 February 2007 e-mail from Sir Tony Hoare on the occasion of some scientists having proved the correctness of an implementation of a smart card software package.

I claim that the principles, techniques and tools expounded in [87–89] — methodologies that are now carefully reviewed in this Dr.Techn. thesis — are necessary prerequisites for proper software development and can be learned by a sufficient number of MSc candidates, in any country, yearly.

Notice that I am not claiming that these principles, techniques and tools are sufficient. Missing are the formal verification principles, techniques and tools (as well as the overall management principles, techniques and tools). The former are, I think, today, August 1, 2008, in such a state that their proper deployment does indeed require heroism.

1.6 Treatment

The presentation of this thesis relies extensively on the reader have ready access to all of [87–89]. We shall assume that when we refer to a chapter, a section, or otherwise, in [87–89], then the reader will have first read the referenced part before going on with the study of this thesis.

THE TEN THESIS ISSUES

The ten issues surveyed in Chap. 1 are examined in the next ten chapters.

The order of this treatment is different from the (unnumbered) ●-listed order on Page 1. The reason is that the earlier chapters cover issues that, albeit implicitly, enter into later chapters' coverage and are thus a prerequisite for these.

On Method and Methodology

For our definitions (characterisations) of the concepts of ‘method’ and ‘methodology’ we refer to

- ★ Vol. 1, Sects. 1.5.1–1.5.2, Page 32 of [87], and
- ★ Vol. 3, Chap. 3, Pages 93–96 of [89].

We bring a set of these characterisations:

1. By a *method* we understand a set of principles for selecting amongst, and applying, a set of designated techniques and tools that allow analysis and construction of artifacts.
2. By *methodology* we understand the study of and knowledge about methods.

For our definitions (characterisations) of the concepts of ‘principle’, ‘technique’, and ‘tool’ we refer to

- ★ Vol. 1, Sect. 1.5, Page 33 of [87], and
- ★ Vol. 3, Chap. 3, Pages 97–98 of [89].

We repeat the Vol. 3, Chap. 3 characterisations here:

3. A *principle* is an accepted or professed rule of action or conduct, . . . , a fundamental doctrine, right rules of conduct.¹
4. A *technique* is the, or a, specific procedure, routine or approach that characterises a technical skill.
5. A *tool* is an instrument for performing mechanical operations, a person used by another for his own ends, . . . , to work or shape with a tool.

We refer the reader for the characterisations of *analysis* and *synthesis* given in Vol. 3’s Sects. 3.3.3–4 (Pages 97–98) [89].

Finally we refer to Vol. 3’s Sect. 3.4.1 (Pages 99) for the meta-principles governing our intended use of method.

¹Jess Stein (Ed.): *The Random House American Everyday Dictionary*, Random House, New York, N.Y., USA, 1949, 1961.

2.1 Background

Two background issues must be dealt with before we deal with the rôle of the concept of 'method' in [87–89]:

- The choice of our definition of the concept of 'method' in the context of standard dictionary definitions.
- The background for the emphasis put, in [87–89], on this concept of methodology and its attendant concepts of 'principle', 'technique' and 'tool'.

2.1.1 Definitions of The Concept of 'Method'

The concept of 'method' is not an easy one to define. Our characterisation is just that: a kind of definition, one that we shall adhere to. But there could be other characterisations. We refer to Appendix Sect. L.1.1 (starting Page 418), for characterisations culled from the conventional references.

Discussion of 'Method' Definitions

We relate our definition of 'method', Item 1 on the preceding page, to those of Appendix Sect. L.1.1 on page 418.

- The OED definition [Items 1(a)–1(c)] is commensurate with our definition.
- The Cambridge definition [Item 2] sharpens the OED Item 1(a) definition, but is still commensurate with our definition.
- And so are the three parts [Items 3(a)–3(c)] of The American Heritage Dictionary definition.
- And likewise for the nine items of the Merriam Webster Unabridged definition [Items 4(a)–4(g)]; Item 4(g) may surprise some — but really, we are, in a sense, setting up a table of contents of principles, techniques and tools.

Merriam Webster Synonyms [Items 5(a)–5(f)] elaborate synonyms for the term 'method'.

'Method' Conclusion

The above discussion of our definition versus the standard reference definitions of the concept of 'method' amounts to an argument that our definition is acceptable and useful — it seems that our characterisation is downward compatible with those of the standard dictionaries..

2.1.2 Why Emphasize the Concept of ‘Method’?

We emphasise the concept of ‘method’ for the following reason:

- The book [87–89] is about engineering.
- An engineer, we claim, expects to work according to some method.
- A group of engineers collaborating on the same target construction, e.g., the same software development, we claim, had better follow the same, commonly understood method.
- We claim that most, if not all textbooks on software engineering fail to enunciate what a method is, and, as a result, do not explicate the constituents of a method (as here: principles, techniques and tools). We claim to “repair” those omissions. And we claim to have done a first, not insignificant step towards that.
- In fact, a derivative reason for emphasising the, or at least, a concept of ‘method’ has been to develop, as I wrote the book [87–89], to unravel, what it meant to delineate and “populate” a concept of ‘method’.

So, as “revealed” by the last item just above, our coverage of ‘method’, as manifested in the very many enunciations of specific principles, specific techniques and specific tools, should, perhaps, be seen with a grain of salt: Much work, we think, is still needed to (even) more precisely develop, “mature” the concept of ‘method’, and to (even) more succinctly express and sub-categorise the specific principles, specific techniques and specific tools.

2.2 Coverage in Book

The book, that is, the three volumes [87–89], more-or-less systematically “afix” (suffix) most major sections of most chapters, or just these chapters, with highlighted paragraphs of **Principles of X**, **Techniques of Y**, and **Tools of Z**. Section L.2 (Pages 421–429) lists all occurrences of such highlighted paragraphs.

A study of whether these specifically highlighted paragraphs do indeed cover all the principles, techniques and tools that have indeed been covered by these sections and chapters reveals the following:

1. Usually no (specific) principles, techniques and tools are identified in introductory chapters of the three volumes.
2. In Vol. 1 [87] we do not identify (specific) principles, techniques and tools in Chaps. 2–6: These chapters cover what is assumed to be classical mathematical notions. We have naturally refrained from enunciating principles etc. for using mathematics !
3. In Vol. 1 [87] we (almost) systematically “attach” a set of (explicitly enunciated) principles, techniques and tools to every abstract data type being otherwise covered. We say ‘almost’ because there seems to be a principle

for process modelling (Page 421, 21.2.5), but no corresponding (explicitly enunciated) techniques and tools — although there should be (and although they are obvious!).

4. In Vol. 2 [88] we (almost) systematically “attach” a set of (explicitly enunciated) principles
 - to each major abstract concept (Chaps. 2–4’s Hierarchy and Composition, Denotation and Computation, and Configuration: Contexts and States),
 - to each major modelling concept (Chaps. 6–9’s Semiotics components [Pragmatics, Semantics and Syntax], and Chaps. 10–11, and 13–15’s Modularity, and Automata and Machines, resp. MSCs and LSCs, Statecharts and Quantitative Models of Time), and
 - to the programming language implementation approaches (Chaps. 16–19’s applicative/functional, imperative, modular and parallel programming language interpreters and compilers).

The principles are not all fully “matched” by techniques and tools. Some of these ought have been explicitly enunciated. And, it seems, that I missed out on attaching explicitly enunciated principles, techniques and tools for (Vol. 2, Chap. 12) Petri Nets [quite a mistake !].² (The text of the chapter, as elsewhere, has these, but, to maintain the whole idea of ‘method’, they ought have been highlighted.)

5. Volume 3 [89] is really the major volume wrt. ‘methodicity’: the principles, techniques and tools of being methodical. There are indeed very many explicit formulations of principles, techniques and tools. We will not go into analytic detail here other than saying the following:
 - (a) The principles, techniques and tools of Vols. 1–2 were mostly about general abstraction and modelling methodology (the choice of specification, syntax and semantics) whereas those of Vol. 3 are primarily about the engineering choice of phases, stages and steps of development, that is, the principles etc. are pragma-driven.
 - (b) Whereas the principles, techniques and tools of Vols. 1–2 were developed by many researchers and practitioners over the last more than 30 years those of Vol. 3 were, to a sizable extent developed by us as from the early 1990s. Those of Vols. 1–2 might have been developed by many different researchers and practitioners, but we think we can claim that our contribution here has been to highlight them as principles, techniques and tools, that is, to enunciate these.
 - (c) In a sense one could say that the principles, techniques and tools of Vols. 1–2 are rather stable (in the sense also of being now generally accepted), but that those of Vol. 3 are less so: several principles and techniques are put forth in the comprehensive framework of [87–89] for the first time.

² Dines: Should you list some such principles here!

2.3 Critique

2.3.1 On a Multitude of Principles, Techniques and Tools

The fact that we have put so much emphasis on clarifying, in [87, 89], the concept of ‘method’, and on summarising in almost every chapter of [87–89] the ‘principles’, ‘techniques’ and ‘tools’ implied by those chapters, may leave the impression that it is relatively easy to learn and to adhere to these methodological issues.

It is not !

There may be two reasons for this:

- It seems almost “against human nature”: One simply cannot remember all these principles, techniques and tools, and one “sort of gets tired” following these methodological mandates slavishly.
- And it likewise seems, to some engineers, to “stand in the way of creativeness”.

What is our answer to this ? If we maintain the importance of these methodological issues, and we do, then we must provide an answer ! And we do:

- The book [87–89] is about principles, techniques and tools, and these principles, techniques and tools are there whether we bother or not — hence they must be emphasised for us to claim that we treat software engineering professionally !
- Adherence to principles, techniques and tools can be facilitated by proper (computerised) tools. If in every phase, at every stage and in every step of development the software engineer — who is anyway using development tools³ — informs the overall development tool where development is taking place and with respect to what, then this overall development tool can list potential and relevant principles, techniques and tools that ought be of interest to the developer.

2.3.2 On an Ontology of Principles, Techniques and Tools

Almost every chapter of [87–89] contains explicit enunciations of principles, techniques and tools.

We could have wished to have performed a close analysis of around a hundred and fifty (150) principle enunciations, around 120 technique enunciations, around 60 tool enunciations. The analysis should have been guided by a desire to find commonalities amongst the many principle enunciations,

³This reference to “development tools” is somewhat “dubious” as we have yet (in a Software Engineering volume on ‘Management’) to outline which such tools are warranted by our triptych.

amongst the many technique enunciations, amongst the many tool enunciations — and, possibly some meta-principles across these (approximately) 320 method constituents.

We have not done so. And that, we think, is a drawback.

2.4 Conclusion

From Sect. 1.4, Item 1 on page 5 we quote (slightly edited):

The contribution of [87–89] is that no (method cum software engineering) textbooks, and hence no claimed methods have heretofore spelled out the principles, techniques and tools of their method and that we have done so in a rather detailed, consistent and reasonably complete manner.

We thus claim that we have shown that [87–89] establish an explicit ‘methodology’ base for software engineering, one that has not been elucidated, by other authors, to anyway near the degree done in [87–89] — perhaps with the exception of Michael Jackson’s works [203–207] which, albeit, “covers less ground”.

On Semiotics

For our definitions (characterisations) of the concepts of ‘semiotics’, ‘pragmatics’, ‘semantics’, and ‘syntax’ we refer to

- Vol. 1, Sect. 1.6.2, Pages 34–38, and
- Vol. 2, Chaps. 6–9 (i.e., Part IV), Pages 145–239.

We bring an edited set of these characterisations:

1. Semiotics = Pragmatics \oplus Semantics \oplus Syntax.
2. **Pragmatics** is the study and practice of the factors that govern our choice of language in social interaction and the effects of our choice on others [144] (Chap. 6, Vol. 2). Thus, by the *pragmatics* of a language we mean its use in social context: why a particular expression is used, what “ultimate” motive lies (seems to lie) behind an utterance, an expression? Why a particular expression used? What “ultimate” motive lies (seems to lie) behind an utterance, an expression. Pragmatics is thus concerned with bridging the explanatory gap between sentence meaning and speaker’s meaning.
3. **Semantics** is the study and knowledge (including specification) of meaning in language [144] (Chap. 7, Vol. 2). *Semantics* is about the meaning of what we express syntactically.
4. By **syntax** we understand (i) the ways in which words are arranged (cf. Greek: *syntaxis*: arrangement) to show meaning (cf. semantics) within and between sentences, and (ii) rules for forming *syntactically correct* sentences [144]. Syntax is about how we can, in our case, write down specifications: rules of form, basic forms and their proper compositions. These rules for formal languages are to be of such a nature that the forms, that is, the language expressions, can be analysed, and such that, from the analysis, one can ‘construct’ (construe) the meaning.

3.1 Mathematical Semiotics

We shall use the term mathematical semiotics to cover those aspects of semiotics which can possibly be expressed mathematically. The resulting formulas need not, however, be computable.

The emphasis in [87–89] has been on formalisations in the form of various kinds of formal specification languages (to wit: BNF, RSL, FOL [Axioms]).

3.2 Why the Semiotics Triplet Emphasis ?

Syntax is what you see! The meaning behind it is often obscured. From 45 years use of software, of so-called user interfaces, and from 45 years among coders, programmers and software engineers, a number of lasting impressions have left their indelible mark. These coders, programmers and software engineers have spent more time — considerable time — wavering with themselves and haggling with colleagues about form (i.e., syntax). And most have not spent any comparable time being concerned about content (i.e., semantics). And worse, in my mind, most have not considered the issue of any underlying pragmatics. And, what in my mind is worse, they were unaware of the distinctions between pragmatics, semantics and syntax.

Perhaps I only, myself, got acutely aware of semiotics and “its” constituents (pragmatics, semantics and syntax), such as I see them, by reading Zemanek’s paper [300] in 1966, and by working in Zemanek’s laboratory in the early 1970s.

The claim being made here, that is, in [87–89], is that proper software design can benefit tremendously from the designer having a clear understanding of the concepts and rôles of pragmatics, semantics and syntax in domains as well as in software.

Hence the emphasis on clarifying the distinctions (pragmatics, semantics and syntax) in [87–89].

3.3 Coverage in Book

We take two views of the concept of ‘coverage’: A “syntactic” view and a “semantic” view. By the syntactic view we mean: Where have we covered the semiotic notions of pragmatics, semantics and syntax ? And by the semantic view we mean: Where have we applied those notions, made use of them, illustrated them ?

3.3.1 Places of Enunciation

We list some of the places where we have enunciated the semiotic ideas:

- Volume 1, Sect. 1.6.2, Pages 34–38.
- Volume 1, Sect. 9.5.5, Pages 178–180.
- Volume 2, Chap. 3: Denotations and Computations, Pages 55–85, in particular Sect. 3.1.2, Page 56.
- Volume 2, Part IV: linguistics, Pages 145–234:
 - ★ Chap. 6: Pragmatics,
 - ★ Chap. 7: Semantics,
 - ★ Chap. 8: Syntax and
 - ★ Chap. 9: Semiotics.
- Volume 2, Part VII: Interpreter and Compiler Definitions, Pages 573–703:
 - ★ Chap. 16: SAL: Simple Applicative Language,
 - ★ Chap. 17: SIL: Simple Imperative Language,
 - ★ Chap. 18: SMIL: Simple Modular Imperative Language and
 - ★ Chap. 18: SPIL: Simple Parallel Imperative Language.
- Volume 3, Sects. 6.4–6.5, Pages 163–169 (A Syntax and A Semantics of Formal Definitions).

The most notable place is, of course Volume 2, Part IV, that is, Chaps. 6–9.

3.3.2 Places of Deployment

The enumeration of Sect. 3.3.1 is, of course, a bit silly; emphasis on “meta-syntax” usually is. What matters is the semantics and the relationship between syntax and semantics. And, we think, semantics and the syntax/semantics relationship permeates all three volumes [87–89]: Vols. 1 and 3 [87, 89] less explicitly than Vol. 2.

An Aside:

The title of Vol. 2 [88]: Specification of Systems and Languages is, in this connection, a give-away: We consider, as also expressed in that volume’s Sect. 9.5 Systems and Languages, computing systems and programming languages to basically cover similar, i.e., “overlapping” concepts.

The systems (or domains) considered have configurations (contexts and states) and are acted upon. These actions are the semantic meanings of syntactic commands, like commands of a programming language.

But the commands may not be so elaborately structured and/or embedded in syntactic structures like those found in many programming languages.

To program well in any programming language one must understand its configurations (for example, contexts [environments] and states [activation stacks and storages]) To handle a system well the system user must likewise understand its configuration components well.

To end this argument we conclude that most abstractions and models (cf. title of Vol. 1) and that domains/requirements/software (cf. title of Vol. 3) are about systems. Semantic and syntactic issues are highlighted accordingly.

3.4 Critique

3.4.1 The Semiotics Claim

The claims (i) that there is a scientific discipline here called semiotics, (ii) that it covers an important area of linguistics and (iii) that it is spanned by pragmatics, semantics and syntax — and not some other finer or coarser “division” — are just that: conjectures (due to Charles Sanders Peirce [246–249] and further first studied by Charles Morris [239, 240] and Rudolf Carnap [132–134]).

We do not argue that claim. Rather we find it useful for the kind of languages (and systems) that we are interested in. Whether a number of facets of natural, spoken languages can be adequately understood from this semiotics perspective is of no concern to us.

3.4.2 The Adequacy Claim

When presenting a programming language, as done in for example Vol. 1’s Chaps. 19–21, and Vol. 2’s Chaps. 3 and 16–19, it is customary to rigidly present the syntax of programs first and then the semantics. When presenting a system, as done in many examples spread over all three volumes, it is customary to rigidly present the semantic types first, then the primitive (or basic) operations on values of these types before the syntax of commands are presented and given meaning. Main examples are those of Vol. 3’s Chaps. 26–27. But we have not rigidly structured every example that way. Take our hierarchical and compositional examples of railways of Sect. 2.3 of Vol. 2 (Pages 40–49).

So there is some wavering here; or, you might say, some not so systematic, or not fully adequate treatment of syntax and semantics wrt. every “darn little, or not so small” example.

When it comes to pragmatics we must say that the concept of pragmatics is, maybe, treated (including illustrated) reasonably well in Vol. 2’s Chap. 6 and in Vol. 3’s Sect. 2.3 (Informative Document Parts), but elsewhere in [87–89] it is not treated very much, if at all. Here the book could have been more illustrative.

3.5 Conclusion

From Sect. 1.4, Item 2 on page 5 we quote (slightly edited):

The contribution of [87–89] is that no (method cum software engineering) textbooks, and hence no claimed methods have heretofore covered semiotics issues and shown their relevance to software engineering, and that I have done so in a rather consistent and reasonably complete manner.

On The Computer and Computing Sciences

The distinction between computer and computing science is first highlighted in

- Vol. 1, Sect. 1.1, Pages 3–4.

We bring one set of these characterisations:

1. *Computer science* is the study and knowledge of what kind of “things” may (or can) exist “inside” computers, that is, *data* (i.e., *values* and their *types*) and *processes*, and hence their *functions*, *events* and *communication*.
2. *Computing science* is the study and knowledge of how to construct those “things”.

Computer science, in general, studies computability, whether in the conventional sense of Turing Machines or λ -Calculus, or in the sense of for example quantum computing, etcetera.

4.1 The Sciences of Computers and Computing

We briefly discuss the above two definitions, i.e., Items 1 and 2.

1. *Examples of Computer Science Studies*: Included in computer science are such classical studies (etc.) as Automata Theory (finite state and push-down automata and machines and their properties), Formal Language Theory (regular, context free, and context sensitive languages and their properties), Computational Models (λ -Calculus, Turing Machines, Markov Normal Algorithm, Post Systems, Petri Nets, etc., and their properties), Abstract Complexity Theory, Proof Theories, Foundations of Formal Specification Languages and Methods (Algebraic Semantics, Model-oriented Semantics), and Type Theory.

2. *Examples of Computing Science Studies*: Included in computing science are such classical studies (etc.) as Algorithmics (and Concrete Complexity), Functional (or Applicative), Imperative, Parallel, Logic and Object-oriented Programming, Refinement Calculi, Verification, Model-checking, and Formal Testing.

We consider [87–89] to cover mostly computing science topics.

3. A “*Continuum*”: First, the two definitions do not provide for clear-cut distinctions. Any one study or any one “piece of knowledge” related to computers or computing may be partly a computer science partly a computing science study or piece of knowledge.

One only has to examine the two discussion items just above to see examples of “overlap”: Proof Theory versus Verification, Abstract versus Concrete Complexity, etc.

4. *Concretisation*: The definitions are abstractly conceptual in the sense that one needs evidence of a particular study or piece of knowledge before one can decide “parthood”, i.e., one must weigh that study (etc.) up against the two definitions in order to decide whether it is a relevant study or piece of knowledge, and if so, “how much of either”.
5. *Computing Science and Programming Methodology*: In [87–89] we “equate” computing science with programming methodology.
6. *Interaction: Computer Science & Computing Science*: Computing science generally “builds on” computer science, i.e., the programming methodological issues of computing science can typically be explained by reference to results of computer science.
7. *Software Engineering*: Software engineering, such as we define it in [87–89], is the engineering of technological artifacts (software) based on scientific insight, and that insight is primarily computing “scientific”. To us the engineer “walks the bridge” between science and technological artifacts: constructs the latter based on scientific insight, studies the latter in order to possibly derive new or modify existing scientific insight.

4.2 Coverage in Book

The book [87–89] primarily exhibits, that is, presents a computing science, that is, a programming methodological study and “body of knowledge”.

Apart from introductory volume chapters and Chaps. 2–9 of Vol. 1, almost all chapters focus rather “single-mindedly” on programming methodology. Chaps. 13–14 of Vol. 2 form one exception: The semantics given there of the Message and Live Sequence Chart and the Statechart formalisms is studied from a computer science point of view.

Chapter 11 of Vol. 2, Automata and Machines, although conventionally regarded a topic of computer science is here treated from the point of computing science — so we claim.

We are not presenting a bouquet of distinct computing science issues, but rather a comprehensive software engineering methodology borne by a consolidated approach to programming methodology. This comprehensive view, of course, exhibits many aspects, but these aspects, we claim, have all been “carefully worked into a balanced whole”.

The book [87–89], however, only treats those aspects of programming methodology which have a firm foundation in computer science (but, as remarked just above, the book rarely reveals this foundation).

So, the “coverage” is not “a little bit here, some chapters or even a part there, etc.”. The coverage is, but I do not really like the term, ‘holistic’.

4.3 Critique

4.3.1 Scant Treatment of Computer Science

A main critique that can be raised about [87–89] is exactly this “single-mindedness”: the almost exclusive focus on computing science issues. It is a valid criticism insofar as some applications of method principles, techniques and tools are predicated, i.e., can only be justified, if certain computer science-formulated properties hold of the specifications being developed (constructed or analysed). Such method-issues typically arise in connection with verification, model-checking and formal testing. Since we omit serious treatment of verification, model-checking and formal testing we can partly justify our main omission of computer science material.

*Perhaps the most questionable part of [87–89] — questionable in the sense only on its lack of proper treatment of computer science foundations — is that of Chap. 16, Vol. 2: the step-wise development (“refinement”) of an interpreter and compiling algorithms for an applicative (i.e., a function) language (SAL). We refer to that chapters’ Sect. 16.4.6 Review (Page 588). It reminds the reader that we are not able to prove the correctness of the step of development from the denotational semantics definition of that chapters’ Sect. 16.3 to the first-order semantics definition of Sect. 16.4. The reason is basically that we cannot provide for a meaningful explanation, in RSL, of the recursive definition of $n\rho$: **let** $n\rho = \rho \uparrow [\nu \mapsto M(\lambda e)n\rho]$ **in** $M(b)n\rho$ **end**, given on Pages 578 (second line of formula [8]) and 582 (both Vol. 2). The sentence, on lines 11 to 10 from bottom, Page 579: “Such definitions of higher-order function types are, in general, not possible in RSL” alludes to the problem: namely that [87–89] does not cover the computer science aspect of when solutions exists to such such recursions.*

We justify this omission as follows: To cover all the necessary computer science foundations for all the computing science techniques and tools (read:

specification languages) covered in [87–89] would entail a major enlargement of the book. Such would not, we claim, further the main issue of the book in this context: namely that its techniques and tools indeed have a theoretical foundation.

4.3.2 Software Engineering Based on Computing Science

There are many more aspects to software engineering than the programming methodological (and a few other) issues primarily covered in [87–89]. The “few other” issues of the book are there in order to provide for a smooth line of unfolding the relevant methodological issues that are covered. The software engineering issues that some critics would claim are “lacking” or “missing” we would normally classify as either being of sociological or of psychological or of management character. As mentioned elsewhere, we would like to have been able to also write a volume on “Software Engineering: Management”. In such a volume we would then have to treat many of these (i) sociological, (ii) psychological and (iii) management issues — and then primarily as the two former sets of issues, (i)–(ii), arise in connection with the management, (iii), of software development and of software products. But our claim would remain: these sociological, psychological and management issues, if they are founded, are not founded on theories of computer science, but possibly on other, perhaps scientific theories.

4.3.3 Missing Computing Science Issues

There are issues of programming methodology, firmly based in computer science which are not in the book. Most significantly we are not treating the very important issues of (i) algorithmics (hence not complexity theory) and (ii) refinement calculi (hence not implementation relations either).

Here we justify the omission as follows:

1. *Completeness*: We cannot possibly cover everything, that is, we cover all those methodological issues that are — “to a first degree of approximation” — related to the development triptych: from domains via requirements to software design. Detailed techniques of functional, imperative, parallel, object-oriented and logic programming are not covered (although the first three, in the form of ‘specification programming’, are to some extent covered in Chaps. 19–21 of Vol. 1).
2. *Well-covered by Existing Books*: Those issues, like (i) algorithmics and (ii) refinement calculi, that are not treated, are either, as is the case with algorithmics reasonably well settled and reasonably covered by existing

textbooks¹, or, as is the case with refinement calculi are still not finally settled, but excellent text cum monograph books are readily available.

3. *More Serious Omissions:* As mentioned earlier and as dealt with more explicitly in Chap. 12, we are omitting serious treatment of (i) program verification, (ii) program model checking, and (iii) formal program testing. All of these disciplines have a firm basis in computer science. The reason for the first omission (i) is simple: For any software engineer well-trained in what this book covers, it is, today, in any interesting cases, a heroic effort to conduct a proof of a realistic property of a realistic specification. Therefore we do not cover program verification. Once principles and techniques of program (theorem) verification can be enunciated they should be taught to the “broad masses” of software engineers. Today any responsible software house will either contract special firms or hire PhDs to conduct such verifications.

And since we do not cover program verification seriously we also do not cover model checking and formal testing seriously.

4.4 A Diatribe

Usually the term diatribe is intended, according to Merriam–Webster [288] to mean: *a prolonged discourse, a bitter and abusive speech or writing, ironical or satirical criticism*. Well, we shall, perhaps not be bitter, abusive, ironical or satirical, but perhaps this section could lead to a prolonged discourse.

The issue of this section is whether the tools that are sometimes used by some computer scientists are appropriate/ This claim should be seen in the context of the tools that can be, or are already provided by computing science. The contrasting tools in question are those of ordinary mathematics and those of formal specification languages. Ordinary mathematics is typically deployed by computer scientists when presenting, or studying the foundations of, typically, computational models. Formal specification languages are typically deployed by computing scientists and software engineers when presenting or verifying (checking and formally testing) properties, typically, of domain or of requirements models or of software designs.

The claim being advanced here is that there are too many examples in which ordinary mathematical “models” are published where those models could be more appropriately expressed in some formal specification language.

A classical example is that of automata theory. In Chap. 11 of Vol. 2 we present finite (and infinite) state and push-down automata and machines using the RAISE specification language (RSL). We could equally well have

¹Still, we would like to see a comprehensive treatment of a comprehensive set of algorithms and their data structures from the point of view of refinement from domain problem description via requirements and stages of software design to executable code.

expressed these concepts using VDM-SL or Z (or some other model-oriented or even algebraic specification language). The objection might, seemingly very valid, be raised that in order to investigate properties of, as here, automata and machines, it is easier to create an own mathematical sub-language than using a formal specification language. We seriously doubt that. The formal specification languages often come with a proof system and its mechanisation, i.e., a theorem prover. Another, also seemingly very valid, objection may be raised, namely that these formal specification languages were not around when the notational foundations were laid for, as here, automata theory.² That objection, we claim, is no longer valid today, 2008, with the presence of an abundance of formal specification languages. The remarks above were tied, as an example, to automata theory. They could as well, as we shall see next, be tied to several other branches of computer science.

Perhaps a more telling example is that of Sect. 13.6 of Vol. 2: *CTP: Communicating Transaction Processes* versus [268]. The idea of combining the Petri Net and the Message (or even Live) Sequence Chart concepts as put forward by A. Roychoudhury and P.S. Thiagarajan [268] is, to us, a very nice idea. But their “formalisation”, in the traditional style of presenting mathematical semantics of Petri Nets or Message (or Live) Sequence Charts is fraught with several problems. Dr. Yang ShaoFa, when working out the semantics of CTP using RSL uncovered a, to us alarming, number of open questions and a, to us alarming, number of “specification” errors. Our claim is that by using RSL and according to the many principles and techniques of the method expounded in [87–89], one is lead to discover “dangling”, i.e., unresolved issues like those that were problematic in the [268] paper. Furthermore, the “formalisation” of [268] can not be the basis for a systematic development of an interpreter for CTP. Such an interpreter was developed by the PhD students of the authors of [268]. But no guarantee could or can be made as to the correctness of that [or those] interpreter[s]. There simply can not be developed a set of formal techniques for “deriving” a software design from a classical mathematical notation³, and since the model was anyway wrong no guarantee can be established. An issue, in our “complaint” is, of course, when the conventional mathematical model has been corrected and is consistent and complete, whether it would then be more elegant or concise, or less implementation-biased than an Event B, an RSL, a VDM-SL, or a Z model. Let us, for the sake of argument, claim that it would not !

• • •

²Dana Scott, Some Definitional Suggestions for Automata Theory Journal of Computer and System Sciences, Vol. 1 (1967), pp. 187-212. Reviewed by Robert F. Barnes in The Journal of Symbolic Logic, Vol. 40, No. 4 (Dec., 1975), pp. 615-616.

³If preparing such a notation for such “derivation” (refinement), i.e., implementation relations, that notation becomes a formal specification language, then all is OK!

An example of “almost” the “opposite”: of a pseudo-formal model that is so wrong that “it gives formal specification a bad name” is that of [172]. Since we, for better reasons, bring an appendix, Appendix H, on the concept of ‘license languages’, and since our “almost bad” example, [172], is of a paper in the area of ‘license languages’, we bring, as Sect. H.7 (Pages 367–380) an excerpt of [172]. In [172], and thus in that excerpt, you will observe how sloppily an attempt of formalising a simple license language can be done. The problematic areas of the formal model in Sect. H.7 are shown as framed texts. The ‘morale’ is this: when one wishes to formalise some relevant language or system then one has to be utterly precise — even to the level of possibly boring the reader !

4.5 Conclusion

From Sect. 1.4, Item 3 on page 6 we quote (slightly edited):

The contribution here is that no method cum software engineering textbooks, and hence no claimed methods have heretofore spelled out the distinction, such as we make it, between computer and computing science, nor have they emphasised the need for the practicing software engineer to possess a firm, but not necessarily deep base in computer science and a reasonably firmer base in computing science — such as we suggest it for the former (computer science) and provide it for the latter (computing science) in [87–89].

We thus claim that the book covers all the programming methodological cum computing science issues that are immediately relevant to the overall exposé of the triptych of software engineering (see Chap. 10). Our major claim, i.e., thesis or conjecture here, is that software engineering can be taught, and — for major kinds of software development — practiced without resorting to computer science foundations.

Mathematics

Few textbooks on software engineering — a delightful exception is Woodcock and Loomes’s *Software Engineering Mathematics* [298]¹ — cover mathematics.

What mathematics must a software engineer be well-versed in — and why? We shall answer the last question first.

5.1 Why Mathematics ?

The artefacts that software engineers construct, whether they be domain descriptions, requirements prescriptions or software designs have properties, as do all human-made artefacts, and these specifications satisfy laws of mathematics — where most other human-made artefacts satisfy laws of physics (including chemistry).

That is the main reason for bringing mathematics into software engineering.

Expressed more radically: software programs, in a model-oriented sense, denote mathematical objects.

Now, expressing domain descriptions, requirements prescriptions and software designs in one or another formal, i.e., mathematics-based specification language — be it textual or graphical or both — has some beneficial side-effects. We cover those next.

1. *Objectivity*: When informally describing domains or informally prescribing requirements or informally designing software (say a la UML), we are modelling domains, requirements, respectively software in the realm of linguistics. The various metaphors [215] that spring to the mind [214] of the modellers and to the readers of these informal model specifications are

¹— but then [298] is not about software engineering but about some of its mathematics

not made explicit. The modeller has no control over how the reader understands these specifications. The modeller and the reader interpretations are subjective.

When describing domains or prescribing requirements or designing software formally, i.e., using some mathematics-based specification language, then the model or models intended by the modeller is, respectively are those that are also “read” by the readers. The modeller and the reader interpretations are objective. Also for the accidental, but unfortunately recurrent situation that the mathematical model denotes **chaos**, i.e., is unintentionally wrong.

2. ***Precision:*** A correctly expressed mathematical model is precise, even when it expresses non-determinacy. That is, if looseness (vagueness) is desired, then it can be precisely expressed: multiple interpretations are then intended — all of them!
3. ***Engineering Standard:*** A mathematical model, suitably explained and annotated in a precise, but informal, natural language, e.g., English, can serve as an engineering standard.²
4. ***Implementability:*** A software design specification, i.e., a mathematical model, can serve as a unique basis for further software development.
5. ***Provability:*** And such a development, from specification to code, cf. Item 4, can possibly be verified. Without a precise mathematical model no precise, objective verification can be done. In fact, as is the intention of Vol. 3’s coverage of the TripTych dogma, the formalisation of the “path” from domain models via requirements models to software design specifications and, finally, code may similarly be subject to formal verification of properties such as some notion of ‘correctness’.
6. ***Mathematical Models:*** In describing the physical world — mechanics, including celestial mechanics, electricity, electromagnetism, electrodynamics, quantum mechanics, aerodynamics, fluid- and hydrodynamics, etcetera — scientists have deployed mathematics. Mathematics have, so far, been the only available notation in which to objectively and unambiguously express models of physics.

Models, as discussed in Vol. 3’s Chap. 4: ‘Models and Modelling’ (Pages 105–117), are either analogic, analytic, or iconic models; models are either descriptive or prescriptive, and models are either extensional or intensional, that is, models attributes are combinations of these. Models are constructed in order to understand, in order to get inspiration and

²So was the case already in the early 1980s when the C.C.I.T.T. (now ITU, Intl. Telecomms. Union) put forward the CHILL (Communications High-level Language) Recommendation Z.200 ISO/IEC 9496. See the CHILL “Blue Book”, Geneva 1989 ISBN 92-61-03801-8. The informal, officially binding language description was officially supported by a formal description [4, 174, 176] (in VDM [111, 112, 157, 158]) under my leadership together with my colleague Dr. Hans Bruun and by my former students, notably Peter L. Haff.

to inspire, in order to present, educate and train, in order to assert and predict, or in order to be “implemented”.

Only mathematics can lend these qualities to models in a believable manner.

5.2 Mathematical Notations

Perhaps we are using the terms ‘mathematics’ and ‘mathematical’ in a misleading way.

5.2.1 Writing Mathematics

Basically we are primarily using mathematical notation rather than extending mathematics.

That is, the use of formal techniques is not meant as other than using mathematical notations and deriving the benefits from doing so — as enumerated in the 6 items of Sect. 5.1.

That is, we are not expecting the software engineer to become or “mimic” mathematicians. The software engineer is not expected to extend existing mathematical theories with new lemmas, propositions and theorems let alone create new theories such as mathematicians are doing it.

5.2.2 Mathematical Abstractions

But to “transcribe” informal, precise descriptions (prescriptions and specifications) into formal form, using mathematical abstractions imply and require mathematical thinking, that is, thinking in terms of concepts and of abstracting these.

We reckon this to be the most important aspect of description, prescription and specification.

It is a common experience that knowing and practicing a modicum of discrete mathematics in expressing abstractions lead to elegant and pleasing, sometimes even beautiful [154] and inspiring models.

5.3 Which Mathematics ?

5.3.1 Two Kinds of Mathematics

Two kinds of mathematics apply to software engineering. The mathematics implied directly by the notations that are being used, and the mathematics

that was used to explain the semantics of these notations as they were brought together in a formal specification language. We briefly remark on the latter first.

Mathematical Underpinnings

Formal specification languages, in order to satisfy the criterion of being formal, must have a precise semantics and a proof system. The semantics is usually expressed in some mathematical framework. By the mathematical underpinning of a formal specification language we mean that mathematical framework. We shall not here go into the mathematics of the semantics of the formal specification language which is mostly used in [87–89] — for the simple, engineering reason that the software engineer should not be concerned with this mathematical underpinning, i.e., “should not be bothered”. (In Vol. 2, Chaps. 13 and 14 we do, however, present “the” semantics of the Message and Live Sequence Chart and the Statechart diagrammatic formalisms. The reason for that is to be able to relate these formalisms to the RAISE specification language RSL.)

Software Engineering Mathematics

By software engineering mathematics we mean the kinds of mathematics that are expressed by the formal languages used for formal domain descriptions, formal requirements prescriptions and formal software design specifications.

They includes: numbers, sets, Cartesians, relations (maps and functions), λ -Calculus, algebras and mathematical logic.

Usually we characterise these kinds of mathematics as “belonging” to ‘discrete mathematics’.

In contrast to discrete mathematics we have the classical calculus, statistics, probability theory, etcetera. With regret these do not figure in [87–89].

5.3.2 Specification Mathematics — which is included

We comment briefly on the some — seemingly arbitrarily chosen — properties of discrete mathematical entities with respect to their use in formal specifications.

Numbers: The ability to deploy numbers — in the form of natural numbers, integers and reals — have proven quite useful in (model-oriented) abstract specifications. They are, however, not used much. But natural numbers, for example, appear as indices of sequences of entities. Within computers numbers are typically restricted to lie within a range $(-2^{n-1}..+2^{n-1})$, where n , typically is a so-called ‘word length’ of the computer). In (model-oriented) abstract specifications there is no need to be concerned with the machine representation of the values being expressed (even) in (model-oriented) specifications.

Sets: Sets form the “workhorse” of (model-oriented) abstract specifications. And, as for reals, there are no concerns with respect to machine representations: thus also indefinitely large (with respect to number of set members) and even infinite sets are quite common in (model-oriented) abstract specifications. One may consider the actual world to be finite or definite, but abstract concepts formed on the basis of physical, actual phenomena, can often be modelled in terms of indefinite or infinite sets. (A “classical” example is the infinite set of routes in a finite transportation network where routes may be circular, i.e., where parts of a route may be repeated indefinitely.) Domain descriptions may feature non-computable sets. The full complement of “standard” set operations apply (\in , $=$, \neq , \subset , \subseteq , \cup , \cap , **card**, etc.).

Cartesians: Cartesians are fixed groupings of usually two or more entities. In programming languages “their equivalent” is called ‘records’ or ‘structures’. Only the $=$ and \neq operations apply.

Maps and Functions: By functions we mean functions in general, partial or total, whether defined or just postulated, whether computable or not. By maps we mean finite or infinite definition set³ enumerations of definition set/range pairs, whether computable or not. Functions and maps are, perhaps even more so that sets, the “workhorse” of model-oriented specifications.

A Critique: Relations versus Maps and Functions: One may reasonably object to my choice of “promoting” maps and functions and not the more general notion of relations, i.e., from the very beginning present maps and functions as relations. (Sect. 6.7 (Vol. 1) does, however, “retell the story” of maps and functions as relations.) So why not “from the beginning” ? My answer is that I want the specifier to think in terms more of maps and functions than in terms of relations when specifying phenomena and concepts. If a phenomenon or concept is more elegantly abstracted and modelled as a relation, then I want the specifier to think in terms of sets of (same type) Cartesians. Again we should recall the distinction between computer and computing science. In the former relations, as a foundational concept, are preferable to maps and functions.

λ -Calculus: The λ -Calculus is our basic means to understand a number of properties of defined functions, their abstraction, their application (i.e., their “being applied” to arguments), and their fix points. Landin was the first to really expound on the usefulness of the λ -Calculus in understanding programming language concepts [219].

³By ‘definition set’ we mean the set of values for which the function is defined. In other texts a ‘definition set’ are referred to as a ‘domain’. To avoid confusion with the TripTych notion of a domain as a universe of discourse, we choose the term ‘definition set’, but, for reasons of tradition, keep the name of the operator, **dom** (for ‘domain’), that applies to enumerable maps and yield their definition sets.

A Critique: Rather The λ -Notation than the λ -Calculus: Our presentation of the concept of λ -Calculus is not a computer science theoretical, or foundational one, but is a practical one: the λ -Calculus as a notation, to be used and understood in a practical, engineering way. Thus, for example, the fix point operator can be used in evaluating recursively defined functions — but it may not (always) denote the, or a smallest fix point operator !

Algebras: Algebras — more or less implicitly — play an important rôle in (abstract) specifications. We think of algebras in the simplest possible sense of universal algebras: possibly infinite sets of entities and usually finite sets of operations, of various arities, over these entities. When describing domains, prescribing requirements or specifying software designs it is useful to think in terms of defining, possibly heterogeneous algebras: defining sets of classes of entities in terms of their types and defining operations on possibly Cartesian compounds of values of these types in terms of function definitions, or just function signatures. Thinking algebraically helps structure descriptions, prescriptions and specifications. Several specification languages, including RSL, provide algebraic language constructs such as RSL's schemes and classes to focus the specifiers attention.

A Critique: Why not Classes and Schemes ? In view of the last sentence above one may therefore question why I have not presented many, or most of my examples as class or scheme definitions. Why not ? My answer is that using any particular formal specification language's modularisation constructs subject me to a number of constraints that I am not, at this time, willing to accept. One thing is the entities and functions (i.e., the algebra) of the phenomenon or concept that I wish to specify; another thing is the “casting” of that algebra into some specification language's modularisation constructs. My position is the following: none of the specification language modularisation constructs that I am familiar with are “optimal”: they constrain your presentation in one way or another; they more or less force you to settle on a specific set of class and/or scheme definitions “from the very beginning” — long time before the development of the initial formalisations are ready to be “cast” into classes and schemes (as in RSL — or whatever the modularisation constructs are called). My position is further: Once a reasonably complete part of a specification (whether description, prescription or other) has been achieved, then it is time to analyse that specification and modularise it.

(The TripTych “story” on modularisation is given in Vol. 2, Chap. 10; and the RSL “story” on modularisation is given in that chapter's Sect. 10.2.)

Mathematical Logic: Mathematical logic, in the sense of a notation with semantics and proof system, is a, if not the major “work horse” of abstraction.

Volume 1's Chap. 9 presents that view. Sect. 9.2 of Vol. 1 focuses in particular on the difference between the classical view of a mathematical logician's view of mathematical logic and the view with which we wish to guide the software engineer. In short the difference, "in the vernacular", is: The mathematical logician, to put the matter perhaps in the extreme, studies one logic or a variety of logics with respect to for example decidability properties. The software engineer selects a logic appropriate to the specification and verification task at hand, uses its notation and, hopefully, sooner or later, conducts proofs of properties of the specification using the proof system of the logic.

A Critique: Absence of Material on Verification ! It has been said before, and it will be said later: we regret to not bring substantial material of how to formulate provable (propositions, lemmas and) theorems, and hence we do not bring material of how to conduct proofs (i.e., proof tactics and proof strategies), let alone even examples of proofs ! And we repeat the reason for this. The state-of-the-art of verification is still, unfortunately and despite more than 35 years of research, to me, in the experimental research stage. It is still too much of a *heroic effort* for anyone to carry through any interesting verification effort.

5.3.3 Specification Mathematics — which is not included

Calculus: Formal specification languages all lack means for expressing continuity. This is a serious hindrance to bringing together the control-theoretic domain modelling of a number of physical systems and the "refinement" of these into requirements for safety critical, embedded, real-time systems. The inability of taking the derivative, or integrating over intervals, of continuous, for example partial differential equations severely limits current formal specification languages.

The main stumbling block for integrating continuity in the sense of classical calculus seems to be that we cannot establish a proper proof system for the integrated result. Currently much research is being devoted to integrating "formal methods" [6, 126, 129, 171, 265], but mostly where the integrated sublanguages are themselves formal; and the languages of (partial) differential equations are not formal in the sense of having a proof system themselves.

In [303] (*An Extended Duration Calculus for Hybrid Real-time Systems*) the Duration Calculus [301] is extended with a first-order mathematical system, for example by first-order differential expressions. We have not observed much further research into, let alone applications of such languages.

Etcetera: There are many other disciplines of mathematics which have been left out of [87–89], to wit: probability theory, statistics, combinatorics, graph theory, etcetera. Many of these disciplines are taught to most software engineers in courses separate from software engineering. And their techniques can be used by software engineers: "on the side", for example, when determining

parameters for software implementations; [301] provides one such example. Software engineering textbooks, for example in database design or in the design of embedded, safety critical real-time systems, etc., cover such material better than is necessary in our book.

5.4 Coverage in Book

The use of the discrete mathematics disciplines mentioned in Sect. 5.3.3 permeates all of [87–89].

5.5 Critique

Section 5.3.3 brings, in indented environments, a number of critiques of the specific use of mathematics in [87–89]. In addition this section will bring a criticism of non-uses of certain forms of ‘mathematics’.

5.5.1 Foundations

From the earliest conception of [87–89] I decided not to present or use any of the mathematics that is at the foundations of the semantics of the formal specification language (RSL) used in [87–89]. In fact, the formal specification language, RSL, used as the foremost language of [87–89], is never “formally”, i.e., systematically introduced as such a formal language, but is introduced in conjunction with illustrating suitable abstractions, cf. Chaps. 2, 9, 11, 13–16 of Vol. 1, and then the emphasis is not on the mathematical foundation of numbers, logic, functions, sets, Cartesians, lists (sequences), and maps and functions (again), respectively.

[87–89] is no worse off since “all” (known) other textbooks in software engineering also do not do that.

But there are cases (I do not list them here, but I am sure “critics” can find them), where [87–89] may “thread too lightly” past foundational issues that, perhaps should have been referred to in order to secure the proper use of certain specification language constructs. A case in point is “quote”-indented in Sect. 4.3.1 on page 35.

5.5.2 Continuity

It was mentioned above that current formal specification languages lack the ability to properly handle continuity. That this presents a problem one can see, in [87–89], from the treatment, for example, of (continuous) time and (continuous) space. We refer, for example, to Vol. 2’s Chap. 5: Time, Space and Space/Time. Although we give an example of an axiomatic foundation for

time the resulting time concept is not further “exploited”. And no axiomatic foundation is, for good reason, attempted for space — as that would “lead us far afield”. Elsewhere, in [87–89], dense sets (like time and space, and usually these) are mentioned without saying much more than that, and certainly this “denseness” is not exploited further. Well, there are some attempts to quantify over these dense sets in order to express properties of functions from elements of dense sets (i.e., time) to some other entities. And these attempts are reasonable, i.e., are OK, but if a firmer foundation had been laid, perhaps more adequate predicates could evolve. The problem is, however, that those parts of [87–89] would require quite substantial extensions wrt. mathematical foundations — and most readers would have been unnecessarily “lost”.

In Vol. 2’s Chap. 15, Quantitative Models of Time, a calculus, the Duration Calculus, is introduced which handle continuous time in a (most) satisfactory way. Perhaps we should have use the Duration Calculus to express the predicates⁴ hinted at in the previous paragraph.

5.5.3 Models of “Other” Mathematics

But formal specification languages can be deployed — instead of conventional mathematical notation — to formalise problems of statistics, probability, combinatorics, graph theory, time series, etc.

In Sect. 4.4 on page 37 we “lamented” the practice, usually by our computer scientist colleagues to use more or less ad hoc mathematical notation to give semantics to computational systems which could, we argued, be better specified by using a formal specification language.

The same objection can now be raised for the case of our operations research colleagues who also use smatterings of mathematical notations — notably from set theory and graph theory — in their modelling endeavours. In fact, as long as one does not need continuity formal specification languages might be a better modelling tool as they are precise and have proof systems.

Appendix J shows the very rough sketch beginnings of an attempt to model Bayesian Networks [209]. We bring that appendix only for this very simple reason, namely to support the claim made in the first paragraph of this section.

⁴Example of predicates over dense time are:

- Let traffic be a continuous function from a dense time (definition) set to locations (again a dense set) of vehicles.
 - ★ for any two times in the definition set of the traffic, if a given vehicle is in the traffic at both of these two times then it is in the traffic at all times between these two times,
 - ★ and if the two times are infinitesimally close then the locations of the vehicle are likewise infinitesimally close.

5.5.4 Possible “Waverings”

A careful reader of all volumes [87–89] may end up thinking that our choice of mathematics may not be that consistent across the three volumes [87–89]. This author at least has that nagging “feeling” without, however, being able to point to specific cases. That is, although “we preach”, or at least deploy mathematical abstractions and hence notations, we are not claiming these deployment to be consistent across [87–89]. In some places we abstract in the property-oriented style of abstract types, i.e., sorts with observer functions and axioms — such as first illustrated in Vol. 1’s Sect. 8.5 (Specification Algebras), part of Sect. 9.6.5 (Examples 9.23–9.25), and Sect. 12.2. In other places we abstract in the model-oriented style of concrete types and explicit function definitions (Vol. 1, Sect. 11.2). It may seem to many readers that we most deploy the latter style of specification and that there is not, except in Vol. 1, Chap. 12, given enough methodological guide lines for which mathematical approach to choose. And even if that is so, it may be that the author (me !) does not strictly follow “own” guidelines ! Be that as it may, perhaps some more explicitly motivated usage might (further) improve that aspect of [87–89].

5.6 Conclusion

From Sect. 1.4, Item 4 on page 6 we quote (slightly edited):

The (minor) contribution here is that only few (method) textbooks, and hence few claimed methods have heretofore bothered to make sure that an introduction to basic mathematics is provided and used — such as we do.

Simple¹ Entities, Operations², Events and Behaviours

The notions of simple entities, functions, events and behaviour are covered

- concretely in Vols. 1–2,
- more conceptually in Vol. 3, Chap. 5., and
- with systematic exemplification in the rest of Vol. 3.

The aggregation of simple entities, functions, events and behaviour as a modelling paradigm amounts to an ontology of modelling. There are many ontological “theories” of “data modelling”, “data analysis”, etc.² The perhaps “grandest” such theory is that of John P. Sowa’s [281] *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Ours is a rather modest, simple one. And it works !

6.1 Introduction

As the title of this chapter indicates our ‘ontology’ of modelling hinges on the four notions of simple entities, functions, events and behaviours — together I shall refer to this ensemble of four concepts as entities. That theory (of phenomena and concepts) is justified, crudely, along two lines, and as follows: Firstly we conjecture that these four meta (or ontological) concepts suffice in modelling domains and requirements, and in achieving elegant models. Secondly: (i) simple entities (in domains and requirements) eventually,

¹In [89] we only used the term ‘entities’, but, as we shall see, now prefer to use the term ‘entity’ to cover the concept of ‘simple entities, operations, events’ and ‘behaviours’.

²We use the terms ‘functions’ and ‘operations’ synonymously, but today prefer to use the term ‘operation’ in connection with domains and ‘function’ elsewhere.

²The term ‘data’ should be understood in a wider sense than just computer data, where computer data are formalised representations of entities (incl. information). To model ‘data’ isolated from the operations, the functions on data makes little sense.

and typically end up as data “inside” computers, (ii) operations eventually, and typically end up as procedures (code) “inside” computers, (iii) events reflect interaction with the environment of computers (or interactions between computing processes), and (iv) behaviours (strands of actions and events, with actions being invocations of functions) reflect computing (and data communication network) processes.

Let us mention the concept of an algebra as a possibly infinite set, E , of entities and a usually finite set of operations, O , such that these operations, $o:O$, of arity n , apply to entities e_1, e_2, \dots, e_n , that is, $o(e_1, e_2, \dots, e_n)$, yielding an entity in the set E . That is, simple entities and operations, our first two specification concepts thus fit with the notion of specifying algebras. We shall in general wish to have all entities, that is, also operations, events and behaviours be the subject of, that is, argument “to” operations. More about this later.

• • •

A contribution of [87–89] is its promulgation of the quadruple concepts: simple entities, functions, events and behaviours as technical “cornerstones” in domain descriptions, requirements prescriptions and software design specifications.

The understanding of the quadruple concepts of simple entities, operations, events and behaviours have developed noticeably since the writing of [87–89]. In this chapter we shall outline the “improvements”. Hence this chapter will summarise the “new” treatment of these concepts. And thus this chapter is slightly different in style and a bit longer than Chaps. 2–5 and 7–11 all of which also review a more-or-less distinct facet of [87–89].

6.1.1 Entities

An *entity* is something that has a distinct, separate existence, though it need not be a material existence, to which we apply functions. With simple entities we associate properties modelled as types and values: a simple entity is of some type and has, at the time it is observed, a value (of that type). Simple entities can be considered either atomic or composite. It is the observer (that is, the specifier) who decides whether to consider a simple entity to be atomic or composite. Atomic entities cannot meaningfully be decomposed into sub-entities, but atomic entities may be analysed into (Cartesian) “compounds” of properties, that is, attributes. Attributes have name, type and value. Composite entities can be meaningfully decomposed into sub-entities, which are entities. The composition of sub-entities into a composite entity “reveals” the, or a mereology of the composite entity: that is, how it is “put together”.

6.1.2 Operations

By an *operation* we shall understand something which when *applied* to some entities, called the *arguments* of the operation, *yields* an entity, called the *result* of the operation application (also referred to as the operation invocation). Operations have signatures, that is, can be grossly described by the oftentimes compounded (that is, Cartesian) type of its arguments and the possibly likewise compounded type of its results. Operations may be total over their argument types, or may be just partial. We shall consider some acceptable operations as “never terminating” — and shall call them processes. We shall, for the sake of consistency, consider all operation invocations as processes (terminating or non-terminating), and shall hence consider all operation definitions as also designating process definitions.

We shall also use the term **function** to mean the same as the term operation.

6.1.3 States

By a *state* we shall loosely understand a collection of one or more simple entities whose value may change.

The concept of state is an elusive one.

Thus, since any collection of simple entities is eligible for “state-hood”, one may ask for guidance as to what “state-selection” criteria may be advised. On one hand there may be many simple entities “in play” in any one specification. So selecting a subset leaves another subset as non-state simple entities. What are they then ?

Roughly speaking we can group simple entities with respect to their rôle as arguments to or results of operations into three classes. Those simple entities which the specifier decides to endow with state-hood; those which are common (or regular, i.e., repeated) arguments to, but not results of operation invocations — often referred to as context or environment entities; and those, occasional, simple entities which, in a loose sense play a rôle of arguments to operation or results of state non-changing operations. In Vol. 1, Sect. 20.6 and in Vol. 2, Chap. 4 we discuss, at length, the notion of configurations, namely that of combinations of states and contexts.

Based on those references we can summarise states as those entities which changes value “frequently” (whereas contexts change value “less frequently”) !

6.1.4 Actions

By an *action* we shall loosely understand something which changes a state. That “something” could be an operation invocation, that is, the application of an operation to some arguments. In applicative specifications the state is then part of or the whole of an argument and part, or the whole of the result. In imperative specifications the state is conventionally represented by a finite set of assignable (that is, update-able) variables.

6.1.5 Events

- An *event* can be characterised by
 - ★ a predicate, p and
 - ★ a pair of (“before”) and (“after”) of pairs of
 - states and
 - times:
 - $p((t_b, \sigma_b), (t_a, \sigma_a))$.
 - ★ Usually the time interval $t_a - t_b$
 - ★ is of the order $t_a \simeq \text{next}(t_b)$.

Sometimes the event times coincide, $t_b = t_a$, in which case we say that the event is instantaneous. The states may then be equal $\sigma_b = \sigma_a$ or distinct !

We call such predicates as p for event predicates.

Informally, by an *event* we shall loosely understand the occurrence of “something” that may either trigger an action, or is triggered by an action, or alter the course of a behaviour, or a combination of these.

6.1.6 Behaviours and Processes

By a *behaviour* we shall informally understand a strand of (sets of) actions and events. In the context of domain descriptions we shall speak of behaviours whereas, in the context of requirements prescriptions and software designs we shall use the term processes.

By a *behaviour* we, more formally, understand a sequence, q of actions and/or events $q_1, q_2, \dots, q_i, q_{i+1}, \dots, q_n$ such that the state resulting from one such action, q_i , or in which some event, q_i , occurs, becomes the state in which the next action or event, q_{i+1} , if it is an action, is effected, or, if it is an event, is the event state.

6.1.7 An Improved “Story”

Since the release of [89] we think that the concept of entities can be given a better presentation than in [89]. We refer to Appendix B, starting Page 155, for that better “story”!

In Appendix B, [103], we

- give a more precise definition of how we might model events and behaviours, and
- speculate on ‘compositionality’ and mereology of composite entities — whether simple, operations, events or behaviours.
- That is, we “raise” operations, events and behaviours to “first class” entities — something that was left out of [89] since it would not be appropriate, in [87–89] to bring research ideas that had yet to mature.

The next three sections, Sects. 6.1.8–6.1.10 (Pages 55–56) summarises our current position on entities.

6.1.8 Overview of Entities

The above characterisations, indeed the entire conglomerate of simple entity, operation, event and behaviour concepts, presume a model-oriented “way of thinking”. In a model-oriented specification there will then be specific language constructs covering each of these four notions: (abstract and concrete) types, (typed) variables and formal (typed) parameters to cover the entity concept; function definitions (including function signature definition), function application and function abstraction to cover the operation concept; “input/output alphabet” (composed into suitable [synchronisation & communication] expressions, as in CSP) to cover an event concept, and sequential (“;”), parallel (||), process definition (usually simple function definitions may suffice), and deterministic and (internal (\square) and external (\square)) non-deterministic process compositions (as in CSP) to cover the behaviour concept. Model-oriented specification languages then offer a considerable number of primitive, i.e., so-called “built-in” functions.

In an algebraic specification language there are basically only the following language constructs: entity sorts (abstract types) and variables, observer and generator function signatures, function application and axioms over function applications and “encoded”³ entities. Algebraic specifications often “spend some considerable” textual space in order to “construct” the assemblance (i.e., appearance) of concrete types, “built-in” operations, events and behaviours. This part of an algebraic specification can be referred to as an ontology of the specified language or system.

The names “data analysis” or “data modelling” are basically erroneous names as the analysis, the modelling, really cannot proceed without any reference to operations (events and behaviours). Algebras, we conclude, are at the basis for both algebraic and model-oriented specifications.

6.1.9 Specific

Any attempt, as that of [87–89] and [103], and as that above, to properly delineate the specification ontological notions of simple entities, operations, events and behaviours soon “runs into” problems of existential, that is, philosophical nature. One need only “Google” these terms and, for example, select the ‘Wikipedia’ entries: <http://en.wikipedia.org/wiki/Entity>, [/Operation](http://en.wikipedia.org/wiki/Operation), [/Event_%28philosophy%29](http://en.wikipedia.org/wiki/Event_%28philosophy%29), and [/Behaviour](http://en.wikipedia.org/wiki/Behaviour), to observe that their characterisations are inconclusive. In Appendix B Sect. B.3 (starting on Page 160), bring a — perhaps — more satisfying essay on this matter — with Sect. B.10.1 on page 192 briefly referring to the philosophical nature of this topic.

The next subsection addresses this issue.

³By ‘encoded’ entities we mean that non-simple entities have to be “coded” as if they were simple, un-interpreted entities — as are the models of events and behaviours in Appendix B, that is, [103].

6.1.10 Entities and Properties as an Ontological Base

We explain operations, events and behaviours on the basis of simple entities and their properties. The following explanation is based on [160].

We operate with two concepts: entities and properties. Entities are perceived of through their properties, but properties are not to be considered to be entities. Amongst properties are those of a simple entity being atomic and having attributes of types and values, or being composite in which case we consider the sub-entities and their mereology, i.e., how these sub-entities are put together, i.e., contribute to the ‘whole’, and the attributes of the composite entity to all be properties of entities. Amongst properties of entities are also whether they additionally can be considered operations, event or behaviours. (This view is not emphasised in [87–89].)

Therefore, as treated in Chap. 10 of Vol. 3, Domain Attributes, and advancing some of the material that more properly should go into the very next section (Sect. 6.2), the properties of entities, in addition to those which can be measured⁴, include (i) being continuous, discrete or chaotic, Vol. 3, Sect. 10.2, (ii) those of being static or dynamic (Vol. 3, Sect. 10.3) with the further dynamic entity properties of being inert, being active with properties autonomous active, biddable active, and programmable active, or being reactive. (iii) Further properties dealt with are tangible and intangible (in Vol. 3, Sect. 10.4), and/or “possessing” dimension (Vol. 3, Sect. 10.5).

On this background an entity which has functional property is an operation, one which satisfies [the] event criteria is an event, and one which satisfies [the] behavioral criteria is a behaviour. For the sake of generality these conceptual entities may possess either atomic or composite properties — as the domain engineer decides to abstract them. Thus we allow full generality.

In [87–89] we have taken the view — and we continue to take this view — that entities are characterised uniquely by their properties and that these properties must be measurable. Domains, “in the final analysis”, such as we wish to describe them, “boils” down to observable phenomena and the structures of concepts that can be (recursively) constructed from these. We are thus excluded from describing phenomena or concepts that cannot be objectively measured — such as feelings, moods, sentiments, “what is art” (cf. Vol. 3, Sect. 6.2), etc. Well, we can write things down about these, but we cannot formalise them.

6.2 Coverage in Book

We briefly refer to major places in [87–89] where we cover simple entities, operations, states, actions, events and behaviours. By operations we do not mean

⁴Measurable properties relate to the five human senses and the physically (incl., chemically) measurable quantities.

the “built-in” functions on entities but the operations that can be postulated, given signature and possibly defined over entities.

6.2.1 Simple Entities

The following data types can occur as simple entities, i.e., as arguments to and results of “built-in” or defined functions: atomic types: numbers, Vol. 1, Sect. 10.2; enumerated tokens, Vol. 1, Sect. 10.3; characters and texts [character strings], Vol. 1, Sect. 10.4; and identifiers and general tokens, Vol. 1, Sect. 10.5; functions (Vol. 1, Chaps. 11 and 17), sets (Vol. 1, Chap. 13), Cartesians (Vol. 1, Chap. 14), sequences (lists, Vol. 1, Chap. 15), maps (Vol. 1, Chap. 16) and functions (Vol. 1, Chaps. 6, 11 and 17). Means of defining subclasses and compositions of these are covered in Vol. 1, Chap. 18 (Types).

A more general, but, as noted above, not a fully satisfactory treatment of entities is given in Vol. 3, Sect. 5.3. The next subsections summarise Vol. 3, Sect. 5.3 (Pages 125–138).

A final treatment on atomic and composite entities, and on operations, events and behaviors being first class entities is given in Appendix B, Sects. B.3–B.7.

Atomic and Composite Entities

The story told in Vol. 3, Sect. 5.3 is a bit too complicated, I think in particular on subsection 5.3.2 ‘Composite Entities’. When writing the paper reproduced in Appendix A, I believe I got a shorter, more concise exposition of the concept of entities, atomic and composite, Sect. A.3 starting Page 126.

Atomic Entities

Atomic entities have no sub-entities, and hence no mereology, but atomic entities may be of, that is, abstractly modelled as a composite type, hence with composite values. See Example A.1 on page 127.

Composite Entities

Composite entities have proper sub-entities, and hence a mereology. See Example A.2 on page 127.

Attributes, Types and Values

We use the term ‘attribute’ to cover both the concept ‘type’ and the associated concept ‘value’. We speak of composite entities and of compound types. They are not the same. Entities are the (atomic or composite) things that we model. Types are part of our way of modelling these entities.

An atomic entity may be modelled in terms of a single atomic type, or in terms of a compound of (typically, but not necessarily) atomic types. A composite entity is always modelled in terms of a compound type (and more).

Entities, Sub-entities and Mereology

The “new” thing in our way of treating composite entities is that of separating into three concerns the modelling of composite entities: they have sub-entities; the composition of the sub-entities is according to some mereology; and the composite entity itself — “independent” of its sub-entities and mereology — has attributes. Cf. Example A.2 on page 127.

On “Data” Modelling

The above sub-sub-sections and paragraphs have been expressed in a way that abstracted from the operations (functions) that are to be performed on the entities. Of course, choosing, for example, set, Cartesian, list or map types, for the “compositional” mereology is very strongly influenced by considerations of these operations.

The term ‘data modelling’, in our opinion, may suggest a focus on types and values which omits consideration of operations.

To us many treatises and so-called methods, cf. UML, on ‘data modelling’, including [281], suffer from the lack of an appropriate formal linguistic “framework”, one that includes means of expressing types sufficiently flexibly and expressing operations and defining functions over the ‘data-modelled’ entities. OK, UML [127, 208, 245, 269] does, as shown in our Vol. 2, Sect. 10.3 (‘UML and RSL’), have means for lumping entities into object classes and naming operations (functions), but it is far too rudimentary and an orderly transition from UML Class diagrams to actual code is somewhat problematic.

6.2.2 Functions

Definition of functions is covered in Vol. 1 [87], Chaps. 6, 7, 11, 16 and 17 and in Vol. 2 [88], Sect. 5.3. In Vol. 1’s Chaps. 6 and 7 from the basic view of functions as mathematical entities. In Vol. 1’s Chaps. 11, 16 and 17 from the view of formal specification languages (here RSL).

The use of functions is covered throughout each and every chapter of Vol. 1–3 [87–89] as from Chap. 11 of Vol. 1 [87]. That chapter introduces the variety of ways in which to define functions: model-oriented explicit, axiomatic and pre/post definitions, and property-oriented axiomatic and algebraic (not that there is much difference !).

6.2.3 Events and Behaviours

The concepts and explicit modelling of events and behaviours are introduced in Vol. 1 [87], Chap. 21: ‘Concurrent Specification Programming’ — and is then used throughout all Vols. of [87–89].

6.3 Critique

Let us recall that the title of this chapter is: ‘Simple Entities, Functions, Events and Behaviours’ and that we are viewing how the book [87–89] is covering this modelling paradigm and whether an adequate coverage is provided.

1. *Specification Ontology*: Since [87–89] is primarily a textbook no comparisons are made to other ontologically structured ways of modelling than focused, “along the ontological dimension” of entities, functions, events and behaviours. Due references are occasionally made to such other ways, however, but maybe not very systematically.
2. *Mereology*: The presentation, in Vol. 3, Sect. 5.3.2 (‘Composite Entities’), as mentioned on Page 57 (Sect. 6.2.1), can be sharpened, and this was done in the published paper of Appendix A. Later I got the opportunity to express the modelling issue of composite entities in the essay contribution to Willem-Paul de Roever’s Festschrift [103]. I have included a draft of this essay as Appendix B.
3. *Events*: We could wish to have defined the concept of events more precisely (Vol. 1, Sect. 21.3.1 (Page 533), Vol. 3, Sect. 5.5.1 (Page 144)). A sharper definition has now been provided in Sect. 6.1.5 and an even more comprehensive one in Appendix Sect. B.6.
4. *Behaviours*: We could likewise wish to have defined the concept of behaviours more precisely (Vol. 3, Sect. 5.5.1 (Page 144)). A sharper definition has now been provided in Sect. 6.1.6 and an even more comprehensive one in Appendix Sect. B.7.

We have, throughout [87–89] avoided overly mathematical definitions of even technical terms such as mereology, event and behaviour. This lack of utter precision is judged OK in a basically software engineering textbook.

6.4 Conclusion

From Sect. 1.4, Item 5 on page 6 we edit:

The not insignificant contribution here is that we provide a simple description (prescription, specification) ontology in terms of simple entities, functions, events and behaviours.

Most (method cum software engineering) textbooks do not even bother to bring this issue up, let alone follow any one description (etc.) ontology.

Descriptions of Phenomena and Concepts

The notions of description and of phenomena and concepts are covered

- in Vol. 3 [89], Chap. 5:
 - ★ phenomena and concepts are specifically defined in Sect. 5.2.3, and
 - ★ description notion is specifically defined in Sect. 5.2.6.
- Chap. 6 of Vol. 3 [89] (‘On Defining and on Definitions’) expands on the concept of description.

7.1 The Issues

7.1.1 Phenomena and Concepts

In the domain there are phenomena, the things we can see, hear, smell, taste and feel (pressure, touch), or the things that can otherwise be measured by physical (including chemical) instruments. As also mentioned in Chap. 6, these phenomena, to us, and according to our modelling paradigm, constitute either simple entities, functions, events or behaviours. From phenomena we may form concepts. That is, one or more phenomena may be abstracted into further concepts — also modelled by simple entities, functions, events and behaviours.

7.1.2 Descriptions

We here use the term ‘description’ in some contexts as description of domains, in other contexts as prescription of requirements, and in yet other contexts as specification of software design — whether, for all three kinds, informally or formally.

Software Design Specification

Let us take the last first: specifications of software design. Here it is clear that the specification, when formal, in the final stage of development, amounts to program code. Therefore, when “informalising” such specifications of software design we use such Danish, or such English, etcetera, for which there is an informal mapping from the informal specification text to the formal specification text — and we are bound, by the choice of our programming language, in what we can specify. That which is specified must be according to the syntax of the programming language.

So, in the phase of software design, we know what we can informally describe: it must be expressible, also for higher level (than code) design, in the chosen programming language(s). But we can probably not otherwise describe that subset of English (or Danish) within which can informally express software design.

Requirements Prescription

In the phase of requirements engineering things are not that obvious. We are not algorithmising the requirements, we are first-and-foremost prescribing properties that should hold of any software design which claims to implement the requirements. And this property description may not itself be executable, yes, indeed, may not itself be computable. The requirements prescription may appeal to properties as follows: “such and such a software design which satisfies the following predicates”! This effectively means that we cannot expect there to be a commonly, say industry-supported requirements prescription languages like there are industry-supported programming languages.¹ The history of the design of programming languages is long, many diverse programming languages have been designed, and, let us hope so, the last has not yet been designed! There are very many aspects of requirements that we today, 2008, do not know how to capture formally and in such a way that we can more-or-less systematically, let alone rigorously or even formally, “derive” an efficient and provably correct software design.

So, for the phase of requirements engineering we can expect that formal requirements prescription languages can be, or are proposed or are available, but that also here the last has yet to be put forward. But we can still not otherwise describe that subset of English (or Danish) within which can informally express requirements prescriptions.

What makes the linguistic forms that are allowed in natural language discourse about requirements interesting is that whatever is being so specified must be computable, well, at least implementable in the form of computing and communication.

¹UML may be claimed to be an industry-supported requirements prescription language, but it is not formal and cannot be formalised.

Domain Description

Thus we reach the issue of domain description languages, both informal and formal. We can simplify the discussion, at this stage to:

- How can we make sure that some informal English (or Danish) text does indeed describe some phenomenon?
- What can be described?
- Are there phenomena and concepts (derived, or abstracted from phenomena) that we do not know how to describe informally?

The view taken in the book with respect to the first of these questions may best be summarised as done in the next three paragraphs.

Volume 1: Volume 1 [87] emphasises formal specification using the RAISE specification language RSL. Accordingly Vol. 1 more-or-less implicitly takes the view that what can be described is what can be formalised in RSL. Now that is, of course, not a satisfactory point of view, but *one must be able to crawl before one can walk!*

Volume 2: Volume 2 [88] then expands the (more-or-less implicit) view of Vol. 1 by introducing a number of additional formal specification tools, that is, diagrammatic as well as textual notations: finite and infinite state and tape machines, Petri nets, Message Sequence Charts (MSCs), Live Sequence Charts (LSCs), Statecharts and temporal logics, in particular that of the Duration Calculus. Now the view of what can be described is what can be formalised using any of these formalisms and RSL. Again that is, of course, not a satisfactory point of view, but *one must be able to walk before one can run!*

Volume 3: Volume 3 [89], with Chaps. 5 and 6, now focus on the “art of describing and defining” (and not on the means of formalising these). So it is only with Vol. 3 that the book gets concerned with the well-nigh philosophical (existential) and linguistics issues of that is superficially treated in the next subsection.

What Can Be Described?

It would be nice if one could say: when expressing yourself, in your informal English (Danish or other national language),

- if you only use nouns that designate known phenomena or concepts which are ‘defined’ (Vol. 3, Chap. 6) from known phenomena or already (so) defined concepts, or concepts that are being defined on that basis;
- if you only use verbs as names for functions that are either already known (but preferably also ‘defined’) in connection with known phenomena, or are being defined in connection with already, or with “at the same time” defined other verbs;

- if you only use adjectives that relate to the aforementioned kinds of nouns (or even, like ‘hedges’ to the aforementioned kinds of verbs) and such that these themselves are themselves objectively “measurable” (no “feelings”); and
- if you put these nouns, verbs and adjectives together in definite denoting phrases,

then, and only then, you are describing something objectively. But it is not clear whether that is sufficient — as it is quite clear that the wording of the four (“bulleted”) items can be made more precise.

One can always try to establish such a framework as hinted at above for natural language expressions (statement), but it seems that such a framework will always be lacking in being the largest subset, of a natural language, for expressing descriptions of the kinds we are interested in,

In other words: Vol. 3 [89] has not ventured into such a linguistics discourse. The author is simply not suitably educated and trained for a study of, let alone for formulating, such a ‘theory’.

7.1.3 On Russel’s Theory of Descriptions

In Appendix K (Pages 413–416) we bring an extensive, slightly edited quote from the Wikipedia Web page http://en.wikipedia.org/wiki/Theory_of_Descriptions.

• • •

So where does the extensive quote of Russel’s theory of descriptions (Appendix K on page 413) leave us? Not much, but just a bit wiser! Making precise that largest subset of our chosen natural, cum national language in which all sentences describe something definite is not possible.

7.1.4 Conclusion

So, the conclusion made in [87–89] was that we use such sentences for which we know how to formalise them in at least one of the languages covered in [87] and [88]: RAISE, FSA/FSM, Petri Nets, MSCs, LSCs, Statecharts and Duration Calculus.

7.1.5 Caveats: Narratives and Annotations

At times (mostly) we omit the term (or keyword) narrative “in front” of a narrative, i.e., an English language text which presents a description (prescription or informal specification). At times we use the term ‘annotation’, in the form of itemised informal text, in lieu of narrative. And sometimes such ‘annotations’ are used to explain formulas.

7.2 What Are Phenomena and Concepts?

So far we have somehow skirted the issue of what we mean by phenomena and concepts.

7.2.1 Phenomena

We simply say that phenomena are the things that we, as humans, can sense with our five sensory organs: vision (eyes), auditory (ears, hearing), somatosensory (skin, touch), gustatory (tongue/mouth, taste), olfactory (nose, smell), and (these are also) the things that can be measured (or recorded, in terms of IS units) by physical and chemical instruments: length (say metre), mass (say kilogram), time (say second), electric current (say Ampere), thermodynamic temperature (say Kelvin), amount of substance (say mole), and luminous intensity (say candela).

But this is far from a fully satisfactory explanation of what phenomena are! Add to the above the following: phenomena are “complexes” of one or more of the kind of observations that can be made as indicated above with the addition, to these complexes, of further attributes.

Here is where computing science “goes beyond” physics and mathematics: in describing domains, in prescribing requirements and in specifying software we are narrating and formalising universes of discourse, contemplated software-based systems and actual software that are far more “complex” in their structure and properties than most, if not all, engineering systems based on the natural sciences only.

7.2.2 Concepts

Combinations of the above International System (IS) of units give us Hertz (frequency), Newton (weight), Pascal (pressure), Joule (energy), Watt (power), Coulomb (electric charge), volt (Volterra, voltage), Farad (Faraday, electric capacitance), Ohm (electric resistance), etcetera, etcetera.

We can say that these and other abstractions of phenomena are concepts, and that abstractions of concepts are also concepts.

A remark similar to the one “quote”-indented above, at the end of Sect. 7.2.1, can likewise be made.

7.3 Coverage in Book

All three volumes of [87–89] abound with narratives (as well as annotations) of phenomena and concepts. Well-nigh any chapter! Each example, basically and especially, from Vol. 1’s Chap. 12 onwards: all of Vol. 2 and 3 abound

bears witness to some narration and some formalisation of some phenomena and/or concept. So there is hardly any reason to emphasise specific places in [87–89] where narrated and formalised phenomena and concepts are especially prominent.

7.4 Critique

7.4.1 Lack of Consistent Use of Narrative and Annotations

We can criticize [87–89] (cf. Sect. 7.1.5) for a lack of consistency with respect to the use of the terms narrative and annotation. In general we have not advocated any strict syntax for informal descriptions, prescriptions and specifications. A professional software house would either see to it that any given project or that all projects adhere to a strict syntax.

Appendix F (Pages 292–312), however, shows, what we consider a suitably consistent use, over a “large” example description, of narratives, formalisations and annotations.

7.4.2 Critique of Coverage of Concept of Descriptions

As is also clear from this chapter, the three volume book is not especially clear on the issue of *how does one use one’s national, i.e., natural language, in order to produce succinct narratives*. The issue continues to challenge computer and computing scientists, logicians, linguists and philosophers of language.

7.4.3 Critique of Coverage of Concept of Phenomena and Concepts

As mentioned above, we have not embarked on this, as we shall take it, philosophical area of (Hegelian) phenomenology and epistemology (what can we know). See Sect. 7.4.5 below.

7.4.4 Critique of Coverage of Descriptions, Phenomena and Concepts

We have thus chosen to only describe such phenomena and concepts for which we have a tool, i.e., a formal language, for their formalisation. This means that there are concepts that we have not described: emotions such as “user-friendliness”, “legalese” distinctions such as delinquency, sloppiness, etc.; and others.

7.4.5 Towards a Philosophy of Descriptions and Phenomena (*Etc.*)

A general critique could be that one must study the ontology, phenomenology and epistemology fields of philosophy deeply before “ordaining” principles and techniques for description and objects of description.

Let us hope that someone will one day do so and then, most likely, realign our principles and techniques for description and objects of description.

7.5 Conclusion

The not insignificant contribution of [87–89] is that we do try to elaborate on what it takes to describe (prescribe, specify) and what it is that one describes (prescribes, specifies).

Most (method cum software engineering) textbooks hardly cover these issues. Jackson’s [206] is the sole exception.

And the similarly, we claim, not insignificant contribution of [87–89] is the breadth (variety) and depth of exemplified phenomena and concepts.

Examples are good to learn from.

Abstraction and Modelling

*an abstraction is an instantiation of some metaphor*¹

The notions of abstraction and modelling are covered in [87–89] as follows:

- abstraction in Vol. 1, Sect. 12.1, Pages 232–235, and
- models and modelling in Vol. 3, Chap. 4, Pages 105–117.

But, of course, examples as well as principles and techniques of and tools for abstraction and modelling permeates most of [87–89].

8.1 The Issues

8.1.1 Abstraction

The essay, essentially by Sir Tony Hoare, in Vol. 1, Sect. 12.1.3, lists the issues of abstraction: (i) the, or an ability to focus on what is essential, and omit what is not; (ii) the “lifting” of those essential things, from the phenomenological level to a conceptual level by means of suitable abstraction and modelling concepts; (iii) with these “liftings”, these abstractions, being pleasant and their further properties revealing further essentials. The lifted abstractions, when relating to simple entities, are typically such as represent mathematical sets, Cartesians, sequences, maps and functions, that is, model-oriented abstractions, or such as represent algebras, that is sorts, observer and generator functions and axioms governing these (sorts and functions). Examples of abstractions formed from two or more of these could be: tables, trees, graphs, etcetera. The lifted abstractions, when relating to operations (functions) on entities, are typically expressed using pre/post conditions, further axioms, and

¹Metaphor: *the description of one thing as something else* — is one of very many characterisations of the term ‘metaphor’

suitable, beneficial non-determinism. Beneficial non-determinism, when used judiciously, oftentimes better reveal salient properties of the operations (functions) than if such are expressed “algorithmically”. Events and behaviours are, in [87–89], often, but not always, expressed within CSP.

8.1.2 Models and Modelling

You may think it strange that we postpone till Vol. 3 (Chap. 4) a deeper treatment of the concept of ‘Models and Modelling’. The reason is this: Volumes 1 and 2, of course, brings models, and says so, by stating, that the meaning of a specification is a model, or a possibly empty, or a possibly infinite, set of models — after having described what a model is (Example 1.7, Pages 35–37, Vol. 1). And Vols. 1 and 2 bring “zillions” of models. Usually these models, that is, their specifications, are from less than a page to 2–3 pages at most. It is not till Vol. 3 that we aim at modelling what may appear as large: in people’s mind, conceptually, and also physically, in terms of numbers of pages of specification. We therefore think it opportune to set aside an entire chapter to discuss types of models: analogic, analytic and iconic models (Vol. 3, Sect. 4.2.1, Pages 107–110), descriptive and prescriptive models (Vol. 3, Sect. 4.2.2, Pages 111–113), extensional and intensional models (Vol. 3, Sect. 4.2.3, Pages 113–115); and to discuss uses, that is, the rôles of models (Vol. 3, Sect. 4.3, Pages 115–116). Volume 3’s Chap. 4 applies equally well to any mathematical modelling, whether control theoretic, operations research or in computer or computing science or in software engineering.

8.2 Coverage in Book

In this section we re-interpret the notions of ‘abstraction’ and ‘modelling’ as follows: by ‘abstraction’ we mean the choice of presenting a simple phenomena or concept by a simple mathematical concept (see Sect. 8.2.1), and by ‘modelling’ we mean the choice of presenting a complex phenomena or concept by a composite mathematical concept (see Sect. 8.2.3). So abstractions, however expressed, denote models. And composing abstractions is then the act of modelling.

8.2.1 Abstraction

Years of observation has shown that the ability to abstract, to find pleasing, elegant and inspiring abstractions, cannot really be taught. It seems that some people have that ability, others not. But one can, or has to elicit it from those that have the ability — “extract it out of those” — by confronting them with appropriate abstractions: ‘learn by doing’. Surely there are some ground rules that must be obeyed. Vol. 1’s Sects. 13.7, 14.6.2, 15.6, 16.6, 17.5 and 18.11.2 summarise their principles, techniques and tools. These and most other chapters of [87–89] provide many examples of abstractions.

8.2.2 Property- Versus Model-Oriented Abstraction

Characterising phenomena and concepts by their properties, for example in the form of axioms over sort defined entity types and operations over entities, is generally thought of as representing “a higher level of abstraction” than using such mathematical entities as sets, Cartesians, lists, maps and function (entailing the use of the primitive operations on these “built-in” types). We are not always convinced. First we discuss property- and model-oriented abstraction in Vol. 1, Chap. 12. In that chapter, as in later chapters’ use of sorts and axioms as a mean to achieve property-oriented abstraction, we are quite satisfied with our examples being “more abstract” than possible model-oriented abstractions. But we have often seen published algebraic specifications (i.e., sorts and axioms) which start out by laboriously abstracting sets, list, maps, etc., in order to use these as auxiliary notions in subsequent “abstraction”.

8.2.3 Composite Modelling Patterns

But there are other abstractions than those based on properties and those based on sets, Cartesians, lists, maps and lifting (i.e., lifted functions).

We claim that the description patterns discussed next — by being presented as they are, namely as abstraction and modelling concepts — constitute a contribution of this Thesis, i.e., of [87–89]:

• <i>Applicativeness</i>	Vol. 1, Sect. 19.7.2
• <i>Imperativeness</i>	Vol. 1, Sect. 20.6.4
• <i>Concurrency</i>	Vol. 1, Chap. 21; Vol. 2, Chaps. 12–14
• <i>Hierarchies versus Compositions</i>	Vol. 2, Sect. 2.2.2
• <i>Denotations versus Computations</i>	Vol. 2, Chap. 3
• <i>Configurations: Contexts and States</i>	Vol. 2, Sect. 4.9
• <i>Time, Space and Space/Time</i>	Vol. 2, Chap. 5

Further abstraction and modelling patterns are presented in Vol. 3, Chap. 10: Domain Attributes.

• • •

When confronted with a “sizable” modelling challenge, the developer has to choose not only pleasing abstractions for “individual” phenomena and concepts but also how to compose one or more of these abstraction patterns into larger models. The above-listed “patterns” (styles) suggest alternative means.

These are just some such means. The book abounds with other, perhaps “less distinguished” patterns (styles).

8.3 Critique

[A first, minor critique could be that we do not elaborate on the term ‘pattern’ as “first” introduced just above. Our use may conflict with the meaning of the same term as used in ‘design patterns’².]

Pursuing abstraction and modelling, as touched upon in this chapter, represents, perhaps the most intellect-intensive part of engineering. As such it is hard, if not impossible, to set precise principles for choosing abstraction and model patterns. Once chosen, the techniques are otherwise covered in listed sections and chapters. But, again, much more could be said. The book [87–89], “by design”, cannot detail the abstraction and modelling principles, techniques and tools for other than general cases. By this we mean that the book should be studied as a prerequisite for studying books specialised in specific areas (cum domains) such as the design of (i) *real-time, embedded safety critical systems*, (ii) *programming language design and compilers*, (iii) *information [i.e., database] systems*, etcetera. Vol. 2, Chap. 15 hints strongly at development techniques for real-time, embedded safety critical systems, Vol. 2, Chaps. 16–18 and Vol. 3’s Sect. 28.2 (Translator Architectures) hints strongly at development techniques for interpreters and compilers, Vol. 3’s Sect. 28.3 (Information Repository Architectures) hints at some issues of database management system development.

8.4 Conclusion

The contribution wrt. abstraction and modelling is that very few (method cum software engineering) textbooks, and hence few claimed methods have focused so intensely on abstraction in connection with modelling and that we do so in a rather consistent and reasonably complete manner — while suggesting new ways of conceptually structuring abstract models (hierarchy and composition, denotation and computation, configurations: contexts and states, temporality and spatially, etc. — issues covered in Vol. 2). Included in the abstraction and modelling issue is that we show how to “UML-ise” formal techniques rather than formalise ‘UML’ (modularity, Petri nets, Message and Live Sequence Charts, and Statecharts — issues also covered in Vol. 2).

²A design pattern is a general reusable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. [Wikipedia]

On Informal and Formal Specifications

The notions of informal and formal specification are covered

- across the entire book, as from Vol. 1's Chap. 12 through Vol. 3's Chap. 28 — where most examples illustrate both informality and formality, and
- with Vol. 3's Sect. 2.5, Pages 70–84 highlighting some methodological principles and techniques of documentation: rough sketching, terminologisation and narration.

One way of characterising what we mean by the term 'formal' is:

- By informal expression we mean a precise expression in some natural, but possibly professional, that is, domain-specific language — possibly “mingled” with “easy-to-grasp” diagrams, drawings, pictures.
By a natural language we mean such as for example English.
- By formal expression we mean an expression in a formal, possibly diagrammatic language, and at least a language that has a clear formal syntax and formal semantics — and, preferably, also a proof system.

Another, more “exclusive” way of characterising what we mean by the an informal expression is:

- An expression (a diagram, etc.) is informal if it is not based on a formal syntax and, even if so, does not have a formal semantics.

9.1 The Issues

9.1.1 The Communication of Descriptions *Etc.*

A main issue here is that of reading. Although the software engineer develops the description (prescription or specification) for further development purposes, see next section, that description (etc.) has to be communicated: descriptions and prescriptions have to be accepted by domain, respectively

requirements stakeholders. (And they do not necessarily [have to] understand formal expressions.)

9.1.2 Developing Descriptions, Prescriptions and Designs

A main issue here is that of development. Is what has been described (prescribed or specified) what the domain or requirements stakeholders actually meant? Developing alongside informal expressions also their formal ‘counterparts’ sharpens the developers’ awareness of possible ambiguities of the informal expressions. And, once formalised, there is now the grounds for proving properties of what is formalised and for formally deriving, as it were, a number of aspects of requirements prescriptions from domain descriptions, as well as the software design specifications from requirements prescriptions — and proving, model checking or formally testing these developments.

9.2 Coverage in Book

We start by highlighting a specific pair of informal and formal descriptions:

Example 2.5 (Pages 40–41) Vol. 2 illustrates what we would characterise as an appropriate narrative. Example 2.6 (Pages 42–44) Vol. 2 illustrates what we would characterise as an appropriate (i.e., abstract) formalisation. The two examples taken together illustrate, in the way we prefer, a pair of coordinated descriptions: the numbered narrative statements correspond, one-to-one, to the numbered formula lines.

Already Vol. 1’s Examples 8.5 and 8.6 (Sects. 8.5.2–3, Pages 134–136) illustrates the pairing of informal narratives with formal specifications. So does Example 9.2 (Sect. 9.5.3, Page 173), and so on. We claim that basically all specification examples, in all volumes of [87–89], present the pairwise illustration of informal and formal specifications. The last example, most of Chap. 27 (Pages 547–578), likewise presents a stage- and stepwise development in the form of a formalisation alternating with informal texts.

Appendix F (Pages 292–312) shows, what we consider a suitably consistent use, over a “large” example description, of narratives, formalisations and annotations.

All these examples show the pairing of informal and formal texts in different ways. No strict guidelines are set for these “pairings”.

9.3 Critique

Possible points of critique of our treatment, in [87–89], of ‘Informal and Formal Specifications’ could be:

- *Consistency of Informality*: By the predicate ‘consistency of informality’ I mean: are the informal, the narrative, including the ‘annotation’ narratives, presented consistently in the same, albeit informal style ? The answer is, basically yes, but, in general, no !
That is, some readers, some lecturers, some students, may wish for a more consistent style. Some may even want a single, say annotation style throughout.
The style changes are deliberate. Each narrator should find an own style as long as all formalisations are also narrated. For a specific customer a strict narration style may be preferred.
- *Consistency of Formality*: By the predicate ‘consistency of formality’ I mean: are the formalisations presented consistently in the same style ? The answer is yes — with variations in style. The style variations are, in most chapters of all volumes, kept in the notation of the RAISE specification language RSL¹.
- *Derivations: Formulas from Narratives or Vice Versa*: As shown in the highlighted Example 2.5 (Pages 40–41) of Vol. 2 one could claim that formulas can be (albeit informally) “derived” from narratives. But one could claim the converse: that one can derive (“strict”) narratives from formalisations. Usually, in the engineering work, formulas arise as a result of informal analysis which includes verbal, usually also written (albeit sometimes rough sketch) expressions. Thus narratives may appear to precede formalisations — in time and, as a consequence, perhaps also “on paper”. As a further result of this and owing to documentary sloppiness the engineer may forget to update the narrative to express what the formulas express.
We have not “regimented”, let alone advocated, any development process whereby narratives are “kept a jour” with respect to formalisations. It is clear, however, that some such document quality assessment and control should be exercised in at least commercial projects.

A point of critique of whether the various phases, stages and step of software development prior to coding should be formalised (at all) is dealt with in the next section (Sect. 9.4).

9.4 Some Sociological Aspects of Formal Specifications

Somewhere in this thesis we must mention — and debate — the strange resistance there has been, in the broader software engineering and even computer sciences community, to formalisation of the various phases, stages and step of software development. This resistance continues in some isolated, but not insignificant quarters of the software community So we do that now.

¹Else the formalisations are kept in the notations covered in respective chapters: Petri nets (Vol. 2, Chap. 12), Message Sequence Charts (Vol. 2, Chap. 12), Live Sequence Charts (Vol. 2, Chap. 13) and State Charts (Vol. 2, Chap. 14).

The term ‘formal method’ has been attached to a development process which involves formalised specifications (of domains, or of requirements, or of “earlier” steps of software design) and therefore the possibility of formalised transition steps (with the further possibility of formal verification of the correctness of later steps wrt. earlier steps). From the early 1970s ‘formal methods’ were characterised by some software engineers (and software engineering scientists) as being too difficult for many, or even most software engineers to learn, let alone practice. Formal notations were criticized as being too mathematical. Lack of support tools were brought forward as an argument (when others did not suffice). We find that most “brick-heavy” textbooks in Software Engineering, still today — of course with the exception of [87–89] — either do not cover ‘formal methods’, or, when they do, “tuck” the treatment away in a chapter “somewhere”. Some such textbooks even state that ‘formal methods’ may at best be useful in connection with the development of real-time, safety-critical embedded systems !

We shall not dwell much on this debate here. But we shall mention the forceful rebuttals made, over the years and as witnessed by the following published articles:

- A. Hall: *Seven Myths of Formal Methods*.
IEEE Software **7**, 5 (1990) pp 11–19
- J. Bowen, M. Hinchey:
 - ★ Seven More Myths of Formal Methods.
IEEE Software, **7**, 5 (1990) pp 34–41
 - ★ Ten Commandments of Formal Methods.
IEEE Computer (April 1995) pp 56–63
 - ★ Ten Commandments of Formal Methods ... Ten Years Later
IEEE Computer, pp 58–66 (January 2006).

Section 32.2, Vol. 3 (Pages 680–685) covers these papers.

We do not hesitate to emphasize that [87–89] appears to be the only textbook cum monograph which fully integrates formal methods with all description (prescription and specification) development phases, stages and steps of software engineering.

A software engineer who cannot learn the formal techniques of [87–89] ought find another job. And: not all programmers need to learn these formal techniques !

9.5 Conclusion

The (minor) contribution here is that we very much emphasise the need for both informal and formal specifications — a need which is not recognised by most (method cum software engineering) textbooks, and hence by few claimed methods. We also emphasise the interplay between informal and formal specifications. This is practised and “preached” by

programming methodologists, but rarely subject to a more systematic treatment — as here.

On The Triptych of Software Engineering

The concept of ‘software engineering’ is first discussed

- in the Preface of Vol. 1, Page XV.

The concept of ‘a triptych of software engineering’ is first mentioned

- in Vol. 1, Sect. 1.2, Pages 7–12; then re-introduced
- in Vol. 3, Chap. 1; and otherwise covered in detail
- in Vol. 3, Parts IV–VI, Pages 193–668 !

The ‘triptych concept of software engineering’ covers that proper software development spans three phases:

- | | |
|-----------------------------|--------------------------------|
| 1. domain engineering | Vol. 3, Part IV, Chaps. 8–16, |
| 2. requirements engineering | Vol. 3, Part V, Chaps. 17–24, |
| 3. and software design | Vol. 3, Part VI, Chaps. 25–30. |

I am the one who has given the name ‘triptych’ to the concept presented in [87–89]. I claim that the tripartite approach to software engineering constitutes a major concept (and contribution) of [87–89].

10.1 The Triptych

In [87–89] we present the triptych dogma as follows:

- before software can be designed we must *understand* its requirements,
 - ★ and there is nothing new in this: conventional wisdom; and
- before requirements can be expressed we must *understand* the application area, or, as we call it, throughout, the domain
 - ★ and this is partially new.

To *understand*, to us, means that we must first describe the domain, respectively prescribe the required software (by its properties). Describing (prescribing) and subsequently reading, i.e., analysing the domain, respectively the requirements is our only means, we claim, of understanding.

The dogma implies the ‘ideal demand’ in the development of software application:

- First develop a domain description covering adequately more than is implied by the foreseen application and according to the principles laid down in Vol. 3, Part IV;
- then develop a requirements prescription according to the principles laid down in Vol. 3, Part V;
- finally develop a software design according to the principles outlined in Vol. 3, Part VI.

We have formulated the above as ‘an ideal demand’. We present the triptych as such. We only comment, after its presentation, on engineering ways of “short-cutting” the ‘ideal’. So we study and we teach the ideal in order that practicing engineers can accommodate to any current reality. That reality, in our mind, should eventually come close to the ‘ideal’.

10.2 Domain Engineering

The full “ensemble” of domain engineering stages and steps are covered in Vol. 3, Part IV, Chaps. 8–16.

10.2.1 The Issues

The core issue is that of constructing a model, i.e., a description of the domain, possibly through one or more refinements.

Ancillary issues are those of (i) acquiring insight into and rough-sketching a description of the domain, (ii) analysing this insight (the rough sketches), (iii) establishing a domain terminology, and (v) validating (and verifying) the model. (iv) The domain description then builds on (i–iii). In the long term, and as based on stable domain models, (vi) a domain theory ought be established.

Not only do I claim that promulgation of this major part of software development is a main contribution of [89], but I also claim that the method of developing domain models by developing them around the concepts of domain facets:

- | | |
|---------------------------------|---------------------------|
| 1. intrinsics, | 4. rules and regulations, |
| 2. supporting technologies, | 5. scripts, and |
| 3. management and organisation, | 6. human behaviour. |

We claim that the concepts of domain facets is new and that it has been “populated” with an appropriate number of (the above listed) facets. We repeat: the identification of domain engineering, as a separate software engineering phase, preceding requirements engineering is new and the domain facets are new.

10.2.2 Coverage in Book

All of the above stages, (i–v), are covered in Part IV, Chaps. 8–16 of [89].

10.2.3 Critique

Not all issues are covered to my fullest satisfaction. I shall comment on some of these.

[1] Domain Facets

The research process of identifying exactly the facets listed above (i.e., intrinsics, supporting technologies, management and organisation, rules and regulations, scripts, and human behaviour) evolved, as is natural, over many years and have stabilised and been subjected to numerous experiments. But other than each domain facet (after intrinsics) being either a refinement or a conservative extension of a former domain specification, we cannot characterise these facets as other than pragmatically determined. The question is therefore: Could there be another decomposition of domain modelling into other facets? [90] speculates over this and other of theoretical issues domain modelling.

[2] Domain Management & Organisation

The treatment of domain management and organisation in Vol.3, Sect. 11.5 is insufficient. Appendix I offers an alternative, we think, more adequate treatment. Yet, at this stage of development of the textbook¹, from an early version of which Appendix I is edited, even this appendix does not, in our mind, present a “best possible” treatment.

The problem, as we currently see it, is that the ‘management and organisation’ issue is a much larger and much more evasive issue than first realized. Much more research need go into understanding the phenomena and concepts that must and (meaning: ‘or’) should be modelled. We leave that as an open research problem.

¹Dines Bjørner: *Software Engineering*; approximately 400 pages; expected published 2009 by Springer

[3] Domain Scripts

The method principles and techniques unfolded in Vol. 3, Sect. 11.7, especially in Examples 11.21–.25 (Pages 288–307) are OK, but the base example is too “narrow”. We now find, after having worked, meanwhile, on the concept of license languages, that they, for example, illustrate a “broader” set of examples of script languages. We refer to Appendix H. With the principles and techniques of design given in Appendix H, ‘Three License Languages’, we now think that there is a suitable methodology for the identification and development of scripts nicely integrated into the entire methodology and concept span of [87–89].

[4] Domain Verification, Model Checking and Formal Testing

The broader issue of verification of properties of domain descriptions (requirements prescriptions, software designs, and of the relations between domain and requirements models, requirements models and software designs, and between refinements of descriptions, prescriptions and designs) is not covered in [87–89]. It is mentioned in several places, notably in Vol. 3, Sect. 14.2 (Domain Verification) [Vol. 3, Sect. 22.2] (Requirements Verification), Vol. 3, Sect. 29.5 ([Software Design] Verification, Model Checking and Testing)², but no verification principles or techniques are given.

In Vol. 3, Sect. 29.5.1 (Page 656 second half and Page 657) explains our position: why we do not cover verification (in the form of theorem proving) in [87–89]. We stand by this position. Please do not confuse the issues: We think formal specification is a must and can today be taught to an increasing large number of software engineering students and candidates. But we think that as of today, 2008, our science of computing has yet to find a suitable didactics and pedagogics for propagating formal (proof) verification.

The term verification is thus used, above, in the sense of theorem proving.

In Sects. 14.2 and 22.2 (still Vol. 3) we use the term to also cover ‘Model Checking’ and ‘Formal Testing’. Again the principles, techniques and tools of model checking and formal verification are also not covered in any serious depth in [87–89]. The current texts are all very much tool-oriented. The best book for model checking is still Gerard J. Holzmann’s *The SPIN Model Checker, Primer and Reference Manual* [192]. I have yet to find a suitable textbook on ‘formal testing’. But see John Rushby’s report: *Automated Test Generation and Verified Software*³ and [124].

²In Chaps. 14 and 22 we refer to ‘Verification’ and mean ‘[Proof] Verification, Model Checking and Testing’.

³Invited position paper for ‘Verified Software: Theories, Tools, Experiments’, Zurich, Switzerland, October 2005. Updated for LNCS volume due 2008 (ed. Bertrand Meyer): <http://www.csl.sri.com/~rushby/papers/vstte07.pdf>.

[5] Domain Theory

The issue of domain theory is characterised in Vol. 3, Chap. 15, but no actual illustration, in the form of formal theorems, is shown. The concept of a ‘domain theory’ is thus left more as a postulate than a reality. This is more-or-less explicitly acknowledged in Vol. 3, Sect. 32.4.2 (‘Grand Challenges’) where the grand challenge specter of ‘Domain Theories’ is raised on Page 690.

10.3 Requirements Engineering

The full “ensemble” of requirements engineering stages and steps are covered in Vol. 3, Part V, Chaps. 17–24.

10.3.1 The Issues

The core issue is that of constructing a model, i.e., a prescription of the requirements, possibly through one or more refinements.

Ancillary issues are those of (i) acquiring the requirements — such as covered in Vol. 3, Chap. 20, (ii) establishing a domain terminology (Vol. 3, Sect. 19.2.3), (iii, v) analysing the rough sketches (Vol. 3, Chap. 21–22) and the requirements model, and (v) validating (and verifying) the model (Vol. 3, Chap. 22). (iv) The domain description then builds on (i–iii). (vi) While constructing the requirements model it must be regularly checked that it is satisfiable and feasible (Vol. 3, Chap. 23).

Not only do I claim that promulgation of this major part of software development is a main contribution of [89], but I also claim that the method of developing requirements models by developing them around the concepts of domain and interface requirements is also a main contribution:

- | | |
|--|---|
| 1. Domain requirements: <ul style="list-style-type: none"> (a) projection, (b) instantiation, (c) determination, (d) extension, and (e) fitting. | 2. Interface requirements: <ul style="list-style-type: none"> (a) entities: initialisation and re-freshment, (b) functions: man-machine and machine-machine dialogues, (c) events, and (d) behaviours. |
|--|---|

We claim that the reformulation of functional and user requirements in terms of domain and interface requirements is relatively new and that this reformulation has been “populated” with an appropriate number of (the above listed) facets.

We also claim that [89] offers a very careful enumeration and analysis of machine requirements (cf. Item 2((e)iv on page 288) in terms of

- Performance
 - ★ Storage
 - ★ Time
 - ★ Software Size
- Dependability
 - ★ Accessibility
 - ★ Availability
 - ★ Reliability
 - ★ Robustness
 - ★ Safety
 - ★ Security
- Maintenance
 - ★ Adaptive
 - ★ Corrective
 - ★ Perfective
 - ★ Preventive
- Platform
 - ★ Development Platform
 - ★ Demonstration Platform
 - ★ Execution Platform
 - ★ Maintenance Platform
- Documentation Requirements
- Other Requirements

10.3.2 Coverage in Book

All of the above stages, (i–vi), are covered in Part V, Chaps. 17–24 of [89].

10.3.3 Critique

Not all requirements engineering issues are covered to my fullest satisfaction. I shall comment on some of these.

[1] Domain Requirements

The research process of identifying exactly the domain to domain requirements operations listed above (i.e., projection, instantiation, determination, extension, and fitting) evolved, as is natural, over many years and have stabilised and been subjected to numerous experiments. But other than each domain to domain requirements operation resulting in either a refinement or a conservative extension, we cannot characterise these operations as other than pragmatically determined. The question is therefore: Could there be another decomposition of domain requirements modelling by means of other operations ? We are curious !

[2] Interface Requirements

Interface requirements are covered in Vol. 3, Sect. 19.5 (Pages 429–445). A sub-division of interface issues is made on Page 430:

- *shared data initialisation requirements,*
- *shared data refreshment requirements,*
- *computational data and control requirements,*
- *man-machine dialogue requirements,*
- *man-machine physiological interface requirements,* and
- *machine-machine dialogue requirements.*

That sub-division is not well motivated. I have later found, what I now think is a much better way of creating a sub-division. It is based on the ‘specification ontology’ also highlighted in Chap. 6 of this thesis.

- *simple entities*,
- *functions*,
- *events*, and
- *behaviours*.

Of main importance in the identification of interface requirements is the notion of *shared* phenomena (and concepts), that is, sharing between the domain and the machine (being requirements prescribed). The concept of a shared phenomenon is highlighted in Vol. 3’s Sect. 19.5 (Page 429 and Example 19.20–.21) and in Sect. 19.6.

The new sub-division now comes about as follows: for each of the specification ontological categories (*simple entities*, *functions*, *events* and *behaviours*.) identify now the shared simple entities, functions, events and behaviours. Out of such an identification we can finally establish, or at least justify the original subdivision:

- **shared simple entities:**
 - ★ *shared data initialisation requirements* and
 - ★ *shared data refreshment requirements*;
- **shared operations:**
 - ★ *computational data and control requirements*,
- **shared events:**
 - ★ *man-machine dialogue requirements* and
 - ★ *machine-machine dialogue requirements*.
- **shared behaviours** — merging several of the above:
 - ★ *man-machine physiological interface requirements*

We refer to Appendix Sect. D.4.4, Pages 242–243. This new sub-division should be reflected in a new edition of [89]. Most, if not all of the material of Vol. 3, Sect. 19.5 is, however, still relevant.

[3] Requirements Verification, Model Checking and Formal Testing

We have basically covered this issue in section ‘[3] Domain Verification’ of Sect. 10.2.3 (Pages 82–82).

10.4 Software Design

A number of principles and techniques of software design are covered in Vol. 3, Part VI [89] (Pages 527–678). Many additional principles and techniques of software design are covered in Vols. 1–2 [87, 88], or are assumed covered in

texts studied prior to the study of [87–89]. We refer to such courses as are listed on Fig. 1 (Page XVII) and Fig. 2 (Page XVIII) of Vol. 1 [87].

Vol. 3 Part VI can therefore focus on the issues of software design that are specifically related to formal techniques as touted by [87–89].

10.4.1 The Issues

The core issue of software design is (i) that of constructing “executable code”.

Subordinate issues of software design are (ii) architecture design, illustrating one aspect of ‘refinement’; (iii) a version of component design, illustrating another aspect of ‘refinement’; (iv) some aspects of domain specific software architectures; and (v) a miscellany assortment of software design issues (Chap. 29).

10.4.2 Coverage in Book

Part VI’s Chaps. 26–28 cover the main issue, (i), of ‘transforming’, in stages, requirements prescriptions into low level software designs, i.e., a specification from where the programmers can “write the code”.

Most chapters of preceding volumes also serve to support this main issue.

Part VI does so by focusing, in turn on the specific subordinate issues: (ii) architecture design (Chap. 26), (iii) a version of component design (Chap. 27), and (iv) some aspects of domain specific software architectures (Chap. 28). They, by themselves, constitute major studies and reflect major activities of software engineers. They are only covered lightly: a major example for each of (ii) and (iii) and a survey for (iv).

We remind the reader that Vol. 3’s Part VI does rely on the kind of programming texts that were implicitly referred to above, in the first paragraph of Sect. 10.4. (A text on object-oriented programming, for example B. Meyer’s [234], is also recommended.)

10.4.3 Critique

[1] Current Fashions

Some of the “current”, or “ongoing fashions” of software engineering are briefly mentioned (Vol. 3, Sect. 29.3): (i) ‘extreme programming’ (XP), (ii) ‘chief-programmer programming’, and (iii) ‘object-oriented programming’ (OO), but “just so”, that is, only briefly. Other recent fashions, to wit, (iv) ‘aspect-oriented programming’ and (v) ‘model-oriented development’, are not mentioned.

The composite term ‘current fashions’ is, perhaps, a bit too arrogant a use of that term. A more polite composite term might be: ‘current principles and techniques’. These ‘fashions’ seem to “come and go”, i.e., to herald “current” interest for a while.

“Superficial” reasons for their brief or no mentioning are simple (i-ii) ‘extreme programming’ and ‘chief-programmer programming’ while using the principles and techniques of [87–89] is just fine, that is, can be practised while otherwise adhering to the principles and techniques espoused by [87–89]; and (iii) ‘OO-programming’ is covered in Vol. 2, Chap. 10. (iv) ‘Aspect-oriented programming’ (AOP) must be rethought in the context of the triptych development paradigm. The ‘separation of concerns’ (of AOP) is already part of the triptych paradigm. The “cross-cutting” concerns of AOP must be recorded already in the requirements prescription process. It seems that “cross-cutting” concerns expressed in requirements, quantify, thus at a meta-level, over “all” other, the “1st level” requirements. It also seems that to achieve aspect-oriented software development (AOSD) one may need a suitably “equipped” aspect-oriented programming language⁴. (v) Model-oriented development is just one of the many approaches that are fully covered in [87–89]. Remarks similar to the above can be made in connection with other “current software engineering fashions”.

But, really, there is a deeper reason for our “scant” treatment of the current fashions. And that is this: (a) with a comprehensive approach to software development such as carefully unfolded in [87–89]: from domains via requirements to software design, and (b) with concomitant and carefully identified principles, techniques and tools, these current fashions need be completely rethought. We have no intention of doing so. Some of them will then turn out, we think, to not be relevant, and others to lose some of their glamour. Very often claims are made as to the “huge” benefits of using these current fashions, claims that are not formulated as scientific statements, but rather as “hype”.

[2] Software Verification, Model Checking and Formal Testing

The verification of properties (also of software design, including correctness of ‘refinements’) is also not treated in [87–89].

We have basically covered this issue in section ‘[3] Domain Verification’ of Sect. 10.2.3 (Pages 82–82).

[3] Transition to Program Texts

We have not, in [87–89], covered in any detail, the final, few steps of software development from low-abstraction-level software design to actual, specific programming language executable code (Vol. 3, Sect. 29.1.5).

Basic features of current programming languages, to wit: Java and C#, include imperative features, such as covered in Vol. 1, Chap. 20; extensive type systems that are a subset of that of RSL (and richly covered already in Vol. 1); and modularity features such as those covered in Vol. 2, Chap. 10.

⁴— like Aspect J (for Java)

Given a chosen, current programming language, the software designer then “steers” the refinement “in the direction” of that language’s imperative, type system and modularity features.

The “quirks” of current programming languages are such that it would unnecessarily bias the book if we had chosen a particular one for illustrating the principles and techniques of transition from low-level RSL abstractions into that chosen language. Besides our text would increase, to basically little avail, by hundreds of pages !

The RAISE Tool set, as do other specification languages’ tool sets, provide for translation from low-level RSL into Ada, SML and Java. Again, for our book to go into detail on how to “gear” such a translation is not of programming methodological (cum computing science) interest. The practicing engineers can easily study that themselves.

10.5 Specific Software Development Models

In Appendix Sect. C.4.3 (starting Page 213) a ‘model’ of development of classes of software is outlined. The idea is that each class is characterised by “belonging” to a specific “software intensity”, but that the underlying domain may vary. There may thus be n kinds of software systems all “focused” more-or-less on the same software intensity — such as information handling by means of a database management system — and where the n software systems range over, for example, (i) transportation systems (i.e., the transportation domain); (ii) cadastral and cartographic, i.e., geographic information systems; (iii) manufacturing parts catalogues; etcetera. Different domains but, for “similar, narrow” (e.g., information intensive) requirements, basically the same underlying software implementation mechanism.

We should have like to have studied this concept more than just the “passing mention” in Appendix C. We consider this a major research topic, taking quite some resources, especially time.

10.6 Characterisations of SE and our Coverage of SE

In Sect. 1.1 of Vol. 3 [89] we bring, in Sect. 1.1.1, some conventional characterisations of software engineering while bringing our own in Sect. 1.1.2. It should be obvious that we cover, in [87–89], all that our characterisation touches.

Some characterisations of Sect. 1.1 of Vol. 3 imply one or another, or several, aspects of what is referred to as ‘management’. We have not covered those in [87–89]. Section 12.1.2 will comment on this and Appendix E will partially redress this omission.

10.7 Conclusion

The contribution here (a major one) is that no (method cum software engineering) textbook, and hence no claimed methods have heretofore introduced or covered the notion of domain engineering and the reliance of requirements engineering on domain models. We do so in Vol. 3, and rather extensively so.

No-one has taken such a “radical” stand on domain engineering as we have in [89]. Many have, seriously and substantially, mentioned and given some examples of domain analysis, but then almost exclusively as part of the requirements engineering process — not highlighted as an altogether separable and independent activity.

We take the stand that ideally domain engineering should be dispensed with before requirements engineering. And we show how to “transform”, to refine, to extend, a domain description into a domain requirements prescription and also, partially, because it can only be partially, into an interface requirements prescription.

Documentation

The notion of documentation is covered

- in Vol. 1, Sect. 1.3 (Pages 13–25) to some depth, and
- in Vol. 3, Chap. 2 (Pages 53–90), in more detail.

11.1 The Issue

It may appear, to many readers, many students, many software engineering researchers and teachers, that the “laborious” concept of documentation brought forward in Vol. 3, Chap. 2 may be a bit of an “overkill”. Obviously we do not think so. Almost all the results of software development, from domain engineering via requirements engineering to software design are documents: either we construct them or we analyse them (and our analysis results in documents !). In so many actual development projects we find that documentation is not taken serious.¹

Appendix Sect. E.9 (Pages 287–290) lists the very many kinds of documents that we think should result from a proper software development.

• • •

At this point we kindly ask the reader to browse those pages.

¹In traditional engineering, viz., civil engineering, drawings (at all levels of abstraction, from the architects’ first conceptions and final drawings via the structural and other engineers’ various embellishments of the architectural plans: the structural engineers’ [for example] introduction of reinforced concrete supports, the electrical engineers’ wiring diagrams, the heating and sanitary engineers’ hot and cold water and waste piping, etc., to the construction supervisors annotations of these drawing document) are taken very serious: drawings of constructions are kept for as long as the construction stands, and beyond, and are carefully inspected whenever construction maintenance is undertaken.

• • •

The concept of ‘document’ — for whatever purpose — is thought to be so important that we bring, in Appendix G, a discourse on documents. This appendix is formulated “sporadically” using some, but not all of the domain engineering principles, techniques and tools.

11.2 Coverage in Book

When we, for example, say a ‘domain description’ we mean a complete one. Similar for ‘requirements prescription’ or ‘software design’. We do not illustrate such complete documents in [87–89].² But we claim that the kind of document extracts that we do show do indeed scale up. And we claim that we have indeed illustrated (or, where that is sufficient, referred to) each and every document kind in [89].

11.3 Critique

There are some aspects of our coverage of documentation which could be improved. Some such are treated next.

11.3.1 Synopsis: Vol. 3, Sect. 2.4.4 (Page 63)

To construct a pertinent synopsis (a capsule description of the “project, at hand”³ is an art. It involves “telling a story” while “presenting pertinent texts” from most emerging informative, description (prescription or design), and relevant analytic documents — even when these may yet have to be written. My ‘Synopsis’ text could be improved.

²I cannot, off hand, refer to any textbook in programming which brings a complete program for other than trivial problems. A non-trivial domain description, informal and formal, would require at least around a hundred pages. On the author’s home page there are URLs to domain description documents which might eventually stabilise — and then be at least a hundred pages “long” ! Three such are (i) ‘An Emerging Report on the Financial Services Industry Domain’ <http://www2.imm.dtu.dk/~db/fsi.pdf>, (ii) ‘An Emerging Postscript Report on the Transportation Domain’ <http://www2.imm.dtu.dk/~db/transport.ps> and ‘A Container Line Industry Domain’ <http://www2.imm.dtu.dk/~db/container-paper.pdf>.

³*Project Kind*: — whether that project be of kind

- a domain engineering,
- a requirements engineering,
- a software design,
- a domain plus requirements engineering,
- a requirements engineering plus software design, or
- a full domain plus requirements engineering plus software design project.

11.3.2 Implicit/Derivative Goals: Vol. 3, Sect. 2.4.6 (Page 65)

For each project kind⁴ a different kind of ‘Implicit/Deviate Goals’ section has to be written. This is not made crystal clear from the text (Vol. 3, Sect. 2.4.6 (Page 65)). It has also not been made clear that it is often hard, if not impossible, to extract, from stakeholders, that which constitute their (sometimes subliminally “experienced”) goals.

11.3.3 Tools

Implicit, in Vol. 3’s Chap. 2, is the need for documentation tools. Other than Exercise 2.8’s ‘Requirements for a Document Tool’ (Page 91), I believe that I have not stressed that aspect sufficiently.

11.4 Conclusion

The contribution here is that no (method cum software engineering) textbook, and hence no claimed methods have heretofore spelled out nor, really, emphasised the pivotal importance of documentation let alone the various types and kinds of documents that need be produced as an integral part of software development, and that I do so in a rather consistent and reasonably complete manner.

⁴Cf. the enumeration of project kinds given in Footnote 3 on the preceding page.

Friday August 1, 2008: Dines Bjorner Dr.techn. Thesis

Part III

SOME NON-ISSUES

On Omissions

Chapters 2–11 have, in some of their ‘Critique’ sections, covered some of issues that are normally considered “belonging” to software engineering but which are not treated in [87–89]. In the present chapter we shall review some of these as well as take up some other omissions.

12.1 Main Omissions

The two main omissions are at “opposite ends” of some conceived software engineering “spectrum”: verification and management. “Opposite” in the sense that one requires programming methodological and mathematical (logic) skills, presently of a highly theory-oriented characters, the other requiring practical, economic and organisational skills not specifically related to computing science.

12.1.1 Verification

We have already, in at least three places:

- Item 3 “*More Serious Omissions*” (Sect. 4.3.3, Page 37),
- Quote-indented Item “*A Critique: Absence of Material on Verification*” (Sect. 5.3.2, Page 47) and
- Item [3] “*A Critique: Absence of Material on Verification*” (Sect. 10.2.3, Pages 82–82),

covered the issue of not treating verification to any substantial depth in [87–89]. We refer the reader to those items.

Here we shall further motivate our decision to not treat verification, especially in the form of theorem proving, in [87–89].

The term ‘verification’, to remind the reader, is, in Sects. 10.2.3, 10.3.3 and 10.4.3, taken to mean theorem proved, model-checked and formally tested verification. We do consider our treatment of these latter two sufficient: Based on choices of specific formal specification and of programming languages the software engineers, familiar with [87–89], can themselves “pick-up” the tools (and often rather tool-specific, specialised texts) that help in model checking and in formally testing (formal description, formal prescription, and formal software design) specifications.

Our main reason for not bringing any substantial material on theorem (and/or proof assistant) proved verification is this:

We think that current principles, techniques and tools for theorem (and/or proof assistant) proved verification are such as to demand “heroic” efforts on behalf of the practicing software engineer.

This is not a critique of these theorem and/or proof assistant verification techniques and tools. It is merely an assessment of today’s (August 1, 2008) state-of-the-methodology. Great efforts are being made these years towards improved techniques and improved tools. But “*what is now a heroic achievement must become routine !*”¹

A prerequisite for theorem (and/or proof assistant) proved verification, as for model checked and formal test-based verification, is that there is a formal specification — and we have covered that, extensively. Formal specification does not “stand or fall” with the ability or inability to formally theorem (and/or proof assistant) verify properties of formal specifications or their refinements, etc. Formal specifications have many other benefits — as we think we have already explained in [87–89] — as well as summarised in Items 1–6, Sect. 5.1 (Pages 41–43).

A Remedy

Section 1.2 of Vol. 2 (Pages 6–8) suggests a remedy. Ten published papers are listed which can serve as a basis for a course on verification. We bring that list as Appendix M (Pages 431–432).

¹Statement in a Friday, February 9, 2007, 10:25 AM E-mail sent from Sir Tony Hoare to a group of researchers who had successfully reported on the proven development of a Mondex [<http://vsr.sourceforge.net/mondex.htm>] case study also pursued by many other researchers, worldwide.

12.1.2 Management

Pages 5–6 of Vol. 1, Chap. 1, Sect. 1.1 (‘Setting the Stage’) mentions a number of software development project management issues (choice and planning of development process, scheduling and allocation of resources, monitoring and control of work progress, monitoring and control of quality: assurance and assessment, version control and configuration management, legacy systems, cost estimation, and legal issues). But other than this no details on these issues are given in [87–89] !

We see the issue of software management as basically constituted by two sets of (more-or-less loosely related) management issues: those of projects aimed at developing an aspect of software ((1) a domain description, (2) a requirements prescription, (3) a software design, or a subset of two or three of these [(1–2), (2–3), or (1–3)]; and those of product management (marketing survey, product [family] identification, sales, service, etc.). We see many of the issues as being rather specific to software, and some of the issues as being common to any product management. We have, in the 1960s and in the 1980–1990s, been involved in the management of several software development projects and in several product management issues — but, for the latter, not in any responsible rôle.

Most of the issues of software project development management mentioned above: choice and planning of development process, scheduling and allocation of resources, monitoring and control of work progress, monitoring and control of quality: assurance and assessment, version control and configuration management take on a new meaning, and, we think, must be rethought “from scratch”, as compared with the meaning of these issues “in fashion” in the 1980s and 1990s (and as covered in common textbooks such as [251, 253, 280]).

An attempt at such a re-evaluation has been made and is reported in Appendix E. In that appendix² we study the management issues of assessing and improving the development process. Assessment and improvement is with respect to the five *capability maturity model* criteria propagated by Watts Humphrey [194].

But a larger study of software project and product management in the context of formal techniques and a corresponding monograph cum textbook has yet to come. Any takers ?

12.2 Other Omissions

The field of “classical” software engineering, such as described in [169, 251, 253, 280, 294], posing so very many methodology challenges, attracts many software engineers and computing scientists to contribute to their study. In this

²— awaiting publication in *Encyclopedia of Software Engineering* [93, to appear], editor: Philip A. Laplante (Taylor & Francis)

section we shall very briefly review some of their contributions — otherwise not covered in [87–89].

12.2.1 Refinement Calculi

An important program development paradigm is that of refinement calculi [9, 238]. Inasmuch as the refinement calculi provide a fascinating way of developing programs I do not think it worthwhile to bring material on such refinements in [87–89]. The kinds of refinement calculi covered by [9, 238] should most definitely be taught in M.Sc. courses in computing science, but, at present they do not seem to scale up to software engineering use in the development of large scale software systems. I hope I am either wrong in this assessment or that I will someday be proven wrong !

12.2.2 UML

The UML we are referring to is represented, for example, by the three books by the three “amigos” authors Booch, Jacobson and Rumbaugh [127, 208, 269].

We have not covered UML as such, but we have done better, I think: we have, in Vol. 2 [88], presented the very basics of some of the techniques and tools of UML: finite state automata and machines (Chap. 11), Modularity (object-orientedness and class diagrams, Chap. 10), Petri nets (Chap. 12), Message and Live Sequence Charts (Chap. 13) and Statecharts (Chap. 14).

Section 20.4 (basically Page 712) of the closing chapter of Vol. 2 explains our view on UML.

The strengths of using finite state automata and machines, class diagrams, Petri nets, message or live sequence charts and statecharts (and hence of their “counterpart” use in UML) is that they provide for oftentimes pleasing use of visuality (i.e., of figures, diagrams). That is why “UML-ise” the diagrammatic formal techniques and tools (finite state automata and machines, class diagrams, Petri nets, message or live sequence charts and statecharts) by bringing them to “blend” with the textual formal techniques and tools of the RAISE specification language RSL. Much more work need be done to further this goal of “UML-ising” formal techniques and tools rather than of formalising UML.

12.2.3 $\exists\forall$: Intentional Programming³

The intentional software development paradigm is the creation of Charles Simonyi⁴.

³This section is copy-edited, with permission from IPSJ/SIGSE, from: Dines Bjørner: The Rôle of Domain Engineering in Software Development. Invited keynote paper and talk: IPSJ/SIGSE Software Engineering Symposium 2006, Oct. 21, 2006, Tokyo

⁴ $\exists\forall$ Intentional Software, Bellevue, Washington, USA; <http://intentsoft.com>

It appears that little if any literature is readily accessible [3, 277–279]. So we shall resort to quoting from *Intentional Software*'s Web page (<http://intentsoft.com/technology/glossary.html>). The quotes are in *slanted font*.

Domain: A domain is an area of business, engineering or society for which a body of knowledge exists. Examples include health care administration, telecommunications, banking, accounting, avionics, computer games and software engineering.

Domain Code: Domain code is the structured code to represent the intentions contributed by subject matter experts for the problem being solved. Domain code includes contributions from all domains relevant to the software problem. Domain code is not executable (as traditional source code is - by compilation or interpretation), but it can be transformed into an implementation solution when it is input to a generator that has been programmed to perform that transformation process.

Domain-Oriented Development: Domain-oriented development is the process of separating the contributions of subject matter experts and programmers to the maximum extent so that generative programming can be applied to structured domain code. This greatly simplifies improvements to the domain and implementation solutions.

Domain Schema: A domain schema is a schema for a specific domain. The domain schema defines the domain terminology and any other information that is needed — for the intentional editor and generator to work — such as parameters, help text, default values, applicable notations and other structure of the domain code. Domain schemas are created by the subject matter experts and programmers working together, and are expressed in a schema language.

Domain Terminology: Domain terminology means the terms of art (words with a special meaning) in a domain, for example “claim payment” in health care administration. Domain terminology is important because it is the usual way to express intentions. Broadly speaking, terminology includes notations normally used by a subject matter expert, such as tables, flowcharts and other symbols. The meaning of the terms is part of the domain knowledge that is shared between subject matter experts and programmers to the extent necessary and ultimately designed into domain schemas and the generator.

Discussion

Intentional software development, it should be clear from the above builds on a number of software development tools which are provided with domain description-like information and which can then significantly automate code generation. Other than that shall neither comment nor speculate on Charles

Simonyi's characterisations.⁵ We believe that the reader can easily see the very tight relations to the triptych phases of development. We find them fascinating. $\exists\forall$ seems to deserve a rôle in making the triptych approach even more viable.

12.2.4 Extreme Programming (XP)

We refer to the following Web pages for information about 'Extreme Programming': <http://www.extremeprogramming.org/> and http://en.wikipedia.org/wiki/Extreme_Programming#XP_values and to [21, 22, 229].

As for other programming "fashions" the 'value system' of **XP** (captured in the concepts of *communication*, *simplicity*, *feedback*, *courage* and *respect*) fit nicely in with the triptych paradigm and with its insistence on both informal and formal development. One can consider them "add-on" paradigms.

12.2.5 Web Programming

The Web is a system of interlinked hypertext documents accessed via the Internet.

As such the Web constitutes a domain⁶ — and a machine. We consider the Web domain to be an abstraction of the machine.

By Web programming we mean an activity that results in the establishment and/or linking of hypertext documents accessible via the Internet. The problem, as I see it, with respect to Web programming, is that there is no domain description of the Web⁷, and hence it is somewhat difficult to express requirements to new Web services in a concise, implementation-free manner.

So I do not cover this most fascinating issue since I have not had the not inconsiderable resources it takes to dig deeper into the Web standards, to thus extract a description of the Web domain, and, from that develop example requirements and software design proposals.

⁵Well, I cannot, of course, refrain from saying that several of my students have founded a number of Danish software companies each of whose corporate asset is a set of domain-specific "automatic" code-generators.

⁶Here we beg the reader's indulgence: when we say 'the Web domain' we do not refer to some (decorated) URL, i.e., some domain name etc., but the universe of discourse that revolves around the Web; that is, we do not refer specifically to the DNS (domain name system) of the Internet.

⁷See, however, http://en.wikipedia.org/wiki/Web_standards whose referenced documents describe implementation-oriented issues of the Web.

Instead I refer in footnote⁸ to some pertinent literature. I have here, in footnote⁸, emphasised the likewise fascinating use of continuations (Continuations are covered in Vol. 2, Sects. 3.2.4, 3.3.4 and 3.3.5).

12.3 Conclusion

We have covered what appears to be the “most glaring” omissions of [87–89]. In our opinion these omissions are well-justified.

⁸See: <http://www-128.ibm.com/developerworks/java/library/j-cb03216/>: Crossing borders: Continuations, Web development, and Java programming. A stateful model for programmers, a stateless experience for users. Bruce Tate (bruce.tate@j2life.com), President, RapidRed.

Survey of Web programming languages: <http://www.objs.com/survey/lang.htm>.

Some publications:

1. William E. Byrd: Web Programming with Continuations, November 20, 2002, <http://www.double.co.nz/pdf/continuations.pdf>.
2. Pettyjohn, G., Clements, J., Marshall, J., Krishnamurthi, S., and Felleisen, M.: Continuations from Generalized Stack Inspection. In Proceedings of the Special Interest Group on Programming Languages (SIGPLAN) International Conference on Functional Programming (Sep 2005), pp. 216-227.
3. Matthews, J., Findler, R. B., Graunke, P. T., Krishnamurthi, S., and Felleisen, M.: Automatically Restructuring Programs for the Web. *Automated Software Engineering Journal* 11, 4 (2004), 337-364.
4. Graunke, P. T., Findler, R. B., Krishnamurthi, S., and Felleisen, M.: Modeling Web Interactions. In Proceedings of the European Symposium on Programming (Apr 2003), pp. 238-252.
5. Graunke, P. T., Krishnamurthi, S., van der Hoeven, S., and Felleisen, M.: Programming the Web with High-Level Programming Languages. In Proceedings of the European Symposium on Programming (Apr 2001), pp. 122-136.

Friday August 1, 2008: Dines Bjorner Dr.techn. Thesis

Part IV

CONCLUSION

The Thesis Reviewed

An Essay: Reflections

We claim that computing science is an empirical science only insofar as observations are made on the quality of software produced by software engineers before and after they have been taught and learned to develop selected principles, techniques and tools of software engineering (programming).¹

In the above paragraph we first used the term ‘computing science’ and then the term ‘software engineering’: we meant only to include, in the above use of the term ‘computing science’ those principles, techniques and tools which carry over to software engineering.

For the purposes of this thesis proposal I do not include observations of the psychological and sociological behaviour of practicing software engineers — inasmuch as these may be relevant.

• • •

This thesis proposal brings software development principles, techniques and tools that can be taught to and learned and practiced by software engineers.

We have observed, through more than 30 years of lecturing, of tutoring and of guiding major software development projects, that software development principles, techniques and tools can be successfully taught, learned and practiced to — already in the late 1970s — approximately half our MSc students every year.

Some comparative studies has been made comparing the use and non-use of formal techniques. A first such is reported in [141]. Another such is reported in Formal and Informal Specifications of a Secure System Component: Final Results in a comparative study. T.M. Brookes,

¹The ‘principles, techniques and tools’ are those of computing science ‘selected’ for use in software engineering.

J.S. Fitzgerald, and P.G. Larsen; *and in FME'96: Industrial Benefit and Advances in Formal Methods*, Springer, March 1996. *These and many other observations have unambiguously shown that it does indeed help, one way or the other, to have studied and learned formal techniques.*² *I do not believe in formal techniques. I just think they are rather more useful and fun to use than not using them.*

Thus I am bringing the material of [87–89] simply because it is possible to develop software using the principles, techniques and tools of these volumes, because it is fun teaching, learning and using this approach, and because it is of scientific interest to study these principles, techniques and tools — and to report on such studies.

13.1 Reminder: Thesis Documents

This Dr.techn. Thesis proposal is made up from the following documents:

1. The three volume book D. Bjørner: *Software Engineering* [87–89]:
 - (a) *Vol. 1: Abstraction and Modelling* (Springer, 2006)
 - (b) *Vol. 2: Specification of Systems and Languages* (Springer, 2006)
 - (c) *Vol. 3: Domains, Requirements and Software Design* (Springer, 2006)
2. The document in which you are reading this just now:

Software Engineering — an Unended Quest, August 1, 2008:

 - ★ Pages V–VIII and 1–120 and Appendices A–F (Pages 123–290).
 - ★ Appendices G–L are not to be considered part of the thesis.

13.2 What Is Being Claimed ?

Chapter 1 first surveyed the ten thesis claims. Chapters 2–11 have then examined these claims in more detail.

We shall summarise these claims, but we shall now reorder the claims such that what we consider the main contributions are summarised first.

13.2.1 A Contribution of Methodology, Didactics and Pedagogics

[0] Meta Contribution

Claim: We claim that [87–89]’s partial ordering and interweaving of many topics, the exposition of their interrelations, and their careful balancing and treatment, contributes to programming methodology in the area of software engineering (all volumes).

²Please note that this is the first time such a claim is made in this thesis and in [87–89] — that is: as far as I can recall. It is now at least four years ago I finished writing (but not copy-editing) [87–89].

13.2.2 Main Methodology Contributions

We consider the next six claims ([1], [1.1–.5]) justified and necessary and sufficient, as a whole, to justify the awarding of the degree of Dr.Techn.

[1] The Triptych Paradigm

Claim: We claim that the triptych paradigm of developing software by first describing the domain, then “deriving” major parts of requirements from a domain model, is new and that our treatment of this paradigm is relevant (Vol. 3, Parts IV–VI).

[1.1] *Domain Engineering*

Claim: We claim that the concept of domain engineering, in the radical form present in [89], is new and that our treatment of this concept is relevant (Vol. 3, Part IV).

[1.2] *Domain Facets*

Claim: We claim that the concept of domain facets is new and that their attendant principles and techniques are new and that our treatment of these are relevant (Vol. 3, Chap. 11). We do not claim they are “final”, “universal” — only that they are of relevance.

[1.3] *Domain to Domain Requirements Operations*

Claim: We claim that the concept of domain to domain requirements operations is new and that their attendant principles and techniques are new and that our treatment of these are relevant (Vol. 3, Sect. 19.4). We do not claim they are “final”, “universal” — only that they are of relevance.

[1.4] *Domain to Interface Requirements Operations*

Claim: We claim that the concept of domain to interface requirements operations is new and that their attendant principles and techniques are new and that our treatment of these are relevant (Vol. 3, Sect. 19.5). We do not claim they are “final”, “universal” — only that they are of relevance.

[1.5] *Machine Requirements*

Claim: We claim that the enumeration of machine requirements is relevant and is a first in a software engineering textbook (Vol. 3, Sect. 19.6).

13.2.3 Main Supporting Methodology Contributions

We consider the claims of this section ([2–8]) justified and to justify the claim ([0]) of the proposal offering a contribution to the didactics and pedagogics of the field of (teaching and studying) software engineering.

[2] Simple Entities, Operations, Events and Behaviours

Claim: We claim that the specification ontology of simple entities, operations, events and behaviours and that our insistence on their use throughout constitutes a contribution to software engineering (Vol. 3, Sects. 5.3–.5).

[3] Informal and Formal Descriptions

Claim: We claim that the suggested (and exemplified) consistent use of both informal, narrative and terminology, and formal specifications constitute a contribution to software engineering (Vol. 3, Sect. 2.5).

[4] Phenomena and Concepts

Claim: We claim that our emphasis on considering and the consistent reference to (domain) phenomena and (domain) concepts constitutes a contribution to software engineering (Vol. 3, Sect. 5.2).

[5] Method Principles, Techniques and Tools

Claim: We claim that our careful definition of what a method is and its enunciation in terms of principles, techniques and tools constitute a contribution to software engineering (Vol. 3, Chap. 3).

[6] Semiotics: Pragmatics, Semantics and Syntax

Claim: We claim that our emphasis, in all phases of software development, on the use of the semiotic concepts of pragmatics, semantics and syntax constitutes a contribution to software engineering (Vol. 3, Chap.).

[7] Documentation

Claim: We claim that our identification of the myriad of documents that need be produced during development and the identification of their properties (eg., their careful structuring and contents) constitutes a contribution to software engineering (Vol. 3, Chap. 2).

[8] Abstraction and Modelling

Claim: We claim that the very many abstraction and modelling concepts, principles and techniques, throughout all volumes of [87–89], and the myriad of examples of abstract models constitutes a contribution to software engineering (Vols. 1–2 and Vol. 3 Chap. 4).

13.2.4 Ancillary Methodology Contributions

We consider the two “small” claims of this section ([9–10]) to be necessary and justified — supporting the didactics of our other contributions.

[9] Mathematics

Claim: We claim that the emphasis on using — especially discrete mathematics — and on models denoting mathematical structures is a clear textbook contribution (Vol. 1, Part II).

[10] Computer versus Computing Science

Claim: We claim that the clarification of a spectrum from computer to computing science and the emphasis that software engineering is “applied” computing science is a clear textbook contribution.

13.3 Does ‘This’ Constitute A Dr.Techn. Proposal ?**13.3.1 An Answer**

The answer to the section display line question is yes:

- The three volume book [87–89] *together with* the present document, contributes to the methodology of (“how to do”) software development, thus to technology.
- [87–89] *etc.*, does not contribute (much) to computer science, and is thus not to be reckoned as a Dr.Scient. or a Dr.Phil. thesis proposal.
- The current thesis builds on 30 years of research.
From those years I list, at the end of Sect. 13.3.4, 94 of my publications whose content has found its way, in one form or another, into this thesis.

13.3.2 Some Justification

I mention just a few early and one recent industry-related formal techniques (etc.) activities of mine.

- My work, as from the earliest listed publications, see Sect. 13.3.3, has influenced mostly researchers and engineers in Europe.
- My students' work on VDM has lead
 - ★ to an ISO VDM Standard³,
 - ★ and to an industry-strength VDM Tool Set which was acquired by a large Japanese software house⁴.
- Two large, international industry-scale projects, CHILL and Ada were initiated by me. Their results were:
 - ★ a formal definition of CHILL which became part of the official CCITT (now ITU) CHILL Recommendation [174]⁵;
 - ★ a full CHILL compiler, the only compiler implementing all of CHILL [176];
 - ★ a formal description of Ada [121];
 - ★ a full Ada compiler, the first in Europe, [141];
 - ★ a commercial (Danish) software house, in existence since 1984, devoted to Ada and related products: DDC Inc. (Phoenix, Ariz., USA) <http://www.ddci.com>;
 - ★ a formal definition of Ada (a joint Danish/Italian project)⁶.
- With my colleague, Prof. Cliff Jones (then at Manchester Univ., now at Univ. of Newcastle), I founded VDM Europe in 1987; as chairman I had VDM Europe renamed into called Formal Methods Europe (FME) in 1991.⁷ FME is “a worldwide association bringing together researchers and practitioners in formal methods developing computing systems and software”. VDM and Formal Methods Europe have organised international symposia and conferences every approximately 18 months since March 1987. These are unquestionably the leading international events in formal methods.

³ISO/IEC 13817-1:1996. An Overview of the ISO/VDM-SL Standard (1992), Nico Plat, Peter Gorm Larsen, ACM SIGPLAN Notices, September 1992. [http://www.webstore.ansi.org/RecordDetail.aspx?sku=INCITS%2fISO%2fIEC+13817-1-1996+\(R2007\)](http://www.webstore.ansi.org/RecordDetail.aspx?sku=INCITS%2fISO%2fIEC+13817-1-1996+(R2007))

⁴CSK Holdings: http://www.csk.com/support_e/vdm/index.html

⁵ITU, CHILL Formal Definition, Volume I, Parts 1, 2, 3, and Volume II, Part 4, Geneva, 1982.

⁶(i) Botta, N., Petersen, J. Storbank: The Draft Formal Definition of Ada, The Static Semantics Definition, Vol 1-4 Jan 87, Dansk Datamatik Center, Lyngby, Denmark; (ii) E.Astesiano and J. Storbank Pedersen: An Introduction to the Draft Formal Definition of Ada. In Proc. 3rd Workshop on Ada Verification, Triangle Park, USA, 1986.

⁷<http://www.fmeurope.org/>

- I was the “founding father” of ForTIA⁸ which is an association of industrial organisations who use and/or supply tools, ideas and services in the area of formal techniques.

13.3.3 Some More Justification

We list some of my publications. All of these reflect the kind of contributions that are being claimed for this thesis. Some of the publications have, in edited form, found their way into [87–89]:

- [25,38]: Chap. 16 of Vol. 2 is a rewrite of this 1977 paper (and 1982 chapter).
- [39]: Chap. 27 of Vol. 3 is a rewrite of this 1981–1982 paper.
- [57,65,85]: Chap. 26 of Vol. 3 is a rewrite of these (basically) 1997–1998 papers.
- [58,114]: Chap. 28 of Vol. 3 is a rewrite of this 1996–1997 paper.

Some papers should be referred to as further examples of domain models:

- [74]: Provides an example of the domain of ‘The Market’ of consumers, retailers, wholesaler and produces. [74] ought be extended with a model of the ‘Supply Chain’.
- [35,37,115,116]: Provide examples of models of data base models (network, hierarchical and relational). These papers ought have been rewritten into a small textbook on databases. We find that the very popular [292] may be acceptable for a college, but not for an academic university course on databases — and would prefer to see a textbook alternating between carefully narrated and annotated formalisations (of database models) and a few of the kind of example otherwise given in [292]. Now the meaning of a relational database query can be rigorously (even formally) ‘derived’ from the formalisation.
- [68,77,79,82–84,91,101,105,106,256,257,289]: Provide (overlapping) models of one or another aspect of transportation systems, specifically railway systems. Ought be rewritten into a monograph on ‘Transportation Systems’ (See also: <http://www.railwaydomain.org/>).
- [54]: Provides an example of a domain model for robots (not robotics). More research on software structures for robotics, such as suggested in internal reports, ought be based, I strongly think, on models like [54].
- [96]: Provides a partial model of a container line industry. My hope is to somehow continue this work.

⁸<http://www.fortia.org/twiki/bin/view/Main/ForTIA>

13.3.4 Some of My Publications

1. [23] H. Bekič, D. Bjørner, W. Henhagl, C.B. Jones, P. Lucas: A Formal Definition of a PL/I Subset. Technical Report 25.139, Vienna, Austria (1974)
2. [25] D. Bjørner: Programming Languages: Formal Development of Interpreters and Compilers. In: *International Computing Symposium 77* (North-Holland Publ.Co., Amsterdam, 1977) pp 1–21
3. [26] D. Bjørner: Programming Languages: Linguistics and Semantics. In: *International Computing Symposium 77* (North-Holland Publ.Co., Amsterdam, 1977) pp 511–536
4. [27] D. Bjørner: Programming in the Meta-Language: A Tutorial. In: *The Vienna Development Method: The Meta-Language, [111]*, ed by D. Bjørner, C.B. Jones (Springer-Verlag, 1978) pp 24–217
5. [28] D. Bjørner: Software Abstraction Principles: Tutorial Examples of an Operating System Command Language Specification and a PL/I-like On-Condition Language Definition. In: *The Vienna Development Method: The Meta-Language, [111]*, ed by D. Bjørner, C.B. Jones (Springer-Verlag, 1978) pp 337–374
6. [29] D. Bjørner: The Systematic Development of a Compiling Algorithm. In: *Le Point sur la Compilation*, ed by Amirchahy, Neel (INRIA Publ. Paris, 1979) pp 45–88
7. [30] D. Bjørner: The Vienna Development Method: Software Abstraction and Program Synthesis. In: *Mathematical Studies of Information Processing*, vol 75 of LNCS (Springer-Verlag, 1979)
8. [31] Edited by D. Bjørner: *Abstract Software Specifications*, vol 86 of LNCS (Springer-Verlag, 1980)
9. [32] D. Bjørner: Application of Formal Models. In: *Data Bases* (INFOTECH Proceedings, 1980)
10. [33] D. Bjørner: Experiments in Block-Structured GOTO-Modelling: Exits vs. Continuations. In: *Abstract Software Specification, [31]*, vol 86 of LNCS, ed by D. Bjørner (Springer-Verlag, 1980) pp 216–247
11. [34] D. Bjørner: Formal Description of Programming Concepts: a Software Engineering Viewpoint. In: *MFCS'80, Lecture Notes Vol. 88* (Springer-Verlag, 1980) pp 1–21
12. [35] D. Bjørner: Formalization of Data Base Models. In: *Abstract Software Specification, [31]*, vol 86 of LNCS, ed by D. Bjørner (Springer-Verlag, 1980) pp 144–215
13. [36] D. Bjørner: The VDM Principles of Software Specification and Program Design. In: *TC2 Work.Conf. on Formalisation of Programming Concepts, Peniscola, Spain* (Springer-Verlag, LNCS Vol. 107 1981) pp 44–74
14. [37] D. Bjørner: Realization of Database Management Systems. In: *See [112]* (Prentice-Hall, 1982) pp 443–456
15. [38] D. Bjørner: Rigorous Development of Interpreters and Compilers. In: *See [112]* (Prentice-Hall, 1982) pp 271–320
16. [39] D. Bjørner: Stepwise Transformation of Software Architectures. In: *See [112]* (Prentice-Hall, 1982) pp 353–378
17. [40] D. Bjørner: Software Architectures and Programming Systems Design. Vols. I-VI. Techn. Univ. of Denmark (1983-1987)
18. [41] D. Bjørner: Project Graphs and Meta-Programs: Towards a Theory of Software Development. In: *Proc. Capri '86 Conf. on Innovative Software Factories and Ada, Lecture Notes on*

- Computer Science*, ed by N. Habermann, U. Montanari (Springer-Verlag, 1986)
19. [42] D. Bjørner: Software Development Graphs — A Unifying Concept for Software Development? In: *Vol. 241 of Lecture Notes in Computer Science: Foundations of Software Technology and Theoretical Computer Science*, ed by K. Nori (Springer-Verlag, 1986) pp 1–9
 20. [43] D. Bjørner: *Software Engineering and Programming: Past-Present-Future*. IPSJ: Inform. Proc. Soc. of Japan **8**, 4 (1986) pp 265–270
 21. [44] D. Bjørner: On The Use of Formal Methods in Software Development. In: *Proc. of 9th International Conf. on Software Engineering, Monterey, California* (1987) pp 17–29
 22. [45] D. Bjørner: The Stepwise Development of Software Development Graphs: Meta-Programming VDM Developments. In: *See [113]*, vol 252 of *LNCS* (Springer-Verlag, Heidelberg, Germany, 1987) pp 77–96
 23. [46] D. Bjørner: *Facets of Software Development: Computer Science & Programming, Engineering & Management*. J. of Comput. Sci. & Techn. **4**, 3 (1989) pp 193–203
 24. [47] D. Bjørner: Specification and Transformation: Methodology Aspects of the Vienna Development Method. In: *TAPSOFT'89*, vol 352 of *Lab. Note* (Springer-Verlag, Heidelberg, Germany, 1989) pp 1–35
 25. [48] D. Bjørner: Formal Software Development: Requirements for a CASE. In: *European Symposium on Software Development Environment and CASE Technology, Königswinter, FRG, June 17–21* (Springer-Verlag, Heidelberg, Germany, 1991)
 26. [49] D. Bjørner: Formal Specification is an Experimental Science (in English). In: *Intl. Conf. on Perspectives of System Informatics* (1991)
 27. [50] D. Bjørner: *Formal Specification is an Experimental Science (in Russian)*. *Programmirovaniye* **6** (1991) pp 24–43
 28. [51] D. Bjørner: Towards a Meaning of 'M' in VDM. In: *Formal Description of Programming Concepts*, ed by E. Neuhold, M. Paul (Springer-Verlag, Heidelberg, Germany, 1991) pp 137–258
 29. [52] D. Bjørner: From Research to Practice: Self-reliance of the Developing World through Software Technology: Usage, Education & Training, Development & Research. In: *Information Processing '92, IFIP World Congress '92, Madrid*, ed by J. van Leeuwen (IFIP Transaction A-12: Algorithms, Software, Architecture, 1992) pp 65–71
 30. [53] D. Bjørner: Trustworthy Computing Systems: The ProCoS Experience. In: *14'th ICSE: Intl. Conf. on Software Eng., Melbourne, Australia* (ACM Press, 1992) pp 15–34
 31. [54] D. Bjørner. *Formal Models of Robots: Geometry & Kinematics*, chapter 3, pages 37–58. Prentice-Hall International, January 1994. Eds.: W. Roscoe and J. Woodcock, *A Classical Mind*, Festschrift for C.A.R. Hoare.
 32. [55] D. Bjørner: Prospects for a Viable Software Industry — Enterprise Models, Design Calculi, and Reusable Modules. In: *First ACM Japan Chapter Conference* (World Scientific Publ, Singapore 1994)
 33. [56] D. Bjørner: Software Systems Engineering — From Domain Analysis to Requirements Capture: An Air Traffic Control Example. In: *2nd Asia-Pacific Software Engineering Conference (APSEC '95)* (IEEE Computer Society, 1995)
 34. [57] D. Bjørner: From Domain Engineering via Requirements to Software. Formal Specification and Design Calculi. In: *SOFSEM'97*, vol 1338 of

- Lecture Notes in Computer Science* (Springer-Verlag, 1997) pp 219–248
35. [58] D. Bjørner: Michael Jackson's Problem Frames: Domains, Requirements and Design. In: *ICFEM'97: International Conference on Formal Engineering Methods*, ed by L. ShaoYang, M. Hinchley (1997)
 36. [59] D. Bjørner: Challenges in Domain Modelling — Algebraic or Otherwise. Research, Department of Information Technology, Software Systems Section, Technical University of Denmark, DK-2800 Lyngby, Denmark (1998)
 37. [60] D. Bjørner: Domains as Prerequisites for Requirements and Software &c. In: *RTSE'97: Requirements Targeted Software and Systems Engineering*, vol 1526 of *Lecture Notes in Computer Science*, ed by M. Broy, B. Rumpe (Springer-Verlag, Berlin Heidelberg 1998) pp 1–41
 38. [61] D. Bjørner: Formal Methods in the 21st Century — An Assessment of Today, Predictions for The Future — Panel position presented at the ICSE'98, Kyoto, Japan. Technical Report, Department of Information Technology, Software Systems Section, Technical University of Denmark (1998)
 39. [62] D. Bjørner: Issues in International Cooperative Research — Why not Asian, African or Latin American 'Esprits' ? Research, Department of Information Technology, Software Systems Section, Technical University of Denmark, DK-2800 Lyngby, Denmark (1998)
 40. [63] D. Bjørner: A Triptych Software Development Paradigm: Domain, Requirements and Software. Towards a Model Development of A Decision Support System for Sustainable Development. In: *Festschrift to Hans Langmaack: Correct Systems Design: Recent Insight and Advances*, vol 1710 of *Lecture Notes in Computer Science*, ed by E.-R. Olderog, B. Steffen (Springer-Verlag, 1999) pp 29–60
 41. [64] D. Bjørner: Challenge '2000: some aspects of: "How to Create a Software Industry". In: *Proceedings of CSIC'99*, Ed.: R. Jalili (1999)
 42. [65] D. Bjørner: *Where do Software Architectures come from ? Systematic Development from Domains and Requirements. A Re-assessment of Software Engineering ?* South African Journal of Computer Science **22** (1999) pp 3–13
 43. [66] D. Bjørner: Domain Engineering, A Software Engineering Discipline in Need of Research. In: *SOFSEM'2000: Theory and Practice of Informatics*, vol 1963 of *Lecture Notes in Computer Science* (Springer Verlag, Milovy, Czech Republic 2000) pp 1–17
 44. [67] D. Bjørner: Domain Modelling: Resource Management Strategies, Tactics & Operations, Decision Support and Algorithmic Software. In: *Millenial Perspectives in Computer Science*, ed by J. Davies, B. Roscoe, J. Woodcock (Palgrave (St. Martin's Press), Houndmills, Basingstoke, Hampshire, RG21 6XS, UK 2000) pp 23–40
 45. [68] D. Bjørner: Formal Software Techniques in Railway Systems. In: *9th IFAC Symposium on Control in Transportation Systems*, ed by E. Schnieder (2000) pp 1–12
 46. [69] D. Bjørner: Informatics: A Truly Interdisciplinary Science — Computing Science and Mathematics. In: *9th Intl. Colloquium on Numerical Analysis and Computer Science with Applications*, ed by D. Bainov (Academic Publications, P.O.Box 45, BG-1504 Sofia, Bulgaria 2000)
 47. [70] D. Bjørner: Informatics: A Truly Interdisciplinary Science — Prospects for an Emerging World. In: *Information Technology and Communication — at the Dawn of the New Millenium*,

- ed by S. Balasubramanian (2000) pp 71–84
48. [71] D. Bjørner: *Pinnacles of Software Engineering: 25 Years of Formal Methods*. Annals of Software Engineering **10** (2000) pp 11–66
 49. [72] D. Bjørner: Informatics Models of Infrastructure Domains. In: *Computer Science and Information Technologies* (Institute for Informatics and Automation Problems, Yerevan, Armenia 2001) pp 13–73
 50. [73] D. Bjørner: On Formal Techniques in Protocol Engineering: Example Challenges. In: *Formal Techniques for Networks and Distributed Systems* (Eds.: Myungchul Kim, Byoungmoon Chin, Sungwon Kang and Danhyung Lee) (Kluwer, 2001) pp 395–420
 51. [74] D. Bjørner: Domain Models of “The Market” — in Preparation for E-Transaction Systems. In: *Practical Foundations of Business and System Specifications* (Eds.: Haim Kilov and Ken Baclawski) (Kluwer Academic Press, The Netherlands 2002)
 52. [75] D. Bjørner: Some Thoughts on Teaching Software Engineering – Central Rôles of Semantics. In: *Liber Amicorum: Professor Jaco de Bakker* (Stichting Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands 2002) pp 27–45
 53. [76] D. Bjørner: Domain Engineering: A “Radical Innovation” for Systems and Software Engineering ? In: *Verification: Theory and Practice*, vol 2772 of *Lecture Notes in Computer Science* (Springer-Verlag, Heidelberg 2003)
 54. [77] D. Bjørner: Dynamics of Railway Nets: On an Interface between Automatic Control and Software Engineering. In: *CTS2003: 10th IFAC Symposium on Control in Transportation Systems* (Elsevier Science Ltd., Oxford, UK 2003)
 55. [78] D. Bjørner: Logics of Formal Software Specification Languages — The Possible Worlds cum Domain Problem. In: *Fourth Pan-Hellenic Symposium on Logic*, ed by L. Kirousis (2003)
 56. [79] D. Bjørner: New Results and Trends in Formal Techniques for the Development of Software for Transportation Systems. In: *FORMS2003: Symposium on Formal Methods for Railway Operation and Control Systems* (Institut für Verkehrssicherheit und Automatisierungstechnik, Techn.Univ. of Braunschweig, Germany, 2003)
 57. [80] D. Bjørner. “What is a Method ?” — *An Essay of Some Aspects of Software Engineering*, chapter 9, pages 175–203. Monographs in Computer Science. IFIP: International Federation for Information Processing. Springer Verlag, New York, N.Y., USA, 2003. Programming Methodology: Recent Work by Members of IFIP Working Group 2.3. Eds.: Annabelle McIver and Carrol Morgan.
 58. [81] D. Bjørner: What is an Infrastructure ? In: *Formal Methods at the Crossroads. From Panacea to Foundational Support* (Springer-Verlag, Heidelberg, Germany 2003)
 59. [82] D. Bjørner: The Grand Challenge – FAQs of the R&D of a Railway Domain Theory. In: *IFIP World Computer Congress, Topical Days: TRain: The Railway Domain* (Kluwer Academic Press, Amsterdam, The Netherlands 2004)
 60. [83] D. Bjørner: The TRain Topical Day. In: *Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions, 22–27 August, 2004, Toulouse, France* — Ed. René Jacquart (Kluwer Academic Publishers, 2004) pp 607–611
 61. [84] D. Bjørner: Towards a Formal Model of CyberRail. In: *Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions, 22–27 August, 2004, Toulouse, France*

- Ed. *René Jacquart* (Kluwer Academic Publishers, 2004) pp 657–664
62. [85] D. Bjørner: Towards “Posite & Prove” Design Calculi for Requirements Engineering and Software Design. In: *Essays and Papers in Memory of Ole-Johan Dahl* (Springer-Verlag, 2004)
 63. [90] D. Bjørner: Domain Theory: Practice and Theories, Discussion of Possible Research Topics. In: *ICTAC’2007*, vol 4701 of *Lecture Notes in Computer Science* (eds. J.C.P. Woodcock et al.) (Springer, Heidelberg 2007) pp 1–17
 64. [91] D. Bjørner: Transportation Systems Development. In: *2007 ISoLA Workshop On Leveraging Applications of Formal Methods, Verification and Validation; Special Workshop Theme: Formal Methods in Avionics, Space and Transport* (2007)
 65. [93] D. Bjørner: *Believable Software Management*. Encyclopedia of Software Engineering **1**, 1 (2008) pp 1–32
 66. [94] D. Bjørner: Domain Engineering. In: *BCS FACS Seminars* (Springer, London, UK 2008) pp 1–42
 67. [95] D. Bjørner: From Domains to Requirements. In: *Montanari Festschrift*, vol 5065 of *Lecture Notes in Computer Science* (eds. Pierpaolo Degano, Rocco De Nicola and José Meseguer) (Springer, Heidelberg 2008) pp 1–30
 68. [96] D. Bjørner: Domain Engineering. In: *The 2007 Lipari PhD Summer School* (Springer, Heidelberg, Germany 2009) pp 1–102
 69. [98] D. Bjørner: *Domain Engineering: “Upstream” from Requirements Engineering and Software Design*. US ONR + Univ. of Genoa Workshop, Santa Margherita Ligure (June 2000)
 70. [100] D. Bjørner, J.R. Cuéllar: *Software Engineering Education: Roles of Formal Specification and Design Calculi*. Annals of Software Engineering **6** (1998) pp 365–410
 71. [101] D. Bjørner, Y.L. Dong, S. Prehn: Domain Analyses: A Case Study of Station Management. In: *KICS’94: Kunming International CASE Symposium, Yunnan Province, P.R. of China* (1994)
 72. [102] D. Bjørner, L.M. Druffel: Industrial Experience in using Formal Methods. In: *Intl. Conf. on Software Engineering* (IEEE Computer Society Press, 1990) pp 264–266
 73. [103] D. Bjørner, A. Eir: Compositionality: Ontology and Mereology of Domains. Some Clarifying Observations in the Context of Software Engineering. In: *Festschrift for Prof. Willem Paul de Roever* (Eds. Martin Steffen, Dennis Dams and Ulrich Hannemann), vol [not known at time of submission of the current paper] of *Lecture Notes in Computer Science* (eds. Martin Steffen, Dennis Dams and Ulrich Hannemann) (Springer, Heidelberg 2008) pp 1–12
 74. [104] D. Bjørner, C.W. George, A.E. Haxthausen et al: “UML”-ising Formal Techniques. In: *INT 2004: Third International Workshop on Integration of Specification Techniques for Applications in Engineering*, vol 3147 of *Lecture Notes in Computer Science* (Springer-Verlag, 2004, ETAPS, Barcelona, Spain) pp 423–450
 75. [105] D. Bjørner, C.W. George, S. Prehn. *Scheduling and Rescheduling of Trains*, chapter 8, pages 157–184. *Industrial Strength Formal Methods in Practice*, Eds.: Michael G. Hinchey and Jonathan P. Bowen. FACIT, Springer-Verlag, London, England, 1999.
 76. [106] D. Bjørner, C.W. George, S. Prehn: Computing Systems for Railways — A Rôle for Domain Engineering. Relations to Requirements Engineering and Software for Control Applications. In: *Integrated Design and Process Technology*. Editors: Bernd Kraemer and John C. Petterson (Society for Design and Process Science,

- P.O.Box 1299, Grand View, Texas 76050-1299, USA 2002)
77. [107] D. Bjørner, A.E. Haxthausen, K. Havelund: *Formal, Model-oriented Software Development Methods: From VDM to ProCoS, and from RAISE to LaCoS*. Future Generation Computer Systems (1992)
 78. [111] Edited by D. Bjørner, C.B. Jones: *The Vienna Development Method: The Meta-Language*, vol 61 of LNCS (Springer-Verlag, 1978)
 79. [112] Edited by D. Bjørner, C.B. Jones: *Formal Specification and Software Development* (Prentice-Hall, 1982)
 80. [113] D. Bjørner, C.B. Jones, M.M. an Airchinnigh, E.J. Neuhold, editors. *VDM – A Formal Method at Work*. Proc. VDM-Europe Symposium 1987, Brussels, Belgium, Springer-Verlag, Lecture Notes in Computer Science, Vol. 252, March 1987.
 81. [114] D. Bjørner, S. Koussobe, R. Noussi, G. Satchok: Michael Jackson’s Problem Frames: Towards Methodological Principles of Selecting and Applying Formal Software Development Techniques and Tools. In: *ICFEM’97: Intl. Conf. on “Formal Engineering Methods”, Hiroshima, Japan*, ed by L. ShaoQi, M. Hinchley (IEEE Computer Society Press, Los Alamitos, CA, USA 1997) pp 263–271
 82. [115] D. Bjørner, H.H. Løvengreen: Formal Semantics of Data Bases. In: *8th Int’l. Very Large Data Base Conf.* (1982)
 83. [116] D. Bjørner, H.H. Løvengreen: Formalization of Data Models. In: *Formal Specification and Software Development, [112]* (Prentice-Hall, 1982) pp 379–442
 84. [117] D. Bjørner, M. Nielsen: Meta Programs and Project Graphs. In: *ETW: Esprit Technical Week* (Elsevier, 1985) pp 479–491
 85. [118] D. Bjørner, J.F. Nilsson: Algorithmic & Knowledge Based Methods — Do they “Unify” ? — with some Programme Remarks for UNU/IIST. In: *International Conference on Fifth Generation Computer Systems: FGCS’92* (ICOT, 1992) pp (Separate folder, “191–198”)
 86. [119] D. Bjørner, O.N. Oest: The DDC Ada Compiler Development Project. In: *Towards a Formal Description of Ada, [121]*, vol 98 of LNCS, ed by D. Bjørner, O.N. Oest (Springer-Verlag, 1980) pp 1–19
 87. [121] Edited by D. Bjørner, O.N. Oest: *Towards a Formal Description of Ada*, vol 98 of LNCS (Springer-Verlag, 1980)
 88. [122] D. Bjørner, S. Prehn: Software Engineering Aspects of VDM. In: *Theory and Practice of Software Technology*, ed by D. Ferrari (North-Holland Publ.Co., Amsterdam, 1983)
 89. [159] P. Folkjær, D. Bjørner: A Formal Model of a Generalised CSP-like Language. In: *Proc. IFIP’80*, ed by S. Lavington (North-Holland Publ.Co., Amsterdam, 1980) pp 95–99
 90. [177] P. Hall, D. Bjørner, Z. Mikolajuk: Decision Support Systems for Sustainable Development: Experience and Potential — a Position Paper. Administrative Report 80, UNU/IIST, P.O.Box 3058, Macau (1996)
 91. [223] H.H. Løvengreen, D. Bjørner: On a Formal Model of the Tasking Concepts in Ada. In: *ACM SIGPLAN Ada Symp.* (1980)
 92. [256] M. Pěnička, D. Bjørner: From Railway Resource Planning to Train Operation — a Brief Survey of Complementary Formalisations. In: *Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions, 22–27 August, 2004, Toulouse, France — Ed. Ren  ne Jacquart* (Kluwer Academic Publishers, 2004) pp 629–636
 93. [257] M. Pěnička, A.K. Strupchanska, D. Bjørner: Train Maintenance Routing. In: *FORMS’2003: Symposium on Formal Methods for Rail-*

- way *Operation and Control Systems* (L'Harmattan Hongrie, 2003)
94. [289] A.K. Strupchanska, M. Pěnička, D. Bjørner: *Railway Staff Rostering*. In: *FORMS2003: Symposium on Formal Methods for Railway Operation and Control Systems* (L'Harmattan Hongrie, 2003)

13.4 An Unending Quest

The sub-title of this thesis is: *An Unended Quest*.

This thesis summarises 35 years of research, from when I worked with fine colleagues (notably with the late Hans Bekič, and with Cliff Jones and Peter Lucas in the early 1970s at the IBM Wiener Labor), till today.

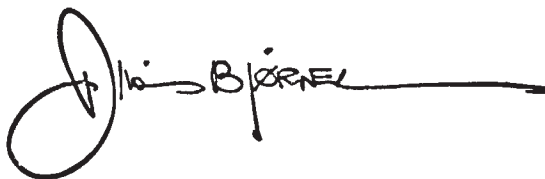
On Page 30 in a Preface section (The Author's Aspirations) of Vol. 1 [87], I wrote: "*I have written these volumes because I wanted to understand how to develop large-scale software systems. When I started, some 25 years ago, writing lecture notes on this subject, I knew less than I do now.*"

I was fortunate, at IBM and with DDC (Dansk Datamatik Center), to be able to see much of what I knew and learned be carried out in "real life",

I am sure I will learn more as I continue toiling with the issues — such as I have learned when working on Chaps. 1–12 and Appendices A–G. Right now, August 1, 2008, I am "halfway" through the writing of a new book [97]!

• • •

The quest goes on.



Fredsvej 11, Holte, Denmark; August 1, 2008

DR. TECHN. APPENDICES

Appendices A–E (Pages 123–290) are to be considered part of the thesis as submitted. They all represent published, or to be published papers.

A

Domain Engineering

This appendix chapter constitutes the invited chapter [94] for a collection of BCS FACS papers to be published by Springer (UK).

Before software can be designed we must know its requirements. Before requirements can be expressed we must understand the domain. So it follows, from our dogma, that we must first establish precise descriptions of domains; then from such, “derive” at least domain requirements; and from those and other (interface and machine) requirements design the software, or, more generally, the computing systems. The preceding was an engineering dogma. Now follows a science dogma: *Just as physicists have studied this universe for centuries (and more), and will continue to do so for centuries (or more), so it is about time that we also study such man-made universes as air traffic, the financial service industry, healthcare, manufacturing, “the market”, railways, indeed transportation in general, and so forth. Just in-and-by-themselves. No need to excuse such a study by stating only engineering concerns. To understand is all. And helps engineering.*

In the main part of this chapter we shall outline what goes into a domain description, not so much how we acquire what goes in. That is: before we can acquire domain “knowledge” we must know what are suitable structures of domain descriptions. Thus we shall outline ideas of modelling the intrinsics (of a domain), the support technologies (of ...), the management and organisation (of ...), the rules and regulations (including [license or contract] scripts) (of ...), and the human behaviours (of a domain).

Before delving into the main part we shall, however first overview what we see as basic principles of describing phenomena and concepts of domains.

At the basis of all descriptive, prescriptive and specificational modeling work is abstraction. Mathematics is a reliable carrier of abstraction. Hence our domain modeling will be presented both as informal, yet short and precise, that is, concise narratives as well as formally. In this chapter we primarily express the formalisations in the RAISE [168] specification language, RSL [166]. We refer to [87,88] for a comprehensive coverage of formal abstractions and models.

Two remarks are now in order. Firstly, there are other specification (cum development or design) languages, Alloy [202], ASM [261, 262], B [1, 131], CafeOBJ [147, 148, 161, 162], CASL [24, 142, 241, 242], VDM-SL [111, 112, 157, 158] and Z [186, 187, 283, 284, 297]. But, secondly, none of these suffices. Each, including RSL, have their limitations in what they were supposed to express with ease. So one needs to combine, to integrate either of the above formal notations with for example the notations of Duration Calculus [301, 302] (DC), Message [199–201] or Live Sequence Charts (MSCs and LSCs) [145, 184, 212], Petri Nets [210, 250, 258–260], Statecharts [180–183, 185] (SCs), TLA+ [217, 218, 232] etcetera.

Chapters 12–15 of [88] presents an extensive coverage of Petri Nets, MSCs and LSCs, SCs and DCs, respectively. This chapter presents an essence of chapters 5, 6 and 11 of [89].

The forthcoming book “Logics of Specification Languages” [108] covers the following formal notations ASM, B, CafeOBJ, CASL, DC, RAISE, VDM-SL, TLA+ and Z. [108] represents an extensive revision of the following published papers: [130, 148, 167, 187, 232, 242, 261].

A.1 Introduction

A.1.1 Application cum Business Domains

By domain we shall loosely speaking understand the same as by application (or business) domain: a universe of discourse, an area of human and societal activity for which, eventually some support in the form of computing and (electronic) communication (c&c) may be desired. Once such c&c, that is, hardware and software has been installed the environment for that machine is the former domain, and the new domain includes the machine, i.e., the hardware and software c&c. The machine interacts with its new domain (with its environment). But we can speak of a domain without ever thinking, or having to think about c&c applications.

Examples of domains are: air traffic, airports, the financial service industry (clients, banks, brokers, securities exchange, portfolio managers, insurance companies, credit card companies, financial service industry “watchdogs”, etc.), freight logistics, health care, manufacturing, and transportation (air lines, railways, roads (private automobiles, busses, taxis, trucking), shipping). These examples could be said to be “grand scale”, and to reflect infrastructure components of a society. Less “grand” examples of domains are: the interconnect cabling of and the electronic and electro-mechanical boxes of either a sound & noise measuring test set-up or a so-called “intelligent” home, the interlocking of groups of rail points (switches) of a railway station, an automobile (with all its mechanical, electro-mechanical and electronic parts and the composition of all these parts, and with the driver and zero, one or more

passengers), or a set or laws (or just rules and regulations) of business enterprise (the domain would then, in this latter cases, include abstractions, relevant to the laws (etc.) of whatever these laws or rules and regulations referred to). In all of these latter cases we usually include the human or technological monitoring and control of these domains.

A.1.2 Physics, Domains and Engineering

Physics, “mother nature” has been studied for millenia. Physicists continue to unravel deeper and deeper understandings of the physically observable universe around us. Classical engineering builds on physics. Every aeronautical & aerospace, chemical, civil, electrical & electronics, and mechanical & control engineer is expected to know all the laws of physics relevant to their field, and much more, and is expected to model, using various branches of mathematics (calculus, statistics, probability theory, graph theory, combinatorics, etc.), phenomena of the domain in which their engineering artifacts are placed (as well as, of course, these artifacts themselves).

Software engineers sometimes know how to model their own artifacts (compilers, operating systems, database management systems, data communication systems, web services, etc.), but they seldom, if ever are expected to model, and they mostly cannot, i.e., do not know how to model the domain in which their software operates.

A.2 Domain Engineering: The Engineering Dogma

- *Before software can be designed we must know its requirements.*
- *Before requirements can be expressed we must understand the domain.*
- *So it follows, from our dogma, that we must*
 - ★ *first establish precise descriptions of domains;*
 - ★ *then from such, “derive” at least domain requirements;*
 - ★ *and from those and other (interface and machine) requirements design the software, or, more generally, the computing systems.*

That is, we propose what we have practiced for many years, that the software engineering process be composed — and that it be iterated over, in a carefully monitored and controlled manner — as follows:

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

We see the domain engineering process as composed from, and iterated over:

1. identification of and regular interaction with stakeholders
2. domain (knowledge) acquisition

3. domain analysis
4. domain modelling
5. domain verification
6. domain validation
7. domain theory formation

In this chapter we shall only look at the principles and techniques of domain modeling, that is, item 4. To pursue items 2.–3. one must know what goes into a domain description, i.e., a domain model.

- *A major part of the domain engineering process is taken up by finding and expressing suitable abstractions, that is, descriptions of the domain.*
- *Principles for identifying, classifying and describing domain phenomena and concepts are therefore needed.*

This chapter focuses on presenting some of these principles and techniques.

A.3 Entities, Functions, Events and Behaviours

In the domain we observe phenomena. From usually repeated such observations we form (immediate, abstract) concepts. We may then “lift” such immediate abstract concepts to more general abstract concepts.

Phenomena are manifest. They can be observed by human senses (seen, heard, felt, smelled or tasted) or by physical measuring instruments (mass, length, time, electric current, thermodynamic temperature, amount of substance, luminous intensity). Concepts are defined.

We shall analyse phenomena and concepts according to the following simple, but workable classification: **entities**, **functions** (over entities), **events** (involving changes in entities, possibly as caused by function invocations, i.e., **actions**, and/or possibly causing such), and **behaviours** as (possibly sets of) sequences of actions (i.e., function invocations) and events.

A.3.1 Entities

- *By an **entity** we shall understand something static; although that “thing” may move, after it has moved it is essentially the same thing, an entity*

Entities are either atomic or composite. The decision as to which entities are considered what is a decision solely taken by the describer.

Atomic Entities

- *By an **atomic entity** we intuitively understand an entity which “cannot be taken apart” (into other, the sub-entities).*

Attributes — Types and Values:

With any entity we can associate one or more attributes.

- By an **attribute** we understand a pair of a **type** and a **value**.

Example A.1. Atomic Entities:

Entity: Person		Entity: Bank Account	
Type	Value	Type	Value
Name	Dines Bjørner	number	212 023 361 918
Weight	118 pounds	balance	1,678,123 Yen
Height	179 cm	interest rate	1.5 %
Gender	male	credit limit	400,000 Yen

“Removing” an attribute from an entity destroys its “entity-hood”.

Composite Entities

- By a *composite entity* we intuitively understand an entity (i) which “can be taken apart” into sub-entities, (ii) where the composition of these is described by its **mereology**, and (iii) which further possess one or more attributes.

Example A.2. Transport Net, A Narrative:

Entity: Transport Net		
Subentities: Segments Junctions		
Mereology: “set” of one or more $s(\text{egment})$ s and “set” of two or more $j(\text{unction})$ s such that each $s(\text{egment})$ is delimited by two $j(\text{unctions})$ and such that each $j(\text{unction})$ connects one or more $s(\text{egments})$		
Attributes		
	Types:	Values:
	Multimodal	Rail, Roads
	Transport Net of	Denmark
	Year Surveyed	2006

To put the above example of a composite entity in context we give an example of both an informal narrative and a corresponding formal specification:

Example A.3. Transport Net, A Formalisation: A transport net consists of one or more segments and two or more junctions. With segments [junctions] we can associate the following attributes: segment [junction] identifiers, the identifiers of the two junctions to which segments are connected [the identifiers of the one or more segments connected to the junction], the mode of a segment [the modes of the segments connected to the junction].

```

type
  N, S, J, Si, Ji, M
value
  obs_Ss: N → S-set,   obs_Js: N → J-set
  obs_Si: S → Si,       obs_Ji: J → Ji
  obs_Jis: S → Ji-set,  obs_Sis: J → Si-set
  obs_M: S → M,         obs_Ms: J → M-set
axiom
  ∀ n:N • card obs_Ss(n) ≥ 1 ∧ card obs_Js(n) ≥ 2
  ∀ n:N • card obs_Ss(n) ≡ card {obs_Si(s)|s:S • s ∈ obs_Ss(n)}
  ∀ n:N • card obs_Js(n) ≡ card {obs_Ji(c)|j:J • j ∈ obs_Js(n)}
  ...
type
  Nm, Co, Ye
value
  obs_Nm: N → Nm, obs_Co: N → Co, obs_Ye: N → Ye

```

Si, Ji, M, Nm, Co, Ye are not entities. They are names of attribute types and, as such, designate attribute values. N is composite, S and J are considered atomic.

-
- By **mereology** we shall understand a theory of part-hood relations. That is, of the relations of part to whole and the relations of part to part within a whole.

The term mereology seems to have been first used in the sense we are using it by the Polish mathematical logician Stanisław Lesniewski [225, 237, 285, 286, 290].

States

- By a domain **state** we shall understand a collection of domain entities chosen by the domain engineer.

The pragmatics of the notion of state is that states are recurrent arguments to functions and are changed by function invocations.

A.3.2 Functions

- By a **function** we shall understand something which when applied to some arguments (i.e., entities) yield some entities called the result of the function (application).
- By an **action** we shall understand the same things as applying a state-changing function to its arguments (including the state).

[The observer functions of the formal example above are not the kind of functions we are seeking to identify in the domain. They are mere technicalities: needed, due to the way in which we formalise — and are deployed in order to express sub-entities, mereologies and attributes.]

Function Signatures

By a function signature we mean the name and type of a function.

```

type
  A, B, ..., C, X, Y, ..., Z
value
  f:  $A \times B \times \dots \times C \rightarrow X \times Y \times \dots \times Z$ 

```

The last line above expresses a schematic function signature.

Function Descriptions

By a function description we mean a function signature and something which describes the relationship between function arguments (the a:A's, b:B's, ..., c:C's and the x:X's, y:Y's, ..., z:Z's).

Example A.4. Well Formed Routes:

```

type
  P = Ji × Si × Ji      /* path: triple of identifiers */
  R' = P*                /* route: sequence of connected paths */
  R = { | r:R' • wf_R(r) | } /* subtype of R': those r's satisfying wf_R(r) */
value
  wf_R: R' → Bool
  wf_R(r) ≡
    ∀ i:Nat • {i,i+1} ⊆ inds r ⇒ let (.,ji')=r(i), (ji'',.)=r(i+1) in ji'=ji'' end

```

The last line above describes the route well-formedness predicate. [The meaning of the “(.,” and “.,)” is that the omitted path components “play no rôle.”]

A.3.3 Events

- *By an **event** we shall understand an instantaneous change of state not directly brought about by some explicitly willed action in the domain, but either by “external” forces. or implicitly as a non-intended result of an explicitly willed action.*

Events may or may not lead to the initiation of explicitly issued operations.

Example A.5. Events: A 'withdraw' from a positive balance bank account action may leave a negative balance bank account. A bank branch office may have to temporarily stop actions, i.e., close, due to a bank robbery. •

Internal events: The first example above illustrates an internal action. It was caused by an action in the domain, but was not explicitly the main intention of the "withdraw" function.

External events: The second example above illustrates an external action. We assume that we have not explicitly modelled bank robberies!

Formal modelling of events: With every event we can associate an event label. And event label can be thought of as a simple identifier. Two or more event labels may be the same.

A.3.4 Behaviours

- *By a **behaviour** we shall understand a structure of actions (i.e., function invocations) and events. The structure may typically be a set of sequences of actions and events.*

A behaviour is either a simple behaviour, or is a concurrent behaviour, or, if the latter, can be either a communicating behaviour or not.

- *By a **simple behaviour** we shall understand a sequence of actions and events.*

Example A.6. Simple Behaviours: The opening of a bank account, the deposit into that bank account, zero, one or more other such deposits, a withdrawal from the bank account in question, etc. (deposits and withdrawals), ending with a closing of the bank account. Any prefix of such a sequence is also a simple behaviour. Any sequence in which one or more events are interspersed is also a simple behaviour. •

- *By a **concurrent behaviour** we shall understand a set of behaviours (simple or otherwise).*

Example A.7. Concurrent Behaviours: A set of simple behaviours may result from two or more distinct bank clients, each operating of their own, distinct, that is, non-shared accounts. •

- *By a **communicating behaviour** we shall understand a set of two or more behaviours where otherwise distinct elements (i.e., behaviours) share events.*

The sharing of events can be identified via the event labels.

Example A.8. Communicating Behaviours: To model that two or more clients can share the same bank account one could model the bank account as one behaviour and each client as a distinct behaviour. Let us assume that only one client can open an account and that only one client can close an account. Let us further assume that sharing is brought about by one client, say the one who

opened the account, identifying the sharing clients. Now, in order to make sure that at most one client accesses the shared account at one one time (in any one “smallest” transaction interval) one may model “client access to account” as a pair of events such that during the interval between the first (begin transaction) and the second (end transaction) event no other client can share events with the bank account behaviour. Now the set of behaviours of the bank account and one or more of the client behaviours is an example of a communicating behavior. •

Formal modelling of behaviours: Communicating behaviours, the only really interesting behaviours, can be modelled in a great variety of ways: from set-oriented models in B [1, 131], RSL [87–89, 97, 165, 166, 168], VDM [111, 112, 157, 158], or Z [186, 187, 283, 284, 297], to models using for example CSP [189, 190, 266, 273] (as for example “embedded” in RSL [166]), or, to diagram models using, for example, Petri nets [210, 250, 258–260], message [199–201] or live sequence charts [145, 184, 212], or statecharts [180–183, 185].

A.3.5 Discussion

The main aim of Sect. A.3 is to ensure that we have a clear understanding of the modelling concepts of entities, functions, events and behaviours. To “reduce” the modelling of phenomena and concepts to these four kinds of phenomena and concepts is, of course, debatable. Our point is that it works, that further classification, as is done in for example John F. Sowa’s [282], is not necessary, or rather, is replaced by how we model attributes of for example entities¹ and how we model facets, such as we shall call them. The modelling of facets is the main aim of this chapter.

A.4 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.*

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

¹For such issues as static and dynamic attributes, dimensionality, tangibility, time and space, etc., we refer to Michael A. Jackson’s [206] or Chap. 10 of [89].

A.4.1 Intrinsic

- By domain **intrinsic** 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 intrinsic initially covering at least one specific, hence named, stakeholder view.

Example A.9. Railway Net Intrinsic: We narrate and formalise three railway net intrinsic.

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

Railway Net Intrinsic:

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

- *Potential train passengers:*

```

scheme N0 =
  class
    type
      N, L, S, Sn, Ln
    value
      obs_Ls: N → L-set, obs_Ss: N → S-set
      obs_Ln: L → Ln, obs_Sn: S → Sn
      obs_Sns: L → Sn-set, obs_Lns: S → Ln-set
    axiom
      ...
  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.

- *Actual train passengers:*

```
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 sorts, related observer functions and axioms.

- *Train operating staff:*

```
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_U: (N|L|S) → U-set
      obs_C: U → C-set
      obs_Σ: U → Σ
      obs_Ω: U → Ω
    axiom
      ...
  end
```

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

Different stakeholder perspectives, not only of intrinsics, as here, but of any facet, lead to a number of different models. The name of a phenomenon of one perspective, that is, of one model, may coincide with the name of a “similar” phenomenon of another perspective, that is, of another model, and so on. If

the intention is that the “same” names cover comparable phenomena, then the developer must state the comparison relation.

Example A.10. Comparable Intrinsic: We refer to Example A.9. We claim that the concept of nets, lines and stations in the three models of Example A.9 must relate. The simplest possible relationships are to let the third model be the common “unifier” and to mandate

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

Thus the third model is seen as the definitive model for the stakeholder views initially expressed. •

Example A.11. Intrinsic of Switches: The intrinsic attribute of a rail switch is that it can take on a number of states. A simple switch $(^c Y_c^{c/})$ has three connectors: $\{c, c_l, c_r\}$. c is the connector of the common rail from which one can either “go straight” c_l , or “fork” c_r (Fig. A.1). So we have that a possible state space of such a switch could be ω_{g_s} :

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

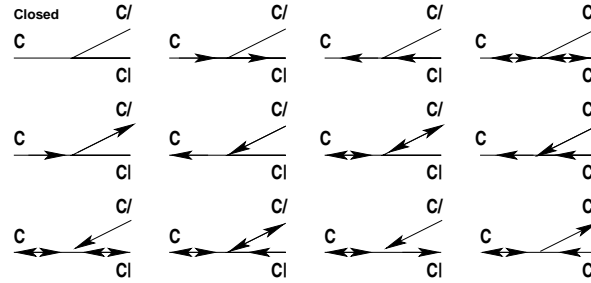


Fig. A.1. Possible states of a rail switch

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

Conceptual Versus Actual Intrinsic

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

On Modelling Intrinsic

Domains can be characterised by intrinsically being entity, or function, or event, or behaviour intensive. Software support for activities in such domains then typically amount to database systems, computation-bound systems, real-time embedded systems, respectively distributed process monitoring and control systems. Modelling the domain intrinsic in respective cases can often be done property-oriented specification languages (like CafeOBJ [147, 148, 161, 162] or CASL [24, 142, 241, 242]), model-oriented specification languages (like B [1, 131], VDM-SL [111, 112, 157, 158], RSL [166], or Z [186, 187, 283, 284, 297]), event-based languages (like Petri nets [210, 250, 258–260] or CSP [189, 190, 266, 273]), respectively process-based specification languages (like MSCs [199–201], LSCs [145, 184, 212], statecharts [180–183, 185], or CSP [189, 190, 266, 273]).

A.4.2 Support Technologies

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

Example A.12. 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 levers³ (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).

²That seemingly complicated domain may seem very complicated, containing hundreds of entities, functions and behaviours. Instead of presenting all the entities, functions, events and behaviours in one “fell swoop”, one presents them in stages: first, around seven such (entities, functions, events and behaviours), then seven more, etc.

(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 A.12 is just a rough sketch. In a proper narrative description the software (cum domain) engineer must describe, in detail, the subsystem of electronics, electro-mechanics and the human operator interface (buttons, lights, sounds, etc.).

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

Example A.13. Probabilistic Rail Switch Unit State Transitions: Figure A.2 indicates a way of formalising this aspect of a supporting technology. Figure A.2 intends to model the probabilistic (erroneous and correct) behaviour of a switch when subjected to settings (to switched (s) state) and resettings (to direct (d) state). A switch may go to the switched state from the direct state when subjected to a switch setting s with probability psd. •

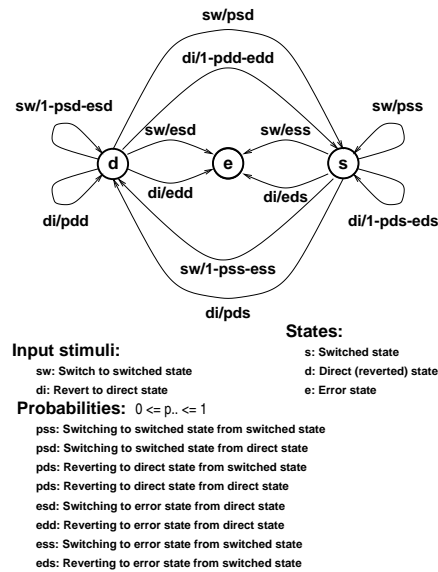


Fig. A.2. Probabilistic state switching

Another example shows another aspect of support technology: Namely that the technology must guarantee certain of its own behaviours, so that software

designed to interface with this technology, together with the technology, meets dependability requirements.

Example A.14. Railway Optical Gates: Train traffic (itf:iTF), intrinsically, is a total function over some time interval, from time (t:T) to continuously positioned (p:P) trains (tn:TN).

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

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

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

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

Railway Optical Gate Technology Requirements:

```

type
  T, TN
  P = U*
  NetTraffic == net:N trf:(TN  $\overrightarrow{m}$  P)
  iTF = T  $\rightarrow$  NetTraffic
  sTF = T  $\overrightarrow{m}$  NetTraffic
  oG = iTF  $\xrightarrow{\sim}$  sTF
value
  [close] c: NetTraffic  $\times$  TN  $\times$  NetTraffic  $\xrightarrow{\sim}$  Bool
axiom
   $\forall$  itt:iTF, og:OG • let stt = og(itt) in
     $\forall$  t:T • t  $\in$  dom stt •
      t  $\in$   $\mathcal{D}$  itt  $\wedge \forall$  Tn:TN • tn  $\in$  dom trf(itt(t))
       $\Rightarrow$  tn  $\in$  dom trf(stt(t))  $\wedge$  c(itt(t),tn,stt(t)) end

```

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

On Modelling Support Technologies

Support technologies in their relation to the domain in which they reside typically reflect real-time embeddedness. As such the techniques and languages for modelling support technologies resemble those for modelling event and process intensity, while temporal notions are brought into focus. Hence typical modelling notations include event-based languages (like Petri nets [210, 250, 258–260] or CSP [189, 190, 266, 273]), respectively process-based specification languages (like MSCs [199–201], LSCs [145, 184, 212], state-charts [180–183, 185], or CSP [189, 190, 266, 273]), as well as temporal languages (like the Duration Calculus [301, 302] and Temporal Logic of Actions, TLA+, [217, 218, 232, 233]).

A.4.3 Management and Organisation

Example A.15. Train Monitoring, I: In China, as an example, rescheduling of trains occurs at stations and involves telephone negotiations with neighbouring stations (“up and down the lines”). Such rescheduling negotiations, by phone, imply reasonably strict management and organisation (M&O). This kind of M&O reflects the geographical layout of the rail net. •

- *By domain **management** we shall understand such people (such decisions) (i) who (which) determine, formulate and thus set standards (cf. rules and regulations, Sect. A.4.4) 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 the structuring of management and non-management staff levels; the allocation of strategic, tactical and operational concerns to within management and non-management staff levels; and hence the “lines of command”: who does what, and who reports to whom, administratively and functionally.*

Example A.16. Railway Management and Organisation: Train Monitoring, II: We single out a rather special case of railway management and organisation. Certain (lowest-level operational and station-located) supervisors are responsible for the day-to-day timely progress of trains within a station and along its incoming and outgoing lines, and according to given timetables. These supervisors and their immediate (middle-level) managers (see below for regional managers) set guidelines (for local station and incoming and outgoing lines) for the monitoring of train traffic, and for controlling trains that are either ahead of or behind their schedules. By an incoming and an outgoing line we mean part of a line between two stations, the remaining part being handled by neighbouring

station management. Once it has been decided, by such a manager, that a train is not following its schedule, based on information monitored by non-management staff, then that manager directs that staff: (i) to suggest a new schedule for the train in question, as well as for possibly affected other trains, (ii) to negotiate the new schedule with appropriate neighbouring stations, until a proper reschedule can be decided upon, by the managers at respective stations, (iii) and to enact that new schedule.⁴ A (middle-level operations) manager for regional traffic, i.e., train traffic involving several stations and lines, resolves possible disputes and conflicts. •

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

Conceptual Analysis, First Part

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

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

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

⁴That 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.

Methodological Consequences

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

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

Conceptual Analysis, Second Part

To relate classical organigrams to formal descriptions we first show such an organigram (Fig. A.3), and then we show schematic processes which — for a rather simple scenario — model managers and the managed!

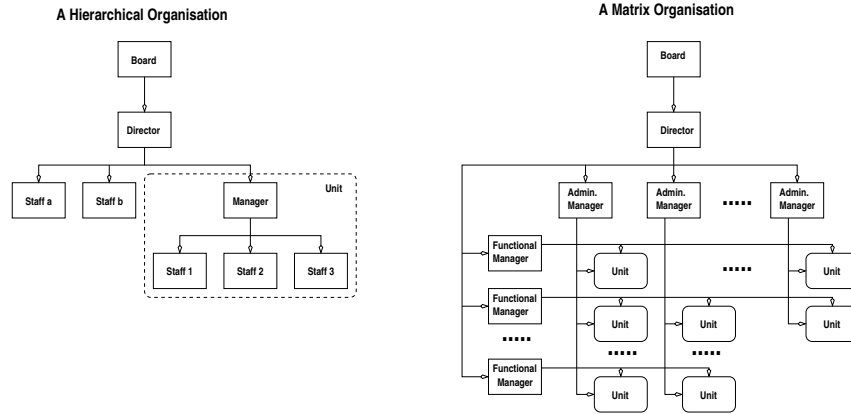


Fig. A.3. Organisational structures

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

Conceptual Model of a Manager-Staff Relation, 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
 $sys() \equiv \parallel \{ st(i)(s\sigma(i)) \mid i:Sx \} \parallel mg(\psi)$

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 nondeterministic internal choice (\parallel).

Conceptual Model of a Manager-Staff Relation, II:

$mg: \Psi \rightarrow \mathbf{in, out} \{ ms[i] \mid i:Sx \} \mathbf{Unit}$
 $mg(\psi) \equiv$
(1) (**let** (ψ', m) = **m_out**(ψ) **in** $\parallel \{ ms[i]!m \mid i:Sx \}; mg(\psi') \mathbf{end}$)
 \parallel
(2) (**let** $\psi' = \parallel \{ \mathbf{let} m = ms[i]? \mathbf{in} m_in(i, m)(\psi) \mathbf{end} \mid i:Sx \} \mathbf{in} mg(\psi') \mathbf{end}$)

 $m_out: \Psi \rightarrow \Psi \times MSG,$
 $m_in: Sx \times MSG \rightarrow \Psi \rightarrow \Psi$

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 nondeterministic internal choice (\parallel).

Conceptual Model of a Manager-Staff Relation, III:

$st: i:Sx \rightarrow \Sigma \rightarrow \mathbf{in, out} \{ ms[i] \} \mathbf{Unit}$
 $st(i)(\sigma) \equiv$
(1) (**let** $m = ms[i]? \mathbf{in} st(i)(st_in(m)(\sigma)) \mathbf{end}$)
 \parallel
(2) (**let** (σ', m) = **st_out**(σ) **in** $ms[i]!m; st(i)(\sigma') \mathbf{end}$)

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

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

The conceptual example also illustrates modelling stakeholder behaviours as interacting (here CSP-like [189, 190, 266, 273]) processes.

On Modelling Management and Organisation

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

A.4.4 Rules and 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 duty, respectively when performing their function.
- By a domain **regulation** we shall understand some text (in the domain) which prescribes what remedial actions are to be taken when it is decided that a rule has not been followed according to its intention.

Example A.17. 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 A.18. Trains Along Lines:

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

A Meta-characterisation of Rules and Regulations

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

A syntactic stimulus, sy_sti , denotes a function, $\text{se_sti:STI}: \Theta \rightarrow \Theta$, 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
 $\text{STI} = \Theta \rightarrow \Theta$
 $\text{RUL} = (\Theta \times \Theta) \rightarrow \mathbf{Bool}$

value

meaning: Stimulus \rightarrow STI
 meaning: Rule \rightarrow RUL

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

valid: Stimulus \times RUL $\rightarrow \Theta \rightarrow \mathbf{Bool}$
 $\text{valid}(\text{sy_sti}, \text{se_rul})(\theta) \equiv \text{se_rul}(\theta, (\text{meaning}(\text{sy_sti}))(\theta))$

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 $\text{Pre_REG} = (\Theta \times \Theta) \rightarrow \mathbf{Bool}$, and a domain configuration-changing function, act_reg:Act_REG , where $\text{Act_REG} = \Theta \rightarrow \Theta$, 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
 $\text{Rul_and_Reg} = \text{Rule} \times \text{Reg}$
 $\text{REG} = \text{Pre_REG} \times \text{Act_REG}$

$$\text{Pre_REG} = \Theta \times \Theta \rightarrow \mathbf{Bool}$$

$$\text{Act_REG} = \Theta \rightarrow \Theta$$

value

interpret: $\text{Reg} \rightarrow \text{REG}$

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

More specifically, there is usually, in any current system configuration, given a set of pairs of rules and regulations. Let $(\text{sy_rul}, \text{sy_reg})$ 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, $\text{pre_reg}(\theta, (\text{meaning}(\text{sy_sti}))(\theta))$, then the action part, act_reg , of the meaning of the regulation, sy_reg , must be applied, $\text{act_reg}(\theta)$, to remedy the situation.

axiom

```

 $\forall (\text{sy\_rul}, \text{sy\_reg}) : \text{Rul\_and\_Regs} \bullet$ 
  let  $\text{se\_rul} = \text{meaning}(\text{sy\_rul}),$ 
     $(\text{pre\_reg}, \text{act\_reg}) = \text{meaning}(\text{sy\_reg})$  in
   $\forall \text{sy\_sti} : \text{Stimulus}, \theta : \Theta \bullet$ 
     $\sim\text{valid}(\text{sy\_sti}, \text{se\_rul})(\theta)$ 
     $\Rightarrow \text{pre\_reg}(\theta, (\text{meaning}(\text{sy\_sti}))(\theta))$ 
     $\Rightarrow \exists n\theta : \Theta \bullet \text{act\_reg}(\theta) = n\theta \wedge \text{se\_rul}(\theta, n\theta)$ 
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

On Modelling Rules and Regulations

Usually rules (as well as regulations) are expressed in terms of domain entities, including those grouped into “the state”, functions, events, and behaviours. Thus the full spectrum of modelling techniques and notations may be needed. Since rules usually express properties one often uses some combination of axioms and well-formedness predicates. Properties sometimes include temporality and hence temporal notations (like Duration Calculus [301, 302] or Temporal Logic of Actions [217, 218, 232, 233]) are used. And since regulations usually express state (restoration) changes one often uses state changing notations (such as found in B [1, 131], RSL [166], VDM-SL [111, 112, 157, 158],

and Z [186,187,283,284,297]). In some cases it may be relevant to model using some constraint satisfaction notation [5] or some Fuzzy Logic notations [264].

A.4.5 Scripts and Licensing Languages

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

Example A.19. A Casually Described Bank Script: We deviate, momentarily, from our line of railway examples, to exemplify one from banking. Our formulation amounts to just a (casual) rough sketch. It is followed by a series of four large examples. Each of these elaborate on the theme of (bank) scripts.

The problem area is that of how repayments of mortgage loans are to be calculated. At any one time a mortgage loan has a balance, a most recent previous date of repayment, an interest rate and a handling fee. When a repayment occurs, then the following calculations shall take place: (i) the interest on the balance of the loan since the most recent repayment, (ii) the handling fee, normally considered fixed, (iii) the effective repayment — being the difference between the repayment and the sum of the interest and the handling fee — and the new balance, being the difference between the old balance and the effective repayment.

We assume repayments to occur from a designated account, say a demand/deposit account. We assume that bank to have designated fee and interest income accounts.

(i) The interest is subtracted from the mortgage holder's demand/deposit account and added to the bank's interest (income) account. (ii) The handling fee is subtracted from the mortgage holder's demand/deposit account and added to the bank's fee (income) account. (iii) The effective repayment is subtracted from the mortgage holder's demand/deposit account and also from the mortgage balance. Finally, one must also describe deviations such as overdue repayments, too large, or too small repayments, and so on. •

Example A.20. 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.

type

C, A, M

$AY' = \mathbf{Real}, AY = \{ | ay:AY' \bullet 0 < ay \leq 10 | \}$

$MI' = \mathbf{Real}, MI = \{ | mi:MI' \bullet 0 < mi \leq 10 | \}$

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

```

Bank = { |  $\beta$ :Bank' • wf.Bank( $\beta$ ) | }
A_Register = C  $\xrightarrow{m}$  A-set
Accounts = A  $\xrightarrow{m}$  Balance
M_Register = C  $\xrightarrow{m}$  M-set
Loans = M  $\xrightarrow{m}$  (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
  ai < mi

```

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

```

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

```

And then the semantics:

```

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

```


•

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

Licensing Languages

A special form of scripts are increasingly appearing in some domains, notably the domain of electronic, or digital media, where these licenses express that the licensor permits the licensee to render (i.e., play) works of proprietary nature CD ROM-like music, DVD-like movies, ec. while obligating the licensee to pay the licensor on behalf of the owners of these, usually artistic works. We refer to [7, 123, 136, 172, 254, 271, 299] for papers and reports on license languages.

On Modelling Scripts

Scripts (as are licenses) are like programs (respectively like prescriptions for program executions). Hence the full variety of techniques and notations for modelling programming (or specification) languages apply [146, 173, 263, 272, 291, 295]. Chapters 6–9 of Vol. 2 of [87–89] cover pragmatics, semantics and syntax techniques for defining languages.

A.4.6 Human Behaviour

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

Example A.21. Banking — or Programming — Staff Behaviour: Let us assume a bank clerk, “in ye olde” days, when calculating, say mortgage repayments (cf. Example A.19).

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 A.20).

We would characterise the programmer as being *diligent* if that person carefully follows the mortgage calculation rules, and throughout the development verifies and tests that the calculations are correct with respect to the rules. We would characterise the programmer as being *sloppy* if that person forgets certain checks and tests when otherwise correcting the computing program under development. We would characterise the programmer as being *delinquent* if that person systematically forgets these checks and tests. And we would characterise the programmer as being a *criminal* if that person intentionally provides a program which miscalculates the mortgage interest, etc., in such a way that the bank (and/or the mortgage client) is cheated out of funds. •

Example A.22. A Human Behaviour Mortgage Calculation:

Example A.20 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'))(\rho,\alpha,\mu,\ell) \equiv
  let (b,d) = \ell(m) in
    if q(\alpha(a),p) /* \alpha(a) \leq p \vee \alpha(a)=p \vee \alpha(a) \leq p \vee ... */
    then
      let i = f_1(interest(mi,b,d'-d)),
          \ell' = \ell \dagger [m \mapsto f_2(\ell(m)-(p-i))]
          \alpha' = \alpha \dagger [a \mapsto f_3(\alpha(a)-p), a_i \mapsto f_4(\alpha(a_i)+i),
                        a "staff" \mapsto f_"staff"(\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 m \in \mu(c)

q: P \times P \rightsquigarrow \text{Bool}
f_1,f_2,f_3,f_4,f_"staff": P \rightsquigarrow P

```

The predicate q and the functions f_1, f_2, f_3, f_4 and $f_{\text{"staff"}}$ of Example A.22 are deliberately left undefined. They are being defined by the “staffer” when performing (incl., programming) the mortgage calculation routine.

The point of Example A.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 A.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 rôle. If we cannot, then we cannot claim that some requirements aim at countering undesirable human behaviour.

A Meta-characterisation of Human Behaviour

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

Our final specification pattern is therefore:

type

Action = $\Theta \rightsquigarrow \Theta$ -infset

value

hum_int: Rule $\rightarrow \Theta \rightarrow \text{RUL-infset}$

action: Stimulus $\rightarrow \Theta \rightarrow \Theta$

hum_beha: Stimulus \times Rules \rightarrow Action $\rightarrow \Theta \rightsquigarrow \Theta$ -infset

hum_beha(sy_sti, sy_rul)(α)(θ) as θ_{set}

post

$\theta_{\text{set}} = \alpha(\theta) \wedge \text{action}(\text{sy_sti})(\theta) \in \theta_{\text{set}}$

$\wedge \forall \theta': \Theta \bullet \theta' \in \theta_{\text{set}} \Rightarrow$

$\exists \text{se_rul}: \text{RUL} \bullet \text{se_rul} \in \text{hum_int}(\text{sy_rul})(\theta) \Rightarrow \text{se_rul}(\theta, \theta')$

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

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

On Modelling Human Behaviour

To model human behaviour is, “initially”, much like modelling management and organisation. But only ‘initially’. The most significant human behaviour modelling aspects is then that of modelling non-determinism and looseness, even ambiguity. So a specification language which allows specifying non-determinism and looseness (like CafeOBJ [147, 148, 161, 162] and RSL [166]) is to be preferred.

A.4.7 Completion

Domain acquisition resulted in typically up to thousands of units of domain descriptions. Domain analysis subsequently also serves to classify which facet

any one of these description units primarily characterise. But some such “compartmentalisations” may be difficult, and may be deferred till the step of “completion”. It may then be, “at the end of the day”, that is, after all of the above facets have been modelled that some description units are left as not having been described, not deliberately, but “circumstantially”. It then behooves the domain engineer to fit these “dangling” description units into suitable parts of the domain description. This “slotting in” may be simple, and all is fine. Or it may be difficult. Such difficulty may be a sign that the chosen model, the chosen description, in its selection of 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”.

A.4.8 Integrating Formal Descriptions

We have seen that to model the full spectrum of domain facets one needs not one, but several specification languages. No single specification language suffices. It seems highly unlikely and it appears not to be desirable to obtain a single, “universal” specification language capable of “equally” elegantly, suitably abstractly modelling all aspects of a domain. Hence one must conclude that the full modelling of domains shall deploy several formal notations. The issues are then the following which combinations of notations to select, and how to make sure that the combined specification denotes something meaningful. The ongoing series of “Integrating Formal Methods” conferences [6, 126, 129, 171, 265] is a good source for techniques, compositions and meaning.

A.5 From Domain Models to Requirements

One rôle for *Domain* descriptions is to serve as a basis for constructing , *Requirements* prescriptions. The purpose of constructing *Requirements* prescriptions is to specify properties (not implementation) of a *Machine*. The *Machine* is the hardware (equipment) and the software that together implements the requirements. The implementation relations is

$$\mathcal{D}, \mathcal{M} \models \mathcal{R}$$

The *Machine* is proven to implement the *Requirements* in the context of [assumptions about] the *Domain*. That is, proofs of correctness of the *Machine* wrt. the *Requirements* often refer to properties of the *Domain*.

A.5.1 Domain Requirements

First, in a concrete sense, you copy the domain description and call it a requirements prescriptions. Then that requirements prescription is subjected to a number of operations: (i) removal (projection away) of all those aspects not needed in the requirements; (ii) instantiation of remain aspects to the specifics of the client's domain; (iii) making determinate what is unnecessarily or undesirably non-deterministic in the evolving requirements prescription; (iv) extending it with concepts not feasible in the domain; and (v) fitting these requirements to those of related domains (say monitoring & control of public administration procedures). The result is called a domain requirements.

A.5.2 Interface Requirements

From the domain requirements one then constructs the interface requirements: First one identifies all phenomena and concepts, entities, functions, event and behaviours shared with the environment of the machine (hardware + software) being requirements specified. Then one requirements prescribe how each shared phenomenon and concept is being initialised and updated: entity initialisation and refreshment, function initialisation and refreshment (interactive monitoring and control of computations), and the physiological man-machine and machine-machine implements.

A.5.3 Machine Requirements

Finally one deals with machine requirements performance, dependability, maintainability, portability, etc., where dependability addresses such issues as availability, accessibility, reliability, safety, security, etc.

A.6 Why Domain Engineering?

A.6.1 Two Reasons for Domain Engineering

We believe that one can identify two almost diametrically opposed reasons for the pursuit of domain descriptions. One is utilitarian, concrete, commercial and engineering goal-oriented. It claims that domain engineering will lead to better software, and to development processes that can be better monitored and controlled. and the other is science-oriented. It claims that establishing domain theories is a necessity, that it must be done, whither we develop software or not.

We basically take the latter, the science, view, while, of course, noting the former, the engineering consequences. We will briefly look at these.

A.6.2 An Engineering Reason for Domain Modelling

In a recent 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 quote⁵:

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

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 dealt with, one-by-one, and in depth, in Vol. 3 of my three volume book [89].

A.6.3 On a Science of Domains

Domain Theories

Although not brought out in this chapter the concept of domain theories must now be mentioned.

- *By a **domain theory** we shall understand a domain description together with lemmas, propositions and theorems that can be proved about the description — and hence can be claimed to hold in the domain.*

To create a domain theory the specification language must possess a proof system. It appears that the essence of possible theorems of — that is, laws about — domains can be found in laws of physics. For a delightful view of the law-based nature of physics — and hence possibly also of man-made universes we refer to Richard Feynmann’s Lectures on Physics [155].

⁵E-Mail to Dines Bjørner, CC to Robin Milner et al., July 19, 2006

A Scientific Reason for Domain Modelling

So, inasmuch as the above-listed issues of Sect. A.6.2, so aptly expressed in Tony's mastery, also of concepts (through his delightful mastery of words), are of course of utmost engineering importance, it is really, in our mind, the science issues that are foremost: We must first and foremost understand. There is no excuse for not trying to first understand. Whether that understanding can be "translated" into engineering tools and techniques is then another matter. But then, of course, it is nice that clear and elegant understanding also leads to better tools and hence better engineering. It usually does.

A.7 Conclusion

A.7.1 Summary

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.

A.7.2 Grand Challenges of Informatics

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.

A.7.3 Acknowledgements

The author (and editors) thank Springer for allowing this chapter — which is basically a rewriting and “condensing-editing” of Chaps. 5 and 11 of [89] to appear also in the present book. The author thanks Paul Boca and Jonathan P. Bowen for inviting him in the first place to give basically this chapter as a BCS FACS seminar in the summer of 2005.⁶ The author thanks Tony Hoare for permission to quote his characterisation of domain engineering (Sect. A.6.2); and the author thanks Prof. Kokichi Futatsugi, of JAIST, for inviting him to JAIST for a full year, 2006, to further study domain engineering and its implications, including arranging for time to compose this chapter. The author finally thanks his home institute (IMM/DTU) for giving him a year's sabbatical.

⁶At that time the book [87–89] was in the hands of the editors of Springer, i.e., not yet published.

A.8 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 [87–89].

B

Compositionality: Ontology and Mereology of Domains

This appendix chapter is the invited paper for the Willem-Paul de Roever Festschrift [103], July 4, 2008, Kiel, Germany.

Abstract

In this discursive paper we discuss compositionality of (i) simple entities, (ii) operations, (iii) events and (iv) behaviours. These four concepts, (i)–(iv), together define a concept of entities. We view entities as “things” characterised by properties. We shall review some such properties. Mereology, the study of part-whole relations is then applied to a study of composite entities. We then speculate on compositionality of simple entities, operations, events and behaviours in the light of their mereologies. entities. We end the paper with some speculations on the rôle of Galois connections in the study of compositionality and domain mereology.

B.1 A Prologue Example

We begin with an example: an informal and formal description of fragments of a domain of transportation. The purpose of such an example is to attach this example to our discussion of entities, and to enlarge the example with further examples to support this discussion of entities, and hence of mereology and ontology. The formalisation of the example narratives is expressed in the RAISE Specification Language, RSL [87–89, 97, 165–168] — but could as well have been expressed in Alloy, ASM, Event B, VDM or Z [1, 111, 112, 131, 157, 158, 186, 187, 202, 261, 262, 283, 284, 297].

Narrative

(0.) There are links and there are hubs, (1.) Links and hubs have unique identifiers. (2.) Transport net consists of links and hubs. We can either model nets as sorts and then observe links and hubs from nets:

type		or
N, L, H,		
value		type
obs_Ls: $N \rightarrow \mathbf{L\text{-}set}$,		L, H,
obs_Hs: $N \rightarrow \mathbf{H\text{-}set}$		$N = \mathbf{L\text{-}set} \times \mathbf{H\text{-}set}$

(3.) Links connect exactly two distinct hubs. (4.) Hubs are connected to one or more distinct links. (5.) From a link one can observe the two unique identifiers of the hubs to which it is connected. (6.) From a hub one can observe the set of one or more unique identifiers of the links to which it is connected. (7.) Observed unique link (hub) identifiers are indeed identifiers of links (hubs) of the net in which the observation takes place.

Formalisation

type
0.–1. L, LI, H, HI,
2. $N = \mathbf{L\text{-}set} \times \mathbf{H\text{-}set}$
axiom
3.–4. $\forall (ls,hs):N \bullet \mathbf{card} \text{ } ls \geq 1 \wedge \mathbf{card} \text{ } hs \geq 2$
value
1. obs_LI: $L \rightarrow \mathbf{LI}$, obs_HI: $H \rightarrow \mathbf{HI}$,
5. obs_HIs: $L \rightarrow \mathbf{HI\text{-}set}$ **axiom** $\forall l:L \bullet \mathbf{card} \text{ } obs_HIs(l)=2$,
6. obs_LIs: $H \rightarrow \mathbf{LI\text{-}set}$ **axiom** $\forall h:H \bullet \mathbf{card} \text{ } obs_LIs(h) \geq 1$
axiom
7. $\forall (ls,hs):N \bullet$
 $\forall l:L \bullet l \in ls \Rightarrow$
 $\forall hi:HI \bullet hi \in obs_HIs(l) \Rightarrow \exists h:H \bullet hi=obs_HI(h) \wedge h \in hs$
 $\wedge \forall h:H \bullet h \in hs \Rightarrow$
 $\forall li:LI \bullet li \in obs_LIs(h) \Rightarrow \exists l:L \bullet li=obs_LI(l) \wedge l \in ls$

Narrative

(8.) There are vehicles (private cars, taxis, buses, trucks). (9.) Vehicles, when “on the net”, i.e., “in the traffic” (see further on), have positions. Vehicle positions are (10.) either at a hub, in which case we could speak of the hub identifier as being a suitable designation of its location, (11.) or along a link, in which case we could speak of of a quadruple of a (from) hub identifier, a(n along) link identifier, a real (a fraction) properly between 0 and 1 as designating a relative displacement “down” the link, and a (to) hub identifier, as

being a suitable designation of its location, (12.) Time is a discrete, dense well-ordered set of time points and time points are further undefined. (13.) Traffic can be thought of as a continuous function from time to vehicle positions. We augment our model of traffic with the net “on which it runs”!

Formalisation

type

8. V
9. $VPos == HubPos \mid LnkPos$
10. $HubPos = HP(hi:HI)$
11. $LnkPos = LP(fhi:HI, li:LI, f:Real, thi:HI)$
12. Time
13. $TRF = (Time \rightarrow (V \xrightarrow{m} VPos)) \times N$

Closing Remarks

We omit treatment here of traffic well-formedness: that time changes and vehicle movement occurs monotonically; that there are no “ghost” vehicles (vehicles “disappear” only to “reappear”), that two or more vehicles “one right after the other” do not “suddenly” change relative positions while continuing to move in the same direction, etc.

B.2 Introduction

The narrow context of this essay is that of domain engineering: the principles, techniques and tools for describing domains, as they are, with no consideration of software, hence also with no consideration of requirements. The example of Sect. B.1 describes (narrates and formalises) some aspects of a domain.

The broader context of this essay is that of formal software engineering: the phase, stage and stepwise development of software, starting with *Domain* descriptions, evolving into *Requirements* prescriptions and ending with *Software* design in such a way that $\mathcal{D}, \mathcal{S} \models \mathcal{R}$, that is: software can be proven correct with respect to requirements with the proofs and the correctness relying on the domain as described.

B.2.1 Domain Engineering

The Domain Engineering Dogma

Before software be designed, we must understand its requirements. Before requirements can be expressed, we must understand the application domain.

The Software Development Triptych

Thus, we must first describe the domain as it is. Then we can prescribe the requirements as we would like to see them implemented in software. First then can we specify the design of that software.

Domain Descriptions

A domain description specifies the domain **as it is**. (The example traffic thus allows vehicles to crash.) A domain description does not hint at requirements let alone software to be designed. A domain description specifies observable domain phenomena and concepts derived from these.

Example: *a vehicle is a phenomenon; a vehicle position is also a phenomenon, but the way in which we suggest to model a position is a concept; similarly for traffic.*

A domain description does not describe human sentiments (**Example:** *the bus ride is beautiful*), opinions and thoughts (**Example:** *the bus ride is a bit too expensive*), knowledge and belief (**Example:** *I know of more beautiful rides* and *I believe there are cheaper bus fares*), promise and commitment (**Example:** *I promise to take you on a more beautiful ride one day*) or other such sentential, modal structures.

A domain description primarily specifies semantic entities of the domain intrinsics (**Example:** *the net, links and hubs are semantics quantities*), semantic entities of support technologies already “in” the domain, semantic entities of management and organisation domain entities, syntactic and semantic of domain rules and regulations, syntactic and semantic of domain scripts (**Example:** *bus time tables respectively the bus traffic*) and semantic aspects of human domain behaviour.

The domain description, to us, is (best) expressed when both informally narrated and formally specified. A problem, therefore, is: can we formalise all the observable phenomena and concepts derived from these? If they are observable or derived, we should be able to formalise. But computing science may not have developed all the necessary formal description tools. We shall comment on that problem as we go along.

B.2.2 Compositionality

We shall view compositionality “in isolation”! That is, not as in the conventional literature where **the principle of compositionality** is the principle that the meaning of a complex expression is determined by the meanings of its constituent expressions and the rules used to combine them. We shall look at not only composite simple entities but also composite operations, events and behaviours in isolation from their meaning but shall then apply the principle of compositionality such that the meaning of a composite operation [event,

behaviour] is determined by the meanings of its constituent operations [event, behaviours] and the *rules used for combining* these. We shall, in this paper only go halfway towards this goal: we look only at possible *rules used to combine* simple entities, functions, events and behaviours.

For simple entities we can say the following about compositionality. A key idea seems to be that compositionality requires the existence of a homomorphism between the entities of a universe \mathcal{A} and the entities in some other universe \mathcal{B} .

Let us think of the entities of one system, \mathcal{A} , as a set, \mathcal{U} , upon which a number of operations are defined. This gives us an algebra $\mathcal{A} = (\mathcal{U}, \mathcal{F}_\nu)_{\nu \in \Gamma}$ where \mathcal{U} is the set of (simple and complex) entities and every \mathcal{F}_ν is an operation on \mathcal{A} with a fixed arity. The algebra \mathcal{A} is interpreted through a meaning-assignment \mathcal{M} ; a function from \mathcal{U} to \mathcal{V} , the set of available meanings for the entities of \mathcal{U} . Now consider \mathcal{F}_ν ; a k -ary syntactic operation on \mathcal{A} . \mathcal{M} is \mathcal{F}_ν -compositional just in case there is a k -ary function \mathcal{G} on \mathcal{V} such that whenever $\mathcal{F}_\nu(u_1, \dots, u_k)$ is defined

$$\mathcal{F}_\nu(u_1, \dots, u_k) = \mathcal{G}(\mathcal{M}(u_1), \dots, \mathcal{M}(u_k)).$$

In denotational semantics we take this homomorphism for granted, while applying to, as we shall call them, syntactic terms of entities. We shall, in this paper, speculate on compositionality of non-simple entities. That is, compositionality of operations, events and behaviours; that is, of interpretations over non-simple entities (as well as over simple entities).

B.2.3 Ontology

By an ontology we shall understand *an explicit, formal specification of a shared conceptualisation*¹.

We shall claim that domain engineering, as treated in [87,90,94], amounts to principles, techniques and tools for formal specification of shared conceptualisations. The conceptualisation is of a domain, typically a business, an industry or a service domain.

One thing is to describe a domain, that is, to present an ontology for that domain. Another thing is for the description to be anchored around a description ontology: a set of principles, techniques and tools for structuring descriptions. In a sense we could refer to this latter as a meta-ontology, but we shall avoid the prefix ‘meta-’ and instead understand it so. The conceptualisation is of the domain of software engineering methodology, especially of how to describe domains.

B.2.4 Mereology

Mereology is the theory of parthood relations: of the relations of part to whole and the relations of part to part within a whole.

¹<http://www-ksl.stanford.edu/kst/what-is-an-ontology.html>

The issue is not simply whether an entity is a proper part, p_p , of another part, p_ω (for example, “the whole”), but also whether a part, p_ι , which is a proper part of p_p can also be a part of another part, p_ξ which is not a part of p_p , etcetera. To straighten out such issues, axiom systems for mereology (part/whole relations) have been proposed [135, 140, 220]. See Appendix B.12.1.

The term mereology seems to have been first used in the sense we are using it by the Polish mathematical logician Stanisław Leśniewski [225, 285].

The concept of Calculus of Individuals [140, 221, Leonard & Goodman (1940) and Clarke (1981)] is related to that of Mereology. See Appendix B.12.2.

We shall return to the issue of mereology much more in this paper. In fact, we shall outline “precisely” what our entity mereologies are.

B.2.5 Paper Outline

The paper is structured as follows: after Sect. B.2’s brief characteristics of domain engineering, compositionality, ontology and mereology, Sect. B.3 overviews what we shall call an ontological aspect of description structures, namely that of entities (having properties). Sections B.4–B.7 will then study (i) simple, (ii) operation, (iii) event and (iv) behaviour entities in more detail, both atomic and composite. For the composite entities we shall then speculate on their mereology. Section B.8 concludes our study of some mereological aspects of composite entities by relating these to definitions and axioms of proposed axiom systems for mereology, cf. Appendix B.12. Section B.9 takes a brief look at rôles that the concept of Galois Connections may have in connection with composite entities.

B.3 An Ontology Aspect of Description Structures

This section provides a brief summary of Sects. B.4–B.7.

The choice of analysing a concept of compositionality from the point of view of simple entities, operations, events and behaviours reflects an ontological choice, that is a choice of how we wish to structure our study of conceptions of reality and the nature of being.

We shall take the view that an ontology for the domain of descriptions evolves around the concepts of entities inseparably from their properties. More concretely, “our” ontology consists of entities of the four kinds of specification types: simple entities, operations, events and behaviours. One set of properties is that of an entity being ‘simple’, being ‘an operation’ (or function), being ‘an event’ or being ‘a behaviour’. We shall later introduce further categories of entity properties.

B.3.1 Simple Entities²

In a rather concrete, “mechanistic” sense, we understand simple entities as follows: simple entities have properties which we model as types and values. When a simple entity is concretely represented, “inside” a computer, it is usually implemented in the form of data.

By a **state**, $\sigma : \Sigma$, we shall understand a designated set of entities.

Entities are the target of operations: function being applied to entities and resulting in entities.

In Sect. B.4 we shall develop this view further.

Examples: *The nets, links, hubs, vehicles and vehicle positions of our guiding example are simple entities*■

Simple domain entities are either atomic or composite. Composite entities are here thought of (i.e., modelled as) finite or infinite sets of (simple) entities: $\{e_1, e_2, \dots, e_n\}$, finite Cartesians (i.e., groupings [records, structures] of (simple) entities): (e_1, e_2, \dots, e_n) , finite or infinite lists (i.e., ordered sequences of (simple) entities): $\langle e_1, e_2, \dots, e_n \rangle$, maps (i.e., finite associations of (simple) entities to (simple) entities): $[e_{d_1} \mapsto e_{r_1}, e_{d_2} \mapsto e_{r_2}, \dots, e_{d_n} \mapsto e_{r_n}]$, and functions (from (simple) entities to (simple) entities): $\lambda v : \mathcal{E}(v)$.³

B.3.2 Operations

To us, an **operation** (synonym for function) is something which **when** applied to an entity or an attribute⁴ yields an entity or an attribute.

If an operation op argument and the resulting entity qualify as states ($\sigma : \Sigma$), then we have a state-changing **action**: $op : [\dots \times] \Sigma \rightarrow \Sigma$.

If an operation argument entity qualifies as a state and if the resulting entity can be thought of as a pair of which (exactly) one element qualifies as a state, then we have a value yielding action with a, perhaps, beneficial *side effect*: $op : [\dots \times] \Sigma \rightarrow (\Sigma \times \text{VAL})$.

If the operation argument does not qualify as a state then we have a value yielding function with no side effect on the state.

Since entities have types we can talk of the signature of an operation as consisting of the name of the operation, the structure of types of its argument entities, and the type of the resulting entities. We gave two such signatures (for operation op) above. (The $[\dots \times]$ indicate that there could be other arguments than the explicitly named state entity Σ .)

²The term ‘simple entity’ is chosen in contrast to the (‘complex’) function, event and behaviour entities. We shall otherwise not use the term ‘complex’ as it has no relation to composition, but may be confused with it.

³**Note:** The decorated es in set, Cartesian, list and map enumerations stand for actual entities whereas the v in $\lambda v : \mathcal{E}(v)$ is a syntactic variable and $\mathcal{E}(v)$ stand for a syntactic expression with a free variable v .

⁴See Sect. B.4.1 for distinction between entity and attribute

Example: *The unique identifier observer functions of our guiding example are operations*■

They apply to entities and yields entities or attributes: $\text{obs_Ls:N} \rightarrow \text{L-set}$ and $\text{obs_Hs:N} \rightarrow \text{H-set}$ yield entities and $\text{obs_LI:L} \rightarrow \text{LI}$ and $\text{obs_HI:H} \rightarrow \text{HI}$ yield attributes.

“First Class” Entities

Before closing this section, Sect. B.3.2, we shall “lift” operations, hence actions and functions to be first class entities!

B.3.3 Events

In [216, Lamport] events are the same as executed atomic actions. We shall not really argue with that assumption. In [216, Lamport] events are of interest only in connection with the concept of processes (for which we shall use the term ‘behaviours’). We shall certainly follow that assumption. We wish to reserve the term ‘event’ for such actions which (i) are either somehow shared between two or more behaviours, (ii) or ‘occur’ in just one behaviour. We assume an “external”, further undefined behaviour. For both of these two cases we need a way of “labelling” events. We do so by labelling, βl_i , behaviours, β_i , that is, ascribing names to behaviours. Let the external behaviour have a distinguished, “own” label (e.g., $\beta_\chi \ell$). Now we can label an event by the set of labels of the processes “in” which the event occur. That is, with either two or more labels, or just one. When the external behaviour label $\beta_\chi \ell$ is in the set then it shall mean that the event either “originates” outside the behaviours of the other labels, or is “directed” at all those behaviours. We do not, however, wish to impose any direction! Here we wish to remind the reader that “our” behaviours take place “in the domain”, that is, they are not necessarily those of computing processes, unless, of course, the domain is, or (“strongly”) includes that of computing; and “in the domain” we can always speak “globally”, that is: we may postulate properties that may not be computable or even precisely observable, that is: two time stamps may be different even though they are of two actions or events that actually did or do take place simultaneously.

Thus: we are not bothered by clocks, that is, we do not enforce a global clock; we do not have to sort out ordering problems of events, but can leave that to a later analysis of the described domain, recommendably along the lines of [216, Lamport].

Time and Time Stamps

Time is some dense set of time points.

A time stamp is just a time designator, t . Two time stamps are consecutive if they differ by some infinitesimal time difference, t_δ . We shall assume the

simplifying notion of a “global” clock. For the kind of distributed systems that are treated in [216, Lamport] this may not be acceptable, but for a any actual domain that is not subject to Einsteinian relativity, and most are, it will be OK. Once we get to implementation in terms of actual systems possibly governed by erroneously set clocks one shall have to apply for example [216, Lamport]’s treatment.

Definition: Event

To us, an event, $\mathbb{E} : \{(\beta\ell_1, \sigma_1, P_1, \sigma'_1, \tau_1), (\beta\ell_2, \sigma_2, P_2, \sigma'_2, \tau_2), \dots, (\beta\ell_n, \sigma_n, P_n, \sigma'_n, \tau_n)\}$ involves a set of behaviours, β_i , and is expressed in terms of a set of event designators, quintuplets containing:

- ★¹ a label $\beta\ell_i$,
 - ★² a before state σ_i ;
 - ★³ a predicate P_i ;
 - ★⁴ an after state σ'_i ;
 - such that $P_i(\sigma_i, \sigma'_i)$
 - but where it may be the case that $\sigma_i = \sigma'_i$;
 - ★⁵ and a time stamp τ_i
 - which is either a time t_i
 - or a time interval $[t'_i, t''_i]$
 - * such that $t''_i - t'_i = \tau_{\delta_i} > 0$
 - * but where τ_{δ_i} is otherwise considered “small”
-

An event, \mathbb{E} , may change one or more behaviour states, selectively, or may not — in which latter case $\sigma_i = \sigma'_i$ for some i .

Thus we do not consider the time(s) when expressing conditions P_i .

Definition: Same Event

We assume two or more distinct behaviours $\beta_1, \beta_2, \dots, \beta_n$. Two or more events $\mathbb{E}_{1_i}, \mathbb{E}_{2_i}$ and \mathbb{E}_{n_i} are said to reflect, i.e., to be the same event iff their models, as suggested above, are ‘identical’ modulo predicates⁵ and time stamps, iff these time stamps differ at most “insignificantly”, a decision made by the domain describer⁶, and iff this model involves the label sets $\beta\ell_1, \beta\ell_2, \dots, \beta\ell_n$ for behaviours $\beta_1, \beta_2, \dots, \beta_n$ ■

This means that any one event which is assumed to be the same and thus to occur more-or-less simultaneously in several behaviours is “identically” recorded (modulo predicates and time stamps) in those behaviours.

⁵The predicates can all “temporarily”, for purposes of “identity”, be set to **true**.

⁶The time stamps can all “temporarily”, for purposes of “identity”, be set to the smallest time interval within which all time stamps of the event are included.

We can accept this definition since it is the domain describer who decides which events to model and since it is anyway only a postulate: we are “observing the domain”!

Definition: Event Designator:

The event $\mathbb{E} : \{(\beta\ell_1, \sigma_1, P_1, \sigma'_1, \tau_1), (\beta\ell_2, \sigma_2, P_2, \sigma'_2, \tau_2), \dots, (\beta\ell_n, \sigma_n, P_n, \sigma'_n, \tau_n)\}$ consists of n event designators $(\beta\ell_i, \sigma_i, P_i, \sigma'_i, \tau_i)$, that is: an event designator is that kind of quintuplet.

Example: *Withdrawal of funds from an account* (i.e., a certain action) leads to either of two events: either *the remaining balance is above or equal to the credit limit*, or *it is not* ■

The withdrawal effects a state change (into state σ'), but “below credit limit” event does not cause a further state change (that is: $\sigma = \sigma'$). In the latter case that event may trigger a corrective action but the ensuing state change (from some (possibly later state) σ'' to, say, σ''' , that is, σ''' is usually not a “next state” after σ').

Example: *A national (or federal) bank changes its interest rate.* This is an action by the behaviour of a national (or federal) bank, but is seen as an event by (the behaviour of) a(ny) local bank, and may cause such a bank to change (i.e., an action) its own interest rate ■

Example: *A local bank goes bankrupt at which time a lot of bank clients loose a lot of their money* ■

Some events are explicitly willed, and are “un-interesting”. Other events are “surprising”, that is, are not willed, and are thus “interesting”. Being interesting or not is a pragmatic decision by the domain describer.

“First Class” Entities

Before closing this section, Sect. B.3.3, we shall “lift” events to be first class entities !

B.3.4 Behaviours

A simple, sequential behaviour, β , is a possibly infinite, possibly empty sequence of actions and events.

Example: *The movement of a single vehicle between two time points forms a simple, sequential behaviour* ■

We shall later construct composite behaviours from simple behaviours. In essence such composite behaviours is “just” a set of simple behaviours. In such a composite behaviour one can then speak of “kinds” of consecutive

or concurrent behaviours. Some concurrent behaviours can be analysed into communicating, joined, forked or “general” behaviours such that any one concurrent behaviour may exhibit two or more of these ‘kinds’.

Section B.7.3 presents definitions of composite behaviours.

“First Class” Entities

Before closing this section, Sect. B.3.4, we shall “lift” behaviours to be first class entities!

B.3.5 First-class Entities

Operations are considered designators of actions. That is, they are action descriptions. We do not, in this paper, consider forms of descriptions of events (labels) and behaviours. In that sense, of not considering, this section is not “completely” symmetrical in its treatment of operations, actions, events and behaviours as first-class entities. Be that as it may.

Operations as Entities

Operations may be (parameterised by being) applicable to operation entities — and we then say that the operations are higher-order operations: Sorting a set of rail units according to either length or altitude implies one sorting operation with either a select rail unit length or a select altitude parameter. (The ‘select’ is an operation.)

Actions as Entities

Similarly operations may be (parameterised by being) applicable to actions: Let an action be the invocation of the parameterised sorting function — cf. above. Our operation may be that of observing storage performance. There are two sorting functions: one according to rail unit length, another according to rail unit altitude. We may now be able, given the action parameter, to observe, for example the execution time!

Events as Entities

Operations may be (parameterised by being) applicable to a set of event entities: Recall that events are dynamic, instantaneous ‘quantities’. A ‘set of event entities’ as a parameter can be such a quantity. One could then inquire as to which one or more events occurred first or last, or, if they had a time duration, which took the longest to occur! This general purpose event handler may then be further parameterised by respective rail or air traffic entities!

Behaviours as Entities

Finally operations may be (parameterised by being) applicable to behaviours. We may wish to monitor and/or control train traffic. So the monitoring & control operation is to be real-time parameterised by train traffics. Similar for air traffic, automobile performance, etc.

• • •

We are not saying that a programming language must provide for the above structures. We are saying that, in a domain, as it is, we can “speak of” these parameterisations. Therefore we conclude that actions, events and behaviours — that these dynamic entities which occur in “real-time” — are entities. Whether we can formalise this “speaking of” is another matter.

B.3.6 The Ontology: Entities and Properties

On the background of the above we can now summarise our ontology: it consists of (“first class”) entities inseparable from their properties. We hinted at properties, in a concrete sense above: as something to which we can ascribe a name, a type and a value. In contrast to common practice in treatises on ontology [160, 170, 222, 231, 287], we “fix” our property system at a concrete modelling level around the value types of atomic simple entities (numbers, Booleans, characters, etc.) and composite simple entities (sets, Cartesians, lists, maps and functions); and at an abstract, orthogonal descriptive level, following Jackson [206], static and inert, active (autonomous, biddable, programmable) and reactive dynamic types; continuous, discrete and chaotic types; tangible and intangible types; one-, two-, etc., n -dimensional types; etc.

Ontologically we could claim that an entity exists qua its properties; and the only entities *that we are interested in* are those that can be formalised around such properties as have been mentioned above.

B.4 Simple Atomic and Composite Entities

Entities are either atomic or composite. The decision as to which entities are considered what is a decision taken solely by the describer. The decision is based on the choice of abstraction level being made.

B.4.1 Simple Attributes — Types and Values

With any entity whether atomic or composite, and then also with its sub-entities, etcetera, one can associate one or more simple attributes.

- *By a **simple attribute** we understand a pair of a designated **type** and a named **value**.*

Attributes are not entities: they merely reflect some of the properties of an entity. Usually we associate a name with an entity. Such an association is purely a pragmatic matter, that is, not a syntactic and not a semantic issue.

B.4.2 Atomic Entities

- By an **atomic entity** we intuitively understand a simple attributes entity which ‘cannot be taken apart’ (into other, the sub-entities).

Example: We illustrate attributes of an atomic entity.

Atomic Entity: Bus Ticket	
Type	Value
Bus Line	Greyhound
From, Departure Time	San Francisco, Calif.: 1:30 pm
To, Arrival Time	Reno, Nevada: 6:40 pm
Price	US \$ 52.00

‘Removing’ attributes from an entity destroys its ‘entity-hood’, that is, attributes are an essential part of an entity.

B.4.3 Composite Entities

- By a *composite entity* we intuitively understand an entity (i) which “can be taken apart” into sub-entities, (ii) where the composition of these is described by its **mereology**, and (iii) which further possess one or more attributes.

Example: We “diagram” the relations between sub-entities, mereology and attributes of transport nets.

Composite Entity: Transport Net		
Sub-entities: Links		
Hubs		
Mereology: “set” of one or more ℓ (inks) and “set” of two or more h (hub’s) such that each ℓ (ink) is delimited by two h (hub’s) and such that each h (ub) connects one or more ℓ (inks)		
Attributes		
	Types:	Values:
	Multimodal	<i>Rail, Roads, Sea_Lane, Air_Corridor</i>
	Transport Net of	<i>Denmark</i>
	Year Surveyed	<i>2008</i>

B.4.4 Discussion

Attributes

Domain entity attributes whether of atomic entities or of composite entities are modelled as a set of pairs of distinctly named types and values. It may be that such entity attributes, some or all, could be modelled differently, for example as a map from type names to values, or as a list of pairs of distinctly named types and values, or as a Cartesian of values, where the position determines the type name — somehow known “elsewhere” in the formalisation, etcetera

But it really makes no difference as remarked earlier: one cannot really remove any one of these attributes from an entity.

Compositions

We formally model composite entities in terms of its immediate sub-entities, and we model these as observable, usually as sets, immediately from the entity (cf. `obs_Hs`, `obs_Ls`, `N`). In the example composite entity (`nets`) above the net can be considered a graph, and graphs, $g:G$, are, in Graph Theory typically modelled, for example, as

```

type
  V
  G = (V × V)-set

```

where vertexes ($v:V$) are thought of as names or references.

We shall comment on such a standard graph-theoretic model in relation to a domain model which somehow expresses a graph: First it has abstracted away all there may otherwise be to say about what the graph actually is an abstraction of. In such models we model edges in terms of pairs of vertexes. That is: edges do not have separate “existence” — as have segments. In other words, since we can phenomenologically point to any junction and a segment we must model them separately, and then we must describe the mereology of nets separate from the description of the parts.

B.5 Atomic and Composite Operations

Entities are either atomic or composite. The decision as to which operations are considered what is a decision solely taken by the describer.

B.5.1 Signatures — Names and Types

With any operation whether atomic or composite, and then also with its sub-operations, etcetera, one can associate a signature which we represent as a triple: the name of the operation, the arguments to which the operation is applicable, and the result, whether atomic or composite.

- *By an **argument** and a **result** we understand the same as an attribute or an entity.*

B.5.2 Atomic Operations

We understand operations as functions in the sense of recursive function theory [230]⁷ but extended with postulated primitive observer (**obs_...**), constructor (**mk...**) and selector (**s_...**) functions, non-determinacy⁸ and non-termination (i.e., the result of non-termination is a well-defined **chaotic** value).

- *By an **atomic operation** we intuitively understand an operation which 'cannot be expressed in terms of other (phenomenological or conceptual), primitive recursive functions.*

Example Atomic Operations

The operation of obtaining the length of a segment, **obs_Lgth**, is an atomic operation. The operation of calculating the sum, **sum**, of two segment lengths is an atomic operation.

```

type
  Lgth
value
  obs_Lgth: L → Lgth,
  sum: Lgth × Lgth → Lgth

```

B.5.3 Composite Operations

- *By a **composite operation** we intuitively understand an operation which can best be expressed in terms of other (phenomenological or conceptual) primitive recursive functions, whether atomic or themselves composite.*

⁷See: <http://www-formal.stanford.edu/jmc/basis1/basis1.html>

⁸Hinted at in [230] as ambiguous functions, cf. Footnote 7.

Example Composite Operations

Finding the length of a route, `R_Lgth`, where a route is a sequence of segments joined together at junctions is a composite operation — its sub-operations are the operation of observing a segment length from a segments, `obs_length`, and the recursive invocation of `route_length`. Finding the total length of all segments of a net is likewise a composite operation.

```

value
  length: L* → Lgth,
  zero_lgth:Lgth,
  length(⟨⟩) ≡ zero_lgth,
  length(ℓ~ℓ') ≡ sum(ℓ,ℓ')

```

The Composition Homomorphism

Usually composite operations are applied to composite entities. In general, we often find that the functions applied to composite entities satisfy the following homomorphism:

$$\mathcal{G}(e_1, e_2, \dots, e_m) = \mathcal{H}(\mathcal{G}(e_1), \mathcal{G}(e_2), \dots, \mathcal{G}(e_n))$$

where \mathcal{G} and \mathcal{H} are suitable functions.

• • •

Example: Consider the *Factorial* and the *List Reversal* functions. This example is inspired by [228]. Let ϕ be the sentence:

$$\exists F \bullet ((F(a) = b) \wedge \forall x \bullet (p(x) \supset (F(x) = H(x, F(f(x)))))$$

which reads: there exists a mathematical function F such that, \bullet , the following holds, namely: $F(a) = b$ (where a and b are not known), and, \wedge , for every (i.e., all) x , it is the case, \bullet , that if $p(x)$ is true, then $F(x) = H(x, F(f(x)))$ is true.

There are (at least) two possible (model-theoretic) interpretations of ϕ . In the first interpretation, we first establish the type Ω of natural numbers and operations on these, and then the specific context ρ :

```

[F ↦ fact, a ↦ 1, b ↦ 1, f ↦ λ n.n-1,
 H ↦ λ m.λ n.m+n,
 p ↦ λ m.m>0 ]

```

We find that ϕ is true for the factorial function, `fact`. In other words, ϕ characterises properties of that function.

In the second interpretation we first establish the type Ω of lists and operations on these: and then the specific context ρ :

$$\begin{aligned}
& [F \mapsto \text{rev}, a \mapsto \langle \rangle, b \mapsto \langle \rangle, f \mapsto \mathbf{tl}, \\
& H \mapsto \lambda \ell_1. \lambda \ell_2. \ell_1 \hat{\sim} (\mathbf{hd} \ell_1), \\
& p \mapsto \lambda \ell. \ell \neq \langle \rangle]
\end{aligned}$$

And we find that ϕ is true for the list reversal function, `rev`, as well. In other words, ϕ characterises properties of that function, and the two H s express a mereological nature of composition. ■

B.6 Atomic and Composite Events

Usually events are considered atomic. But for the sake of argument — that is, as a question of scientific inquiry, of the kind: *why not investigate*, seeking “orthogonality” throughout, now that it makes sense to consider atomic and composite entities and operations — we shall explore the possibility of considering composite events.

Let us first recall that we model an event by: $\mathbb{E} : \{(\beta \ell_1, \sigma_1, P_1, \sigma'_1, \tau_1), (\beta \ell_2, \sigma_2, P_2, \sigma'_2, \tau_2), \dots, (\beta \ell_n, \sigma_n, P_n, \sigma'_n, \tau_n)\}$, where \mathbb{E} is just a convenient name for us to refer to the event, $\beta \ell_i$ is the label of a behaviour β_i , σ_i and σ'_i are (‘before event’, respectively ‘after event’) states (of behaviour β_i), P_i is a predicate which characterises the event as seen by behaviour β_i , and τ_i is a time, t_i , or a time interval, $[t_{ib}, t_{ie}]$, time stamp.

B.6.1 Atomic Events

Examples: (i) \mathbb{E}_1 : a vehicle “drives off” a net link at high velocity; (ii) \mathbb{E}_2 : a link “breaks down”: (ii.a) \mathbb{E}_{2_1} : a bridge collapses, or (ii.b) \mathbb{E}_{2_2} : a mud slide covers the link ■

That is \mathbb{E}_2 is due to either \mathbb{E}_{2_1} or \mathbb{E}_{2_2} .

One can discuss whether these examples really can be considered atomic: (ii.a) the bridge may have collapsed due to excess load and thus the moment at which the load exceeded the strength limit could be considered an event causing the bridge collapse; (ii.b) the mud slide may have been caused by excessive rain due to rainstorm gutters exceeding their capacity and thus the moment at which capacity was exceeded could be considered an event causing the mud slide.

We take the view that it is the decision of the domain describer to “fix” the abstraction level and thus decide whether the above are atomic or composite events.

In general we could view an event, such as summarised above, which involves two or more distinct behaviours as a composite event. We shall take that view.

B.6.2 Definitions: Atomic and Composite Events

Definition: Atomic Event: An *atomic event* is either a *single [atomic] internal event*: $\{(\beta\ell_i, \sigma_i, P_i, \sigma'_i, \tau_i)\}$, that is, consists of just one event designator, or is a *single [atomic] external event*, that is, is a pair event designators where one of these involves the external behaviour: $\{(\beta\chi\ell, \sigma_{\text{nil}}, \mathbf{true}, \sigma_{\text{nil}}, \tau_\chi), (\beta\ell_i, \sigma_i, P_i, \sigma'_i, \tau_i)\}$, that is, consists of two event designators, an external and an internal ■

Definition: Composite Event: A *composite event* is an event which consists of two or more internal “identical” event designators, that is, event designators from two or more simple, non-external behaviours, and possibly also an event designator from an external behaviour “identical” to these internal event designators ■

B.6.3 Composite Events

Examples: (i) *two or more cars crash* and (ii) *a bridge collapse causes one or more cars or bicyclists and people to plunge into the abyss* ■

Synchronising Events

Events in two or more simple behaviours are said to be synchronising iff they are identical.

Example: *Two cars crashing means that the surfaces of the crash is a channel on which they are synchronising and that the messages being exchanged are “you have crashed with me”* ■

Sub-Events

A composite event defines one or more sub-events.

Definition Sub-event: An event $\mathbb{E}_s:eds'$, is a sub-event of another event $\mathbb{E}:eds$, iff $eds' \subset eds$, that is the set eds' of event designators of \mathbb{E}_s is a proper subset eds of the event designators of \mathbb{E} ■

Sequential Events

One way in which a composite event is structured can be as a “sequence” of “follow-on” sub-events. One sub-event: $\mathbb{E}_{s_{12}}: \{(\beta\ell_1, \sigma_1, P_1, \sigma'_1, \tau_1), (\beta\ell_2, \sigma_2, P_2, \sigma'_2, \tau_2)\}$, for example, “leads on” to another sub-event: $\mathbb{E}_{s_{23}}: \{(\beta\ell_2, \sigma'_2, P'_2, \sigma''_2, \tau'_2), (\beta\ell_3, \sigma_3, P_3, \sigma'_3, \tau_3)\}$, etcetera, “leads on” to a final event: $\mathbb{E}_{s_{mn}}: \{(\beta\ell_m, \sigma''_m, P_m, \sigma'''_m, \tau_m), (\beta\ell_n, \sigma_n, P_n, \sigma'_n, \tau_n)\}$. The “leads on” relation should appear obvious from the above expressions.

Example: *The multiple-car crash in which the cars crash, “immediately” one after the other, as in a accordion movement* ■

(This is, of course, an idealised assumption.)

Embedded Events

Another way in which a composite event is structured is as an “iteratively” (or finite “recursively”) embedded “repetition” of (albeit distinct) sub-events. Here we assume that the τs stand for time intervals and that $\tau s' \sqsubseteq \tau s$ it means that the time interval $\tau s'$ is embedded with τs , that is, let $\tau s = [t_b, t_e]$ and $\tau s' = [t'_b, t'_e]$, then for $\tau s' \sqsubseteq \tau s$ means that $t_b \leq t'_b$ and $t'_e \leq t_e$. Now we postulate that one event (or sub-event) \mathbb{E}_i embeds a sub-event \mathbb{E}_{i_j} , ..., embeds an “innermost” sub-event $\mathbb{E}_{i_j \dots k}$.

Example: The following represents an idealised description of how a computing system interrupt is handled.

- (i) *A laptop user hits the ENTER keyboard key at time t_b .*
- (ii) *The computing system interrupt handler reacts at time t'_b ($t_b \leq t'_b$), to the hitting of the ENTER keyboard key.*
- (iii) *The interrupt handler forwards, at time t''_b , the hitting of the ENTER keyboard key to the appropriate input/output handler of the computing system keyboard handler.*
- (iv) *The keyboard handler forwards, at time t'''_b , the hitting of the ENTER keyboard key to the appropriate application program routine.*
- (v) *The application program routine calculates an appropriate reaction between times t'''_b and t'''_e .*
- (vi) *The application program routine returns its reaction to the keyboard handler at time t'''_e .*
- (vii) *The keyboard handler returns, at time t''_e , that reaction to the interrupt handler.*
- (viii) *The interrupt handler marks the interrupt as having been fully served at time t'_e ,*
- (ix) *while whatever (if anything that has been routed to, for example, the display associated with the keyboard) is displayed at time t_e* ■

The pairs (i,ix), (ii,viii), (iii,vii) and (iv,vi) form pairwise embedded events: (ii,vii) is directly embedded, \sqsubseteq , in (i,ix), (iii,vii) is directly embedded, \sqsubseteq , in (ii,viii) and (iv,vi) is directly embedded, \sqsubseteq , in (iii,vii).

We have abstracted the time intervals to be negligible.

Event Clusters

A final way of having composite events, is for them, as a structure, to be considered a set of sub-events, each eventually involving a time or a time

period that is “tightly” related to those of the other sub-events in the set and where the relation is not that of “follow-on” or embeddedness.

Example: *A (i) car crash results in a (ii) person being injured, while a (iii) robber exploits the confusion to steal a purse, etcetera* ■

B.7 Atomic and Composite Behaviours

Our treatment of behaviours in Sect. B.3.4 was very brief. In this section it will be more detailed. First a preliminary.

B.7.1 Modelling Actions and Events

In modelling behaviours, we model actions by a triple, $(\beta\ell, \alpha, \tau s)$, consisting of a behaviour label, $\beta\ell:\text{BehLbl}$, an operation denotation, $\alpha:[\dots \times]\Sigma \rightarrow \Sigma[\times\dots]$, and a time stamp, τs . Events are modelled by as above.

B.7.2 Atomic Behaviours

Time-stamped actions and atomic events are the only atomic behaviours. We shall model atomic behaviours as singleton sequences of a time-stamped action or an event.

B.7.3 Composite Behaviours

Simple Traces

A simple (finite/infinite) trace, τ , is a (finite/infinite) sequence of one or more time-stamped atomic actions and time-stamped (atomic or composite) events. Trace time stamps occur in monotonically increasing dense order, i.e., separated by consecutive (overall) time stamps. That is, two traces may operate not only on different clocks, but have varying time intervals between consecutive actions or events. The “overall” time stamp of a composite event is the smallest time interval which encompasses all time and time stamps of event designators of the composite event.

Simple Behaviours

A simple behaviour, β , is a simple trace of length two or more.

Example: *The movement of two or more vehicles between two time points forms a simple, concurrent behaviour* ■

One can usually decompose a simple behaviour into two or more consecutive behaviours, and hence one can compose a consecutive behaviour from two or more simple behaviours. Consecutive behaviours are simple behaviours.

Consecutive Behaviours

A consecutive behaviour is a pair of simple behaviours, of which the first is finite, such that the time stamp of the first action or event of the second behaviour is consecutive to the time stamp of the last action or event of the first behaviour, cf. Fig. B.1 on the next page.

Example: *A train travel, seen from the point of view of one train passenger, from one city to another, involving one or more train changes, and including the train passenger's behaviours at train stations of origin, intermediate stations and station of destination as well as during the train rides proper, forms a consecutive behaviour.*■

Concurrent Behaviours

A concurrent behaviour is a set of two or more simple behaviours $\{\beta_1, \beta_2, \dots, \beta_n\}$ such that for each behaviour $\beta_i \in \{\beta_1, \beta_2, \dots, \beta_n\}$ there is a set of one or more different behaviours $\{\beta_{i_j}, \beta_{i_k}, \dots, \beta_{i_\ell}\} \subseteq \{\beta_1, \beta_2, \dots, \beta_n\}$ such that there is a set of one or more consecutive (dense) time stamps that are shared between behaviours β_i and $\{\beta_{i_j}, \beta_{i_k}, \dots, \beta_{i_\ell}\}$.

Example: *The movement of two vehicles between two time points (i.e., in some interval) forms a concurrent behaviour.*■

Concurrent behaviours come in several forms. These are defined next.

Communicating Behaviours

A communicating behaviour is a concurrent behaviour in which two or more (simple) behaviours contain identical (modulo predicate and time stamp) events.

Example: *The movement of two vehicles between two time points (i.e., in some interval), such that, for example, the two vehicles, after some time point in the interval, at which both vehicles have observed their “near-crash”, keeps moving along, may be said to be a simple, cooperating behaviour. Their “near-crash” is an event. In fact the vehicles may be engaged in several such “near-crashes” (drunken driving!).*■

Example: *The action of a vehicle, at a hub, which effects both a turning to the right down another link, and a sequence of one or more gear changes, throttling down, then up, the velocity, while moving along in the traffic, forms a general, structured behaviour.*■

Example: *A crash between two vehicles defines an event with the two vehicles being said to be synchronised and exchanging messages at that event.*■

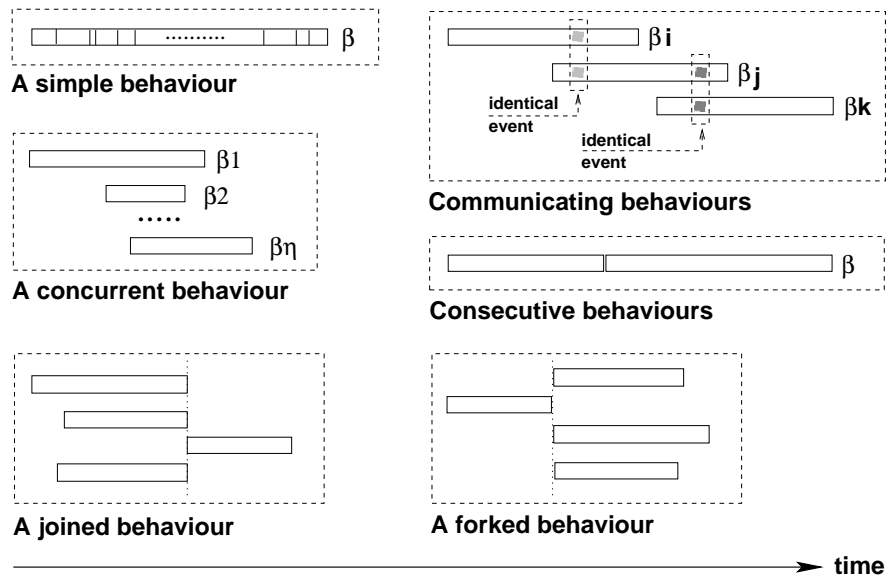


Fig. B.1. Two simple and four composite behaviours

Each rectangle designates a simple behaviour. Figure indicates 17 such

Joined Behaviours

A joined behaviour is a pair of a finite set, $\{\beta_1, \beta_2, \dots, \beta_n\}$, of finite (“first”) simple behaviours and a (“second”) simple behaviour, such that the time stamp of the first action or event of the second behaviour is consecutive to the time stamp of the last action or event of each of the the first behaviours. You can think of the joined behaviour as pictured in Fig. B.1.

Example: This example assumes a mode of travel by vehicles in which they (sometimes) travel in platoons, or convoys, as do military vehicles and — maybe future private cars. *A behaviour which starts with n (n being two or more) vehicles travelling by themselves, as n concurrent behaviours; where independent vehicles, at one time or another, join into convoy behaviours involving two or more vehicles, form a joined behaviour.*■

Forked Behaviours

A forked behaviour is a pair of a finite (“first”) simple behaviour β and a finite set, $\{\beta_1, \beta_2, \dots, \beta_n\}$, of (“second”) simple behaviours, such that the time stamp of the first action or event of each of the second behaviours is consecutive to the time stamp of the last action or event of the first behaviour. You can think of the joined behaviour as pictured in Fig. B.1.

Example: Continuing the example just above: *A behaviour which starts as the joined, convoy behaviour of two or more (i.e., n) vehicles which then*

proceeds by individual vehicles, at one time or another, leaving the convoy, i.e., “forking out” into concurrent behaviours, forms a forked behaviour■

B.7.4 General Behaviours

We claim that any set of behaviours can be formed from atomic behaviours by applying one or more of the compositions outlined above: simple, concurrent, communicating, consecutive, joined and forked behaviours.

By “any set of behaviours” you may well think of any multi-set of time stamped actions and time stamped events, i.e., of atomic behaviours. From this set one can then “glue” together one or more behaviours first forming a set of simple behaviours; then concurrent behaviours; then identifying possible communicating behaviours; then possibly joining and forking suitable behaviours, etc.

There may very well be many “solutions” to such a “gluing” construction from a basic set of atomic behaviours.

B.7.5 A Model of Behaviours

type

```
ActBeh, EvnBeh,
Beh = ABeh|CBeh,
ABeh = ActBeh|EvnBeh,
CBeh = fSimBeh|ifSimBeh|CurBeh|ComBeh|CnsBeh|FrkBeh|JoiBeh,
fSimBeh == mkSi(s_sb:ABeh*),
ifSimBeh == mkSi(s_sb:ABehω),
CurBeh == mkConc(s_cb:SimBeh-set),
ComBeh == mkComm(s_cb:SimBeh-set),
CnsBeh == mkCons(s_fst:fSimBeh,s_lst:ifSimBeh),
FrkBeh == mkFork(s_fst:fSimBeh,s_lst:ifSimBeh-set),
JoiBeh == mkJoin(s_fst:fSimBeh-set,s_lst:ifSimBeh)
```

value

```
wf_Beh: Beh → Bool
wf_Beh(beh) ≡ ...
```

B.8 Mereology and Compositionality Concluded

B.8.1 The Mereology Axioms

We wish to explain the compositionality constructs of simple entities (Sect. B.8.2), operations (Sect. B.8.3), events (Sect. B.8.4) and behaviours (Sect. B.8.5), where the references are to sections where the compositionality constructs are informally summarised. We wish that the explanation be in terms of the predicates of known axiomatisations of mereology, that is, of proposed such mereologies. We refer to Appendices B.12.1 on page 194 and B.12.2 on page 196

where such predicates are brought forward. Let x, y , and z denote “first class” entities. Then:

1. $\mathcal{P}xy$ expresses that x is a part of y ;
2. $\mathcal{PP}xy$ expresses that x is a proper part of y ;
3. $\mathcal{O}xy$ expresses that x and y overlap;
4. $\mathcal{U}xy$ expresses that x and y underlap;
5. $\mathcal{C}xy$ expresses that x is connected to y ;
6. $\mathcal{DC}xy$ expresses that x is disconnected from y ;
7. $\mathcal{DR}xy$ expresses that x is discrete from y ;
8. $\mathcal{TP}xy$ expresses that x is a tangential part of y ; and
9. $\mathcal{NTP}xy$ expresses that x is a non-tangential part of y .

B.8.2 Composite Simple Entities

Mereology

The part-whole mereological relations of composite simple entities are typically expressed by such defining phrases as: (i) “An x consists of a set of ys ” (modelled by $X=Y\text{-set}$); (ii) “an x consists of a grouping of a y , a z , ... and a u ” (modelled by $X=Y \times Z \times \dots \times U$); (iii) “an x consists of a list of ys ” (modelled by $X=Y^*$); (iv) “an x consists of an association of ys to zs ” (modelled by $X=Y \xrightarrow{m} Z$); and some more involved phrases, including recursively expressed ones.

Usually such defining phrases define too much. In such cases further sentences are needed in order to properly delimit the class of xs being defined.

Example: 14. A bus time table lists the bus line name 15. and one or more named journey descriptions, that is, journey names are associated with (maps into) journey descriptions. 16. Bus line and journey names are further undefined. 17. A journey description sequence of two or more bus stop visits. 18. A bus stop visit is a triple: the name of the bus stop, the arrival time to the bus stop, and the departure time from the bus stop. 19. Bus stop names are hub identifiers. 20. A bus time table further contains a description of the transport net. 21. The description of the transport net of the transport net. associates (that is, maps) each bus stop name hub identifier to a set of one or more bus stop name hub identifiers. 22. A bus time table is well-formed iff 23. adjacent bus stop visits name hubs that are associated in the transport net description; 24. arrival times are before departure times; etc.

type

16. BLNm, JNm
- 14., 20. $\text{BTT}' = \text{BLNm} \times \text{NmdBusJs} \times \text{NetDescr}$
22. $\text{BTT} = \{ | \text{btt} : \text{BTT}' \bullet \text{wf_BTT}(\text{btt}) \mid \}$
15. $\text{NmdBusJs} = \text{JNm} \xrightarrow{m} \text{BusJ}$
17. $\text{BusJ} = \text{BusStopVis}^*$
18. $\text{BusStopVis} = \text{Time} \times \text{HI} \times \text{Time}$

21. $\text{NetDesr} = \text{HI} \multimap \text{HI-set}$
value
 22. $\text{wf_BTT}: \text{BTT} \times \text{NetDesr} \rightarrow \text{Bool}$
 $\text{wf_BTT}(_, \text{jrs}, \text{nd}) \equiv$
 $\forall \text{bj}:\text{BusJ} \bullet \text{bj} \in \text{rng jrs} \Rightarrow$
 $\forall (\text{at}, \text{hi}, \text{dt}):\text{BusStopVis} \bullet (\text{at}, \text{hi}, \text{dt}) \in \text{elems bj} \Rightarrow$
 $\text{hi} \in \text{dom nd} \wedge \text{at} < \text{dt} \wedge \dots$

The well-formedness predicate expresses part of the mereology of how bus time tables are composed. Note that we have not said that net description is commensurate with the actual transportation net ■

That is, we go from regular via context free to context sensitive and even generally computable or, alas, not necessarily computable forms. Thus there are rich opportunities to study suitable subsets of natural language mereology descriptions.

For the specific example of transportation nets, and as formalised in Sect. B.1, we can prove that the following axiom system predicates hold as theorems:

- $\mathcal{PP}xy$ (Item 2 on the facing page) holds for x =links or hubs of net, n , and $y=n$;
- $\mathcal{C}xy$ (Item 5 on the preceding page) holds for such links x which connect hubs y , respectively such hubs x from which links y emanate; and
- $\mathcal{TP}xy$ (Item 8 on the facing page) holds for such links x which connect hubs y , respectively such hubs x from which links y emanate.
- Let us introduce a notion of link/hub connectors. Any link x which is incident upon a hub y is said to define a connector $j : \mathcal{J}$ such that for

type J
value $\text{obs_J}: (\text{L}|\text{H}) \rightarrow \text{J-set}$
axiom
 $\forall \text{l}:\text{L}, \text{h}:\text{H} \bullet \text{card obs_J}(\text{l})=2 \wedge$
 $\text{obs_J}(\text{l}) \cap \text{obs_J}(\text{h}) \neq \{\} \Rightarrow \text{obs_HIs}(\text{l}) \cap \text{obs_HI}(\text{h}) \neq \{\}$

Now let x be the connector of link y and hub z , then $\mathcal{O}xy$ and $\mathcal{O}xz$ (Item 3 on the preceding page) hold.

- Etcetera.

Compositionality

The conventional compositionality principle implied a syntax of composite expressions and spoke of the semantics of composite expressions. We extend this principle to cover other than utterings of natural or formal specification and programming languages. We extend the principle to cover any structures that we may wish to contemplate.

To get the reader “tuned” to that idea we first give three, perhaps slightly “surprising” examples, and then return to examples in line with the main, the transport, example of this paper.

Example: *The design of a bread-toaster denotes the infinite set of all bread-toasters that satisfy the design, or the infinite set of all the production processes that construct such bread toasters, etc.* ■

Example: *The request from the marketing department of the producer of the bread toaster to the design department suggesting a bread toaster that satisfies certain market requirements denote the set of all bread-toaster designs that satisfy these market requirements, etc.* ■

Example: *The request from executive management to the marketing department requesting that measures be taken too win market share denote, amongst others, the kind of requests alluded to in the previous example* ■

Now to the examples that fit into the main example of this paper.

Examples: (i) *A meaning of a link could be the set of all paths that vehicles can traverse the link*, where a link path could be modelled as a triple of link connected hub identifiers and the link identifier. (ii) *A meaning of a hub could be the set of all paths that vehicles can traverse the hub*, where a hub path could be modelled as a triple of link identifiers and the connecting hub identifier; (iii) *A meaning of a net could be the set of all routes through the net*, where a route is a suitable sequence of either link paths or of hub paths ■

Compositionality of Simple Composite Entities: The meaning of atomic entities are expressed by simple (recursive) functions.

The meaning of composite entities, in order to follow the principle of compositionality, must be a function of the meanings of the immediate sub-entities. Here the possibilities are “ad-infinitum” ! Classically, in computing, the principle of compositionality was first applied to programming, then to specification languages. Typically the meaning of an atomic statement, as a syntactic simple entity, of an imperative programming language was that of a function from environments to state to state transformers, so the meaning of the composition of two or more statements was then the function composition of the meaning of each statement. The meaning of a logic program could be modelled as a set of resolutions: bindings of identifiers to terms. The meaning of a parallel, say CSP [190, Hoare], program can be denotationally, that is, according to the principle of compositionality, for example, given either one of three kinds of semantics: in terms of traces, in terms of failures, and in terms of divergences — as all explained in [266, Roscoe, Chapter 8].

• • •

Whether all reasonably expressed meanings of all conceivable composite simple entities can be expressed compositionally is not known, well, is not knowable.

B.8.3 Composite Operations

Mereology

It appears that \mathcal{H} (in $\mathcal{G}(e_1, e_2, \dots, e_m) = \mathcal{H}(\mathcal{G}(e_1), \mathcal{G}(e_2), \dots, \mathcal{G}(e_m))$), and the way in which \mathcal{G} distributes over (e_1, e_2, \dots, e_m) , in the abstract expresses the mereology of function composition. In the concrete the mereological nature of a given composite operation is reflected in the way in which it is structured from primitive recursive functions and other composite operations.

In the sense of recursive function theory it does not seem to make any sense to apply any of the 9 operators (Page 178) of Sect. B.8.1.

Compositionality

The compositionality of functions is here taken to be expressed by the function composers of extended recursive function theory. Extended recursive function theory defines (i) constant entities, $f()$ (i.e., values as zero-ary functions); (ii) variables, v (as functions from an environment to an entity); (iii) sets $\{e_1, e_2, \dots, e_n\}$, Cartesians (e_1, e_2, \dots, e_n) , and lists $\langle e_1, e_2, \dots, e_n \rangle$ of entities; (iv) a finite set of primitive functions, f_n , of arity n and type $f_n: \text{Entity}^* \rightarrow \text{Entity}$ (v) a finite set of primitive predicates, p_n , of arity n and type $p_n: \text{Entity}^* \rightarrow \text{Bool}$ (vi) function composition $f(g(e_1, e_2, \dots, e_n))$, (vii) conditionals $(c_1 \rightarrow e_1, c_2 \rightarrow e_2, \dots, c_n \rightarrow e_n)$, (viii) ..., and (ix) Recursive function theory then tells us how to interpret these forms in some context ρ which binds variables to entities and function and predicate function symbols to their functions and predicates. The extended recursive function theory is a simple encoding from recursive function theory. So compositionality of operations is explained by recursive function theory.

B.8.4 Composite Events

Mereology

Mereologically events can be (i) sequenced, “connected” in time; (ii) “recursively” embedded, with the time interval for an “outer”, embedding event embracing the time interval for an immediately embedded event, and so forth; or can be (iii) clustered.

For specific, formalised examples of the three kinds of events it may then be possible to prove the following: (i) $\mathcal{PP}xy$ where x is an event of either a sequenced event, or of a recursively embedding, or of a clustered event y (x is embedded in the recursively embedding event y); (ii) $\mathcal{C}xy$ where x and

y are two consecutive events of a sequenced event z (that is, (ii:A) $\mathcal{PP}xz \wedge \mathcal{PP}xz$, (ii:B) $\mathcal{TP}xy$ and (ii:C) $\mathcal{U}xy$ (of z)), (ii:D) while $\mathcal{DC}uv$ where $(\mathcal{PP}uz \wedge \mathcal{PP}vz) \wedge \sim \mathcal{C}uv$; (iii) $\mathcal{O}xy$ may hold for x and y being part of a clustered event z (that is, $\mathcal{PP}xz \wedge \mathcal{PP}xz$ and $\mathcal{U}xy$ (of z)). Etcetera.

Compositionality

Please note that we are not giving meaning to syntactic designators whose meaning is that of events. We are confronted with the issue of giving meaning, in some sense, to events. Simple such meanings are concerned with concrete event analysis: did to events occur at the same time, or in a given time period, or one before the other. For such analyses we refer to [216, Lamport]. We can think of obtaining more elaborate meanings for sequenced and recursively embedded events, but first we would need to build up a rather elaborate set of definitions, find elaborate examples, etcetera, and that would blow the size of this paper way out of proportion. So the best is to say, and this also applies to clustered events, here is an interesting research topic: to come up with compositionality interpretations for composite events !

B.8.5 Composite Behaviours

Mereology

Let $\{\beta_1, \beta_2, \dots, \beta_n\}$ be, for each case below, the suitable set of (simple) behaviours. From the point of view of the mereology of the composition of behaviours, such as we have modelled behaviours, there are the following composition operators:

- (i) creating simple behaviours, $\beta_s: (\text{fSimBeh} | \text{ifSimBeh})$, from suitable atomic ones $(a_1, a_2, \dots, a_m, e_1, e_2, \dots, e_n)$;
- (ii) creating concurrent behaviours, $\beta_{con}: \text{CurBeh}$, from suitable simple behaviours;
- (iii) creating communicating behaviours, $\beta_{com}: \text{ComBeh}$, from a set of suitable simple behaviours;
- (iv) creating consecutive behaviours, $\beta_{seq}: \text{CnsBeh}$, from a set of suitable simple behaviours;
- (v) creating joined behaviours $\beta: \text{JoiBeh}$ from a set of suitable simple behaviours; and
- (vi) creating forked behaviours, $\beta: \text{FrkBeh}$, from a set of suitable simple behaviours.

Given a specific composite behaviour is may then be possible to prove

- (i) For all distinct atomic behaviours $\beta'_{\alpha\epsilon}$ and $\beta''_{\alpha\epsilon}$ and for all simple behaviours, β_s , for which $\mathcal{P}\beta'_{\alpha\epsilon}\beta_s \wedge \mathcal{P}\beta''_{\alpha\epsilon}\beta_s$ holds to prove that $\mathcal{PP}\beta'_{\alpha\epsilon}\beta_s \wedge \mathcal{PP}\beta''_{\alpha\epsilon}\beta_s$; holds;

- (ii) for a set, βs , of suitable simple behaviours to prove that $\mathcal{PP}\beta_i\text{mkConcurBeh}(\beta s)$ holds for any β_i in βs ;
- etcetera.

Compositionality

Behaviours are the meaning of domain descriptions, or or requirements prescriptions or software programs that specify concurrency. So the meaning functions that we might apply to behaviours would then typically be those of analysing behaviours: comparing behaviours, say with respect to efficiency, or to access to shared resources, or (say human) interface response times. We leave it to the reader to continue this line of thought.

B.9 Galois Connections⁹

In the following sections, we look at some rôles Galois connections may have in relation to composition of entities. Galois connections is a fundamental mathematical concept from order- and lattice theory. The reason for bringing it up here is that it involves set-based composition of elements as well as the composition of dual, order-decreasing functions. However, this is just the reason why we considered Galois connections in the first place. In fact we want to argue that it is beneficial to incorporate non-traditional modelling aspects in order to get more insight; both in the discipline of domain engineering, and in a current domain of discourse.

In the following, we shall (i) define the notion of Galois connections, (ii) outline some of the different uses of that concept, and (iii) consider the concept in context of modelling composite entities (following the ontology presented in this paper). The sections are driven by examples.

Example: Toasters and Their Designs. *Let $(d:D)$ be a design of a toaster $(t:T)$. From the design we may be able to produce a collection of different toasters because the design does not specify everything, and due to the fact that we could produce the “same” kind of toaster over and over again. Let us look at a “time glimp” and let $(ts:T\text{-set})$ denote the set of such toasters obeying the design¹⁰. If we impose that “sequentially” further designs are all for the same toaster, then the number of toasters decreases¹¹ because they all*

⁹This section is main-authored by Asger Eir.

¹⁰We shall — as common in modelling — assume a *possible worlds semantics* in the sense that the collection of toasters are the toasters existing in one possible world. Are there more produced or some destroyed, it is another possible world. We shall not be further concerned with this, nor the many philosophical issues that can be claimed. We refer to [275] and [10] which among many other issues take up this discussion.

¹¹Actually, the number could stay the same but that would mean including identical designs. In general, we shall not be that concerned with the equal-situation for that same reason.

need to satisfy the new designs too. Between the set of designs and the set of toasters they denote, is a Galois connection ■

Example: Designs and Market Analysis. The designs are also the denotation of something. It could be the market analysis indicating the need for certain toaster products — or more generally, for certain new kinds of kitchen equipment. Between the market analysis and the designs also stands a Galois connection; hence there is also a Galois connection between the market analysis and the toasters. The Galois connection (being an order-decreasing pair of functions) ensures that we can only produce toasters which obeys the designs, and that we can only design toasters which satisfy the needs outlined in the market analysis ■

B.9.1 Definition

Definition B.1. A Galois connection is a dual pair of mappings $(\mathcal{F}, \mathcal{G})$ between two orderings (P, \sqsubseteq) and (Q, \sqsubseteq) . Most often P and Q are ordered sets based on set-inclusion (\subseteq) and this is also the version we shall use in this paper. In order to avoid misinterpretation, we write \sqsubseteq and not \leq or \subseteq as seen in other treatments of the subject. The mappings must be monotonically decreasing¹²:

type

P, Q

value

$\mathcal{F}: P\text{-set} \rightarrow Q\text{-set}$

$\mathcal{G}: Q\text{-set} \rightarrow P\text{-set}$

axiom

$\forall ps_1, ps_2: P\text{-set}, qs_1, qs_2: Q\text{-set} \bullet$

$ps_1 \sqsubseteq ps_2 \Rightarrow \mathcal{F} ps_2 \sqsubseteq \mathcal{F} ps_1,$

$qs_1 \sqsubseteq qs_2 \Rightarrow \mathcal{G} qs_2 \sqsubseteq \mathcal{G} qs_1,$

$ps_1 \sqsubseteq \mathcal{G} \mathcal{F} ps_1,$

$qs_1 \sqsubseteq \mathcal{F} \mathcal{G} qs_1$

The dual ordering of Galois connections is illustrated on Figure B.2. In [164, Ganter & Wille: FCA], the following Theorem is given on Galois connections:

¹²Note, that there are in fact two different definitions of Galois connections in the literature: the *monotone* Galois connection and the *antitone* Galois connection. We follow Ganter and Wille and assume the former [164].

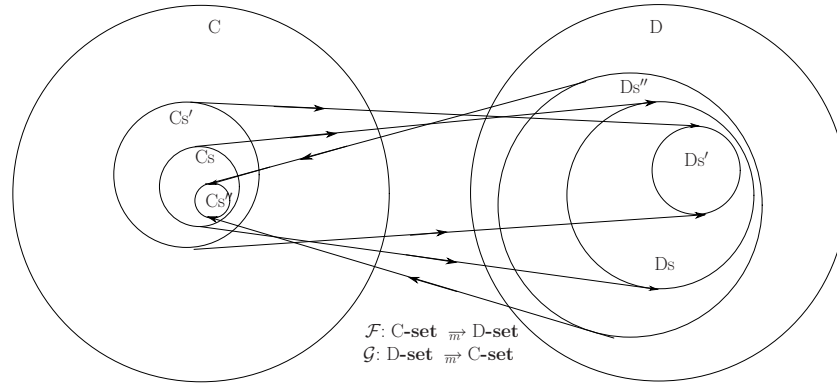


Fig. B.2. A Galois connection. ■

Theorem B.2 (Galois Connection¹³). For every binary relation $R \subseteq M \times N$, a Galois connection (φ_R, ψ_R) between M and N is defined by

$$\begin{aligned}\varphi_R X &:= X^R (= y \in N \mid xRy \text{ for all } x \in X) \\ \psi_R Y &:= Y^R (= x \in M \mid xRy \text{ for all } y \in Y).\end{aligned}$$

From the above, we see that all y must stand in the relation R to each x in order for the connection to hold. However, R could mean “*does not stand in a relation to*”. That would still yield a Galois connection but the domain knowledge it expresses is different. Let X be a collection of coffee cups and let Y be a collection properties concerning form, colour, texture and material. We may define R to be “*coffee cup x has property y* ”. However, we could also define it as “*coffee cup x does not have property y* ”. In both cases we would have a Galois connection. However, the latter may be somewhat strange from a classification point of view.

The notion of Galois connections has served as foundation for a variety of applications like order theory, the theory of dual lattices, and — in computer science — semantics of programming languages and program analysis.

However, it has also been utilized in a number of conceptualization principles. These principles are not pure mathematical treatments, but utilize Galois connections in specific domains. We shall look at three such areas in the following.

B.9.2 Concept Formation in Formal Concept Analysis [FCA]

In the area of formal concept analysis [164, Ganter & Wille: FCA], the notion of Galois connections is used as foundation for the lattice-oriented theory

¹³In [164] this is named Theorem 2.

used for concept formation. In FCA, concepts are defined from a collection of objects by looking at which objects have common properties. The approach includes algorithms for automatic concept formation, given a collection of objects or a collection of properties. The fact that we can choose either to form concepts from objects (the extension of the concepts) or the properties (the intensions of the concepts) shows the duality between objects and properties.

B.9.3 Classification of Railway Networks

In [196–198] Ingleby et al. use Galois connections in order to classify railway networks. The approach is similar to the approach of concept formation in FCA, but Ingleby understands the notion of properties in a broader sense: a property of a route may be the segments involved in the route. Here Ingleby understands routes and segments in a safety–security sense as his quest is to cluster routes and segments such that the complexity of safety proof over the railway network, is reduced. That is, the Galois connection is used for defining cluster segments (in FCA, corresponding to concepts) such that the number of free variables are reduced when proving safety properties of software/hardware for instance.

B.9.4 Relating Domain Concepts Intensionally

In [152, 153, Eir], we utilized the notion of Galois connections for relating domain concepts intensionally. The domain concepts related were concepts that were not bound under subsumption; i.e. they are not specialization/generalization pairs.

Consider the domain concepts: *Budgets* and *Project Plans*. From a budget we can observe the set of project plans that can be executed within the financial restrictions of the budget. From a project plan we can observe the set of budgets that designate the necessary figures for executing the project plan. Generalizing this gives two interpretation functions: one from a set of budgets to the set of project plans that are all executable within the restriction of each budget in the set; and another from a set of project plans to the set of budgets that all designate the necessary expenses for executing each project plan.

The pair of interpretation functions is a Galois connection. This approach is utilized in order to suggest a modelling approach for relating domain concepts and placing their models (i.e. their abstractions) in conceptual structures. For the two concepts mentioned above, the conceptual structure maintains the systematics of concretising information from budgeting to project planning.

B.9.5 Further Examples

We may easily produce other examples of domain concept pairs of which the objects relate in some way. Consider the following examples:

Example: Bus Time Tables and Traffic. Let btt be some bus time table ($btt:(bln, busjs, nd)$). To btt there corresponds a set of bus traffics, $sobustrfs$, on the net. Express such bus traffic as $(bustrf, n)$ where $(bustrf, n) \in sobustrfs$ and where $bustrf$ is the time-varying function from buses to their positions on the net, and nd is related to n in some way (one is a net description, the other is “the” (or that) net). We furthermore stipulate that each bus traffic $(bustrf, n)$ “obeys” the timetable $(bln, busjs, nd)$. To a set of timetables, $sobustts$, over the same net there corresponds the union set of all those sets of bus traffics, $usobustrfs$, that “obey” all timetables in $sobustts$ ■

We seek to understand the relationship between $sobustts$ and $usobustrfs$ in terms of the concept of Galois connections.

Example: Traffic and Buses – The Dual Case. We reverse the relation. We start with a bus traffic $(bustrf, nd)$ and can, by arguments similar to above, postulate a set of bus timetables, $sobustts$ (on the same net), such that each bus timetable properly records the arrival and departure times of buses at bus stops on that net. We can then “lift” this relation $((bustrf, nd), sobustts)$ to a relation from sets of bus traffics to the union set of sets of bus timetables ■

We seek to understand the relationship between $sobustrfs$ and $usobustts$ in terms of the concept of Galois connections.

The two examples above each define what we in B.9.4 called *interpretation functions*. They are interpretation functions in the sense that they — in the domain — “interpret” the time table entities as traffic entities; and vice versa.

B.9.6 Generalisation

The element that these example have in common is that the values of one concept characterize the values of the other concept — in some way.

This is similar to FCA where we have a Galois connection between values and their common properties. However, in this case the properties are extrinsic properties. The budget relates to a specific set of project plans because it possesses the property of standing in a certain relation to these other values. The property is *extrinsic* as the property is possessed assuming the existence of other values; as opposed to *intrinsic* properties.

In a sense it means that we break the traditional distinction between values and properties as assumed in FCA. Furthermore, we utilize the same principles as utilized in denotational semantics — namely that we can assign meaning to values (e.g., a budget) and the meaning to composition of values (e.g., a set of budgets). The meaning of the composition is here more than the meanings of the individual parts because the composition of budgets (the budget set inclusion) implies a more narrow restriction of the set of executable project plans. It is so because combining two budgets has influence on the meaning in the sense that the meaning is the composition of the corresponding project plans as well (satisfying the Galois functions of being “decreasing”).

Mereologically, what is added when composing a whole is actually the axioms in the Galois connection.

However, we go a little further than denotational semantics of programming languages because we may consider any domain concept a subject for defining a Galois connection.

Another observation is that the notion of Galois connection is *domain neutral*. The Galois connection is a general mathematical framework and hence not what contributes to *why* two concepts relate intensionally.

B.9.7 Galois Connections and Ontology

In the ontology presented throughout this paper, we have exercised the importance of compositionality. I.e. we have defined compositionality for each of the four entity parametrisations made. In this sections, we shall look at how these can be understood in the general, domain and ontology neutral framework of Galois connections. When we say that Galois connections in this sense are ontologically neutral it is not entirely true. Many ontologies — especially in philosophy — concerns the existence of (say) mathematical entities; hence also heavily touching (perhaps disturbing) the foundation on which Galois connections are defined. However, this is not our quest here. When we consider Galois connections ontologically neutral it is in fact similar to saying that they are neutral to the ontology of entities that we have suggested. Whether entities include mathematical entities or these is “outside”, that is, is outside the scope of this paper. For further exploration, we refer to [275].

If simple entities, events, behaviour and operations are all entities, it should imply that we can make the same considerations involving such values in Galois connections. We shall try to do so in the following, and we intend in that context to outline the issues as we go along. The important thing is, however, not whether Galois connections can be established, but whether the Galois connection complies with the current intuition as the connection between objects and their common properties does.

The traditional use of Galois connections — as ‘exercised order theory’ and as used by Ganter and Wille focuses on the properties that objects have in *common*. However, we may turn this order upside-down such that we look at the total set of properties. This will be a Galois connection as well but in the case of domains, it expresses a different aspect. In some situations, it is natural to consider the former — in other situations, we may prefer the latter. This depends on how we perceive the domain and — perhaps also — the purpose of our domain model; i.e. our perspective.

By the above we indicate that the study of Galois connections in the context of domain engineering could be interesting because it has to do with how we choose to perceive, abstract, model and formalize the domain. Hence, what we present in the following may open up for such research areas for further clarification. We are, however, not saying that these *will* be interesting

or that it does make sense to make distinctions like the one above. We just say that this area deserves further exploration.

Let us in the following assume that Galois connections concern relations between two ordered sets of entities and the essence that the entities in these sets characterize each other. The issue is now that in some cases, characterization usually obeys the axioms of Galois; but in some situations it may not.

Composite Entities

We have already seen a couple of examples of Galois connections between two ordered sets of elements, where the ordering has been set-inclusion.

We assume the understanding of composite entities as presented in Sects. B.4 and B.8.2.

Example: Hospital Staff and Rostering (I). *Doctors and nurses forming surgery teams. From a team (possibly empty or singleton), we can observe the collection of time slots where they are all available. If we include more doctors and nurses, we will have a smaller set of time slots. And vice versa. This is an important domain aspect when we are going to talk about planning and staffing (either in domain descriptions and specifications, or in software requirements). This is a Galois connection* ■

Example: Hospital Staff and Rostering (II). *Again consider doctors and nurses forming surgery teams. From a team, we can observe the collection of possible surgeries they may perform. If we include more doctors and nurses, we increase the collection of surgeries. And vice versa* ■

This is not a Galois connection, though interesting from a domain perspective anyway.

Composite Operations

We assume the understanding of composite entities as presented in Sects. B.5 and B.8.3.

Example: Building Constructions and Parts. *Consider a set of building constructions: molding of foundations, mounting of bricks into walls, and establishing the roof, etc. Then consider the set of building parts involved in a construction. By building parts, we shall both understand the materials and elements consumed by constructions, and the results of other constructions. Thus, a building part can be a specific brick, a pre-cast concrete wall, the foundation, etc. That is, the building parts are those either created, mounted on, changed in some way, or demolished. For each construction, we can observe the building parts involved: consumed or produced. Building constructions can be composite in the sense that one construction constructs the foundation and another construction mounts the walls on the foundation. The former*

construction is a function from certain amounts of sand, stone, cement and water, to a foundation (here we shall exclude the tools needed). The latter construction is a function from a foundation, a collection of bricks, water, cement and insulation, to the product consisting of foundation and walls. For a construction — atomic or composite — we can observe the building parts involved in all constructions. If we include more constructions in a composite construction, the building parts involved in all constructions will decrease ■

The connection between composite constructions and building parts involved is a Galois connection.

The connection is interesting when modelling the planning and scheduling of construction works as a crucial element is that construction workers cannot always work on the same building parts at the same time.

Example: Building Operations and Consumed Materials. Now consider the approach where for each building operation we observe the materials needed. For a collection of operations, we can likewise observe the total quantity of materials needed. That is, the total amount of sand, stones, bricks; the total quantity of beams, doors, windows of each type and measure; etc. Including more building operation will increase the amount and quantity of materials needed; simply because we then build more. Then we have a situation where the more operations we include, the more products. That is, set-inclusion of operations implies an increase of the observed materials and parts. The reason is that we here that each building operation contributes with a result. Instead of considering the common materials as characterizing the composite operation, we shall consider that the complete set of materials involved characterize the composite operation ■

In a sense this is more natural as we then include all the aspects of the compositionality. However, in the present case, we do not have Galois connection because including more operations in the composite, implies including more materials and results. Hence, dual ordering is increasing; not decreasing.

Composite Events

We assume the understanding of composite entities as presented in Sects. B.6 and B.8.4.

Example: Traffic Accidents and Responsible Persons. Consider a traffic accident. This is an event and for the accident, we can observe the collection of persons involved and of these the persons bearing some kind of responsibility in the accident. Assume that we look at a collection of traffic accidents. Here, we can observe the persons involved in all accidents and for these the ones being responsible for the accidents. Including more traffic accidents will

reduce the number of persons involved in all accidents; hence, also the number of persons being responsible in all accidents ■

The connection between sets of traffic accident events and the set of persons being responsible, is a Galois connection.

The connection may be interesting when modelling the analysis of traffic accident patterns and statistics which may influence the definition of insurance premium.

Example: Traffic Accidents and Persons Involved. *Now, consider traffic accidents as events again. From a traffic accident, we can observe the insurance policies of the involved persons. Likewise, from a collection of traffic accidents (i.e. a composite event being a cluster of individual events), we can observe the collection of persons involved in at least one of the accidents; that is, the total collection of persons involved in one or more of the accidents. If we include more accidents, the collection of persons involved will increase* ■

This is not a Galois connection. Though the connection may be interesting when modelling correlation between accidents.

We should also be able to construct examples for composite events being sequential or embedded.

Composite Behaviours

We assume the understanding of composite entities as presented in Sects. B.7 and B.8.5.

Example: Meetings and Applicable Rooms. *Consider a collection of persons engaged in a meeting. We shall consider having a meeting a behaviour. The meeting can be composite in the sense that we may join two or more meetings held in the same time interval and involving the same persons. In the present case we shall consider behaviour composition as communicating. E.g. we may join department meetings for several company departments if the topic of the meetings is common and should be shared. From a meeting, we can observe the rooms applicable. We shall assume that a room is only applicable if it can host the number of meeting participants, has the equipment necessary for the meeting, etc. If we include more meeting behaviours in a composite behaviour, the collection of rooms applicable will decrease* ■

This is a Galois connection. The connection is interesting when planning collaborative work among meeting participants.

Example: Engineering Work and Skills. *Consider a collection of engineers engaged in a project. We shall consider their work a behaviour which is concurrent — perhaps also communicating to an extent. From each engineering behaviour we can observe the engineering skills utilized and practiced. If*

we include a collection of work behaviours as a composite behaviour, it implies that we include more engineers and thus also more engineering skills ■

This is not a Galois connection; though interesting when modelling skills, skills management, project communication and interaction, staffing, etc.

B.9.8 Galois Connections Concluded

So what went wrong in the cases where we did not have a Galois connection? Or we could ask: what did we explore by looking at the domain through Galois eyes? The examples examined above clearly shows that there are two different kinds of connections between entity compositions; hence, orderings.

- The former yields a Galois connection. It does so because composite entities of the one ordering are *all* characterizing composite entities of the other. Thereby, we believe to have outlined how Galois connections and ordering theory in general plays an important rôle in compositionality of entities.
- The latter does not yield a Galois connection as it is an order-preserving connection. In the examples examined we have seen a general pattern composition of the one kind of entity, yields composition (actually just set-inclusion) of the other kind of entity.

Both kinds of connections show that even though the connections (Galois being order-reversing and the order-preserving) are ontologically and domain neutral, they do express interesting domain intrinsics when it comes to compositionality. We suggest that the rôle, use and axioms/theorems of such ordering connections are explored further within the context of domain engineering. Furthermore, we encourage exploring other such concepts and their ability of promoting domain engineering as a discipline.

B.10 Conclusion

B.10.1 Ontology

Ontology plays an important rôle in studies of epistemology and phenomenology. In the time-honoured tradition of philosophical discourse philosophers present proposals for one or another ontology, and discusses these while usually not settling definitively on any specific ontology; and many issues are deliberately left open.¹⁴ In this paper we cannot afford this “luxury”. Our objective is to clarify notions of ontology in connection with the use of specific ways of informally and formally describing domains where the formal description language is fixed.

¹⁴Such as whether properties of entities are themselves entities, etc.

Many of the issues of domain modelling evolve close to issues of metaphysics. We find [222, Michael J. Loux] *Metaphysics, a contemporary introduction*, [170, Pierre Grenon and Barry Smith] *SNAP and SPAN: Towards Dynamic Spatial Ontology*, [276, Peter Simons] *Parts: A Study in Ontology*, and [231, D. H. Mellor and Alex Oliver] *Properties*, relevant for a deeper study of the meta-physical issues of the current essay.

B.10.2 Mereology

Mereology has been given a more concrete interpretation in this paper compared to the “standard” treatments in the (mostly philosophical) literature. It seems that Douglass T. Ross [267] was among the first computing scientists to see the relevance of Leśniewski’s ideas [225, 285]. Too late for a study we found [252, Chia-Yi Tony Pi]’s 287 page PhD (linguistics) thesis: *Mereology in Event Semantics*. Perhaps it is worth a study.

B.10.3 Research Issues

The paper has touched upon many novel issues. Some are reasonably well established, at least from a programming methodological point of view. Several issues could benefit from some deeper study. We mention three.

Compositionality

A precise study of how composite functions, events and behaviours can be understood according to the principle of compositionality.

Mereology

A more precise presentation of a mereology axiom system for the kind of simple entities, function entities, event entities and behaviour entities outlined in Sects. B.4–B.7.

Ontology

A more precise comparison of the “computability”-motivated ontology of this paper as compared with for example the ontological systems mentioned in [222, Michael J. Loux], [170, Pierre Grenon and Barry Smith], [276, Peter Simons] and [160, Chris Fox].

Galois Connections

A further study, going beyond that of [152, 153, Asger Eir], of relations between compositionally and **Galois connections**. For that study one should probably start with [188, Hoare and He].

• • •

That we have not really studied the compositionality issue as listed above is a major drawback of this paper but we needed to clarify first the nature of “compositeness” of events, functions and behaviours before taking up the future study of their compositionality.

B.10.4 Acknowledgement

The first author is most grateful to his former PhD student, Dr. Asger Eir, for his willingness to co-author this paper.

B.11 Bibliographical Notes

[94, to appear] gives a concise overview of domain engineering; [95, to appear] gives a “complete” example of domain and requirements engineering; and [93, to appear] relates domain engineering, requirements engineering and software design to software management. [90] presents a number of domain engineering research challenges.

B.12 Two Axiom Systems for Mereology

B.12.1 Parts and Places¹⁵

A mereological system requires at least one primitive binary relation (dyadic predicate). The most conventional choice for such a relation is *Parthood* (also called “inclusion”), “ x is a part of y ,” written:

$$\mathcal{P}xy$$

Nearly all systems require that *Parthood* partially order the universe. The following defined relations, required for the axioms below, follow immediately from *Parthood* alone:

An immediate defined predicate is “ x is a proper part of y ,” written $\mathcal{PP}xy$, which holds (i.e., is satisfied, comes out true) if $\mathcal{P}xy$ is true and $\mathcal{P}yx$ is false. If *Parthood* is a partial order, *ProperPart* is a strict partial order:

¹⁵This section is based on [135].

$$\mathcal{P}\mathcal{P}xy \leftrightarrow (\mathcal{P}xy \wedge \sim \mathcal{P}xy) \quad (\text{B.1})$$

An object lacking proper parts is an atom. The mereological universe consists of all objects we wish to think about, and all of their proper parts:

Overlap: x and y overlap, written $\mathcal{O}xy$, if there exists an object z such that $\mathcal{P}zx$ and $\mathcal{P}zy$ both hold.

$$\mathcal{O}xy \leftrightarrow \exists z[\mathcal{P}zx \wedge \mathcal{P}zy] \quad (\text{B.2})$$

The parts of z , the “overlap” or “product” of x and y , are precisely those objects that are parts of both x and y .

Underlap: x and y underlap, written $\mathcal{U}xy$, if there exists an object z such that x and y are both parts of z .

$$\mathcal{U}xy \leftrightarrow \exists z[\mathcal{P}xz \wedge \mathcal{P}yz] \quad (\text{B.3})$$

Overlap and Underlap are reflexive, symmetric, and intransitive.

Systems vary in what relations they take as primitive and as defined. For example, in extensional mereologies (defined below), Parthood can be defined from Overlap as follows:

$$\mathcal{P}xy \leftrightarrow (\mathcal{O}zx \rightarrow \mathcal{O}zy) \quad (\text{B.4})$$

The Axioms

Parthood partially orders the universe:

M1, Reflexive: An object is a part of itself.

$$\mathcal{P}xx \quad (\text{B.5})$$

M2, Antisymmetric: If $\mathcal{P}xy$ and $\mathcal{P}yx$ both hold, then x and y are the same object.

$$(\mathcal{P}xy \wedge \mathcal{P}yx) \rightarrow x = y \quad (\text{B.6})$$

M3, Transitive: If $\mathcal{P}xy$ and $\mathcal{P}yz$, then $\mathcal{P}xz$.

$$(\mathcal{P}xy \wedge \mathcal{P}yz) \rightarrow \mathcal{P}xz \quad (\text{B.7})$$

M4, Weak Supplementation: If $\mathcal{P}\mathcal{P}xy$ holds, there exists a z such that $\mathcal{P}zy$ holds but $\mathcal{O}zx$ does not.

$$\mathcal{P}\mathcal{P}xy \rightarrow \exists z[\mathcal{P}zy \wedge \sim \mathcal{O}zx] \quad (\text{B.8})$$

M5, Strong Supplementation: Replace “ $\mathcal{P}\mathcal{P}xy$ holds” in M4 with “ $\mathcal{P}yx$ does not hold.”

$$\sim \mathcal{P}yx \rightarrow \exists z[\mathcal{P}zy \wedge \sim \mathcal{O}zx] \quad (\text{B.9})$$

M5', Atomistic Supplementation: If $\mathcal{P}xy$ does not hold, then there exists an atom z such that $\mathcal{P}zx$ holds but $\mathcal{O}zy$ does not.

$$\sim \mathcal{P}xy \rightarrow \exists z[\mathcal{P}zx \wedge \sim \mathcal{O}zy \wedge \sim \exists x[\mathcal{P}\mathcal{P}vz]] \quad (\text{B.10})$$

Top: There exists a “universal object”, designated Ω , such that $\mathcal{P}x\Omega$ holds for any x .

$$\exists \Omega \forall x[\mathcal{P}x\Omega] \quad (\text{B.11})$$

Bottom: There exists an atomic “null object”, designated $\mathcal{U}\text{oid}$, such that $\mathcal{P}\mathcal{U}\text{oid}x$ holds for any x .

$$\exists \mathcal{U}\text{oid} \forall x[\mathcal{P}\mathcal{U}\text{oid}x] \quad (\text{B.12})$$

M6, Sum: If $\mathcal{U}xy$ holds, there exists a z , called the “sum” or “fusion” of x and y , such that the parts of z are just those objects which are parts of either x or y .

$$\mathcal{U}xy \rightarrow \exists z \forall v[\mathcal{O}vz \leftrightarrow (\mathcal{O}vx \vee \mathcal{O}vy)] \quad (\text{B.13})$$

M7, Product: If $\mathcal{O}xy$ holds, there exists a z , called the “product” of x and y , such that the parts of z are just those objects which are parts of both x and y .

$$\mathcal{O}xy \rightarrow \exists z \forall v[\mathcal{P}vz \leftrightarrow (\mathcal{P}vx \wedge \mathcal{P}vy)] \quad (\text{B.14})$$

If $\mathcal{O}xy$ does not hold, x and y have no parts in common, and the product of x and y is defined iff Bottom holds.

M8, Unrestricted Fusion: Let $\phi(x)$ be a first-order formula in which x is a free variable. Then the fusion of all objects satisfying ϕ exists.

$$\exists xz[\phi(x) \rightarrow \forall y[\mathcal{O}yz \leftrightarrow (\phi(x) \wedge \mathcal{O}yx)]] \quad (\text{B.15})$$

M8', Unique Fusion: The fusions whose existence M8 asserts are also unique.

M9, Atomicity: All objects are either atoms or fusions of atoms.

$$\exists yz[\mathcal{P}yx \wedge \sim \mathcal{P}\mathcal{P}zy] \quad (\text{B.16})$$

B.12.2 From: A Calculus of Individuals Based on ‘Connection’¹⁶

Taking $\mathcal{C}xy$ as a rendering of x is connected to y we can introduce a definition of $\mathcal{DC}xy$ (x is disconnected from y) and the standard mereological definitions of $\mathcal{P}xy$ (x is a part of y), $\mathcal{P}\mathcal{P}xy$ (x is a proper part of y), $\mathcal{O}xy$ (x overlaps y), and $\mathcal{DR}xy$ (x is discrete from y) as follows:

¹⁶This section is taken from [140].

$$\mathcal{DC}xy \equiv \sim \mathcal{C}xy \quad (\text{B.17})$$

$$\mathcal{P}xy \equiv (\forall z)(\mathcal{C}zx \supset \mathcal{C}zy) \quad (\text{B.18})$$

$$\mathcal{PP}xy \equiv \mathcal{P}xy \wedge \sim \mathcal{P}yx \quad (\text{B.19})$$

$$\mathcal{O}xy \equiv (\exists z)(\mathcal{P}zx \wedge \mathcal{P}xy) \quad (\text{B.20})$$

$$\mathcal{DR}xy \equiv \sim \mathcal{O}xy \quad (\text{B.21})$$

This distinction between $\mathcal{C}xy$ and x , constitutes the virtue of this new calculus. It gives us the power to define $\mathcal{EC}xy$ (x is externally connected to y), $\mathcal{TP}xy$ (x is a tangential part of y), and $\mathcal{NTP}xy$ (x is a non-tangential part of y) as follows:

$$\mathcal{EC}xy \equiv \mathcal{C}xy \wedge \sim \mathcal{C}xy \quad (\text{B.22})$$

$$\mathcal{TP}xy \equiv (\forall z)(\mathcal{EC}zx \wedge \mathcal{EC}xy) \quad (\text{B.23})$$

$$\mathcal{NTP}xy \equiv \mathcal{P}xy \wedge \sim (\forall z)(\mathcal{EC}zx \wedge \mathcal{EC}xy) \quad (\text{B.24})$$

Our axiomatization requires only two axioms: a mereological axiom,

$$(\forall x)[\mathcal{C}xx \wedge (\forall y)\mathcal{C}xy \supset \mathcal{C}yx] \quad (\text{B.25})$$

and an axiom involving identity, analogous to the axiom of extension in set theory,

$$(\forall x)(\forall y)[(\forall z)(\mathcal{C}zx = \mathcal{C}zy) \supset x = y]. \quad (\text{B.26})$$

C

Domain Theory: Practice and Theories A Discussion of Possible Research Topics

This appendix chapter constitutes the invited paper for ICTAC 2007, The 4th International Colloquium on Theoretical Aspects of Computing [90], 26–28 September, Springer 2007, Macau SAR, China. Permission to bring this paper here is being applied for.

Maybe the title of the paper need be explained: The second part of the title: ‘Practice and Theories’ shall indicate that there is an engineering practice (i.e., methodology) of developing domain descriptions and that any such domain description forms the basis for a specific domain theory. The first part of the title: ‘Theories’ shall indicate that we need support the practice, i.e., the methodology, by theoretical insight, and that there probably are some theoretical insight that applies across some or all domain theories.

C.1 Introduction

C.1.1 A Preamble

This paper is mostly a computing science paper. This paper is less of a computer science paper. Computer science is the study and knowledge about the “things” that can exist “inside” computers, and of what computing is. Computing science is the study and knowledge about how to construct computers and the “things” that can exist “inside” computers. Although the main emphasis of ‘Domain Theory and Practice’ is computing science, some of the research topics identified in this paper have a computer science nature.

C.1.2 On Originality

This paper is an invited paper. It basically builds on and extends a certain part (Part IV Domain Engineering) of Vol. 3, [89], of my book [87–89],

I shall therefore not bring a lot of motivation nor put my possible contributions in a broader context other than saying this: as far as I can see from the literature my concept of domain engineering is new. It may have appeared in rudimentary forms here and there in the literature (as from JSL and then notably in Michael Jackson's work), but in the nine chapters (Chaps. 8–16) of Part IV, [89], it receives a rather definitive and fully comprehensive treatment. But even that treatment can be improved. The present paper is one such attempt.

C.1.3 Structure of Paper

In a first semi-technical section we briefly express the triptych software engineering dogma, its consequences and its possibilities. We relate software verification to the triptych and present a first research topic. Then we list some briefly explained domains, and we present three more research topics. In the main technical section of this paper we present five sets of what we shall call domain facets (intrinsic, support technology, management and organisation, rules and regulations, and human behaviour). Each of these will be characterised but not really exemplified. We refer to [89] for details. But we will again list corresponding research topics. The paper ends first with some thoughts about what a 'domain theory' is, then on relations to requirements, and finally on two rather distinct benefits from domain engineering. In that final part of the paper we discuss a programming methodology notion of 'requirements specific development models' and its research topics.

C.2 Domain Engineering: A Dogma and its Consequences

C.2.1 The Dogma

First the dogma: Before software can be designed its requirements must be understood. Before requirements can be prescribed the application domain must be understood.

C.2.2 The Consequences

Then the "idealised" consequences: In software development we first describe the domain, then we prescribe the requirements, and finally we design the software. As we shall see: major parts of requirements can be systematically "derived"¹ from domain descriptions. In engineering we can accommodate for less idealised consequences, but in science we need investigate the "ideals".

¹By "derivation" we here mean one which is guided by humans (i.e., the domain and requirements engineers in collaboration with the stakeholders).

C.2.3 The Triptych Verification

A further consequence of this triptych development is that

$$\mathcal{D}, \mathcal{S} \models \mathcal{R},$$

which we read as: in order to prove that *Software* implements the *Requirements* the proof often has to make assumptions about the *Domain*.

C.2.4 Full Scale Development: A First Suggested Research Topic

Again, presupposing much to come we can formulate a first research topic.

- ℜ 1. **The $\mathcal{D}, \mathcal{S} \models \mathcal{R}$ Relation:** Assume that there is a formal description of the *Domain*, a formal prescription of the *Requirements* and a formal specification of the *Software* design. Assume, possibly, that there is expressed and verified a number of relations between the *Domain* description and the *Requirements* prescription. Now how do we express the assertion: $\mathcal{D}, \mathcal{S} \models \mathcal{R}$ — namely that the software is correct? We may assume, without loss of generality, that this assertion is in some form of a pre/post condition of \mathcal{S} — and that this pre/post condition is supported by a number of assertions “nicely spread” across the *Software* design (i.e., the code). The research topic is now that of studying how, in the pre/post condition of \mathcal{S} (the full code) and in the (likewise pre/post condition) assertions “within” \mathcal{S} , the various components of \mathcal{R} and \mathcal{D} “appear”, and of how they relate to the full formal pre- and descriptions, respectively.

C.2.5 Examples of Domains

The Examples

Lest we loose contact with reality it is appropriate here, however briefly, to give some examples of (application) domains.

Air Traffic: A domain description includes descriptions of the entities, functions, events and behaviours of aircraft, airports (runways, taxi-ways, apron, etc.), air lanes, ground, terminal, regional, and continental control towers, of (national [CAA, CAAC, FAA, SLV, etc.] and international [JAA, CAO]) aviation authorities, etc.

Airports: A domain description includes descriptions of the flow of people (passengers, staff), material (catering, fuel, baggage), aircraft, information (boarding cards, baggage tags) and control; of these entities, of the operations performed by or on them, the events that may occur (cancellation or delay of flights, lost luggage, missing passenger), and, hence, of the many concurrent and intertwined (mutually “synchronising”) behaviours that entities undergo.

Container Shipping: A domain description includes descriptions of containers, container ships, the stowage of containers on ships and in container

yards, container terminal (ports), the loading and unloading of containers between ships and ports and between ports and the “hinterland” (including cranes, port trucking and feeder trucks, trains and barges), the container bills of lading (or way bills), the container transport logistics, the (planning and execution, scheduling and allocation) of voyages, the berthing (arrival and departure) of container ships, customer relations, etc.

Financial Service Industry: A domain description includes descriptions of banks (and banking: [demand/deposit, savings, mortgage] accounts, [opening, closing, deposit, withdrawal, transfer, statements] operations on accounts), insurance companies (claims processing, etc.), securities trading (stocks, bonds, brokers, traders, exchanges, etc.), portfolio management, IPOs, etc.

Health care: A domain description includes descriptions of the entities, operations, events and behaviours of healthy people, patients and medical staff, of private physicians, medical clinics, hospitals, pharmacies, health insurance, national boards of health, etc.

The Internet: The reader is encouraged to fill in some details here!

Manufacturing: Machining & Assembly: The reader is encouraged to also fill in some details here!

“The” Market: A domain description includes descriptions of the entities, operations, events and behaviours of consumers, retailers, wholesalers, producers, the delivery chain and the payment of (or for) merchandise and services.

Transportation: A domain description includes descriptions of the entities, functions, events and behaviours of transport vehicles (cars/trucks/busses, trains, aircraft, ships), [multimodal] transport nets (roads, rail lines, air lanes, shipping lanes) and hubs (road intersections [junctions], stations, airports, harbours), transported items (people and freight), and of logistics (scheduling and allocation of transport items to transport vehicles, and of transport vehicles to transport nets and hubs). Monomodal descriptions can focus on just air traffic or on container shipping, or on railways.

The Web: The reader is encouraged to “likewise” fill in some details here!

There are many “less grand” domains: railway level crossings, the interconnect cabling between the oftentimes dozens of “boxes” of some electronic/mechanical/acoustical measuring set-up, a gas burner, etc. These are all, rather one-sidedly, examples of what might be called embedded, or real-time, or safety critical systems.

We can refer to several projects at UNU-IIST which have produced domain specifications for railway systems (China), ministry of finance (Vietnam), telephone systems (The Philippines), harbours (India), etc.; and to dozens of MSc projects which have likewise produced domain specifications for airports, air traffic, container shipping, health care, the market, manufacturing, etc. I give many, many references in [89]. I also refer the reader to <http://www.railwaydomain.org/> for documents, specifically <http://www.railwaydomain.org/book.pdf> for domain models of railway systems.

Some Remarks

A point made by listing and explaining the above domains is the following: They all display a seeming complexity in terms of multitude of entities, functions, events and interrelated behaviours; and they all focus on the reality of “what is out there”: no mention is (to be) made of requirements to supporting computing systems let alone of these (incl. software).

C.2.6 Domains: Suggested Research Topics

From the above list we observe that the ‘transportation item’ “lifts” those of ‘air traffic’ and ‘container shipping’. Other examples could be shown. This brings us, at this early stage where we have yet to really outline what domain engineering is, to suggest the following research topics:

- ℔ 2. **Lifted Domains and Projections:** We observe, above, that the ‘transportation’ domain seems to be an abstraction of at least four more concrete domains: road, rail, sea and air transportation. We could say that ‘transportation’ is a commensurate “lifting” of each of the others, or that these more concrete could arise as a result of a “projection” from the ‘transportation’ domain. The research topic is now to investigate two aspects: a computing science cum software engineering aspect and a computer science aspect. The former should preferably result in principles, techniques and tools for choosing levels of “lifted” abstraction and “projected” concretisation. The latter should study the implied “lifting” and “projection” operators.
- ℔ 3. **What Do We Mean by an Infrastructure ?** We observe, above, that some of the domains exemplify what is normally called infrastructure² components. According to the World Bank: *‘Infrastructure’ is an umbrella term for many activities referred to as ‘social overhead capital’ by some development economists, and encompasses activities that share technical and economic features (such as economies of scale and spillovers from users to nonusers).* The research is now to study whether we can reformulate the sociologically vague World Bank definition in precise mathematical terms.

²Winston Churchill is quoted to have said, during a debate in the House of Commons, in 1946: ... *The young Labourite speaker that we have just listened to, clearly wishes to impress upon his constituency the fact that he has gone to Eton and Oxford since he now uses such fashionable terms as ‘infra-structures’.* [I have recently been in communication with the British House of Commons information office enquiries manager, Mr. Martin Davies in order to verify and, possibly pinpoint, this statement. I am told that “as the Hansard debates in question are not available electronically, it could only be found via a manual search of hard copy Hansard”. So there it stands.]

- ℔ 4. **What Is an Infrastructure Component ?** We observe, above, that not all of the domains exemplified are what is normally called infrastructure components.³ The research is now to study whether we can formulate and formalise some “tests” which help us determine whether some domain that we are about to model qualifies as part of one or more infrastructure components.

We bring these early research topic suggestions so that the reader can better judge whether domain engineering principles and techniques might help in establishing a base for such research. Throughout the paper we shall “spice it” with further suggestions of research topics.

• • •

We do not cover the important methodological aspects of stake holder identification and liaison, domain acquisition and analysis, domain model verification and validation. For that we refer to Vol. 3 Chaps. 9–10 and 12–14 [89].

C.3 Domain Facets

The rôle, the purpose, of domain engineering is to construct, to develop, and research domain descriptions. It is both an engineering and a scientific task. It is engineering because we do know, today, a necessary number of principles, techniques and tools with which to create domain models. It is scientific, i.e., of research nature, because, it appears, that we do not necessarily know, today, whether what we know is sufficient.

C.3.1 Stages of Domain Development

By domain development we mean a process, consisting of a number of reasonably clearly separable stages which when properly conducted leads to a domain description, i.e., a domain model. We claim that the following are meaningful and necessary domain development stages of development, each with their attendant principles, techniques and tools: (i) identification of stake holders, (ii) rough domain identification, (iii) domain acquisition, (iv) analysis of rough domain description units, (v) domain modelling, (vi) domain verification, (vii) domain validation and (viii) domain theory formation. We shall focus on domain modelling emphasising the modelling concept of domain facets.

³‘Manufacturing’ and ‘The Market’ appear, in the above list to not be infrastructure components, but, of course, they rely on the others, the infrastructure components.

C.3.2 The Facets

By domain modelling we mean the construction of both an informal, narrative and a formal domain description.

We claim that the following identified facets (i.e., “steps”) (later to be briefly explained) are necessary parts of the domain modelling process: (i) intrinsics, (ii) support technologies, (iii) management and organisation, (iv) rules and regulations, (v) and human behaviour. Ideally speaking one may proceed with these “steps” in the order listed. Engineering accommodates for less ideal progressions. Each “step” produces a partial domain description. Subsequent “steps” ‘extend’ partial descriptions into partial or even (relative) complete descriptions.

In this section, Sect. C.3, we will not give concrete examples but will rely on such already given in Chap. 11 of [89].

C.3.3 Intrinsics

By the intrinsics of a domain we shall understand those phenomena and concepts, that is, those entities, functions, events and behaviours in terms of which all other facets are described.

The choice as to what constitutes the intrinsics of a domain is often determined by the views of the stake holders. Thus it is a pragmatic choice, and the choice cannot be formalised in the form of an **is_intrinsics** predicate that one applies to phenomena and concepts of the domain.

- ℜ 5. **Intrinsics:** What is, perhaps, needed, is a theoretically founded characterisation of “being intrinsic”.

C.3.4 Support Technology

By a support technology of a domain we shall understand either of a set of (one or more) alternative entities, functions, events and behaviours which “implement” an intrinsic phenomenon or concept. Thus for some one or more intrinsic phenomena or concepts there might be a technology which supports those phenomena or concepts.

Sampling Behaviour of Support Technologies

Let us consider **intrinsic Air Traffic** as a continuous function (\rightarrow) from **Time** to **Flight Locations**:

type
 T, F, L
 $iAT = T \rightarrow (F \xrightarrow{m} L)$

But what is observed, by some support technology, is not a continuous function, but a discrete sampling (a map \overrightarrow{m}):

$$\text{sAT} = T \xrightarrow{\overrightarrow{m}} (F \xrightarrow{\overrightarrow{m}} L)$$

There is a support technology, say in the form of **radar** which “observes” the intrinsic traffic and delivers the sampled traffic:

value

$$\text{radar}: \text{iAT} \rightarrow \text{sAT}$$

Probabilistic cum Statistical Behaviour of Support Technologies

But even the radar technology is not perfect. Its positioning of flights follows some probabilistic or statistical pattern:

type

$$P = \{ |r: \mathbf{Real} \bullet 0 \leq r \leq 1| \}$$

$$\text{ssAT} = P \xrightarrow{\overrightarrow{m}} \text{sAT-infset}$$

value

$$\text{radar}': \text{iAT} \xrightarrow{\sim} \text{ssAT}$$

The radar technology will, with some probability produce either of a set of samplings, and with some other probability some other set of samplings, etc.⁴

Support Technology Quality Control, a Sketch

How can we express that a given technology delivers a reasonable support ? One approach is to postulate intrinsic and technology states (or observed behaviours), Θ_i, Θ_s , a support technology τ and a “closeness” predicate:

type

$$\Theta_i, \Theta_s$$

value

$$\tau: \Theta_i \rightarrow P \xrightarrow{\overrightarrow{m}} \Theta_s\text{-infset}$$

$$\text{close}: \Theta_i \times \Theta_s \rightarrow \mathbf{Bool}$$

and then require that an experiment can be performed which validates the support technology.

The experiment is expressed by the following axiom:

⁴Throughout this paper we omit formulation of type well-formedness predicates.

```

value
  p_threshold:P
axiom
   $\forall \theta_i:\Theta_i \bullet$ 
    let  $p\theta_{ss} = \tau(\theta_i)$  in
     $\forall p:P \bullet p > p\_threshold \Rightarrow$ 
       $\theta_s:\Theta_s \bullet \theta_s \in p\theta_{ss}(p) \Rightarrow \text{close}(\theta_i, \theta_s)$  end

```

The $p_threshold$ probability has to be a-priori determined as one above which the support technology renditions of the intrinsic states (or behaviours) are acceptable.

Support Technologies: Suggested Research Topics

- ℜ 6. **Probabilistic and/or Statistical Support Technologies:** Some cases should be studied to illuminate the issue of probability versus statistics. More generally we need more studies of how support technologies “enter the picture”, i.e., how “they take over” from other facet. And we need to come up with precise modelling concepts for probabilistic and statistical phenomena and their integration into the formal specification approaches at hand.
- ℜ 7. **A Support Technology Quality Control Method:** The above sketched a ‘support technology quality control’ procedure. It left out the equally important ‘monitoring’ aspects. Develop experimentally two or three distinct models of domains involving distinct sets of support technologies. Then propose and study concrete implementations of ‘support technology quality monitoring and control’ procedures.

C.3.5 Management and Organisation

By the management of an enterprise (an institution) we shall understand a (possibly stratified, see ‘organisation’ next) set of enterprise staff (behaviours, processes) authorised to perform certain functions not allowed performed by other enterprise staff (behaviours, processes) and where such functions involve monitoring and controlling other enterprise staff (behaviours, processes). By organisation of an enterprise (an institution) we shall understand the stratification (partitioning) of enterprise staff (behaviours, processes) with each partition endowed with a set of authorised functions and with communication interfaces defined between partitions, i.e., between behaviours (processes).

An Abstraction of Management Functions

Let **E** designate some enterprise state concept, and let **stra_mgt**, **tact_mgt**, **oper_mgt**, **wrkr** and **merge** designate strategic management, tactical management, operational management and worker actions on states such that

these actions are “somehow aware” of the state targets of respective management groups and or workers. Let **p** be a predicate which determines whether a given target state has been reached, and let **merge** harmonise different state targets into an agreeable one. Then the following behaviour reflects some aspects of management.

```

type
  E
value
  stra_mgt, tact_mgt, oper_mgt, wrkr, merge: E×E×E×E → E
  p: E* → Bool
  mgt: E → E
  mgt(e) ≡
    let e' = stra_mgt(e,e'',e''',e'''),
        e'' = tact_mgt(e,e'',e''',e'''),
        e''' = oper_mgt(e,e'',e''',e'''),
        e'''' = wrkr(e,e'',e''',e''') in
    if p(e,e'',e''',e''')
      then skip
      else mgt(merge(e,e'',e''',e'''))
    end end

```

The recursive set of $e' \dots' = f(e, e'', e''', e''')$ equations are “solved” by iterative communication between the management groups and the workers. The arrangement of these equations reflect the organisation and the various functions, **stra_mgt**, **tact_mgt**, **oper_mgt** and **wrkr** reflect the management. The frequency of communication between the management groups and the workers help determine a quality of the result.

The above is just a very crude, and only an illustrative model of management and organisation.

We could also have given a generic model, as the above, of management and organisation but now in terms of, say, CSP processes. Individual managers are processes and so are workers. The enterprise state, $e : E$, is maintained by one or more processes, separate from manager and worker processes. Etcetera.

Management and Organisation: Suggested Research Topics

- ℔ 8. **Strategic, Tactical and Operation Management:** We made no explicit references to such “business school of administration” “BA101” topics as ‘strategic’ and ‘tactical’ management. Study Example 9.2 of Sect. 9.3.1 of Vol. 3 [89]. Study other sources on ‘Strategic and Tactical Management’. Question Example 9.2’s attempt at delineating ‘strategic’ and ‘tactical’ management. Come up with better or other proposals, and/or attempt clear, but not necessarily computable predicates which (help) determine whether an operation (above they are alluded to as ‘stra’ and ‘tact’) is one of strategic or of tactical concern.

- ℔ 9. **Modelling Mgt. and Org.: Applicatively or Concurrently:** The abstraction of ‘management and organisation’ on Page C.3.5 was applicative, i.e., a recursive function — whose auxiliary functions were hopefully all continuous. Suggest a CSP rendition of “the same idea” ! Relate the applicative to the concurrent models.

C.3.6 Rules and Regulations

By a rule of an enterprise (an institution) we understand a syntactic piece of text whose meaning apply in any pair of actual present and potential next states of the enterprise, and then evaluates to either true or false: the rule has been obeyed, or the rule has been (or will be, or might be) broken. By a regulation of an enterprise (an institution) we understand a syntactic piece of text whose meaning, for example, apply in states of the enterprise where a rule has been broken, and when applied in such states will change the state, that is, “remedy” the “breaking of a rule”.

Abstraction of Rules and Regulations

Stimuli are introduced in order to capture the possibility of rule-breaking next states.

type

Sti, Rul, Reg
 RulReg = Rul \times Reg
 Θ
 STI = $\Theta \rightarrow \Theta$
 RUL = $(\Theta \times \Theta) \rightarrow \mathbf{Bool}$
 REG = $\Theta \rightarrow \Theta$

value

meaning: Sti \rightarrow STI, Rul \rightarrow RUL, Reg \rightarrow REG
 valid: Sti \times Rul $\rightarrow \Theta \rightarrow \mathbf{Bool}$
 valid(sti,rul) $\theta \equiv$ (meaning(rul))(θ ,meaning(sti) θ)

axiom

\forall sti:Sti,(rul,reg):RulReg, θ : Θ •
 \sim valid(sti,rul) $\theta \Rightarrow$ meaning(rul)(θ ,meaning(reg) θ)

Quality Control of Rules and Regulations

The axiom above presents us with a guideline for checking the suitability of (pairs of) rules and regulations in the context of stimuli: for every proposed pair of rules and regulations and for every conceivable stimulus check whether the stimulus might cause a breaking of the rule and, if so, whether the regulation will restore the system to an acceptable state.

Rules and Regulations Suggested Research Topic:

- ℜ 10. **A Concrete Case:** The above sketched a quality control procedure for ‘stimuli, rules and regulations’. It left out the equally important ‘monitoring’ aspects. Develop experimentally two or three distinct models of domains involving distinct sets of rules and regulations. Then propose and study concrete implementations of procedures for quality monitoring and control of ‘stimuli, rules and regulations’.

C.3.7 Human Behaviour

By human behaviour we understand a “way” of representing entities, performing functions, causing or reacting to events or participating in behaviours. As such a human behaviour may be characterisable on a per phenomenon or concept basis as lying somewhere in the “continuous” spectrum from (i) diligent: precise representations, performances, event (re)actions, and behaviour interactions; via (ii) sloppy: occasionally imprecise representations, performances, event (re)actions, and behaviour interactions; and (iii) delinquent: repeatedly imprecise representations, performances, event (re)actions, and behaviour interactions; to (iv) criminal: outright counter productive, damaging representations, performances, event (re)actions, and behaviour interactions.

Abstraction of Human Behaviour

We extend the formalisation of rules and regulations.

Human actions (**ACT**) lead from a state (Θ) to any one of possible successor states (Θ -**infset**) — depending on the human behaviour, whether diligent, sloppy, delinquent or having criminal intent. The **human interpretation** of a rule (**Rul**) usually depends on the current state (Θ) and can be any one of a possibly great number of semantic rules (**RUL**). For a delinquent (...) user the rule must yield truth in order to satisfy “being delinquent (...)”.

type

$$\text{ACT} = \Theta \rightarrow \Theta\text{-infset}$$

value

$$\text{hum_int: Rul} \rightarrow \Theta \rightarrow \text{RUL-infset}$$

$$\text{hum_behav: Sti} \times \text{Rul} \rightarrow \text{ACT} \rightarrow \Theta \rightarrow \Theta\text{-infset}$$

$$\text{hum_behav(sti,rul)}(\alpha)(\theta) \text{ as } \theta_s$$

$$\text{post } \theta_s = \alpha(\theta) \wedge$$

$$\forall \theta': \Theta \bullet \theta' \in \theta_s \Rightarrow$$

$$\exists \text{se_rul:RUL} \bullet \text{se_rul} \in \text{hum_int(rul)}(\theta) \Rightarrow \text{se_rul}(\theta, \theta')$$

Human behaviour is thus characterisable as follows: It occurs in a context of a **stimulus**, a **rule**, a present state (θ) and (the choice of) an action (α :**ACT**) which may have either one of a number of outcomes (θ_s). Thus let θ_s be the

possible spread of diligent, sloppy, delinquent or outright criminal successor states. For each such successor states there must exist a rule interpretation which satisfies the pair of present an successor states. That is, it must satisfy being either diligent, sloppy, delinquent or having criminal intent and possibly achieving that!

Human Behaviour Suggested Research Topics:

Section 11.8 of Vol. 3 [89] elaborates on a number of ways of describing (i.e., modelling) human behaviour.

- ℜ 11. **Concrete Methodology:** Based on the abstraction of human behaviour given earlier, one is to study how one can partition the set, $\alpha(\theta)$, of outcomes of human actions into ‘diligent’, ‘sloppy’, ‘delinquent’ and ‘criminal’ behaviours — or some such, perhaps cruder, perhaps finer partitioning — and for concrete cases attempt to formalise these for possible interactive “mechanisation”.
- ℜ 12. **Monitoring and Control of Human Behaviour:** Based on possible solutions to the previous research topic one is to study general such interactive “mechanisation” of the monitoring and control of human behaviour.

C.3.8 Domain Modelling: Suggested Research Topic

- ℜ 13. **Sufficiency of Domain Facets:** We have covered five facets: intrinsics, support technology, management and organisation, rules and regulations and human behaviour. The question is: are these the only facets, i.e., views on the domain that are relevant and can be modelled? Another question is: is there an altogether different set of facets, “cut up”, so-to-speak, “along other lines of sights”, using which we could likewise cover our models of domains?

One might further subdivide the above five facets (intrinsics, support technology, management and organisation, rules and regulations and human behaviour) into “sub”-facets. A useful one seems to be to separate out from the facet of rules and regulations the sub-facet of scripts.

• • •

We have finished our overview of domain facets.

C.4 Domains: Miscellaneous Issues

C.4.1 Domain Theories

- *By a domain theory we shall understand a domain description together with lemmas, propositions and theorems that may be proved about the description — and hence can be claimed to hold in the domain.*

To create a domain theory the specification language must possess a proof system. It appears that the essence of possible theorems of — that is, laws about — domains can be found in laws of physics. For a delightful view of the law-based nature of physics — and hence possibly also of man-made universes we refer to Richard Feynman’s Lectures on Physics [155].

Example Theorem of Railway Domain Theory

Let us hint at some domain theory theorems: **Kirchhoff’s Law for Railways:** Assume regular train traffic as per a modulo κ hour time table. Then we have, observed over a κ hour period, that the number of trains arriving at a station minus the number of trains ending their journey at that station plus the number of trains starting their journey at that station equals the number of trains departing from that station.

Why Domain Theories ?

Well, it ought be obvious ! We need to understand far better the laws even of man-made systems.

Domain Theories: Suggested Research Topics:

- ℜ 14. **Domain Theories:** We need to experimentally develop and analyse a number of suggested theorems for a number of representative domains in order to possibly ‘discover’ some meta-theorems: laws about laws !

C.4.2 Domain Descriptions and Requirements Prescriptions

From Domains to Requirements

Requirements prescribe what “the machine”, i.e., the hardware + software is expected to deliver. We show, in Vol. 3, Part V, Requirements Engineering, and in particular in Chap. 19, Sects. 19.4–19.5 how to construct, from a domain description, in collaboration with the requirements stakeholders, the domain (i.e., functional) requirements, and the interface (i.e., user) requirements.

Domain requirements are those requirements which can be expressed only using terms from the domain description. Interface requirements are those requirements which can be expressed only using terms from both the domain description and the machine — the latter means that terms of computers and software are also being used.

Domain requirements are developed as follows: Every line of the domain description is inspected by both the requirements engineer and the requirements stakeholders. For each line the first question is asked: *Shall this line*

of description prescribe a property of the requirements ? If so it is “copied” over to the requirements prescription. If not it is “projected away”. In similar rounds the following questions are then raised and answered: *Shall the possible generality of the description be instantiated to something more concrete ? Shall possible non-determinism of the description be made less non-deterministic, more deterministic ? Shall the domain be “extended” to allow for hitherto infeasible entities, functions, events and behaviours ? Shall the emerging requirements prescription be “fitted” to elsewhere emerging requirements prescriptions ?* Similar “transformation” steps can be applied in order to arrive at (data initialisation and refreshment, GUI, dialogue, incremental control flow, machine-to-machine communication, etc.) interface requirements.

Domain and Interface Requirements: Suggested Research Topics

- ℞ 15. **Domain and Interface Requirements:** Vol. 3, Part V, Sects. 19.4–19.5 give many examples of requirements “derivation” principles and techniques. But one could wish for more research in this area: more detailed principles and techniques, on examples across a wider spectrum of problem frames.

Machine Requirements

The issues of machine requirements are listed below but are not treated in this paper.

- | | |
|-------------------|---------------------|
| • Performance | ★ adaptive |
| ★ space | ★ corrective |
| ★ time | ★ perfective |
| • Dependability | ★ preventive |
| ★ Accessibility | ★ extensional |
| ★ Availability | • Platform |
| ★ Integrity | ★ development |
| ★ Reliability | ★ demonstration |
| ★ Safety | ★ execution |
| ★ Security | ★ maintenance |
| • Maintainability | • Documentation ... |

See Sect. 19.6 of Vol. 3.

C.4.3 Requirements-Specific Domain Software Development Models

A long term, that one: ‘requirements-specific domain software development models’ ! The term is explained next.

Software “Intensities”

One can speak of ‘software intensity’. Here are some examples. Compilers represent ‘translation’ intensity. ‘Word processors’, ‘spread sheet systems’, etc., represent “workpiece” intensity. Databases represent ‘information’ intensity. Real-time embedded software represent ‘reactive’ intensity. Data communication software represent connection intensity. Etcetera.

“Abstract” Developments

Let \mathcal{R} denote the “archetypal” requirements for some specific software ‘intensity’. Many different domains $\{\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_i, \dots, \mathcal{D}_j, \dots\}$ may be subject to requirements \mathcal{R} -like prescriptions. For each such a set of possible software, $\mathcal{S}_{i_1}, \mathcal{S}_{i_2}, \dots, \mathcal{S}_{i_{j_i}}, \dots$, may result. The “pseudo-formula” below attempts, albeit informally, to capture this situation:

$$\left\{ \begin{array}{c} \mathcal{D}_1 \\ \mathcal{D}_2 \\ \dots \\ \mathcal{D}_i \\ \dots \\ \mathcal{D}_k \\ \dots \end{array} \right\} \sim \mathcal{R} \mapsto \left[\begin{array}{c} \{\mathcal{S}_{1_1}, \mathcal{S}_{1_2}, \dots, \mathcal{S}_{1_{j_1}}, \dots\} \\ \{\mathcal{S}_{1_1}, \mathcal{S}_{1_2}, \dots, \mathcal{S}_{1_{j_2}}, \dots\} \\ \dots \\ \{\mathcal{S}_{i_1}, \mathcal{S}_{i_2}, \dots, \mathcal{S}_{i_{j_i}}, \dots\} \\ \dots \\ \{\mathcal{S}_{k_1}, \mathcal{S}_{k_2}, \dots, \mathcal{S}_{k_{j_k}}, \dots\} \\ \dots \end{array} \right]$$

Several different domains, to wit: road nets and railway nets, can be given the “same kind” of (road and rail) maintenance requirements leading to information systems. Several different domains, to wit: road nets, railway nets, shipping lanes, or air lane nets, can be given the “same kind” of (bus, train, ship, air flight) monitoring and control requirements (leading to real-time embedded systems). But usually the specific requirements skills determine much of the requirements prescription work and especially the software design work.

Requirements-Specific Devt. Models: Suggested Research Topics

- ℜ 16_j. **Requirements-Specific Development Models, \mathcal{RSDM}_j :** We see these as grand challenges: to develop and research a number of requirements-specific domain (software) development models \mathcal{RSDM}_j .

The “pseudo-formal” $[\Pi(\Sigma_i \mathcal{D}_i)] \mathcal{R} [\Sigma_{i,j} \mathcal{S}_{i_j}]$ expression attempts to capture an essence of such research: The Π “operator” is intended to project those domains, \mathcal{D}_i , in the sum, $\Sigma_i \mathcal{D}_i$, for which \mathcal{R} may be relevant. The research explores the projections Π , the possible \mathcal{R} s and the varieties of software $\Sigma_{i,j} \mathcal{S}_{i_j}$.

C.4.4 On Two Reasons for Domain Modelling

Thus there seems to be two entirely different, albeit, related reasons for domain modelling: one justifies domain modelling on engineering grounds, the other on scientific grounds.

An Engineering Reason for Domain Modelling

In an e-mail, in response, undoubtedly, to my steadfast, perhaps conceived as stubborn insistence, on domain engineering, Sir Tony Hoare summed up his reaction, in summer of 2006, to domain engineering as follows, and I quote⁵:

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

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 describe⁶ the⁷ whole range of possible designs for the software, and the whole range of technologies available for its realisation.
4. They provide a framework for a full analysis of requirements, which is wholly independent of the technology of implementation.
5. They enumerate and analyse the decisions that must be taken earlier or later in any design project, and identify those that are independent and those that conflict. Late discovery of feature interactions can be avoided.”

All of these issues are dealt with, one-by-one, and in some depth, in Vol. 3 [89] of my three volume book.

A Science Reason for Domain Modelling

So, inasmuch as the above-listed issues of Sect. C.4.4, so aptly expressed in Tony’s mastery, also of concepts (through his delightful mastery of words), are of course of utmost engineering importance, it is really, in our mind, the science issues that are foremost: We must first and foremost understand. There is no excuse for not trying to first understand. Whether that understanding can be “translated” into engineering tools and techniques is then another matter. But then, of course, it is nice that clear and elegant understanding also leads to better tools and hence better engineering. It usually does.

Domains Versus Requirements-Specific Development Models

Sir Tony’s five statements are more related, it seems, to the concept of requirements-specific domain software development models than to merely the concept of domain models. His statements help us formulate the research programme $\mathfrak{R}16$ of requirements specific domain software development models. When, in his statements, you replace his use of the term ‘models’ with our term ‘requirements-specific development models *based on domain models*’, then “complete harmony” between the two views exists.

⁵E-Mail to Dines Bjørner, CC to Robin Milner et al., July 19, 2006

⁶read: imply

⁷read: a

C.5 Conclusion

C.5.1 What Has Been Achieved ?

I set out to focus on what I consider the crucial modelling stage of describing domain facets and to identify a number of their research issues. I've done that. Cursorily, the topic is “near-holistic”, so an overview is all that can be done. The issue is that of that of a comprehensive methodology. Hence the “holism” challenge.

C.5.2 What Needs to Be Achieved ?

Well, simply, to get on with that research. There are two sides to it: the 16 research topics mentioned above, and the ones mentioned below. The latter serves as a carrier for the former research.

Domain Theories: Grand Challenge Research Topics

The overriding research topic is that of:

- ℜ 17_i. **Domain Models: \mathcal{D}_i :** We see this as a set of grand challenges: to develop and research a family of domain models \mathcal{D}_i .

C.5.3 Acknowledgements

I thank the organisers for inviting me to present a (this ?) talk. I thank UNU-IIST for inviting me and my wife back to Macau to a place where I spent great years. I consider UNU/IIST (as we spelled it in those days) one of my main achievements, so I also thank all those people who made it possible. They may have suffered then. But they too can be very proud now. I thank Sir Tony for fruitful discussions during the writing of this paper.

D

From Domains to Requirements: A Rigorous Approach

This appendix chapter constitutes the invited paper for The Ugo Montanari Festschrift [95, to appear], May 2008, to be published by Springer, editors: Pierpaolo Degano, Jose Meseguer and Rocco De Nicola.

Permission to bring this paper here is being applied for.

Abstract This is a discursive paper. That is, it shows some formulas (but only as examples so that the reader may be convinced that there is, perhaps, some substance to our claims), no theorems, no proofs. Instead it postulates. The postulates are, however, firmly rooted, we think, in Vol.3 ('Domains, Requirements and Software Design') of the three volume book 'Software Engineering' (Springer March 2006) [87–89].

First we present a summary of essentials of domain engineering, its motivation, and its modelling of abstractions of domains through the modelling of the intrinsics, support technologies, management and organisation, rules and regulations, scripts, and human behaviour of whichever domain is being described.

Then we present the essence of two (of three) aspects of requirements: the domain requirements and the interface requirements prescriptions as they relate to domain descriptions and we survey the basic operations that "turn" a domain description into a domain requirements prescription: projection, instantiation, determination, extension and fitting. An essence of interface requirements is also presented: the "merging" of shared entities, operations, events and behaviours of the domain with those of the machine (i.e., the hardware and software to be designed).

An objective of the paper is to summarise my work in recent years. Another objective is make a plea for what I consider a more proper approach to software development.

D.1 Introduction

This paper is not a computer science paper — where by computer science, sometimes strangely referred to even as theoretical computer science, we mean the study and knowledge of the things that may exist inside computers.

The paper is more of a computing science paper — where by computing science we mean the study and knowledge of how to construct the things that can exist inside computers.

The borderline between these two disciplines is sharp, but most interesting papers which purports to be computing science papers also, oftentimes strongly, contains text of computer science nature.

Some computer science papers present new models of computation and/or analyses and theorems about such models.

This paper presents a model of early stages of software development that is not conventional. The model is presented in two alternating ways: (i) we present some of the principles and techniques of that unconventional software development method, and (ii) we present — what in the end, that is, taken across the paper, amounts to a relatively large example.

One aspect of the non-conventionality of the present paper is its total lack of ‘references to related work’ by others. Instead we shall solely refer, now, to our own work related to the topic of the current paper. In those referenced works you should find ‘references to related work’.

The topic pair of domain and of requirements engineering — on which the present paper relies — is treated in depth in [89, Chaps. 8–24, Pages 193–524 (!)]. Recent papers elaborate on related (possible) research topics [90], or on software management [93, to appear], or gives more extensive summaries on domain engineering, one without a leading, extensive example but with a more proper discussion of domain modelling issues and ‘related work’ [94, to appear], and one with a considerably larger example [96, to appear (the example appendix, pages 32–97, illustrates a Container Line Industry domain)].

In summary: the objective of the present paper is to relate domain engineering to requirements engineering and to show that one can obtain an altogether different basis for requirements engineering.

D.2 The Triptych Principle of Software Engineering

We start, unconventionally, by enunciating a principle. The principle expresses how we see software development as centrally consisting of three “programming-like” phases based on the following observation: before software can be designed we must understand its requirements, and before requirements can be prescribed we must understand the application domain. We therefore see software development proceeding, ideally, in three phases: a first phase of domain engineering, a second phase of requirements engineering, and a third phase of software design.

The first paragraphs of Sects. D.3 and D.4 explain what the objectives of domain engineering and requirements engineering are. The sections otherwise outline major development stages and steps of these two phases.

D.3 Domain Engineering

The objective of domain engineering is to create a domain description. A domain description specifies entities, functions, events and behaviours of the domain such as the domain stakeholders think they are. A domain description thus (indicatively [206]) expresses what there is. A domain description expresses no requirements let alone anything about the possibly desired (required) software.

D.3.1 Stages of Domain Engineering

To develop a proper domain description necessitates a number of development stages: (i) identification of stakeholders, (ii) domain knowledge acquisition, (iii) business process rough-sketching, (iv) domain analysis, (v) domain modelling: developing abstractions and verifying properties, (vi) domain validation and (vii) domain theory building.

Business process (BP) rough-sketching amount to rough, narrative outlines of the set of business processes as experienced by each of the stakeholder groups. BP engineering is in contrast to BR re-engineering (BPR) which we shall cover later, but briefly in Sect. D.4.2.

We shall only cover domain modelling.

D.3.2 First Example of a Domain Description

We exemplify a transportation domain. By transportation we shall mean *the movement of vehicles from hubs to hubs along the links of a net*.

Rough Sketching — Business Processes

The basic *entities* of the transportation “business” are the (i) *nets* with their (ii) *hubs* and (iii) *links*, the (iv) *vehicles*, and the (v) *traffic* (of vehicles on the net). The basic *functions* are those of (vi) vehicles entering and leaving the net (here simplified to entering and leaving at hubs), (vii) for vehicles to make movement transitions along the net, and (viii) for inserting and removing links (and associated hubs) into and from the net. The basic *events* are those of (ix) the appearance and disappearance of vehicles, and (x) the breakdown of links. And, finally, the basic behaviours of the transportation business are those of (xi) vehicle journey through the net and (xii) net development & maintenance including insertion into and removal from the net of links (and hubs).

Narrative — Entities

By an *entity* we mean *something we can point to, i.e., something manifest, or a concept abstracted from, such a phenomenon or concept thereof*.

Among the many entities of transportation we start with nets, hubs, and links.

A transportation net consists of hubs and links. Hubs and links are different kinds of entities. Conceptually hubs (links) can be uniquely identified. From a link one can observe the identities of the two distinct hubs it links. From a hub one can observe the identities of the one or more distinct links it connects.

Other entities such as vehicles and traffic could as well be described. Please think of these descriptions of entities as descriptions of the real phenomena and (at least postulated) concepts of an actual domain.

Formalisation — Entities

```

type
  H, HI, L, LI
  N = H-set × L-set
value
  obs_HI: H → HI, obs_LI: L → LI,
  obs_HIs: L → HI-set, obs_LIs: H → LI-set
axiom
  ∀ (hs,ls):N •
    card hs ≥ 2 ∧ card ls ≥ 1 ∧
    ∀ h:H • h ∈ hs ⇒
      ∀ li:LI • li ∈ obs_LIs(h) ⇒
        ∃ l':L • l' ∈ ls ∧ li = obs_LI(l') ∧ obs_HI(h) ∈ obs_HIs(l') ∧
    ∀ l:L • l ∈ ls ⇒
      ∃ h',h'':H • {h',h''} ⊆ hs ∧ obs_HIs(l) = {obs_HI(h'), obs_HI(h'')}
value
  xtr_HIs: N → HI-set, xtr_LIs: N → LI-set

```

Narrative — Operations

By an *operation* (of a domain) we mean *a function that applies to entities of the domain and yield entities of that domain — whether these entities are actual phenomena or concepts of these or of other phenomena*.

Actions (by domain stakeholders) amount to the execution of operations.

Among the many operations performed in connection with transportation we illustrate some on nets. To a net one can join new link in either of three ways: The new link connects two new hubs — so these must also be joined ,

or The new link connects a new hub with an existing hub — so it must also be joined, or The new link connects two existing hubs. In any case we must either provide the new hubs or identify the existing hubs.

From a net one can remove a link. Three possibilities now exists: The removed link would leave its two connected hubs isolated unless they are also removed — so they are; The removed link would leave one of its connected hubs isolated unless it is also removed — so it is; or The removed link connects two hubs into both of which other links are connected — so all is OK. (Note our concern for net invariance.) Please think of these descriptions of operations as descriptions of the real phenomena and (at least postulated) concepts of an actual domain. (Thus they are not prescriptions of requirements to software let alone specifications of software operations.)

Formalisation — Operations

type

```
NetOp = InsLnk | RemLnk
InsLnk == 2Hs(h1:H,l:L,h2:H)|1H(hi:HI,l:L,h:H)|0H(hi1:HI,l:L,hi2:HI)
RemLnk == RmvL(li:LI)
```

value

```
int_NetOp: NetOp → N ≅ N
pre int_NetOp(op)(hs,ls) ≡
  case op of
    2Hs(h1,l,h2) →
      {h1,h2} ∩ hs = {} ∧ l ∉ ls ∧
      obs_HIs(l) = {obs_HI(h1), obs_HI(h2)} ∧
      {obs_HI(h1), obs_HI(h2)} ∩ xtr_HIs(hs) = {} ∧
      obs_LIs(h1) = {li} ∧ obs_LIs(h2) = {li},
    1H(hi,l,h) →
      h ∉ hs ∧ obs_HI(h) ∉ xtr_HIs(hs,ls) ∧
      l ∉ ls ∧ obs_LI(l) ∉ xtr_LIs(hs,ls) ∧
      ∃ h':H • h' ∈ hs ∧ obs_HI(h') = hi,
    0H(hi1,l,hi2) →
      l ∉ ls ∧ hi1 ≠ hi2 ∧ {hi1,hi2} ⊆ xtr_HIs(hs,ls) ∧
      ∃ h1,h2:H • {h1,h2} ∈ hs ∧ {hi1,hi2} = {obs_HI(h1), obs_HI(h2)},
    RmvL(li) → ∃ l:L • l ∈ ls ∧ obs_LI(l) = li
  end

int_NetOp(op)(hs,ls) ≡
  case op of
    2Hs(h1,l,h2) → (hs ∪ {h1,h2}, ls ∪ {l}),
    1H(hi,l,h) →
      (hs \ {xtr_H(hi,hs)} ∪ {h, aLI(xtr_H(hi,hs), obs_LI(l))}, ls ∪ {l}),
    0H(hi1,l,hi2) →
```

```

    let hsδ =
      {aLI(xtr_H(hi1,hs),obs_LI(l)),aLI(xtr_H(hi2,hs),obs_LI(l))} in
      (hs \ {xtr_H(hi1,hs),xtr_H(hi2,hs)} ∪ hsδ,ls ∪ {l}) end,
    RmvL(li) → ...
  end

xtr_H: HI × H-set  $\leadsto$  H
xtr_H(hi,hs)  $\equiv$  let h:H • h ∈ hs ∧ obs_HI(h)=hi in h end
pre ∃ h:H • h ∈ hs ∧ obs_HI(h)=hi

aLI: H × LI → H, sLI: H × LI → H
aLI(h,li) as h'
  pre li ∉ obs_LIs(h)
  post obs_LIs(h') = {li} ∪ obs_LIs(h) ∧ ...
sLI(h',li) as h
  pre li ∈ obs_LIs(h')
  post obs_LIs(h) = obs_LIs(h') \ {li} ∧ ...

```

The ellipses, \dots , shall indicate that previous properties of h holds for h' .

Narrative — Events

By an *event* of a domain we shall here mean an *instantaneous change of domain state* (here, for example, “the” net state) *not directly brought about by some willed action of the domain but either by “external” forces or implicitly, as an unintended result of a willed action.*

Among the “zillions” of events that may occur in transportation we single out just one. A link of a net ceases to exist as a link.¹

In order to model transportation events we — ad hoc — introduce a transportation state notion of a net paired with some — ad hoc — “conglomerate” of remaining state concepts referred to as $\omega : \Omega$.

Formalisation — Events

```

type
  Link_Disruption == LiDi(li:LI)
channel
  x:(Link_Disruption|...)
value
  transportation_transition: (N × Ω) → in x (N × Ω)
  transportation_transition(n,ω)  $\equiv$ 
    ...

```

```

    [] let xv = x? in
      case xv of
        LiDi(li) → (int_NetOp(RmvL(li))(hs,ls),line_dis(ω))
        ...
      end end
    [] ...

line_dis:  $\Omega \rightarrow \Omega$ 

```

Narrative — Behaviours

By a *behaviour* we mean a possibly infinite sequence of zero, one or more actions and events.

We illustrate just one of very many possible transportation behaviours.

A net behaviour is a sequence of zero, one or more executed net operations: the openings (insertions) of new links (and implied hubs) and the closing (removals) of existing links (and implied hubs), and occurrences of external events (limited here to link disruptions).

Formalisation — Behaviours

channel

x:...

value

transportation_transition: $(N \times \Omega) \rightarrow \text{in } x \ (N \times \Omega)$

transportation_transition(n,ω) ≡

...

[] let xv = x? in case xv of ... end end

[] let op:NetOp • pre IntNetOp(op)(n) in IntNetOp(op)(n) end

...

transportation: $(N \times \Omega) \rightarrow \text{in } x \ \text{Unit}$

transportation(n,ω) ≡

let (n',ω') = transportation_transition(n,ω) in

transportation (n',ω') end

D.3.3 Domain Modelling: Describing Facets

In this, a major, methodology section of the current paper we shall focus on principles and techniques domain modelling, that is, developing abstractions and verifying properties. We shall only cover ‘developing abstractions’.

Domain modelling, as we shall see, entails modelling a number of domain facets.

By a *domain facet* we mean *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.*

These are the facets that we find “span” a domain in a pragmatically sound way: intrinsics, support technology, management & organisation, rules & regulations, scripts and human behaviour: We shall now survey these facets.

Domain Intrinsics

By *domain intrinsics* we mean *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.*

In the large example of Sect. D.3.2, we claim that the net, hubs and links were intrinsic phenomena of the transportation domain; and that the operations of joining and removing links were not: one can explain transportation without these operations. We will now augment the domain description of Sect. D.3.2 with an intrinsic concept, namely that of the states of hubs and links: where these states indicate desirable directions of flow of movement.

A Transportation Intrinsics — Narrative

With a hub we can associate a concept of hub state. The pragmatics of a hub state is that it indicates desirable directions of flow of vehicle movement from (incoming) links to (outgoing) links. The syntax of indicating a hub state is (therefore) that of a possibly empty set of triples of two link identifiers and one hub identifier where the link identifiers are those observable from the identified hub.

With a link we can associate a concept of link state. The pragmatics of a link state is that it indicates desirable directions of flow of vehicle movement from (incoming, identified) hubs to (outgoing, identified) hubs along an identified link. The syntax of indicating a link state is (therefore) that of a possibly empty set of triples of pairs of identifiers of link connected hub and a link identifier where the hub identifiers are those observable from the identified link.

A Transportation Intrinsics — Formalisation

type

$X = LI \times HI \times LI$ [crossings **of** a hub]

$P = HI \times LI \times HI$ [paths **of** a link]

$H\Sigma = X\text{-set}$ [hub states]

```

LΣ = P-set [link states]
value
  obs_HΣ: H → HΣ
  obs_LΣ: L → LΣ
  xtr_Xs: H → X-set, xtr_Ps: L → P-set
  xtr_Xs(h) ≡ { (li,hi,li') | li,li':LI,hi:HI • {li,li'} ⊆ obs_LIs(h) ∧ hi=obs_HI(h) }
  xtr_Ps(l) ≡ { (hi,li,hi') | hi,hi':HI,li:LI • {hi,hi'} = obs_HIs(l) ∧ li=obs_LI(l) }
axiom
  ∀ n:N, h:H; l:L • h ∈ obs_Hs(n) ∧ l ∈ obs_Ls(n) ⇒
    obs_HΣ(h) ⊆ xtr_Xs(h) ∧ obs_LΣ(l) ⊆ xtr_Ps(l)

```

Domain Support Technologies

By *domain support technologies* we mean ways and means of implementing certain observed phenomena or certain conceived concepts.

A Transportation Support Technology Facet — Narrative, 1

Earlier we claimed that the concept of hub and link states was an intrinsic facet of transport nets. But we did not describe how hubs or links might change state, yet hub and link state changes should also be considered intrinsic facets. We there introduce the notions of hub and link state spaces and hub and link state changing operations. A hub (link) state space is the set of all states that the hub (link) may be in. A hub (link) state changing operation can be designated by the hub and a possibly new hub state (the link and a possibly new link state).

A Transportation Support Technology Facet — Formalisation, 1

```

type
  HΩ = HΣ-set, LΩ = LΣ-set
value
  obs_HΩ: H → HΩ, obs_LΩ: L → LΩ
axiom
  ∀ h:H • obs_HΣ(h) ∈ obs_HΩ(h) ∧ ∀ l:L • obs_LΣ(l) ∈ obs_LΩ(l)
value
  chg_HΣ: H × HΣ → H, chg_LΣ: L × LΣ → L
  chg_HΣ(h, hσ) as h'
    pre hσ ∈ obs_HΩ(h) post obs_HΣ(h')=hσ
  chg_LΣ(l, lσ) as l'
    pre lσ ∈ obs_LΩ(l) post obs_LΣ(l')=lσ

```

A Transportation Support Technology Facet — Narrative, 2

Well, so far we have indicated that there is an operation that can change hub and link states. But one may debate whether those operations shown are really examples of a support technology. (That is, one could equally well claim that they remain examples of intrinsic facets.) We may accept that and then ask the question: How to effect the described state changing functions ? In a simple street crossing a semaphore does not instantaneously change from red to green in one direction while changing from green to red in the cross direction. Rather there are intermediate sequences of green/yellow/red and red/yellow/green states to help avoid vehicle crashes and to prepare vehicle drivers. Our “solution” is to modify the hub state notion.

*A Transportation Support Technology Facet — Formalisation, 2***type**

Colour == red | yellow | green
 $X = LI \times HI \times LI \times \text{Colour}$ [crossings of a hub]
 $H\Sigma = X\text{-set}$ [hub states]

value

obs_HΣ: $H \rightarrow H\Sigma$, xtr_Xs: $H \rightarrow X\text{-set}$
 $\text{xtr_Xs}(h) \equiv \{(li, hi, li', c) \mid$
 $li, li': LI, hi: HI, c: \text{Colour} \bullet \{li, li'\} \subseteq \text{obs_LI}(h) \wedge hi = \text{obs_HI}(h)\}$

axiom

$\forall n: N, h: H \bullet h \in \text{obs_Hs}(n) \Rightarrow \text{obs_H}\Sigma(h) \subseteq \text{xtr_Xs}(h) \wedge$
 $\forall (li1, hi2, li3, c), (li4, hi5, li6, c'): X \bullet$
 $\{(li1, hi2, li3, c), (li4, hi5, li6, c')\} \subseteq \text{obs_H}\Sigma(h) \wedge$
 $li1 = li4 \wedge hi2 = hi5 \wedge li3 = li6 \Rightarrow c = c'$

A Transportation Support Technology Facet — Narrative, 3

We consider the colouring, or any such scheme, an aspect of a support technology facet. There remains, however, a description of how the technology that supports the intermediate sequences of colour changing hub states.

We can think of each hub being provided with a mapping from pairs of “stable” (that is non-yellow coloured) hub states $(h\sigma_i, h\sigma_f)$ to well-ordered sequences of intermediate “un-stable” (that is yellow coloured) hub states paired with some time interval information $\langle (h\sigma', t\delta'), (h\sigma'', t\delta''), \dots, (h\sigma'^{\dots'}, t\delta'^{\dots'}) \rangle$ and so that each of these intermediate states can be set, according to the time interval information,² before the final hub state $(h\sigma_f)$ is set.

*A Transportation Support Technology Facet — Formalisation, 3***type**

TI [time interval]


```

Signalling = (HΣ × TI)*
Sema = (HΣ × HΣ)  $\xrightarrow{m}$  Signalling
value
  obs_Sema: H → Sema,
  chg_HΣ: H × HΣ → H,
  chg_HΣ_Seq: H × HΣ → H
  chg_HΣ(h, hσ) as h' pre hσ ∈ obs_HΩ(h) post obs_HΣ(h') = hσ
  chg_HΣ_Seq(h, hσ) ≡
    let sigseq = (obs_Sema(h))(obs_Σ(h), hσ) in sigseq(h)(sigseq) end

  sig_seq: H → Signalling → H
  sig_seq(h)(sigseq) ≡
    if sigseq = ⟨ ⟩ then h else
    let (hσ, tδ) = hd sigseq in
    let h' = chg_HΣ(h, hσ); wait tδ;
    sig_seq(h')(tl sigseq) end end end

```

Domain Management & Organisation

By *domain management* we mean people (such decisions) (i) who (which) determine, formulate and thus set standards (cf. rules and regulations, a later lecture topic) 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.

We use the connective ‘&’ (ampersand) in lieu of the connective ‘and’ in order to emphasise that the joined concepts (A & B) hang so tightly together that it does not make sense to discuss one without discussing the other.

By *domain organisation* we mean the structuring of management and non-management staff levels; the allocation of strategic, tactical and operational concerns to within management and non-management staff levels; and hence the “lines of command”: who does what and who reports to whom — administratively and functionally.

A Transportation Management & Organisation Facet — Narrative

In the previous section on support technology we did not describe who or which “ordered” the change of hub states. We could claim that this might very well be a task for management.

(We here look aside from such possibilities that the domain being modelled has some further support technology which advises individual hub controllers

as when to change signals and then into which states. We are interested in finding an example of a management & organisation facet — and the upcoming one might do!)

So we think of a ‘net hub state management’ for a given net. That management is divided into a number of ‘sub-net hub state managements’ where the sub-nets form a partitioning of the whole net. For each sub-net management there are two kinds management interfaces: one to the overall hub state management, and one for each of interfacing sub-nets. What these managements do, what traffic state information they monitor, etcetera, you can yourself “dream” up. Our point is this: We have identified a management organisation.

A Transportation Management & Organisation Facet — Formalisation

type

$\text{HIsLIs} = \text{HI-set} \times \text{LI-set}$

$\text{MgtNet}' = \text{HIsLIs} \times \mathbb{N}$

$\text{MgtNet} = \{ | \text{mgtnet} : \text{MgtNet}' \bullet \text{wf_MgtNet}(\text{mgtnet}) | \}$

$\text{Partitioning}' = \text{HIsLIs-set} \times \mathbb{N}$

$\text{Partitioning} = \{ | \text{partitioning} : \text{Partitioning}' \bullet \text{wf_Partitioning}(\text{partitioning}) | \}$

value

$\text{wf_MgtNet} : \text{MgtNet}' \rightarrow \mathbf{Bool}$

$\text{wf_MgtNet}((\text{his}, \text{lis}), n) \equiv$

[The his component contains all the hub ids. of links identified in lis]

$\text{wf_Partitioning} : \text{Partitioning}' \rightarrow \mathbf{Bool}$

$\text{wf_Partitioning}(\text{hisliss}, n) \equiv$

$\forall (\text{his}, \text{lis}) : \text{HIsLIs} \bullet (\text{his}, \text{lis}) \in \text{hisliss} \Rightarrow \text{wf_MgtNet}((\text{his}, \text{lis}), n) \wedge$

[no sub-net overlap and together they “span” n]

Etcetera.

Domain Rules & Regulations

Domain Rules

By a *domain rule* we mean some text (in the domain) which prescribes how people or equipment are expected to behave when dispatching their duty, respectively when performing their function.

Domain Regulations

By a *domain regulation* we mean 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.

A Transportation Rules & Regulations Facet — Narrative

The purpose of maintaining an appropriate set of hub (and link) states may very well be to guide traffic into “smooth sailing” — avoiding traffic accidents etc. But this requires that vehicle drivers obey the hub states, that is, the signals. So there is undoubtedly a rule that says: *Obey traffic signals*. And, in consequence of human nature, overlooking or outright violating signals there is undoubtedly a regulation that says: *Violation of traffic signals is subject to fines and . . .*

A Transportation Rules & Regulations Facet — Formalisation

We shall, regretfully, not show any formalisation of the above mentioned rule and regulation. To do a proper job at such a formalisation would require that we formalise traffics, say as (a type of) continuous functions from time to pairs of net and vehicle positions, that we define a number of auxiliary (traffic monitoring) functions, including such which test whether from one instance of traffic, say at time t to a “next” instance of time, t' , some one or more vehicles have violated the rule³, etc. The “etcetera” is ominous: It implies modelling traffic wardens (police trying to apprehend the “sinner”), ‘etc.’ ! We rough-sketch an incomplete formalisation.

type

T [time]
 V [vehicle]
 Rel_Distance = { | f:Rel • 0 < f < 1 | }
 VPos == VatH(h:H) | VonL(hif:HI,l:L,hit:HI,rel_distance:Rel_Distance)
 Traffic = T → (N × (V \xrightarrow{m} VPos))

value

violations: Traffic → (T × T) → V-set

Vehicle positions are either at hubs or some fraction f down a link (l) from some hub (hit) towards the connected hub (hit). Traffic maps time into vehicle positions. We omit a lengthy description of traffic well-formedness.

Domain Scripts

By a *domain script* we mean *the structured, almost, if not outright, formally expressed, wording of a rule or a regulation that has legally binding power, that is, which may be contested in a court of law.*

A Transportation Script Facet — Narrative

Regular buses ply the network according to some time table. We consider a train time table to be a script. Let us take the following to be a sufficiency narrative description of a train time table. For every train line, identified by a line number unique to within, say a year of operation, there is a list of train hub visits. A train hub visit informs of the intended arrival and departure times at identified hubs (i.e., train stations) such that “neighbouring” hub visits, (t_{a_i}, h_i, t_{d_i}) and (t_{a_j}, h_j, t_{d_j}) , satisfy the obvious that a train cannot depart before it has arrived, and cannot arrive at the next, the “neighbouring” station before it has departed from the previous station, in fact, $t_{a_j} - t_{d_i}$ must be commensurate with the distance between the two stations.

A Transportation Script Facet — Formalisation

```

type
  TLin
  HVis = T × HI × T
  Journey' = HVis*, Journey = { |j:Journey' • len j ≥ 2 | }
  TimTbl' = (TLin  $\xrightarrow{m}$  Journey) × N
  TimTbl = { | timtbl:TimTbl' • wf_TimTbl(timtbl) | }

value
  wf_TimTbl: TimTbl' → Bool
  wf_TimTbl(tt,n) ≡
    [ all hubs designated in tt must be hubs of n ]
    [ and all journeys must be along feasible links of n ]
    [ and with commensurate timing net n constraints ]

```

Domain Human Behaviour

By *human behaviour* we mean 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.

Transportation Human Behaviour Facets — Narrative

We have already exemplified aspects of human behaviour in the context of the transportation domain, namely vehicle drivers not obeying hub states. Other example can be given: drivers moving their vehicle along a link in a non-open direction, drivers waving their vehicle off and on the link, etcetera. Whether rules exists that may prohibit this is, perhaps, irrelevant. In any case we can “speak” of such driver behaviours — and then we ought formalise them !

Transportation Human Behaviour Facets — Formalisation

But we decide not to. For the same reason that we skimmed proper formalisation of the violation of the “obey traffic signals” rule. But, by now, you’ve seen enough formulas and you ought trust that it can be done.

value

off_on_link: Traffic $\rightarrow (T \times T) \xrightarrow{\sim} (V \xrightarrow{\overline{m}} VPos \times VPos)$
 wrong_direction: Traffic $\rightarrow T \xrightarrow{\sim} (V \xrightarrow{\overline{m}} VPos)$

D.3.4 Discussion

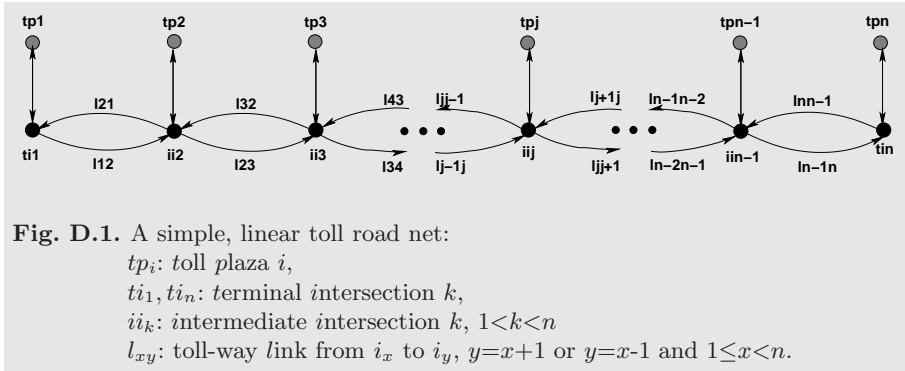
We have given a mere glimpse of a domain description. A full description of a reasonably “convincing” domain description will take years to develop and will fill many pages (hundreds, ...!).

D.4 Requirements Engineering

The objective of requirements engineering is to create a requirements prescription: A requirements prescription specifies externally observable properties of entities, functions, events and behaviours of *the machine* such as the requirements stakeholders wish them to be. The *machine* is what is required: that is, the *hardware* and *software* that is to be designed and which are to satisfy the requirements. A *requirements prescription* thus (*putatively* [206]) expresses what there should be. A requirements prescription expresses nothing about the design of the possibly desired (required) software. We shall show how a major part of a requirements prescription can be “derived” from “its” prerequisite domain description.

The Example Requirements

The domain was that of transportation. The requirements is now basically related to the issuance of tickets upon vehicle entry to a toll road net⁴ and payment of tickets upon the vehicle leaving the toll road net both issuance and collection/payment of tickets occurring at toll booths⁵ which are hubs somehow linked to the toll road net proper. Add to this that vehicle tickets are sensed and updated whenever the vehicle crosses an intermediate toll road intersection.



D.4.1 Stages of Requirements Engineering

The following are the stages of requirements engineering: stakeholder identification, *business process re-engineering*, *domain requirements development*, *interface development*, machine requirements development, requirements verification and validation, and requirements satisfiability and feasibility.

The domain requirements development stage consists of a number of steps: projection, instantiation, determination, extension, and fitting

We shall basically only cover business process re-engineering, domain requirements development, and interface development

D.4.2 Business Process Re-engineering

Business process re-engineering (BPR) re-evaluates the intrinsics, support technologies, management & organisation, rules & regulations, scripts, and human behaviour facets while possibly changing some or all of these, that is, possibly rewriting the corresponding parts of the domain description.

Re-engineering Domain Entities

The net is arranged as a linear sequence of two or more (what we shall call) intersection hubs. Each intersection hub has a single two-way link to (what we shall call) an entry/exit hub (toll plaza); and each intersection hub has either two or four one-way (what we shall call) toll-way links: the first and the last intersection hub (in the sequence) has two toll-way links and all (what we shall call) intermediate intersections has four toll-way links. We introduce a pragmatic notion of net direction: “up” and “down” the net, “from one end to the other”. This is enough to give a hint at the re-engineered domain.

Re-engineering Domain Operations

We first briefly sketch the tollgate Operations. Vehicles enter and leave the toll-way net only at entry/exit hubs (toll plazas). Vehicles collect and return their tickets from and to tollgate ticket issuing, respectively payment machines. Tollgate ticket issuing machines respond to sensor pressure from “passing” vehicles or by vehicle drivers pressing ticket issuing machine button by issuing ticket. Tollgate payment machines accept credit cards, bank notes or coins in designated currencies as payment and returns any change.

We then briefly introduce and sketch an operation performed when vehicles cross intersections: The vehicle is assume to possess the ticket issued upon entry (in)to the net (at a tollgate). At the crossing of each intersection, by a vehicle, its ticket is sensed and is updated with the fact that the vehicle crossed the intersection.

The updated domain description section on support technology will detail the exact workings of these tollgate and internal intersection machines and the domain description section on human behaviour will likewise explore the man/machine facet.

Re-engineering Domain Events

The intersections are highway-engineered in such a way as to deter vehicle entry into opposite direction toll-way links, yet, one never knows, there might still be (what we shall call ghost) vehicles, that is vehicles which have somehow defied the best intentions, and are observed moving along a toll-way link in the wrong direction.

Re-engineering Domain Behaviours

The intended behaviour of a vehicle of the toll-way is to enter at an entry hub (collecting a ticket at the toll gate), to move to the associated intersection, to move into, where relevant, either an upward or a downward toll-way link, to proceed (i.e., move) along a sequence of one or more toll-way links via connecting intersections, until turning into an exit link and leaving the net at an exit hub (toll plaza) while paying the toll.

• • •

This should be enough of a BPR rough sketch for us to meaningfully proceed to requirements prescription proper.

D.4.3 Domain Requirements Prescription

A domain requirements prescription is that part of the overall requirements prescription which can be expressed solely using terms from the domain de-

scription. Thus to construct the domain requirements prescription all we need is collaboration with the requirements stakeholders (who, with the requirements engineers, developed the BPR) and the possibly rewritten (resulting) domain description.

Domain Projection

By *domain projection* we mean a subset of the domain description, one which leaves out all those entities, functions, events, and (thus) behaviours that the stakeholders do not wish represented by the machine.

The resulting document is a *partial domain requirements prescription*.

Domain Projection — Narrative

We copy the domain description and call the copy a 0th version domain requirements prescription. From that document we remove all mention of link insertion and removal functions, to obtain a 1st version domain requirements prescription.⁶

Domain Projection — Formalisation

We do not show the resulting formalisation.

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 entities, functions, events, and (thus) behaviours of the partial domain requirements prescription more concrete, more specific. Instantiations usually render these concepts less general.

Domain Instantiation — Narrative

The 1st version domain requirements prescription is now updated with respect to the properties of the toll way net: We refer to Fig. D.1 and the preliminary description given in Sect. D.4.2. There are three kinds of hubs: tollgate hubs and intersection hubs: terminal intersection hubs and proper, intermediate intersection hubs. Tollgate hubs have one connecting two way link. linking the tollgate hub to its associated intersection hub. Terminal intersection hubs have three connecting links: one, a two way link, to a tollgate hub, one one way link emanating to a next up (or down) intersection hub, and one one way link incident upon this hub from a next up (or down) intersection hub. Proper intersection hubs have five connecting links: one, a two way link, to a

tollgate hub, two one way links emanating to next up and down intersection hubs, and two one way links incident upon this hub from next up and down intersection hub. (Much more need be narrated.) As a result we obtain a 2nd version domain requirements prescription.

Domain Instantiation — Formalisation, Toll Way Net

type

$TN = ((H \times L) \times (H \times L \times L))^* \times H \times (L \times H)$

value

$abs_N: TN \rightarrow N$

$abs_N(tn) \equiv (tn_hubs(tn), tn_hubs(tn))$

pre $wf_TN(tn)$

$tn_hubs: TN \rightarrow H\text{-set},$

$tn_hubs(hll, h, (_, hn)) \equiv$

$\{h, hn\} \cup \{thj, hj | ((thj, tlj), (hj, lj, lj')) : ((H \times L) \times (H \times L \times L)) \bullet ((thj, tlj), (hj, lj, lj')) \in \mathbf{elems} \ hll\}$

$tn_links: TN \rightarrow L\text{-set}$

$tn_links(hll, (_, ln, _)) \equiv$

$\{ln\} \cup \{tlj, lj, lj' | ((thj, tlj), (hj, lj, lj')) : ((H \times L) \times (H \times L \times L)) \bullet ((thj, tlj), (hj, lj, lj')) \in \mathbf{elems} \ hll\}$

theorem $\forall tn:TN \bullet wf_TN(tn) \Rightarrow wf_N(abs_N(tn))$

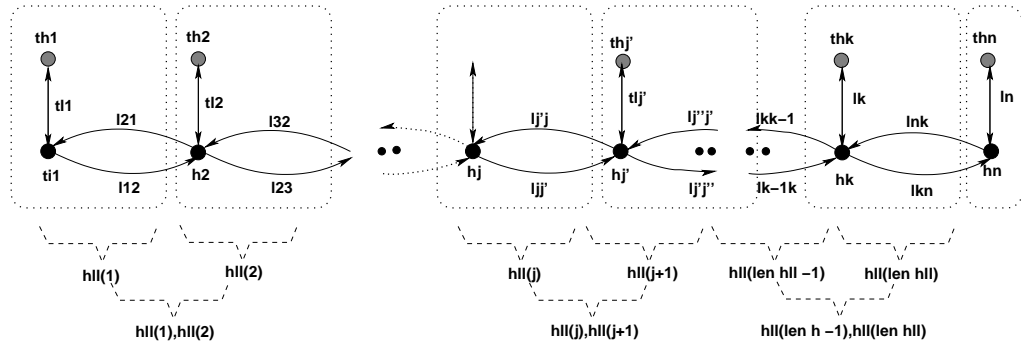


Fig. D.2. A simple, linear toll road net:

th_i : toll plaza i ,

h_1, h_n : terminal intersections,

h_2, h_j, h'_j, h_k : intermediate intersections, $1 < j \leq k, k = n-1$

l_{xy}, l_{yx} : toll-way link from h_x to h_y and from h_y to h_x , $1 \leq x < n$.

l_{x-1x}, l_{xx-1} : toll-way link from h_{x-1} to h_x and h_x to h_{x-1} , $1 \leq x < n$,

dashed links are not in formulas.

Domain Instantiation — Formalisation, Well-formedness

```

type
  LnkM == plaza | way
value
  wf_TN: TN → Bool
  wf_TN(tn:(hll,h,(ln,hn))) ≡
    wf_Toll_Lnk(h,ln,hn)(plaza) ∧ wf_Toll_Ways(hll,h) ∧
    wf_State_Spaces(tn) [to be defined under Determination]

value
  wf_Toll_Ways: ((H×L)×(H×L×L))* × H → Bool
  wf_Toll_Ways(hll,h) ≡
    ∀ j:ℕ • {j,j+1} ⊆ inds hll ⇒
      let ((thj,tlj),(hj,ljj',lj'j)) = hll(j),
        (__,(hj',__,__)) = hll(j+1) in
        wf_Toll_Lnk(thj,tlj,hj)(plaza) ∧
        wf_Toll_Lnk(hj,ljj',hj')(way) ∧ wf_Toll_Lnk(hj',lj'j,hj)(way) end ∧
    let ((thk,tlk),(hk,lk,lk')) = hll(len hll) in
      wf_Toll_Lnk(thk,tlk,hk)(plaza) ∧
      wf_Toll_Lnk(hk,lk,hk')(way) ∧ wf_Toll_Lnk(hk',lk',hk)(way) end

value
  wf_Toll_Lnk: (H×L×H) → LnkM → Bool
  wf_Toll_Lnk(h,l,h')(m) ≡
    obs_Ps(l) = {(obs_HI(h),obs_LI(l),obs_HI(h')),
                  (obs_HI(h'),obs_LI(l),obs_HI(h))} ∧
    obs_Σ(l) = case m of
      plaza → obs_Ps(l),
      way → {(obs_HI(h),obs_LI(l),obs_HI(h'))} end

```

Domain 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 entities, functions, events, and (thus) behaviours of the partial domain requirements prescription less non-determinate, more determinate. Instantiations usually render these concepts less general.

Domain Determination — Narrative

We single out only two 'determinations': The link state spaces There is only one link state: the set of all paths through the link, thus any link state space is the singleton set its only link state. The hub state spaces are the singleton sets of the "current" hub states which allow these crossings: from terminal link back to terminal link, from terminal link to emanating toll-way link, from incident toll-way link to terminal link, and from incident toll-way link to emanating toll-way link Special provision must be made for expressing the entering from the outside and leaving toll plazas to the outside.

Domain Determination — Formalisation

```

wf_State_Spaces: TN  $\rightarrow$  Bool
wf_State_Spaces(hll,hn,(thn,tln))  $\equiv$ 
  let ((th1,tl1),(h1,l12,l21)) = hll(1),
      ((thk,ljk),(hk,lkn,lkn)) = hll(len hll) in
    wf_Plaza(th1,tl1,h1)  $\wedge$  wf_Plaza(thn,tln,hn)  $\wedge$ 
    wf_End(h1,tl1,l12,l21,h2)  $\wedge$  wf_End(hk,tln,lkn,lkn,hn)  $\wedge$ 
     $\forall j:\mathbf{Nat} \bullet \{j,j+1,j+2\} \subseteq \mathbf{inds} \text{ hll} \Rightarrow$ 
      let (, (hj,ljj,lj'j)) = hll(j),
          ((thj',tlj'),(hj',lj'j',lj'j')) = hll(j+1) in
        wf_Plaza(thj',tlj',hj')  $\wedge$ 
        wf_Interm(ljj,lj'j,hj',tlj',lj'j',lj'j') end end

wf_Plaza(th,tl,h)  $\equiv$ 
  obs_H $\Sigma$ (th) = { [ crossings at toll plazas ]
    ("external",obs_HI(th),obs_LI(tl)),
    (obs_LI(tl),obs_HI(th),"external"),
    (obs_LI(tl),obs_HI(th),obs_LI(tl)) }  $\wedge$ 
  obs_H $\Omega$ (th) = {obs_H $\Sigma$ (th)}  $\wedge$ 
  obs_L $\Omega$ (tl) = {obs_L $\Sigma$ (tl)}

wf_End(h,tl,l')  $\equiv$ 
  obs_H $\Sigma$ (h) = { [ crossings at 3-link end hubs ]
    (obs_LI(tl),obs_HI(h),obs_LI(tl)),
    (obs_LI(tl),obs_HI(h),obs_LI(l)),
    (obs_LI(l'),obs_HI(h),obs_LI(tl)),
    (obs_LI(l'),obs_HI(h),obs_LI(l)) }  $\wedge$ 
  obs_H $\Omega$ (h) = {obs_H $\Sigma$ (h)}  $\wedge$ 
  obs_L $\Omega$ (l) = {obs_L $\Sigma$ (l)}  $\wedge$ 
  obs_L $\Omega$ (l') = {obs_L $\Sigma$ (l')}

```

$$\begin{aligned}
& \text{wf_Interm}(\text{ul_1}, \text{dl_1}, \text{h}, \text{tl}, \text{ul}, \text{dl}) \equiv \\
& \text{obs_H}\Sigma(\text{h}) = \{ [\text{crossings at properly intermediate, 5-link hubs}] \\
& \quad (\text{obs_LI}(\text{tl}), \text{obs_HI}(\text{h}), \text{obs_LI}(\text{tl})), \\
& \quad (\text{obs_LI}(\text{tl}), \text{obs_HI}(\text{h}), \text{obs_LI}(\text{dl_1})), \\
& \quad (\text{obs_LI}(\text{tl}), \text{obs_HI}(\text{h}), \text{obs_LI}(\text{ul})), \\
& \quad (\text{obs_LI}(\text{ul_1}), \text{obs_HI}(\text{h}), \text{obs_LI}(\text{tl})), \\
& \quad (\text{obs_LI}(\text{ul_1}), \text{obs_HI}(\text{h}), \text{obs_LI}(\text{ul})), \\
& \quad (\text{obs_LI}(\text{ul_1}), \text{obs_HI}(\text{h}), \text{obs_LI}(\text{dl_1})), \\
& \quad (\text{obs_LI}(\text{dl}), \text{obs_HI}(\text{h}), \text{obs_LI}(\text{tl})), \\
& \quad (\text{obs_LI}(\text{dl}), \text{obs_HI}(\text{h}), \text{obs_LI}(\text{dl_1})), \\
& \quad (\text{obs_LI}(\text{dl}), \text{obs_HI}(\text{h}), \text{obs_LI}(\text{ul})) \} \wedge \\
& \text{obs_H}\Omega(\text{h}) = \{ \text{obs_H}\Sigma(\text{h}) \} \wedge \\
& \text{obs_L}\Omega(\text{tl}) = \{ \text{obs_L}\Sigma(\text{tl}) \} \wedge \\
& \text{obs_L}\Omega(\text{ul}) = \{ \text{obs_L}\Sigma(\text{ul}) \} \wedge \\
& \text{obs_L}\Omega(\text{dl}) = \{ \text{obs_L}\Sigma(\text{dl}) \}
\end{aligned}$$

Not all determinism issues above have been fully explained. But for now we should — in principle — be satisfied.

Domain Extension

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

Domain Extension — Narrative

The domain extension is that of the controlled access of vehicles to and departure from the toll road net: the entry to (and departure from) tollgates from (respectively to) an "an external" net — which we do not describe; the new entities of tollgates with all their machinery; the user/machine functions: upon entry: driver pressing entry button, tollgate delivering ticket; upon exit: driver presenting ticket, tollgate requesting payment, driver providing payment, etc.

One added (extended) domain requirements: as vehicles are allowed to cruise the entire net payment is a function of the totality of links traversed, possibly multiple times. This requires, in our case, that tickets be made such as to be sensed somewhat remotely, and that intersections be equipped with sensors which can record and transmit information about vehicle intersection crossings. (When exiting the tollgate machine can then access the exiting vehicles sequence of intersection crossings — based on which a payment fee calculation can be done.)

All this to be described in detail — including all the thinks that can go wrong (in the domain) and how drivers and tollgates are expected to react.

Domain Extension — Formalisation

We suggest only some signatures:

type

Mach, Ticket, Cash, Payment, Map_TN

value

obs_Cash: Mach \rightarrow Cash, obs_Tickets: M \rightarrow Ticket-**set**

obs_Entry, obs_Exit: Ticket \rightarrow HI, obs_Ticket: V \rightarrow (Ticket|nil)

calculate_Payment: (HI \times HI) \rightarrow Map_TN \rightarrow Payment

press_Entry: M \rightarrow M \times Ticket [gate up]

press_Exit: M \times Ticket \rightarrow M \times Payment

payment: M \times Payment \rightarrow M \times Cash [gate up]

Domain Extension — Formalisation of Support Technology

This example provides a classical requirements engineering setting for embedded, safety critical, real-time systems, requiring, ultimately, the techniques and tools of such things as Petri nets, statecharts, message sequence charts or live sequence charts and temporal logics (DC, TLA+).

Requirements Fitting

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.

By requirements fitting we mean *a harmonisation of $n > 1$ domain requirements that have overlapping (common) not always consistent parts and which results in n ‘modified and partial domain requirements’, and m ‘common domain requirements’ that “fit into” to two or more of the ‘modified and partial domain requirements’.*

By a *modified and partial domain requirements* we mean a domain requirements which is short of (that is, is missing) some description parts: text and formula. By a *common domain requirements* we mean a domain requirements. By the m common domain requirements parts, *cdrs*, *fitting into* the n modi-

fied and partial domain requirements we mean that there is for each modified and partial domain requirements, $mapdr_i$, an identified subset of $cdrs$ (could be all of $cdrs$), $scdrs$, such that textually conjoining $scdrs$ to $mapdr$ can be claimed to yield the “original” d_{r_i} .

Requirements Fitting Procedure — A Sketch

Requirements fitting consists primarily of a pragmatically determined sequence of analytic and synthetic (‘fitting’) steps. It is first decided which n domain requirements documents to fit. Then a ‘manual’ analysis is made of the selected, n domain requirements. During this analysis tentative common domain requirements are identified. It is then decided which m common domain requirements to single out. This decision results in a tentative construction of n modified and partial domain requirements. An analysis is made of the tentative modified and partial and also common domain requirements. A decision is then made whether to accept the resulting documents or to iterate the steps above.

Requirements Fitting — Narrative

We postulate two domain requirements: We have outlined a domain requirements development for software support for a toll road system. We have earlier hinted at domain operations related to insertion of new and removal of existing links and hubs. We can therefore postulate that there are two domain requirements developments, both based on the transport domain: one, $d_{r_{toll}}$, for a toll road computing system monitoring and controlling vehicle flow in and out of toll plazas, and another, $d_{r_{maint.}}$, for a toll link and intersection (i.e., hub) building and maintenance system monitoring and controlling link and hub quality and for development.

The fitting procedure now identifies the shared of awareness of the net by both $d_{r_{toll}}$ and $d_{r_{maint.}}$ of nets (N), hubs (H) and links (L). We conclude from this that we can single out a common requirements for software that manages net, hubs and links. Such software requirements basically amounts to requirements for a database system. A suitable such system, say a relational database management system, DB_{rel} , may already be available with the customer.

In any case, where there before were two requirements ($d_{r_{toll}}, d_{r_{maint.}}$) there are now four: (i) $d'_{r_{toll}}$, a modification of $d_{r_{toll}}$ which omits the description parts pertaining to the net; (ii) $d'_{r_{maint.}}$, a modification of $d_{r_{maint.}}$ which likewise omits the description parts pertaining to the net; (iii) $d_{r_{net}}$, which contains what was basically omitted in $d'_{r_{toll}}$ and $d'_{r_{maint.}}$; and (iv) $d_{r_{db:i/f}}$ (for database interface) which prescribes a mapping between type names of $d_{r_{net}}$ and relation and attribute names of DB_{rel} .

Much more can and should be said, but this suffices as an example in a software engineering methodology paper.

Requirements Fitting — Formalisation

We omit lengthy formalisation.

Domain Requirements Consolidation

After projection, instantiation, determination, extension and fitting, it is time to review, consolidate and possibly restructure (including re-specify) the domain requirements prescription before the next stage of requirements development.

D.4.4 Interface Requirements Prescription

By an *interface requirements* we mean a *requirements prescription* which *refines* and *extends* the domain requirements by considering those requirements of the domain requirements whose entities, operations, events and behaviours are “*shared*” between the domain and the machine (being requirements prescribed).

‘Sharing’ means (a) that an *entity* is represented both in the domain and “inside” the machine, and that its machine representation must at suitable times reflect its state in the domain; (b) that an *operation* requires a sequence of several “on-line” interactions between the machine (being requirements prescribed) and the domain, usually a person or another machine; (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 (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 entities, operations, events and behaviours.

Each such shared phenomenon shall then be individually dealt with: *entity sharing* shall lead to interface requirements for data initialisation and refreshment; *operation 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. *behaviour sharing* shall lead to interface requirements for action and event dialogues between the machine and its environment.

• • •

We shall now illustrate these domain interface requirements development steps with respect to our ongoing example.

Shared Entities

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

We “split” our interface requirements development into two separate steps: the development of $d_{r_{\text{net}}}$ (the common domain requirements for the shared hubs and links), and the co-development of $d_{r_{\text{db:i/f}}}$ (the common domain requirements for the interface between $d_{r_{\text{net}}}$ and DB_{rel} — under the assumption of an available relational database system DB_{rel}).

When planning the common domain requirements for the net, i.e., the hubs and links, we enlarge our scope of requirements concerns beyond the two so far treated ($d_{r_{\text{toll}}}$, $d_{r_{\text{maint.}}}$) in order to make sure that the shared relational database of nets, their hubs and links, may be useful beyond those requirements. We then come up with something like hubs and links are to be represented as tuples of relations; each net will be represented by a pair of relations a hubs relation and a links relation; each hub and each link may or will be represented by several tuples; etcetera. In this database modelling effort it must be secured that “standard” operations on nets, hubs and links can be supported by the chosen relational database system DB_{rel} .

Data Initialisation

As part of $d_{r_{\text{net}}}$ one must prescribe data initialisation, that is provision for an interactive user interface dialogue with a set of proper display screens, one for establishing net, hub or link attributes (names) and their types and, for example, two for the input of hub and link attribute values. Interaction prompts may be prescribed: next input, on-line vetting and display of evolving net, etc. These and many other aspects may therefore need prescriptions.

Essentially these prescriptions concretise the insert link operation.

Data Refreshment

As part of $d_{r_{\text{net}}}$ one must also prescribe data refreshment: an interactive user interface dialogue with a set of proper display screens one for updating net, hub or link attributes (names) and their types and, for example, two for the update of hub and link attribute values. Interaction prompts may be prescribed: next update, on-line vetting and display of revised net, etc. These and many other aspects may therefore need prescriptions.

These prescriptions concretise remove and insert link operations.

Shared Operations

The main shared operations are related to the entry of a vehicle into the toll road system and the exit of a vehicle from the toll road system.

Interactive Operation Execution

As part of $d_{r_{\text{toll}}}$ we must therefore prescribe the varieties of successful and less successful sequences of interactions between vehicles (or their drivers) and the toll gate machines.

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

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

Shared Events

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

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

The prescription of the above necessitates extensive fault analysis.

Shared Behaviours

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

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

D.5 Discussion

D.5.1 An ‘Odyssey’

Our ‘Odyssey’ has ended. A long example has been given.

We have shown that requirements engineering can have an abstraction basis in domain engineering; and we have shown that we do not have to start software development with requirements engineering, but that we can start software development with domain engineering and then proceed to a more orderly requirements engineering phase than witnessed today.

D.5.2 Claims of Contribution

What is essentially new here is the claim and its partial validation that one can and probably should put far more emphasis on domain modelling, the domain modelling concepts, principles and techniques of business process domain intrinsics, domain support technologies, domain management and organisation, domain rules and regulations, domain scripts and domain human behaviour; the identification of, and the decomposition of the requirements development process into, domain requirements, interface requirements and machine requirements; the domain requirements “derivation” concepts, principles and techniques of projection, instantiation, determination, extension and fitting and the identification of structuring of the interface ground requirements shared entities, shared operations, shared events and shared behaviours.

D.5.3 Comparison to Other Work

Jackson’s Problem Frame approach [207] cleverly alternates between domain analysis, requirements development and software design. For more satisfactory comparisons between our domain engineering approach and past practices and writings on domain analysis we refer to [94].

D.5.4 A Critique

A major presentation of domain and of requirements engineering is given in [89, Chaps. 8–16 and 17–24]. [94] provides a summary, more complete presentation of domain engineering than the present paper allows, while [90] discusses a set of research issues for domain engineering. Papers, like [90, 94], but for requirements engineering, with more a complete presentation, respectively a discussion of research issues for this new kind of requirements engineering might be desirable. The current paper’s Sect. D.4 provided a slightly revised structuring of the interface requirements engineering.

Some of the development steps within the domain modelling and likewise within the requirements modelling are refinements, and some are extensions. If we ensure that the extensions are what is known as Conservative extensions then all theorems of the source of the extension go through and are also valid in the extension. Although such things are here rather clear much more should be said here about ensuring Conservative extensions. We do not since the current paper is not aimed at the finer issues of the development but at the domain to requirements “derivation” issues.

D.6 Acknowledgments

I gratefully acknowledge support from Université Henri Poincaré (UHP), Nancy, and from INRIA (l'Institut National de Recherche en Informatique et en Automatique) both of France, for my two month stay at LORIA (Laboratoire Lorrain de Recherche en Informatique et ses Applications), Nancy, in the fall of 2008. I especially and warmly thank Dominique Méry for hosting me. And I thank the organisers of Ugo Montanari's Festschrift, Pierpaolo Degano, Jose Meseguer and Rocco De Nicola, for inviting me — thus forcing me to willingly write this paper.

E

Believable Software Management

This appendix chapter constitutes the invited paper for *Encyclopedia of Software Engineering* [93, to appear], editor: Philip A. Laplante (Taylor & Francis).

Abstract

By ‘believable software management’ we shall understand a “state of software development” (as an established practice) in which all developers, that is, programmers and managers at all levels, in their conscience, are confident that everybody is working according to state-of-the-art methods.

In this Encyclopedia of Software Engineering entry on ‘believable software management’ we shall therefore survey what it means to be a programmer or a manager working according to state-of-the-art methods.

To us, the concept of ‘software management’, covers both software development project management and software product management. We shall therefore have something to say about both. Much about the former and just a tiny bit about the latter. Many aspects of software product management refers to software development project management.

Keywords: software, development, management, project, product, domain engineering, requirements engineering, software design, documents, formal techniques.

E.1 Introduction

To develop other than “own” software on industrial scale, whether turn-key or for COTS¹ appears to be difficult. ‘Own’ software is here defined to be

¹Commercial off-the-shelf (COTS) is a term for software or hardware, generally technology or computer products, that are ready-made and available for sale, lease, or license to the general public (Wikipedia: http://en.wikipedia.org/wiki/Commercial_off-the-shelf).

such software that is developed by a single, knowledgeable programmer and for own, private, uncommitted use only.

“Non-own” software usually requires the resources of more than one programmer. The interaction between such two or more programmers need be managed. Such software is usually committed to be used by other than the programmers who developed and the managers who directed the development of the software, and hence the quality of the software need be assured (i.e., managed).

In this Encyclopedia of Software Engineering paper we shall take a look at state-of-the-art possibilities for professional, industry-strength software development project management.

In order to make ensure that the writer and the reader has a chance of understanding as near the same by “what software and what software development entails”, we characterise these and related concepts in Sects. E.2.2–E.2.2.

And in order to make ensure that the writer and the reader has a chance of understanding as near the same by “what management, in general, entails”, we characterise these and related concepts in Sects. E.2.2–E.2.2.

Although ‘management’ is the main subject of concern in this Encyclopedia of Software Engineering entry we need elaborate on the topic of software (first over-viewed in Sects. E.2.2–E.2.2). We do so in Sect. E.3. This section can, of course, not be an introduction to software development in general — that would require an entire book, for example [87–89]. But we cannot just assume general knowledge of what software development entails. That would set us back considerably. Classical textbooks, [251, 253, 280], should not be assumed as they do not represent a state-of-the-art view of software development, a view which is at the base of several dozens of software houses around the world².

Section E.3 shows several so-called process diagrams and a single software development graph. Based on these a combined notion of project graphs is introduced and a notion of project traces (“routes” of graphs) is introduced. Subsequently a notion of an actual development conforming to the project graph is introduced — all in preparation for Sects. E.4–E.5.

Section E.4 covers two important aspects of software project management, namely that of software process assessment and improvement. Our view here is primarily based on Humphrey’s Capability Maturity Modelling [194], but is enhanced considerably by considerations of the use of formal techniques.

Section E.5 then constitutes the core of this Encyclopedia of Software Engineering entry’s main message. In addition to further coverage of process management, Sect. E.5.1, we cover the management issues related to each and every phase, state and step of software development as outlined in Sect. E.3, and from two points of view: not only the software project, but also the soft-

²Cf., for example, <http://www.fortia.org/twiki/bin/view/Main/ForTIA>, the home page of some 40+ software houses sharing interests in state-of-the-art use of formal techniques in software development.

ware product points of view, Sect. E.5.2. As mentioned above, the treatment of Sect. E.5.2 is cursory. Much more should be covered — but the current author is not “believable” on those other points !

Section E.6 summarises Sects. E.4–E.5 emphasising a notion of ‘believability’.

The presentation is informal. No examples of informal or of formal domain descriptions, requirements prescriptions or software specifications will be shown.

E.2 Background

This and the next section sets the scene for our subsequent treatment of issues of software management.

E.2.1 General

To understand a concept of ‘management’ presupposes an understanding of “that which is being managed” (software and resources). We shall therefore, in the next subsection, characterise a number of concepts related to and including software, resources, management and development methods.

E.2.2 Characterisations

Since we are dealing primarily with very general notions and with issues of human behaviour (including management), we cannot define these concepts succinctly — but we can delineate these notions, we think, sufficiently well by characterisations rather than formal definitions.

Software

By software we mean a collection of documents, in paper and/or in electronic form. This collection comprises the usual kind of material: (1) the executable code for a suitable computing system, (2,3) the training and user manuals, and (4,5) the installation and service manuals. But, to us, the software collection of documents also comprise (6) all the development documents pertaining to the executable code — and this chapter of the Encyclopedia of Software Engineering will indeed show you that the development documents are many and

varied, and (7) the verification³, validation⁴ and test⁵ documents — which, in turn, comprise (8) chronological records of all the verifications, validations and tests being performed and the results of these verifications, validations and tests, whether successful or not. Finally this software collection of documents, to us, also comprises (9) all the tools that management and developers used in managing the development project and developing, verifying, validating and testing, installing, using and maintaining the software.

Software Engineering

By software engineering we mean the study and knowledge about the construction of software.

When putting forth our view on Believable Software Management we shall base that view on a particular view of software engineering. What we have to say about Believable Software Management will be true also for views on software engineering that appear to differ from our view. What we put forward is relevant, we claim, in those other views as well, since our view, the so-called TripTych view, includes all the important software engineering concepts of these other, more classical views.

Our view on software engineering is based on the following dogma:

The TripTych Dogma:

- Before software can be designed we must understand its requirements.
- Before we can prescribe the requirements we must understand the application domain.

Hence we shall view software engineering as comprising three bodies of studies and knowledge:

The TripTych of Software Engineering:

1. domain engineering,
2. requirements engineering and
3. software design.

³Verification aims at showing not only whether the executable code performs correctly, i.e., does, whatever it is supposed to do, in a right manner, i.e., that *the software is right*, but verification also aims at verifying properties of the domain description, the requirements prescription and the higher level software designs.

⁴Validation aims at showing whether the executable code performs whatever the users intended the the executable code to do, i.e., that they got *the right software*, and it does so by validations of the domain description and the requirements prescription.

⁵Testing aims at uncovering errors not only in the executable code but in the domain description and the requirements prescription, so tests are to be performed on domain descriptions, requirements prescriptions, high-level designs and executable code — as are verifications and validations.

As we shall see later in this chapter we may look at these three parts of software engineering as ideally implying three consecutive phases of engineering, i.e., of development.

Further Characterisations

Domain.

By a domain we shall understand a universe of discourse, something in which people and/or equipment made by humans act and hence something about which we, the humans, can speak.⁶

Examples of domains are: (i) financial service institutions [banks, insurance companies, securities trading (stock exchanges), credit cards, etc.], (ii) health care [hospitals, pharmacies, private physicians, rehabilitation, convalescent and other clinics, etc.], (iii) air, railway, road and/or ship transportation [(iii.1) airports, airlines, air traffic; (iii.2) development, monitoring and control of rail nets, train operations (including scheduling and allocation of staff and train car resources, monitoring & control of train movement, etc.), ticket handling,], (iv) manufacturing, (v) “the market” [consumers, retailers, wholesalers, producers, the supply chain, etc.], etcetera.

Domain Description.

By a domain description we shall understand a document which describes a domain **as it is** without any reference to requirements to software to serve in that domain, let alone to such future software.

The domain description, as we have shown in [89, Part IV, Chaps. 8–16], can be expressed both informally, say in narrative form, and formally, in terms of mathematical specifications. If formally presented we assume that that formal presentation is strongly linked to an informal narrative and terminology.

We otherwise refer to [79, 82, 84, 90–92, 94, 96] for a set of examples of domain descriptions.

Domain Engineering.

By domain engineering we understand the study and knowledge about the practice and theory of the methods for developing domain descriptions.

We shall later, in Sect. E.3.2 on page 257, detail stages and steps of a domain engineering method.

⁶We have deliberately constrained our definition, for the sake of this Encyclopedia of Software Engineering entry, to domains “in which people act”. Domains, in general, also cover any area of nature — as otherwise describable in physics. But let us be content with the narrower characterisation.

Machine.

By a machine we understand a combination of computing system hardware and software that is the target for, or the result of the required software development.

Requirements.

By a requirements we understand “A condition or capability needed by a user to solve a problem or achieve an objective” (IEEE Standard 610.12 [195]).

Requirements Prescription.

By a software requirements prescription we shall understand set of documents which prescribe the properties that some desired machine is expected to satisfy.

Requirements Engineering.

By requirements engineering we understand the study and knowledge about the practice and theory of the methods for developing requirements prescriptions.

We shall later, in Sect. E.3.2 on page 259, detail stages and steps of a requirements engineering method.

Software.

For a characterisation of the term ‘software’ we refer to Sect. E.2.2 on page 249.

Software Design.

By a software design we shall understand a set of documents which specify how some software requirements are implemented in the form of executable code.

We shall later, in Sect. E.3.2 on page 261, hint at stages and steps of software design.

Software Development.

By software development we understand the full development of a possibly verified and possibly validated domain description, of a possibly verified and a possibly validated requirements prescription, and of a software design possibly verified correct with respect to the requirements prescription (in the context of the domain description).

Software Engineering.

For a characterisation of the term ‘software engineering’ we refer to Sect. E.2.2 on page 250.

Software House

By a software house we mean a commercial enterprise with owners, management, non-management employees, usually including programmers, who develops and markets software and who provides software service (including computing facilities management⁷) to customers.

Resources

By resources we mean such things as people, time, monies, goodwill⁸, facilities (offices, equipment, software tools and other tangible or intangible entities) and products (spare parts as well as end products, including, as here, software).

Management

Management is a relation between owners of an enterprise and the actors (people, i.e., employees and customers) and resource-consuming activities of that enterprise.

By management we thus mean a structure, an organisation, of people, i.e., the managers, who carry out strategic and tactical management⁹, that is, formulate (decide upon) plans for allocation¹⁰ and scheduling¹¹ of resources and actions upon resources. Strategic and tactical management usually perform some of these actions themselves. Operations management monitors and controls that remaining plans (actions) are indeed carried out.

By quality management we mean management which ensures some measure of resource optimality: that production and service deadlines, cost, and product and service qualities are met, and that the employees (lower levels of managers and non-management employees) enjoy their work and that their professionalism is continually improved.

⁷**Facilities management:** the management, services and operations of computing systems.

⁸**Goodwill:** (1) the favor or advantage that a business has acquired especially through its brands and its good reputation; (2) the value of projected earnings increases of a business especially as part of its purchase price; (3) the excess of the purchase price of a company over its book value which represents the value of goodwill as an intangible asset for accounting purposes [288]

⁹Definitions of strategic, tactical and operational management are given on Page 254.

¹⁰**Allocation:** assignment of some resources other than time to actors or actions. (Actors are also resources, and can thus themselves be assigned.)

¹¹**Scheduling:** assignment of time periods to actions, i.e., the timed performance of development phases, stages or step. [Phase: domain development, requirements development, or software design.]

Strategic Management.

Strategic management is related to a software house's long term commitments and is thus, with respect to 'software', primarily related to issues of software product management.

By strategic management we mean such management decisions and actions that primarily are concerned about the meta-organisational, meta-financial and meta-product issues of an enterprise: management and organisational restructuring, mergers and acquisition, conversion of goodwill to new capital, whether to maintain a current product-line or to abandon it, start a new product line or merge the present product line with new product possibilities, etcetera.

The main aim of strategic management is to meet owner expectations: market share, profit, growth, etcetera.

Tactical Management.

Tactical management is related to a software house's medium term commitments and is thus primarily, with respect to 'software', related to issues of software development and product management.

By tactical project management we mean such management issues and actions that primarily are concerned about the relationships within and between current and immediate future software development projects, that is, issues such as the decomposition of the development of a software product into a staged line, "tree" or "bouquet", of either separately marketable software products, or of basic versions of such products versus the offering of "optional features". Tactical management also formulates guide lines and possibly adopts standards for quality assurance¹².

By tactical product management we mean such management issues and actions that primarily are concerned with the marketing, sales and service of products.

The main aim of tactical management is to compete in the market, to respond to customer demands in a timely and qualitative fashion, and to contribute to the aims of strategic management.

Operations Management.

Operations management is related to the enterprise's short term, i.e., day-to-day commitments and is thus exclusively related to issues of project and product management. The former will be dealt with in detail in this paper. The latter, which includes marketing, sales, the management of customer satisfaction, training, service and product maintenance, will not be covered in this Encyclopedia of Software Engineering entry.

¹²**Quality assurance:** a program for the systematic monitoring and evaluation of the various aspects of a project, service, or facility to ensure that standards of quality are being met [288].

By operations management we mean such management issues that establish detailed, i.e., micro-allocations and -schedules for product development, marketing, sales, training, service and product maintenance and monitors and controls that these micro-allocations and -schedules are indeed carried out in accordance with quality guide lines and standards.

Software Management

By software management we either mean the management of the development, service and maintenance of specific software (i.e., software development management), or we mean the management of which software products to develop (software product management).

Method and Methodology

People, in enterprises, are expected to act professionally. That is, to carry out their responsibilities and duties according to certain methods, that is, methodologically.

Method.

By a method we mean (the study and knowledge about) a set of principles, to be adhered to, fulfilled, by humans, for selecting and applying a number of analysis and synthesis (construction, action) techniques, possibly using tools, in order to achieve a well defined goal: a product design (an artifact) or a service delivery, etcetera.

Methodology.

By methodology we mean the comparative study and knowledge about a set of methods.

In connection with software development there are, as we shall see, many methods being offered. The most prominent feature of most methods are their tools, and the most prominent tools are those of specification and programming languages. So, a language, whether a human, spoken (and written) language, or formal, is also a tool.

E.2.3 Discussion

In this Encyclopedia of Software Engineering entry we shall primarily cover software project (i.e., development) management. Thus our concerns are primarily aimed at software-related operations management; and thus we shall not be concerned, at all, with strategic management and only with some aspects of tactical management. Those aspects not covered are generic, that is, are not specific to software.

E.3 On Software Development Processes

E.3.1 Processes, Process Specifications and Process Models

By a process we mean a set of related sequences (i.e., traces) where each sequence (trace) in the set is an ordered list of actions and events. Actions change a state. Events are like process inputs or outputs. A sequence, p_i , of actions, a_{i_k} , and events, e_{i_ℓ} , may relate to another such sequence, p_j , by sharing an event, e , in the form of the shared event being identical to one of the events, e_{i_m} (i.e., $e_{i_m} \equiv e$), in p_i and one of the events, e_{j_n} (i.e., $e_{j_n} \equiv e$), in p_j such that this event designates the communication between the two processes, that is, their synchronisation and the simultaneous exchange of a resource (or a possibly empty set of resources) between them. (Simple “assignment” actions may then bind these resources to appropriate names of the input process.) A simple process is just a single sequence whose events, if any, communicates with a further undefined environment. A single process is either a simple process or is a single sequence whose events communicates with other processes. Thus processes are, in general, composed from several processes.

Operations management is thus about the detailed monitoring and control of processes: development processes, marketing processes, sales processes, service processes, training processes, etcetera.

In order to manage in a meaningful way, including in a manner where ‘management’ can itself be monitored and controlled, that is, be evaluated and improved, the managed processes must be well understood. We take that as meaning that there must be a reasonably precise specification, that is, a model of the processes to be managed.

By a process specification we mean a syntactic entity: some text and/or diagram(s) that name and specify the actions to be performed and the events that may relate two (or more) processes, including a naming and specification of the resources being, or to be communicated.

By a process model we mean a semantic entity: the meaning of a process specification — with that meaning being a possibly infinite set of sets of processes (i.e., set of sets of traces).

By a process specification-based software development we mean either one of the process sets denoted by the process specification.

E.3.2 Software Development Process Descriptions

We have argued earlier that software development consists of three phases. As we shall soon see, carrying out these phases each result in quite distinct sets of documents. Figure E.1 on the next page shows the three phases as connected by directed DO and REDO labelled edges, i.e., arrows.

The boxes (i.e., the phases) denote processes. DO labelled edges infix two boxes, the from and the to box as designated by the edge direction. The DO labelled arrows shall ideally mean that no activities of the processes of the to box can

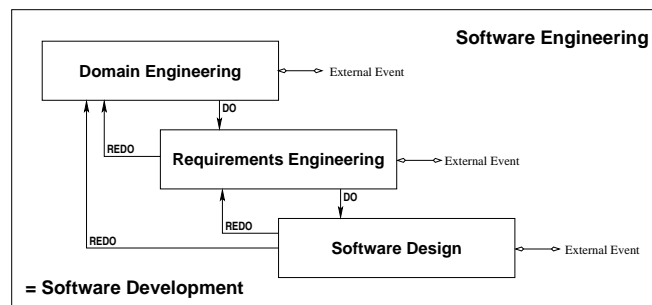


Fig. E.1. The TripTych phases of software development

commence before all activities of the processes of the from box have completed. The DO labelled arrows thus designate “internal” events that synchronise the infixed process and which communicate the documents resulting from the ‘from box’ to the ‘to box’. The REDO labelled edges can be said to infix one ‘from box’ with one or two ‘to boxes’, now in the reverse order of the DO labelled edge infixed boxes. The REDO labelled arrows (i.e., “internal events”) shall ideally mean that all activities of the ‘from box’ must halt once one such activity requires that earlier work be redone. We do not here detail which ‘to box’ is selected nor how the development then proceeds.

The “dangling”, but bi-directed (“external event”) edges designate that box processes require input from or delivers output to an external world (an external process) — typically the human developers.

All directed or bi-directed edges also designate the communication of documents. We do not here detail how these documents are otherwise produced or consumed.

Phases consists of stages and stages of steps. The phases are logically well distinguished. “Boundaries” between stages or steps are pragmatically justified. Next we shall cover the stage and step concepts.

Domain Engineering

The top-left box of Fig. E.1 is shown in detail in Fig. E.2 on the next page. External edges designate the input of document information (including answers to clarifying question) from stakeholders and output of documents and questions to stakeholders including software development management.

Figure E.2 on the following page expresses that domain engineering consists of many stages and that (some of) these stages consists of many steps. We now explain these stages. We cover the left part of Fig. E.2 on the next page first.

(a) First the stakeholders of the domain are identified and arrangements made for future liaison. (b–f) Knowledge about the domain is acquired in five steps; (b) by studying existing literature, the Web, observations in the domain, etc.; (c)

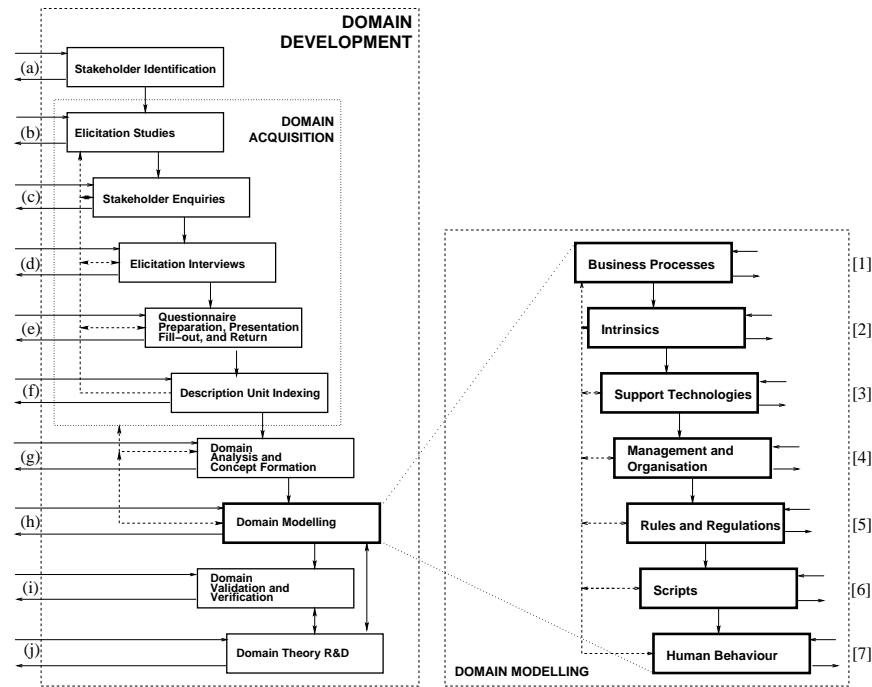


Fig. E.2. Stages and steps of the domain engineering phase

contacts with the stakeholders; (d) talks with these; (e) helping them filling out questionnaires, that is, collecting domain description units; (f) and sorting these out. (g) Domain description units are then analysed and preliminary concepts may be formed. (h) Based on this analysis the major work on domain description is carried out. (i) The domain description is then analysed, verified and validated. (j) Finally, where relevant, properties not explicitly formulated in the domain description are established. [1–7] The major stage, (h), of engineering a domain description (that is, of domain modelling, right part of Fig. E.2) consists of six steps [2–7], each covering a facet of the domain. [1] But first rough sketches are made of all the most pertinent business processes (that is, the entities, the set of functions and events over entities, and the behaviours) of the domain. [2] Based on such sketches the very basics, the entities, functions, events and behaviours that are common to all subsequent facets are described. [3] Then the support technologies of the domain, those which support entities, functions, events and behaviours of the domain are described. [4] The management functions and organisational structures are described. [5] And so are the rules and regulations which (ought) “govern” human behaviour in the domain. [6] Some specific structures (somehow ordered sets) of rules and regulations qualify as scripts, that is, as pseudo-programs, and these are described. [7] Finally the spectrum of possible

or actual human behaviours are described: diligent, sloppy, negligent as well as near- or outright criminal behaviours.

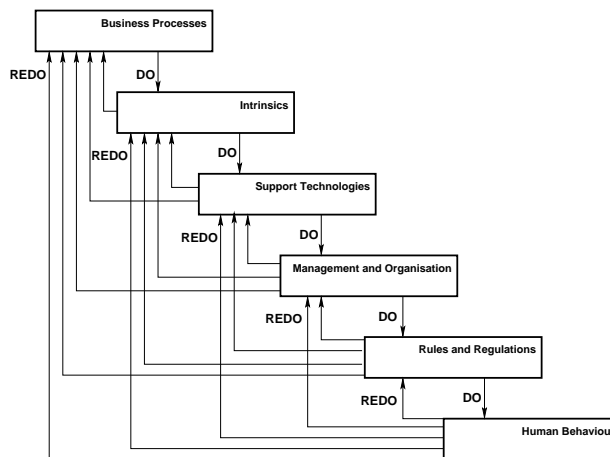


Fig. E.3. The domain modelling stage

Describing these facets usually involves trial-and-error descriptions, that is, iterations between steps. These iterations must be managed. Figure E.3 illustrates the possibilities of “endless thrashing” within just the domain modelling stage.

Requirements Engineering

We consider requirements to be analysable into three categories: domain, interface and machine requirements. Domain requirements are those requirements which can be expressed solely using terms from (or allowed in) the domain description. Machine requirements are those requirements which can be expressed without using terms from the domain description; instead terms are used from the machine (the hardware and the software to be designed). Interface requirements are those requirements which can be expressed using terms both from (or allowed in) the domain description and from (or allowed in) the machine specification. Where domain descriptions express *what there is in the domain*, the requirements prescriptions express *what there shall or must be in the machine*.

Once a domain description is considered complete work on requirements can commence. Figure E.4 on the next page records the six stages of requirements development. The major stage, requirements modelling, is detailed in Fig. E.5 on page 261. That figure shows domain requirements engineering actions in the top-left quadrant, interface requirements engineering actions in the lower-left quadrant, and machine requirements engineering actions in the right half of the diagram.

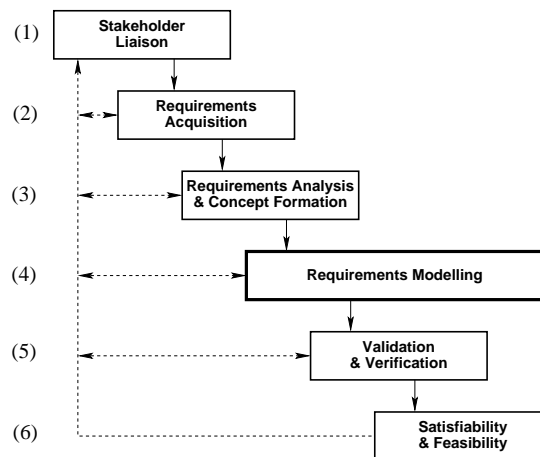


Fig. E.4. The requirements engineering phase. The modelling stage is detailed in Fig. E.5

One aspect of domain requirements modelling stage (box 4 of Fig. E.4) can be summarised as follows: the requirements engineer works with the requirements stakeholders and as follows: First the domain requirements are constructed illustratively by asking the stakeholders to identify (b) which parts of the domain description should be “carried over” into, i.e., projected onto the requirements prescription while (c) possibly instantiating these (now) prescriptions into special cases, and/or (d) making the prescriptions more deterministic, and/or (e) extending the domain description with descriptions of entities, functions, events and behaviours that were not feasible in the domain but are feasible with (the advent of) computing, and (f) finally fitting the emerging domain requirements prescriptions to those of related, other software development projects, if any. Step (a) deals with those requirements, business process re-engineering (BPR), which are not “implemented” as computing, but are relied upon in the correct functioning of the emerging software, that is, which reflect assumptions that must be made about the environment (humans and equipment, including other computing systems). The BPR must also be implemented and adhered to, but by the management and the staff of the user of the required software.

Figure E.5 on the facing page also shows details of the interface and machine requirements steps — for which we refer the reader to [89, Sects. 19.5–6].

Figure E.6 on the next page intends to indicate that a number of the machine requirements modelling steps can take place independent of one another, i.e., in parallel.

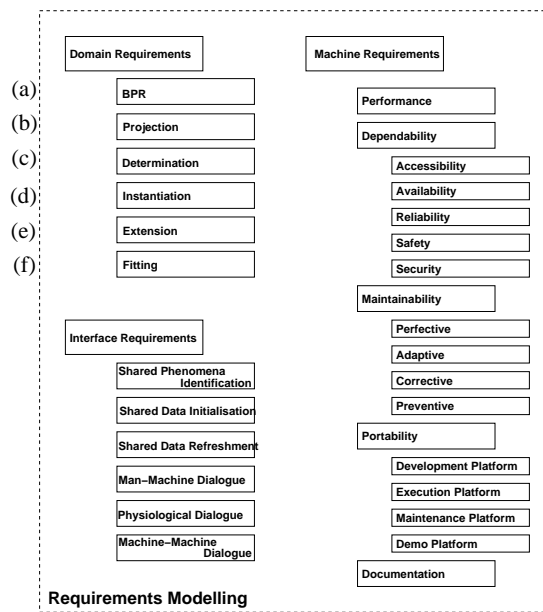


Fig. E.5. The requirements modelling stages

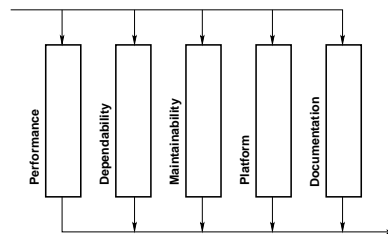


Fig. E.6. The machine requirements modelling stage: five concurrent steps

Software Design

Once a complete requirements prescription has been achieved one can start the software design. Figure E.7 on the following page shows a generic, summary process description for all three phases of software development while emphasising, in its lower three-quarter, the stages and steps of software design: from a possibly stepwise software architecture design via a stage of stepwise refinement of the software components identified by the architecture design, to the final coding step.

The i “stacks” of c_{ij} component boxes shall indicate that the software components may be stepwise refined ending up with executable code k_i . The component part of Fig. E.7 on the next page is rather idealised. One usually experiences that

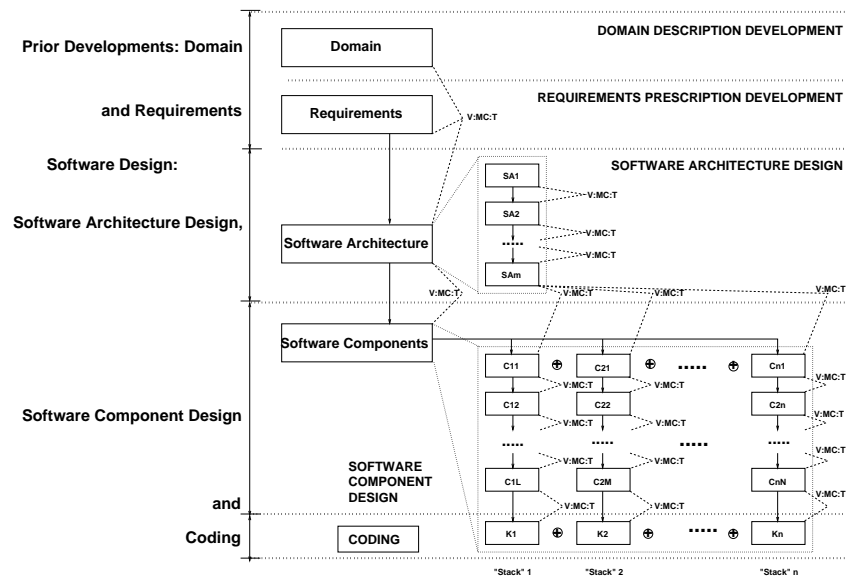


Fig. E.7. The software design phase

components can be shared across the software design, thus the component part of Fig. E.7 should be shown more realistically as a lattice of refinement steps.

E.3.3 Documents

Work within each step, stage and phase results in documents. Some are necessarily informal, others can both be formulated informally and formally. To each phase we can therefore attach a number of documents. Appendix E.9¹³ give an overview of these phase documents. Bearing in mind the span and wealth of software related documents we can almost say: *“All we do, in software development, is writing documents — and a few can serve as the basis for computations by machines.”*

E.3.4 Formal Techniques

It is now commonly accepted that professional software engineering entails the use of a set of one or more formal techniques. Examples of formal techniques, often referred to as formal methods, are Alloy [202], ASM (Abstract State Machines) [261], B and event-B [1, 130], DC (Duration Calculus) [301], MSC and LSC (Message and Live Sequence Charts) [184, 199–201], Petri Nets [210, 250, 258–260], Statecharts [180–183, 185], RAISE (Rigorous Approach to Industrial Software Engineering) [87–89, 166–168], TLA+ (Temporal Logic of Actions) [217, 218, 232],

¹³Appendix Sects. E.9.1–E.9.3

VDM (Vienna Development Method) [111, 112, 158] and Z [187, 283, 284, 297]. The EATCS¹⁴ Monograph [109] arose from [130, 148, 167, 187, 232, 242, 261] and covers ASM, B and event-B, CafeOBJ, CASL, DC, RAISE, TLA+, VDM and Z.

E.3.5 Software Development Graphs

The process description diagrams of Sect. E.3.2 were generic and need be instantiated to specific classes of software. Instantiated process description diagrams will be called software development graphs [41, 42, 45, 117]. In Fig. E.8 we show one such instantiation, the graph for developing a compiler for a programming language such as CHILL [175, 176], Ada [120, 141], Java, etc.

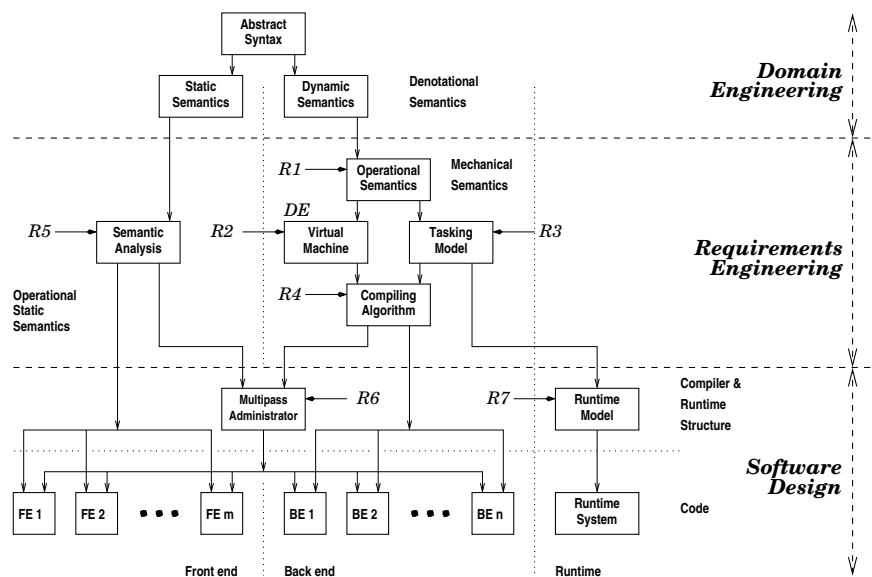


Fig. E.8. A compiler software development graph

The domain description amounts, for the case of compiler development, to a description of the syntax and semantics of the programming language for which a compiler is to be developed. The requirements, on one hand, in a series of steps concretise the semantics while injecting additional, typically interface and machine requirements — such as requiring, for example, the compiler to be a debugging compiler, and/or to require that the compiler itself can execute in a limited addressing space, and/or to require that the compiler generate code for indefinitely large programs, and/or to require that the generated code fit in a limited addressing space memory, and/or to require that the compiler generate

¹⁴EATCS: European Association for Theoretical Computer Science

code for interactive debugging, or to require that the compiler generate “near” optimal code, etc. These interface and machine requirements determine choices of concretisation. Finally a multi-pass administrator for the compiler can be designed and the passes coded from the refined, concretised requirements.

E.3.6 The Process Model Graph

The process diagrams of Sect. E.3.1 represents one, the generic view of the phased, staged and stepwise development of software. The software development graph of Fig. E.8 on the preceding page represents the specific view of a software development. The two views must be reconciled into what we may call process model graphs.

Process Model Graph Construction

We do not illustrate such a process model graph here — but think of a way to “take the cross-products” of the two kinds of diagrams and you may get “a picture” of what is involved: the three simple boxes of the domain engineering part of Fig. E.8 on the previous page correspond to the single domain modelling box in the left part “column” of boxes of Fig. E.2 on page 258. “Add” now the remaining boxes of Fig. E.2 on page 258, Fig. E.4 on page 260, Fig. E.5 on page 261, and Fig. E.7 on page 262, etcetera, and one obtains the compiler domain engineering process graph.

One thing is the process model, viz., the graph-like structures shown in, for example, Fig. E.2 on page 258, Figs. E.4 on page 260, E.5 on page 261 and Fig. E.7 on page 262. (These are syntactic structures, but have semantic meanings.) Another thing is the actual usage of such models, that is, the actual processes that the software developers (domain, requirements and software design engineers) “steer through” when developing domain models, requirements models and software designs.

Graphs and Graph Traversals (Traces)

Assume some graph-like, let us call it, process model, see Fig. E.9 on the next page.

So Fig. E.9 on the facing page shows a process model and two traversal traces (to be defined shortly). REDOs, that is, iterations of phases, stages and steps lead to additional traces. Let us call the totality (set) of these traces for OK traces. And “jumping” or just “skipping” phases, stages and steps lead to further additional traces. Let us call these “jumped” or “skipped” traces for NOK traces. A process model thus denotes a possibly infinite set of such OK and NOK traces.

The leftmost part of Fig. E.9 on the next page shows an acyclic graph. The graph consists of distinctly labeled nodes and (therefrom distinctly) labeled edges. The center and right side of the figure shows some possible traversal traces. By a traversal trace we understand a sequence of wave-fronts.

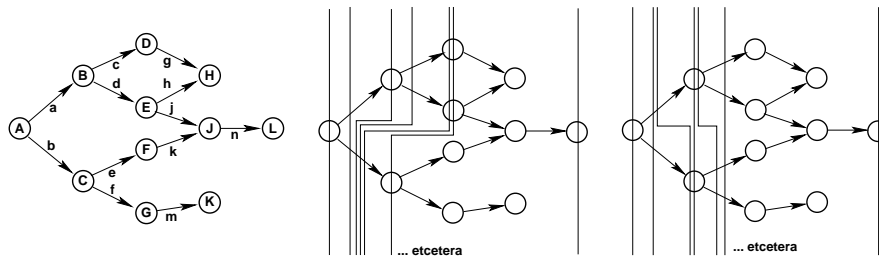


Fig. E.9. A graph (left) and two (incomplete) traversal traces (center and right)

By a wavefront we understand a set of node and edge labels such that no two of these are on the same path from an input (i.e., in-degree zero) to an output (i.e., out-degree zero) node, and such that there is a contribution to the set from any path from an input to an output node.¹⁵

The third wave of the two traces shown in the two rightmost figures can thus be represented by $\{B, b\}$ and $\{a, C\}$.

Process Models and Processes

A process model graph thus has boxes which denote activities that result in information and description, prescription or specification documents and edges which denote precedence relations on activities.

A development process is any trace over sets of these activities.

The center graph of Fig. E.9 thus portrays the following initial trace:

$$\langle \{A\}, \{a, b\}, \{B, b\}, \{c, d, b\}, \{D, E, b\}, \{D, E, C\}, \dots, \{L\} \rangle$$

Thus a process model denotes a set of such traces.

Incomplete and Extraneous Processes

The trace:

$$\langle \{A\}, \{a, b\}, \{c, d, b\}, \{D, E, b\}, \{D, E, C\}, \dots \rangle$$

appears to have skipped the activity (phase, stage or step) designated by B . Loosely speaking we call such processes incomplete with respect to their underlying (i.e., assumed) process model (Fig. E.9, the leftmost graph).

The trace:

¹⁵In-degree zero nodes are nodes of a graph upon which no edges are incident. Out-degree zero nodes are nodes of a graph from which no edges emanate. A path is a sequence of node and edge labels such that any edge of a path infix the two neighbouring nodes in the graph.

$$\langle \{A\}, \{a,z\}, \{X\}, \{D,Y,b\}, \{D,E,C\}, \dots \rangle$$

appears to have performed some activities (z, X, Y) not designated by the process model of Fig. E.9 on the preceding page (the leftmost graph). Loosely speaking we call such processes extraneous (or ad hoc) with respect to their underlying process model.

Process Iterations

The trace

$$\langle \{A\}, \{a,b\}, \{B,b\}, \{a,b\}, \{B,b\}, \{c,d,b\}, \{B,b\}, \{c,d,b\}, \{D,E,b\}, \{D,E,C\}, \dots \rangle$$

designates an iterated process. After action B in the first $\{B,b\}$ of the trace the process “goes back” to perform action b (in $\{a,b\}$); and after (either of) actions c or d in $\{c,d,b\}$ the process “goes back” to perform action B in $\{B,b\}$. Loosely speaking we call such processes iterated with respect to their underlying process model.

The above trace only shows simple “one step” (or stage or phase) “backward and then onward” iterations. But the REDO idea, also indicated in Fig. E.1 on page 257, can be extended to any number of steps (etc.).

• • •

Section E.5.1 will follow up on the above and with respect to the management aspects.

E.3.7 Classical Process Models

Standard textbooks on software engineering [19, 251, 253, 280] record a variety of process models. Notably Winston’s Waterfall process model [296] and Boehm’s Spiral model [11]. Many other process models and variations thereof are mentioned in the literature: Agile Software Development¹⁶, Design by Contract [235], Extreme Programming [21, 22, 229], Model-driven Development [193], Design Patterns [163], Prototyping [128], Software Evolution [143], Test Driven [20], V-model process models [156], etcetera. Common to all of them are that they allow for a TripTych interpretation, that is, an “embedding” within the TripTych process model.

E.4 CMM: The Capability Maturity Model

We choose here to “anchor” our discourse of software management by referring to Humphrey’s *Capability Maturity Model* (CMM) [194]. CMM postulates five

¹⁶See the Agile Software Development Manifesto: <http://agilemanifesto.org/>

levels of maturity of development groups. Level 1 being a lowest, in a sense “least desirable”, and level 5 being the highest, “most desirable” level of professionalism that Humphrey finds it useful to define. Process improvement, by a development group, is now the improvement of the development processes such that the group (i.e., the software house) advances from level i to level $i+j$ where i, j are positive numbers and $i+j$ is less than 6. So let us first, in this section, review Humphrey’s notion of CMM, before we, in the next section (Sect. E.5) detail these and other management aspects.

The following characterisations are “lifted” from http://en.wikipedia.org/wiki/Capability_Maturity_Model.

E.4.1 Level 1, Initial

“At maturity level 1, processes are usually ad hoc and the organization (i.e., the software house) usually does not provide a stable environment. Success in these organizations depends on the competence and heroics of the people in the organization and not on the use of proven processes. In spite of this ad hoc, chaotic environment, maturity level 1 organizations often produce products and services that work; however, they frequently exceed the budget and schedule of their projects.”

Maturity level 1 organizations are characterized by a tendency to over commit, abandon processes in the time of crisis, and not be able to repeat their past successes again.”

As we shall later argue, following the very basics of the methodological approaches, phases, stages and steps of the TripTych approach, as outlined in the Sect. E.3, immediately brings us at a CMM maturity level “far from” level 1.

E.4.2 Level 2, Repeatable

“At maturity level 2, software development successes are repeatable. The organization may use some basic project management to track cost and schedule.

Process discipline helps ensure that existing practices are retained during times of stress. When these practices are in place, projects are performed and managed according to their documented plans.

Project status and the delivery of services are visible to management at defined points (for example, at major milestones and at the completion of major tasks).

Basic project management processes are established to track cost, schedule, and functionality. The minimum process discipline is in place to repeat earlier successes on projects with similar applications and scope. There is still a significant risk of exceeding cost and time estimate.”

As we shall later argue, following the very basics of the methodological approaches, phases, stages and steps of the TripTych approach, as outlined in the Sect. E.3, immediately brings us at a CMM maturity level higher than level 2.

E.4.3 Level 3, Defined

“The organization’s set of standard processes, which is the basis for level 3, is established and improved over time. These standard processes are used to establish consistency across the organization. Projects establish their defined processes by the organization’s set of standard processes according to tailoring guidelines.

The organization’s management establishes process objectives based on the organization’s set of standard processes and ensures that these objectives are appropriately addressed.

A critical distinction between level 2 and level 3 is the scope of standards, process descriptions, and procedures. At level 2, the standards, process descriptions, and procedures may be quite different in each specific instance of the process (for example, on a particular project). At level 3, the standards, process descriptions, and procedures for a project are tailored from the organization’s set of standard processes to suit a particular project or organizational unit.”

The standard processes referred to above can be interpreted to be the phases, tasks and steps covered in Sect. E.3.2 and shown by Figs. E.2 on page 258, E.4 on page 260, E.5 on page 261 and E.7 on page 262.

As we shall later argue, following the very basics of the methodological approaches, phases, stages and steps of the TripTych approach, as outlined in the Sect. E.3, immediately brings us at a CMM maturity level higher than level 3.

E.4.4 Level 4, Managed

“Using precise measurements, management can effectively control the software development effort. In particular, management can identify ways to adjust and adapt the process to particular projects without measurable losses of quality or deviations from specifications.

Subprocesses are selected that significantly contribute to overall process performance. These selected subprocesses are controlled using statistical and other quantitative techniques.

A critical distinction between maturity level 3 and maturity level 4 is the predictability of process performance. At maturity level 4, the performance of processes is controlled using statistical and other quantitative techniques, and is quantitatively predictable. At maturity level 3, processes are only qualitatively predictable.”

The ‘precise measurements’ alluded to above are, in the TripTych approach afforded by the extensive documentation mentioned earlier and by the use for formal techniques (formal specification, formal verification and formal testing). The precise measurements include such things as (i) how “closely” an ideal project development graph is being adhered to, (ii) the number and “size” of iterations caused by the need to ‘redo’ certain steps and even stages,

(iii) the ratio of successfully discharged proof obligations to the number of proof obligations raised by the formal specifications; etc.

We shall later argue that following the very basics of the methodological approaches, phases, stages and steps of the TripTych approach, as outlined in the Sect. E.3, immediately brings us at CMM maturity level 4.

E.4.5 Level 5, Optimising

“Maturity level 5 focuses on continually improving process performance through both incremental and innovative technological improvements. Quantitative process-improvement objectives for the organization are established, continually revised to reflect changing business objectives, and used as criteria in managing process improvement. The effects of deployed process improvements are measured and evaluated against the quantitative process-improvement objectives. Both the defined processes and the organization’s set of standard processes are targets of measurable improvement activities.

Process improvements to address common causes of process variation and measurably improve the organization’s processes are identified, evaluated, and deployed.

Optimizing processes that are nimble, adaptable and innovative depends on the participation of an empowered workforce aligned with the business values and objectives of the organization. The organization’s ability to rapidly respond to changes and opportunities is enhanced by finding ways to accelerate and share learning.”

We shall later argue that following the very basics of the methodological approaches, phases, stages and steps of the TripTych approach, as outlined in the Sect. E.3, immediately brings us at CMM maturity level 4 — and, that deploying these methodological approaches with deeper and deeper commitment to formal techniques will eventually bring a software house to maturity level 5.

E.5 Software Management

By software management we mean the management aspects of deciding (i) which software to develop, (ii) how to develop it and (iii) how to market, sell and service that software. We group Items (i) and (iii) as part of software product management and Item (ii) as software project management. Item (ii) potentially includes the monitoring and control of all aspects of the software development process.

E.5.1 Software Project Management

By software project management we mean the management aspects of deciding how to develop software.

You may interpret Sect. E.3 as primarily focused on software development (project) processes. (In Sect. E.5.2 we shall, in contrast, interpret Sect. E.3 as also significantly implying software product possibilities.) We shall now summarise this first interpretation.

Summary of Software Development Project Processes

A proper software development project management implies three phases: domain description development, requirements prescription development and software design development. Ideally, these phases are sequentially ordered, as listed above. Each phase then consists of several, also ideally sequenced stages, and within stages there are then any number of one or more steps. Stages and generic steps particular to the specific phases were mentioned extensively in Sect. E.3. The phases, stages and steps were hinted at in Figs. E.1–E.7. The resulting documents are listed in Appendix E.9.

On these phases, stages and steps, as diagrammed by Figs. E.1–E.7, we can now superimpose a software development graph, as shown in Fig. E.8 on page 263. The result, not shown in general, is a graph, such as the leftmost graph of Fig. E.9 on page 265.

A specific software development process can now be “pictured” as a trace (over the augmented graph), such as, for example, the center graph of Fig. E.9 on page 265.

• • •

Software development project management has a spectrum of tasks to perform. At one end, say the creative end, there are the planning, allocation and scheduling tasks, that is, the tasks of deciding which software development processes should be followed and their resourcing. At the other end there are the monitoring and control of the software development processes.

The Creative Aspects of Software Project Management

The creative management tasks are enumerated next.

1. **TripTych Model Adaptation.** If a suitable description already exists for the domain of the planned application area, then that domain model may be adopted. Else a suitable domain description development must be included in the project. If a suitable prescription already exists for the requirements of the planned software, then that requirements model may be adopted. Else a suitable requirements prescription development must be included in the project. (We tacitly assume that a requirements prescription will not be accepted unless a suitably related domain description is already at hand.) If a suitable architecture design specification already exists for the planned application area, then that architecture design specification may be adopted. (We tacitly assume that a architecture design

specification will not be accepted unless a suitably related requirements prescription is already at hand.)

Thus there are a number of intertwined management decisions to be made with respect to the above suitabilities. These decisions cannot be made before some initial, “rough-guess” resource allocation (costing) and task scheduling (timing) has been made.

2. **Creating the Software Development Graph.** For a well-known area of software development, as for example compiler development, one may adopt an existing software development graph. But for a completely new, i.e., un-experienced area of software development, one never tried before (at least “by this software house”), there may (thus) not be an existing software development graph — and it only transpires as development (according to the TripTych model) proceeds. In the former case (of a well-known software development graph) we may speak of a proper development project whereas in the latter case (of no known software development graph) we may speak of a proper research project.
3. **Process Model (I): Superposition of the Software Development Graph.** Thus superposition is either an a-priori, strict affair, or is an ongoing “add on”.
4. **Resource Estimation (I).** The resulting process model is then the basis for a rough resource estimation: cost and time for each box, i.e., step, stage and phase.
5. **Deciding TripTych (Process) Model Degrees of Adherence.** Such costing and timing considerations (as mentioned in the previous paragraph) may also determine the degree to which a given project is to follow all, or only some of the “idealised” tasks implied by the full TripTych model. We give some examples (A,B,C).
 - (A) The degree to which, for example, domain or requirements acquisition involves a broadest or just a narrow spectrum of stakeholders, and the degree (depth) to which solicitation is pursued are two aspects to be decided upon. Similarly with respect to “exhaustiveness” or “superficiality” of domain or requirements analysis.
 - (B) The degree to which, for example, domain or requirements validation is done is yet another aspect to be decided upon.
 - (C) With respect to the use of formal techniques (formal specifications and formal verifications, etc.) one may speak of
 - (i) ‘systematic formal development’ (or ‘formal development “lite” ’),
 - (ii) ‘rigorous formal development’ and
 - (iii) ‘full, formal development’.
 In (i) one usually omits stating, let alone discharging proof obligations. With respect to (ii) one usually states some (interesting, “crucial”) proof obligations while discharging only crucial proofs. With respect to (iii) one usually states and discharges all proof obligations.
6. **Process Model (II): Allocation of Resources to Steps, Stages and Phases.** Depending on the evolving status of the project, whether

a proper development project or a proper research project, or somewhere in-between, one may now either be able to rather precisely estimate and allocate (abstract) resource consumptions: people, time, equipment, and hence monies for part of or the entire software development, or only be able to very roughly estimate these “parameters”, and then usually only for the current phase, or stages, or steps.

7. **Process Model (III): Scheduling Development of Step, Stage and Phase Tasks.** Similarly one may be able to precisely or only loosely determine a time schedule.
8. **Process Model (IV): Review and Re-planning of Process Model (III).** The above planning items may be performed either in sequence and then re-iterated, or in some form of interleaving. In any case, a state will be reached in which the planners, i.e., management, wishes to evaluate the project being planned. Is it within acceptable resources (people, time, monies) or not? Must project short-cuts be made? Et cetera. Decisions here lead to modifications of the process model so far diagrammed.
9. **Resource Estimation (II).** Rough estimates can then be refined.
10. **Process Model (V): Finalisation of Process Model.** Once management agreement has been reached the process model can be finalised: that is, the process model can be annotated with concrete, committed people, committed dates and committed monies.

The project proper can start.

The Monitoring and Control Aspects of Software Project Management

The software development project alternates between operations management monitoring the project as it proceeds. The monitoring is with respect to the project’s process model. In this section we shall outline a number of the more important aspects of this monitoring. We shall formulate monitoring aspects in terms of process assessments. Once a sub-process has been assessed (i.e., evaluated) to be in want of improvement, then either the relevant operations manager, usually called a sub-project leader, institutes the necessary process improvement, or defers to (seeks guidance from or hands over project control responsibility to) “upper” management.

We can define two notions of process model compliance, a syntactic and a semantic. The syntactic notion of process model compliance has to do with “the degree” to which an actual process matches a possibly iterated, i.e., an OK trace of a process model. The semantic notion of process model compliance is concerned with adherence to the semantics of boxes.

Suffice it to summarise that an ongoing process, i.e., an ongoing software development project can be assessed wrt. its syntactic and its semantics compliance wrt. its process model. One can precisely state which activities have been omitted (incompleteness), and which activities were extraneous (or ad hoc).

Syntactic and Semantic Process Compliance.

We first deal with syntactic compliance, then with semantics compliance. This monitoring and control aspect of project management is, by far, the most resource-consuming aspect. It requires constant, i.e., daily vigilance.

SOFTWARE PROCESS ASSESSMENT 1 Syntactic Process Compliance: *Given the generic process models diagrammed in Figs. E.1–E.7 and given the project-specific software development graph as exemplified by Fig. E.8, one can now, in a process claimed to adhere to these models and graphs, quite simply assess whether that actual process follows those diagrams.*

We assume that assessment takes place “regularly”, that is, with a frequency higher than process wave transitions, that is, more often than the process evolves through steps and stages. Otherwise it may be too late (or too cumbersome) to “catch and do” an omitted step.

SOFTWARE PROCESS IMPROVEMENT 1 Syntactic Process Compliance: *Adherence to the process model can, at least “formally”, be improved by actually ensuring that the process steps and stages (or even phases) that were assessed to not having been performed, that these be performed.*

But syntax is only that, namely superficial. The semantics of what is being developed, i.e., of the resulting — typically description, prescription, specification and analysis — documents is what is most important. The next assessment is more one of management than of the documents.

SOFTWARE PROCESS ASSESSMENT 2 Process Model Syntax and Semantics: *In order to handle process improvement (à la CMM, from a lower to a higher level) — using the TripTych approach — managers (as well as, of course, developers), must be intimately familiar with the syntax and semantics of the documents produced and expected to be produced by process model node activities.*

This is a strong requirement and can not be expected by just any software development organisation. And there are really no shortcuts.¹⁷ Process improvement — wrt. the precision of monitoring resource usage — is predicated on this assumption: that management is strongly based on professional awareness of process model¹⁸ principles, techniques and tools. The “degree”¹⁹ to

¹⁷In other branches of engineering project managers (i.e., project leaders) and developers, the “engineers at floor level” basically all have the same, normalising education. Hence they are intimately familiar with the syntax and semantics of their tasks. The problem is in software engineering.

¹⁸We here use the broader term ‘process model’ than the term TripTych. In this way the assessment and improvement guidance statements apply more generally — provided, of course, such “another process model” has a clear semantics.

¹⁹The “degree” notion is not defined here.

which a development document adheres to the syntax and semantics of the relevant box thus provides an assessment.

Several industries, worldwide, the most well known is perhaps Praxis High Integrity Systems, <http://www.praxis-his.com>, practices this on a regular basis. So do other members of ForTIA: The Formal Techniques Industrial Association, www.fortia.org.

SOFTWARE PROCESS IMPROVEMENT 2 Process Model Syntax and Semantics: *To improve this general aspect of the possible processes that developers and managers might be able to pursue under the banner of the TripTych (or other) Process Model one simply has to resort to education and training. Or hire better educated and trained staff, including managers. There is no substitute.*

We remind the reader of our earlier remarks on the “standard, conventional development” versus the possible “research” nature of a software “development” project (cf. Items 2 on page 271 and 6 on page 271).

A “Base 0” for TripTych Developments.

By a TripTych development we mean a development which applies the principles, techniques and tools as prescribed by the TripTych dogma. Either in a systematic, or in a rigorous, or in a formal way. A TripTych development process therefore, “by definition” has its base point at level 4 in the CMM scale. This does not mean that a software development process which claims to follow the TripTych dogma (or the software house within which that process occurs) at least measures at level 4. The dogma sets standards. The process may follow, or may not follow such standards. Whether they are followed or not is now an “easy” matter to resolve. The degree to which the dogma, in all its very many instantiations, is followed is now “fairly easy” to resolve. The “ease” (or “easiness”) depends on how well developers and management understand the many TripTych (or, more generally, the specifically chosen process model’s) principles, techniques and tools, how well they understand the prescribed syntax and semantics of required documents, and on how well they understand their pragmatics, that is, the reason for these principles, techniques and tools.

Pragmatics.

The pragmatics is what makes management interesting. Well mastered pragmatics allows the managers leeway (i.e., discretion) in the dispatch of their duties, that is, allow them to skip (or “go light” on) certain activities, including choosing whether a step or even a stage should be performed “lightly” or more-or-less “severely”, that is, be informal, or formal (and then in a scale from systematic via rigorous to formal).

SOFTWARE PROCESS ASSESSMENT 3 Planned Syntactic and Semantics Compliance: *If a process is assessed (SPA) to be in full compliance, syntactically and semantically with the process model then we claim that the software development in this case is at CMM level 4 (or higher).*

SOFTWARE PROCESS IMPROVEMENT 3 Planned Syntactic and Semantics Compliance: *If it is assessed that a process has not reached CMM level 4, and that at least CMM level 4 is desired, then one must first secure syntactic compliance, see process improvement # 2 (Page 273), thereafter ensure that each of the steps (or stages, or phases) whose semantic compliance was assessed too low be redone and according to their semantic intents.*

Resource Planning.

This aspect relates more to the ‘creative’ aspect of the project than the monitoring and control aspect.

SOFTWARE PROCESS ASSESSMENT 4 Resource Planning: *How can one assess a software development project plan (i.e., graph), that is, something which designates something yet to happen? Well, one can compare with previous software development graphs purporting to cover “similar” (if not identical) development problems and their eventual outcome, that is, the process that resulted from following those graphs. Based on actual resource usage accounts one can now — “to the best of anyone’s ability” — draw a software development graph and ascribe resource consumption estimates (time, people, equipment) to each and every node. Thus ‘assessment’ here was “speculated assessment” of an upcoming project.*

Thus, if that ‘speculated assessment’ of an upcoming project is felt, by the assessors, i.e., the management, to be flawed, to be questionable, then one has to proceed to improvement:

SOFTWARE PROCESS IMPROVEMENT 4 Resource Planning: *One must first improve the precision with which one designs the domain-specific project development graphs. Then the precision with which we associate resource usage with each box of such a graph. Etcetera. Some development projects are very much “repeats” of earlier such projects and one can expect improvement in project development graphs for each “repeat”. Other projects are very much tentative, explorative, that is, are actually applied research projects — for which one only knows of a project development graph at the end of the project, and then that graph is not necessarily a “best such”!*

Monitoring & Controlling Resource Usage.

As the project (i.e., the process) evolves management can now check a number of things: adherence to schedule and allocation, and adherence to the syntactic and the semantic notions of process model compliance.

Most process models do not possess other than rather superficial and then mostly syntactic notions of compliance. In the TripTych process model semantic compliance is at the very core: Every box of the process models have precise syntax and semantics of the documents that are the expected results of these (box) activities.

SOFTWARE PROCESS ASSESSMENT 5 Resource Usage: *No problems here. As each step (of the development process) unfolds one can assess its compliance to estimated plan.*

Should a resource usage assessment reveal that there are problems (for example: all resources used well before completion of step) then something must be done:

SOFTWARE PROCESS IMPROVEMENT 5 Resource Usage: *Well, perhaps not this time around, when all planned resources have already been consumed — no improvement can undo that — but perhaps “next” time around. An audit may reveal what the cause of the over-consumption was. Either a naïve, too low resource estimate, or unqualified staff, or some simple or not so simple mistakes? Improvement now means: make precautions to avoid a repetition.*

Resource usage is at a very detailed and accountable level and can thus be better assessed. Slips (usually excess usage) can be better foreseen and discovered and more clearly defined remedies, should milestones be missed or usage exceeded, can then be prescribed — including skipping stages and steps whose omission are deemed acceptable.

Skipping stages and steps result in complete, perhaps extraneous (ad hoc) processes. Given that management has an “ideal” process model and hence an understanding of desirable, possibly iterated processes, management can now better assess which are acceptable slips.

From Informal to Formal Development.

By process improvement, to repeat and to enlarge on our previous characterisation of what is meant by process improvement, we understand something which improves the quality of resulting software. We “translate” the term ‘resulting software’ into the term ‘resulting documents’. These documents can — as listed in Appendix E.9 — be developed either informally (without any use of any formalism other than the final programming language²⁰), or systematically formal, or rigorously formal or formally formal!

²⁰Thus we do not consider UML to be a formalism. For a “formalism” to qualify as being properly formal it must have a precise syntax, the syntax must have a

Informal Development,

It is an indispensable property of the TripTych approach to software development that the formalisable steps domain engineering, requirements engineering and software design be pursued in some systematic via rigorous to formal manner. Hence the informal aspects of development is restricted to the development of only the informative documents. Informative documents are usually “developed” by project leaders and managers. Hence an “upper” level of management is doing the process assessment and possibly prescribing process improvements to a “lower” level of management!

SOFTWARE PROCESS ASSESSMENT 6 Informal Development of Informative Documents: *We refer to Appendix Sects. E.9.1–E.9.3. Items 1. (Information) of those sections lists the kind of informative documents to be carefully developed — and hence assessed. Since no prescribed syntax, let alone formal semantics can be given for these documents — whose purpose is mainly pragmatic — assessment is a matter of style. It is easy to write non-sensible, “pat” informative documents which do not convey any essence, any insight. Assessment hence has to evaluate: does a particular, of the many informative documents listed in Items 1. of Appendix Sects. E.9.1–E.9.3 (Information), really convey, in succinct form, an essence of the project being initiated?*

SOFTWARE PROCESS IMPROVEMENT 6 Informal Development of Informative Documents: *If an informative document is assessed to not convey its intended message succinctly, with necessary pedagogical and didactic “bravour”, then it must be improved. “Seasoned”, i.e., experienced managers may be able to do this.*

Systematic, Rigorous and Formal Development.

The development of domain description, requirements prescription and software design documents as well as the development of analytic documents (including tests, verification, model checking and validation) can be done in a spectrum from systematically via rigorously to formally.

SOFTWARE PROCESS ASSESSMENT 7 Staff and Tool Qualification: *Given the syntax and semantics of the specific step — in the process model — of the tasks to be assessed a syntax- and semantics-knowledgeable person cum a project (task or step) leader or a manager, can assess compliance. That assessment is greatly assisted by the software tools²¹ that support activities of those tasks: If they can process the*

precise semantics, and there must be a congruent proof system, that is, a set of proof rules such that the semantics satisfy the proof rules.

²¹These software tools mainly support the use of the main tools, namely the specification languages, their transformation (or refinement) and their proof systems. [See paragraph **Tools** on the next page.]

documents then something seems OK. If not, assessment will have to be negative.

There are now two distinct, “extreme” reasons for a failure to meet assessment criteria — with any actual reason possibly being a combination of these two “extremes”. One is that the quality of the staff performing the affected tasks is not up to expectations. The other is that the tools being deployed are not capable of supporting the problem solution task.

Staff Qualification.

If the assessment of ‘Systematic, Rigorous and Formal Development of Specifications and Their Analysis’ is judged negative due to inadequate development decisions then we suggest the following kind of improvement.

SOFTWARE PROCESS IMPROVEMENT 7 Staff Qualification: *It is suggested that improvement, when deemed necessary, takes either of three forms: Either “move” from a systematic to a rigorous level of development, or from a rigorous to a formal level of development when that is possible and redo the task(s) affected. Or educate and train staff to re-perform the affected task(s) more accurately (while remaining systematic, rigorous, or formal as the case may be). Or replace affected staff with better educated and trained staff and redo the task(s) affected.*

These kinds of improvement decisions are serious ones.

Tools.

There are different categories of tools.

Tools can serve management: for the design of software development graphs (a la Fig. E.8 on page 263) and their “fusion” into the appropriate process model diagrams (a la Fig. E.2 on page 258, Figs. E.4 on page 260 and E.5 on page 261, and Fig. E.7 on page 262) and for the monitoring and control (i.e., assessment and improvement) of the process with respect to these diagrams.

And tools can serve developers: syntactic and semantic description, prescription and software design tools as well as analytic tools: for testing, model checking and verification (proof assistance or theorem provers). These tools embody, that is, represent the formalisms of the textual or diagrammatic notations used — whether Alloy [202]²², B [1, 131], Duration Calculus [301, 302], LSCs [145, 184, 212], MSCs [199–201], Petri Nets [210, 250, 258–260], RAISE RSL

²²The present author is duly fascinated by the elegance and “power” of Alloy, but, by including Alloy in this list does not mean that we know whether it scales up to industrial use. Alloy is certainly strongly recommended, as are the others in the list, for teaching purposes. We do think, however, that the remaining languages do scale up — but that the tools could be improved for all of these to be truly industry oriented.

[87–89, 97, 165, 166, 168], Statecharts [180–183, 185], TLA+ [217, 218, 232, 233], VDM-SL [111, 112, 157, 158], or Z [186, 187, 283, 284, 297]. Thus the formal notations of the above listed eleven languages, whether textual or diagrammatic, or combinations thereof, are tools, as are the software packages that support uses of these linguistic and analytic means.

Tool Qualification.

If assessment of ‘Systematic, Rigorous and Formal Development of Specifications and Their Analysis’ is judged negative due to inadequate tools then we suggest the following kind of improvement:

SOFTWARE PROCESS IMPROVEMENT 8 **Tool Qualification:** *Better tools must be selected and applied to the task(s) affected (i.e., judged negatively assessed). These tools are either intellectual, that is, the specification languages, whether textual or diagrammatic, and their refinement and proof systems, or they are the manifest software tools that support the intellectual tools.*

These are likewise serious improvement decisions.

Review of Process Assessment and Process Improvement Issues

We have surveyed, somewhat cursorily, a number of software process assessment and software process improvement issues. We characterise these from a another viewpoint below.

1. **Process Model Syntax and Semantics Assessment and Improvement:** We refer to Page 273, Assessment and Improvement Item 2. The issue here is whether the management and development staff really understands and, to a satisfactory degree, can handle the TripTych process model in all its myriad of phases, stages and steps, specificationally and analytically, and with all its myriad of documentation demands. If not, then they cannot be effectively assessed and subjected to “standard” improvement measures. This is an assessment (and improvement) issue which precedes proper project start.
2. **Syntactic Process Compliance Assessment and Improvement:** We refer to Page 273, Assessment and Improvement Item 1. This issue is a “going concern”, that is, an ongoing, effort of regular assessment and possibly an occasional improvement. It merely concerns whether a mandated step (or stage or even phase) of development and its expected production of related documents has taken or is taking place.
3. **Planned Syntactic and Semantics Compliance Assessment and Improvement:** We refer to Page 275, Assessment and Improvement Item 3. This is an assessment (and improvement) issue which, in a sense, sets a proper framework for the project: Does management wish to attain

- at least CMM level 4, or higher or lower? In that sense it precedes project start while determining the rigour with which the next assessments and improvements are to be pursued.
4. **Resource Planning Assessment and Improvement:** We refer to Page 275, Assessment and Improvement Item 4. This item of assessment and improvement takes place at project start and may have to be repeated when resource consumption exceeds plans. Assessment and improvement may involve “layers” of project leaders and management.
 5. **Resource Usage Assessment and Improvement:** We refer to Page 276, Assessment and Improvement Item 5. This item of assessment and improvement takes place at regular intervals during an entire project and involves “layers” of project leaders and management. It may lead to re-planning, see Item 4.
 6. **Informative Document Assessment and Improvement:** We refer to Page 277, Assessment and Improvement Item 6 and to Appendix Sects. E.9.1–E.9.3, specifically to Items 1 (Information). Informative documents are usually directed primarily at client and software house management and not so much at software house software engineers. As such they are often the result of the combined labour of client and software house management. Assessments take place while the planned project is being discussed between these partners. Improvements may then be suggested at such mutual project planning meetings.
 7. (a) **Staff and Tool Qualification Assessment** We refer to Page 277, Assessment and Improvement Item 7. This form of assessment is probably the most crucial aspect of SPA (and hence of SPI). It strikes at the core of software development. The resources spent in what is being assessed conventionally represent a very large, a dominating percentage of resource expenditures. Thus this complex of “myriads” of process step, stage and phase (document) assessment must be subject to utmost care.
 7. (b) **Staff Qualification Improvement:** We refer to Page 278, Assessment and Improvement Item 7. The implications of even minor staff improvement actions may be serious: staff well-being, in-availability of staff, serious delays are just a few. Thus improvement planning must be subject to utmost care, both technically and socio-economically, but also as concerns human relations.
 8. **Tool Qualification Improvement:** We refer to Page 279, Assessment and Improvement Item 8. The implications of even minor tool improvement actions may be serious: serious retraining or restaffing, serious time delays, and hence serious cost overruns.

E.5.2 Software Product Management

By software product management we mean the management aspects of deciding which software products to develop and how to price, schedule, market,

sell, service, maintain and, last, but certainly not least, to further develop these products in an evolutionary manner.

You may additionally interpret Sect. E.3, see the second paragraph of Sect. E.5.1, as also significantly implying software product possibilities. We shall outline this latter interpretation below.

• • •

Our treatment of software product management (this section, Sect. E.5.2) will not be as detailed as that of software project management (Sect. E.5.1). The reason for this is twofold: (i) an essence of software product management is that it can be less related to firm principles and techniques, let alone to tools; (ii) the author, although having quite some insight into the area is less sure of “brashly” promulgating such principles and techniques — as compared to those so promulgated for software project management. The field of software product management is too young — software development techniques having yet to stabilise — to expect firm principles and techniques for software product management. The literature in this area is only now beginning to appear (viz. papers and books on, for example, “software product line management” [151, 211, 227]) and this literature is based, so it appears to this author, on rather “old fashioned” views of (only informal) software development.

Summary of Requirements Development Stages

We refer to Figs. E.4 on page 260 and E.5 on page 261. Recall that the domain description usually covers “much more” than is going to be covered by the requirements. That is, the first “act” of domain requirements is to ‘project’ away much of the domain description. Some of the parts that are left out may appear in other domain requirements. In general: Certain kinds of software houses specialises in providing software for a single domain, but for applications spanning an increasingly larger part of that domain. The domain description-to-domain requirements prescription operations of projection, instantiation, determination, and extension aim at one software product or a possibly customizable set of end-user products.

Implied Domain-Specific Software Product Possibilities

Another set of projection, instantiation, determination, and extension domain description-to-domain requirements prescription operations aim at another such (possibly customizable) software product. The domain description-to-domain requirements prescription operation of fitting may then fit two or more products together. And so forth. Once this “spinning off” of a number of domain-specific products has been tried a number of times the software house “discovers”, through the ‘fitting’ operation that ‘features’ of one product are very much like ‘features’ of another product. The paper ‘Development of

Transportation Systems' [91, to appear] shows how a generic, i.e., abstract domain model of transportation can be refined into concrete domain models for road, rail, air and ship transport. Thus [91, to appear] implies sets of requirements for each of these areas. The resulting software designs can, like the requirements prescriptions, be parameterised into software that is then customisable to specific transport modes.

“Middle-layer” and Systems Software Products

But there is another kind of domain-specific software, namely where the domain can be said to reflect a certain intensiveness of the deployed software. Let us roughly speak of information, or process, or translation, or connection, or workpiece software intensities. By information intensive software we understand software which focuses on handling (co-ordinating) large amounts of information. That is, we speak typically of database software. By process intensive software we understand software which focuses on handling (co-ordinating) large numbers of processes. That is, we speak typically of real-time process-switching software. By translation intensive software we understand software which focuses on handling syntactic entities. That is, we speak typically of compilers, interpreters, and theorem provers. By connection intensive software we understand software which focuses on handling (co-ordinating) large numbers of messages between processes. That is, we speak typically of data communication software.

We have, before this subsection, focused on end-user domain-specific applications (such as residing in for example the financial services, the health care, the transportation and other domains). Take, for example, the transportation domain, whether it is for road (personal automobile, taxi or bus) or rail traffic or for the transport of oil or natural gas in pipelines. Common to these is that almost any software support or any of these domains must build on a database of the transportation. Such databases would represent, most likely, road segments, linear rail units and pipeline segments, that is, links between neighbouring street intersections, rail switches (crossovers, etc.) and pipeline joints, in very similar form. Likewise such databases would represent street intersections, rail switches(, crossovers, etc.) and pipeline joints also in very similar form. The idea is therefore obvious, namely to prescribe a database management system that can serve all three application domains. We disregard, for the moment the obvious choice of “building” upon an existing relational database management system. [91, to appear], amongst other software targets, also indicate this shared database property. The point we wish to make is that some software products “cut across” domains. Some (such) systems software may cover several seemingly, or otherwise distinct domains. So a software house that may have started out as identifying itself with a specific domain may soon end up identifying itself with a generic class of “middle-ware” or systems software — or, to paraphrase it, from being a turn-key software house to a commercial, off-the-shelf software (COTS) house.

The Creative Aspects of Software Product Management

Creation, the “Eureka” act of discovery, in this case, of discovering which products to focus on, does not result, we think, as a result of explicit management. But “pointers”, hints, can be given.

Now the software house learns to focus on features and on how to compose these into products. More specifically the software house develops in-house tools to help it develop and compose features. A very fascinating approach to this form of development has been proposed and extensively studied by Batory et al. [12–18].

(And then there is the software that is chance upon truly creatively, typically by young university students unhindered by the conventional wisdom propagated at the Alma Mater.)

Software Evolution

By software evolution we mean that a given, usually field-installed and context-customised COTS product evolves: (i) new features (originating with the original software house), (ii) corrections to identified bugs, (iii) improvements of various kind, etcetera, are introduced to all, or most, or at least a number of such field installations: the software is ‘upgraded’, the software evolves. That evolution sometimes clashes with customisation. To provide for smooth software evolution a number of design precautions must be made during the original software project development, i.e., be present in the original customisable software (COTS) product.

The concept of ‘Evolvable Software Product’ is only now being subject to serious scientific studies [149,150]. Other than mentioning a fascinating project and referring to Sestoft’s lectures [274] we shall not cover this important topic which significantly dictates a number of important software product management issues.

• • •

There are many other software product concepts that help determine the issues facing software product management. We leave it to other entries in this Encyclopedia of Software Engineering to cover these properly.

• • •

Above, in Sects. E.5.2–E.5.2, we have viewed the emergence of new products from the, we could call it “top down” perspective of end-user application domain specificity. When the domain is that of operating computer and communications hardware by means of its system software: Apple OS-X, Linux, Microsoft Vista or other, then a clear description of the domain, that is, of the users and their operating needs: their needs for “hooking” up various equipment and other software, must first be achieved — but rarely is. Many, today typically net-oriented and IT security software packages, emerge from considerations such as just hinted at.

E.6 Believable Software Managers and Management

As a preamble to a closing enumeration of issues of believable managers and management we first, very briefly enumerate issues of project and product believability.

E.6.1 Project Believability \equiv Project Quality

A project is believable, i.e., has quality, if the following properties can all be achieved.

- 1A. Planning: A development project can be planned, and is hence a development project, if a method can a-priori be identified, a method which in deterministic steps of development concisely prescribes how to develop the product.
- 1B. Planning: A development cum applied research project can be planned, and is hence an experimental, applied research project, if management and programmers, that is, domain engineers, requirement engineers and software designers, can agree that it is an experimental, applied research project, and hence needs co-develop the project graph for that application.
- 2A. Estimation: A projects' consumption of resources (time, manpower, machine and other development tools) can be estimated if for each sub-task (of a phase, stage or step) that the method (any method) requires, the effort to carry through this task can be established a-priori.
- 2B. Estimation: For an experimental, applied research project, it must be agreed that estimates are very approximate, in fact: non-binding; otherwise it is not a believable project.
3. Resourcing: A project is 'resourcable' if expected and available resources can be "nicely" mapped onto required resources.
4. Economic: A project is economic, firstly, if the estimated required resources are within expectations, and, secondly, if the actual resources consumed stays within the budget!
5. Trustworthy: A project is trustworthy if (i) the development (or experimental, applied research project) staff and the customer at all times believes that the management is in full control, (ii) if the developers believe that their method will bring them through the project, and (iii) if the customer believes he will get the quality software in the time that the customer expects.
6. Enjoyable: A project is enjoyable if the development staff has intellectual fun, and is being further educated in pursuing it.
Mathematics is great fun: being able to move software development to the intellectual level of treating programs and programming as formal objects is no less fun. The beauty of concise theories, as expressed by elegant specifications and designs, is rewarding. The ability to stand back, abandoning dogmatic, almost religious beliefs (attachments) to one's specifications

and designs, and being able, at little expense, to revise one's thinking dramatically, is a treasure to behold.

Sum total: If a development project is to be believable then all of the above criteria (1A., 2A., 3.–6.) must be fulfilled. If it is to be an experimental, applied research project then criteria 1B., 2B. and 6. must be full-filled.

E.6.2 Product Believability \equiv Project Quality

A product has quality, i.e., is believable, if the following properties can all be achieved.

The Right Product: The product delivers what the customers expect — and may further deliver properties, facilities, that the customers, i.e., user are pleased to use.

To achieve this quality we strongly advice developers to base development on the full TripTych spectrum: domain engineering is what ultimately secures that quality.

The Product is Right: And whatever the product delivers it does so correctly: No bugs, no errors.

To achieve this quality we strongly advice developers to base development on formal development: proofs, model checks and formal tests.

Smooth Transition: The product is “easy” to learn and use.

The product will be “easy to learn and use” if the concepts of the domain clearly “shine” through the requirements, the software design and the code.

Price & Time: The price is right and the delivery time is right.

Smooth Growth: The product allows the customer to “naturally grow into” follow-on products: Enhancements are easy to assimilate.

The product has been designed so as to allow customisation and evolution.

Intellectually Stimulating: Use of the product allow the users to better understand their own domain.

A proper, wide-spanning domain model from which the product can be “simply” domain and interface requirements-developed is a product whose intrinsic concepts reflect a proper understanding of the domain and hence ‘educates’ its users.

E.6.3 Believable Software Managers

A software development project manager is believable if the manager is regarded by the managed staff to be in full control of the part of the software development project under the managers' control, that is, understands clearly and can “translate” into meaningful actions the development method being deployed (its syntax, semantics and pragmatics) such as for example outline in Sect. E.3, the issues as software process assessments and improvements outlined in Sect. E.4, and can relate these SPA and SPI issues to the chosen method — as outlined in Sect. E.5.

E.6.4 Believable Software Management

A software development project's management is believable if all its the managers are believable. And a software house's management is believable if all its software development projects are led by believable managements.

E.7 Conclusion

The conclusions above, that is, in Sects. E.6.3–E.6.4, can be formulated as tersely as they are because we have carefully built up, in the referenced sections, Sects. E.3–E.5, a full set of software development concepts.

Many issues we not dealt with properly: (i) other process models than the TripTych model outlined in Sect. E.3; (ii) risk management, (iii) quality assurance; (iv) software product lines, Sect. E.5.2; (v) software evolution, Sect. E.5.2; (vi) software maintenance in general; etcetera.

It is trusted, in the present entry of the Encyclopedia of Software Engineering that other entries will cover, more adequately, not only these “less properly” dispensed issues, but most like also issues that are treated more substantially in the current entry. Thereby the interested reader will obtain a more realistic picture of the state-of-the-art of Software Engineering 2008.

E.8 Bibliographical Notes

[94, to appear] gives a concise overview of domain engineering; [95, to appear] relates domain and requirements engineering; [90] presents a number of domain engineering research challenges; [96, to appear] additionally presents a rather large example of the container line industry domain. [91, to appear] shows a generic, i.e., abstract domain model of road, rail, air and ship transport.

Finally [87–89], except for this, the management aspects of software engineering, present all the other issues of this Software Engineering Encyclopedia entry in “excruciating” details!

E.9 Software Development Documents

There are three kinds of documents: informative (Items 1. in the documents listings of Sects. E.9.1–E.9.3.), specificational (Items 2. in the documents listings of Sects. E.9.1–E.9.3.) and analytic (Items 3. in the documents listings of Sects. E.9.1–E.9.3.). Informative documents view software development projects as values. Analytic documents view specificational (description, prescription and specification) documents as values.

E.9.1 Domain Engineering Documents

We refer to Fig. E.2 on page 258.

1. Information
 - (a) Name, Place and Date
 - (b) Partners
 - (c) Current Situation
 - (d) Needs and Ideas
 - (e) Concepts and Facilities
 - (f) Scope and Span
 - (g) Assumptions and Dependencies
 - (h) Implicit/Derivative Goals
 - (i) Synopsis
 - (j) Standards Compliance
 - (k) Contracts
 - (l) The Teams
 - i. Management
 - ii. Developers
 - iii. Client Staff
 - iv. Consultants
 - (m) Plans
 - i. Project Graph
 - ii. Budget
 - iii. Funding
 - iv. Accounts
 - (n) Management
 - i. Assessment
 - ii. Improvement
 - A. Plans
 - B. Actions
- i. Studies
- ii. Interviews
- iii. Questionnaires
- iv. Indexed Description Units
- (c) Terminology
- (d) Business Processes
- (e) Facets:
 - i. Intrinsic
 - ii. Support Technologies
 - iii. Management and Organisation
 - iv. Rules and Regulations
 - v. Scripts
 - vi. Human Behaviour
- (f) Consolidated Description
3. Analyses
 - (a) Domain Analysis and Concept Formation
 - i. Inconsistencies
 - ii. Conflicts
 - iii. Incompleteness
 - iv. Resolutions
 - (b) Domain Validation
 - i. Stakeholder Walk-throughs
 - ii. Resolutions
 - (c) Domain Verification
 - i. Theorems and Proofs
 - ii. Model Checking
 - iii. Test Cases and Tests
 - (d) (Towards a) Domain Theory
2. Descriptions
 - (a) Stakeholders
 - (b) The Acquisition Process

E.9.2 Requirements Engineering Documents

We refer to Figs. E.4 and E.5 on page 261.

1. Information
 - (a) Name, Place and Date
 - (b) Partners
 - (c) Current Situation
 - (d) Needs and Ideas (Eurekas, I)
 - (e) Concepts & Facilities (Eurekas, II)
 - (f) Scope & Span
 - (g) Assumptions & Dependencies
 - (h) Implicit/Derivative Goals
 - (i) Synopsis (Eurekas, III)
 - (j) Standards Compliance
 - (k) Contracts, with Design Brief
 - (l) The Teams
 - i. Management
 - ii. Developers
 - iii. Client Staff
 - iv. Consultants
 - (m) Plans
 - i. Project Graph
 - ii. Budget
 - iii. Funding
 - iv. Accounts
 - (n) Management
 - i. Assessment
 - ii. Improvement
 - A. Plans
 - B. Actions
2. Prescriptions
 - (a) Stakeholders
 - (b) The Acquisition Process
 - i. Studies
 - ii. Interviews
 - iii. Questionnaires
 - iv. Indexed Description Units
 - (c) Rough Sketches (Eurekas, IV)
 - (d) Terminology
 - (e) Facets:
 - i. Business Process Re-engineering
 - Sanctity of the Intrinsics
 - Support Technology
 - Management and Organisation
 - Rules and Regulation
 - Human Behaviour
 - Scripting
 - ii. Domain Requirements
 - Projection
 - Determination
 - Instantiation
 - Extension
 - Fitting
3. Analyses
 - (a) Requirements Analysis and Concept Formation
 - i. Inconsistencies
 - ii. Conflicts
 - iii. Incompleteness
 - iv. Resolutions
 - iii. Interface Requirements
 - Shared Phenomena and Concept Identification
 - Shared Data Initialisation
 - Shared Data Refreshment
 - Man-Machine Dialogue
 - Physiological Interface
 - Machine-Machine Dialogue
 - iv. Machine Requirements
 - Performance
 - ★ Storage
 - ★ Time
 - ★ Software Size
 - Dependability
 - ★ Accessibility
 - ★ Availability
 - ★ Reliability
 - ★ Robustness
 - ★ Safety
 - ★ Security
 - Maintenance
 - ★ Adaptive
 - ★ Corrective
 - ★ Perfective
 - ★ Preventive
 - Platform
 - ★ Development Platform
 - ★ Demonstration Platform
 - ★ Execution Platform
 - ★ Maintenance Platform
 - Documentation Requirements
 - Other Requirements
 - v. Full Reqs. Facets Doc.

- (b) Requirements Validation
 - i. Stakeholder Walk-through and Reports
 - ii. Resolutions
- (c) Requirements Verification
 - i. Theorem Proofs
 - ii. Model Checking
 - iii. Test Cases and Tests
- (d) Requirements Theory
- (e) Satisfaction and Feasibility Studies
 - i. Satisfaction: Correctness, unambiguity, completeness, consistency, stability, verifiability, modifiability, traceability
 - ii. Feasibility: Technical, economic, BPR

E.9.3 Software Design Engineering Documents

We refer to Fig. E.7 on page 262.

- 1. Information
 - (a) Name, Place and Date
 - (b) Partners
 - (c) Current Situation
 - (d) Needs and Ideas
 - (e) Concepts and Facilities and Facilities
 - (f) Scope and Span
 - (g) Assumptions and Dependencies
 - (h) Implicit/Derivative Goals
 - (i) Synopsis
 - (j) Standards Compliance
 - (k) Contracts
 - (l) The Teams
 - i. Management,
 - ii. Developers,
 - iii. Consultants
 - (m) Plans
 - i. Project Graph
 - ii. Budget, Funding, Accounts
 - (n) Management
 - i. Assessment Plans & Actions
 - ii. Improvement Plans & Actions
 - 2. Software Specifications
 - (a) Architecture Design ($S_{a_1} \dots S_{a_n}$)
 - (b) Component Design ($S_{c_{1_i}} \dots S_{c_{n_j}}$)
 - (c) Module Design ($S_{m_1} \dots S_{m_m}$)
 - (d) Program Coding (S_{k_1}, \dots, S_{k_n})
 - 3. Analyses
 - (a) Analysis Objectives and Strategies
 - (b) Verification ($S_{i_p}, S_i \sqsupseteq_{L_i} S_{i+1}$)
 - i. Theorems and Lemmas L_i
 - ii. Proof Scripts \wp_i
 - iii. Proofs Π_i
 - (c) Model Checking ($S_i \sqsupseteq P_{i-1}$)
 - i. Model Checkers
 - ii. Propositions P_i
 - iii. Model Checks \mathcal{M}_i
 - (d) Testing ($S_i \sqsupseteq T_i$)
 - i. Manual Testing
 - Manual Tests $M_{S_1} \dots M_{S_\mu}$
 - ii. Computerised Testing
 - A. Unit (or Module) Tests C_u
 - B. Component Tests C_c
 - C. Integration Tests C_i
 - D. System Tests $C_s \dots C_{s_{its}}$
 - (e) Evaluation of Adequacy of Analysis
- Legend:
- S Specification
 - L Theorem or Lemma
 - \wp_i Proof Scripts
 - Π_i Proof Listings
 - P Proposition
 - \mathcal{M} Model Check (run, report)
 - T Test Formulation
 - M Manual Check Report
 - C Computerised Check (run, report)
 - \sqsupseteq "is correct with respect to (wrt.)"
 - \sqsupseteq_ℓ "is correct, modulo ℓ , wrt."

Items 3(b)–3(d) above have been detailed (i–iii, i–iii, i–ii, respectively) more than the corresponding Items 3((c)i–3((c)iii (Page 287, Sect. E.9.1) and

Items 3((c)i–3((c)iii (Page 289, Sect. E.9.2). Naturally, also actions implied by these items need be pursued and documented as diligently as for software design.

F

An Example Domain Model: Intrinsic¹

This appendix consists of an excerpt from a new book: Dines Bjørner: *Software Engineering*:

- Vol. I: The Triptych Method, *Management and Organisation*; and
- Vol. II: A Model Development, Appendix F: *Intrinsic*.

(The book *Software Engineering* (approximately 400 pages) is currently being written.²)

The example of this appendix serves to illustrate:

- a reasonably “complete”³ domain description;
- a systematic use of ‘narrative’, ‘formalisation’ and ‘annotation’ (of formulas)⁴;
- an improved understanding of the event and behaviour concepts, cf. Sects. F.6 and F.8, and
- corresponding examples (see Pages 312–312, respectively Pages 310–311).

¹From Appendix F of [97].

²The (new) *Software Engineering* book is expected to find a final form after fall 2008 and spring/summer 2009 lectures at Techn. Univ. of Graz, Politecnico di Milano, University of Saarland, and Christian-Albrechts University of Kiel.

³We put double, “tongue in cheek” quotes, around the term ‘complete’ since the domain model, of course, does not represent a complete description. But it is suitably indicative of how we envisage such a domain model to be presented.

⁴The current RSL annotations assume a reader (of the new book) who does not know RSL, do know basic mathematical logic, and is not going to learn formal modelling (say using RSL).

F.1 A Transport Example

F.2 An Essence of ‘Transport’

We exemplify a transportation domain. By *transport* we shall mean *the movement of vehicles from hubs to hubs along the links of a net*.

F.3 Business Processes

We sketch an example of some business processes.

Rough Sketching of Some Transport Processes

The basic *entities* of the transportation “business” are the (i) *nets* with their (ii) *hubs* and (iii) *links*, the (iv) *vehicles*, and the (v) *traffic* (of vehicles on the net). The basic *functions* are those of (vi) vehicles entering and leaving the net (here simplified to entering and leaving at hubs), (vii) for vehicles to make movement transitions along the net, and (viii) for inserting and removing links (and associated hubs) into and from the net. The basic *events* are those of (ix) the appearance and disappearance of vehicles, and (x) the breakdown of links. And, finally, the basic behaviours of the transportation business are those of (xi) vehicle journey through the net and (xii) net development and maintenance including insertion into and removal from the net of links (and hubs).

F.4 Entities

F.4.1 Basic Entities

Nets, Hubs and Links

Narrative

1. There are hubs and links.
2. There are nets, and a net consists of a set of two or more hubs and one or more links.

Formalisation

type

- 1 H, L,
- 2 $N = H\text{-set} \times L\text{-set}$

axiom

- 2 $\forall (hs, ls):N \bullet \text{card } hs \geq 2 \wedge \text{card } ls \geq 1$

RSL Annotations

- 1: The type clause **type** H, L, defines two abstract types, also called sorts, H and L, of what is meant to abstractly model “real” hubs and nets. H and L are hereby introduced as type (i.e., sort) names. (The fact that the type clause (1) is “spread” over two lines is immaterial.)
- 2: the type clause **type** N = **H-set** \times **L-set** defines a concrete type N (of what is meant to abstractly model “real” nets).
 - ★ The equal sign, =, defines the meaning of the left-hand side type name, N, to be that of the meaning of
 - ★ **H-set** \times **L-set**, namely Cartesian groupings of, in this case, pairs of sets of hubs (**H-set**) and sets of links (**L-set**), that is,
 - ★ \times is a type operator which, when infix applied to two (or more) type expressions yields the type of all groupings of values from respective types, and
 - ★ **-set** is a type operator which, when suffix applied, to, for example H, i.e., **H-set**, constructs, the type power-set of H, that is, the type of all finite subsets of type H.
 - ★ Similarly for **L-set**.

(The fact that type clause (2), as it appears in the formalisation, is not preceded immediately by the literal **type**, is (still) immaterial: it is part of the type clause starting with **type** and ending with the clause 2.)
- 2: The axiomSet Operations **axiom** $\forall (hs,ls):N \bullet \text{card } hs \geq 2 \wedge \text{card } ls \geq 1$
- Thus we see that a type clause starts with the keyword (or literal) **type** and ends just before another such specification keyword, here **axiom**. That is, a type clause syntactically consists of the keyword **type** followed by one or more sort and concrete type definitions (there were three above).
- And we see that a fragment of a formal specification consists of either type clauses, or axioms, or of both, or, as we shall see later, “much more” !

Hub and Link Identifiers

Narrative

1. There are hub and link identifiers.
2. Each hub (and each link) has an own, unique hub (respectively link) identifiers (which can be observed from the hub [respectively link]).

Formalisation**type**

1 HI, LI

value2a obs_HI: $H \rightarrow HI$, obs_LI: $L \rightarrow LI$

axiom

- 2b $\forall h, h': H, l, l': L \bullet$
 $h \neq h' \Rightarrow \text{obs_HI}(h) \neq \text{obs_HI}(h') \wedge l \neq l' \Rightarrow \text{obs_LI}(l) \neq \text{obs_LI}(l')$

RSL Annotations

- 1: introduces two new sorts;
- 2a: introduces two new observer functions:
 - ★ \rightarrow is here an infix type operators.
 - ★ Infixing L and LI it constructs the type of functions (i.e., function values) which apply to values of type L and yield values of type LI .
- and
- 2b: expresses the uniqueness of identifiers.

In order to model the physical (i.e., domain) fact that links are delimited by two hubs and that one or more links emanate from and are, at the same time incident upon a hub we express the following:

Mutual Hub and Link Referencing

Narrative

1. From any link of a net one can observe the two hubs to which the link is connected.
 - (a) We take this ‘observing’ to mean the following: From any link of a net one can observe the two distinct identifiers of these hubs.
2. From any hub of a net one can observe the one or more links to which are connected to the hub.
 - (a) Again: by observing their distinct link identifiers.
3. Extending Item 1: the observed hub identifiers must be identifiers of hubs of the net to which the link belongs.
4. Extending Item 2: the observed link identifiers must be identifiers of links of the net to which the hub belongs.

We used, above, the concept of ‘identifiers of hubs’ and ‘identifiers of links’ of nets. We define, below, functions (iohs , iols) which calculate these sets.

Formalisation**value**

- 1a $\text{obs_HIs}: L \rightarrow \text{HI-set},$
 2a $\text{obs_LIs}: H \rightarrow \text{LI-set},$

axiom

- 1b $\forall l: L \bullet \text{card } \text{obs_HIs}(l) = 2 \wedge$
 2b $\forall h: H \bullet \text{card } \text{obs_LIs}(h) = 1 \wedge$
 $\forall (hs, ls): N \bullet$
 1(a) $\forall h: H \bullet h \in hs \Rightarrow \forall li: LI \bullet li \in \text{obs_LIs}(h) \Rightarrow$

$$\begin{aligned}
& \exists l':L \bullet l' \in ls \wedge li = \text{obs_LI}(l') \wedge \text{obs_HI}(h) \in \text{obs_HIs}(l') \wedge \\
2(a) \quad & \forall l:L \bullet l \in ls \Rightarrow \\
& \quad \exists h',h'':H \bullet \{h',h''\} \subseteq hs \wedge \text{obs_HIs}(l) = \{\text{obs_HI}(h'), \text{obs_HI}(h'')\} \\
3 \quad & \forall h:H \bullet h \in hs \Rightarrow \text{obs_LIs}(h) \subseteq \text{iols}(ls) \\
4 \quad & \forall l:L \bullet l \in ls \Rightarrow \text{obs_HIs}(h) \subseteq \text{iohs}(hs) \\
\text{value} \quad & \\
& \text{iohs: } H\text{-set} \rightarrow HI\text{-set}, \text{ iols: } L\text{-set} \rightarrow LI\text{-set} \\
& \text{iohs}(hs) \equiv \{\text{obs_HI}(h) \mid h:H \bullet h \in hs\} \\
& \text{iols}(ls) \equiv \{\text{obs_LI}(l) \mid l:L \bullet l \in ls\}
\end{aligned}$$

RSL Annotations

- 1a,2a: Two observer functions are introduced.
- 1b,2b: Universal quantification secure that all hubs and links have prerequisite number of unique (reference) identifiers.
 - ★ 1(a): We read $\forall h:H \bullet h \in hs \Rightarrow \forall li:L \bullet li \in \text{obs_LIs}(h) \Rightarrow \exists l':L \bullet l' \in ls \wedge li = \text{obs_LI}(l') \wedge \text{obs_HI}(h) \in \text{obs_HIs}(l')$: For all hubs (h) of the net ($\forall h:H \bullet h \in hs$) it is the case (\Rightarrow) that for all link identifiers (li) of that hub ($\forall li:L \bullet li \in \text{obs_LIs}(h)$) it is the case that there exists a link of the net ($\exists l':L \bullet l' \in ls$) where that link's (l')s identifier is li and the identifier of h is observed in the link l'.
 - ★ 2(a): We read $\forall l:L \bullet l \in ls \Rightarrow \exists h',h'':H \bullet \{h',h''\} \subseteq hs \wedge \text{obs_HIs}(l) = \{\text{obs_HI}(h'), \text{obs_HI}(h'')\}$: for all ... further reading is left as exercise to the reader.
- 3: Reading is left as exercise to the reader.
- 4: Reading is left as exercise to the reader.
- iohs,iols: These two lines define the signature: name and type of two functions.
- iohs(hs) calculates the set ($\{\dots\}$) of all hub identifiers ($\text{obs_HI}(h)$) for which h is a member of the set, hs, of net hubs.
- iols(ls) calculates in the same manner as does iohs(hs).
We can read the set comprehension expression to the left of the definition symbol \equiv : “the set of all $\text{obs_LI}(l)$ for which (l) l is of type L and such that (•) l is in ls”.

F.4.2 Further Entity Properties

In the above extensive example we have focused on just five entities: nets, hubs, links and their identifiers. The nets, hubs and links can be seen as separable phenomena. The hub and link identifiers are conceptual models of the fact that hubs and links are connected — so the identifiers are abstract models of ‘connection’, or, as we shall later discuss it, the mereology of nets, that is, of how nets are composed. These identifiers are attributes of entities.

F.4.3 Entity Projections

Links and hubs have been modelled to possess link and hub identifiers. A link's "own" link identifier enables us to refer to the link, A link's two hub identifiers enables us to refer to the connected hubs. Similarly for the hub and link identifiers of hubs.

Projection of Unique Identifiers

Narrative

1. Assume conceptual types of links and hubs such that such "pseudo" links and hubs can be compared for equality where the comparison does not include their own or their reference identifiers.
2. By a 'link (hub) identifier reset'
 - (a) we understand a function, `reset_I` which applies to links or hubs, and results in a pseudo-link, respectively a pseudo-hub.
 - (b) For any pseudo-link (pseudo-hub), `reset_I` applied to the result of applying `restore_I` to that pseudo-link (pseudo-hub) results in that pseudo-link (pseudo-hub).
3. By a 'link (hub) identifier restore'
 - (a) we understand a function, `restore_I` which applies to pseudo-links or pseudo-hubs, and results in a link, respectively a hub.
 - (b) For any link (hub), `restore_I` applied to the result of applying `reset_I` to that link (hub) results in that link (hub).
4. By an "other than identifier hub", respectively "... link", comparison', `non_I_eq`, we understand a predicate function which applies either to a pair of hubs or pseudo-hubs or to a pair of links or pseudo-links and yields truth value `true`, if the hubs or pseudo-hubs (links or pseudo-links) are "equal" except for their identifiers.
5. That is, a hub (link) is `non_I_equal` to its reset version.

Formalisation

type

1 `pseudo_H`, `pseudo_L`

value

2(a) `reset_I`: $(H \rightarrow \text{pseudo_H}) \mid (L \rightarrow \text{pseudo_L})$

3(a) `restore_I`: $(\text{pseudo_H} \rightarrow H) \mid (\text{pseudo_L} \rightarrow L)$

axiom

2(b),3(b) $\forall h:H, l:L, ph:\text{Pseudo_H}, pl:\text{Pseudo_H} \bullet$
 $\text{restore_I}(\text{reset_I}(h))=h \wedge \text{restore_I}(\text{reset_I}(l))=l \wedge$
 $\text{reset_I}(\text{restore_I}(ph))=ph \wedge \text{reset_I}(\text{restore_I}(pl))=pl$

value

4 `non_I_eq`: $((H \mid \text{pseudo_H}) \times (H \mid \text{pseudo_H}) \rightarrow \mathbf{Bool})$
 $\mid ((L \mid \text{pseudo_L}) \times (L \mid \text{pseudo_L}) \rightarrow \mathbf{Bool})$

axiom

5 $\forall h:H, l:L \bullet \text{non_I_eq}(h, \text{reset_I}(h)) \wedge \text{non_I_eq}(l, \text{reset_I}(l)) \text{ etc.}$

RSL Annotations

- 1: `pseudo_H` and `pseudo_L` are further undefined types.
- 2(a): The type union (`|`) expression $(H \rightarrow \text{pseudo_H}) \mid (L \rightarrow \text{pseudo_L})$ expresses that `reset_L` either applies to hubs or to links.
- 3(a): Similar to `reset_L`.
- 2(b),3(b),5: The axioms governing the `pseudo_H` and `pseudo_L` types and the `reset_L` and `restore_L` functions
- 4: As for predicates `reset_L` and `restore_L` (line 2(a), respectively line 3(a)), the type of the postulated `non_L_eq` predicate function is of union type.

F.5 Operations

To illustrate the concept of operations⁵ on transport nets we postulate those which “build” and “maintain” the transport nets, that is those road net or rail net (or other) development constructions which add or remove links. (We do not here consider operations which “just” add or remove hubs.) By an operation designator we shall understand the syntactic clause whose meaning (i.e., semantics) is that of an action being performed on a state. The state is here the net. We can also think of an operation designators as a “command”.

Initialising a net must then be that of inserting a link with two new hubs into an “empty” net. Well, the notion of an empty net has not been defined. The axioms, which so far determine nets and which has been given above, appears to define a “minimal” net as just that: two linked hubs !

F.5.1 Syntax

First we treat the syntax of operation designators (“commands”).

Link Insertion and Removal**Narrative**

1. To a net one can insert a new link in either of three ways:
 - (a) Either the link is connected to two existing hubs — and the insert operation must therefore specify the new link and the identifiers of two existing hubs;
 - (b) or the link is connected to one existing hub and to a new hub — and the insert operation must therefore specify the new link, the identifier of an existing hub, and a new hub;
 - (c) or the link is connected to two new hubs — and the insert operation must therefore specify the new link and two new hubs.

⁵We use the terms functions and operations synonymously.

- (d) From the inserted link one must be able to observe identifier of respective hubs.
2. From a net one can remove a link. The removal command specifies a link identifier.

Formalisation

type

```

1 Insert == Ins(s_ins:Ins)
1   Ins = 2xHubs | 1x1nH | 2nHs
1(a)   2xHubs == 2oldH(s_hi1:HI,s_l:L,s_hi2:HI)
1(b)   1x1nH == 1oldH1newH(s_hi:HI,s_l:L,s_h:H)
1(c)   2nHs == 2newH(s_h1:H,s_l:L,s_h2:H)

```

axiom

```

1(d)  $\forall 2oldH(hi',l,hi''):\text{Ins} \bullet hi' \neq hi'' \wedge \text{obs\_LIs}(l) = \{hi', hi''\} \wedge$ 
 $\forall 1old1newH(hi,l,h):\text{Ins} \bullet \text{obs\_LIs}(l) = \{hi, \text{obs\_HI}(h)\} \wedge$ 
 $\forall 2newH(h',l,h''):\text{Ins} \bullet \text{obs\_LIs}(l) = \{\text{obs\_HI}(h'), \text{obs\_HI}(h'')\}$ 

```

type

```

2 Remove == Rmv(s_li:LI)

```

RSL Annotations

- 1: The type clause **type** `Ins = 2xHubs | 1x1nH | 2nHs` introduces the type name `Ins` and defines it to be the union (`|`) type of values of either of three types: `2xHubs`, `1x1nH` and `2nHs`.
 - ★ 1(a): The type clause **type** `2xHubs == 2oldH(s_hi1:HI, s_l:L, s_hi2:HI)` defines the type `2xHubs` to be the type of values of record type `2oldH(s_hi1:HI,s_l:L,s_hi2:HI)`, that is, Cartesian-like, or “tree”-like values with record (root) name `2oldH` and with three sub-values, like branches of a tree, of types `HI`, `L` and `HI`. Given a value, `cmd`, of type `2xHubs`, applying the selectors `s_hi1`, `s_l` and `s_hi2` to `cmd` yield the corresponding sub-values.
 - ★ 1(b): Reading of this type clause is left as exercise to the reader.
 - ★ 1(c): Reading of this type clause is left as exercise to the reader.
 - ★ 1(d): The axiom **axiom** has three predicate clauses, one for each category of `Insert` commands.
 - ◇ The first clause: $\forall 2oldH(hi',l,hi''):\text{Ins} \bullet hi' \neq hi'' \wedge \text{obs_LIs}(l) = \{hi', hi''\}$ reads as follows:
 - For all record structures, `2oldH(hi',l,hi'')`, that is, values of type `Insert` (which in this case is the same as of type `2xHubs`),
 - that is values which can be expressed as a record with root name `2oldH` and with three sub-values (“freely”) named `hi'`, `l` and `hi''`

- (where these are bound to be of type Hl , L and Hl by the definition of $2xHubs$),
- the two hub identifiers hi' and hi'' must be different,
- and the hub identifiers observed from the new link, l , must be the two argument hub identifiers hi' and hi'' .
- ◇ Reading of the second predicate clause is left as exercise to the reader.
- ◇ Reading of the third predicate clause is left as exercise to the reader.

The three types $2xHubs$, $1x1nH$ and $2nHs$ are disjoint: no value in one of them is the same value as in any of the other merely due to the fact that the record names, $2oldH$, $1oldH1newH$ and $2newH$, are distinct. This is no matter what the “bodies” of their record structure is, and they are here also distinct: $(s_hi1:Hl, s_l:L, s_hi2:Hl)$, $(s_hi:Hl, s_l:L, s_h:H)$, respectively $(s_h1:H, s_l:L, s_h2:H)$.

- 2; The type clause **type Remove == Rmv(s_li:Ll)**
 - ★ (as for Items 1(b) and 1(c))
 - ★ defines a type of record values, say rmv ,
 - ★ with record name Rmv and with a single sub-value, say li of type Ll
 - ★ where li can be selected from by rmv selector s_li .

F.5.2 Semantics

Then we consider the meaning of the Insert operation designators.

Semantic Well-formed of Insert Operations

Narrative

1. The insert operation takes an **Insert** command and a net and yields either a new net or **chaos** for the case where the insertion command “is at odds” with, that is, is not semantically well-formed with respect to the net.
2. We characterise the “is not at odds”, i.e., is semantically well-formed, that is: $pre_int_Insert(op)(hs,ls)$, as follows: it is a propositional function which applies to Insert actions, op , and nets, (hs,ls) , and yields a truth value if the below relation between the command arguments and the net is satisfied.
Let (hs,ls) be a value of type N .
 1. If the command is of the form $2oldH(hi',l,hi')$ then
 - ★1 hi' must be the identifier of a hub in hs ,
 - ★2 l must not be in ls and its identifier must (also) not be observable in ls , and
 - ★3 hi'' must be the identifier of a(nother) hub in hs .
 2. If the command is of the form $1oldH1newH(hi,l,h)$ then

- ★1 hi must be the identifier of a hub in hs ,
 - ★2 l must not be in ls and its identifier must (also) not be observable in ls , and
 - ★3 h must not be in hs and its identifier must (also) not be observable in hs .
3. If the command is of the form $2newH(h',l,h'')$ then
- ★1 h' — left to the reader as an exercise (see formalisation !),
 - ★2 l — left to the reader as an exercise (see formalisation !), and
 - ★3 h'' — left to the reader as an exercise (see formalisation !).

Formalisation Conditions concerning the new link (second ★s, ★2, in the above three cases) can be expressed independent of the insert command category.

value

```

1  int_Insert: Insert  $\rightarrow$  N  $\leadsto$  N
2' pre_int_Insert: Ins  $\rightarrow$  N  $\rightarrow$  Bool
2'' pre_int_Insert(Ins(op))(hs,ls)  $\equiv$ 
★2 s_l(op)  $\notin$  ls  $\wedge$  obs_LI(s_l(op))  $\notin$  iols(ls)  $\wedge$ 
   case op of
1)   2oldH(hi',l,hi'')  $\rightarrow$  {hi',hi''}  $\in$  iohs(hs),
2)   1oldH1newH(hi,l,h)  $\rightarrow$ 
      hi  $\in$  iohs(hs)  $\wedge$  h  $\notin$  hs  $\wedge$  obs_HI(h)  $\notin$  iohs(hs),
3)   2newH(h',l,h'')  $\rightarrow$ 
      {h',h''}  $\cap$  hs = {}  $\wedge$  {obs_HI(h'),obs_HI(h'')}  $\cap$  iohs(hs) = {}
   end
```

RSL Annotations

- 1: The value clause **value** $int_Insert: Insert \rightarrow N \leadsto N$ names a value, int_Insert , and defines its type to be $Insert \rightarrow N \leadsto N$, that is, a partial function (\leadsto) from $Insert$ commands and nets (N) to nets.
(int_Insert is thus a function. What that function calculates will be defined later.)
- 2': The predicate $pre_int_Insert: Insert \rightarrow N \rightarrow \mathbf{Bool}$ function (which is used in connection with int_Insert to assert semantic well-formedness) applies to $Insert$ commands and nets and yield truth value **true** if the command can be meaningfully performed on the net state.
- 2'': The action $pre_int_Insert(op)(hs,ls)$ (that is, the effect of performing the function pre_int_Insert on an $Insert$ command and a net state is defined by a case distinction over the category of the $Insert$ command. But first we test the common property:

- $\star 2$: $s_l(op) \notin ls \wedge obs_LI(s_l(op)) \notin iols(ls)$, namely that the new link is not an existing net link and that its identifier is not already known.
- ★ 1): If the **Insert** command is of kind $2oldH(hi', l, hi'')$ then $\{hi', hi''\} \in iohs(hs)$, that is, then the two distinct argument hub identifiers must not be in the set of known hub identifiers, i.e., of the existing hubs hs .
- ★ 2): If the **Insert** command is of kind $1oldH1newH(hi, l, h)$ then ... exercise left as an exercises to the reader.
- ★ 3): If the **Insert** command is of kind $2newH(h', l, h'')$... exercise left as an exercises to the reader.

Some Auxiliary Functions: Hub and Link “Extraction”

Narrative

1. Given a net, (hs, ls) , and given a hub identifier, (hi) , which can be observed from some hub in the net, $xtr_H(hi)(hs, ls)$ extracts the hub with that identifier.
2. Given a net, (hs, ls) , and given a link identifier, (li) , which can be observed from some link in the net, $xtr_L(li)(hs, ls)$ extracts the hub with that identifier.

Formalisation

value

```

1:  $xtr\_H$ :  $HI \rightarrow N \rightsquigarrow H$ 
1:  $xtr\_H(hi)(hs, \_ ) \equiv \text{let } h:H \bullet h \in hs \wedge obs\_HI(h)=hi \text{ in } h \text{ end}$ 
    $\text{pre } hi \in iohs(hs)$ 
2:  $xtr\_L$ :  $HI \rightarrow N \rightsquigarrow H$ 
2:  $xtr\_L(li)(\_, ls) \equiv \text{let } l:L \bullet l \in ls \wedge obs\_LI(l)=li \text{ in } l \text{ end}$ 
    $\text{pre } li \in iols(ls)$ 

```

RSL Annotations

- 1: Function application $xtr_H(hi)(hs, _)$ yields the hub h , i.e. the value h of type H , such that $(\bullet) h$ is in hs and h has hub identifier hi .
- 1: The wild-card, $_$, expresses that the extraction (xtr_H) function does not need the **L-set** argument.
- 2: Left as an exercise for the reader.

Auxiliary Functions: Hub and Link Identifier “Updates”

Narrative

1. When a new link is joined to an existing hub then the observable link identifiers of that hub must be updated to reflect the link identifier of the new link.

2. When an existing link is removed from a remaining hub then the observable link identifiers of that hub must be updated to reflect the removed link (identifier).

Formalisation

value

```

aLI:  $H \times LI \rightarrow H$ , rLI:  $H \times LI \rightsquigarrow H$ 
1: aLI(h,li) as h'
   pre li  $\notin$  obs_LIs(h)
   post obs_LIs(h') = {li}  $\cup$  obs_LIs(h)  $\wedge$  non_Leq(h,h')
2: rLI(h',li) as h
   pre li  $\in$  obs_LIs(h')  $\wedge$  card obs_LIs(h')  $\geq 2$ 
   post obs_LIs(h) = obs_LIs(h')  $\setminus$  {li}  $\wedge$  non_Leq(h,h')
```

RSL Annotations

- 1: The add link identifier function aLI:
 - ★ The function definition clause aLI(h,li) as h' defines the application of aLI to a pair (h,li) to yield an update, h' of h.
 - ★ The pre-condition pre li \notin obs_LIs(h) expresses that the link identifier li must not be observable h.
 - ★ The post-condition post obs_LIs(h) = obs_LIs(h') \setminus {li} \wedge non_Leq(h,h') expresses that the link identifiers of the resulting hub are those of the argument hub except (\setminus) that the argument link identifier is not in the resulting hub.
- 2: The remove link identifier function rLI:
 - ★ The function definition clause rLI(h',li) as h defines the application of rLI to a pair (h',li) to yield an update, h of h'.
 - ★ The pre-condition clause pre li \in obs_LIs(h') \wedge card obs_LIs(h') ≥ 2 expresses that the link identifier li must not be observable h.
 - ★ post-condition clause post obs_LIs(h) = obs_LIs(h') \setminus {li} \wedge non_Leq(h,h') expresses that the link identifiers of the resulting hub are those of the argument hub except that the argument link identifier is not in the resulting hub.

Semantics of the Insert Operation

Narrative

1. If the Insert command is of kind 2newH(h',l,h'') then the updated net of hubs and links, has
 - the hubs hs joined, \cup , by the set {h',h''} and
 - the links ls joined by the singleton set of {l}.
2. If the Insert command is of kind 1oldH1newH(hi,l,h) then the updated net of hubs and links, has

- 2.1 : the hub identified by hi updated, hi' , to reflect the link connected to that hub.
- 2.2 : The set of hubs has the hub identified by hi replaced by the updated hub hi' and the new hub.
- 2.2 : The set of links augmented by the new link.
- 3. If the **Insert** command is of kind $2oldH(hi',l,hi'')$ then
 - 3.1–2 : the two connecting hubs are updated to reflect the new link,
 - 3.3 : and the resulting sets of hubs and links updated.

Formalisation

```

int_Insert(op)(hs,ls)  $\equiv$ 
 $\star_i$  case op of
1   2newH( $h',l,h''$ )  $\rightarrow$  ( $hs \cup \{h',h''\}, ls \cup \{l\}$ ),
2   1oldH1newH( $hi,l,h$ )  $\rightarrow$ 
2.1   let  $h' = aLI(xtr\_H(hi,hs),obs\_LI(l))$  in
2.2   ( $hs \setminus \{xtr\_H(hi,hs)\} \cup \{h,h'\}, ls \cup \{l\}$ ) end,
3   2oldH( $hi',l,hi''$ )  $\rightarrow$ 
3.1   let  $hs\delta = \{aLI(xtr\_H(hi',hs),obs\_LI(l)),$ 
3.2    $aLI(xtr\_H(hi'',hs),obs\_LI(l))\}$  in
3.3   ( $hs \setminus \{xtr\_H(hi',hs),xtr\_H(hi'',hs)\} \cup hs\delta, ls \cup \{l\}$ ) end
 $\star_j$  end
 $\star_k$  pre pre_int_Insert(op)(hs,ls)

```

RSL Annotations

- $\star_i \text{--} \star_j$: The clause **case** op **of** $p_1 \rightarrow c_1, p_2 \rightarrow c_2, \dots, p_n \rightarrow c_n$ **end** is a conditional clause.
- \star_k : The pre-condition expresses that the insert command is semantically well-formed — which means that those reference identifiers that are used are known and that the new link and hubs are not known in the net.
- $\star_i + 1$: If op is of the form $2newH(h',l,h''$ then — the narrative explains the rest;
 - else
- $\star_i + 2$: If op is of the form $1oldH1newH(hi,l,h)$ then
 - ★ 2.1: h' is the known hub (identified by hi) updated to reflect the new link being connected to that hub,
 - ★ 2.2: and the pair $[(\text{updated } hs, \text{updated } ls)]$ reflects the new net: the hubs have the hub originally known by hi replaced by h' , and the links have been simple extended (\cup) by the singleton set of the new link;
 - else
- $\star_i + 3$: 3: If op is of the form $2oldH(hi',l,hi'')$ then
 - ★ 3.1: the first element of the set of two hubs ($hs\delta$) reflect one of the updated hubs,

- ★ 3.2: the second element of the set of two hubs ($hs\delta$) reflect the other of the updated hubs,
 - ★ 3.3: the set of two original hubs known by the argument hub identifiers are removed and replaced by the set $hs\delta$;
- else — well, there is no need for a further ‘else’ part as the operator can only be of either of the three mutually exclusive forms !

Semantics of the Remove Operation

Narrative

1. The remove command is of the form $Rmv(li)$ for some li .
2. We now sketch the meaning of removing a link:
 - (a) The link identifier, li , is, by the pre_int_Remove pre-condition, that of a link, l , in the net.
 - (b) That link connects to two hubs, let us refer to them as h' and h'' .
 - (c) For each of these two hubs, say h , the following holds wrt. removal of their connecting link:
 - i. If l is the only link connected to h then hub h is removed. This may mean that
 - either one
 - or two hubs
 are also removed when the link is removed.
 - ii. If l is not the only link connected to h then the hub h is modified to reflect that it is no longer connected to l .
 - (d) The resulting net is that of the pair of adjusted set of hubs and links.

Formalisation

value

```

1 int_Remove:  $Rmv \rightarrow N \leadsto N$ 
2 int_Remove( $Rmv(li)$ )( $hs, ls$ )  $\equiv$ 
2(a) let  $l = xtr\_L(li)(ls)$ ,  $\{hi', hi''\} = obs\_HIs(l)$  in
2(b) let  $\{h', h''\} = \{xtr\_H(hi', hs), xtr\_H(hi'', hs)\}$  in
2(c) let  $hs' = cond\_rmv(h', hs) \cup cond\_rmv\_H(h'', hs)$  in
2(d)  $(hs \setminus \{h', h''\} \cup hs', ls \setminus \{l\})$  end end end
2(a) pre  $li \in iols(ls)$ 

```

$cond_rmv: LI \times H \times H\text{-set} \rightarrow H\text{-set}$

$cond_rmv(li, h, hs) \equiv$

2((c)i) **if** $obs_HIs(h) = \{li\}$ **then** $\{\}$

2((c)ii) **else** $\{sLI(li, h)\}$ **end**

pre $li \in obs_HIs(h)$

RSL Annotations

- 1: The `int_Remove` operation applies to a remove command `Rmv(li)` and a net (hs, ls) and yields a net — provided the remove command is semantically well-formed.
- 2: To Remove a link identifier by `li` from the net (hs, ls) can be formalised as follows:
 - ★ 2(a): obtain the link `l` from its identifier `li` and the set of links `ls`, and
 - ★ 2(a): obtain the identifiers, $\{h', h''\}$, of the two distinct hubs to which link `l` is connected;
 - ★ 2(b): then obtain the hubs $\{h', h''\}$ with these identifiers;
 - ★ 2(c): now examine `cond_rmv` each of these hubs (see Lines 2((c)i)–2((c)ii)).
 - The examination function `cond_rmv` either yields an empty set or the singleton set of one modified hub (a link identifier has been removed).
 - 2(c) The set, hs' , of zero, one or two modified hubs is yielded.
 - That set is joined to the result of removing the hubs $\{h', h''\}$
 - and the set of links that result from removing `l` from `ls`.
 The conditional hub remove function `cond_rmv`
 - ★ 2((c)i): either yields the empty set (of no hubs) if `li` is the only link identifier `inh`,
 - ★ 2((c)ii): or yields a modification of `h` in which the link identifier `li` is no longer observable.

F.6 Events

F.6.1 Some General Comments

This section shall be very brief. The reason is this: The concept of events and their description is very important. But examples of event descriptions are closely intertwined with examples of behaviour descriptions. We shall therefore postpone the illustration of serious event descriptions till Sect. F.8. After some tiny examples of events and before example of behaviours we insert a section, Sect. F.7, a section which introduces some concepts, like time and time intervals, which are necessary to properly describe events and behaviours.

But first we informally illustrate a number of event scenarios.

F.6.2 Transport Event Examples

(i) A link, for some reason “ceases to exist”; for example: a bridge link falls down, or a level road link is covered by a mud slide, or a road tunnel is afire, or a link is blocked by some vehicle accident. (ii) A vehicle enters or leaves the net. (iii) A hub is saturated with vehicles.

Relating the above three sets of examples of events to our “formal” definition of an event we have these remarks: (i) The state is the transport net of hubs and links at the times observed. We can think of a “link ceasing to exist” as an instantaneous event, i.e., $t_a = t_b$, or as an event that occurs over some time, i.e., $t_a \geq t_b$ — in all cases $\sigma_a \neq \sigma_b$ (see the `int_Remove` function definition Pages 304–305 and our ‘Net Behaviour’ example Pages 311–312).

(ii) The state is the traffic at the times, t_a, t_b , observed. We would say that $t_a = t_b$ but that $\sigma_a \neq \sigma_b$ (state σ_a is state σ_b “plus” or less the vehicle — provided we consider just one vehicle). (iii) The state is the traffic at the times observed. Times are different by a fraction: $t_a \geq t_b$. States are different.

F.6.3 Banking Event Examples

(i) Withdrawal of funds from an account (i.e., a certain action) leads to either of two events: (i.A) either the remaining balance is above or equal to the credit limit, (i.B) or it is not. In the latter case that event may trigger a corrective action. (ii) A national (or federal) bank interest rate change is an action by the the national (or federal) bank, but is seen as an event by any local bank, and may cause such a bank to change (i.e., an action) its own interest rate. (iii) A local bank goes bankrupt.

We leave it to the reader to comment on the time and state relations for the above banking examples.

F.7 Some Fundamental Modelling Concepts

Before we illustrate formal examples of traffic events (Pages 312–312) we must formalise concepts of vehicle (Pages 310–311) and net (Pages 311–312) behaviours. But first we need introduce (and describe: narrate and formalise) some further entities: time (Pages 306–307), time intervals (Pages 307–308), link and hub positions (Pages 308–308), traffic (Pages 309–309) and various notions of traffic well-formedness (meaningful net positions: Pages 309–310, monotonic vehicle movements: Pages 310–311 and no erratic vehicle movements: Pages ??–??).

F.7.1 Time and Time Intervals

Time
Narrative <ol style="list-style-type: none"> 1. Time is here considered an ordered, infinite set of points <ol style="list-style-type: none"> (a) such that for each time there is a unique next time. 2. A proper subset of Time <ol style="list-style-type: none"> (a) is an ordered, possibly infinite set of Time points

- (b) such that there is a minimum, i.e., a smallest, or begin time point and a maximum, i.e., a largest, or an end time point and
- (c) such that for each time in the proper subset, other than the end point time, there is a unique next time.
- 3. Traffics are here considered to be discrete functions from a proper subset of **Time** to pairs of nets and positions of vehicles.
- 4. Positions of vehicles are discrete functions from vehicles to positions.

Formalisation

type

1 **Time**

value

1(a) $\text{next_T: Time} \leadsto \text{Time}$, **pre** $\text{pre_next_T}(t)$

1(a) $\text{pre_next_T: T} \rightarrow \mathbf{Bool}$

1(a) $\text{pre_next_T}(t) \equiv \sim \text{is_end_T}(t)$

type

2 $\text{PSoTime} = \{ | \text{tset: Time-set} \bullet \text{wf_PSoTime}(\text{tset}) | \}$

value

2 $\text{wf_PSoTime: Time-set} \rightarrow \mathbf{Bool}$

2 $\text{wf_PSoTime}(\text{ts}) \equiv$

2(a) $\text{is_ordered}(\text{ts}) \wedge$

2(b) $\exists t_begin, t_end: \text{Time} \bullet$
 $\{t_begin, t_end\} \subseteq \text{ts} \wedge$
 $\forall t: \text{Time} \bullet t \in \text{ts} \Rightarrow t_begin \leq t \leq t_end \wedge$

2(c) $\forall t: \text{Time} \bullet t \in \text{ts} \setminus \{t_end\} \bullet \text{next_T}(t) \in \text{ts}$

type

3 V, P [to be defined later, see Items 1–1]
 [and Items 2(a)–2(b), Page 308]

3 $\text{TF} = \text{Time} \xrightarrow{\text{m}} \mathbf{N} \times \mathbf{VP}$

4 $\mathbf{VP} = \mathbf{V} \xrightarrow{\text{m}} \mathbf{P}$

Time Intervals

Narrative

1. A time interval is a finite passage of time.
2. One cannot add two times, but one can subtract an earlier time from a later time and obtain a time interval.
3. One can add (and subtract) two time intervals and obtain a time interval.
4. One can multiply a real with a time intervals and obtain a time interval.
5. One can divide a time interval by another time interval and obtain a real.
6. One can divide a time interval by a real and obtain a time interval.

7. One can compare pairs of times and pairs of time intervals for smaller than or equal, smaller than, equality, inequality, larger than, or larger than or equal.

We do not specify these operations.

Formalisation

type

1 TI

value

2 $-$: $\text{Time} \times \text{Time} \rightarrow \text{TI}$

3 $*$: $\mathbf{Real} \times \text{TI} \rightarrow \text{TI}$

4 $+, -$: $\text{TI} \times \text{TI} \rightarrow \text{TI}$

5 $/$: $\text{TI} \times \text{TI} \rightarrow \mathbf{Real}$

6 $/$: $\text{TI} \times \mathbf{Real} \rightarrow \text{TI}$

7 $\leq, <, =, \neq, >, \geq$: $(\text{Time} \times \text{Time}) | (\text{TI} \times \text{TI}) \rightarrow \mathbf{Bool}$

We thus use the overloaded operators $-$, $*$, $/$, \leq , $<$, $=$, \neq , $>$, \geq also for time related functions.

F.7.2 Vehicles and Hub and Link Positions

Vehicles and Hub and Link Positions

Narrative

1. There are vehicles, and vehicles are further undefined.
2. There are positions, and a position is either on a link or in a hub.
 - (a) A hub position is indicated just by a triple: the identifier of the hub in question, and a pair of (from and to) link identifiers, namely of links connected to the identified hub.
 - (b) A link position is identified by a quadruplet: The identifier of the link, a pair of hub identifiers (of the link connected hubs), designating a direction, and a real number, properly between 0 and 1, denoting the relative offset from the from hub to the to hub.

Formalisation

type

1 V

2 $P = \text{HP} \mid \text{LP}$

2(a) $\text{HP} == \text{hpos}(s_hi:\text{HI}, s_fli:\text{LI}, s_tli:\text{LI})$

2(b) $\text{LP} == \text{lpos}(s_li:\text{HI}, s_fhi:\text{LI}, s_tli:\text{LI}, s_offset:\text{Frac})$

2(b) $\text{Frac} = \{r:\mathbf{Real} \mid 0 < r < 1\}$

F.8 Behaviours

F.8.1 Traffic as a Behaviour

Traffic

Narrative

1. Traffic is a discrete function from a ‘Proper subset of Time’ to pairs of nets and vehicle positions.
2. Vehicles positions is a discrete function from vehicles to vehicle positions.

We shall have much to say, later, on the well-formedness of traffics.

Formalisation

type

- 1 $\text{TF} = \text{PSoTime} \xrightarrow{\text{m}} (\text{N} \times \text{VehPos}')$
- 2 $\text{VehPos}' = \text{V} \xrightarrow{\text{m}} \text{P}$

axiom

RSL Annotations

- 1 $\xrightarrow{\text{m}}$ is an infix type operator. Applied to types PSoTime and VehPos' it constructs the traffic type of all discrete maps (i.e., function) from values of type PSoTime to values of type VehPos' .
- 2 As for 1.

Traffic: Well-formedness, l, Positions

Narrative

1. All positions recorded in traffics must be positions of the net of the traffic.

Formalisation

value

- 1 $\text{wf_TFc}: \text{TF} \rightarrow \text{Bool}$
 $\text{wf_TFc}(\text{tf}) \equiv$
 $\forall ((\text{hs}, \text{ls}), \text{vp}): (\text{N} \times \text{VehPos}') \bullet ((\text{hs}, \text{ls}), \text{vp}) \in \text{rng } \text{tf} \Rightarrow$
 $\forall \text{p}: \text{P} \bullet \text{p} \in \text{rng } \text{vp} \Rightarrow$
case p of
 $\text{hpos}(\text{li}', \text{hi}, \text{li}'') \rightarrow$
 $\text{hi} \in \text{iohs}(\text{hs}) \wedge$
let $\text{h} = \text{xtr_H}(\text{hi}, \text{hs})$ **in** $\{\text{li}', \text{li}''\} \subseteq \text{obs_HIs}(\text{h})$ **end,**
 $\text{lpos}(\text{hi}', \text{li}, \text{hi}'', \text{f}) \rightarrow$
 $\text{li} \in \text{iols}(\text{ls}) \wedge$
let $\text{l} = \text{xtr_L}(\text{li}, \text{ls})$ **in** $\{\text{hi}', \text{hi}''\} = \text{obs_LIs}(\text{l})$ **end end**

Traffic: Well-formedness, II, Monotonicity

Narrative

1. A traffic must satisfy the following well-formedness properties:
 - (a) If a vehicle is in the traffic at times t' and t'' then it is also in the traffic at all times, t , between t' and t'' .
 - (b) If a vehicle is in the traffic at time t and at position p and at time $next_T(t)$, then its position at time $next_T(t)$ is $next_P(p)$ where $next_P(p)$ is
 - i. either p (the vehicle has either not moved along a link, or is still at a hub),
 - ii. or if p is a link position, $lpos(l_i, fh_j, th_k, f)$, then $next_P(p)$
 - A. is either a hub position ($hpos(l_i, h_j, l_k)$) provided f is infinitesimally [or just very, very] close to 1,
 - B. or is a link position $lpos(l_i, fh_j, th_k, f')$ where f' is $f + \delta_f$, $f + \delta_f < 1$ where δ_f is a positive, very small real between 0 and 1. (That is: the vehicle has moved, but just a little bit.)
 - iii. or if p is a hub position, $hpos(l_i, h_j, l_k, f)$, then $next_P(p)$
 - A. is either the same position p ,
 - B. or is a link position $lpos(h_j, l_k, h_\ell, \delta_f)$ for the other hub identifier, h_ℓ , of the link identified by l_k .
 - (c) A vehicle behaviour, during some time interval, can be seen
 - i. either as a “degenerated” traffic of only one vehicle,
 - ii. or as a sequence of that vehicle’s positions.

It follows from the above that vehicles cannot change direction of movement. We can relax this constraint, but will not do so here.

Formalisation**value**

$$v_in_tf_at_time: V \times TF \times Time \rightarrow \mathbf{Bool}$$

$$v_in_tf_at_time(v, (_, tvp), t) \equiv t \in \mathbf{dom} \, tf \wedge v \in \mathbf{dom} \, tf(t)$$

$$1 \text{ wf_TFa: } (PSoTime \xrightarrow{m} VehPos) \rightarrow \mathbf{Bool}$$

$$wf_TFa(tvp) \equiv$$

$$1(a) \quad \forall t, t': Time \bullet \{t, t'\} \subseteq \mathbf{dom} \, tvp \Rightarrow$$

$$\quad \forall v: V \bullet v_in_tf_at_time(v, tvp, t) \wedge v_in_tf_at_time(v, tvp, t') \Rightarrow$$

$$\quad \forall t'': Time \bullet t < t'' < t' \Rightarrow v_in_tf_at_time(v, tvp, t'')$$
value

$$1 \text{ tf: } TF, v: V, ft, tt: Time$$
axiom

$$1 \text{ is_in_TF}(v)(ft, tt)(tf)$$
value

$$\star 1 \text{ is_in_TF: } V \rightarrow (Time \times Time) \rightarrow TF \rightarrow \mathbf{Bool}$$

```

*1 is_in_TF(v)(ft,tt)(tf) ≡
*2 {ft,tt} ⊆ dom tf ∧
*3 assert {ft,tt} ⊆ dom tf ⇒ ∀ t:Time • ft < t < tt ⇒ t ∈ dom tf
*4 ∀ t:Time • ft < t < tt ⇒ v ∈ dom tf(t)

1((c)i) VehBehtf: V → (Time × Time) → TF
      VehBehtf(v)(ft,tt)(tf) ≡ [ t ↦ [ v ↦ ((tf)(t))(v) ] | t:Time • ft < t < tt ]
      pre is_in_TF(v)(ft,tt)(tf)

1((c)ii) VehBehseq: V → (Time × Time) → P*
      VehBehseq(v)(ft,tt)(tf) ≡ ⟨ ((tf)(t))(v) | t:Time • t in {ft..tt} ⟩
      pre is_in_TF(v)(ft,tt)(tf)

```

F.8.2 A Net Behaviour

A Net Behaviour

Narrative

1. Nets constantly undergo changes:
 - (a) New links are properly inserted.
 - (b) Old links are properly removed.
 - (c) Links “suddenly” ceases to “function”, i.e., appears as having been (improperly) removed.

Formalisation

value

n:N

variable

net:N := n

type

Road_Event == L_Ev(s.li:LI)|...

channel

rch:(Insert|Remove)

ech:Road_Event

value

system: Unit → Unit

dept_of_publ_works: Unit → out ch Unit

road_net: Unit → in rch Unit

net_events: Unit → out ech Unit

system() ≡ dept_of_publ_works() || road_net() || net_events()

dept_of_publ_works() ≡

```

(.. []
  let ins:Insert • pre_int_Insert(ins)(c net) in
  ins!ch end
  []
  let rem:Remove • pre_int_Remove(rem)(c net) in
  ins!ch end
  [] ...); dept_of_publ_works()

net_events() ≡
(skip []
  let Link_Event(li):Road_Event • li ∈ iols(c net) in
  Link_Event(li)!evc end); net_events()

road_net() ≡
(... []
  let cmd = (rch?||ech?) in
  case cmd of
1(a)   Ins(ins) → net := int_Insert(cmd)(c net),
1(b)   Rmv(li) → net := int_Remove(Rmv(li))(c net),
1(c)   L_Ev(li) → net := int_Remove(Rmv(li))(c net),
        ... → ...
  end end); road_net()

```

We model the net behaviour in terms of a system of concurrent behaviours: that of a public works department which non-deterministically ($[]$) issues orders to insert or remove links; that of a net events behaviour which non-deterministically either does nothing or “signals” an appropriate breakdown of a link; and that of the road net behaviour which (external) deterministically reacts to the insert or remove orders or to the link breakdown.

Channels connect these subsidiary, recursive behaviours. A global variable represents the net. It is initialised to some net.

F.9 Traffic Events

The reader shall have to wait for [97] to be published to see the text and formalisation of this section.

F.10 Discussion

The reader shall likewise have to wait for [97] to be published to see the text of this section.

MISCELLANEOUS APPENDICES

Appendices G–J (Pages 315–411) are not to be considered part of the thesis as submitted. They merely represent rough sketch material that may eventually emerge into publishable reports. Attempts to edit some of these appendices into such potential papers will begin during the Fall of 2008.

Appendices K–O (Pages 413–453: excerpted texts, lists, biodata and literature references) are administratively necessary for the main thesis document (Pages 1–120) and its supporting appendices, G–J (Pages 315–411), that is the first 312 pages up till the end of the previous appendix.

G

Documents

This appendix contains the full text and all figures of a report on one aspect of documentation.

- **Documents: A Domain Analysis**
An Experiment in Domain Engineering
- Paper presented at IBM Tokyo Research Laboratory, Wedn. 9 August 2006

This appendix is but a torso.

We aim at exemplifying applications of Domain Engineering. That is, to describe, informally and formally, a man-made universe of discourse. As it is. Without any reference to requirements to any computing system, let alone software to support activities in that domain. The domain is that of documents. We wish to understand what documents are: *original* and *master* documents, *versions*, *copies*, etc., document *creation*, *editing*, *reading*, *copying*, etc. The idea of domain modelling and the style used in this presentation is basically the same — more-or-less independent of the domain. The current version of this report is skimpy on the issue of system space of documents (and agents), of the dynamic (time-wise) movement of documents, and hence on all the very many constraints that govern documents and their locations.

G.1 Introduction

G.1.1 Aims and Objectives

The abstract said it all. But we can reiterate some points. In software development we proceed from modelling the application or business domain, via constructing requirements from the domain model, to designing the computing system, including software. We naturally wish to understand the business area well before we embark on requirements.

Aims

So the aims are to show you aspects of domain modelling. What it includes doing. How one might go about doing it (including analysis).

Objectives

And the objectives are to convince you to do likewise. To do software development professionally.

G.1.2 Domain Engineering

Domain engineering is the engineering, the managed construction of domain descriptions. As such domain engineering includes [1.] identifying, contacting, and throughout consulting all possibly relevant domain stakeholders; [2.] domain knowledge acquisition; [3.] analysing acquired domain knowledge; [4.] creating the domain model¹; [5.–6.] verifying and then validating the model²; [7.] and possibly turning the domain model into a domain theory³. In this paper we shall only illustrate item [4.]. We refer to Part III (Chaps. 8–16) of volume 3 of the three volume book on Software engineering [87–89] for “the ‘full’ story” on Domain Engineering.

G.1.3 Related Work

For the kind of document domain analysis, as we delineate it, there is, to our knowledge, no related work. Well, there is probably some AI-related knowledge engineering (KIF⁴) work and there is definitely some Web-oriented (for example, DAML⁵ Web-ontology) on ‘documents’. There was also some work, in the 1980s, “open (or office) document architecture” (ODA)⁶. But it appears that all of this work is about the textual format and content of documents — well-nigh the only thing we are not interested in modelling in this paper. That is, our model is not about the textual (statistical, linguistic, formatting

¹A domain description is a syntactic construction. A domain model is a selected meaning (semantics) of that description.

²Verification shall ensure that the model is correct (i.e., right). Validation shall ensure that we have the right model.

³A domain theory is the domain model + a(ny) number of theorems about the model: its properties, so to speak.

⁴<http://logic.stanford.edu/kif/kif.html>

⁵The DAML (The DARPA Agent Markup Language) homepage is <http://www.daml.org/>

⁶ODA is a standard document file format created by the ITU-T to replace all proprietary document file formats. It should not be confused with the OASIS Open Document Format for Office Applications, also known as OpenDocument http://en.wikipedia.org/wiki/Open_Document_Architecture.

or other) properties of documents, but of how documents are handled irrespective of their format, that is syntax, and content, that is, its semantics. Well, some of the ODA work might apply!

G.1.4 A Caveat

This is, in principle, a first complete draft of a report, perhaps eventually a publishable paper, on the issue of domain analysis of one view of the domain of documents. It is based on various fragments, some appearing in the authors three volumes on software engineering [87–89], others appearing in draft versions of a report worked on during the early spring of 2006 at JAIST. The present draft report is a complete rewrite of those earlier fragments — with no rereading of these having being done immediately before or during the writing of the current report. It is released to colleagues and some IBM TRL staff before or on 9 August 2006. It is much expected that the author will have time to polish this report into a possibly publishable paper during the early fall of 2006.

G.1.5 Structure of Paper

There are basically three sub-parts to the main part of this paper. In the first we analyse basic aspects of documents: which kinds of documents there are, the operations on documents, document history, etc. In the second part we analyse notions of locations, time and agents. And in the third part we enlarge the scope to that of spaces of documents and agents. A concluding sub-part reviews what has been done, the possible shortcomings of the present work, and hopes for future improvement and extensions.

G.2 Documents: A Domain Analysis

G.2.1 Basics

Analysis: Basic Entities

Think of *documents* as stapled or bound collections a sheets of paper. From blank sheets you can *create* a basically blank document. You and others can write something on the sheets (i.e., *edit* them). You and others can *read* these sheets, i.e., the document.

You can *copy* a document. Lets call the document being copied the *master*. Now two documents exists, one per copying action. To make n (say 100) copies is to copy the either the master a n times, or to mix copying copies (which thereby become “new masters”), or the “original” master. The point is: You know what you are doing: which way you did the copying. Our domain model of documents must capture that.

You can *edit* a document. After editing the document upon which the editing was done no longer “exists”. An edited *version* has been made of that “prior” document. If, say, that basis document (the prior one) was typed, and your editing was made by pencil, then from the edited version you can, in a sense, “capture” (“just now edited”) un-edited (the base, prior) document. In a sense you can “undo” the editing. Our domain model of documents must capture that.

You can *read* a document, say at location ℓ and time τ . Before you read it (at location ℓ and time τ) the document was known to have not been read by you at location ℓ and time τ . Afterwards, that is, after time τ , it is known to have been read by you at location ℓ and time τ .

You can *move*, say distribute, a document. Before it was moved it was at some location ℓ . After the move it is at some other location ℓ' . No two documents can at any time at a same location. A document cannot be at two distinct locations at any (one) time. Our domain model of documents must capture that.

Our domain model of documents must capture a lot more!

Narrative: Basic Entities

1. We postulate sorts, i.e., a types of documents, locations, time, agents and *tex*.
2. With a document we can associate, i.e., we can *observe*
 - (a) the current *location* of the document,
 - (b) the *time* it was most recently operated upon,
 - (c) some identity of the *agent* (f.ex., person) who performed the most recent operation on the document,
 - (d) which *kind of agent operation* was last applied to the document,
 - (e) and we can observe some *text*, “the document contents”.⁷
3. The ‘kinds of operations’ are *tokens*. (See items 2 below.)

Formalisation: Basic Entities

type

1. D, L, T, A

value

- 2(a. $\text{obs_L}: D \rightarrow L$
- 2(b. $\text{obs_T}: D \rightarrow T$
- 2(c. $\text{obs_A}: D \rightarrow A$
- 2(c. $\text{obs_OpKind}: D \rightarrow \text{OpKind}$
- 2(e. $\text{obs_Txt}: D \rightarrow \text{Txt}$

type

3. $\text{OpKind} == \text{cr|ed|rd|cp|sh|mv|...}$

⁷Other than being able to observe the text we shall not deal any further with format, statistical or linguistic properties of texts.

Narrative: Basic Functions

1. From a document one can *observe* the triple of *location*, *time* and performing *agent* of ‘most recent operation’.
2. Documents can be
 - (a) created,
 - (b) edited,
 - (c) read,
 - (d) copied,
 - (e) shredded,
 - (f) moved,
 - (g) and a few other things (...).⁸
1. Editing a document d results in a pair, $(d', (f_e, b_e))$:
 - (a) f_e is a forward editing function.
 - (b) b_e is the inverse, “backward” editing function,
 - (c) and $d' = f_e(d)$ and $d = b_e(d')$,
 - (d) that is, $f_e(b_e(d')) = d'$ and $b_e(f_e(d)) = d$.
 - (e) The pair (f_e, b_e) is a result of the editing. It can only be known a posteriori.

Formalisation: Basic Functions**type**

L, T, A, E
 $LTA = L \times T \times A$

value

- 2(a. crea: $LTA \rightarrow D$
- 2(b. edit: $LTA \rightarrow D \rightarrow D \times E$
- 2(c. read: $LTA \rightarrow D \rightarrow D$
- 2(d. copy: $LTA \rightarrow D \rightarrow D \times D$
- 2(e. shrd: $LTA \rightarrow D \rightarrow \mathbf{Unit}$

1. obs_LTA: $D \rightarrow LTA$

Formalisation: The Editing Functions**type**

$E = FE \times BE$

- 1(a. FE = $D \rightarrow D$
- 1(b. BE = $D \rightarrow D$

value

⁸We shall postpone till later considerations of moving (distributing), sharing , calculating over (incl. searching for) and tracing documents.

1. edit: $LTA \rightarrow D \rightarrow D \times E$

axiom

$\forall d:D, (fe, be):E \bullet d = be(fe(d)), \text{ i.e., } fe \circ be = \lambda d.d \wedge be \circ fe = \lambda d.d$

i.e.:

$\forall lta:LTA, d:D \bullet \text{let } (d', (fe, be)) = \text{edit}(lta)(d) \text{ in}$

1(c. $d' = fe(d) \wedge d = be(d')$ **end**

i.e.:

1(d. $d' = fe(be(d')) \wedge d = be(fe(d))$)

Analysis: Document Identifiers

Let's do a bit more analysis. Documents which have just been created are referred to as *originals*. With any original document we can associate a *unique document identifier*. You may think of the unique document identifier being a simple isomorphic function of three quantities: the identity of the agent who (or which) created the document, a unique representation of the location at which the document was created, and a unique representation of the time at which the document was created. No agent can be at two locations at any one time and perform more than one operation at a time. and operations requires agents, hence all original document identifiers are unique. We shall soon see that no two otherwise distinct documents which may exist at any time can have the same unique identifier.

Thus there are original (“blank”) documents distinguishable, albeit, by unique document identifiers. *Editing* a document need not change its identification. Its is, for example, still “one and the same” sheet of papers, albeit with, perhaps, a different textual contents⁹. *Reading* and *moving* a document need not change its identification either. *Copying* a document results in one more document: the *master* whose identification could (hence in our model will) be the same, since it is “physically” the same, as the document being copied, and the *copy*, the, or a, “new” document — whose identification must be made unique, hence different from the identification of the master document. We suggest that the identification of the copy document be an *isomorphic function* of the identification of the master document and the triplet identification of the location, the location and the agent of copying. Since no agent can be performing more than one operation at a time all documents will have unique identifiers.

Formalisation: Document Identifiers

type

⁹And, if not a collection of sheets of paper, then it might be an electronic document for which we would want the same property as for the humanly manifest document. Yes? Yes!

```

L, T, A, D, E, UDI
LTA = L × T × A
value
  crea: LTA → Unit → D
  edit: LTA → D → D × E
  read: LTA → D → D
  copy: LTA → D → D × D
  ...
  obs_UDI: D → UDI
  udi: (LTA|(LTA×LTA)) → UDI
axiom
  ∀ ℓta:LTA,d:D,e:E •
    obs_UDI(crea(ℓta))=udi(ℓta) ∧
    obs_UDI(d) = let (d',efs)=edit(ℓta)(d) in obs_UDI(d') end ∧
    obs_UDI(d) = obs_UDI(read(ℓta)(d)) ∧
    let (md,cd) = copy(ℓta)(d) in
      obs_UDI(d) = obs_UDI(md) ∧
      obs_UDI(cd) = udi(obs_UDI(md),ℓta) ∧ ... end

```

The last line of the above axiom does not guarantee uniqueness across all documents. That issue will be addressed below.

Analysis: Document Kinds

So there are originals, that is, documents that have been created, but upon which no actions have so far been performed. Then there are documents which have been edited, read, are copies of masters, and hence there are master documents; and there are documents which have been moved. For the time being we shall be content with just these kinds of documents (and hence ‘kind of operation’ tokens).

An edited document may be textually distinct from the document from which it was edited. A document which has been read (at some location and time and by some agent) is, textually, the same as when that document had not been read (at that location and time and by that agent). A document which has been copied becomes a master document, otherwise textually indistinguishable from the document from which the copy was made. The copy of a master document is a copied document and is distinguishable, not textually, but “kind-wise”, from the master document.

Analysis: Document History

So “most recently” edited documents are distinct textually distinct from the documents from which they were edited.

The reason, that is, the pragmatics behind why we introduce the notion of ‘document kind’ is so that we can reason about documents: “That document

has been read, by such and such, several times.” “That document is an edited version of a document which we was read by agent α , from a document which was copied by ..., etc.” We wish to also be able to also reason about locations and times of most recent actions, and about the document as it was before such an action: “the document from which it resulted”.

Although in today’s practical life one may not record, on documents, who first created them (where and when), who edited them (where and when), who read them (where and when), who copied them (where and when), or from which masters they were copied (where and when), etc., we may still, in this everyday world, be able to reason about such things. Therefore we must model it.¹⁰

Narrative: Document Kinds

1. Thus documents are “most recently” either
 - (a) created, i.e., they are [blank] “originals”,
 - (b) edited,
 - (c) read,
 - (d) the basis for copying, i.e., they are “masters”, or
 - (e) the result of copying, i.e., they are copies,
 - (f) etcetera.

Formalisation: Document Kinds

type

1 $mD, D = oD \mid eD \mid rD \mid cD$

axiom

$\forall odoc:oD, edoc:eD, rdoc:rD, cdoc:cD \bullet$

- 1(a) $obs_OpKind(odoc) = cr$
- 1(b) $obs_OpKind(edoc) = ed$
- 1(c) $obs_OpKind(rdoc) = rd$
- 1(d) $obs_OpKind(cdoc) = cp$

$\forall lta:LTA, d:D \bullet$

- 1(a) $obs_OpKind(crea(lta)) = cr$
- 1(b) $obs_OpKind(edit(lta)(d)) = (ed, efs)$
- 1(c) $obs_OpKind(read(lta)(d)) = rd$
- 1(d) **let** $(d', d'') = copy(lta)(d)$ **in** $obs_OpKind(d') = cp \wedge obs_OpKind(d'') \in \{cr, ed, rd, cp, \dots\}$ **end**

...

¹⁰The issue of the certainty (i.e., uncertainty) with which we may say reason will be discussed later.

Narrative: Document History

1. With a document we can associate its history of operations performed on that document. More specifically we can think of a document history to be a list of triples: the operation performed, the document, and a location, time and agent triple.
2. An original document has no pre-history. It has the history that it was created at such and such a location and time by such and such an agent.
3. An edited document has a history which consists of the its pre-history, namely the history of the document from which it was edited, and then the most recent historical fact, namely that it was edited at such and such a location and time by such and such an agent, and that the editing can be captured, i.e., the text of the document from which it was edited, as well as the text resulting from the editing.
4. A document which has “just” been read has the pre-history of the document before it was “just” read as well as the most recent historical fact (location, time and agent of reading).
5. The two documents which have partaken in a copying action: the master and the copy documents have different “most recent” histories, but the same pre-history:
 - (a) The master copy has, as “most recent” history that it served as a master for a copying action (by whom, when and where).
 - (b) The copy has, as “most recent” history that it was the result of a copying action (by whom, when and where).
 - (c) Since the unique identifier of the copy “embeds” that of the master one can identify the master.
 - (d) Perhaps one should “insert” as part of the “most recent” master history the unique identifier of the copy: that identifier can anyway be constructed!
6. Etcetera.

Formalisation: Document History**type**1 $H == D^*$ **value**1 $obs_H: D \rightarrow H$ $wf_H(dl) \equiv len\ dl \geq 1 \wedge dl(1) = create(lta) \wedge create(lta) \notin elems\ tl\ dl$ **axiom** $\forall lta:LTA, d:D \bullet$ 2 $obs_H(create(lta)) = \langle create(lta) \rangle \wedge$ 3 $obs_H(edit(lta)(d)) = obs_H(d) \hat{\ } \langle edit(lta)(d) \rangle \wedge$ 4 $obs_H(read(lta)(d)) = obs_H(d) \hat{\ } \langle read(lta)(d) \rangle \wedge$ 5 **let** (md,cd) = copy(lta)(d) **in**5(a) $obs_H(md) = obs_H(d) \hat{\ } \langle md \rangle \wedge$

5(b) $\text{obs_H}(\text{cd}) = \text{obs_H}(\text{d})^{\wedge} \langle \text{cd} \rangle \wedge$
 5(c) ... **end**

The axiom above essentially defines an observer function

```
value
  obs_prev_D: D → D | nil
axiom
  obs_prev_D(create(lta)) = nil ∧
  obs_prev_D(edit(lta)(d)) = d ∧
  obs_prev_D(read(lta)(d)) = d ∧
  let (md,cd)=copy(lta)(d) in obs_prev_D(md)=d ∧ obs_prev_D(cd)=d end ∧
  ...
```

In other words: It is, of course, always possible, in the domain, to reason about past versions versions of a document. Whether one's reasoning is accurate — one might have forgotten essential aspects — is another matter¹¹

G.2.2 Locations, Time and Agents

We have mentioned that we do not study textual issues. But we are concerned about locations, time and agents.

There are two notions of location: the location at which an operation on a document took, or takes place, and the location of the document “right now”. The latter is, of course (?), the same as the location to where the most recent move action brought that document!

There are two similar notions of time the time at which an operation on a document took, or takes place, and the time “right now”, or at any point in the past, or at any point in the future.

And there is a notion of agent: the person (or other) who (resp. which) performs actions on documents. A document is “full” of location, time and agent attributions. The location, time and agent notions deserve a special study.

Locations

Analysis: Points

We assume a space as a dense set of points, with a point being atomic, but not necessarily infinitesimally “small” in the physical sense of XYX dimension units. Thus any two points that are distinct at most shares that they may be more-or-less close, or more-or-less distant from one another. We may even introduce a notion of some point being “next to” another point, in which case

¹¹— which we should model by introducing non-determinism into our description, but we will leave that to a full, formal paper!

there will probably be an indefinite if not infinite number of points close to a given point, and such that if two points π_i and π_j are next to one another and π_j and π_k are also next to one another, then either π_i is the same as π_k , or π_j is the same as π_k , or they are all three distinct.

Narration: Locations

1. By a location we shall understand a dense set of points,
 - (a) such that for every two distinct points π_i and π_k in the location (if the location contains more than one point)
 - i. there is an ordered set of at least two distinct points in the location, $\{\pi_i, \pi_{i_1}, \dots, \pi_{i_j}, \dots, \pi_{i_{k-1}}, \pi_{i_k}\}$,
 - ii. such that each pair of these points, $\pi_{i_{j-1}}, \pi_{i_j}$, or $\pi_{i_j}, \pi_{i_{j+1}}$, for $i \leq j-1$ and $j \leq k$,
 - iii. are ‘next to’ one another.
1. Two locations are ‘different’ if the two sets are different.
2. Two locations are ‘distinct’ if they are different and they have no points in common (that is, share no points).
3. Two locations are ‘next to’ one another
 - (a) if they are different,
 - (b) if there is at least a non-empty set of points in each location, ps_i and ps_j
 - (c) whose points are “not in the location of the other”,
 - (d) such that pairs of points in ps_i and ps_j are “next to” one another
 - (e) and such that all other points in the two locations are shared.
4. Two locations are ‘neighbouring’ if they are distinct and there is at least one point in each location, p_i and p_j such that these are “next to” one another.

Formalisation: Locations

```

type
1  P, L
value
1  obs_Ps: L  $\rightarrow$  P-set
axiom
1   $\forall \ell:L \bullet \text{let } ps = \text{obs\_Ps}(\ell) \text{ in}$ 
1(a   $\forall pi,pk:P \bullet \{pi,pk\} \subseteq ps \bullet$ 
1((a)i  let  $pl:P^* \bullet \text{elems } pl = ps \wedge \text{len } pl = \text{card } ps \text{ in}$ 
1((a)ii   $\forall j:\text{Nat} \bullet \{j,j+1\} \subseteq \text{inds } pl \bullet$ 
1((a)iii   $\text{next\_to}(pl(j), pl(j+1))$ 
end end
```

Three location relations:

value

- 1 $\text{diff}(\ell_i, \ell_j) \equiv \text{obs_Ps}(\ell_i) \neq \text{obs_Ps}(\ell_j)$
- 2 $\text{dist}(\ell_i, \ell_j) \equiv \text{obs_Ps}(\ell_i) \text{isect } \text{obs_Ps}(\ell_j) = \{\}$
- 3 $\text{next_to}(\ell_i, \ell_j) \equiv$
 - 3(a) $\text{diff}(\ell_i, \ell_j) \wedge$
 - 3(b) $\exists \text{ps_i, ps_j: P-set} \bullet \text{ps_i} \subseteq \text{obs_Ps}(\ell_i) \wedge \text{ps_j} \subseteq \text{obs_Ps}(\ell_j) \wedge$
 - 3(c) $\forall \text{p_i, p_j: P} \bullet \text{p_i} \in \text{ps_i} \wedge \text{p_j} \in \text{ps_j} \bullet \text{p_i} \notin \text{obs_Ps}(\ell_j) \wedge \text{p_j} \notin \text{obs_Ps}(\ell_i) \wedge$
 - 3(d) $\text{next_to}(\text{p_i}, \text{p_j}) \wedge$
 - 3(e) $\text{obs_Ps}(\ell_i) \setminus \{\text{ps_i}\} = \text{obs_Ps}(\ell_j) \setminus \{\text{ps_j}\}$

- 1 $\text{diff: } L \times L \rightarrow \mathbf{Bool}$
- 2 $\text{dist: } L \times L \rightarrow \mathbf{Bool}$
- 3 $\text{next_to: } (P \times P) \mid (L \times L) \rightarrow \mathbf{Bool}$

A remaining location relation:

- 4 $\text{neighbour: } L \times L \rightarrow \mathbf{Bool}$
- 4 $\text{neighbour}(\ell_i, \ell_j) \equiv$
 - 4 $\text{distinct}(\ell_i, \ell_j) \wedge$
 - 4 $\exists \text{ps_i, ps_j: P-set} \bullet$
 - 4 $\text{ps_i} \subseteq \text{obs_Ps}(\ell_i) \wedge \text{ps_j} \subseteq \text{obs_Ps}(\ell_j) \wedge \text{ps_i} \neq \{\} \wedge \text{ps_j} \neq \{\} \wedge$
 - 4 $\forall \text{p_i, p_j: P} \bullet \text{p_i} \in \text{ps_i} \wedge \text{p_j} \in \text{ps_j} \bullet \text{p_i} \notin \text{obs_Ps}(\ell_j) \wedge \text{p_j} \notin \text{obs_Ps}(\ell_i) \wedge$
 - 4 $\text{next_to}(\text{p_i}, \text{p_j})$

Time and Time Intervals*Analysis: Time*

Time, besides philosophically being an elusive issue, can also, as here, be taken concretely. We shall present a set of axioms due to Johan van Benthem [293]. We shall think of time to be a notion that is absolute wrt. some calendar, that is, time includes year, month, day, hour, minute, second, etc., in some simple encoded form.

*Formalisation: Time***axiom**

- [TRANS: Transitivity] $\forall \text{p, p', p'': P} \bullet \text{p} < \text{p'} < \text{p''} \Rightarrow \text{p} < \text{p''}$
- [IRREF: Irreflexitivity] $\forall \text{p: P} \bullet \text{p} \not< \text{p}$
- [LIN: Linearity] $\forall \text{p, p': P} \bullet (\text{p} = \text{p'} \vee \text{p} < \text{p'} \vee \text{p} > \text{p'})$

- [L-LIN: Left Linearity]
 $\forall p, p', p'': P \bullet (p' < p \wedge p'' < p) \Rightarrow (p' < p'' \vee p' = p'' \vee p'' < p')$
- [BEG: Beginning] $\exists p: P \bullet \sim \exists p': P \bullet p' < p$
- [END: Ending] $\exists p: P \bullet \sim \exists p': P \bullet p < p'$
- [SUCC: Successor]
 - [PAST: Predecessors] $\forall p: P, \exists p': P \bullet p' < p$
 - [FUTURE: Successor] $\forall p: P, \exists p': P \bullet p < p'$
- [DENS: Dense] $\forall p, p': P (p < p' \Rightarrow \exists p'': P \bullet p < p'' < p')$
- [DENS: Converse Dense] \equiv [TRANS: Transitivity]
 $\forall p, p': P (\exists p'': P \bullet p < p'' < p' \Rightarrow p < p')$
- [DISC: Discrete]
 $\forall p, p': P \bullet (p < p' \Rightarrow \exists p'': P \bullet (p < p'' \wedge \sim \exists p''': P \bullet (p < p''' < p''))) \wedge$
 $\forall p, p': P \bullet (p < p' \Rightarrow \exists p'': P \bullet (p'' < p' \wedge \sim \exists p''': P \bullet (p'' < p''' < p')))$

Comments on the Axiomatisation: Time

A *strict partial order*, *SPO*, is a point structure satisfying TRANS and IREF. TRANS, IREF and SUCC imply infinite models. TRANS and SUCC may have finite, “looping time” models.

We choose the *SPO* model of time.

Analysis: Time Intervals

Time is “absolute” wrt. some “zero”. A time interval may be measured in years, months, etc., but these are year intervals, month intervals, etc., not “absolute” wrt. some calendar. When we subtract one time (a date etc.) from another such time we get a time interval. We cannot add time, but we can add a time interval to a time and get a time. We can add, subtract and multiply time intervals. If we divide a time interval by a non-zero such we get a fraction, i.e., a real number. Etcetera. At the stage of this paper (/today) we shall not need the notion of time interval, nor the operations on times.

Formalisation: Time and Time Intervals

type

T, TI

value

elapsed_time: $T \times T \rightarrow TI$

–: $(T|TI) \times TI \rightarrow TI$

$$\begin{aligned}
& -: T \times T \rightarrow TI \\
& +: T \times TI \rightarrow T \\
& +: TI \times TI \rightarrow TI \\
& *: TI \times \mathbf{Real} \rightarrow TI \\
& /: TI \times TI \xrightarrow{\sim} \mathbf{Real}
\end{aligned}$$
axiom

$$\begin{aligned}
& \forall t, t': T \bullet t' > t \Rightarrow \exists t\delta: TI \bullet t\delta = t' - t \\
& \forall t: T, t\delta: TI \bullet \exists t': T \bullet \Rightarrow t + t\delta = t' \vee t - t\delta = t' \\
& \forall ti, ti': TI \bullet \exists ti'': TI \bullet ti + ti' = ti'' \vee ti - ti' = ti'' \\
& \dots
\end{aligned}$$
Time and Space*Analysis: Time and Space*

We show an analysis of some time/space notions. The analysis is due to Wayne D. Blizard [125]. In the axiomatisation below entities are the documents (or agents, A and B) and points are (our) locations (p, q and r), with A_q^t expresses that an entity A at time t is at location p (axiom (I)). See our comments below.

Formalisation: Time and Space:

(I)	$\forall A \forall t \exists p : A_p^t$	
(II)	$(A_p^t \wedge A_q^t) \supset p = q$	
(III)	$(A_p^t \wedge B_p^t) \supset A = B$	
(IV)	$(A_p^t \wedge A_p^{t'}) \supset t = t'$	
(V i)	$\forall p, q : N(p, q) \supset p \neq q$	Irreflexivity
(V ii)	$\forall p, q : N(p, q) = N(q, p)$	Symmetry
(V iii)	$\forall p \exists q, r : N(p, q) \wedge N(p, r) \wedge q \neq r$	No isolated pts.
(VI i)	$\forall t : t \neq t'$	
(VI ii)	$\forall t : t' \neq 0$	
(VI iii)	$\forall t : t \neq 0 \supset \exists \tau : t = \tau'$	
(VI iv)	$\forall t, \tau : \tau' = t' \supset \tau = t$	
(VII)	$A_p^t \wedge A_q^{t'} \supset N(p, q)$	
(VIII)	$A_p^t \wedge B_q^t \wedge N(p, q) \supset \sim (A_q^{t'} \wedge B_p^{t'})$	

Annotations: Time and Space

- (II–IV, VII, VIII): The axioms are universally ‘closed’, that is, we have omitted the usual $\forall A, B, p, q, ts$.
- (I): For every entity, A , and every time, t , there is a location, p , at which A is located at time t .
- (II): An entity cannot be in two locations at the same time.
- (III): Two distinct entities cannot be at the same location at the same time.

- (IV): Entities always move: An entity cannot be at the same location at different times. *This is more like a conjecture, and could be questioned.*
- (V): These three axioms define N .
- (V i): Same as $\forall p : \sim N(p, p)$. “Being a neighbour of”, is the same as “being distinct from”.
- (V ii): If p is a neighbour of q , then q is a neighbour of p .
- (V iii): Every location has at least two distinct neighbours.
- (VI): The next four axioms determine the time successor function $'$.
- (VI i): A time is always distinct from its successor: Time cannot rest. There are no time fix points.
- (VI ii): Any time successor is distinct from the begin time. Time 0 has no predecessor.
- (VI iii): Every non-begin time has an immediate predecessor.
- (VI iv): The time successor function $'$ is a one-to-one (i.e., a bijection) function.
- (VII): The *continuous path axiom*: If entity A is at location p at time t , and it is at location q in the immediate next time t' , then p and q are neighbours.
- (VIII): No “switching”: If entities A and B occupy neighbouring locations at time t then it is not possible for A and B to have switched locations at the next time t' .

Discussion of the Blizzard Model of Space/Time

Except for axiom (IV) the system applies to systems of entities that “sometimes” rest, i.e., do not move. These entities are spatial and occupy at least a point in space. If some entities “occupy more” space volume than others, then we may suitably “repair” the notion of the point space P (etc.), however, this is not shown here.

Agents

Analysis: Agents

Agents create and perform operations on documents. Typically agents are humans — but could be machines, i.e., computers.

Narrative: Agents

1. An agent, besides having an identity, $a:A$, is a further undefined notion.
2. No two agents have the same identity, so if two arbitrarily chosen agents (one could “choose” the same agent twice) are different, then they are at least different because of their distinct identity.
3. One could choose to associate various other attributes with agents:
 - (a) the set of documents in *possession* of the agent, say referenced by unique document identifiers,

- (b) the current location of the agent,
 - i. with the possibility of also constraining agents to occupy distinct locations,
 - (c) the granted authority to perform certain (or all) operations on certain (or all), including creating, documents
- but we choose to not deal with that issue in this paper.

Formalisation: Agents

type

- 1. AG, A

value

- 1. obs_A: AG \rightarrow A

axiom

- 3. $\forall a, a': A \bullet a = a' \equiv \text{obs_A}(a) = \text{obs_A}(a') \wedge \text{obs_A}(a) \neq \text{obs_A}(a') \Rightarrow a \neq a'$

value

- 3(a. obs_Ds: AG \rightarrow D-set

- 3(b. obs_L: AG \rightarrow L

axiom

- 3((b)i. $\forall a, a' \bullet a \neq a' \Rightarrow \text{distinct}(\text{obs_L}(a), \text{obs_L}(a'))$)

type

Auth

value

- 3(c. obs_Auths: AG \rightarrow Auth-set

G.2.3 Spaces of Documents and Agents

Narrative: Spaces of Documents and Agents

1. There is a concept of space.
2. Documents and agents coexist in space.
3. All documents and all agents in that space
 - (a) have distinct unique document identifiers and
 - (b) distinct agent identifiers.
4. Some properties of documents can now be re-expressed
 - No two document can have the same
 - i. “most recent time and location”,
 - ii. no two document can have the same “most recent location and agent”, and
 - iii. no two document can have the same “most recent time and agent”¹².
 - (a) We also need to redefine the signature of all document operations — and the related axioms.

¹²Since our axioms implicitly enjoy the universal time quantification this means that no two past operations can have the take place at the same location, at the same location and time and agent.

Formalisation: Spaces of Documents and Agents**type**

1. Ω
 $LT = L \times T, LA = L \times A, TA = T \times A$

value

1. $obs_Ds: \Omega \rightarrow \mathbf{D-set}, obs_As: \Omega \rightarrow \mathbf{AF-set}$
 $obs_LT: D \rightarrow LT, obs_LA: D \rightarrow LA, obs_TA: D \rightarrow TA,$

axiom

3. $\forall \omega: \Omega, d, d': D, ag, ag': AG \bullet \{d, d'\} \subseteq obs_Ds(\omega) \wedge \{ag, ag'\} \subseteq obs_AGs(\omega) \Rightarrow$
 - 3(a). $d = d' \equiv obs_UDI(d) = obs_UDI(d') \wedge$
 - 3(b). $ag = ag' \equiv obs_A(ag) = obs_A(ag') \wedge$
4. $\forall \omega: \Omega, d, d': D \bullet \{d, d'\} \subseteq obs_Ds(\omega) \Rightarrow$
 - 4(i). $d \neq d' \Rightarrow obs_LT(d) \neq obs_LT(d') \wedge$
 - 4(ii). $d \neq d' \Rightarrow obs_LA(d) \neq obs_LA(d') \wedge$
 - 4(iii). $d \neq d' \Rightarrow obs_TA(d) \neq obs_TA(d') \wedge$

Formalisation: Types of Document Operations**value**

- 2(a). $crea: LTA \rightarrow \Omega \times UDI$
- 2(b). $edit: LTA \rightarrow UDI \rightarrow \Omega \rightarrow \Omega \times E$
- 2(c). $read: LTA \rightarrow UDI \rightarrow \Omega \rightarrow \Omega$
- 2(d). $copy: LTA \rightarrow UDI \rightarrow \Omega \rightarrow \Omega \times UDI$
- 2(e). $shrd: LTA \rightarrow UDI \rightarrow \Omega \rightarrow \Omega$

axiom

... /* left as an exercise ! */

G.2.4 Dynamic System of Documents

The dynamics of a agents and documents views these are evolving over time. That is, the system is a total function from time to space of agents and documents.

type
 $DYN = T \rightarrow \Omega$
axiom

...

Axioms over $dyn: DYN$ expresses much the same as did the axioms over space hinted at earlier.

G.2.5 Document Traces

Usually creation and editing of documents are based on a possibly empty set of other documents. Once created or every time edited one would like to be able to trace, not only, as before, the document history of the created or edited document, but also those of the reference documents. So a document trace is a set of document histories. Together the set forms a tree of histories: At the root is the document being traced. Branch nodes designate document versions, with one branch for history of each document reference. See Fig. G.1.

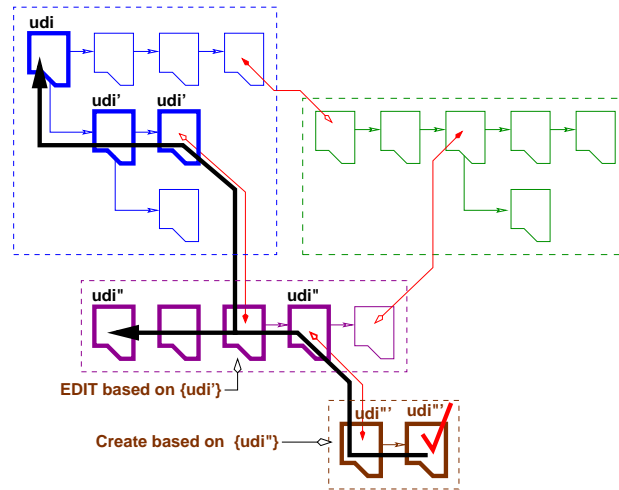


Fig. G.1. Document udi''' trace

Figure G.1 shows a number of documents each with their operations. There are four document families: at top there are two: left and right. In the middle and at bottom of Fig. G.1 there is one each. Each document family (a concept we only diagram) consists of a number of documents — shown as family leaves (to the left in each family): three, two, one and one. The four creates are the leftmost root in each document family. The copies are at the “forks”, and where there are arrows inserted between (non-created) documents of same or different families edits have taken place. References are shown as double-headed links. Might as well insert forward references in those documents which we referenced.

Note:

At the time of the referencing edits the referenced documents must be the most recent document versions of the udi references.

First: Extension of Create and Edit Operations

We extend our create and edit operations. When creating or editing documents we now do so on the basis of zero, one or more references to “background” documents.

We need define an observer function, `obs_UDIs_LTAs`, which when applied to any document, d yields the set of pairs of unique document identifiers (UDIs) and the location, time and agent identifier triples (LTAs) of the documents referenced in d . One needs the LTA triples as the document identified by the UDI may have undergone (several) operations since (first) referenced. Think of the LTA triples being “inserted” in the created or edited document together with the UDI — where these LTA elements are those of the version of the UFI identified document at the time of the create, respectively the edit operation.

value

`crea`: $LTA \times UDI\text{-set} \rightarrow \Omega \rightarrow \Omega \times UDI$

`obs_D`: $UDI \rightarrow \Omega \rightarrow D$

`obs_UDIs_LTAs`: $D \rightarrow (UDI \times LTA)\text{-set}$

axiom

$\forall lta:LTA, udis:UDI\text{-set} \bullet$

let $(\omega', udi) = crea(lta, udis)(\omega)$ **in**

let $od = obs_D(udi)(\omega')$ **in**

$obs_UDI(od) = udi \wedge obs_LTA(od) = lta \wedge$

let $udis_lta = obs_UDIs_LTAs(od)$ **in**

$udis = \{udi' \mid (udi', lta') : UDI \times LTA \bullet (udi', lta') \in udis_lta\}$

end end end

Similar for editing.

Then: Definition of Traces

We leave this as an exercise!

G.2.6 Summary

We could go on and define additional domain notions. The notion of document family used, but not defined above. We could define concepts of document classes, document access authority, agent authorisations: by class, operation and/or document, etcetera. We choose to stop here. In another report: *Public Government: A Domain Analysis* you will find all of the above plus more, put in the pragmatic context of the many agencies of the three branches of government: the law making (parliament, provincial and city councils), the law enforcing (state and local administration) and the law interpreting (the judiciary) branches.

G.3 From Domain Models to Requirements

One rôle for *Domain* descriptions is to serve as a basis for constructing *Requirements* prescriptions. The purpose of constructing *Requirements* prescriptions is to specify properties (not implementation) of a *Machine*. The *Machine* is the hardware (equipment) and the software that together implements the requirements. The implementation relations is

$$\mathcal{D}, \mathcal{M} \models \mathcal{R}$$

The *Machine* is proven to implement the *Requirements* in the context of [assumptions about] the *Domain*. That is, proofs of correctness of the *Machine* wrt. the *Requirements* often refer to properties of the *Domain*.

G.3.1 Domain Requirements

First, in a concrete sense, you copy the domain description and call it a requirements prescriptions. Then that requirements prescription is subjected to a number of operations: (i) removal (projection away) of all those aspects not needed in the requirements; (ii) instantiation of remain aspects to the specifics of the client's domain; (iii) making determinate what is unnecessarily or undesirably non-deterministic in the evolving requirements prescription; (iv) extending it with concepts not feasible in the domain; and (v) fitting these requirements to those of related domains (say monitoring & control of public administration procedures). The result is called a domain requirements.

G.3.2 Interface Requirements

From the domain requirements one then constructs the interface requirements: First one identifies all phenomena and concepts, entities, functions, event and behaviours shared with the environment of the machine (hardware + software) being requirements specified. Then one requirements prescribe how each shared phenomenon and concept is being initialised and updated: entity initialisation and refreshment, function initialisation and refreshment (interactive monitoring and control of computations), and the physiological man-machine and machine-machine implements.

G.3.3 Machine Requirements

Finally one deals with machine requirements performance, dependability, maintainability, portability, etc., where dependability addresses such issues as availability, accessibility, reliability, safety, security, etc.

G.4 Why Domain Engineering?

G.4.1 Two Reasons for Domain Theories

We believe that one can identify two almost diametrically opposed reasons for the pursuit of domain theories. One is utilitarian, concrete, commercial and engineering goal-oriented. It claims that domain engineering will lead to better software, and to development processes that can be better monitored and controlled. and the other is science oriented. It claims that establishing domain theories is a necessity, that it must be done, whither we develop software or not.

We basically take the latter, the science, view, while, of course, noting the former, the engineering consequences. We will briefly look at these.

G.4.2 A Utilitarian, an Engineering Reason

In a recent e-mail, in response, undoubtedly to my steadfast, perhaps conceived as stubborn insistence, on domain engineering (DE), Tony Hoare, possibly in order to come to grips with this “animal” (DE), summed up his reaction to DE as follows, and I quote¹³:

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

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 any design project, and identify those that are independent and those that conflict. Late discovery of feature interactions can be avoided.”

All of these issues are dealt with, in depth, in Vol. 3 my three volume book:

- *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.

¹³E-Mail to Dines Bjørner, CC to Robin Milner et al., July 19, 2006

G.4.3 A Scientific Reason

But, inasmuch as the above-listed issues, so aptly expressed in Tony's mastery, also of words, are of course of utmost engineering importance, it is really, in our mind, the science issues that are foremost: We must first and foremost understand. There is no excuse for not trying to first understand whether that understanding can be "translated" into engineering tools and techniques is then less important. But then, of course, it is nice that understanding also leads to better engineering. It usually does.

G.5 Conclusion

G.5.1 What Have We Shown?

We have established a framework for reasoning about a concept of documents: originals, un-edited and edited documents, documents that have been read or not read, copies of master documents, etc. We have endowed documents with such attributes as unique document identifiers, the location, time and agent of operations performed on documents, the 'kind of operation' ("most recently") performed on documents, document history, etc.

And we have "embedded" documents and agents in a space of such while expressing some, but (we think) far from all relevant properties (axiomatically). In other words, we have started establishing a domain theory for the kind of documents that are of the kinds and have the operations and properties expressed in this paper.

The formalisation of the domain description has been carried out using the RAISE specification language, RSL [166,168]. Emphasis has been on an algebraic style specification using sorts, observer functions and axioms.

G.5.2 What Have We Not Shown?

Non-determinism

We have not shown the vagaries (!) of the domain of documents. In a sense we have shown an idealisation. We can also show the — or an — "actual" domain of documents. In an actual world, documents disappear, for no explainable reason, cannot be accurately "undone", that is, agents can not precisely recall the document text that was edited, partially or fully "loose" their location, time and agent identity, cannot be accurately or fully traced, and many other things.

Actual worlds are non-determinate. So we must model non-determinism. And can.¹⁴

¹⁴In an appendix we show a full formalisation of a non-deterministic domain of documents. This appendix is not included in version

Two Differences between Domains and Requirements

Domains are usually not computable. Requirements must designate computable domain support.

Non-determinism may not be desirable in computable domain support. A purpose of requirements is to secure only desirable non-determinism maybe even remove all such non-determinism for the future domain!

G.5.3 Shortcomings

Immediate shortcomings are believed to be: uncertainty as to whether all relevant properties (axioms) have been formulated, uncertainty as to whether the axioms that have been expressed are consistent (they are not thought of as being complete). Less immediate shortcomings, we think, have to do with the following: Have we expressed the sorts, function signatures and the axioms as succinctly as we would desire? A “gut feeling” about issues that we ought to have also covered!

G.5.4 What Needs Be Done?

Obviously we need to now review the study of the document domains in order to improve it along the lines outlined in the previous section. This will take time. Hopefully colleagues will wish to study this paper. And hopefully the author — and the paper — can then benefit from resulting discussions.

H

Three License Languages

This document was written by Dines Bjørner during the summer and fall of 2006. The co-authors were, at the time, MsC, PhD and Post Doc. students at JAIST, Japan Advanced Institute for Science and Technology, near Kanazawa, Ishikawa Province, West Japan.

- Arimoto Yasuhito, JAIST,
- Dines Bjørner, JAIST and DTU Informatics
- Chen Xiaoyi, JAIST,
- Xiang Jianwen, JAIST,

Summary. Classical digital rights license languages [8, 137–139, 172, 178, 191, 213, 224, 226, 236, 243, 244, 254, 255, 271] were (and are) applied to the electronic “downloading”, payment and rendering (playing) of artistic works (for example music, literature readings and movies). In this document we generalise such applications languages and we extend the concept of licensing to also cover work authorisation (work commitment and promises) in health care and in public government. The digital works for these two new application domains are patient medical records and public government documents.

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’. Proper mathematical definition of licensing languages seeks to ensure smooth and correct computerised management of licenses.

We shall motivate and exemplify three license languages, their pragmatics, syntax and informal as well as formal semantics.

H.1 Introduction

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 On-line [288]

The concepts of licenses and licensing express relations between *actors* (licensors (the authority) and licensees), *simple entities* (artistic works, hospital patients, public administration and citizen documents) and *operations* (on simple 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 operations on which entities the licensee is allowed (is licensed, is permitted) to perform. In this paper we shall consider three 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, and (iii) documents related to public government.

The *permissions* and *obligations* issues are, (i) for the owner (agent) of some intellectual property to be paid (i.e., an *obligation*) by users when they perform *permitted* operations (rendering, copying, editing, sub-licensing) on their works; (ii) for the patient to be professionally treated — by medical staff who are basically *obliged* to try to cure the patient; and (iii) 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.

H.1.1 What Kind of Science Is This?

It is experimental computing science: The study and knowledge of how to design and construct software that is right, i.e., correct, and the right software, i.e., what the user wants. To study methods for getting the right software is interesting. To study methods for getting the software right is interesting. Domain development helps us getting the right software. Deriving requirements from domain descriptions likewise. Designing software from such requirements helps us get the software right. Understanding a domain and then designing license languages from such an understanding is new. We claim that computer-supported management of properly designed license languages is a hallmark of the e-Society.

H.1.2 What Kind of Contributions?

The experimental nature of the project being reported on is as follows: Postulate three domains. Describe these informally and formally. Postulate the possibility of license languages (LLs) that somehow relate to activities of respective domains. Design these – experimentally. Try discover similarities and differences between the three LLs (LL_{DRM} , LL_{HHLL} , LL_{PALL}). Formalise the common aspects: C_{LL} . Formalise the three LLs — while trying to “parameterise” the C_{LL} to achieve LL_{DRM} , LL_{HHLL} , LL_{PALL} . This investigation is bound to tell us something, we hope.

H.2 Pragmatics of The Three License Languages

- 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 three domains of (i) production, distribution and consumption of artistic works (Sect. H.2.1), (ii) the hospitalisation of patient, i.e., hospital health care (Sect. H.2.2), and (iii) the handling of law-based document in public government (Sect. H.2.3). The emphasis is on the pragmatics of the terms, i.e., the language used in these three domains.

H.2.1 The Performing Arts: Producers and Consumers

The intrinsic simple 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 will be analysed in [299].

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.

Past versus Future

In the past all a consumer of digital works could do was to download and possibly copy. We would like, in this document, to investigate what it might entail, with respect to a digital rights license language, to license the consumer to also (copy,) edit and sub-license such digital works.

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.

The Artistic Electronic 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 fulfilment of the first). The second assumption is that the consumer is not allowed to, or cannot operate¹ 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.

An Artistic Digital Rights License Language

In Sects. H.4.1, H.5.1, and H.5.2 we shall design a suitably flexible digital artistic works license language and, through its precise informal and formal description provide one set answers to the above issue.

¹render, copy and edit

H.2.2 Hospital Health Care: Patients and Medical Staff

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 medically 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.

Hospital Health Care: 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.

Hospital Health Care: 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.

A Hospital Health Care License Language

In Sects. H.4.2, H.5.1, and H.5.3 we shall design a suitably flexible hospital health care license language and, through its precise informal and formal description provide one set answers to the above issue.

H.2.3 Public Government and the Citizens

The Three Branches of Government

By public government we shall, following Charles de Secondat, baron de Montesquieu (1689–1755)², understand a composition of three powers: the law-making (legislative), the law-enforcing and the law-interpreting parts of public

²*De l'esprit des lois* (*The Spirit of the Laws*), published 1748

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, including the police and state and district attorneys); and law-interpreting government is called the judiciary (that is, judiciary system, which includes juries and judges etc.).

Documents

A crucial means of expressing public administration is through *documents*.³

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 initiate the following operations on documents: **create**, **edit**, **read**, **copy**, **distribute** (and to whom distributed), **share**, **perform calculations** and **shred**.

With these operations on documents, and hence as attributes of documents one can, again conceptually, associate the *location* and *time* of these operations.

Finally we shall associate with documents the following attributes: (i) **operations**: the name of operations that may be performed on the document; (ii) **actors**: the name of actors and which operations they may perform on the document; (iii) **time-stamped transactions and locations**: a chronological list of operations actually performed on the document and the location at which the document was placed at that time; etcetera. Many other attributes may be associatable. Please recall that we are “in the domain”. This means that we can indeed talk about the above attributes being observable from documents. The documents may just be “good, old-fashioned” paper documents. But someone, some persons, knows, or recalls, or believes in the validity of such attributes or have stamp-marked or “ascribed” the documents in such a way that these attributes can be deduced. Appendix G (Pages 315–337, which reflects [86]) presents “the beginnings” of “a theory of documents” in which these attribute assignments and observations can be done and made.

³Documents are, for the case of public government to be the “equivalent” of artistic works.

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.

A Public Administration Document License Language

Sects. H.4.3, H.5.1, and H.5.4 we shall design a suitably flexible public government document license language and, through its precise informal and formal description provide one set answers to the above issue.

H.3 Semantic Intents of Licensor and Licensee Actions

In this section we shall briefly analyse some of the common semantics of the three kinds of license languages that we intend to design.

H.3.1 Overview

There are two parties to a license: the *licensor* and the *licensee*. And there is a common agreement concerning a shared “item” between them, namely: the *license* and the *work item*: the artistic work, the patient, the document.

The licensor gives the licensee permission, or mandates the licensee to be obligated to perform certain actions on designated “items”.

The licensee performs, or does not perform permitted and/or obligated actions

And the licensee may perform actions not permitted or not obligated.

The license shall serve to ensure that only permitted actions are carried out, and to ensure that all obligated actions are carried out.

Breach of the above, that is, breach of the contracted license may or shall result in revocation of the license.

H.3.2 Licenses and Actions

Licenses

Conceptually a licensor o (for owner) may issue a license named ℓ to licensee u (for user) to perform some actions. The license may syntactically appear as follows:

ℓ : **licensor** o **licenses licensee** u
 to perform actions $\{a_1, a_2, \dots, a_n\}$ **on work item** w

Actions

And, conceptually, the licensee (u) may perform actions which can be expressed as follows:

$\ell : a(w); \ell : a'(w); \dots; \ell : a''(w); \dots; \ell : a'''(w)$

These actions ($a, a', \dots, a'', \dots, a'''$) may be in the set $\{a_1, a_2, \dots, a_n\}$, mentioned in the license, or they may not be in that set. In the latter case we have a breach of license ℓ .

Two Languages

Thus there is *the language of licenses* and *the language of actions*.

We advise the reader to take note of the distinction between the permitted or obligated actions enumerated in a license and the license-name-labelled actions actually requested by a licensee.

H.3.3 Sub-licensing, Scheme I

A licensee u may wish to delegate some of the work associated with performing some licensed actions to a colleague (or customer). If, for example the license originally stated:

ℓ : **licensor** o **licenses licensee** u
 to perform actions $\{a_1, a_2, \dots, a_n\}$ **on work item** w

the licensee (u) may wish a colleague u' to perform a subset of the actions, for example

$\{a_i, a_j, \dots, a_k\} \subseteq \{a_1, a_2, \dots, a_n\}$

Therefore u would like if the above license

ℓ : **licensor** o **licenses licensee** u
 to perform actions $\{a_1, a_2, \dots, a_n\}$ **on work item** w

instead was formulated as:

ℓ : **licensor** o **licenses licensee** u
to perform actions $\{a_1, a_2, \dots, a_n\}$ **on work item** w
allowing sub-licensing of actions $\{a_i, a_j, \dots, a_k\}$

where

$$\{a_i, a_j, \dots, a_k\} \subseteq \{a_1, a_2, \dots, a_n\}$$

Now licensee u can perform the action

ℓ : **license actions** $\{a', a'', a'''\}$ **to** u'

where $\{a', a'', a'''\} \subseteq \{a_i, a_j, \dots, a_k\}$.

The above is an action designator. Its practical meaning is that a license is issued by u :

$\eta(\ell, u, t)$: **licensor** u **licenses licensee** u'
to perform actions $\{a', a'', a'''\}$ **on work item** w

The above license can be easily “assembled” from the action including the action named license: the context determines who (namely u) is issuing the license, and who or which is the work item. η is a function which applies to license name, agent identifications and time and yields unique new license names.

H.3.4 Sub-licensing, Scheme II

The subset relation

$$\{a_i, a_j, \dots, a_k\} \subseteq \{a_1, a_2, \dots, a_n\}$$

mentioned in the sub-licensing part of license

ℓ : **licensor** o **licenses licensee** u
to perform actions $\{a_1, a_2, \dots, a_n\}$ **on work item** w
allowing sub-licensing of actions $\{a_i, a_j, \dots, a_k\}$

may be omitted. In fact one could relax the relation completely and allow any actions $\{a_i, a_j, \dots, a_k\}$ whether in $\{a_1, a_2, \dots, a_n\}$ or not ! That is, the original licensor may mandate that a licensee allow a sub-licensee to perform operations that the licensee is not allowed to perform. Examples are: a licensee may break the shrink-wrap around some licensed software package — an action which may not be performed by the licensor; a medical nurse (i.e., a licensee) may perform actions on patients not allowed performed by the licensor (say, a medical doctor); and a civil servant (say, an archivist) may copy, distribute or shred documents, actions that may not be allowed by the licensor (i.e., the manager of that civil servant), while that civil servant (the archivist) is not allowed to create or read documents.

H.3.5 Multiple Licenses

Consider the following scenario: A licensee L is performing actions a_p, a_q, \dots, a_r , on work item ω , and has licensed L' to perform actions a_i, a_j, \dots, a_k , also on work item ω . The action whereby L licenses L' occurs at some time. At that time L has performed none or some of the actions a_p, a_q, \dots, a_r (on work item ω), but maybe not all. What is going to happen? Can L and L' go on, in parallel, performing actions on the same work item (ω)? Our decision is yes, and they can do so in an interleaved manner, not concurrently but alternating, i.e., not accessing the same work item simultaneously.

H.4 Syntax and Informal Semantics

We distinguish between the pragmatics, the semantics and the syntax of languages. Leading textbooks on (formal) semantics of programming languages are [146, 173, 263, 272, 291, 295].

We have already covered the concept of pragmatics and we have, in covering some basic issues of semantics, illustrated their application to some issues of license language design.

We shall now illustrate the use of syntax and semantics in license language design.

- By **syntax** we mean (i) the ways in which words are arranged to show meaning (cf. semantics) within and between sentences, and (ii) rules for forming syntactically correct sentences.
- By **semantics** we mean the study and knowledge, including specification, of meaning in language [144].
- By **informal semantics** we mean a semantics which is expressed in concise natural language, for example, as here, English.

H.4.1 A General Artistic License Language

We refer to the abstract syntax formalised below (that is, formulas 0.–8. [Page 349] and 9.–14. [Page 351]). The work on the specific form of the syntax has been facilitated by the work reported by Xiang JenWen [299].⁴

The Syntax

The syntax has two parts. One for licenses being issued by licensors. And one for the actions that licensees may wish to perform.

⁴As this work, [299], has yet to be completed the syntax and annotations given here may change.

Licenses

We first present an abstract syntax of the language of licenses, then we annotate this abstract syntax, and finally we present an informal semantics of that language of licenses.

type

0. L_n, N_m, W, S, V
1. $L = L_n \times \text{Lic}$
2. $\text{Lic} == \text{mkLic}(\text{licensor}:N_m, \text{licensee}:N_m, \text{work}:W, \text{cmds}:\text{Cmd-set})$
3. $\text{Cmd} == \text{Rndr} \mid \text{Copy} \mid \text{Edit} \mid \text{RdMe} \mid \text{SuLi}$
4. $\text{Rndr} = \text{mkRndr}(\text{vw}:(V \mid \text{"work"}), \text{sl}:S^*)$
5. $\text{Copy} = \text{mkCopy}(\text{fvw}:(V \mid \text{"work"}), \text{sl}:S^*, \text{tv}:V)$
6. $\text{Edit} = \text{mkEdit}(\text{fvw}:(V \mid \text{"work"}), \text{sl}:S^*, \text{tv}:V)$
7. $\text{RdMe} = \text{"readme"}$
8. $\text{SuLi} = \text{mkSuLi}(\text{cs}:\text{Cmd-set}, \text{work}:V)$

Syntax Annotations

0: Syntax Sorts: (0.) Licenses are given names, $\text{ln}:L_n$, so are actors (owners, licensors, and users, licensees), $\text{nn}:N_m$. By $w:W$ we mean a (net) reference to (a name of) the downloaded possibly segmented artistic work being licensed, and where segments are named ($s:S$), that is, $s:S$ is a selector to either a segment of a downloaded work or to a segment of a copied and or and edited work.

License Name and License Body: (1.) Every license ($\text{lic}:L$) has a unique name ($\text{ln}:L_n$). (2.) A license ($\text{lic}:L$) contains four parts: the name of the licensor, the name of the licensee, a reference to (the name of) the work and a set of command designators (that may be permitted to be performed on the work).

Commands: (3.) A command is either a **render**, a **copy** or an **edit** or a **readme**, or a sub-licensing (**sub-license**) command.

Render, Copy and Edit: (4.–6.) The render, copy and edit commands are each “decorated” with an ordered list of selectors (i.e., selector names) and a (work) variable name. The license command

copy $\langle s_1, s_7, s_2 \rangle v$

means that the licensed work, ω , may have its sections s_1 , s_7 and s_2 copied, in that sequence, into a new variable named v . Other copy commands may specify other sequences. Similarly for render and edit commands.

Read Me: (7.) The "readme" license command, in a license, `ln`, referring, by means of `w`, to work ω , somehow displays a graphical/textual "image" of information about ω . We do not here bother to detail what kind of information may be so displayed. But you may think of the following display information names of artistic work, artists, authors, etc., names and details about licensed commands, a table of fees for performing respective licensed commands, etcetera.

Sub-Licensing: (8.) The license command

```
license cmd1,cmd2,...,cmdn on work v
mkSuLi({cmd1,cmd2,...,cmdn},v)
```

means that the licensee is allowed to act as a licensor, to name sub-licensees (that is, licensees) freely, to select only a (possibly full) subset of the sub-licensed commands (that are listed) for the sub-licensees to enjoy. The license need thus not mention the name(s) of the possible sub-licensees. But one could design a license language, i.e., modify the present one, to reflect such constraints. The license also does not mention the payment fee component. As we shall see under licensor actions such a function will eventually be inserted.

Informal Semantics

A license licenses the licensee to render, copy, edit and license (possibly the results of editing) any selection of downloaded works. In any order — but see below — and any number of times. For every time any of these operations take place payment takes place according to the payment function (which can be inspected by means of the "readme" command). The user can render the downloaded work and can render copies of the work as well as edited versions of these. Edited versions are given own names. Editing is always of copied versions. Copying is either of downloaded or of copied or edited versions. This does not transpire from the license syntax but is expressed by the licensee, see below, and can be checked and duly paid for according to the payment function.

The payment function is considered a partial function of the selections of the work being licensed.

Please recall that licensed works are proprietary. Either the work format is known, or it is not supposed to be known. In any case, the rendering, editing, copying and the license-“assembling” functions (see next section) are part of the license and the licensed work and are also assumed to be proprietary. Thus the licensee is not allowed to and may not be able to use own software for rendering, editing, copying and license assemblage.

Licenses specify sets of permitted actions. Licenses do not normally mandate specific sequences of actions. Of course, the licensee, assumed to be an un-cloned human, can only perform one action at a time. So licensed actions are carried out sequentially. The order is not prescribed, but is decided upon

by the licensee. Of course, some actions must precede other actions. Licensees must copy before they can edit, and they usually must edit some copied work before they can sub-license it. But the latter is strictly speaking not necessary.

Actions

type

9. V
10. $Act = Ln \times (Rndr|Copy|Edit|License)$
11. $Rndr == mkR(sel:S^*, wrk:(W|V))$
12. $Copy == mkC(sel:S^*, wrk:(W|V), into:V)$
13. $Edit == mkE(wrks:V^*, into:V)$
14. $License == mkL(ln:Ln, licensee:Nm, wrk:V, cmds:Cmd\text{-}set, fees:PF)$

Annotations and Informal Semantics:

Variables: (9.) By V we mean the name of a variable in the users own storage into which downloaded works can be copied (now becoming a local work). The variables are not declared. They become defined when the licensee names them in a copy command. They can be overwritten. No type system is suggested.

Actions: (10.) Every action of a licensee is tagged by the name of a relevant license; if the action is not authorised by the named license then it is rejected; render and copy actions mention a specific sequence of selectors; and if this sequence is not an allowed (a licensed) one, then the action is rejected. (Notice that the license may authorise a specific action, a with different sets of sequences of selectors — thus allowing for a variety of possibilities as well as constraints.)

Render: (11.) The licensee, having now received a license, can render selections of the licensed work, or of copied and/or edited versions of the licensed work. No reference is made to the payment function. When rendering, the semantics is that this function is invoked and duly applied. That is, render payments are automatically made: subtracted from the licensees account and forwarded to the licensor.

Copy: (12.) The licensee can copy selections of the licensed work, or of previously copied and/or edited versions of the licensed work. The licensee identifies a name for the local storage file where the copy will be kept. No reference is made to the payment function. When copying, the semantics is that this function is invoked and duly applied. That is, copy payments are automatically made: subtracted from the licensees account and forwarded to the licensor.

Edit: (13.) The licensee can edit selections of the licensed work, or of copied and/or previously edited versions of the licensed work. The licensee identifies a name for the local storage file where the new edited version will be kept. The result of editing is a new work. No reference is made to the payment function. When editing, the semantics is that this function is invoked and duly applied. That is, edit payments are automatically made: subtracted from the licensee's account and forwarded to the licensor.

(a) Although no reference is made to any edit functions these are made available to the licensee when invoking the edit command. (b) You may think of these edit functions being downloaded at the time of downloading the license. (c) Other than this we need not further specify the editing functions.

Remarks (a,b,c) apply also to the above copying function.

Sub-Licensing: (14.) The licensee can further sub-license copied and/or edited work. The licensee must give the license being assembled a unique name. And the licensee must choose to whom to license this work. A sub-license, like does a license, authorises which actions can be performed, and then with which one of a set of alternative selection sequences. No payment function is explicitly mentioned. It is to be semi-automatically derived (from the originally licensed payment fee function and the licensee's payment demands) by means of functionalities provided as part of the licensed payment fee function.

The sub-license command information is thus compiled (assembled) into a license of the form given in (1.–3.) and schematised in the “ $\eta(\ell, u, t)$ ” labelled command designator on Page 347. The licensee becomes the licensor and the recipient of the new, the sub-license, become the new licensee. The assemblage refers to the context of the action. That context knows who, the licensor, is issuing the sub-license. From the license label of the command it is known whether the sub-license actions are a subset of those for which sub-licensing has been permitted.

H.4.2 A Hospital Health Care License Language

A reading of this section assumes that of having read the previous section.

We refer to the abstract syntax formalised below (that is, formulas 0.–7. Page 354). The work on the specific form of the syntax has been facilitated by the work reported in [7].⁵

A Notion of License Execution States

In the context of the Artistic License Language licensees could basically perform licensed actions in any sequence and as often as they so desired. There were, of course, some obvious constraints. Operations (not to be confused with hospital surgery on patients) on local works could not be done before these

⁵As this work, [7], has yet to be completed the syntax and annotations given here may change.

had been created — say by copying. Editing could only be done on local works and hence required a prior action of, for example, copying a licensed work. In the context of hospital health care most of the actions can only be performed if the patient has reached a suitable state in the hospitalisation. We refer to Fig. H.1 for an idealised hospitalisation plan.

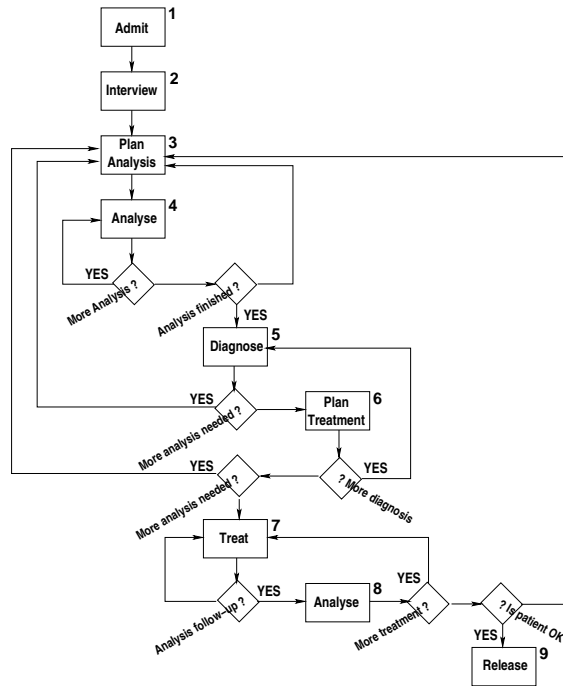


Fig. H.1. An example hospitalisation plan. States: {1,2,3,4,5,6,7,8,9}

We therefore suggest to join to the licensed commands an indicator which prescribe the (set of) state(s) of the hospitalisation plan in which the command action may be performed.

Two or more medical staff may now be licensed to perform different (or even same!) actions in same or different states. If licensed to perform same action(s) in same state(s) — well that may be “bad license programming” if and only if it is bad medical practice! One cannot design a language and prevent it being misused!

The Syntax

The syntax has two parts. One for licenses being issued by licensors. And one for the actions that licensees may wish to perform.

*Licenses***type**

0. Ln, Mn, Pn
1. License = Ln \times Lic
2. Lic == mkLic(s_staff1:Mn,s_mandate:ML,s_pat:Pn)
3. ML == mkML(s_staff2:Mn,s_to-perform-acts:CoL-**set**)
4. CoL = Cmd | ML | Alt
5. Cmd == mkCmd(s_σs:Σ-**set**,s_stmt:Stmt)
6. Alt == mkAlt(cmds:Cmd-**set**)
7. Stmt = **admit** | **interview** | **plan-analysis** | **do-analysis**
| **diagnose** | **plan-treatment** | **treat** | **transfer** | **release**

The above syntax is correct RSL [87–89,97,165,166,168]. But it is decorated!
The subtypes {**boldface keyword**} are inserted for readability.

Syntax: Annotations: (0.) Licenses, medical staff and patients have names.

- (1.) Licenses further consist of license bodies (Lic).
- (2.) A license body names the licensee (Mn), the patient (Pn), and,
- (3.) through the “mandated” licence part (ML), it names the licensor (Mn) and which set of commands (C) or (o) implicit licenses (L, for a “total” mnemonic identifier: CoL) the licensor is mandated to issue.
- (4.) An explicit command or licensing (CoL) is either a command (Cmd), or a sub-license (ML) or an alternative (Alt).
- (5.) A command (Cmd) is a set-of-states-labelled statement.
- (3.) A sub-license (ML) just states the command set that the sub-license licenses. As for the Artistic License Language the licensee chooses an appropriate subset of commands. The context “inherits” the name of the patient. But the sub-licensee is explicitly mandated in the license!
- (6.) An alternative (Alt) is also just a set of commands. The meaning is that either the licensee choose to perform the designated actions or, as for ML, but now freely choosing the sub-licensee, the licensee (now new licensor) chooses to confer actions to other staff.
- (7.) A statement (Stmt) is either an admit, an interview, a plan analysis, an analysis, a diagnose, a plan treatment, a treatment, a transfer, or a release directive. Information given in the patient medical report, for the designated state, inform medical staff as to the details of analysis, what to base a diagnosis on, of treatment, etc.

Actions

8. Action = Ln \times Act
9. Act = Stmt | SubLic
10. SubLic = mkSubLic(sublicensee:Ln,license:ML)

Syntax Annotations: (8.) Each action actually attempted by a medical staff refers to the license, and hence the patient name.

(9.) Actions are either of an admit, an interview, a plan analysis, an analysis, a diagnose, a plan treatment, a treatment, a transfer, or a release actions. Each individual action is only allowed in a state σ if the action directive appears in the named license and the patient (medical record) designated state set σs .

(10.) Or an action can be a sub-licensing action. Either the sub-licensing action that the licensee is attempting is explicitly mandated by the license (4. ML), or is an alternative one thus implicitly mandated (6.). The full sub-license, as defined in (1.–3. on Page 354) is compiled from contextual information.

Informal Semantics

An informal, rough-sketch semantics (here abbreviated) would state that a prescribed action is only performed if the patient, cum patient medical record is in an appropriate state; that the patient is being treated according to the action performed; and that records of this treatment and its (partially) analysed outcome is introduced into the patient medical record. The next state of the patient, cum patient medical record, depends on the outcome of the treatment⁶; and hence the patient medical record carries with it, i.e., embodies a, or the, hospitalisation plan in effect for the patient, and a reference to the current state of the patient.

H.4.3 A Public Administration License Language

In this appendix we shall assume that the reader has studied, or at least can refer to Appendix G (Pages 315–337, [86]).

We refer to the abstract syntax formalised below (that is, formulas (0.–19.), Page 356 and formulas (20.–31.), starting Page 358). The work on the specific form of the syntax has been facilitated by the work reported in [86, 99, 136].⁷

The Syntax: Licenses

The syntax has two parts. One for licenses being issued by licensors. And one for the actions that licensees may wish to perform.

⁶Cf. the diamond-shaped decision boxes in Fig. H.1 on page 353.

⁷As part this work, [136], has yet to be completed the syntax and annotations given here may change.

*The Form of Licenses***type**

- 0. Ln, An, Dn, DCn, Cfn
- 1. L == Grant | Extend | Restrict | Withdraw
- 2. Grant == mkG(s_license:Ln,s_licensor:An,s_ops:OpDocs,s_licensee:An)
- 3. Extend == mkE(s_licensor:An,s_licensee:An,s_license:Ln,s_with_ops:OpDocs)
- 4. Restrict == mkR(s_licensor:An,s_licensee:An,s_license:Ln,s_to_ops:OpDocs)
- 5. Withdraw == mkW(s_licensor:An,s_licensee:An,s_license:Ln)
- 6a. OpDocs = Op \overrightarrow{m} (Dn \overrightarrow{m} DCn)
- 6b. Op == Crea|Edit|Read|Copy|Licn|Shar|Rvok|Rlea|Rtur|Calc|Shrd

*Licensed Operations***type**

- 7. UDI
- 8. Crea == mkCr(s_dn:Dn,s_doc_class:DCn,s_based_on:UDI-**set**)
- 9. Edit == mkEd(s_doc:UDI,s_based_on:UDI-**set**)
- 10. Read == mkRd(s_doc:UDI)
- 11. Copy == mkCp(s_doc:UDI)
- 12. Licn == mkLi(s_kind:LiTy)
- 13. LiTy == grant | extend | restrict | withdraw
- 14. Shar == mkSh(s_doc:UDI,s_with:An-**set**)
- 15. Rvok == mkRv(s_doc:UDI,s_from:An-**set**)
- 16. Rlea == mkRl(s_dn:Dn)
- 17. Rtur == mkRt(s_dn:Dn)
- 18. Calc == mkCa(s_fcts:CFn-**set**,s_docs:UDI-**set**)
- 19. Shrd == mkSh(s_doc:UDI)

Syntax & Informal Semantics Annotations: Licenses

(0.) The are names of licenses (Ln), actors (An), documents (Dn), document classes (DCn), calculation functions (Cfn).

(1.) There are four kinds of licenses: granting, extending, restricting and withdrawing.

(2.) Actors (licensors) grant licenses to other actors (licensees). An actor is constrained to always grant distinctly named licenses. No two actors grant identically named licenses.⁸ A set of operations on named documents (OpDocs) are granted.

(3.–5.) Actors who have issued named licenses may extend, restrict or withdraw the license rights (wrt. operations).

⁸This constraint can be enforced by letting the unique actor name be part of the license name.

(6a.) To each granted operation there is associated a set of document names (each of which is associated with a document class name, i.e., a type).

(6b.) There are nine kinds of operation (Op) authorisations.

Some of the next explications also explain parts of some of the corresponding actions (see (20.–32.) Page 358).

(7.) There are unique document identifiers. Several documents may be “of the same” name, but each created document “of that name” is ascribed a unique document identifier.

(8.) Creation results in an initially void document which is not necessarily uniquely named (dn:Dn) (but that name is associated with the unique document identifier created when the document is created):

value

obs_Dn: UID \rightarrow Dn, obs_DCn: UID \rightarrow DCn, obs_An: UID \rightarrow An

The created document is typed by a document class name (dcn:DCn) which, like the name of the licensee, can also be observed from the unique document identifier). The created document is possibly based on⁹ one or more identified documents (over which the licensee (at least) has reading rights). We can presently omit consideration of the document class concept. The “based on” documents are moved¹⁰ from licensor to licensee.

(9.) Editing a document may be based on “inspiration” from, that is, with reference to a number of other documents (over which the licensee (at least) has reading rights). What this “be based on” means is simply that the edited document contains those references. (They can therefore be traced.) The “based on” documents are moved¹¹ from licensor to licensee — if not already so moved as the result of the specification of other authorised actions.

(10.) Reading a document only changes its “having been read” status (etc.) — as per Appendix G [86]. The read document, if not the result of a copy, is moved from licensor to licensee — if not already so moved as the result of the specification of other authorised actions.

(11.) Copying a document increases the document population by exactly one document. All previously existing documents remain unchanged except that the document which served as a master for the copy has been so marked. The copied document is like the master document except that the copied document is marked to be a copy (etc.) — as per Appendix G [86]. The master document, if not the result of a create or copy, is moved from licensor to licensee — if not already so moved as the result of the specification of other authorised actions.

(12.) A licensee can sub-license certain operations to be performed by other actors.

⁹“Based on” means that the initially void document contains references to those (zero, one or more) documents. They can therefore be traced (etc.) — as per [86].

¹⁰A discussion on this choice, of “move”, should weigh this against licensee be able to remotely access the “based on” document, etc., etc.

¹¹See Footnote 9.

(13.) The granting, extending, restricting or withdrawing permissions (a) cannot name a license (the user has to do that); (b) do not need to refer to the licensor (the licensee issuing the sub-license); and (c) leaves it open to the licensor to freely choose a licensee. One could, instead, for example, constrain the licensor to choose from a certain class of actors. The licensor (the licensee issuing the sub-license) must choose a unique license name.

(14.) A document can be shared between two or more actors. One of these is the licensee, the others are implicitly given read authorisations. (One could think of extending, instead, the licensing actions with a **shared** attribute.) The shared document, if not the result of a create and edit or copy, is moved from licensor to licensee — if not already so moved as the result of the specification of other authorised actions. Sharing a document does not move nor copy it.

(15.) Sharing documents can be revoked. That is, the reading rights are removed.

(16.) The release operation: if a licensor has authorised a licensee to create a document (and that document, when created got the unique document identifier `udi:UDI`) then that licensee can release the created, and possibly edited document (by that identification) to the licensor, say, for comments. The licensor thus obtains the master copy.

(17.) The return operation: if a licensor has authorised a licensee to create a document (and that document, when created got the unique document identifier `udi:UDI`) then that licensee can return the created, and possibly edited document (by that identification) to the licensor — “for good”! The licensee relinquishes all control over that document.

(18.) One or more documents can be subjected to any one of a set of permitted calculation functions. These documents, if not the result of creates and edits or copies, are moved from licensor to licensee — if not already so moved as the result of the specification of other authorised actions. Observe that there can be many calculation permissions, over overlapping documents and functions.

(19.) A document can be shredded.

The Syntax: Actions

type

20. $\text{Action} = \text{Ln} \times \text{Clause}$

21. $\text{Clause} = \text{Cre} \mid \text{Edt} \mid \text{Rea} \mid \text{Cop} \mid \text{Lic} \mid \text{Sha} \mid \text{Rvk} \mid \text{Rel} \mid \text{Ret} \mid \text{Cal} \mid \text{Shr}$

22. $\text{Cre} == \text{mkCre}(\text{dcn:DCn}, \text{based_on_docs:UID-set})$

23. $\text{Edt} == \text{mkEdt}(\text{uid:UID}, \text{based_on_docs:UID-set})$

24. $\text{Rea} == \text{mkRea}(\text{uid:UID})$

25. $\text{Cop} == \text{mkCop}(\text{uid:UID})$

26. $\text{Lic} == \text{mkLic}(\text{license:L})$

27. $\text{Sha} == \text{mkSha}(\text{uid:UID}, \text{with:An-set})$

28. $\text{Rvk} == \text{mkRvk}(\text{uid:UID}, \text{from:An-set})$

- 29. Rel == mkRel(dn:Dn,uid:UID)
- 30. Ret == mkRet(dn:Dn,uid:UID)
- 31. Cal == mkCal(fct:Cfn,over_docs:UID-set)
- 32. Shr == mkShr(uid:UID)

Preliminary Remarks

A clause elaborates to a state change and usually some value. The value yielded by elaboration of the above create, copy, and calculation clauses are unique document identifiers. These are chosen by the “system”.

Syntax & Informal Semantics Annotations: Actions

(20.) Actions are tagged by the name of the license with respect to which their authorisation and document names has to be checked. No action can be performed by a licensee unless it is so authorised by the named license, both as concerns the operation (create, edit, read, copy, license, share, revoke, calculate and shred) and the documents actually named in the action. They must have been mentioned in the license, or, created or copies of downloaded (and possibly edited) documents or copies of these — in which cases operations are inherited.

(22.) A licensee may create documents if so licensed — and obtains all operation authorisations to this document.

(23.) A licensee may edit “downloaded” (edited and/or copied) or created documents.

(24.) A licensee may read “downloaded” (edited and/or copied) or created and edited documents.

(25.) A licensee may (conditionally) copy “downloaded” (edited and/or copied) or created and edited documents. The licensee decides which name to give the new document, i.e., the copy. All rights of the master are inherited to the copy.

(26.) A licensee may issue licenses of the kind permitted. The licensee decides whether to do so or not. The licensee decides to whom, over which, if any, documents, and for which operations. The licensee looks after a proper ordering of licensing commands: first grant, then sequences of zero, one or more either extensions or restrictions, and finally, perhaps, a withdrawal.

(27.) A “downloaded” (possibly edited or copied) document may (conditionally) be shared with one or more other actors. Sharing, in a digital world, for example, means that any edits done after the opening of the sharing session, can be read by all so-granted other actors.

(28.) Sharing may (conditionally) be revoked, partially or fully, that is, wrt. original “sharers”.

(29.) A document may be released. It means that the licensor who originally requested a document (named dn:Dn) to be created is now being able to

see the results — and is expected to comment on this document and eventually to re-license the licensee to further work.

(30.) A document may be returned. It means that the licensor who originally requested a document (named `dn:Dn`) to be created is now given back the full control over this document. The licensee will no longer operate on it.

(31.) A license may (conditionally) apply any of a licensed set of calculation functions to “downloaded” (edited, copied, etc.) documents, or can (unconditionally) apply any of a licensed set of calculation functions to created (etc.) documents. The result of a calculation is a document. The licensee obtains all operation authorisations to this document (— as for created documents).

(32.) A license may (conditionally) shred a “downloaded” (etc.) document.

H.4.4 Discussion

Comparisons

We have “designed”, or rather proposed three different kinds of license languages. In which ways are they “similar”, and in which ways are they “different”? Proper answers to these questions can only be given after we have formalised these languages. The comparisons can be properly founded on comparing the semantic values of these languages.

But before we embark on such formalisations we need some informal comparisons so as to guide our formalisations. So we shall attempt such analysis now with the understanding that it is only a temporary one.

Differences

Work Items: The work items of the artistic license language(s) are essentially “kept” by the licensor. The work items of the hospital health care license language(s) are fixed and, for a large set of licenses there is one work item, the patient which is shared between many licensors and licenses. The work items of the public administration license language(s) — namely document — are distributed to or created and copied by licenses and may possibly be shared.

Operations: The operations of the artistic license language(s) are essentially “kept” by the licensor. The operations of the hospital health care license language(s) are essentially “kept” by the licensees (as reflected in their professional training and conduct). The operations of the public administration license language(s) are essentially “kept” by the licensees (as reflected in their professional training and conduct).

Permissions and Obligations: Generally we can say that the modalities of the artistic license language(s) are essentially permissions with **payment** (as well as use of licensor functions) being an obligation; that the modalities of the hospital health care license language(s) are essentially obligations to

maintain professional standards and the Hippocratic oath; and, as well, that the modalities of the public administration license language(s) are essentially obligations to maintain professional and legal standards. We may have more to say about permissions and obligations later.

H.5 Formal Semantics

By formal semantics we understand a definition expressed in a formal language, that is, a language with a mathematical syntax, a mathematical semantics, and a consistent and relative complete proof system. We shall initially deploy the CSP [189,190,266,273] Specification Language embedded in a Landin-like notation of **let** clauses¹². This embedding is expressed, as were the syntaxes, in the RAISE Specification Language, RSL [87–89,97,165,166,168].

H.5.1 A Model of Common Aspects

Actors: Behaviours and Processes

We see the system as a set of actors. An actor is either a licensor, or a licensee, or, usually, such as we have envisaged our license languages, both. To each actor we associate a behaviour — and we model actor behaviours as CSP processes. So the system is then modelled as a set of concurrent behaviours, that is, parallel (\parallel) CSP processes. Actors are uniquely identified (Aid).

System States

With each actor behaviour we associate a state ($\omega:\Omega$). “Globally” initial such state components are modelled as maps from actor identifiers to states. We shall later analyse these states into major components.

type

Aid, Ω
 $\Omega s = \text{Aid} \xrightarrow{\text{map}} \Omega$

System Processes

Actor processes communicate with one another over channels. There is a set of actor identifier indexed channels. The channels carry actor name decorated messages (Aid \times M). Potential licensees request licenses. Licensors issue licenses in response to requests. Work items are communicated over these channels. As are payments and reports on use of licensed operations on licensed work items. So there is a large variety of messages. An actor is either proactive, requesting licenses, sending payment or reports, or re-active: responding to license requests, sending work items. An actor non-deterministically (\parallel) alternates between these activities.

¹²— known since the very early 1960’s

```

type
  M = Lic | Pay | Rpt | ...
channel
  {a[i]|i:Aid} (Aid×M)
value
  actor: j:Aid ×  $\Omega \rightarrow \mathbf{in, out} \{a[i]|i:Aid \bullet i \neq j\} \mathbf{Unit}$ 
  actor(j)( $\omega$ )  $\equiv \mathbf{let} \ \omega' = \mathbf{pro\_act}(j)(\omega) \parallel \mathbf{re\_act}(j)(\omega) \mathbf{in} \ \mathbf{actor}(j)(\omega') \mathbf{end}$ 

  system:  $\Omega_s \rightarrow \mathbf{Unit}$ 
  system( $\omega_s$ )  $\equiv \parallel \{ \mathbf{actor}(i)(\omega_s(i)) | i:Aid \}$ 

```

Actor Processes

We have identified two kinds of actor processes: (a) pro-active, during which the actor, by own initiative, (as a prospective licensee) may request a license from a prospective licensor, or, as an actual licensee, and as the result of performing licensed actions, sends payments or reports (or other) to the licensor; and (b) re-active, during which the actor, in response to requests (as a licensor) sends a requesting actor a license (whereby the requester becomes a licensee), or “downloads” (access to) requested works or functions. functions.

The Pro-active Actor Behaviour: In the pro-active behaviour an actor, (1.), at will, i.e., (2.) non-deterministically internal choice (\parallel), decides to either request a license (rl) or to perform some action (op). In the former case the actor inquires (4., l_{iq}) an own state to find out from which licensor (k), and which kind of license requirements (l_{rq}) is to be requested. This licensor and these requirements are duly noted in the state (ω'). After (5.) sending the request the actor continues being an actor in the duly noted state (ω'). In the latter case (6., op) there may be many “next” actions to do. The actor inquires (a_{iq}) an own state (ω) to find out which action (op_i) is “next”. The actor then (7.) performs (act) the designated operation. It is here, for simplification assumed that all operation completions imply a “completion” message (m: a payment, a report, or other) to the operation licensing actor (k). So such a message is sent (8.) and the operation-updated local state (ω') is yielded — whereby the pro-active actor “resumes” being an actor as from that state.

```

type
  M = Lic | Pay | Rpt | ...
channel
  {a[i]|i:Aid} (Aid×M)
value
  pro_act: j:Aid  $\rightarrow \Omega \rightarrow \mathbf{in, out} \{a[i]|i:Aid \bullet j \neq i\} \ \Omega$ 
  1. pro_act(j)( $\omega$ )  $\equiv$ 
  2.   let what_to_do = rl  $\parallel$  op in
  3.   case what_to_do of

```



```

4.    rl → let (k,lrq,ω')=iq_l_Ω(ω) in
5.        a(k)!(j,lrq);ω' end
6.    op → let op_i=iq_a_Ω(ω) in
7.        let (k,m,ω')=act(op_i)(ω) in
8.        a(k)!(j,m) ; ω' end end
    end end

```

The Re-active Actor Behaviour: In the re-active behaviour an actor (9., j), is willing to engage in communication with other actors. This is formalised by a non-deterministic external choice (10., \square) between either of a set ($\{\dots\}$) of (zero, or more) other actors ($k:Aid \setminus \{j\}$) who are trying to contact the re-active actor. The communicated message (k,m) reveals the identity (k) of the requesting, i.e., the pro-active actor,¹³ The message, m, reveals what action ($act(m)(\omega)$) the re-active actor is requested to perform. The actor does so (11.). This results in a reply message m' and a state change (ω'). The reply message ($a(k)!(j,m')$) is sent (12.) to the requesting actor (k); and the re-active actor (j) yields the requested action-updated state (ω') — whereby the re-active actor “resumes” being an actor as from that state.

type

$M = Lic \mid Pay \mid Rpt \mid \dots$

channel

$\{a[i] \mid i:Aid\} (Aid \times M)$

value

$re_act: j:Aid \rightarrow \Omega \rightarrow in, out \{a[i] \mid i:Aid \bullet j \neq i\} \Omega$

```

9.    re_act(j)(ω) ≡
10.   let (k,m) =  $\square \{a(k)? \mid k:Aid\}$  in
11.   let (m',ω') = act(m)(ω) in
12.   a(k)!(j,m');ω' end end

```

Functions

We first list (and “read”) the signatures of the two auxiliary ($iq_l_Ω$, $iq_a_Ω$) and one elaboration (act) function assumed in the definition of the pro- and re-active actor processes. After that we discuss the former and suggest definitions of the latter.

Auxiliary Function Signatures

The inquire license function ($iq_l_Ω$) inspects the actor’s state to (“eureka”) determine which most desirable licenser ($k:Aid$) offers which one kind of desired license requirements (License_Requirements). The inquire action function ($iq_a_Ω$) inspects the actor’s state to (somewhat “eureka”) determine

¹³Do not get confused by the two k’s on either side of the let clause. The left k is yielded by the (input) communication $a(k)?$. The defining scope of the right side k, as in $a(k)$, is just the right-hand side of the left clause.

which action is “next” to be performed. That action is being designated (Action_Designator).

```

type
  License_Requirements, Action_Designation
value
  iq_l_Ω: Ω → Aid × License_Requirements × Ω
  iq_a_Ω: Ω → Action_Designator

```

By ‘eureka’¹⁴ is meant that the inquiry is internal non-deterministic, that is, is not influenced by an outside, could have any one of very many outcomes, and can thus only be rather loosely defined.

Elaboration Function Signature

The action performing function (act) “finds” the designated operation in the current state, applies it in the current state, and yields (“read” backwards) a possibly new state ($\omega : \Omega$), a message ($m:M$) to be sent to the licensor ($k:Aid$) who authorised the operation and may need or which to have a payment, a report, or some such thing “back”!

```

type
  Action_Designation
value
  act: Action_Designation → Ω → Aid × M × Ω

```

Discussion of Auxiliary Functions

The auxiliary functions are usually not computable functions. The actors are not robots. And it is not necessary to further define these functions beyond stating their signatures as they are usually performed by human actors. The signature of the inquire license function expresses a possible change to the inquired state. One would think of the inquiring actor somehow noting down, remembering, as it were, which inquiries were attempted or had been made. The signature of the inquire actions function does not express such a state change. But it could be expressed as well.

Schema Definitions of Elaboration Functions

The narrative and formalisation of the schema definitions of elaboration functions is left as an Exercise.

¹⁴ “Eureka” used to express triumph on a discovery, heuristics

H.5.2 A Model of The General Artistic License Language**A Licensor/Licensee State**

The narrative and formalisation of licensor and licensee states is left as an Exercise.

Auxiliary Functions

The narrative and formalisation of the auxiliary functions is left as an Exercise.

Elaboration Functions

The narrative and formalisation of the elaboration functions is left as an Exercise.

H.5.3 A Model of The Hospital Health Care License Language**A Patient [Medical Record] State**

The narrative and formalisation of patient states is left as an Exercise.

A Medical Staff State

The narrative and formalisation of staff states is left as an Exercise.

Auxiliary Functions

The narrative and formalisation of auxiliary functions is left as an Exercise.

Elaboration Functions

The narrative and formalisation of elaboration functions is left as an Exercise.

H.5.4 A Model of The Public Administration License Language**A Public Administrator State**

The narrative and formalisation of public administrator states is left as an Exercise.

Auxiliary Functions

The narrative and formalisation of auxiliary functions is left as an Exercise.

Elaboration Functions

The narrative and formalisation of elaboration functions is left as an Exercise.

H.5.5 Discussion

Since we have left an interesting part of this appendix “as an exercise” there is little we can do wrt. discussing the implication of the formalisations. An important aspect that is thus being missed is “what these formalisations tell us !” Such an analysis has to wait till someone, we (?), get around to do the chores !

H.6 Conclusion

It is too early — in the development of this report — to conclude!

H.6.1 Summary: What Have We Achieved?

Or rather, at this early, incomplete stage, what do we wish to achieve? In a first round we wish to achieve the following: an understanding of different kinds of license languages; an understanding of obligations and permissions (yet to be “designed” more explicitly into the three languages; a formalisation of both common aspects of the license systems (as a “vastly” distributed set of very many actors acting on even more licenses “competing” for resources, etc.), as well as of each individual language.

H.6.2 Open Issues**An Open Issue: Rôle of Modal Logics**

Temporal logic Deontic logic (permission and obligation) Logic of knowledge and belief etc.

Another Open Issue: Fair Use

Fair use: ... etc.

H.6.3 Acknowledgements

The second author wishes to express that it has been an interesting experience to work with his three co-author JAIST “students”: Mr. Yasuhito Arimoto, MSc student, Miss Chen Xiaoyi, PhD student, and Dr. Xiang Jianwen, Post-doc.

H.7 The Gunter et al. Model — And its Reformulation

The unframed, mostly itemized text represents a transcription into L^AT_EX of [172]. The transcription was done by Mr. Arimoto Yasuhito at JAIST. The framed text was first introduced by Dr. Xiang Jianwen at JAIST and was based on group “discoveries” of “bugs” in the Gunter et al. text. The RSL formalisation was done by Dines Bjørner.

H.7.1 Digital Rights Licensing

What is Digital Rights?

- Digital rights deal with the rights of owners of artistic expressions
 - ★ music, movies, readings, photographs, paintings, ...
- in the context of the downloading of these
- via the Internet to users
- who are then supposed to pay for the right to render,
- i.e., to listen, see, hear, and see–see, ..., these.

Realities by Users and Licenses Issued by Owners

- Users perform **payment** and **rendering** events.
- Sequences of events as performed by users make up realities.
- Owners issue licenses which describe which realities are permissible.

Digital Rights Management (DRM)

- DRM is then about
 - ★ the design of appropriate license languages
 - ★ and the enforcement of user realities
 - ★ in order for these to not breach, but to fulfil the licenses.

Structure of Presentation

- First a loyal, but problematic transcription of a published paper.
- Then a “similar”, but believed correct reformulation (in RSL).

H.7.2 Transcript of the Gunter/Weeks/Wright Paper

Actions, Events, Realities and Licenses

- The model centers around
 - ★ a domain of *realities* and
 - ★ a domain of *licenses*,
- where
 - ★ A *reality* is a sequence of events.
 - ★ A *license* is a set of realities.
- The semantics of a rights management languages can be expressed as a function
- that maps terms of the language to elements of their domain of licenses.
- Their abstract model
 - ★ represents an *event*, $e \in Event$, as a pair of a *time*, $t \in Time$,
 - ★ and an *action*, $a \in Action$:

$$e ::= t : a$$

- *Time* is totally ordered by $<$.
- The function $+$:
 - ★ $Time \times Period \rightarrow Time$
 - ★ adds a period, $p \in Period$, to a time.
- There are two kinds of actions:

$$a ::= \text{render}[w, d] \mid \text{pay}[x]$$

- $w \in Work$ denotes a rights-managed work.
- $d \in Device$ represents a DRM-enabled device.
- x is a decimal.
- Action $\text{render}[w, d]$ represents rendering of work w by rights-enabled device d .
- Action $\text{pay}[x]$ represents a payment of amount x of some currency from a licensee to a license issuer.
- The event $t : \text{render}[w, d]$ means that at time t , work w was rendered on device d .
- The event $t : \text{pay}[x]$ means that at time t , a payment of amount x was made.
- A *reality*, $r \in Reality$,
 - ★ is a finite set of events,
 - ★ such that all events occur at distinct times:

$$Reality = \{ E \mid E: \text{Event-set} \bullet t:a \in E \wedge t:a' \in E \Rightarrow a=a' \}$$

where

- ★ $P(x)$ represents the power set of E (the set of all subsets of E).
- ★ Notation:

- $r_{\leq t}$ represents the prefix of r that occurs at or before time t ;
- that is: $r_{\leq t} = \{t' : a \in r \mid t' \leq t\}$.
- ★ Notation:
 - $r \sqsubseteq r'$ indicates that r is a prefix of r' ;
 - that is, there exists a t such that $r = r'_{\leq t}$.

- In the model, a *license*, $l \in License$, is a set of realities:

$$l \in License = P(Reality)$$

- Let us take an example:

$$l_A = \left\{ \begin{array}{l} \{8:00:pay[\$1], 8:01:render[w_1, d_1]\}, \\ \{8:00:pay[\$1], 8:02:render[w_1, d_1]\}, \\ \{8:00:pay[\$1], 8:03:render[w_1, d_1], \\ \quad 8:04:render[w_1, d_1]\}, \\ \{8:00:pay[\$1]\} \end{array} \right\}$$

- To give a more formal meaning to a license,
- suppose \mathbf{r} is the (unique) complete reality
- that actually occurs over the entire lifetime of the DRM system.
- Let $\mathbf{r}[l]$ be events of \mathbf{r} attributed to license l .

- **Definitions**

- ★ Reality $r \in l$ of license l is *viable* for $\mathbf{r}[l]$ at t iff $\mathbf{r}[l]_{\leq t} \sqsubseteq r$.
- ★ License l is *fulfilled* by $\mathbf{r}[l]$ at t iff $\mathbf{r}[l]_{\leq t} \in l$.
- ★ License l is *breached* by $\mathbf{r}[l]$ at t iff there does not exist $r \in l$ that is viable for $\mathbf{r}[l]$ at t .

- **Example 1:**

$$\mathbf{r}[l_A] = \{ 8:00:pay[\$1], 8:01:render[w_1, d_1], 8:05:render[w_1, d_1] \}.$$

For $\mathbf{r}[l_A]$,

- ★ at $t < 8:01$, all four realities of license l_A are viable
- ★ at t , $8:01 \leq t < 8:05$, only the first reality is viable
- ★ at $t \geq 8:05$, no reality is viable

- **Example 2:**

$$\mathbf{r}[l_A] = \{ 8:00:pay[\$1], 8:01:render[w_1, d_1], 8:05:render[w_1, d_1] \}.$$

License l_A is

- ★ unfulfilled by $\mathbf{r}[l_A]$ for $t < 8:00$
- ★ fulfilled for $8:00 \leq t < 8:05$
- ★ breached for $t \geq 8:05$

- **Example 3:**

$$\mathbf{r}'[l_A] = \{ 8:00:pay[\$1], 8:03:render[w_1, d_1] \}.$$

license l_A is

- ★ unfulfilled for $t < 8:00$

- ★ fulfilled for $8:03 \leq t < 8:03$
- ★ unfulfilled for $8:03 \leq t < 8:04$
- ★ breached for $t \geq 8:04$

H.7.3 Standard Licenses

Up Front Licenses

- The “UP Front” license provides access
- to any work in set $W \in P(Work)$
- on any device in set $D \in P(Device)$
- beginning at time t_0 for period p ,
- for an up-front payment of x :

$$\begin{aligned} \text{UpFront}(t_0, x, p, W, D) = & \\ & \{ t_0 : \text{pay}[x], \\ & \quad t_1 : \text{render}[w_1, d_1], \dots, t_n : \text{render}[w_n, d_n] \\ & | \ n \geq 0, \\ & \quad t_0 < t_1 < \dots < t_n < t_0 + p, \\ & \quad w_1, \dots, w_n \in W, d_1, \dots, d_n \in D \} \end{aligned}$$

Problems

- The use of n is confusing.
- On one hand it is used to express up to n renderings.
- On the other hand subscript n is used for ranging both works and devices.
- The three uses of n should be separated into, say, i, j and k .
- The same comments apply to the Flat Rate and Per Use formulations.
- Why not allow $t_n \leq t_0 + p$?

Flat Rate Licenses

- The “Flat Rate” license provides access
- to any work in set W
- on any device in set D
- beginning at time t_0 for period p ,
- for a payment of x at the end of the period:

$$\begin{aligned} \text{FlatRate}(t_0, x, p, W, D) = & \\ & \{ t_1 : \text{render}[w_1, d_1], \dots, t_n : \text{render}[w_n, d_n] \\ & \quad t_{n+1} : \text{pay}[x], \\ & | \ n \geq 0, \\ & \quad t_0 < t_1 < \dots < t_n < t_{n+1} < t_0 + p, \\ & \quad w_1, \dots, w_n \in W, d_1, \dots, d_n \in D \} \end{aligned}$$

Problems

- Why not allow $t_n \leq t_0 + p$?

Per Use Licenses

- The “Per Use” license is provides access
- to any work in set W
- on any device in set D
- beginning at time t_0
- for a period of length p ,
- for a payment of x per use at the end of the period:

$$\begin{aligned} \text{PerUse}(t_0, x, p, W, D) = & \\ & \{t_1 : \text{render}[w_1, d_1], \dots, t_n : \text{render}[w_n, d_n] \\ & \quad t_{n+1} : \text{pay}[nx], \\ & \mid n \geq 0, \\ & \quad t_0 < t_1 < \dots < t_n < t_{n+1} < t_0 + p, \\ & \quad w_1, \dots, w_n \in W, d_1, \dots, d_n \in D \} \end{aligned}$$

Problems

- Why not allow $t_{n+1} \leq t_0 + p$?

Up to Expiry Date Licenses

- A license that a consumer can accept any time before some future
 - ★ can be constructed by quantifying over t_0
 - ★ for any of the three families defined on preceding slides.
- For example,

$$\bigcup_{t_0 < t_{\text{expires}}} \text{UpFront}(t_0, x, p, W, D)$$

- This license allows the period of the use to begin anytime prior to t_{expires} .

Problems

- Why not allow $t_0 \leq t_{\text{expires}}$?

Non-cancelable Multi-use Licenses

- To construct multi-period non-cancelable licenses
- by composing single-period licenses,
- an operator Δ is defined.

$$l_1 \Delta l_2 = \{r_1 \cup r_2 \mid r_1 \in l_1, r_2 \in l_2\}$$

- here all of the events of l_1 occur at different times from the events of l_2 .

Problems

- The above is not the same as: $l_1 \Delta l_2 = l_1 \cup l_2$
- How does Δ associate?
- We guess: $l \Delta l' \Delta l'' = l \Delta (l' \Delta l'')$

- $\text{UpFront}^\Delta(t_0, x, p, W, D, m)$
- provides access to any work in set W
- on any device in set D
- beginning at time t_0
- for m periods of length p ,
- for payments of x at the beginning of each period:

$$\begin{aligned} \text{UpFront}^\Delta(t_0, x, p, W, D, m) = & \\ & \text{UpFront}(t_0, x, p, W, D) \Delta \dots \Delta \\ & \text{UpFront}(t_{m-1}, x, p, W, D) \\ & \text{where } t_i = t_0 + ip \text{ for } i \text{ from } 0 \text{ to } m-1 \end{aligned}$$

Cancelable Multi-use Licenses

- With the \triangleright operator cancelable multi-period licenses can be defined.

$$l_1 \triangleright l_2 = \{r_1 \cup r_2 \mid r_1 \in l_1, r_1 \neq \emptyset, r_2 \in l_2\} \cup \{\emptyset\}$$

- here all of the events of l_1 occur at different times from the events of l_2 .

Problems

- Same question concerning associativity.
- What, exactly is the rôle of the empty set $\{\}$, or, rather, $\{\{\}\}$ (in Gunter et al. paper: $\{\emptyset\}$).
- We guess: To have only actions from l_1 and then “get out”!
- Also: There is a problem with t_i : it is defined but never used!

- $\text{UpFront}^\triangleright(t_0, x, p, W, D, m)$

- provides access to any work in set W
- on any device in set D
- beginning at time t_0
- for m periods of length p ,
- for payments of x at the beginning of each period,
- cancelable after any period:

$$\begin{aligned} \text{UpFront}^\triangleright(t_0, x, p, W, D, m) = \\ \text{UpFront}(t_0, x, p, W, D) \triangleright \dots \triangleright \\ \text{UpFront}(t_{m-1}, x, p, W, D) \\ \text{where } t_i = t_0 + ip \text{ for } i \text{ from } 0 \text{ to } m-1 \end{aligned}$$

Problems

- Same question concerning associativity.
- There is still a problem with t_i : it is defined but never used!

H.7.4 A License Language

Syntax

The language *DigitalRights* is defined by the following grammar:

$$\begin{aligned} e ::= & (\text{at } t \mid \text{until } t) \\ & (\text{for } p \mid \text{for } [\text{up to } m] p) \\ & \text{pay } x \text{ (upfront} \mid \text{flatrate} \mid \text{peruse)} \\ & \text{for } W \text{ on } D \end{aligned}$$

Examples

until 01/01/03
for up to 12 months
pay \$ 10.00 upfront
for "Jazz Classics"
on "devices registered to license holder"

Semantics

$$\begin{aligned} \mathcal{M}[\text{at } t \ z] &= \mathcal{M}_1[z](t) \\ \mathcal{M}[\text{until } t \ z] &= \bigcup_{t' < t} \mathcal{M}_1[z](t') \\ \mathcal{M}_1[\text{for } p \ z](t_0, p) &= \mathcal{M}_2[z](t_0, p) \\ \mathcal{M}_1[\text{for up to } m \ p \ z](t_0) &= \mathcal{M}_2[z](t_0, p) \triangleright \dots \triangleright \mathcal{M}_2[z](t_{m-1}, p) \\ &\quad \text{where } t_i = t_0 + ip \text{ for } i \text{ from } 0 \text{ to } m-1 \\ \mathcal{M}_1[\text{for } m \ p \ z](t_0) &= \mathcal{M}_2[z](t_0, p) \triangle \dots \triangle \mathcal{M}_2[z](t_{m-1}, p) \\ &\quad \text{where } t_i = t_0 + ip \text{ for } i \text{ from } 0 \text{ to } m-1 \end{aligned}$$

$$\begin{aligned}\mathcal{M}_2[\text{pay } x \text{ upfront for } W \text{ on } D](t, p) &= \text{UpFront}(t, x, p, W, D) \\ \mathcal{M}_2[\text{pay } x \text{ flatrate for } W \text{ on } D](t, p) &= \text{FlatRate}(t, x, p, W, D) \\ \mathcal{M}_2[\text{pay } x \text{ peruse for } W \text{ on } D](t, p) &= \text{PerUse}(t, x, p, W, D)\end{aligned}$$

Problems

- Parameters (t_0, p) in first left hand side of \mathcal{M}_1 is wrong,
- should be just (t_0) .

H.7.5 An RSL Model

Actions, Events, Realities and Licences

type

T, W, D, P
 $E = T \times A$
 $A == \text{mkR}(w:W, d:D) \mid \text{mkP}(x:P) \text{ or: } \mathbf{rندر}(w, d) \mid \mathbf{pay}(x)$
 $R = \{ \mid \text{evs}:E\text{-set} \bullet \text{wfEvs}(\text{evs}) \mid \}$

Annotations

- T, W, D, and P stands for time, work, device and payment.
- Events, $e:E$, are Cartesian pairs of times (i.e., time stamps) and actions.
- Actions are discriminated, either renderings (which are records $\text{mkR}(w, d)$) of works and devices. or payments ($\text{mkP}(x)$) of currency amounts.
- The trailing “or” shows our schematic way of representing actions.
- Realities, $r:R$, are well-formed sets of events.

value

$\text{wfEvs}: E\text{-set} \rightarrow \mathbf{Bool}$
 $\text{wfEvs}(\text{evs}) \equiv \forall (t, a), (t', a'): E \bullet \{(t, a), (t', a')\} \subseteq \text{evs} \wedge t=t' \Rightarrow a=a'$

$r_{\leq t}$: prefix: $R \rightarrow T \rightarrow R$
 $r_{\leq t} \equiv \text{prefix}(r)(t) \equiv \{(t', a) \mid (t', a): E \bullet (t', a) \in r \wedge t' \leq t\}$

$r' \sqsubseteq r$: is-prefix: $R \times R \rightarrow \mathbf{Bool}$
 $r' \sqsubseteq r \equiv \text{is_prefix}(r', r) \equiv \exists t: T \bullet r' = \text{prefix}(r)(t)$

Annotations

- A set of events form a reality if no two events have the same time stamp.
- The prefix of a reality up to and including time t is the set of all those events of the reality whose time stamp is equal to or less than t .
- A reality is a prefix of another event if there is a time t such that the former reality is a prefix of the latter reality.

```

type
  L = R-set

value
  /* viable: capable of working */
  is_viable: R → R → L → T → Bool
  is_viable(r')(r)(l)(t) ≡ r ∈ l ∧ is_prefix(prefix(r')(t),r)

  /* fulfilled: a consumer reality r' satisfies a license reality */
  is_fulfilled: R → L → T → Bool
  is_fulfilled(r')(l)(t) ≡ prefix(r')(t) ∈ l

  is_breached: R → L → T → Bool
  is_breached(r')(l)(t) ≡ ∼∃ r:R • r ∈ l ∧ is_viable(r')(r)(l)(t)

```

Annotations

- A reality r' is viable at a time t wrt. a reality r of a license l if r' is a prefix, at that time, of r .
- A reality r' is, at time t fulfilled wrt. a license l if the prefix of r' at time t is a reality of the license.
- License l is breached by consumer reality r' if there is no reality r in l that is viable for r' at t .

H.7.6 Standard Licences

Syntax

```

type
  I
  Std_L = UpFront|FlatRate|PerUse|Until|NonCanMuUpFro|CanMuUpFro
  Basics = T×P×I×W-set×D-set
  UpFront == mkUF(sb:Basics)
  FlatRate == mkFR(sb:Basics)
  PerUse == mkPU(sb:Basics)
  Until == mkU(sb:Basics,st:T)
  NonCanMuUpFro == mkNCMF(sb:Basics,sm:Nat)
  CanMuUpFro == mkCMF(sb:Basics,sm:Nat)

```

Annotations

- I stands for an interval, i.e., a time period.
- Std_L stands for standard licenses.
- There are six forms of standard licences: UpFront, FlatRate, PerUse, Until, NonCanMuUpFro and CanMuUpFro.

- They all share some basic information Basics, a Time limit, Payment amount, Interval, identification of the set of Works covered by the license, and identification of the set of Devices covered by the license.
- The `NonCanMuUpFro` and `CanMuUpFro` commands further specify a natural, usually non-zero number (of times of rendering).
- The `Std_` definition defines the set of commands as a union using the `|` type constructor.
- Each individual type is then defined by distinctly named record type constructors: `mkUF`, `mkFR`, `mkPU`, `mkU`, `mkNCMF` and `mkCMF`.
- We have for ease of recalling these mnemonics chosen to name the constructors with an initial `mk` (for ‘make’) and then an abbreviation of the type name being defined.
- The `s...` parts of the body of the record type expressions designate selector functions.
- Meta-linguistically:

type

A, B, ..., C

R == mkR(sa:A, sb:B, ..., sc:C)

axiom

$\forall r:R, a:A, b:B, \dots, c:C \bullet$
 $r = \text{mkR}(\text{sa}(r), \text{sb}(r), \dots, \text{sc}(r)) \wedge$
 $a = \text{sa}(\text{mkR}(a, b, \dots, c)) \wedge$
 $b = \text{sb}(\text{mkR}(a, b, \dots, c)) \wedge$
 $\dots \wedge$
 $c = \text{sc}(\text{mkR}(a, b, \dots, c))$

Semantics

value

M: Std_L \rightarrow L

$M(\text{mkUF}(t_0, x, p, \text{ws}, \text{ds})) \equiv$

$\{\{(t_0, \text{pay}(x))\} \cup$
 $\text{let } \text{ls} = [\text{ti} \mapsto \text{rndr}(w, d) \mid \text{ti}:T, w:W, d:D \bullet \text{ti} \in \text{ts} \wedge w \in \text{ws}' \wedge d \in \text{ds}'] \text{ in}$
 $\{(ti, \text{ls}(ti)) \mid ti:T \bullet ti \in \text{dom } \text{ls}\} \text{ end}$
 $\mid n:\text{Nat}, \text{ts}:T\text{-set}, \text{ws}':W\text{-set}, \text{ds}':D\text{-set} \bullet$
 $\text{card } \text{ts} = n \wedge \text{ws}' \subseteq \text{ws} \wedge \text{ds}' \subseteq \text{ds} \wedge t_0 < \text{min}(\text{ts}) \wedge \text{max}(\text{ts}) \leq t_0 + p\}$

Annotations

- The above formulation follows that of Gunter et al.,
- but where their model is plain wrong: it does not designate all combinations of works and devices, ours is right:
 - ★ At t_0 payment is issued;

- ★ the above expression¹⁵ has two set comprehensions:
- ★ $\{\{A\} \cup \{B|C \cdot D\} \mid E \cdot F\}$
- ★ The inner comprehension $\{B|C \cdot D\}$ expresses all possible sequences of n renderings involving any combination of works w and devices d from subsets ws and ds of works and devices.
- ★ The outer comprehension “selects” an arbitrary, finite n , a set ts of n time points all of which lies between t_0 and $t_0 + p$, and arbitrary subsets ws and ds of works and devices of those given,
- ★ The inner comprehension ensures that all we express all runs of renderings of length n over all combinations of works and devices.
- ★ The outer comprehension ensures that we express all possible and indefinite length runs.
- A better model would be one which, instead of constructively designating all possible runs, expresses the property that a run is an up front run and all such, for the given arguments, are present.

type

PayEvent = $T \times (\{|pay|\} \times Nat)$
 RndEvent = $T \times (\{|render|\} \times W \times D)$
 UpFrontLicense = $\{ \mid \ell s : L \bullet wf_UPL(\ell s) \mid \}$

value

wf_UPL: $L \rightarrow Bool$
 wf_UPL(ℓ) $\equiv \exists t:T, x:Nat, p:P, ws:W\text{-set}, ds:D\text{-set} \bullet is_ufl(t_0, x, p)(ws, ds, ts)(\ell)$

is_ufl: $(T \times Nat \times P) \rightarrow (W\text{-set} \times D\text{-set} \times T\text{-set}) \rightarrow L \rightarrow Bool$

is_ufl(t_0, x, p)(ws, ds, ts)(ℓ) \equiv
 $\exists t':T, x':X \bullet (t', (pay, x')) \in \ell \Rightarrow t' = t_0 \wedge x' = x \wedge$
 $\forall ti:T, w:W, d:D \bullet ti \in ts \wedge w \in ws \wedge d \in ds \Rightarrow$
 $(ti, (render, w, d)) \in \ell \wedge$
 $\sim \exists (t, (render, w', d')) : RndEvent \bullet$
 $(t, (render, w', d')) \in \ell \wedge t \notin ts \wedge w' \notin ws \wedge d' \notin ds$

- A set ℓs of licenses
- is the set of all up front licenses
- designated by $M(mkUF(t_0, x, p, ws, ds))$
- if it satisfies $are_all_ufl(t_0, x, p)(ws, ds, ts)(\ell s)$.

value

are_all_ufl: $T \times Nat \times P \times W\text{-set} \times D\text{-set} \rightarrow L\text{-set} \rightarrow Bool$
 are_all_ufl(t_0, x, p)(ws, ds, ts)(ℓs) \equiv
 $\forall \ell:L \bullet \ell \in \ell s \Rightarrow$
 $\exists n:Nat, ws':W\text{-set}, ds':D\text{-set}, ts:T\text{-set} \bullet$

¹⁵ $\{ \{ (t_0, pay(x)) \} \cup \{ (ti, rndr(w, d)) \mid ti:T, w:W, d:D \bullet ti \in ts \wedge w \in ws' \wedge d \in ds' \} \mid$
 $n:Nat, ts:T\text{-set}, ws':W\text{-set}, ds':D\text{-set} \bullet cardts = n \wedge ws' \subseteq ws \wedge ds' \subseteq ds \wedge t_0 < \min(ts)$
 $\wedge \max(ts) \leq t_0 + p \}$

$$ws' \subseteq ws \wedge ds' \subseteq ds \wedge \mathbf{card} \ ts = n \Rightarrow \text{is_ufi}(t0, x, p)(ws', ds', ts)(\ell)$$

Annotations

- The “do not exists” clauses shall ensure both largest sets of up front licenses over appropriate time spans, works, and devices and that there is no “junk”.
- Otherwise we leave it to the reader to decipher the formulas.

value

$$\begin{aligned} M(\text{mkFR}(t, x, p, ws, ds)) \equiv & \\ & \{\mathbf{let} \ ls = [ti \mapsto \mathbf{rndr}(w, d) | ti: T, w: W, d: D \bullet ti \in ts \wedge w \in ws' \wedge d \in ds'] \ \mathbf{in} \\ & \quad \{(ti, ls(ti)) | ti: T \bullet ti \in \mathbf{dom} \ ls\} \ \mathbf{end} \\ & \quad \cup \{(tn', \mathbf{pay}(x))\} \\ & \quad | n: \mathbf{Nat}, tn': T, ts: T\text{-set}, ws': W\text{-set}, ds': D\text{-set} \bullet \\ & \quad \mathbf{card} \ ts = n \wedge ws' \subseteq ws \wedge ds' \subseteq ds \wedge t0 \leq \mathbf{min}(ts) \wedge \mathbf{max}(ts) < tn' \leq t0 + p\} \end{aligned}$$

value

$$\begin{aligned} M(\text{mkPU}(t, x, p, ws, ds)) \equiv & \\ & \{\mathbf{let} \ ls = [ti \mapsto \mathbf{rndr}(w, d) | ti: T, w: W, d: D \bullet ti \in ts \wedge w \in ws' \wedge d \in ds'] \ \mathbf{in} \\ & \quad \{(ti, ls(ti)) | ti: T \bullet ti \in \mathbf{dom} \ ls\} \ \mathbf{end} \\ & \quad \cup \{(tn', \mathbf{pay}(n * x))\} \\ & \quad | n: \mathbf{Nat}, tn': T, ts: T\text{-set}, ws': W\text{-set}, ds': D\text{-set} \bullet \\ & \quad \mathbf{card} \ ts = n \wedge ws' \subseteq ws \wedge ds' \subseteq ds \wedge t0 \leq \mathbf{min}(ts) \wedge \mathbf{max}(ts) < tn' \leq t0 + p\} \end{aligned}$$

value

$$M(\text{mkU}(te, x, p, ws, ds)) \equiv \cup \{M(\text{mkUF}(t0, x, p, ws, ds)) | t0: T \bullet t0 \leq te\}$$

$$M(\text{mkNCMF}((t, x, p, ws, ds), m)) \equiv \text{UpFront}^\Delta((t, x, p, ws, ds), m)$$

$$M(\text{mkCMF}((t, x, p, ws, ds), m)) \equiv \text{UpFront}^\triangleright((t, x, p, ws, ds), m)$$

value

$$\begin{aligned} \Delta: L^* &\rightarrow L \\ \Delta(\mathbf{ll}) &\equiv \\ & \mathbf{case} \ \mathbf{ll} \ \mathbf{of} \\ & \quad \langle l \rangle \rightarrow l, \\ & \quad \langle l \rangle \wedge \mathbf{ll}' \rightarrow \{r \cup r' | r, r': R \bullet r \in l \wedge r' \in \Delta(\mathbf{ll}')\} \\ & \mathbf{end} \end{aligned}$$

$$\begin{aligned} \triangleright: L^* &\rightarrow L \\ \triangleright(\mathbf{ll}) &\equiv \\ & \mathbf{case} \ \mathbf{ll} \ \mathbf{of} \\ & \quad \langle l \rangle \rightarrow l, \\ & \quad \langle l \rangle \wedge \mathbf{ll}' \rightarrow \{r, r \cup r' | r, r': R \bullet r \in l \wedge r \neq \{\} \wedge r' \in \triangleright(\mathbf{ll}')\} \\ & \mathbf{end} \end{aligned}$$

value

$\text{UpFront}^\Delta: \text{Basics} \times \mathbf{Nat} \rightarrow L$
 $\text{UpFront}^\Delta((t, x, p, \text{ws}, \text{ds}), m) \equiv$
 $\Delta(\langle M(\text{mkUF}(ti, x, p, \text{ws}, \text{ds})) | i: \mathbf{Nat}, ti: T \bullet i: \{0..m-1\} \wedge ti = t0 + i \times p \rangle)$

 $\text{UpFront}^\triangleright: \text{Basics} \times \mathbf{Nat} \rightarrow L$
 $\text{UpFront}^\triangleright((t, x, p, \text{ws}, \text{ds}), m) \equiv$
 $\triangleright(\langle M(\text{mkUF}(ti, x, p, \text{ws}, \text{ds})) | i: \mathbf{Nat}, ti: T \bullet i: \{0..m-1\} \wedge ti = t0 + i \times p \rangle)$

H.7.7 A License Language**type**

$\text{DRExpr} = \text{Time} \times \text{Repetition} \times \text{Payment} \times \text{WorksDevices}$
 $\text{Time} == \text{mkAt}(t: T) \mid \text{mkUntil}(t: T)$
 $\text{Repetition} == \text{mkFor}(p: I) \mid \text{mkRepeat}(m: \mathbf{Nat}, p: I) \mid \text{mkUpTo}(m: \mathbf{Nat}, p: I)$
 $\text{Payment} = P \times \text{Kind}$
 $\text{Kind} == \text{upfront} \mid \text{flatrate} \mid \text{peruse}$
 $\text{WorksDevices} = \mathbf{W\text{-set}} \times \mathbf{D\text{-set}}$

value

$M0(\text{mkAt}(t), r, (x, k), \text{wds}) \equiv M1(r, (x, k), \text{wds})(t)$
 $M0(\text{mkUntil}(t), r, (x, k), \text{wds}) \equiv \cup \{M1(r, (x, k), \text{wds})(t') \mid t': T \bullet t' < t\}$

 $M1(\text{mkFor}(p), (x, k), \text{wds})(t) \equiv M2(x, k)(t, p)$
 $M1(\text{mkRepeat}(m, p), (x, k), \text{wds})(t) \equiv$
 $\triangleright(\langle M2((x, k), \text{wds})(ti, p) | i: \mathbf{Nat}, ti: T \bullet i: \{0..m-1\} \wedge ti = t0 + i \times p \rangle)$
 $M1(\text{mkUpTo}(m, p), (x, k), \text{wds})(t) \equiv$
 $\Delta(\langle M2((x, k), \text{wds})(ti, p) | i: \mathbf{Nat}, ti: T \bullet i: \{0..m-1\} \wedge ti = t0 + i \times p \rangle)$

 $M2((x, \text{upfront}), (\text{ws}, \text{ds}))(t, p) \equiv M(\text{mkUF}(t, x, p, \text{ws}, \text{ds}))$
 $M2((x, \text{upfront}), (\text{ws}, \text{ds}))(t, p) \equiv M(\text{mkFR}(t, x, p, \text{ws}, \text{ds}))$
 $M2((x, \text{upfront}), (\text{ws}, \text{ds}))(t, p) \equiv M(\text{mkPU}(t, x, p, \text{ws}, \text{ds}))$

H.7.8 End of “Gunter” Paper

- On one hand:
 - ★ An introduction to the standard view of digital rights licenses.
- On the other hand:
 - ★ An illustration of
 - a pseudo-formal erroneous model
 - a correct formal presentation.

versus

- Now we are ready
 - ★ to study digital rights license languages in general.

I

Management and Organisation¹

This appendix consists of two edited excerpts from a new book: Dines Bjørner: Software Engineering:

- Vol. I: The Triptych Method, Chap. 2, Sect. 2.8.5: *Management and Organisation*; and
- Vol. II: A Model Development, Appendix H: *Management and Organisation*.

(The book Software Engineering (approximately 400 pages) is currently being written.²)

The two edited excerpts are in Appendix

- Sect. I.1,
- respectively Sects. I.2–I.5.

These two parts should basically replace Vol. 3's Sect. 11.5, Pages 276–282.

- Section I.1 presents a methodology of modelling domain management and organisation facets whereas Sects. I.2–I.5 (Pages 392–406) provides “concrete” examples.

We illustrate three aspects of the management and organisation facet. Two rough-sketch models suggest how “layers” of management collaborate and in two different styles and at two different levels of detail: as a simple functional model, Sect. I.2, and as a not quite so simple model based on communicating sequential processes Sect. I.3. The two formal models share the same basic narrative. That narrative is given in Sect. I.2.1. One rough-sketch model illustrates organisational aspects (Sect. I.5).

¹The text of this appendix is very tentative. A final draft version is expected early fall 2008. See next footnote — below!

²The (new) Software Engineering book is expected to find a final form after fall 2008 and spring/summer 2009 lectures at Techn. Univ. of Graz, Politecnico di Milano, University of Saarland, and Christian-Albrechts University of Kiel.

I.1 Management and Organisation

I.1.1 Management

Management is an elusive term. Business schools and private consultancy firms excel in offering degrees and 2–3 day courses in ‘management’. In the mind of your author most of what is being taught — and even researched — is a lot of “hot air”. Well, the problem here, is, of course, that your author was educated at a science & technology university³. In the following we shall repeat some of this ‘hot air’. And after that we shall speculate on how to properly describe the outlined (“cold air”) management concepts.

Characterisation. Domain Management: By *domain management* we mean people (i) who determine, formulate and thus set standards (cf. rules and regulations, a later lecture topic) 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. ■

*Management Issues*⁴

Management in simple terms means the act of getting people together to accomplish desired goals. Management comprises (vi) planning, (vii) organizing, (viii) resourcing, (ix) leading or directing, and (x) controlling an organization (a group of one or more people or entities) or effort for the purpose of accomplishing a goal. Resourcing encompasses the (xi) deployment and manipulation of human resources, (xii) financial resources, (xiii) technological resources, and (xiv) natural resources

*Basic Functions of Management*⁵

These are normally seen as management issues:

Planning: (xv) deciding what needs to happen in the future (today, next week, next month, next year, over the next 5 years, etc.) (xvi) and generating plans for action. **Organizing:** (xvii) making optimum use of the resources (xix) required to enable the successful carrying out of plans. **Leading/Motivating:** (xx) exhibiting skills in these areas (xxi) for getting others to play an effective part in achieving plans. **Controlling:** (xxii) monitoring – (xxiii) checking progress against plans, (xxiv) which may need modification based on feedback.

³— which, alas, now also offers such ‘management’ degree courses !

⁴Ref.: <http://en.wikipedia.org/wiki/Management>

⁵See Footnote 4

*Formation of Business Policy*⁶

(xxvi) The **mission** of a business seems to be its most obvious purpose – which may be, for example, to make soap. (xxvii) The **vision** of a business is seen as reflecting its aspirations and specifies its intended direction or future destination. (xxviii) The **objectives** of a business refers to the ends or activity at which a certain task is aimed⁷. The business **policy** is a guide that stipulates (xix) rules, regulations and objectives, (xxx) and may be used in the managers' decision-making. (xxxi) It must be flexible and easily interpreted and understood by all employees. The business **strategy** refers to (xxxii) the coordinated plan of action that it is going to take, (xxxiii) as well as the resources that it will use, to realize its vision and long-term objectives. (xxxiv) It is a guideline to managers, stipulating how they ought to allocate and utilize the factors of production to the business's advantage. (xxxv) Initially, it could help the managers decide on what type of business they want to form.

*Implementation of Policies and Strategies*⁸

(xxxvi) All policies and strategies are normally discussed with managerial personnel and staff. (xxxvii) Managers usually understand where and how they can implement their policies and strategies. (xxxviii) A plan of action is normally devised for the entire company as well as for each department. (xxxix) Policies and strategies are normally reviewed regularly. (xxxvii) Contingency plans are normally devised in case the environment changes. (xl) Assessments of progress are normally and regularly carried out by top-level managers. (xli) A good environment is seen as required within the business.

*Development of Policies and Strategies*⁹

(xlii) The missions, objectives, strengths and weaknesses of each department or normally analysed to determine their rôles in achieving the business mission. (xlili) Forecasting develops a picture of the business's future environment. (xliv) Planning unit are often created to ensure that all plans are consistent and that policies and strategies are aimed at achieving the same mission and objectives. (xlv) Contingency plans are developed — just in case ! (xli) Policies are normally discussed with all managerial personnel and staff that is required in the execution of any departmental policy.

⁶See Footnote 4 on the preceding page

⁷Pls. note that, in this book, we otherwise make a distinction between aims and objectives: Aims is what we plan to do; objectives are what we expect to happen if we fulfill the aims.

⁸See Footnote 4 on the facing page

⁹See Footnote 4 on the preceding page

Management Levels

A careful analysis has to be made by the domain engineer of how management is structured in the domain being described. One view, but not necessarily the most adequate view for a given domain is that management can be seen as composed from the board of directors (representing owners, private or public, or both), the senior level or strategic (or top, upper or executive) management, the mid level or tactical management, the low level or operational management, and supervisors and team leaders. Other views, other “management theories” may apply. We shall briefly pursue the above view.

Resources

Management is about resources. A resource is any physical or virtual entity of limited availability such as, for example, time and (office, factory, etc.) space, people (staff, consultants, etc.), equipment (tools, machines, computers, etc.), capital (cash, goodwill, stocks, etc.), etcetera.

Resources have to be managed allocated (to [factory, sales, etc.] processes, projects, etc.), and scheduled (to time slots).

Resource Conversion

Resources can be traded for other resources: capital funds can be spent on acquiring space, staff and equipment, services and products can be traded for other such or for monies, etc.

The decisions as to who schedules, allocates and converts resources are made by strategic and tactical management. Operational management transforms abstract, general schedules and allocations into concrete, specific such.

Strategic Management

A strategy is a long term plan of action designed to achieve a particular goal. Strategy is differentiated from tactics or immediate actions with resources at hand by its nature of being extensively premeditated, and often practically rehearsed. Strategies are used to make business problems easier to understand and solve. Strategic management deals with conversion of long term resources involving financial issues and with long term scheduling issues.

Among examples of strategic management issues (in supply chain management) we find:¹⁰ (xlvii) strategic network optimization, including the number, location, and size of warehouses, distribution centers and facilities; (xlviii) strategic partnership with suppliers, distributors, and customers, creating communication channels for critical information and operational improvements such as cross docking, direct shipping, and third-party logistics; (xlix)

¹⁰Cf. http://en.wikipedia.org/wiki/Supply_chain_management#Strategic

product design coordination, so that new and existing products can be optimally integrated into the supply chain, load management; (l) information technology infrastructure, to support supply chain operations; (li) where-to-make and what-to-make-or-buy decisions; and (lii) aligning overall organizational strategy with supply strategy. The problem, in domain modelling, is to find suitable abstractions of these mundane activities.

Strategic management¹¹ (liii) requires knowledge of management rôles and skills; (liv) have to be aware of external factors such as markets; (lv) decisions are generally of a long-term nature; (lvi) decision are made using analytic, directive, conceptual and/or behavioral/participative processes; (lvii) are responsible for strategic decisions; (lviii) have to chalk out the plan and see that plan may be effective in the future; and (lix) is executive in nature.

Tactical Management

Tactical management deals with shorter term issues than strategic management, but longer term issues than operational management. Tactical management deals with allocation and short term scheduling.

Among examples of tactical management issues (in supply chain management) we find:¹² (lx) sourcing contracts and other purchasing decisions; (lxi) production decisions, including contracting, locations, scheduling, and planning process definition; (lxii) inventory decisions, including quantity, location, and quality of inventory; (lxiii) transportation strategy, including frequency, routes, and contracting; (lxiv) benchmarking of all operations against competitors and implementation of best practices throughout the enterprise; (lxv) milestone payments; and (lxvi) focus on customer demand. The problem, in domain modelling, is to find suitable abstractions of these mundane activities.

Operational Management

Operational management deals with day-to-day and week-to-week issues where tactical management deals with month-to-month and quarter-to-quarter issues and strategic management deals with year-to-year and longer term issues. (Operational management is not to be confused with the concept of operational research and operational analysis which deals with optimising resource usage (allocation and scheduling)).

Among examples of operational management issues (in supply chain management) we find:¹³ (lxviii) daily production and distribution planning, including all nodes in the supply chain; (lxix) production scheduling for each manufacturing facility in the supply chain (minute by minute); (lxx) demand planning and forecasting, coordinating the demand forecast of all customers

¹¹See Footnote 4 on page 382

¹²See Footnote 10 on the preceding page.

¹³See Footnote 10 on the facing page.

and sharing the forecast with all suppliers; (lxxi) sourcing planning, including current inventory and forecast demand, in collaboration with all suppliers; (lxxii) inbound operations, including transportation from suppliers and receiving inventory; (lxxiii) production operations, including the consumption of materials and flow of finished goods; (lxxiv) outbound operations, including all fulfillment activities and transportation to customers; (lxxv) order promising, accounting for all constraints in the supply chain, including all suppliers, manufacturing facilities, distribution centers, and other customers. The problem, in domain modelling, is to find suitable abstractions of these mundane activities.

Supervisors and Team Leaders

We make here a distinction between managers, “on one side”, and supervisors and team leaders, “on the other side”. The distinction is based on managers being able to make own decisions without necessarily having to confer or discuss these beforehand or to report these afterwards, while supervisors and team leaders normally are not expected to make own decisions: if they have to make decisions then such are considered to be of “urgency”, must normally be approved of beforehand, or, at the very least, reported on afterwards.

Supervisors basically monitor that work processes are carried out as planned and report other than minor discrepancies. Team leaders coordinate work in a group (“the team”) while participating in that work themselves; additionally they are also supervisors.

Description of ‘Management’

On the last several pages (382–386) we have outlined conventional issues of management.

The problems confronting us now are: Which aspects of domain management are we to describe? How are we describe, especially formally, the chosen issues?

The reason why these two “leading questions” questions are posed is that the management issues mentioned on pages 382–386 are generally “too lofty”, “too woolly”, that is, are more about “feelings” than about “hard, observable facts”.

We, for example, consider the following issues for “too lofty”, “too woolly”: Item (xix) Page 382: “to enable the successful . . .” is problematic; Item (xx) Page 382: how to check that managers “exhibit these skills”?; Item (xxi) Page 382: “play an effective part” is problematic; Item (xxvii) Page 383: how to check that vision is being or is achieved?; Item (xxviii) Page 383: the objectives must, in order to be objectively checked, be spelled out in measurable details; Item (xxxi) Page 383: how to check “flexible” and “easily”; Item (xxxiii) Page 383: how to check that the deployed resources are those that contribute

to “achieving vision and long term objectives; Item (xxxiv) Page 383: “guideline”, “factors of production” and “advantage” cannot really be measured; Item (xxxv) Page 383: “what type of business they want to form” is too indeterminate; Item (xxxvi) Page 383: how to describe (and eventually check) “are normally or must be discussed” other than “just check” without making sure that managerial personnel and staff have really understood the issues and will indeed follow policies and strategies; Item (xxxvii) Page 383: how does one describe “managers must, or usually understand where and how” ?; Item (xxxix) Page 383: in what does a review actually consists ?; Item (xli) Page 383: how does one objectively describe “a good environment” ?; Item (xlii) Page 383: how does one objectively describe that which is being “analysed”, the “analysis” and the “determination” processes ?; Item (xlili) Page 383: how is the “development” and a “picture” objectively described ?; etcetera.

As we see from the above “quick” analysis the problems hinge on our [in]ability to formally, let alone informally describe many management issues. In a sense that is acceptable in as much as ‘management’ is clearly accepted as a non-mechanisable process, one that requires subjective evaluations: “feelings”, “hunches”, and one that requires informal contacts with other managerial personnel and staff.

But still we are left with the problems: Which aspects of domain management are we to describe ? How are we describe, especially formally, the chosen issues ?

Our simplifying and hence simple answer is: the domain engineer shall describe what is objectively observable or concepts that are precisely defined in terms of objectively observable phenomena and concepts defined from these and such defined concepts.

This makes the domain description task a reasonable one, one that can be objectively validated and one where domain description evaluators can objectively judge whether (projected) requirements involving these descriptions may be feasible and satisfactory.

Review of Support Examples

There are three examples (i) a grossly simplifying abstraction: *the enterprise function*, which focuses on the abstract interplay between management groups, workers, etc.; and the formal model is expressed in a recursive function style; (ii) a grossly simplifying abstraction *the enterprise processes*, which focuses on the sequential, non-deterministic processes with input/output messages that communicate between management groups, workers, etc.; the formal model is expressed in the CSP style.

The Enterprise Function

The *enterprise function* is narrated in Sect. I.2.1, and formalised in Sect. I.2.2 on page 390; the formalisation is explained and commented upon on Pages 390–

392 (Sect. I.2.3). Here we shall just briefly discuss meta-issues of domain description, modelling and abstraction.

The description is grossly ‘abstracted’: it leaves out any modelling of what distinguishes top-level, executive, strategic management from mid-level, tactic management, and these from “low-level” operations management, and all of these from supervisors, team leaders and workers. Emphasis has been put solely on abstractions of their intercommunication in order to achieve a “next step” state.

The formalisation of **enterprise** is, formally speaking, doubtful. The semantics of the formal specification language, **RSL**, does not allow such recursions, or rather, put far too severe restrictions on the state space Σ , for the definition to be of even pragmatic interest. Thus the definition is really a fake: at most it hints at what goes on, such as outlined on Pages 390–392 (Sect. I.2.3). Why is the definition a fake? Or rather: Why do we show this “definition”?

In order for a recursive function definition, **enterprise**, (as here over states Σ) to make sense the type Σ must satisfy some ordering properties and so must the component types whose values are involved in any of the auxiliary functions invoked by **enterprise**. Since we have not specified any of these types we take the position that function definition, **enterprise**, is just a pseudo function. It is indicative of “what is going on”, and that is why we bring it!

The Enterprise Processes

The **enterprise** processes are narrated and formalised, alternatively, in Sect. I.3 on Pages 392–400, Here we shall just briefly discuss meta-issues of domain description, modelling and abstraction.

I.1.2 Organisation

Characterisation. *Domain Organisation:* By *domain organisation* we mean the structuring of management and non-management staff levels; the allocation of strategic, tactical and operational concerns to within management and non-management staff levels; and hence the “lines of command”: who does what and who reports to whom — administratively and functionally. ■

I.2 A Simple, Functional Description of Management

By a functional description we mean a description which focuses on functions, that is, which explains things in terms of functions. By a simple description we mean a description which is short.

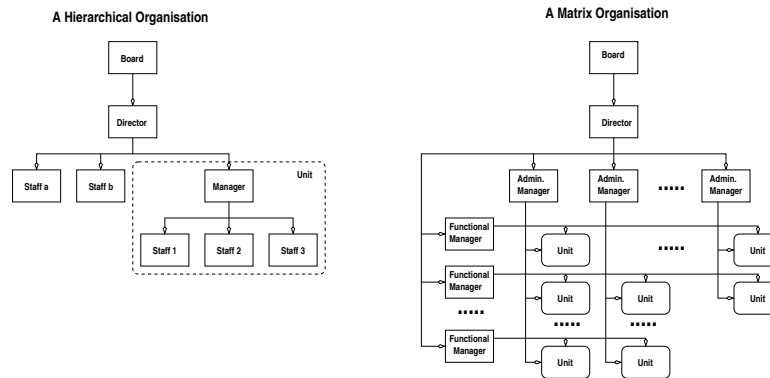


Fig. I.1.1. Two organisational structures

I.2.1 A Base Narrative

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. To make the description simple we think of each of the six categories (i–vi) of personnel and staff as functions, that is, there are six major domain functions related to management.

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. We refer to Sect. I.1.1, Page 388 for a caveat: The doubly¹⁴ recursive definition of the **enterprise** function is a pseudo-definition. It is not a mathematically proper definition.

¹⁴By doubly recursive we mean (i) that the **enterprise** function recurses and (ii) that the definition of its next state transpires from a recursive set of local (**let**) definitions.

I.2.2 A Formalisation

type

0. Σ

value

1. str, tac, opr, sup, tea, wrk: $\Sigma \rightarrow \Sigma$
2. ame_s, ame_t, ame_o, ame_u, ame_e, ame_w: $\Sigma \rightarrow \Sigma^5 \rightarrow \Sigma$
3. objective: $\Sigma^6 \rightarrow \mathbf{Bool}$
4. enterprise, ameliorate: $\Sigma^6 \rightarrow \Sigma$
5. enterprise($\langle \sigma_s, \sigma_t, \sigma_o, \sigma_u, \sigma_e, \sigma_w \rangle$) \equiv
6. **let** $\sigma'_s = \text{ame_s}(\text{str}(\sigma_s))(\langle \sigma'_t, \sigma'_o, \sigma'_u, \sigma'_e, \sigma'_w \rangle)$,
7. $\sigma'_t = \text{ame_t}(\text{tac}(\sigma_t))(\langle \sigma'_s, \sigma'_o, \sigma'_u, \sigma'_e, \sigma'_w \rangle)$,
8. $\sigma'_o = \text{ame_o}(\text{opr}(\sigma_o))(\langle \sigma'_s, \sigma'_t, \sigma'_u, \sigma'_e, \sigma'_w \rangle)$,
9. $\sigma'_u = \text{ame_u}(\text{sup}(\sigma_u))(\langle \sigma'_s, \sigma'_t, \sigma'_o, \sigma'_e, \sigma'_w \rangle)$,
10. $\sigma'_e = \text{ame_e}(\text{tea}(\sigma_e))(\langle \sigma'_s, \sigma'_t, \sigma'_o, \sigma'_u, \sigma'_w \rangle)$,
11. $\sigma'_w = \text{ame_w}(\text{wrk}(\sigma_w))(\langle \sigma'_s, \sigma'_t, \sigma'_o, \sigma'_u, \sigma'_e \rangle)$ **in**
12. **if** objective($\langle \sigma'_s, \sigma'_t, \sigma'_o, \sigma'_u, \sigma'_e, \sigma'_w \rangle$)
13. **then** ameliorate($\langle \sigma'_s, \sigma'_t, \sigma'_o, \sigma'_u, \sigma'_e, \sigma'_w \rangle$)
14. **else** enterprise($\langle \sigma'_s, \sigma'_t, \sigma'_o, \sigma'_u, \sigma'_e, \sigma'_w \rangle$)
- end end**

I.2.3 A Discussion of The Formal Model

A Re-Narration

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.

On The Environment *Ec*.

The model does not explicitly cover interaction with the enterprise customers, suppliers, etc., the enterprise board and other such external domain “players”.

Either we can include those “players” as an additional number of actions like those of *str*, *tac*, . . . , *wrk*, each with their states, or we can think of their states and hence their state changes and interaction (communication — see below) with the enterprise being integrated into the enterprise state.

Thus the omission of the environment is not serious: its modelling is just a simple extension to the given model.

On Intra-communication

The model does not explicitly cover communication between different enterprise staff group members or between these and the environment. We claim now that these forms of communication are modelled by the enterprise state: in each atomic action step such intended communications are reflected in “messages” of the resulting state where these messages are, or are not handled by appropriate other enterprise staff groups in some next atomic step.

On Recursive Next-state Definitions

Above, in Items 1.–14., we gave an intuition of the enterprise operating modes. But we have left un-explained the non-traditional recursive definition and use of mediated states of formula lines 6.–11. We now explain this unconventional recursion.

Let us consider just two such group activities:

$$\begin{aligned}
 &\dots \\
 &\sigma'_\alpha = \text{ame}_\alpha(\text{alpha}(\sigma_\alpha))(< \sigma'_\beta, \sigma'_\gamma, \sigma'_\delta, \sigma'_\epsilon, \sigma'_\zeta >) \\
 &\sigma'_\beta = \text{ame}_\beta(\text{beta}(\sigma_\beta))(< \sigma'_\alpha, \sigma'_\gamma, \sigma'_\delta, \sigma'_\epsilon, \sigma'_\zeta >) \\
 &\dots
 \end{aligned}$$

We observe that the values σ'_α and σ'_β depend on each other. Thus formula lines 6.–11. recursively defines six values. Mathematically such recursive definitions may have solutions. If so, then such solutions are said to be fix points of the equations. In conventional computer science one normally seeks what

is called least fixed point solutions. Such demands are not necessary in the domain. Mathematically one can explain a process that converges towards a solution to the set of recursive equations as an iterative process. If some solution exists then the process converges and terminates in one atomic step. If it does not exist then the process does not terminate — the enterprise is badly managed and goes bankrupt !

Summary

We have sketched a formal model. It captures some aspects of enterprise management and work. It abstracts most of this management and work: there is no hint at the nature of and differences between strategic, tactic, etc., work. That is, we have neither narrate-described such work to a sufficiently concrete level nor, obviously formalised it.

I.3 A Simple, Process Description of Management

I.3.1 An Enterprise System

In this model we view the six “kinds” of manager and worker behaviours as six “kinds” of processes centered around a shared state process.

There are any number,

- \underline{CARD} StrIdx, of strategic,
- \underline{CARD} TacIdx, of tactic and
- \underline{CARD} OpIdx, of operations managers,
- \underline{CARD} SupIdx, of supervisors,
- \underline{CARD} TeaIdx, of team leaders and
- \underline{CARD} WrkIdx, of workers

\underline{CARD} NamIdx expresses the **card**inality of the set of further undefined indexes in NamIdx. The staff index sets, StrIdx, TacIdx, OpIdx, SupIdx, TeaIdx and WrkIdx are pairwise disjoint. The single state process operates concurrently with all the concurrently operating manager, supervisor, team leader and worker behaviours.

I.3.2 States and The System Composition

type

$\Omega, \text{Idx}\Omega = \text{Idx} \xrightarrow{m} \Omega, \text{value } \text{idx}\omega:\text{Idx}\Omega,$
 $\Sigma, \text{value } \sigma:\Sigma$

value

enterprise: **Unit** \rightarrow **Unit**

enterprise() \equiv shared_state(σ) \parallel
 $\parallel \{\text{strateg_process}(i)(\text{idx}\omega(i))|i:\text{StrIdx}\} \parallel \{\text{tactic_process}(i)(\text{idx}\omega(i))|i:\text{TacIdx}\}$
 $\parallel \{\text{operat_process}(i)(\text{idx}\omega(i))|i:\text{OpeIdx}\} \parallel \{\text{superv_process}(i)(\text{idx}\omega(i))|i:\text{SupIdx}\}$
 $\parallel \{\text{teamld_process}(i)(\text{idx}\omega(i))|i:\text{TeaIdx}\} \parallel \{\text{worker_process}(i)(\text{idx}\omega(i))|i:\text{WrkIdx}\}$

The signature of these seven functions will be given shortly.

I.3.3 Channels and Messages

Staff interaction with one another is modelled by messages sent over channels. Staff obtains current state from and “delivers” updated states to the state process, also via channels, one for each staff process.

We postulate a linear ordering, $<$, on indexes. Channels are bidirectional — so there is only a need for $n \times (n - 1)$ channels to serve n staff behaviours.

type

```
Idx = StrIdx | TacIdx | OpeIdx | SupIdx | TeaIdx | WrkIdx
[ axiom : pairwise disjoint ]
[ StrIdx  $\cap$  (TacIdx  $\cup$  OpeIdx  $\cup$  SupIdx  $\cup$  TeaIdx  $\cup$  WrkIdx) =  $\{\}$  ]
[ TacIdx  $\cap$  (OpeIdx  $\cup$  SupIdx  $\cup$  TeaIdx  $\cup$  WrkIdx) =  $\{\}$  ]
[ OpeIdx  $\cap$  (SupIdx  $\cup$  TeaIdx  $\cup$  WrkIdx) =  $\{\}$  ]
[ SupIdx  $\cap$  (TeaIdx  $\cup$  WrkIdx) =  $\{\}$  ]
[ TeaIdx  $\cap$  WrkIdx =  $\{\}$  ]
```

channel

```
 $\sigma\_ch[i:i:ChIdx]: (get\_ \Sigma | \Sigma),$ 
 $staff\_ch[i,j|i,j:ChIdx \bullet j < i]: Msg$ 
```

type

```
Msg, get_ $\Sigma$ 
```

I.3.4 Process Signatures

value

```
shared_state:  $\Sigma \rightarrow \mathbf{in, out} \sigma\_ch[j:Idx] \ \mathbf{Unit}$ 
strateg_process:
  j:StrIdx  $\rightarrow \Omega \rightarrow \mathbf{in, out} \sigma\_ch[j], staff\_ch[j,i|i:Idx \bullet i < j] \ \mathbf{Unit}$ 
tactic_process:
  j:TacIdx  $\rightarrow \Omega \rightarrow \mathbf{in, out} \sigma\_ch[j], staff\_ch[j,i|i:Idx \bullet i < j] \ \mathbf{Unit}$ 
operat_process:
  j:OpeIdx  $\rightarrow \Omega \rightarrow \mathbf{in, out} \sigma\_ch[j], staff\_ch[j,i|i:Idx \bullet i < j] \ \mathbf{Unit}$ 
superv_process:
  j:SupIdx  $\rightarrow \Omega \rightarrow \mathbf{in, out} \sigma\_ch[j], staff\_ch[j,i|i:Idx \bullet i < j] \ \mathbf{Unit}$ 
teamld_process:
  j:TeaIdx  $\rightarrow \Omega \rightarrow \mathbf{in, out} \sigma\_ch[j], staff\_ch[j,i|i:Idx \bullet i < j] \ \mathbf{Unit}$ 
worker_process:
  j:WrkIdx  $\rightarrow \Omega \rightarrow \mathbf{in, out} \sigma\_ch[j], staff\_ch[j,i|i:Idx \bullet i < j] \ \mathbf{Unit}$ 
```

I.3.5 The Shared State Process

The `shared_state` process recurses around the following triplet of actions: waiting for a `get state` (`get_Σ`) message from any staff process, `j`; forwarding the state, `σ`, to that staff process; waiting for an updated state to be returned from staff process `j`.

value

```
shared_state: Σ → in, out σ_ch[j:Idx] Unit
shared_state(σ) ≡
  [] {let msg = σ_ch[j]? in
    case msg of
      req_Σ → (σ_ch[j]!σ ; shared_state(σ_ch[j]?)),
      _ → shared_state(σ_ch[j]?) end end | j:Idx}
```

From the definition of the `enterprise` and the staff processes one can prove that the message, `msg`, is either the `req_Σ` token or a state.

I.3.6 Staff Processes

There are six different kinds of staff processes:

- `strateg_process`, • `operat_process`, • `teamld_process` and
- `tactic_process`, • `superv_process`, • `worker_process`.

We define a process, `staff_process`, to generically model any of these six processes.

I.3.7 A Generic Staff Behaviour

We narrate the staff behaviour: (0.) We can model staff members as having three “alternative” behaviours; (2.–4.) doing their **own** work; (5.–9.) taking an initiative to act with other staff (i); and (10.–13.) being prompted by other staff (i) to react.

(0.) Each staff behaviour **selects** whether to “do own work”, to act, or to react; **select** is assumed to internally non-deterministically (`[]`) choose “what to do”.

(2.) Doing own work means to work on an “own state” (ω'). (3.) The result, ω'' , is “merged” into the global state which is request-obtained from the shared state.

(5.) Acting means to act on the global state (σ); (6.) by performing some “local” operations (`staff_actj(ω, σ)`) (7.) which result in local and global state changes (ω', σ') and the identification of another staff member from whom to request some action (`req`) which then result in some new global state (σ'') and a “local” result (`res`). (8.) The new global state is updated “locally” (9.) as is the local state.

(10.) Reacting means to accept a request (**req**) from some other staff member (**i**); (11.) to then perform some “local” operation (**staff_react_j(req, ω, σ)**) which result in local and global state changes (ω', σ'') and some result (**res**) (12.) The new global state is updated “locally” (13.) as is the local state.

(4., 9., 13.) The staff process iterates (by “tail-recursion”).

value

```

0. staff_process(j)(ω) ≡ let (ω', wtd) = select_wtd(ω) in
1.   case wtd of
2.     own → let ω'' = own_work(ω') in
3.       (σ_ch[j]!σ_updatej(ω'', (σ_ch[j]!get_Σ ; σ_ch[j]?)) ||
4.       staff_process(j)(ω'')) end
5.     act → let σ = (σ_ch[j]!get_Σ ; σ_ch[j]?) in
6.       let (i, req, ω'', σ') = staff_actj(ω', σ) in
7.       let (res, σ'') = (staff_ch[j, i]!(req, σ') ; staff_ch[j, i]?) in
8.       (σ_ch[j]!σ_updatej(req, res, ω'', σ'') ||
9.       staff_process(j)(ω_updatej(req, res, ω''))) end end end
10.    react → let (i, req, σ') = ∏{staff_ch[j, i]? | i:Idx} in
11.      let (res, ω'', σ'') = staff_reactj(req, ω', σ') in
12.      (staff_ch[j, i]!(res, σ_updatej(req, res, ω'', σ'')) ||
13.      staff_process(j)(ω_updatej(req, res, ω''))) end end end end

```

A Diagrammatic Rendition

Let us consider Fig. I.2 on the following page: The digits of Fig. I.2 on the next page refer to the line numbers of the **staff_process** definition (Page 395). The figure intends to show the trace of three processes: the **shared state** process, **staff process j** serving in the **own work** mode as well as in the **active work** mode, and **staff process i** (serving in the **reactive work** mode).

Auxiliary Functions

A number of auxiliary functions have been used.

The **select_wtd** (“select what to do”) clause

type

WhatToDo = own | act | react

value

select_wtd: $\Omega \rightarrow \Omega \times \text{WhatToDo}$

internally, non-deterministically chooses one of the three **WhatToDo** alternatives as also shown in Lines 2., 6. and 11 Page 395. The choice is based on the local state (ω). The outcome of the choice, whether **own**, **act** or **react**, reflects,

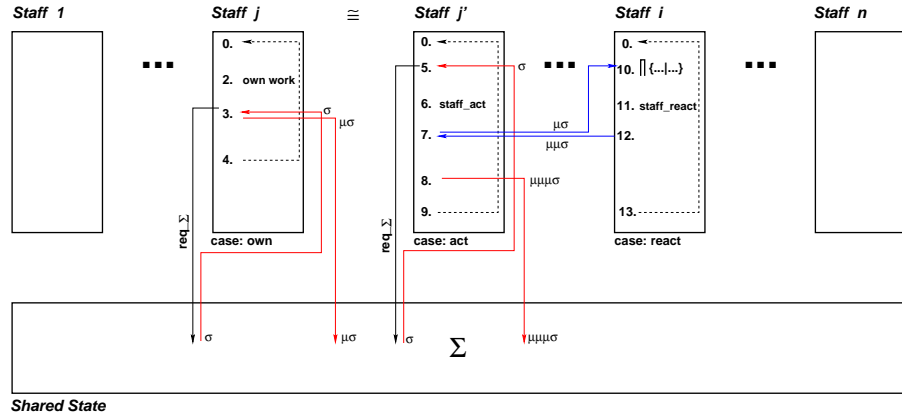


Fig. I.2. Own and ‘action’-‘reaction’ process traces: $\mu\sigma=\sigma'$, $\mu\mu\sigma=\sigma''$, $\mu\mu\mu\sigma=\sigma'''$

we could claim, a priority need for either of these alternatives, or just reflects human vagary! The choice is recorded in an update local state (ω').

The `own_work` function

value `own_work`: $\Omega \rightarrow \Omega$

is “own” as it applies only to the local state (ω). The modelling idea is the following: The ‘own work’, by any staff member, is modelled as taking place, not on the global state, but the result is eventually (see Line 3. Page 395 and, next, the σ_update_j) into the global state. This models, we claim, that staff works locally on “copies” of the global state.

The σ_update_j function

value σ_update_j : $\Omega \times \Sigma \rightarrow \Sigma$

models the “merging” of a local state (ω) into the global state (σ). We do not describe this ‘merging’. But such a description should be made, if need be, separately, for each case of the `own_work` (Line 3.), `staff_act` (Line 8.) and `staff_react` (Line 12., Page 395) alternatives.

The `staff_act` function

type

Request = $\text{Req}_1 \mid \text{Req}_2 \mid \dots \mid \text{Req}_m$

value

`staff_act`: $\Omega \times \Sigma \rightarrow \text{Idx} \times \text{Request} \times \Omega \times \Sigma$

applies to both the local and the global state. The purpose of the `staff_act` query is (line 6., Page 395) to determine with which other staff (`i:Idx`) the

current staff (j) need interact and for what purposes (req), The `staff_act` query updates both the local and the global states (ω'', σ'). The `staff_act` query is assumed to take very little time.

The `ω_{update}` function (Lines 9. and 13., Page 395)

value `ω_{update}` : Request \times Result $\times \Omega \rightarrow \Omega$

updates the local state with the action request and result (req, res) being made to and yielded by staff i . We do not describe this ‘merging’. But such a description should be made, if need be, separately, for each case of the `staff_act` (Line 9.) and `staff_react` (Line 13., Page 395) alternatives.

The `staff_react j` function is, for each j , a (usually large) set of functions:

type

Result = Res₁ | Res₂ | ... | Res _{m}

value

`staff_react j` : Request $\times \Omega \times \Sigma \rightarrow$ Result $\times \Omega \times \Sigma$

`staff_react j (req, ω, σ) \equiv`

(**case** req **of** $r_1 \rightarrow \text{op}_{j_1}(r_1), r_2 \rightarrow \text{op}_{j_2}(r_2), \dots, r_m \rightarrow \text{op}_{j_m}(r_m)$ **end**)(ω, σ)

`op j_i` : Req _{i} $\rightarrow \Omega \times \Sigma \rightarrow$ Res _{i} $\times \Omega \times \Sigma$

Each of these operations are assumed to be of the kind: prepare for this operation to be carried out when doing ‘own work’. We shall comment on these operations below (Sect. I.3.8 [Pages 398–400]).

Assumptions

A number of assumptions have been made in expressing the staff process: The time-duration of the inter-process communications and the ω and σ updates are zero; and the time-durations of `staff_act j (ω, σ)` and `staff_react j (req, ω, σ)` operations are “near”-zero. These two functions do not cause any interaction with neither the shared state nor other staff processes.

The time-duration of the `own_work(ω)` operation may be any length of time.

(The real work is done during the local state change `own_work(ω)` operation and the result of this work is eventually “fed back into the global state”!)

When offering to act or react the designated partner staff behaviour will accept the offer and within a reasonably realistic time interval.

With these assumptions fulfilled it is acceptable to model global state changes as non-interleaved.

I.3.8 Management Operations

So, which are the functions $\text{op}_{j_i}(\omega, \sigma)$? As is obvious from Sect. I.1.1 there are zillions of management operations. They are loosely suggested in Sect. I.1.1, Pages 382–387. Thus we shall not further define these here. But some comments are in order.

Focus on Management

We focus on management rather than on “workers”. Operations by workers, say in a railway transport system, deal with: selling and cancelling tickets, starting, driving and stopping trains, setting signals, laying down new and maintaining rails, etcetera. Management operations are of two kinds: Own, preparatory ‘work’ — that may take hours and days; and dispensing or receiving order — that may take “down to” fractions of minutes. This view of ‘management operations’ partly justifies our staff model.

Own and Global States

Workers spend most, and managers some of their time on ‘own work’ — and then apply the operations of that ‘own work’ to local states. Worker local states are usually very clearly delineated “copies” of the global states somehow made inaccessible to other staff while subject to ‘own work’. Manager local states are usually not so clearly delineated. ‘Own work’ is “reflected back into”, that is, updates, the global state (cf. Line 3. Page 395).

State Classification

We have presented a notion of local ($\omega : \Omega$) global states ($\sigma : \Sigma$) without really saying much about these. We may also have given the impression that these states were inert, that is, changed only when operated upon by the staff. We now redress this impression, that is, we now make it clear that states may have several components, and that some individual state components may be (i) inert dynamic¹⁵, (ii) active dynamic¹⁶ comprising: (ii.1) autonomous active dynamic¹⁷, (ii.2) biddable active dynamic¹⁸ and (ii.3) programmable active dynamic¹⁹ and (iii) reactive dynamic²⁰.

¹⁵An entity is inert dynamic if it never changes value of its own volition.

¹⁶An entity is active dynamic if it changes value of its own volition.

¹⁷An entity is autonomous active dynamic if it changes value only of its own volition.

¹⁸An entity is biddable active dynamic if it can be advised to change state — but it does not have to follow that advice, i.e., that control.

¹⁹An entity is programmable active dynamic if its state changes, over some future time interval, can be fully controlled.

²⁰An entity is reactive dynamic if it performs not necessarily fully predictable state changes in response to external stimuli (i.e., control).

Transport System States

In a transport system these are some of the state components.

Transport Net State Changes: (i) the transport net which changes state due to (i.1) wear and tear of the net, (i.2) setting and resetting of signals, (i.3) insertion and removal of links, etcetera.

Net Traffic State Changes: (ii) the net traffic which changes state due to (ii.1) vehicles entering, moving around, and leaving the net, (ii.2) vehicles accidents, road/bridge/tunnel breakdowns, (ii.3) the state changes of the underlying net, etcetera.

Managed State Changes: (iii) management state changes due to (iii.1) changed transport vehicle timetables being inserted, (iii.2) changed toll road fee schedules being enacted, (iii.3) changed speed limits, (iii.4) changed signalling rules, (iii.5) changed resource (incl. public vehicle) allocation, etcetera.

I.3.9 The Overall Managed System

One can speak of an *overall managed system* which we consider as consisting of all staff, all explicitly shared state components, and the more implicitly observable state components of the environment. Figure I.3 tries to conceptually “picture” such an overall managed system

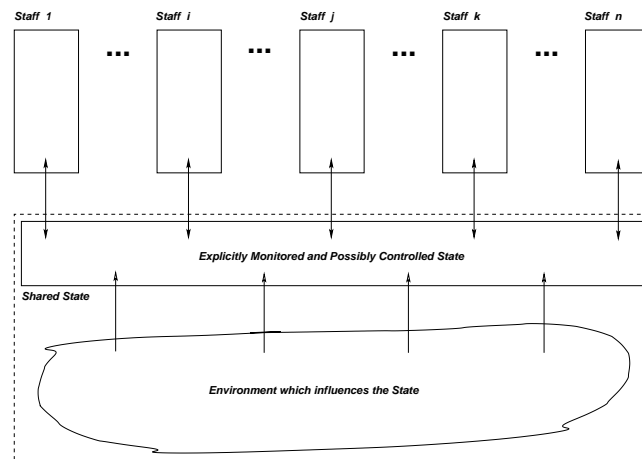


Fig. I.3. Staff and explicit (shared) and implicit states

In a domain model we try, to our best, to describe the n staff behaviours: executive, strategic, tactic and operations managers, supervisors, team leader and workers, and the various explicitly known state components of the domain,

whether only observable (that is: monitorable) or also controllable (that is: generable). In a domain model we may have, through the modelling of the state components, to thus implicitly model environment state concepts.

I.3.10 Discussion

Management Operations

We leave it to the reader to draw the necessary conclusions: (i) only some state components are of concern to management, (ii) not all such state components can be controlled by management, (iii) to model the all the system state changes is thus not a concern of modelling management (and organisation), and (iv) to model all management operations is not feasible.

Managed States

I.4 Discussion of First Two Management Models

I.4.1 Generic Management Models

The first two management models, Sects. I.2–I.3, Pages 388–400, the functional, Sect. I.2, Pages 388–392, and the process descriptions, Sect. I.3, Pages 392–400, really did not show any management aspects of transportation systems.

The two models can be claimed to be generic. As such they apply to a wide variety of domain management.

The two models can also be claimed to be one another's "inverse". The process description, Sect. I.3, can be claimed to "implement" the functional description, Sect. I.2; each "step" of the staff processes can be claimed to correspond to an "iteration" of the "solving" of the recursive equations of Lines 6.–11., Page 390. We leave it as a research challenge for the reader "clean-up" the two formal definitions, that is, express them in a formalism, such that a theorem expressing that the process model "implements" the functional, doubly recursive model. Such a 'clean-up' might possibly involve rewriting the functional, doubly recursive model into an imperative tail-recursive model.

I.4.2 Management as Scripts

It was said, above, that management is manifested in "zillions" of actions, some occurring concurrently, some occurring in strict sequence, etcetera. But nothing was then said, above, of the order, if any, of these actions. Some actions cannot be meaningfully applied before others have been applied. (i) The management decision to remove a specific link, ℓ , must occur some time after a (thus previous) management decision to insert that specific link, ℓ . (ii) The management decision to construct a (new) train timetable must not occur

before reasonable completion dates for the construction of the underlying rail net, the purchase of required rolling stock, and the hiring of required net and train operation staff have all been established. (iii) the management go-ahead for the start of train traffic according to a new timetable must not occur before the completion of the construction of a train (staff) rostering plan, the construction of a train maintenance plan, and the rail net construction, etcetera.

One — not so extreme — interpretation of the above is that we cannot meaningfully describe specific concurrent and sequential sets of management actions but must basically, in most cases of systems, accept any such patterns of actions.

Another — slightly less extreme — interpretation of the above is that we can in some cases describe what we shall later define as a family of management scripts Let that suffice for the time being.

I.5 Transport Enterprise Organisation

Transportation is “home” to many different kinds of enterprises. Each of these enterprises is concerned with relatively distinct and reasonably non-overlapping issues: road (bridge, etc.) building, \mathcal{E}_{rd_b} , road (etc.) maintenance, \mathcal{E}_{rd_m} , road & car traffic signaling, \mathcal{E}_{rd_s} , bus services (i), \mathcal{E}_{bs_i} , fire brigade, \mathcal{E}_{fi} , police, \mathcal{E}_{po} , rail building, \mathcal{E}_{rl_b} , rail maintenance, \mathcal{E}_{rl_m} , rail & train signaling, \mathcal{E}_{rl_s} , train services (j), \mathcal{E}_{tp_j} , map making (k) \mathcal{E}_{mm_k} , etcetera. But they “share” the transportation net.

I.5.1 Transport Organisations

Each kind of transportation enterprise, say \mathcal{E}_i , covers a subset, say $\mathcal{NET}_{\mathcal{E}_i}$, of the net, that is, not necessarily the entire net. For two or more $i, j, i \neq j$, it may be that $\mathcal{NET}_{\mathcal{E}_i} \cap \mathcal{NET}_{\mathcal{E}_j} \neq \{\}$. Each transportation enterprise has its own distinct staff, that is, sets of strategics managers, $\mathcal{STR}_{\mathcal{E}_i}$, tactics managers, $\mathcal{TAC}_{\mathcal{E}_i}$, operations managers, $\mathcal{OPS}_{\mathcal{E}_i}$, supervisors, $\mathcal{SUP}_{\mathcal{E}_i}$, team leaders, $\mathcal{TLD}_{\mathcal{E}_i}$, and workers, $\mathcal{WRK}_{\mathcal{E}_i}$. For some managers, supervisors, team leaders and workers the areas of the net for which they are responsible are proper, disjoint subsets of the net.

I.5.2 Analysis

To describe the transportation domain one has to model for each transportation enterprise, \mathcal{E}_i , separate subsets of the net $\mathcal{NET}_{\mathcal{E}_{rd_b}}, \mathcal{NET}_{\mathcal{E}_{rd_m}}, \mathcal{NET}_{\mathcal{E}_{rd_s}}, \mathcal{NET}_{\mathcal{E}_{bp_i}}, \mathcal{NET}_{\mathcal{E}_{rl_b}}, \mathcal{NET}_{\mathcal{E}_{rl_m}}, \mathcal{NET}_{\mathcal{E}_{rl_s}}, \mathcal{NET}_{\mathcal{E}_{tp_i}}, \mathcal{NET}_{\mathcal{E}_{fi}}, \mathcal{NET}_{\mathcal{E}_{po}}$, and $\mathcal{NET}_{\mathcal{E}_{mm_k}}$; and, for each such transportation enterprise, one has to model a number of separate enterprise structures: $\mathcal{E}_{rd_b}, \mathcal{E}_{rd_m}, \mathcal{E}_{rd_s}, \mathcal{E}_{bp_i}, \mathcal{E}_{rl_b}, \mathcal{E}_{rl_m}, \mathcal{E}_{rl_s}, \mathcal{E}_{tp_i}, \mathcal{E}_{fi}, \mathcal{E}_{po}$, and \mathcal{E}_{mm_k} .

I.5.3 Modelling Concepts

Net Kinds

In order to model the various nets, $\mathcal{NET}_{\mathcal{E}_i}$, given that we have a base model N , we introduce a notion of ‘net kind’: one for each of the nets $\mathcal{NET}_{\mathcal{E}_{rd_b}}$, $\mathcal{NET}_{\mathcal{E}_{rd_m}}$, $\mathcal{NET}_{\mathcal{E}_{rd_s}}$, $\mathcal{NET}_{\mathcal{E}_{bp_i}}$, $\mathcal{NET}_{\mathcal{E}_{rl_b}}$, $\mathcal{NET}_{\mathcal{E}_{rl_m}}$, $\mathcal{NET}_{\mathcal{E}_{rl_s}}$, $\mathcal{NET}_{\mathcal{E}_{tp_t}}$, $\mathcal{NET}_{\mathcal{E}_{fi}}$, $\mathcal{NET}_{\mathcal{E}_{po}}$, $\mathcal{NET}_{\mathcal{E}_{mm_k}}$, etcetera. A ‘net kind’, $k:K$, is like a type designator

type

```

K = SimP|BusK|TrainK|MapMK
SimK == road_b|road_m|road_s|rail_b|rail_m|rail_s|fireb|police
BusK == bus_p1|bus_p2|...|bus_pm
TrainK == train_p1|train_p2|...|train_pn
MapMK == map_m1|map_m2|...|map_mo

```

To each hub and link in a net we then associate zero, one or more net kinds. We then postulate an observer function:

value

```
obs_K: (H|L) → K-set
```

Given a net kind we can then “extract” the net of hubs and links “carrying” that net kind:

value

```
xtr_N: N × K → N
```

To guarantee that the extracted net is indeed a (well-formed) net we must make sure that any assignment of net kinds to hubs and links results in well-formed net kind nets:

value

```
wf_NK: N → Bool
```

To express wf_NK we define a function

value

```
xtr_Ks: N → K-set
```

which collects all net kinds from all hubs and links of the net.

value

```

xtr_Ks(hs,ls) ≡
  ∪{obs_Ks(h)|h:H•h ∈ hs} ∪ ∪{obs_Ks(l)|l:L•l ∈ ls}

wf_N'(hs,ls) ≡
  ∀ k:K • k ∈ xtr_Ks(hs,ls) ⇒
    wf_N({h|h:H•h ∈ hs ∧ k ∈ obs_Ks(h)}, {l|l:L•l ∈ ls ∧ k ∈ obs_Ks(l)})

```


The predicate wf_N' extends the predicate wf_N which corresponds to the satisfaction of all the axioms given in Sect. F.4, Pages 292–297.

$$\begin{aligned} \text{xtr_N}((\text{hs}, \text{ls}), k) \equiv \\ (\{h|h:\text{H} \bullet h \in \text{hs} \wedge k \in \text{obs_Ks}(h)\}, \{l|l:\text{L} \bullet l \in \text{ls} \wedge k \in \text{obs_Ks}(l)\}) \\ \text{pre } k \in \text{xtr_Ks}(\text{hs}, \text{ls}) \end{aligned}$$

Enterprise Kinds

In order to model the various enterprises, \mathcal{E}_i , given that we have a base model for business staff, SIdx , we introduce a notion of ‘enterprise kind’: one for each of the enterprise kind: \mathcal{E}_{rd_b} , \mathcal{E}_{rd_m} , \mathcal{E}_{rd_s} , \mathcal{E}_{bp_i} , \mathcal{E}_{rl_b} , \mathcal{E}_{rl_m} , \mathcal{E}_{rl_s} , \mathcal{E}_{tp_t} , \mathcal{E}_{fi} , \mathcal{E}_{po} , \mathcal{E}_{mm_k} , etcetera. A ‘enterprise kind’, b:B , is like a type designator

type

$\text{E} = \text{SimE} | \text{BusE} | \text{TrainE} | \text{MapMakE}$
 $\text{SimE} == \text{road_b} | \text{road_m} | \text{road_s} | \text{rail_b} | \text{rail_m} | \text{rail_s} | \text{fireb} | \text{police}$
 $\text{BusE} == \text{bus_p1} | \text{bus_p2} | \dots | \text{bus_pm}$
 $\text{TrainE} == \text{train_p1} | \text{train_p2} | \dots | \text{train_pn}$
 $\text{MapMakE} == \text{map_m1} | \text{map_m2} | \dots | \text{map_mo}$

Staff Kinds

To each staff we then associate an enterprise kind.

value

$\text{obs_B}: \text{SIdx} \rightarrow \text{B}$

We may further have to impose that interaction between staff, viz.:

$\text{staff_ch}[i, j]$

be subject to an “internal business constraint”:

$\text{obs_B}(i) = \text{obs_B}(j)$

Staff Kind Constraints

Narrative

The staff, $\text{Staff}_{\mathcal{E}_i}$, of each transportation enterprise, \mathcal{E}_i , can be roughly categorised into: strategic managers, $\text{STR}_{\mathcal{E}_i}$, tactics managers, $\text{TAC}_{\mathcal{E}_i}$, operations managers, $\text{OPS}_{\mathcal{E}_i}$, supervisors, $\text{SUP}_{\mathcal{E}_i}$, team leaders, $\text{TLD}_{\mathcal{E}_i}$, and workers $\text{WRK}_{\mathcal{E}_i}$, as already mentioned (Page 401). And each of these can be further sub-categorised.

Formalisation

We suggest the “beginnings” of a formalisation.

type

StaffK == stra|tac|ope|sup|tld|wrk

value

obs_StaffK: SIdx \rightarrow StaffK

A more realistic model, for a given enterprise, would provide a far more detailed categorisation. Typically there might be several “layers” of strategic and of tactic and of operations management. Similarly a model might detail different kinds of supervisor, team leader and worker (“blue collar”) staff.

Hierarchical Staff Structures

We refer to Fig. I.1, Page 389.

*Matrix Staff Structures***Net and Enterprise Kind Constraints***Narrative*

A link (or a hub) is either a road or a rail link (hub). If a link is a road link then the two hubs that it connects are road hubs. If a hub is a road hub then all the links emanating from the hub are road links. If a link is a rail link then the two hubs that it connects are rail hubs. If a hub is a rail hub then all the links emanating from the hub are rail links. In consequence we may speak of disjoint road nets and rail nets. The enterprises associated with a road [rail] net must be road [rail] related (building, maintenance, signaling, bus [train] services, police, etc.).

*Formalisation***type**

NetK: road|rail|...

value

obs_NetK: (H|L) \rightarrow NetK

xtr_L: LI \rightarrow L-set \rightarrow L, xtr_H: HI \rightarrow H-set \rightarrow H

xtr_L(li)(ls) $\equiv \exists!l:L \bullet l \in ls \wedge \text{obs_LI}(l)=li$

xtr_H(hi)(hs) $\equiv \exists!h:H \bullet h \in hs \wedge \text{obs_HI}(h)=hi$

wf_N'': N \rightarrow **Bool**

wf_N''(hs,ls) \equiv

$\forall h:H \bullet h \in hs \Rightarrow$

$$\begin{aligned}
& \forall li:LI \bullet li \in \text{obs_LIs}(h) \Rightarrow \text{obs_NetK}(\text{xtr_L}(li)(ls)) = \text{obs_NetK}(h) \wedge \\
& \forall l:L \bullet l \in ls \Rightarrow \\
& \quad \forall hi:HI \bullet hi \in \text{obs_HIs}(l) \Rightarrow \text{obs_NetK}(\text{xtr_H}(hi)(hs)) = \text{obs_NetK}(l)
\end{aligned}$$

The predicate wf_N'' extends the predicate wf_N' given earlier (Page 402).

value

```

netk: N → NetK-set
netk(hs,ls) ≡ {obs_NetK(h)|h:H•h ∈ hs} ∪ {obs_NetK(l)|l:L•l ∈ ls}

road_k: NetK-set = {road_b, road_m, road_s, bus_p1, bus_p2, ..., bus_pm, fireb, police},
rail_k: NetK-set = {rail_b, rail_m, rail_s, train_p1, train_p2, ..., train_pn, fireb, police}

wf_N''': N → Bool
wf_N'''(hs,ls) ≡
  let nks = netk(hs,ls) in
  case nks of
    {road} → xtr_Ks(hs,ls) ⊆ road_k, {rail} → xtr_Ks(hs,ls) ⊆ rail_k, _ → false
  end end

```

The predicate wf_N''' extends wf_N'' given earlier (Page 404).

I.5.4 Net Signaling

In the previous section on support technology we did not describe who or which “ordered” the change of hub states. We could claim that this might very well be a task for management.

(We here look aside from such possibilities that the domain being modelled has some further support technology which advises individual hub controllers as when to change signals and then into which states. We are interested in finding an example of a management & organisation facet — and the upcoming one might do!)

Narrative

So we think of a ‘net hub state management’ for a given net. That management is divided into a number of ‘sub-net hub state managements’ where the sub-nets form a partitioning of the whole net. For each sub-net management there are two kinds management interfaces: one to the overall hub state management, and one for each of interfacing sub-nets. What these managements do, what traffic state information they monitor, etcetera, you can yourself “dream” up. Our point is this: We have identified a management organisation.

Formalisation**type**

$$\text{HIsLIs} = \text{HI-set} \times \text{LI-set}$$

$$\text{MgtNet}' = \text{HIsLIs} \times \mathbb{N}$$

$$\text{MgtNet} = \{ | \text{mgtnet} : \text{MgtNet}' \bullet \text{wf_MgtNet}(\text{mgtnet}) | \}$$

$$\text{Partitioning}' = \text{HIsLIs-set} \times \mathbb{N}$$

$$\text{Partitioning} = \{ | \text{partitioning} : \text{Partitioning}' \bullet \text{wf_Partitioning}(\text{partitioning}) | \}$$
value

$$\text{wf_MgtNet} : \text{MgtNet}' \rightarrow \mathbf{Bool}$$

$$\text{wf_MgtNet}((\text{his}, \text{lis}), n) \equiv$$

$$[\text{The his component contains all the hub ids. of links identified in lis}]$$

$$\text{wf_Partitioning} : \text{Partitioning}' \rightarrow \mathbf{Bool}$$

$$\text{wf_Partitioning}(\text{hisliss}, n) \equiv$$

$$\forall (\text{his}, \text{lis}) : \text{HIsLIs} \bullet (\text{his}, \text{lis}) \in \text{hisliss} \Rightarrow \text{wf_MgtNet}((\text{his}, \text{lis}), n) \wedge$$

$$[\text{no sub-net overlap and together they "span" } n]$$
I.6 Discussion

The reader shall have to wait for [97] to be published to see the text of this section — well, in general, a more satisfactory presentation of its (intended) material !.

J

Bayesian Networks

This Appendix chapter constitutes a fragment of a ‘Technical Note’. It is not part of the ‘Thesis’. Section 5.5.3 refers to this Appendix chapter. I wish I had time to also complete this note.

The full note was prepared for a PhD student, Mr. Yin Hongli at NUS.¹ Mr. Yin was then to complete the narration and formalisation of more comprehensive concept of Bayesian Networks (than hinted at here) as part of his PhD thesis.

J.1 Introduction

The background for this report is:

- The fact that there is now an abundance of what is known as formal methods for the provably correct development of software: B [1, 131], CafeOBJ [147, 148, 161, 162], CASL [24, 142, 241, 242], VDM [111, 112, 157, 158], RAISE [87–89, 97, 165, 166, 168], Z [186, 187, 283, 284, 297], etc.
- The fact that the formal specification languages of software engineering can be used far more widely than for just software development.
- The fact that many research and technical reports in other fields than computer and computing science usually use an ad hoc mixture of informal (and hopefully precise) English, informal mathematical notation and “snapshot pictures” (figures, diagrams).
- The fact that this latter is indeed the case for a number of papers in biomedical computing, viz., papers on the representation of knowledge using Bayesian networks.

¹Department of Computer Science, School of Computing, National University of Singapore, 3 Science Drive 2, Singapore S-117543, Republic of Singapore

J.1.1 Aims & Objectives**Aims**

The aims of the present working report (in progress) are:

- To show an example stepwise development of provably correct software for the handling of Bayesian networks.
- To show this example in the form of a so-called formal methods “lite” approach.

Objectives

The objectives of the present working report (in progress) are:

- To possibly convince the reader that it is about time that formal methods “lite” approaches be applied more widely,
- by showing that it can be done, and oftentimes more elegantly, more transparently than normally achieved in the literature.

J.1.2 Motivation

Our motivation extends the above:

- To achieve the above objectives.
- To experiment with formalisations of representations of knowledge.
- To, in the longer run, achieve a unifying approach to modelling representations of knowledge.

J.1.3 Structure of Document

The report is structured as follows:

Chapter 2. We illustrate four ways of presenting Bayesian networks:

1. Informally, in reasonably precise English, and
2. semi-formally to formally in terms of
 - (a) a picture of a particular (ie., the running example) Bayesian network,
 - (b) a tabular representation of the running example Bayesian network and
 - (c) a formalisation of the type of all Bayesian networks.

Our objective with the above stepwise unfolding (1., 2.1, 2.2 and 2.3) of ways of presenting Bayesian networks is to show that pictures and tabular representations can not present the full space, i.e., the type of all Bayesian networks. Only an informal, but precise English and the formal type definition can.

Chapter 3. On the background of the formal type definition of Bayesian networks we give, in this chapter, a systematic treatment of this formalisation and a derivative:

1. First we cover the formalisation of full Bayesian networks and their well-formedness conditions.
2. Then we cover functions over such full Bayesian networks, auxiliary and what we shall call trimming functions. These are functions that (i) discover context independence (also therefore context sensitivity) and (ii) remove (trim) unnecessary links in the Bayesian network (i.e., graph).
3. Finally we cover types and functions which represent simplified Bayesian network trees and their conversion from full Bayesian networks.

Chapter 4. In this chapter we informally show how to transform the abstract (applicative) functions given in Chap. 3 into imperative procedures over assignable variables.

Chapter 5. The function definitions of Chap. 4 can then serve as a basis for informally, but systematically developing Java methods.

Chapter 6. We conclude speculating on future work and by dispensing acknowledgements.

J.2 Bayesian Networks

J.2.1 Informal Introduction to Bayesian Networks

Let there be given a finite set of variables, $v_i : V$, for $1 \leq i \leq n$. Each variable, v_i has values ranging over a usually small, but always finite set of discrete values, $\nu_i : VAL$. With each variable is associated a probability of it taking on a specific value in its value range. That probability is a function of the values of one or more of the other variables. We can model the collection of Bayesian variables and their values as follows.

J.2.2 Representations of Bayesian Networks

Pictures: Network Graphs

Consider Fig. J.1 on the next page.

Bayesian Tables

$[v_1 \mapsto [\text{ON} \mapsto 0.5, \text{OFF} \mapsto 0.5], v_2 \mapsto [\text{RED} \mapsto 0.2, \text{BLU} \mapsto 0.3, \text{GRE} \mapsto 0.5]]$

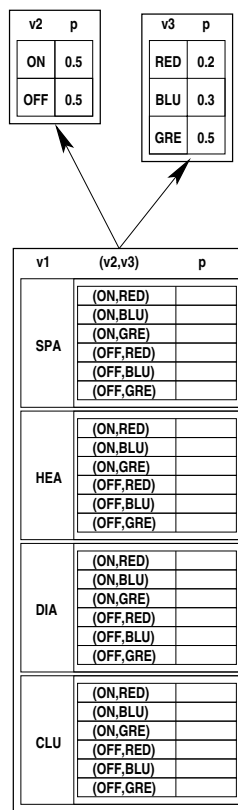


Fig. J.1. A Bayesian Network

Formalisation

First Rough Formal Model of Bayesian Networks

```

type
  Var, VAL, TypNm
  P = { | p:Real • 0 ≤ p ≤ 1 | }
  TypDef = Var  $\overline{\mapsto}$  TypNm
  Types = TypNm  $\rightarrow$  VAL-set
  BN = Var  $\overline{\mapsto}$  (VAL  $\overline{\mapsto}$  ((Var  $\overline{\mapsto}$  VAL)  $\overline{\mapsto}$  P))

```

Annotation :

Var : Set of variables.

VAL : Set of values.

P : Probability

RV = V $\overline{\mapsto}$ VAL-set Set of maps from variables to their finite set of discrete values.

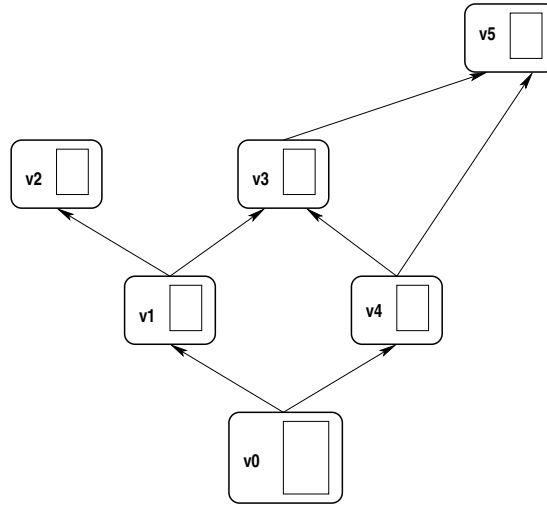


Fig. J.2. A Schematic Bayesian Network

$PB = V \xrightarrow{m} (VAL \xrightarrow{m} ((V \xrightarrow{m} VAL) \xrightarrow{m} P))$: Set of maps from variables to maps of values to maps from pairs of variables and values to probabilities.

An Example

An Example Formal Value

value

```

v1,v2,v3,v4,v5,v6:Var
on,off,red,blue,green,club,spade,heart,diamond:VAL
td:TypDef
tys:Types
t1,t2,t3:TypNm
bn:BN

```

axiom

```

tyd = [ v1↦t1,v2↦t2,v3↦t3,v4↦t1,v5↦t2,v6↦t13 ]
[ t1↦{on,off},t2↦{red,blue,green},t3↦{club,spade,heart,diamond} ]
bn = [ v1 ↦ [ [ on ↦ [ [ v2 ↦ red ] ↦ 0.1,
                      [ v2 ↦ blue ] ↦ 0.2,
                      [ v2 ↦ green ] ↦ 0.7 ],
              off ↦ [ ... ] ] ]
      v2 ↦ ...
      v3 ↦ ... ]

```


K

On Russel's' Theory of Descriptions

This appendix chapter simply brings an edited version of http://en.wikipedia.org/wiki/Theory_of_Descriptions:

K.1 Denoting Sentences

Russell's theory of descriptions was most clearly expressed in his 1905 essay "On Denoting" [270]. Russell's theory is about the logical form of expressions involving denoting phrases, which he divides into three groups:

1. *Denoting phrases* which do not denote anything, for example "the present King of France".
2. *Definite phrases* which denote one definite object, for example "the present King of England" (Edward VII at the time Russell was writing). We need not know which object the phrase refers to for it to be unambiguous, for example "the tallest spy" is a unique individual but his or her actual identity is unknown).
3. *Indefinite phrases* which denote ambiguously, for example, "a man".

K.2 Definite Descriptions

Definite descriptions involve Russell's second group of denoting phrases, and *indefinite descriptions* involve Russell's third group. Descriptions typically appear to be of the standard subject-predicate form. Russell proposed his theory of descriptions in order to solve several problems in the philosophy of language. The two major problems are of (1) *co-referring expressions* and (2) *non-referring expressions*.

K.2.1 Co-referring Expressions

The problem of *co-referring expressions* originated primarily with Gottlob Frege as the problem of informative identities. For example, if the morning star and the evening star are the same planet in the sky (indeed they are), how is it that someone can think that the morning star rises in the morning but the evening star does not? That is, someone might find it surprising that the two names refer to the same thing (i.e. the identity is informative). This is apparently problematic because although the two expressions seem to denote the same thing, one cannot substitute one for the other, which one ought to be able to do with identical or synonymous expressions.

K.2.2 Non-referring Expressions

The problem of *non-referring expressions* is that certain expressions that are meaningful do not seem to refer to anything. For example, by “any man is sexist” it is not meant that there is a particular individual, namely any man, that has the property of being sexist (similar considerations go for “some man”, “every man”, “a man”, and so on). Likewise, by “the present King of France is bald” it is not meant that there is some individual, namely the present King of France, who has the property of being bald (France is presently not a monarchy, so there is currently no King of France). Thus, what Russell wants to avoid is admitting mysterious non-existent entities into his ontology. Furthermore, the law of the excluded middle requires that one of the following propositions, for example, must be true: either “the present King of France is bald” or “it is not the case that the present King of France is bald”. Normally, propositions of the subject-predicate form are said to be true if and only if the subject is in the extension of the predicate. But, there is currently no King of France. So, since the subject does not exist, it is not in the extension of either predicate (it is not on the list of bald people or non-bald people). Thus, it appears that this is a case in which the law of excluded middle is violated, which is also an indication that something has gone wrong. Russell says in his paper, in a typically sly dig at a school of philosophy with which he disagreed, that “Hegelians, who love a synthesis, will probably conclude that he wears a wig.”

K.3 Definite Descriptions

Russell analyzes definite descriptions similarly to indefinite descriptions, except that the individual is now uniquely specified. Take as an example of a definite description the sentence “the present King of France is bald”. Russell analyzes this phrase into the following component parts (with x and y representing variables):

1. there is an x such that x is the King of France;

2. there is no y , other than x , such that y is the King of France (i.e., x is the only King of France);
3. x is bald.

Thus, a definite description (of the general form ‘the F is G ’) becomes the following existentially quantified phrase in classic symbolic logic (where x and y are variables and F and G are predicates — in the example above, F would be “is the King of France”, and G would be “is bald”):

$$\exists x(F(x) \wedge \forall y(F(y) \supset x = y) \wedge G(x)).$$

Informally, this reads as follows: something exists with the property F , there is only one such thing, and this unique thing also has the property G .

This analysis, according to Russell, solves the two problems noted above as related to definite descriptions:

1. “The morning star rises in the morning” no longer needs to be thought of as having the subject-predicate form. It is instead analyzed as “there is one unique thing such that it is the morning star and it rises in the morning”. Thus, strictly speaking, the two expressions “the morning star ...” and “the evening star ...” are not synonymous, so it makes sense that they cannot be substituted (the analysed description of the evening star is “there is one unique thing such that it is the evening star and it rises in the evening”).
2. Since the phrase “the present King of France is bald” is not a referring expression, according to Russell’s theory it need not refer to a mysterious non-existent entity. Russell says that if there are no entities C with property F , the proposition “ C has property G ” is false for all values of G .

Russell writes:

Thus “the present King of France is bald” is certainly false; and “the present King of France is not bald” is false if it means “There is an entity which is now King of France and is not bald” but it is true if it means “It is false that there is an entity which is now King of France and is bald” [270].

Russell says that all propositions in which the King of France has a primary occurrence are false. The denials of such propositions are true, but in these cases the King of France has a secondary occurrence (the truth value of the proposition is not a function of the truth of the existence of the King of France).

K.4 Indefinite Descriptions

Take as an example of an indefinite description the sentence “some man is being obnoxious”. Russell analyzes this phrase into the following component parts (with x and y representing variables):

1. There is an x such that x is a man.
2. x is being obnoxious.

Thus, an indefinite description (of the general form “an F is G ”) becomes the following existentially quantified phrase in classic symbolic logic (where x and y are variables and F and G are predicates):

$$\exists x(F(x) \wedge G(x))$$

Informally, this reads as follows: “there is something such that it is F and G ”.

This analysis, according to Russell, solves the second problem noted above as related to indefinite descriptions. Since the phrase “some man is being obnoxious” is not a referring expression, according to Russell's theory, it need not refer to a mysterious non-existent entity. Furthermore, the law of excluded middle need not be violated (i.e. it remains a law), because “some man is being obnoxious” comes out true: there is a person that is both a man and obnoxious. Thus, Russell's theory seems to be a better analysis insofar as it solves several problems.

L

Repository

This appendix serves as a repository for miscellaneous pieces of information.

L.1 Definitions	417
L.1.1 'Method'	418
L.1.2 'Principle', 'Technique' and 'Tool'	420
L.2 Methodological Indexes	421
L.2.1 Volume 1	421
Principles	421
Techniques	421
Tools	422
L.2.2 Volume 2	422
Principles	422
Techniques	423
Tools	424
L.2.3 Volume 3	424
Principles	424
Techniques	426
Tools	428

L.1 Definitions

Some characterisations used in [87–89] are based on sources such as the *Compact Oxford English Dictionary*¹, *Cambridge International Dictionary of En-*

¹Compact Oxford English Dictionary of Current English, Oxford University Press, Third Edition, eds. Catherine Soanes and Sara Hawker, ISBN-10: 0-19-861022-X and ISBN-13: 978-0-19-861022-9, Publication date: 23 June 2005

glish², *The American Heritage Dictionary of the English Language*³, and *Merriam Webster Unabridged*⁴. All were consulted in their on-line versions (with proper citation).

L.1.1 'Method'

The concept of method was thought of such importance as to warrant a rather thorough search for and analysis of variant definitions.

1. Compact Oxford English Dictionary:

- (a) a way of doing something;
- (b) orderliness of thought or behaviour;
- (c) Greek *methodos* 'pursuit of knowledge'.

2. Cambridge International Dictionary of English:

- (a) a particular way of doing something

3. The American Heritage Dictionary of the English Language:

- (a) A means or manner of procedure, especially a regular and systematic way of accomplishing something.
- (b) Orderly arrangement of parts or steps to accomplish an end.
- (c) The procedures and techniques characteristic of a particular discipline or field of knowledge.

4. Merriam Webster Unabridged, Definitions:

- (a) a procedure or process for attaining an object;
- (b) a systematic procedure; technique, or set of rules employed in philosophical inquiry;
- (c) particular approach to problems of truth or knowledge:
 - i. the pragmatic method tries to interpret each notion by tracing its respective practical consequences – William James;
 - ii. the dialectical method assumes the primacy of matter;
 - iii. the method of the positivists applied to philosophy the procedures of the natural sciences.
- (d) a discipline or system sometimes considered a branch of logic that deals with the principles applicable to inquiry into or exposition of some subject;
- (e) a systematic procedure, technique, or mode of inquiry employed by or proper to a particular science, art, or discipline.
- (f) a methodical exposition
- (g) a table of contents

²Cambridge International Dictionary of English, ed. Paul Procter, Cambridge University Press, ISBN-10: 0521482364, ISBN-13: 978-0521482363, Publication date: April 28 1995

³The American Heritage Dictionary of the English Language, Houghton Mifflin Company. eds. Pickett, Joseph P. et al., ISBN: 0-395-82517-2, Publication date: 2000

⁴<http://unabridged.merriam-webster.com/>

- (h) an arrangement that follows a plan or design c : orderliness and regularity or habitual practice of them in action (thrift was as much in her nature as method – Sylvia T. Warner) (time enough to do everything if only you used method – Angela Thirkell)
- 5. **Merriam Webster Unabridged, Synonyms:**
 METHOD, MODE, MANNER, WAY, FASHION, and SYSTEM can all indicate the means used or the procedure followed in doing a given kind of work or achieving a given end.
 - (a) METHOD can apply to any plan or procedure but usually implies an orderly, logical, effective plan or procedure, connoting also regularity (the crude methods of trial and error – Henry Suzzallo) (the method of this book is to present a series of successive scenes of English life – G.M.Trevelyan) (Marx's doctrine is not a system of scientific truths, it merely represents a method – one possible approach to social and historical reflection – Paolo Milano) (surely not to leave to fitful chance the things that method and system and science should order and adjust – B.N.Cardozo)
 - (b) MODE, sometimes interchangeable with METHOD, seldom implies order or logic, suggesting rather custom, tradition, or personal preference (a rational mode of dealing with the insane – W.R.Inge) (this intuition is essentially an aesthetic mode of apprehension – H.J.Muller) (the mode of reproduction of plants and animals, however, is fundamentally identical – Encyc. Americana)
 - (c) MANNER usually suggests a personal or peculiar course or procedure, often interchanging with MODE in this sense (the manner by which the present pattern of land ownership in this country has evolved – A.F.Gustafson) (it is not consistent with his manner of writing Latin – G.C.Sellery) (bearing loaves of sweet bread and of cornbread made with yeast in the Portuguese manner – Dana Burnet)
 - (d) WAY is general and interchangeable with METHOD, MODE, or
 - (e) MANNER (a special way to raise orchids) (the way the machine works) (the town's way of life) (one's way of tying his tie) FASHION, in this comparison, may be distinguished from WAY in often suggesting a more superficial origin or source as in a mere fashion or ephemeral style (was so popular that his subjects took to wearing monocles, in his fashion – Time) (Harvard has stoutly and successfully resisted the fashion by which the grounds of an American college have come to be known as a campus – Official Register of Harvard University) (who were poor in a fashion unknown to North America – Herbert Agar)
 - (f) SYSTEM suggests a fully developed, often carefully formulated method, usually emphasizing the idea of rational orderliness (every new discovery claims to form an addition to the system of science as transmitted from the past – Michael Polanyi) (behavior which is not in accord

with the individual's system elicits responses of fear – Ralph Linton)
 (an earnest plea for radical reformation of the system of assessment
 and taxation – C.A.Duniway)

6. **Wikipedia:**

Method may refer to:

- (a) *Discourse on Method*, a philosophical and mathematical treatise by René Descartes
- (b) 'Scientific method', a series of steps taken to acquire knowledge
- (c) 'Method' (computer science), a piece of code associated with a class or object to perform a task
- (d) 'Method' (software engineering), a series of steps taken to build software
- (e) 'Method acting', a style of acting in which the actor attempts to replicate the conditions under which the character operates
- (f) "The Method of Mechanical Theorems", part of the Archimedes Palimpsest
- (g) 'Method' (music), a kind of textbook to help students learning to play a musical instrument

L.1.2 'Principle', 'Technique' and 'Tool'

From Merriam Webster Unabridged we cull:

- 1. Principle
 - (a) a comprehensive and fundamental law, doctrine, or assumption
 - (b) a rule or code of conduct
 - (c) habitual devotion to right principles
 - (d) the laws or facts of nature underlying the working of an artificial device
 - (e) a primary source
 - (f) an underlying faculty or endowment
 - (g) an ingredient (as a chemical) that exhibits or imparts a characteristic quality
- 2. Technique
 - (a) the manner in which technical details are treated (as by a writer) or basic physical movements are used (as by a dancer)
 - (b) ability to treat such details or use such movements
 - (c) a body of technical methods (as in a craft or in scientific research)
 - (d) a method of accomplishing a desired aim
- 3. Tool
 - (a) a handheld device that aids in accomplishing a task
 - (b) the cutting or shaping part in a machine or machine tool
 - (c) machine for shaping metal

- (d) something (as an instrument or apparatus) used in performing an operation or necessary in the practice of a vocation or profession (a scholar's books are his tools)
- (e) an element of a computer program (as a graphics application) that activates and controls a particular function (a drawing tool)
- (f) a means to an end (a book's cover can be a marketing tool)
- (g) one that is used or manipulated by another

L.2 Method Indexes

L.2.1 Volume 1

Principles

- 7.9.4 λ -Abstraction, 122
- 8.7.2 Algebraic Semantics, 139
- 10.3.4 Enumerated Tokens, 210
- 10.5.3 Unique Universe of Discourse Identifiers, 215
- 10.6.2 Atomic Entities, 216
- 12.2.4 Property-Orientedness, 240
- 12.5.1 Model-Oriented Specification, 254
- 12.5.1 Property-Oriented Specification, 254
- 12.5.5 Property-Oriented vs Model-Oriented Specifications, 257
- 13.7 Set Abstraction and Modelling, 288
- 14.6.2 Cartesian Abstraction and Modelling, 315
- 15.6 List Abstraction and Modelling, 342
- 16.6 Pointer-Based Data Structures, 387
- 16.6 Type Invariance, 387
- 16.6 Types versus Values, 387
- 16.6 Map Abstraction and Modelling, 386
- 17.5 Functions as Denotations, 407
- 17.5 Function Abstraction and Modelling, 407
- 18.11.2 Type Abstraction and Modelling, 423
- 19.7.2 Binding Contexts, 454
- 19.7.2 Typed Values, 454
- 20.3.3 First Semantics then Syntax, 479
- 20.3.4 Type, Value and Location Homomorphisms, 480
- 20.5.4 Applicative to Imperative Function Translation, 495
- 21.2.5 Process Modelling, 521

Techniques

- 7.9.4 λ -Conversion, 123
- 8.7.2 Algebra Construction, 139
- 10.3.4 Enumerated Tokens, 210

- 10.5.3 Unique Universe of Discourse Identifiers, 215
- 10.6.2 Atomic Entities, 217
- 12.2.4 Property-Orientedness, 240
- 12.5.2 Property-Oriented Specifications, 255
- 12.5.3 Model-Oriented Specifications, 256
- 13.7 Set Abstraction and Modelling, 288
- 14.6.2 Cartesian Abstraction and Modelling, 316
- 15.6 List Abstraction and Modelling, 343
- 16.6 Map Abstraction and Modelling, 387
- 17.5 Function Abstraction and Modelling, 407
- 18.11.2 Type Abstraction and Modelling, 424
- 20.6.1 Applicative Contexts vs. State Function Argument, 499
- 20.6.1 Applicative Contexts, 499
- 20.6.1 Applicative States, 499
- 20.6.2 Imperative States, 502
- 20.6.3 Imperative Block-Structured Contexts (I), 504
- 20.6.3 Imperative Block-Structured Contexts (II), 505
- 20.6.3 Imperative Block-Structured Contexts (III), 505

Tools

- 7.9.4 The λ -calculus, 123
- 8.7.2 Algebra, 139
- 10.3.4 Enumerated Tokens, 210
- 10.5.3 Unique Universe of Discourse Identifiers, 215
- 12.2.4 Property-Orientedness, 240
- 13.7 Set Abstraction and Modelling, 288
- 14.6.2 Cartesian Abstraction and Modelling, 316
- 15.6 List Abstraction and Modelling, 343
- 16.6 Map Abstraction and Modelling, 388
- 17.5 Function Abstraction and Modelling, 408
- 18.11.2 Type Abstraction and Modelling, 424

L.2.2 Volume 2

Principles

- 2.2.2 Compositional Development, 39
- 2.2.2 Development and Presentation, 38
- 2.2.2 Development, 38
- 2.2.2 Hierarchical Development, 38
- 2.2.2 Presentation, 38
- 2.3.3 From Phenomena to Concepts, 46
- 2.4 Choosing Compositional Development and/or Presentation, 49
- 3.2 Denotational Semantics, 57
- 3.2.5 Denotational Semantics, 73

- 3.4 Denotations versus Computations, 86
- 4.4.2 Context and State, 103
- 4.9 Configurations — Contexts and States, 116
- 6.4.2 Pragmatics (I), 148
- 6.4.2 Pragmatics (II), 148
- 7.9.2 Semantics (I), 169
- 7.9.2 Semantics (II), 169
- 8.2.2 Types of Atomic Value Names, 177
- 8.9.2 Syntax (I), 204
- 8.9.2 Syntax (II), 204
- 8.9.2 Syntax (III), 204
- 9.5.6 Physical Systems, 233
- 9.6.2 Semiotics, 234
- 10.4.2 Modularisation, 281
- 11.3.11 Finite State Automata, 299
- 11.4.5 Finite State Machines, 307
- 11.5.3 pushdown stack device, 310
- 13.7.2 Choosing Sequence Charts, 468
- 14.8.2 Choosing Statechart, 509
- 15.6.2 Quantitative Models of Time, 567
- 16.11.2 Functional Programming Language Implementation, 654
- 17.5.2 Imperative Programming Language Implementation, 668
- 18.4.2 Modular Programming Language Implementation, 679
- 19.7.2 Parallel Programming Language Definition, 703

Techniques

- 2.2.2 Composition Development, 39
- 2.2.2 Hierarchy Development, 39
- 2.3.3 From Phenomena to Concepts, 47
- 3.2.5 Direct and Continuation Semantics, 73
- 4.4.2 Context Design, 103
- 4.4.2 State Design, 103
- 4.6.2 Process Context, 111
- 4.6.2 Process State, 111
- 4.9 Configurations — Contexts and States, 116
- 6.4.2 Pragmatics, 148
- 7.9.2 Semantics, 169
- 8.2.2 Types of Atomic Value Names, 178
- 8.9.2 Syntax, 204
- 9.5.6 Physical Systems, 233
- 10.4.2 Modularisation, 282
- 11.3.11 Finite State Automata, 300
- 11.4.5 Finite State Machine, 307
- 11.5.3 pushdown stack device, 311

- 13.7.2 Creating Sequence Charts, 469
- 14.8.2 Statechart, 509
- 15.6.2 Quantitative Models of Time, 567
- 16.11.2 Functional Programming Language Implementation, 655
- 17.5.2 Imperative Programming Language Implementation, 669
- 18.4.2 Modular Programming Language Implementations, 679
- 19.7.2 Parallel Programming Language Definitions, 703

Tools

- 7.9.2 Semantics, 169
- 8.9.2 Syntax, 204
- 10.4.2 Modularisation, 282
- 13.7.2 Sequence Charts, 469
- 13.7.2 Play-Engine, LSC, 469
- 15.6.2 Quantitative Models of Time, 567
- 15.6.2 (Ana)Tempura, ITL, 567
- 15.6.2 DCVALID, Duration Calculus, 567
- 15.6.2 TLA+, 567
- 16.11.2 Functional Programming Language Implementation, 656
- 17.5.2 Imperative Programming Language Implementation, 669
- 18.4.2 Modular Programming Language Implementations, 679
- 19.7.2 Parallel Programming Language Definitions, 703

L.2.3 Volume 3

Principles

- 2.4.10 Information Document Construction, 70
- 2.4.10 Information Documents, 70
- 2.5.1 Rough Sketching, 74
- 2.5.2 Terminologisation, 75, 77
- 2.5.3 Narration, 80
- 2.5.4 Formalisation, 83
- 2.6.5 Analysis Document, 88
- 2.6.5 Analysis, 87
- 2.7.2 Documentation, 89
- 3.4.1 Development Choice, 99
- 3.4.1 Methodicity, 99
- 3.4.2 A Principle of Types, 100
- 3.4.2 Algebra, 102
- 3.4.2 Functions, 100
- 3.4.2 Logic, 103
- 3.4.2 Relations, 101
- 4.2.1 Analogic Models, 108
- 4.2.1 Analytic Models, 109

- 4.2.1 Iconic Models, 109
- 4.2.2 Descriptive Models, 112
- 4.2.2 Prescriptive Model, 112
- 4.2.3 Extensional Models, 114
- 4.2.3 Intensional Models, 115
- 4.3 Model Role, 116
- 4.4 Modelling, 116
- 5.3.13 Entities, 137
- 5.6.3 Phenomena and Concepts, 152
- 6.4.2 Definition/Unique Recognition, 166
- 6.6.2 Characterisation and Definition, 169
- 7.2.4 Choice of Description Style, 183
- 7.2.4 Conceptual Framework, 181
- 7.2.5 Type Versus Value (Instantiation) Modelling, 183
- 7.3.1 Definitions, 185
- 7.3.1 The Narrow Bridge, 185
- 7.3.3 Exploring Theory Bases, 186
- 9.4.2 Domain Stakeholder Perspective, 208
- 9.4.2 Domain Stakeholder, 208
- 10.2.2 Continuities, 214
- 10.2.3 Discreteness, 215
- 10.2.3 Hybridities, 220
- 10.2.4 Chaos, 221
- 10.3.1 Static Entity, 225
- 10.3.2 Autonomicity, 229
- 10.3.2 Biddable Active Dynamics, 236
- 10.3.2 Inert Dynamic Phenomena, 226
- 10.3.2 Programmable Active Dynamics, 239
- 10.3.2 Reactive Dynamics, 240
- 10.5.2 One-Dimensional Phenomena, 247
- 10.5.3 Multidimensional, 248
- 11.1.1 Separation of Facets, 253
- 11.10.1 From Big Lies via Smaller Lies to the Truth, 317
- 11.2 Describing Domain Business Process Facets, 254
- 11.2.2 Business Processes, 257
- 11.2.6 Describing Domain Business Process Facets, 263
- 11.3 Describing the Domain Intrinsic Facets, 264
- 11.3 Domain Intrinsic, 264
- 11.3.3 Intrinsic, 270
- 11.3.6 Describing the Domain Intrinsic Facets, 271
- 11.4 Describing the Domain Support Technologies Facets, 271
- 11.4.2 Support Technology, 275
- 11.4.4 Describing the Domain Support Technologies Facets, 276
- 11.5 Describing the Domain Management and Organisation Facets, 276
- 11.5.5 Management and Organisation, 281

- 11.5.7 Describing the Domain Management and Organisation Facets, 282
- 11.6 Describing the Domain Rules and Regulations Facets, 282
 - 11.6.4 Rules and Regulations, 286
 - 11.6.5 Describing the Domain Rules and Regulations Facets, 287
- 11.7 Describing the Domain Script Facets, 287
 - 11.7.3 Describing the Domain Script Facets, 308
- 11.8 Describing the Domain Human Behaviour Facet, 308
 - 11.8.5 Describing the Domain Human Behaviour Facet, 315
- 11.9 Domain Facets, 316
- 12.3.4 Domain Acquisition, 331
- 13.5.2 Concept Formation, 339
- 13.5.2 Domain Analysis, 339
- 14.4.2 Domain Validation, 348
- 14.4.2 Domain Verification, 348
- 15.7 Domain Theory, 356
- 17 Requirements Engineering, 367
 - 17.1 Requirements Adequacy, 368
 - 17.1 Requirements Implementability, 369
 - 17.1 Requirements Verifiability and Validability, 369
 - 17.1.2 Requirements, 369
- 18.4.2 Requirements Stakeholder Perspective, 386
- 18.4.2 Requirements Stakeholder, 386
- 19.8.2 Requirements Facets, 474
- 20.5.4 Requirements Acquisition, 492
- 21.5.2 Concept Formation, 500
- 21.5.2 Requirements Analysis, 499
- 22.4.2 Requirements Validation, 508
- 22.4.2 Requirements Verification, 508
- 23.6.2 Requirements Economic Feasibility, 517
- 23.6.2 Requirements Satisfiability, 517
- 23.6.2 Requirements Technical Feasibility, 517
- 25.7 Hardware/Software Codesign, 529
- 26.7.2 Component, 544
- 26.7.2 Software Architecture, 544
- 27.11.2 Component Development, Stepwise Discovery, I, 579
- 27.11.2 Component Development, Stepwise Extension, II, 579
- 27.11.2 Component Development, Stepwise Refinement, III, 580
- 28.8.2 Domain-Specific Architectures, 641

Techniques

- 2.4.10 Information Document Construction, 70
- 2.5.1 Rough Sketching, 74
- 2.5.2 Terminology Documentation, 77
- 2.5.3 Narrative Documents, 80

- 2.5.4 Formalisation, 83
- 3.4.1 Development Choice, 99
- 3.4.1 Methodicity, 99
- 3.4.2 Algebra, 102
- 3.4.2 Functions, 100
- 3.4.2 Logic, 103
- 3.4.2 Relations, 101
- 3.4.2 Types, 100
- 4.2.1 Analogic Model, 110
- 4.2.1 Analytic Model, 110
- 4.2.1 Iconic Model, 110
- 4.2.2 Narrative Descriptive Models, 112
- 4.2.2 Narrative Prescriptive Models, 113
- 4.2.3 Formal Extensional Models, 115
- 4.2.3 Formal Intensional Models, 115
- 5.3.13 Entities, 137
- 5.6.3 Phenomena and Concepts, 152
- 6.4.2 Definition/Unique Recognition, 166
- 6.6.2 Characterisation and Definition, 169
- 7.2.4 Framework Model, 181
- 7.3.1 Definitions, 185
- 7.4.1 Refutable Assertion, 187
- 7.4.3 Dangling Assertions, 189
- 9.4.2 Domain Stakeholder Liaison, 209
- 10.2.2 Continuity, 214
- 10.2.3 Discreteness, 215
- 10.2.3 Hybridicity, 220
- 10.2.4 Chaos, 222
- 10.3.1 Static Attribute, 225
- 10.3.2 Autonomicity, 230
- 10.3.2 Biddability, 236
- 10.3.2 Inert Dynamic Phenomena, 227
- 10.3.2 Programmable Active Dynamics, 239
- 10.3.2 Reactive Dynamics, 240
- 10.5.2 One-Dimensional Phenomena, 247
- 10.5.3 Multidimensional, 248
- 11.2.2 Business Processes, 257
- 11.3.3 Intrinsic, 270
- 11.4.2 Support Technology, 275
- 11.5.3 Management and Organisational, 279
- 11.5.5 Management and Organisation, 281
- 11.6.2 Rules and Regulations, 284
- 11.6.4 Rules and Regulation, 287
- 11.7.3 Domain Scripts, 308
- 11.8.2 Human Behaviour (III), 314

- 11.8.2 Human Behaviour, 313
- 12.3.4 Domain Acquisition, 331
- 13.5.2 Concept Formation, 340
- 13.5.2 Domain Analysis, 339
- 14.4.2 Domain Validation, 348
- 14.4.2 Domain Verification, 348
- 15.7 Domain Theory, 356
- 18.4.2 Requirements Stakeholder Liaison, 386
- 19.8.2 Requirements Facets, 475
- 20.5.4 Requirements Acquisition, 492
- 21.5.2 Concept Formation, 500
- 21.5.2 Requirements Analysis, 500
- 22.4.2 Requirements Validation, 508
- 22.4.2 Requirements Verification, 508
- 23.6.2 Requirements Economic Feasibility, 518
- 23.6.2 Requirements Satisfiability, 517
- 23.6.2 Requirements Technical Feasibility, 517
- 25.7 Hardware/Software Codesign, 530
- 26.7.2 Architecture Design, 544
- 27.11.2 Component Development, 580
- 28.8.2 Domain-Specific Architecture, 641

Tools

- 2.4.10 Information Document Construction, 70
- 2.5.1 Rough Sketching, 75
- 2.5.2 Terminology Documentation, 77
- 2.5.3 Narrative Documentation, 81
- 2.5.4 Formalisation Documentation, 83
- 3.4.2 Algebra, 102
- 3.4.2 Functions, 100
- 3.4.2 Logic, 103
- 3.4.2 Relations, 101
- 3.4.2 Types, 100
- 5.3.13 Entities, 138
- 5.6.3 Modelling Phenomena and Concepts, 153
- 6.6.2 Characterisation and Definition, 168
- 9.4.2 Domain Stakeholder Liaison, 209
- 10.3.2 Reactive Dynamics Phenomena, 240
- 11.2.2 Business Processes, 257
- 11.7.3 Domain Scripts, 308
- 12.3.4 Domain Acquisition, 332
- 13.5.2 Concept Formation, 340
- 13.5.2 Domain Analysis, 340
- 14.4.2 Domain Validation, 348
- 14.4.2 Domain Verification, 348

- 15.7 Domain Theory, 356
- 18.4.2 Requirements Stakeholder Liaison, 386
- 19.8.2 Requirements Facets, 475
- 20.5.4 Requirements Acquisition, 492
- 21.5.2 Concept Formation, 500
- 21.5.2 Requirements Analysis, 500
- 22.4.2 Requirements Validation, 508
- 22.4.2 Requirements Verification, 509
- 29.5.1 Theorem Provers: Isabelle/HOL, PVS, 657
- 29.5.1 Proof Assistant Raise, 657
- 29.5.2 Model Checkers: FDR2, SMV, Spin, 658

M

List of Papers on Verification

For a good introduction to a number of leading approaches to software verification we refer to the following papers:

1. J. U. Skakkebæk, A. P. Ravn, H. Rischel, and Zhou Chaochen. *Specification of embedded, real-time systems*. Proceedings of 1992 Euromicro Workshop on Real-Time Systems, pages 116–121. IEEE Computer Society Press, 1992.
2. Zhou Chaochen, M. R. Hansen, A. P. Ravn, and H. Rischel. *Duration specifications for shared processors*. Proceedings Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems, Nijmegen 6-10 Jan. 1992, LNCS, 1992.
3. A. P. Ravn, H. Rischel, and K. M. Hansen. *Specifying and verifying requirements of real-time systems*. IEEE Trans. Software Engineering, 19:41–55, 1992.
4. C. W. George. *A theory of distributing train rescheduling*. In FME'96: Industrial Benefits and Advances in Formal Methods, proceedings, LNCS 1051, 1996.
5. C. W. George. *Proving safety of authentication protocols: a minimal approach*, in International Conference on Software: Theory and Practice (ICS 2000), 2000.
6. A. Haxthausen and X. Yong. *Linking DC together with TRSL*. Proceedings of 2nd International Conference on Integrated Formal Methods (IFM 2000), Schloss Dagstuhl, Germany, November 2000, number 1945 in Lecture Notes in Computer Science, pages 25–44. Springer-Verlag, 2000.
7. A. Haxthausen and J. Peleska, *Formal development and verification of a distributed railway control system*, IEEE Transaction on Software Engineering, 26(8), 687–701, 2000.
8. M. P. Lindegaard, P. Viuf and A. Haxthausen, *Modelling railway interlocking systems*, Eds.: E. Schnieder and U. Becker, Proceedings of the 9th IFAC Symposium on Control in Transportation Systems 2000, June 13–15, 2000, Braunschweig, Germany, 211–217, 2000.

9. A. E. Haxthausen and J. Peleska, *A domain specific language for railway control systems*, Sixth Biennial World Conference on Integrated Design and Process Technology, (IDPT 2002), Pasadena, California, Society for Design and Process Science, P. O. Box 1299, Grand View, Texas 76050-1299, USA, June 23-28, 2002.
10. A. Haxthausen and T. Gjaldbæk, *Modelling and verification of interlocking systems for railway lines*, 10th IFAC Symposium on Control in Transportation Systems, Tokyo, Japan, August 4-6, 2003.

N

Biography

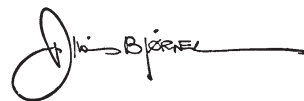
- **Family &c.:** Dines Bjørner (DB) was born in Odense, Denmark, 4 October 1937. His father had an MSc degree in Mathematics (from Copenhagen University, 1931) and his mother a BA degree in Nordic and Modern English/America Literature (also from Copenhagen University, 1929). Since 1965 DB has been married to Kari Skallerud Bjørner (Oslo, Norway). They have two children, Charlotte and Nikolaj, and five grandchildren.
- **Educational Background:** DB graduated, in 1956, with a senior high school degree in Mathematics and Natural Sciences from the Århus Cathedral School (founded in 1142). DB graduated in January 1962 with an MSc in Electronics Engineering and with a Ph.D. in Computer Science in January 1969 from the Technical University of Denmark (founded by Hans Christian Ørsted in 1828).
- **IBM Career:** DB joined IBM in March 1962 at their Nordic Laboratories (founded by Cai Kinberg) in Stockholm, Sweden (where DB also first met Jean Paul Jacob and Gunnar Wedell). DB was transferred to the IBM Systems Development Division (IBM SDD) at San Jose, California, USA, in December 1963. While doing his Ph.D. (September 1965–January 1969) DB was a lecturing consultant to IBM's European Systems Research Institute (ESRI) at Geneva, Switzerland (where DB received valuable guidance from Carlo Santacrose and where DB's friendship with Gerald Weinberg started) (1967–1968). In 1969 DB worked at IBM's Advanced Computing Systems (IBM ACS) Laboratory, Menlo Park, California, and, later that year until early 1973 at IBM Research, San Jose (again Jean Paul Jacob became a colleague). Transferred to the IBM Vienna Laboratory (directed then by Heinz Zemanek), Austria, DB resigned from IBM in August 1975 to return to Denmark after basically 13 years abroad.
- **Career Outside and After IBM:** During his stay at IBM Research DB was a visiting lecturer, for several quarters, at University of California at Berkeley (1971–1972), instigated by Lotfi Zadeh whom DB considers his main mentor and for whom DB has the fondest regards. DB was a visiting guest professor at Copenhagen University in the academic year 1975–1976, before taking up his present chair in September 1976 at the Technical University of Denmark (DTU). During the summer semester of 1980 DB was the Danish Chair Professor at the Christian-Albrechts University of Kiel, Germany — hosted by Prof. Dr. Hans Langmaack. Together with a colleague, Prof. Christian Gram, DB instigated the Dansk Datamatik Center (DDC) in the summer of 1979. During the 1980s DB was chief scientist of DDC. In 1982–

1984 DB was chairman of a Danish Government (Ministry of Education) Commission on Informatics. DB was the founding and first UN Director of UNU-IIST, the United Nations University's International Institute for Software Technology, located in Macau. DB was a visiting professor at NUS: National University of Singapore in the academic year 2004–2005, and a research guest professor at JAIST, Japan Advanced Institute of Science and Technology, Ishikawa Prefecture, Japan for basically the calendar year 2006 — where the initial writing of this thesis was begun. DB was a visiting professor at Université Henri Poincaré and at INRIA/LORIA, Nancy, France, for two months: Oct.–Dec., 2007. During the fall, winter, spring and summer of 2008–2009 DB will be lecturing (i) at the Techn. Univ. of Graz, Austria (5–6 weeks, Oct.–Dec.), (ii) at the Politecnico di Milano, Italy (Feb., 2009), (iii) University of Saarland, Saarbrücken, Germany (March 2009) and (iv) is scheduled to occupy the “Danish Chair” at Christian–Albrechts University of Kiel, Germany (summer semester: April–Sept., 2009).

- **Lectures and Graduates:** DB has lectured and regularly lectures on six continents in almost 50 countries and territories and has advised more than 130 MSc's and almost two dozen PhDs.
- **Research &c. Work:** At IBM DB first worked in the hardware (logic and systems) design of such equipment as the IBM 1070 (Sweden), the IBM 1800 and IBM 1130 computers (San Jose), and, finally, with Gene Amdahl and Ed Sussenguth, on the IBM ACS/1 supercomputer (Menlo Park). At Research DB worked with the late John W. Backus and the late Ted Codd on Functional Languages, resp. Relational Data Base Systems. At Vienna, DB, together with such colleagues as Peter Lucas, the late Hans Bekič, Kurt Walk, and Cliff B. Jones, worked on a Denotational (–like) Semantics Description of PL/I while, with his colleagues conceiving, researching, developing and using VDM (the Vienna software Development Method). At DTU and at DDC, supported by the European Community, DB initiated several advanced research & development projects: (1) Formal Semantics Description of and (2) full language compiler for CHILL (the Intl. Telecommunications Unions Communications [C.C.I.T.T.] High Level Language) — both significantly developed by Peter Hall (and Søren Prehn), (3) Formal Semantics Description of and (4) the first European US DoD officially validated compiler for the US DoD Ada embedded systems programming language — with significant and indispensable contributions by my colleague Dr. Hans Bruun and, again, Søren Prehn, (5) RAISE (Rigorous Approach to Industrial Software Engineering, headed by Søren Prehn and Chris George), (6) Formal Semantics Definition of VDM–SL (the VDM Specification Language, Bo Stig Hansen and Peter Gorm Larsen), (7) ProCoS (Provably Correct Systems) with, amongst others Profs. Sir Tony Hoare (then Oxford, now Microsoft Research, Cambridge, UK), Hans Langmaack (Kiel) and Ernst–Rüdiger Olderog (Oldenburg), Anders P. Ravn, Hans Rischel, and others, etc.
- **UNU-IIST:** At UNU-IIST DB had a rather free hand, and was able, with a small team of excellent colleagues (Prof. Zhou Chaochen (Academician, the Chinese Academy of Science), Søren Prehn, Chris W. George, Richard Moore, Tomasz Janowski, Dang Van Hung, Xu Qi Wen and Kees Middelburg), to further explore the research issues still occupying DB's interest, and to apply them (i.e., test them out) in a number of joint R&D projects with institutions in developing and newly industrialised countries [including newly independent states] (Argentina, Belarus, Brasil, Cameroun, China, Gabon, India, Indonesia, Mongolia, North Korea, Pakistan, Philippines, Poland, Ro-

mania, Russia, South Africa, South Korea, Thailand, Vietnam, Ukraine, Uruguay, etc.).

- **Societal Work:** DB was a co-founder of VDM-Europe in 1987 and moved VDM-Europe onto FME: Formal Methods Europe in 1991. DB co-chaired two of the VDM Symposia (1987, 1990), and the International Conference on Software Engineering (ICSE) in 1989 in Pittsburgh, Pennsylvania, USA. DB was chairman of the IFIP World Congress in Dublin, Ireland in 1986, and was the instigator and General Chairman of the first World Congress on Formal Methods, FM'99, in Toulouse, France, September 20–24, 1999. DB has otherwise been involved in about 60 other scientific conferences.
- **Awards &c.:** DB is a Knight of The Danish Flag; is a member of Academia Europaea (MAE) and is the chairman of its Informatics Section; member of The Russian Academy of Natural Sciences (MRANS [AB]), and of IFIP Working Groups 2.2 (resigned) and 2.3 (resigned summer 2008 due to lack of travel funds). DB has received the John von Neumann Medal of the JvN Society of Hungary and the Ths. Masaryk Gold Medal from the Masaryk University, Brno, The Czech Republic. DB received the Danish Engineering Society's (IDA) Informatics Division's (IDA-IT) first BIT prize, March 1999. DB was given the degree of honorary doctor from the Masaryk University, Brno, The Czech Republic, in 2004. DB is an ACM Fellow and an IEEE Fellow.
- **Publications:** DB has authored more than 100 published papers and co-authored and co-edited some 15 books and written three books [87–89].
- **Research Interests:** DB's research interests, since his Vienna days, have centered on programming methodology: Methods as sets of principles for selecting and applying mathematics-based analysis and construction techniques and tools in order efficiently to construct efficient artefacts — notably software. DB sees his main contributions to be in the research, development and propagation of formal specification principles and techniques. Currently DB focuses on the triptych of domain engineering, requirements engineering and software architecture and program organisation methods — emphasising such that relate these in mathematical as well as technical ways: (1) Intrinsic, support technology, management & organisation, rules & regulation, and human behaviour facets of domains; (2) projection, instantiation, extension and initialisation of domain requirements, etc.; (3) software architectures as refinements of domain requirements, and program organisation as refinements of machine requirements — with interface requirements (currently) being refinements of either and both!
- **Acknowledgements:** Among the very many people who have been kind to DB and helped DB in his professional career, he wishes to bear tribute, in approximate chronological order, to (the late) Cai Kinberg, Gunnar Wedell, Jean Paul Jacob, Gerald Weinberg, Carlo Santacrose, Gene Amdahl, Ed Sussenguth, Tien Chi (T.C.) Chen, Lotfi Zadeh, (the late) Ted Codd, (the late) John W. Backus, Peter Lucas, Cliff Jones, (the late) Hans Bekič, Kurt Walk, Christian Gram, Ole N. Oest, Erich Neuhold, (the late) Søren Prehn, Sir Tony Hoare, Hans Langmaack, Zhou Chao Chen and Chris George.



Technical University of Denmark – August 1, 2008

O

References

1. J.-R. Abrial: *The B Book: Assigning Programs to Meanings* (Cambridge University Press, Cambridge, England 1996)
2. ACM: *Programming Languages and Pragmatics*. Communications of the ACM **9**, 6 (1966)
3. W. Aitken, B. Dickens, P. Kwiatkowski et al: Transformation in intentional programming. In: *Fifth International Conference on Software Reuse, ICSR'98* (Victoria, Canada)
4. Anon: *C.C.I.T.T. High Level Language (CHILL), Recommendation Z.200, Red Book Fascicle VI.12* (ITU (Intl. Telecomm. Union), Geneva, Switzerland 1980 – 1985)
5. K.R. Apt: *Principles of Constraint Programming* (Cambridge University Press, August 2003)
6. K. Araki, A. Galloway, K. Taguchi, editors. *IFM 1999: Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, York, UK, June 1999. Springer. Proceedings of 1st Intl. Conf. on IFM.
7. Y. Arimoto, D. Bjørner: Hospital Healthcare: A Domain Analysis and a License Language. Technical Note, JAIST, School of Information Science, 1-1, Asahidai, Tatsunokuchi, Nomi, Ishikawa, Japan 923-1292 (2006)
8. A. Arnab, A. Hutchison: Fairer Usage Contracts for DRM. In: *Proceedings of the Fifth ACM Workshop on Digital Rights Management (DRM'05)* (2005) pp 65–74
9. R.-J. Back, J. von Wright: *Refinement Calculus: A Systematic Introduction* (Springer-Verlag, Heidelberg, Germany 1998)
10. M. Balaguer: *Platonism and Anti-Platonism in Mathematics* (Oxford University Press, 1998)
11. Barry Boehm: *The Spiral Model for Software Development and Enhancement*. IEEE Computer **21**, 5 (1988) pp 61–72
12. D. Batory: Intelligent Components and Software Generators. Technical Report, University of Texas at Austin, Austin, TX, USA (1997)
13. D. Batory. *Feature Modularity in Software Product Lines*. Lecture Notes in Computer Science. Springer, Heidelberg, Tutorials. The Lipari Summer School, July 2007, on Advances in Software Engineering, eds. Egon Börger and Alfredo Ferro [To appear] 2008.

14. D. Batory, J. Liu, J. Sarvela: *Refinements and Multi Dimensional Separation of Concerns*. ACM SIGSOFT (2003)
15. D. Batory, R. Lopez-Herrejon, J.-P. Martin: *Generating Product-Lines of Product Families*. Automated Software Engineering (2002)
16. D. Batory, S. O'Malley: *The design and implementation of hierarchical software systems with reusable components*. ACM Trans. Softw. Eng. Methodol. **1**, 4 (1992) pp 355–398
17. D. Batory, V. Singhal, M. Sirkin, J. Thomas: *Scalable software libraries*. SIGSOFT Softw. Eng. Notes **18**, 5 (1993) pp 191–199
18. D. Batory, S. Thaker: Towards Safe Composition of Product Lines. Department of Computer Sciences University of Texas at Austin Austin, Texas, 78712 U.S.A. 2007 (2007)
19. E.J. Baude: *Software Design: From Programming to Architecture* (Wiley, 2003)
20. K. Beck: *Test-driven development* (Addison-Wesley, Boston, MA 2003)
21. K. Beck: *Extreme Programming Explained: Embrace Change* (Addison-Wesley, (October 5, 1999)
22. K. Beck, M. Fowler: *Planning Extreme Programming* (Addison-Wesley, October 13, 2000)
23. H. Bekič, D. Bjørner, W. Henhapl, C.B. Jones, P. Lucas: A Formal Definition of a PL/I Subset. Technical Report 25.139, Vienna, Austria (1974)
24. M. Bidoit, P.D. Mosses: *CASL User Manual* (Springer, 2004)
25. D. Bjørner: Programming Languages: Formal Development of Interpreters and Compilers. In: *International Computing Symposium 77* (North-Holland Publ.Co., Amsterdam, 1977) pp 1–21
26. D. Bjørner: Programming Languages: Linguistics and Semantics. In: *International Computing Symposium 77* (North-Holland Publ.Co., Amsterdam, 1977) pp 511–536
27. D. Bjørner: Programming in the Meta-Language: A Tutorial. In: *The Vienna Development Method: The Meta-Language, [111]*, ed by D. Bjørner, C.B. Jones (Springer-Verlag, 1978) pp 24–217
28. D. Bjørner: Software Abstraction Principles: Tutorial Examples of an Operating System Command Language Specification and a PL/I-like On-Condition Language Definition. In: *The Vienna Development Method: The Meta-Language, [111]*, ed by D. Bjørner, C.B. Jones (Springer-Verlag, 1978) pp 337–374
29. D. Bjørner: The Systematic Development of a Compiling Algorithm. In: *Le Point sur la Compilation*, ed by Amirchahy, Neel (INRIA Publ. Paris, 1979) pp 45–88
30. D. Bjørner: The Vienna Development Method: Software Abstraction and Program Synthesis. In: *Mathematical Studies of Information Processing*, vol 75 of *LNCS* (Springer-Verlag, 1979)
31. Edited by D. Bjørner: *Abstract Software Specifications*, vol 86 of *LNCS* (Springer-Verlag, 1980)
32. D. Bjørner: Application of Formal Models. In: *Data Bases* (INFOTECH Proceedings, 1980)
33. D. Bjørner: Experiments in Block-Structured GOTO-Modelling: Exits vs. Continuations. In: *Abstract Software Specification, [31]*, vol 86 of *LNCS*, ed by D. Bjørner (Springer-Verlag, 1980) pp 216–247

34. D. Bjørner: Formal Description of Programming Concepts: a Software Engineering Viewpoint. In: *MFCS'80, Lecture Notes Vol. 88* (Springer-Verlag, 1980) pp 1–21
35. D. Bjørner: Formalization of Data Base Models. In: *Abstract Software Specification, [31]*, vol 86 of *LNCS*, ed by D. Bjørner (Springer-Verlag, 1980) pp 144–215
36. D. Bjørner: The VDM Principles of Software Specification and Program Design. In: *TC2 Work.Conf. on Formalisation of Programming Concepts, Peniscola, Spain* (Springer-Verlag, LNCS Vol. 107 1981) pp 44–74
37. D. Bjørner: Realization of Database Management Systems. In: *See [112]* (Prentice-Hall, 1982) pp 443–456
38. D. Bjørner: Rigorous Development of Interpreters and Compilers. In: *See [112]* (Prentice-Hall, 1982) pp 271–320
39. D. Bjørner: Stepwise Transformation of Software Architectures. In: *See [112]* (Prentice-Hall, 1982) pp 353–378
40. D. Bjørner: Software Architectures and Programming Systems Design. Vols. I–VI. Techn. Univ. of Denmark (1983–1987)
41. D. Bjørner: Project Graphs and Meta-Programs: Towards a Theory of Software Development. In: *Proc. Capri '86 Conf. on Innovative Software Factories and Ada, Lecture Notes on Computer Science*, ed by N. Habermann, U. Montanari (Springer-Verlag, 1986)
42. D. Bjørner: Software Development Graphs — A Unifying Concept for Software Development? In: *Vol. 241 of Lecture Notes in Computer Science: Foundations of Software Technology and Theoretical Computer Science*, ed by K. Nori (Springer-Verlag, 1986) pp 1–9
43. D. Bjørner: *Software Engineering and Programming: Past-Present-Future*. IPSJ: Inform. Proc. Soc. of Japan **8**, 4 (1986) pp 265–270
44. D. Bjørner: On The Use of Formal Methods in Software Development. In: *Proc. of 9th International Conf. on Software Engineering, Monterey, California* (1987) pp 17–29
45. D. Bjørner: The Stepwise Development of Software Development Graphs: Meta-Programming VDM Developments. In: *See [113]*, vol 252 of *LNCS* (Springer-Verlag, Heidelberg, Germany, 1987) pp 77–96
46. D. Bjørner: *Facets of Software Development: Computer Science & Programming, Engineering & Management*. J. of Comput. Sci. & Techn. **4**, 3 (1989) pp 193–203
47. D. Bjørner: Specification and Transformation: Methodology Aspects of the Vienna Development Method. In: *TAPSOFT'89*, vol 352 of *Lab. Note* (Springer-Verlag, Heidelberg, Germany, 1989) pp 1–35
48. D. Bjørner: Formal Software Development: Requirements for a CASE. In: *European Symposium on Software Development Environment and CASE Technology, Königswinter, FRG, June 17–21* (Springer-Verlag, Heidelberg, Germany, 1991)
49. D. Bjørner: Formal Specification is an Experimental Science (in English). In: *Intl. Conf. on Perspectives of System Informatics* (1991)
50. D. Bjørner: *Formal Specification is an Experimental Science (in Russian)*. Programmirovanie **6** (1991) pp 24–43
51. D. Bjørner: Towards a Meaning of ‘M’ in VDM. In: *Formal Description of Programming Concepts*, ed by E. Neuhold, M. Paul (Springer-Verlag, Heidelberg, Germany, 1991) pp 137–258

52. D. Bjørner: From Research to Practice: Self-reliance of the Developing World through Software Technology: Usage, Education & Training, Development & Research. In: *Information Processing '92, IFIP World Congress '92, Madrid*, ed by J. van Leeuwen (IFIP Transaction A-12: Algorithms, Software, Architecture, 1992) pp 65–71
53. D. Bjørner: Trustworthy Computing Systems: The ProCoS Experience. In: *14'th ICSE: Intl. Conf. on Software Eng., Melbourne, Australia* (ACM Press, 1992) pp 15–34
54. D. Bjørner. *Formal Models of Robots: Geometry & Kinematics*, chapter 3, pages 37–58. Prentice-Hall International, January 1994. Eds.: W.Roscoe and J.Woodcock, *A Classical Mind*, Festschrift for C.A.R. Hoare.
55. D. Bjørner: Prospects for a Viable Software Industry — Enterprise Models, Design Calculi, and Reusable Modules. In: *First ACM Japan Chapter Conference* (World Scientific Publ, Singapore 1994)
56. D. Bjørner: Software Systems Engineering — From Domain Analysis to Requirements Capture: An Air Traffic Control Example. In: *2nd Asia-Pacific Software Engineering Conference (APSEC '95)* (IEEE Computer Society, 1995)
57. D. Bjørner: From Domain Engineering via Requirements to Software. Formal Specification and Design Calculi. In: *SOFSEM'97*, vol 1338 of *Lecture Notes in Computer Science* (Springer-Verlag, 1997) pp 219–248
58. D. Bjørner: Michael Jackson's Problem Frames: Domains, Requirements and Design. In: *ICFEM'97: International Conference on Formal Engineering Methods*, ed by L. ShaoYang, M. Hinchley (1997)
59. D. Bjørner: Challenges in Domain Modelling — Algebraic or Otherwise. Research, Department of Information Technology, Software Systems Section, Technical University of Denmark, DK-2800 Lyngby, Denmark (1998)
60. D. Bjørner: Domains as Prerequisites for Requirements and Software &c. In: *RTSE'97: Requirements Targeted Software and Systems Engineering*, vol 1526 of *Lecture Notes in Computer Science*, ed by M. Broy, B. Rumpe (Springer-Verlag, Berlin Heidelberg 1998) pp 1–41
61. D. Bjørner: Formal Methods in the 21st Century — An Assessment of Today, Predictions for The Future — Panel position presented at the ICSE'98, Kyoto, Japan. Technical Report, Department of Information Technology, Software Systems Section, Technical University of Denmark (1998)
62. D. Bjørner: Issues in International Cooperative Research — Why not Asian, African or Latin American 'Esprits' ? Research, Department of Information Technology, Software Systems Section, Technical University of Denmark, DK-2800 Lyngby, Denmark (1998)
63. D. Bjørner: A Triptych Software Development Paradigm: Domain, Requirements and Software. Towards a Model Development of A Decision Support System for Sustainable Development. In: *Festschrift to Hans Langmaack: Correct Systems Design: Recent Insight and Advances*, vol 1710 of *Lecture Notes in Computer Science*, ed by E.-R. Olderog, B. Steffen (Springer-Verlag, 1999) pp 29–60
64. D. Bjørner: Challenge '2000: some aspects of: "How to Create a Software Industry". In: *Proceedings of CSIC'99, Ed.: R. Jalili* (1999)
65. D. Bjørner: *Where do Software Architectures come from ? Systematic Development from Domains and Requirements. A Re-assessment of Software Engineering ?* South African Journal of Computer Science **22** (1999) pp 3–13

66. D. Bjørner: Domain Engineering, A Software Engineering Discipline in Need of Research. In: *SOFSEM'2000: Theory and Practice of Informatics*, vol 1963 of *Lecture Notes in Computer Science* (Springer Verlag, Milovy, Czech Republic 2000) pp 1–17
67. D. Bjørner: Domain Modelling: Resource Management Strategics, Tactics & Operations, Decision Support and Algorithmic Software. In: *Millenial Perspectives in Computer Science*, ed by J. Davies, B. Roscoe, J. Woodcock (Palgrave (St. Martin's Press), Houndmills, Basingstoke, Hampshire, RG21 6XS, UK 2000) pp 23–40
68. D. Bjørner: Formal Software Techniques in Railway Systems. In: *9th IFAC Symposium on Control in Transportation Systems*, ed by E. Schnieder (2000) pp 1–12
69. D. Bjørner: Informatics: A Truly Interdisciplinary Science — Computing Science and Mathematics. In: *9th Intl. Colloquium on Numerical Analysis and Computer Science with Applications*, ed by D. Bainov (Academic Publications, P.O.Box 45, BG-1504 Sofia, Bulgaria 2000)
70. D. Bjørner: Informatics: A Truly Interdisciplinary Science — Prospects for an Emerging World. In: *Information Technology and Communication — at the Dawn of the New Millenium*, ed by S. Balasubramanian (2000) pp 71–84
71. D. Bjørner: *Pinnacles of Software Engineering: 25 Years of Formal Methods*. *Annals of Software Engineering* **10** (2000) pp 11–66
72. D. Bjørner: Informatics Models of Infrastructure Domains. In: *Computer Science and Information Technologies* (Institute for Informatics and Automation Problems, Yerevan, Armenia 2001) pp 13–73
73. D. Bjørner: On Formal Techniques in Protocol Engineering: Example Challenges. In: *Formal Techniques for Networks and Distributed Systems* (Eds.: Myungchul Kim, Byoungmoon Chin, Sungwon Kang and Danhyung Lee) (Kluwer, 2001) pp 395–420
74. D. Bjørner: Domain Models of “The Market” — in Preparation for E-Transaction Systems. In: *Practical Foundations of Business and System Specifications* (Eds.: Haim Kilov and Ken Baclawski) (Kluwer Academic Press, The Netherlands 2002)
75. D. Bjørner: Some Thoughts on Teaching Software Engineering – Central Rôles of Semantics. In: *Liber Amicorum: Professor Jaco de Bakker* (Stichting Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands 2002) pp 27–45
76. D. Bjørner: Domain Engineering: A “Radical Innovation” for Systems and Software Engineering ? In: *Verification: Theory and Practice*, vol 2772 of *Lecture Notes in Computer Science* (Springer-Verlag, Heidelberg 2003)
77. D. Bjørner: Dynamics of Railway Nets: On an Interface between Automatic Control and Software Engineering. In: *CTS2003: 10th IFAC Symposium on Control in Transportation Systems* (Elsevier Science Ltd., Oxford, UK 2003)
78. D. Bjørner: Logics of Formal Software Specification Languages — The Possible Worlds cum Domain Problem. In: *Fourth Pan-Hellenic Symposium on Logic*, ed by L. Kirousis (2003)
79. D. Bjørner: New Results and Trends in Formal Techniques for the Development of Software for Transportation Systems. In: *FORMS2003: Symposium on Formal Methods for Railway Operation and Control Systems* (Institut für Verkehrssicherheit und Automatisierungstechnik, Techn.Univ. of Braunschweig, Germany, 2003)

80. D. Bjørner. “What is a Method ?” — *An Essay of Some Aspects of Software Engineering*, chapter 9, pages 175–203. Monographs in Computer Science. IFIP: International Federation for Information Processing. Springer Verlag, New York, N.Y., USA, 2003. Programming Methodology: Recent Work by Members of IFIP Working Group 2.3. Eds.: Annabelle McIver and Carrol Morgan.
81. D. Bjørner: What is an Infrastructure ? In: *Formal Methods at the Crossroads. From Panacea to Foundational Support* (Springer-Verlag, Heidelberg, Germany 2003)
82. D. Bjørner: The Grand Challenge – FAQs of the R&D of a Railway Domain Theory. In: *IFIP World Computer Congress, Topical Days: TRain: The Railway Domain* (Kluwer Academic Press, Amsterdam, The Netherlands 2004)
83. D. Bjørner: The TRain Topical Day. In: *Building the Information Society, IFIP 18th World Computer Congress, Tpic Sessions, 22–27 August, 2004, Toulouse, France* — Ed. Renéne Jacquart (Kluwer Academic Publishers, 2004) pp 607–611
84. D. Bjørner: Towards a Formal Model of CyberRail. In: *Building the Information Society, IFIP 18th World Computer Congress, Tpic Sessions, 22–27 August, 2004, Toulouse, France* — Ed. Renéne Jacquart (Kluwer Academic Publishers, 2004) pp 657–664
85. D. Bjørner: Towards “Posite & Prove” Design Calculi for Requirements Engineering and Software Design. In: *Essays and Papers in Memory of Ole-Johan Dahl* (Springer-Verlag, 2004)
86. D. Bjørner: Documents: A Domain Analysis. Technical Note, JAIST, School of Information Science, 1-1, Asahidai, Tatsunokuchi, Nomi, Ishikawa, Japan 923-1292 (2006)
87. D. Bjørner: *Software Engineering, Vol. 1: Abstraction and Modelling* (Springer, 2006)
88. D. Bjørner: *Software Engineering, Vol. 2: Specification of Systems and Languages* (Springer, 2006)
89. D. Bjørner: *Software Engineering, Vol. 3: Domains, Requirements and Software Design* (Springer, 2006)
90. D. Bjørner: Domain Theory: Practice and Theories, Discussion of Possible Research Topics. In: *ICTAC’2007*, vol 4701 of *Lecture Notes in Computer Science* (eds. J.C.P. Woodcock et al.) (Springer, Heidelberg 2007) pp 1–17
91. D. Bjørner: Transportation Systems Development. In: *2007 ISoLA Workshop On Leveraging Applications of Formal Methods, Verification and Validation; Special Workshop Theme: Formal Methods in Avionics, Space and Transport* (2007)
92. D. Bjørner: A Container Line Industry Domain. Technical Report (incomplete, draft), Technical University of Denmark, Institute of Informatics and Mathematical Modelling (2007. www.imm.dtu.dk/~db/container-paper.pdf)
93. D. Bjørner: *Believable Software Management*. Encyclopedia of Software Engineering **1**, 1 (2008) pp 1–32
94. D. Bjørner: Domain Engineering. In: *BCS FACS Seminars* (Springer, London, UK 2008) pp 1–42
95. D. Bjørner: From Domains to Requirements. In: *Montanari Festschrift*, vol 5065 of *Lecture Notes in Computer Science* (eds. Pierpaolo Degano, Rocco De Nicola and José Meseguer) (Springer, Heidelberg 2008) pp 1–30

96. D. Bjørner: Domain Engineering. In: *The 2007 Lipari PhD Summer School* (Springer, Heidelberg, Germany 2009) pp 1–102
97. D. Bjørner: *Software Engineering, Vol. I: The Triptych Approach, Vol. II: A Model Development* (To be submitted to Springer for evaluation, expected published 2009)
98. D. Bjørner: *Domain Engineering: “Upstream” from Requirements Engineering and Software Design*. US ONR + Univ. of Genoa Workshop, Santa Margherita Ligure (June 2000)
99. D. Bjørner, X. Chen: Public Government: A Domain Analysis. Technical Note, JAIST, School of Information Science, 1-1, Asahidai, Tatsunokuchi, Nomi, Ishikawa, Japan 923-1292 (2006)
100. D. Bjørner, J.R. Cuéllar: *Software Engineering Education: Rôles of Formal Specification and Design Calculi*. Annals of Software Engineering **6** (1998) pp 365–410
101. D. Bjørner, Y.L. Dong, S. Prehn: Domain Analyses: A Case Study of Station Management. In: *KICS'94: Kunming International CASE Symposium, Yunnan Province, P.R. of China* (1994)
102. D. Bjørner, L.M. Druffel: Industrial Experience in using Formal Methods. In: *Intl. Conf. on Software Engineering* (IEEE Computer Society Press, 1990) pp 264–266
103. D. Bjørner, A. Eirg: Compositionality: Ontology and Mereology of Domains. Some Clarifying Observations in the Context of Software Engineering. In: *Festschrift for Prof. Willem Paul de Roever* (Eds. Martin Steffen, Dennis Dams and Ulrich Hannemann, vol [not known at time of submission of the current paper] of *Lecture Notes in Computer Science* (eds. Martin Steffen, Dennis Dams and Ulrich Hannemann) (Springer, Heidelberg 2008) pp 1–12
104. D. Bjørner, C.W. George, A.E. Haxthausen et al: “UML”-ising Formal Techniques. In: *INT 2004: Third International Workshop on Integration of Specification Techniques for Applications in Engineering*, vol 3147 of *Lecture Notes in Computer Science* (Springer-Verlag, 2004, ETAPS, Barcelona, Spain) pp 423–450
105. D. Bjørner, C.W. George, S. Prehn. *Scheduling and Rescheduling of Trains*, chapter 8, pages 157–184. *Industrial Strength Formal Methods in Practice*, Eds.: Michael G. Hinchey and Jonathan P. Bowen. FACIT, Springer-Verlag, London, England, 1999.
106. D. Bjørner, C.W. George, S. Prehn: Computing Systems for Railways — A Rôle for Domain Engineering. Relations to Requirements Engineering and Software for Control Applications. In: *Integrated Design and Process Technology. Editors: Bernd Kraemer and John C. Petterson* (Society for Design and Process Science, P.O.Box 1299, Grand View, Texas 76050-1299, USA 2002)
107. D. Bjørner, A.E. Haxthausen, K. Havelund: *Formal, Model-oriented Software Development Methods: From VDM to ProCoS, and from RAISE to LaCoS*. Future Generation Computer Systems (1992)
108. Edited by D. Bjørner, M. Henson: *Logics of Specification Languages* (Springer, 2007)
109. Edited by D. Bjørner, M.C. Henson: *Logics of Specification Languages* (Springer, 2007)
110. Edited by D. Bjørner, M.C. Henson: *Logics of Specification Languages — see [131, 147, 157, 165, 179, 186, 233, 241, 262]* (Springer, Heidelberg, Germany 2008)

111. Edited by D. Bjørner, C.B. Jones: *The Vienna Development Method: The Meta-Language*, vol 61 of *LNCS* (Springer-Verlag, 1978)
112. Edited by D. Bjørner, C.B. Jones: *Formal Specification and Software Development* (Prentice-Hall, 1982)
113. D. Bjørner, C.B. Jones, M.M. an Airchinnigh, E.J. Neuhold, editors. *VDM – A Formal Method at Work*. Proc. VDM-Europe Symposium 1987, Brussels, Belgium, Springer-Verlag, Lecture Notes in Computer Science, Vol. 252, March 1987.
114. D. Bjørner, S. Koussobe, R. Noussi, G. Satchok: Michael Jackson's Problem Frames: Towards Methodological Principles of Selecting and Applying Formal Software Development Techniques and Tools. In: *ICFEM'97: Intl. Conf. on "Formal Engineering Methods", Hiroshima, Japan*, ed by L. ShaoQi, M. Hinchley (IEEE Computer Society Press, Los Alamitos, CA, USA 1997) pp 263–271
115. D. Bjørner, H.H. Løvgreen: Formal Semantics of Data Bases. In: *8th Int'l. Very Large Data Base Conf.* (1982)
116. D. Bjørner, H.H. Løvgreen: Formalization of Data Models. In: *Formal Specification and Software Development, [112]* (Prentice-Hall, 1982) pp 379–442
117. D. Bjørner, M. Nielsen: Meta Programs and Project Graphs. In: *ETW: Esprit Technical Week* (Elsevier, 1985) pp 479–491
118. D. Bjørner, J.F. Nilsson: Algorithmic & Knowledge Based Methods — Do they “Unify” ? — with some Programme Remarks for UNU/IIST. In: *International Conference on Fifth Generation Computer Systems: FGCS'92* (ICOT, 1992) pp (Separate folder, “191–198”)
119. D. Bjørner, O.N. Oest: The DDC Ada Compiler Development Project. In: *Towards a Formal Description of Ada, [121]*, vol 98 of *LNCS*, ed by D. Bjørner, O.N. Oest (Springer-Verlag, 1980) pp 1–19
120. D. Bjørner, O.N. Oest: *The DDC Ada Compiler Development Project. [121]* (1980) pp 1–19
121. Edited by D. Bjørner, O.N. Oest: *Towards a Formal Description of Ada*, vol 98 of *LNCS* (Springer-Verlag, 1980)
122. D. Bjørner, S. Prehn: Software Engineering Aspects of VDM. In: *Theory and Practice of Software Technology*, ed by D. Ferrari (North-Holland Publ.Co., Amsterdam, 1983)
123. D. Bjørner, A. Yasuhito, C. Xiaoyi, X. Jianwen: A Family of License Languages. Technical Report, JAIST, Graduate School of Information Science, 1-1, Asahidai, Tatsunokuchi, Nomi, Ishikawa, Japan 923-1292 (2006)
124. N. Bjørner: Models and Software Model Checking of a Distributed File Replication System. In: *Formal Methods and Hybrid Real-Time Systems*, vol 4700 of *Lecture Notes in Computer Science, Festschrift Essays in Honour of Dines Bjørner and Zhou Chaochen on the Occasion of Their 70th Birthdays* (Springer, 2007) pp 1–23
125. W.D. Blizard: *A Formal Theory of Objects, Space and Time*. The Journal of Symbolic Logic **55**, 1 (1990) pp 74–89
126. E.A. Boiten, J. Derrick, G. Smith, editors. *IFM 2004: Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, London, England, April 4-7 2004. Springer. Proceedings of 4th Intl. Conf. on IFM. ISBN 3-540-21377-5.
127. G. Booch, J. Rumbaugh, I. Jacobson: *The Unified Modeling Language User Guide* (Addison-Wesley, 1998)

128. R. Budde, K. Kuhlenkamp, K. Kautz, H. Zulighoven: *Prototyping: An Approach to Evolutionary System Development* (Springer, New York, Inc. Secaucus, NJ, USA 1992)
129. M.J. Butler, L. Petre, K. Sere, editors. *IFM 2002: Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, Turku, Finland, May 15-18 2002. Springer. Proceedings of 3rd Intl. Conf. on IFM. ISBN 3-540-43703-7.
130. D. Cansell, D. Méry: *Logical Foundations of the B Method*. Computing and Informatics **22**, 1–2 (2003)
131. D. Cansell, D. Méry. *Logics of Specification Languages*, chapter The event-B Modelling Method: Concepts and Case Studies, pages 47–152 in [110]. Springer, 2008.
132. R. Carnap: *The Logical Syntax of Language* (Harcourt Brace and Co., N.Y. 1937)
133. R. Carnap: *Introduction to Semantics* (Harvard Univ. Press, Cambridge, Mass. 1942)
134. R. Carnap: *Meaning and Necessity, A Study in Semantics and Modal Logic* (University of Chicago Press, 1947 (enlarged edition: 1956))
135. R. Casati, A. Varzi: *Parts and Places: the structures of spatial representation* (MIT Press, 1999)
136. X. Chen, D. Bjørner: Public Government: A Domain Analysis and a License Language. Technical Note, JAIST, School of Information Science, 1-1, Asahidai, Tatsunokuchi, Nomi, Ishikawa, Japan 923-1292 (2006)
137. C.N. Chong, R. Corin, S. Etalle: LicenseScript: A novel digital rights languages and its semantics. In: *Proc. of the Third International Conference WEB Delivering of Music (WEDELMUSIC'03)* (IEEE Computer Society Press, 2003) pp 122–129
138. C.N. Chong, R.J. Corin, J.M. Doumen et al: *LicenseScript: a logical language for digital rights management*. Annals of telecommunications special issue on Information systems security (2006)
139. C.N. Chong, S. Etalle, P.H. Hartel: Comparing Logic-based and XML-based Rights Expression Languages. In: *Confederated Int. Workshops: On The Move to Meaningful Internet Systems (OTM)*, no 2889 of *LNCS* (Springer, Catania, Sicily, Italy 2003) pp 779–792
140. B.L. Clarke: *A calculus of individuals based on “connection”*. Notre Dame J. Formal Logic **22**, 3 (1981) pp 204–218
141. G. Clemmensen, O. Oest: Formal Specification and Development of an Ada Compiler – A VDM Case Study. In: *Proc. 7th International Conf. on Software Engineering*, 26.-29. March 1984, Orlando, Florida (1984) pp 430–440
142. CoFI (The Common Framework Initiative): *CASL Reference Manual*, vol 2960 of *Lecture Notes in Computer Science (IFIP Series)* (Springer-Verlag, 2004)
143. J. Connell, L. Shafer: *Structured rapid prototyping: an evolutionary approach to software development* (Yourdon Press, Upper Saddle River, NJ, USA 1989)
144. D. Crystal: *The Cambridge Encyclopedia of Language* (Cambridge University Press, 1987, 1988)
145. W. Damm, D. Harel: *LSCs: Breathing Life into Message Sequence Charts*. Formal Methods in System Design **19** (2001) pp 45–80
146. J. de Bakker: *Control Flow Semantics* (The MIT Press, Cambridge, Mass., USA, 1995)
147. R. Diaconescu. *Logics of Specification Languages*, chapter A Methodological Guide to the CafeOBJ Logic, pages 153–240 in [110]. Springer, 2008.

148. R. Diaconescu, K. Futatsugi, K. Ogata: *CafeOBJ: Logical Foundations and Methodology*. Computing and Informatics **22**, 1–2 (2003)
149. Y. Dittrich: *Rethinking the Software Life Cycle: About the Interlace of Different Design and Development Activities*. Position Paper for the Dagstuhl Seminar: End User Software Engineering (2007)
150. Y. Dittrich, P. Sestoft: Designing Evolvable Software Products; Coordinating the Evolution of different Layers in Kernel Based Software Products; Project Description. http://www.itu.dk/research/sdg/doku.php?id=evolvable_software_products:esp. IT University of Copenhagen, Software Development Group, Rued Langaardsvej 7, DK-2300 Copenhagen, Denmark. (2005)
151. D.M. Weiss and C.T.R. Lai: *Software product-line engineering: a family-based software development process* (Addison-Wesley Longman Publishing Co., 1999)
152. A. Eir: Construction Informatics — issues in engineering, computer science, and ontology. PhD Thesis, Dept. of Computer Science and Engineering, Institute of Informatics and Mathematical Modeling, Technical University of Denmark, Building 322, Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark (2004)
153. A. Eir. *Formal Methods and Hybrid Real-Time Systems*, chapter Relating Domain Concepts Intensionally by Ordering Connections, pages 188–216. Springer (LNCS Vol. 4700, Festschrift: Essays in Honour of Dines Bjørner and Zhou Chaochen on the Occasion of Their 70th Birthdays), 2007.
154. W. Feijen, A. van Gasteren, D. Gries, J. Misra, editors. *Beauty is Our Business*, Texts and Monographs in Computer Science, New York, NY, USA, 1990. Springer-Verlag. A Birthday Salute to Edsger W. Dijkstra.
155. R. Feynmann, R. Leighton, M. Sands: *The Feynmann Lectures on Physics*, vol Volumes I–II–II (Addison-Wesley, California Institute of Technology 1963)
156. B. Fitzgerald, N.L. Russo, T. O’Kane: *Software development method tailoring at Motorola*. Commun. ACM **46**, 4 (2003) pp 64–70
157. J.S. Fitzgerald. *Logics of Specification Languages*, chapter The Typed Logic of Partial Functions and the Vienna Development Method, pages 453–487 in [110]. Springer, 2008.
158. J.S. Fitzgerald, P.G. Larsen: *Developing Software using VDM-SL* (Cambridge University Press, The Edinburgh Building, Cambridge CB2 1RU, England 1997)
159. P. Folkjær, D. Bjørner: A Formal Model of a Generalised CSP-like Language. In: *Proc. IFIP’80*, ed by S. Lavington (North-Holland Publ.Co., Amsterdam, 1980) pp 95–99
160. C. Fox: *The Ontology of Language: Properties, Individuals and Discourse* (CSLI Publications, Center for the Study of Language and Information, Stanford University, California, ISA 2000)
161. K. Futatsugi, R. Diaconescu: *CafeOBJ Report The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification* (World Scientific Publishing Co. Pte. Ltd., 5 Toh Tuck Link, SINGAPORE 596224. Tel: 65-6466-5775, Fax: 65-6467-7667, E-mail: wspc@wspc.com.sg 1998)
162. K. Futatsugi, A. Nakagawa, 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.

163. E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design patterns: elements of reusable object-oriented software* (Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA 1995)
164. B. Ganter, R. Wille: *Formal Concept Analysis — Mathematical Foundations* (Springer-Verlag, January 1999)
165. C. George, A.E. Haxthausen. *Logics of Specification Languages*, chapter The Logic of the RAISE Specification Language, pages 349–399 in [110]. Springer, 2008.
166. C.W. George, P. Haff, K. Havelund et al: *The RAISE Specification Language* (Prentice-Hall, Hemel Hempstead, England 1992)
167. C.W. George, A.E. Haxthausen: *The Logic of the RAISE Specification Language*. Computing and Informatics **22**, 1–2 (2003)
168. C.W. George, A.E. Haxthausen, S. Hughes et al: *The RAISE Method* (Prentice-Hall, Hemel Hempstead, England 1995)
169. C. Ghezzi, M. Jazayeri, D. Mandrioli: *Fundamentals of Software Engineering* (Prentice Hall, 2002)
170. P. Grenon, B. Smith: *SNAP and SPAN: Towards Dynamic Spatial Ontology*. Spatial Cognition and Computation **4**, 1 (2004) pp 69–104
171. W. Grieskamp, T. Santen, B. Stoddart, editors. *IFM 2000: Integrated Formal Methods*, volume of *Lecture Notes in Computer Science*, Schloss Dagstuhl, Germany, November 1–3 2000. Springer. Proceedings of 2nd Intl. Conf. on IFM.
172. C.A. Gunter, S.T. Weeks, A.K. Wright: Models and Languages for Digital Rights. In: *Proc. of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)* (IEEE Computer Society Press, Maui, Hawaii, USA 2001) pp 4034–4038
173. C. Gunther: *Semantics of Programming Languages* (The MIT Press, Cambridge, Mass., USA, 1992)
174. Edited by P. Haff: *The Formal Definition of CHILL* (ITU (Intl. Telecomm. Union), Geneva, Switzerland 1981)
175. Edited by P. Haff: *The Formal Definition of CHILL* (ITU (Intl. Telecomm. Union), Geneva, Switzerland 1981)
176. P. Haff, A. Olsen: Use of VDM within CCITT. In: [113] (Springer-Verlag, 1987) pp 324–330
177. P. Hall, D. Bjørner, Z. Mikolajuk: Decision Support Systems for Sustainable Development: Experience and Potential — a Position Paper. Administrative Report 80, UNU/IIST, P.O.Box 3058, Macau (1996)
178. J.Y. Halpern, V. Weissman: A Formal Foundation for XrML. In: *Proc. of the 17th IEEE Computer Security Foundations Workshop (CSFW'04)* (2004)
179. M.R. Hansen. *Logics of Specification Languages*, chapter Duration Calculus, pages 299–347 in [110]. Springer, 2008.
180. D. Harel: *Statecharts: A Visual Formalism for Complex Systems*. Science of Computer Programming **8**, 3 (1987) pp 231–274
181. D. Harel: *On Visual Formalisms*. Communications of the ACM **33**, 5 (1988)
182. D. Harel, E. Gery: *Executable Object Modeling with Statecharts*. IEEE Computer **30**, 7 (1997) pp 31–42
183. D. Harel, H. Lachover, A. Naamad et al: *STATEMATE: A Working Environment for the Development of Complex Reactive Systems*. Software Engineering **16**, 4 (1990) pp 403–414

184. D. Harel, R. Marelly: *Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine* (Springer-Verlag, 2003)
185. D. Harel, A. Naamad: *The STATEMATE Semantics of Statecharts*. ACM Transactions on Software Engineering and Methodology (TOSEM) **5**, 4 (1996) pp 293–333
186. M.C. Henson, M. Deutsch, S. Reeves. *Logics of Specification Languages*, chapter Z Logic and Its Applications, pages 489–596 in [110]. Springer, 2008.
187. M.C. Henson, S. Reeves, J.P. Bowen: *Z Logic and its Consequences*. Computing and Informatics **22**, 1–2 (2003)
188. C.A.R. Hoare, J.F. He: *Unifying Theories of Programming* (Prentice Hall, 1997)
189. T. Hoare: *Communicating Sequential Processes* (Prentice-Hall International, 1985)
190. T. Hoare. Communicating Sequential Processes. Published electronically: <http://www.usingcsp.com/cspbook.pdf>, 2004. Second edition of [189]. See also <http://www.usingcsp.com/>.
191. M. Holzer, S. Katzenbeisser, C. Schallhart: Towards a Formal Semantics for ODRL. In: *Proc. of the First International ODRL Workshop* (2004)
192. G.J. Holzmann: *The SPIN Model Checker, Primer and Reference Manual* (Addison-Wesley, Reading, Massachusetts 2003)
193. P. Hruby: *Model-driven Design Using Business Patterns* (Springer, 2006)
194. W. Humphrey: *Managing The Software Process* (Addison-Wesley, 1989)
195. IEEE Computer Society: IEEE–STD 610.12-1990: Standard Glossary of Software Engineering Terminology. Technical Report, IEEE, IEEE Headquarters Office, 1730 Massachusetts Avenue, N.W., Washington, DC 20036-1992, USA. Phone: +1-202-371-0101, FAX: +1-202-728-9614 (1990)
196. M. Ingleby: Safety properties of a control network: local and global reasoning in machine proof. In: *Proceedings of Real Time Systems* (Paris, 1994)
197. M. Ingleby: A Galois theory of local reasoning in control systems with compositionality. In: *Proceedings of Mathematics of Dependable Systems* (Oxford UP (UK), 1995)
198. M. Ingleby, I. Mitchell: Proving Safety of a Railway Signaling System Incorporating Geographic Data. In: *SAFECOM'92 Conference Proceedings of IFAC*, ed by H. Frey (Pergamon Press, Zürich (CH) 1992) pp 129–134
199. ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992.
200. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1996.
201. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1999.
202. D. Jackson: *Software Abstractions Logic, Language, and Analysis* (The MIT Press, Cambridge, Mass., USA April 2006)
203. M.A. Jackson: *Principles of Program Design* (Academic Press, 1969)
204. M.A. Jackson: *System Design* (Prentice-Hall International, 1985)
205. M.A. Jackson. *Software Development Method*, chapter 13, pages 215–234. Prentice Hall Intl., 1994. Festschrift for C. A. R. Hoare: *A Classical Mind*, Ed. W. Roscoe.
206. M.A. Jackson: *Software Requirements & Specifications: a lexicon of practice, principles and prejudices* (Addison-Wesley Publishing Company, Wokingham, nr. Reading, England; E-mail: ipc@awpub.add-wes.co.uk 1995)

207. M.A. Jackson: *Problem Frames — Analyzing and Structuring Software Development Problems* (Addison-Wesley, Edinburgh Gate, Harlow CM20 2JE, England 2001)
208. I. Jacobson, G. Booch, J. Rumbaugh: *The Unified Software Development Process* (Addison-Wesley, 1999)
209. F.V. Jensen: *Bayesian Networks and Decision Graphs* (Springer, 1991)
210. K. Jensen: *Coloured Petri Nets*, vol 1: Basic Concepts (234 pages + xii), Vol. 2: Analysis Methods (174 pages + x), Vol. 3: Practical Use (265 pages + xi) of *EATCS Monographs in Theoretical Computer Science* (Springer-Verlag, Heidelberg 1985, revised and corrected second version: 1997)
211. Klaus Pohl and Günter Böckle and Frank J. van der Linden: *Software Product Line Engineering: Foundations, Principles and Techniques* (Springer, 2005)
212. J. Klose, H. Wittke: An Automata Based Interpretation of Live Sequence Charts. In: *TACAS 2001*, ed by T. Margaria, W. Yi (Springer-Verlag, 2001) pp 512–527
213. R. Koenen, J. Lacy, M. Mackay, S. Mitchell: *The long march to interoperable digital rights management*. Proceedings of the IEEE **92**, 6 (2004) pp 883–897
214. G. Lakoff: *Women, Fire, and Dangerous Things: What Categories Reveal about the Mind* (University of Chicago Press, Chicago, Ill., USA 1987)
215. G. Lakoff, M. Johnson: *Metaphors We Live by* (Chicago University Press, Chicago, Ill., USA 1980)
216. L. Lamport: *Time, Clocks, and the Ordering of Events in a Distributed System*. Communications of the ACM **21**, 7 (1978) pp 558–565
217. L. Lamport: *The Temporal Logic of Actions*. Transactions on Programming Languages and Systems **16**, 3 (1995) pp 872–923
218. L. Lamport: *Specifying Systems* (Addison-Wesley, Boston, Mass., USA 2002)
219. P. Landin: *A Correspondence Between ALGOL 60 and Church's Lambda-Notation (in 2 parts)*. Communications of the ACM **8**, 2-3 (1965) pp 89–101 and 158–165
220. C. Lejewski: *A note on Leśniewski's axiom system for the mereological notion of ingredient or element*. Topoi **2**, 1 (June, 1983) pp 63–71
221. H. Leonard, N. Goodman: *The Calculus of Individuals and Its Uses*. Journal of Symbolic Logic **5** (1940) pp 45–55
222. M.J. Loux: *Metaphysics, a contemporary introduction* (Routledge, London and New York 1998 (2nd ed., 2020))
223. H.H. Løvengreen, D. Bjørner: On a Formal Model of the Tasking Concepts in Ada. In: *ACM SIGPLAN Ada Symp.* (1980)
224. I.S.P. Ltd. Open Digital Rights Language (ODRL). <http://odrl.net>, 2001.
225. E. Lushei: *The Logical Systems of Leśniewski* (North Holland, Amsterdam, The Netherlands 1962)
226. G.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 (2002)
227. M Jazayeri and A Ran and F van der Linden: *Software architecture for product families: principles and practice* (Addison-Wesley Longman Publishing Co., 2000)
228. Z. Manna: *Mathematical Theory of Computation* (McGraw-Hill, 1974)

229. M. Marchesi, G. Succi, editors. *Proceedings: Extreme Programming and Agile Processes in Software Engineering: 4th International Conference, XP 2003*, number 2675 in LNCS, Genova, Italy, May 2003. Springer-Verlag.
230. J. McCarthy: Towards a Mathematical Science of Computation. In: *IFIP World Congress Proceedings*, ed by C. Popplewell (1962) pp 21–28
231. D.H. Mellor, A. Oliver: *Properties* (Oxford Univ Press, May 1997)
232. S. Merz: *On the Logic of TLA+*. Computing and Informatics **22**, 1–2 (2003)
233. S. Merz. *Logics of Specification Languages*, chapter The Specification Language TLA⁺, pages 401–451 in [110]. Springer, 2008.
234. B. Meyer: *Object-oriented Software Construction* (Prentice Hall, 1997)
235. B. Meyer: *Object-oriented software construction (2nd ed.)* (Prentice-Hall, Inc., Upper Saddle River, NJ, USA 1997)
236. S. Michiels, K. Verslype, W. Joosen, B.D. Decker: Towards a Software Architecture for DRM. In: *Proceedings of the Fifth ACM Workshop on Digital Rights Management (DRM'05)* (2005) pp 65–74
237. D. Miéville, D. Vernant: *Stanisław Leśniewski aujourd'hui* (NoPublisher, Grenoble October 8-10, 1992)
238. C.C. Morgan: *Programming from Specifications* (Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK 1990)
239. C. Morris: Foundations of the Theory of Signs. In: *International Encyclopedia of Unified Science* (Univ. of Chicago Press, 1938)
240. C. Morris: *Signs, Languages and Behaviour* (G. Brazillier, New York, 1955)
241. T. Mossakowski, A. Haxthausen, D. Sannella, A. Tarlecki. *Logics of Specification Languages*, chapter CASL – the Common Algebraic Specification Language, pages 241–298 in [110]. Springer, 2008.
242. T. Mossakowski, A.E. Haxthausen, D. Sannella, A. Tarlecki: *CASL — The Common Algebraic Specification Language: Semantics and Proof Theory*. Computing and Informatics **22**, 1–2 (2003)
243. D. Mulligan, A. Burstein: Implementing copyright limitations in rights expression languages. In: *Proc. of 2002 ACM Workshop on Digital Rights Management*, vol 2696 of *Lecture Notes in Computer Science* (Springer-Verlag, 2002) pp 137–154
244. D.K. Mulligan, J. Han, A.J. Burstein: How DRM-Based Content Delivery Systems Disrupt Expectations of “Personal Use”. In: *Proc. of The 3rd International Workshop on Digital Rights Management* (2003) pp 77–89
245. Object Management Group: *OMG Unified Modelling Language Specification*, version 1.5 edn (OMG/UML, <http://www.omg.org/uml/> 2003)
246. C.S. Peirce: *Reasoning and the Logic of Things*, Edited by Kenneth Laine Ketner (Harvard University Press, 1 Feb 1993)
247. C.S. Peirce: *Pragmatism as a Principle and Method of right thinking: The 1903 Harvard Lectures on Pragmatism* (State Univ. of N.Y. Press, and Cornell Univ. Press, 14 July 1997)
248. C.S. Peirce: *Peirce on Signs: Writings on Semiotics* (Univ. of North Carolina Press, Editor: James Hoopes, 15 Dec 1991)
249. C.S. Peirce: *Writings: A Chronological Edition* (Indiana University Press, 15 Jan 1994)
250. C.A. Petri: *Kommunikation mit Automaten* (Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962)
251. S.L. Pfleeger: *Software Engineering, Theory and Practice*, 2nd edn (Prentice-Hall, 2001)

252. C.-Y.T. Pi: Mereology in Event Semantics. PhD, McGill University, Montreal, Canada (1999)
253. R.S. Pressman: *Software Engineering, A Practitioner's Approach*, 5th edn (McGraw-Hill, 1981–2001)
254. R. Pucella, V. Weissman: A Logic for Reasoning about Digital Rights. In: *Proc. of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)* (IEEE Computer Society Press, 2002) pp 282–294
255. R. Pucella, V. Weissman: A Formal Foundation for ODRL. In: *Proc. of the Workshop on Issues in the Theory of Security (WIST'04)* (2004)
256. M. Pěnička, D. Bjørner: From Railway Resource Planning to Train Operation — a Brief Survey of Complementary Formalisations. In: *Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions, 22–27 August, 2004, Toulouse, France — Ed. René Jacquart* (Kluwer Academic Publishers, 2004) pp 629–636
257. M. Pěnička, A.K. Strupchanska, D. Bjørner: Train Maintenance Routing. In: *FORMS'2003: Symposium on Formal Methods for Railway Operation and Control Systems* (L'Harmattan Hongrie, 2003)
258. W. Reisig: *Petri Nets: An Introduction*, vol 4 of *EATCS Monographs in Theoretical Computer Science* (Springer Verlag, 1985)
259. W. Reisig: *A Primer in Petri Net Design* (Springer Verlag, 1992)
260. W. Reisig: *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets* (Springer Verlag, 1998)
261. W. Reisig: *The Expressive Power of Abstract State Machines*. Computing and Informatics **22**, 1–2 (2003)
262. W. Reisig. *Logics of Specification Languages*, chapter Abstract State Machines for the Classroom, pages 15–46 in [110]. Springer, 2008.
263. J.C. Reynolds: *The Semantics of Programming Languages* (Cambridge University Press, 1999)
264. F. Van der Rhee, H. Van Nauta Lemke, J. Dukman: *Knowledge Based Fuzzy Control of Systems*. IEEE Trans. Autom. Control **35**, 2 (1990) pp 148–155
265. J.M. Romijn, G.P. Smith, J.C. van de Pol, editors. *IFM 2005: Integrated Formal Methods*, volume 3771 of *Lecture Notes in Computer Science*, Eindhoven, The Netherlands, December 2005. Springer. Proceedings of 5th Intl. Conf. on IFM. ISBN 3-540-30492-4.
266. A.W. Roscoe: *Theory and Practice of Concurrency* (Prentice-Hall, 1997)
267. D.T. Ross: Toward foundations for the understanding of type. In: *Proceedings of the 1976 conference on Data : Abstraction, definition and structure* (ACM, New York, NY, USA 1976) pp 63–65
268. A. Roychoudhury, P. Thiagarajan: Communicating Transaction Processes: An MSC-based Model of Computation for Reactive Embedded Systems. In: *Proc. of the 3rd IEEE International Conference on Application of Concurrency in System Design (ACSD'03)* (IEEE Press, 2003) pp 157–166
269. J. Rumbaugh, I. Jacobson, G. Booch: *The Unified Modeling Language Reference Manual* (Addison-Wesley, 1998)
270. B. Russell: *On Denoting*. Mind **14** (1905) pp 479–493
271. P. Samuelson: *Digital rights management {and, or, vs.} the law*. Communications of ACM **46**, 4 (2003) pp 41–45
272. D.A. Schmidt: *Denotational Semantics: a Methodology for Language Development* (Allyn & Bacon, 1986)

273. S. Schneider: *Concurrent and Real-time Systems — The CSP Approach* (John Wiley & Sons, Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England 2000)
274. P. Sestoft. *Evolvable Software Products*. Lecture Notes in Computer Science. Springer, Heidelberg, Tutorials. The Lipari Summer School, July 2007, on Advances in Software Engineering, eds. Egon Börger and Alfredo Ferro [To appear] 2008.
275. S. Shapiro: *Philosophy of Mathematics — structure and ontology* (Oxford University Press, 1997)
276. P.M. Simons: *Parts: A Study in Ontology* (Clarendon Press, 1987)
277. C. Simonyi: The Death of Computer Languages, the Birth of Intentional Programming. In: *NATO Science Committee Informatics Conference* (1995)
278. C. Simonyi: *The Future is Intentional*. Computer **32**, 5 (1999) pp 56–57
279. C. Simonyi: Intentional Programming: Asymptotic Fun? In: *Position Paper, SDP Workshop, Vanderbilt University* (December, 2001)
280. I. Sommerville: *Software Engineering*, 6th edn (Addison-Wesley, 1982–2001)
281. J.F. Sowa: *Knowledge Representation: Logical, Philosophical, and Computational Foundations* (Pws Pub Co, August 17, 1999)
282. J.F. Sowa: *Knowledge Representation: Logical, Philosophical, and Computational Foundations* (Pws Pub Co, August 17, 1999)
283. J.M. Spivey: *Understanding Z: A Specification Language and its Formal Semantics*, vol 3 of *Cambridge Tracts in Theoretical Computer Science* (Cambridge University Press, 1988)
284. J.M. Spivey: *The Z Notation: A Reference Manual*, 2nd edn (Prentice Hall International Series in Computer Science, 1992)
285. Edited by J. Srzednicki, Z. Stachniak: *Leśniewski's Lecture Notes in Logic* (Dordrecht, 1988)
286. J. Srzednicki, Z. Stachniak: *Leśniewski's Systems Protothetic* (Dordrecht, 1998)
287. Edited by S. Staab, R. Stuber: *Handbook on Ontologies* (Springer, Heidelberg 2004)
288. Staff of Merriam Webster. Online Dictionary: <http://www.m-w.com/home.htm>, 2004. Merriam–Webster, Inc., 47 Federal Street, P.O. Box 281, Springfield, MA 01102, USA.
289. A.K. Strupchanska, M. Pěnička, D. Bjørner: Railway Staff Rostering. In: *FORMS2003: Symposium on Formal Methods for Railway Operation and Control Systems* (L'Harmattan Hongrie, 2003)
290. Edited by S.J. Surma, J.T. Srzednicki, D.I. Barnett, V.F. Rickey: *Stanislaw Leśniewski: Collected works (2 Vols.)* (Dordrecht, Boston – New York 1988)
291. R. Tennent: *The Semantics of Programming Languages* (Prentice–Hall Intl., 1997)
292. J.D. Ullman, J. Widom: *A First Course in Database Systems* (Prentice Hall, October 2, 2001)
293. J. van Benthem: *The Logic of Time*, vol 156 of *Synthese Library: Studies in Epistemology, Logic, Methodology, and Philosophy of Science (Editor: Jaakko Hintikka)*, 2nd edn (Kluwer Academic Publishers, P.O.Box 17, NL 3300 AA Dordrecht, The Netherlands 1983, 1991)
294. H. van Vliet: *Software Engineering: Principles and Practice* (John Wiley & Sons, Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England 2000)
295. G. Winskel: *The Formal Semantics of Programming Languages* (The MIT Press, Cambridge, Mass., USA, 1993)

- 296. Winston Royce: Managing the Development of Large Software Systems. In: *Proceedings of IEEE WESCON 26* (1970) pp 1–9
- 297. J.C.P. Woodcock, J. Davies: *Using Z: Specification, Proof and Refinement* (Prentice Hall International Series in Computer Science, 1996)
- 298. J.C.P. Woodcock, M. Loomes: *Software Engineering Mathematics* (Pitman, London, 1988)
- 299. J. Xiang, D. Bjørner: The Electronic Media Industry: A Domain Analysis and a License Language. Technical Note, JAIST, School of Information Science, 1-1, Asahidai, Tatsunokuchi, Nomi, Ishikawa, Japan 923-1292 (2006)
- 300. H. Zemanek: Semiotics and Programming Languages. In: [2] (1966) pp 139–143
- 301. C.C. Zhou, M.R. Hansen: *Duration Calculus: A Formal Approach to Real-time Systems* (Springer-Verlag, 2004)
- 302. C.C. Zhou, C.A.R. Hoare, A.P. Ravn: *A Calculus of Durations*. Information Proc. Letters **40**, 5 (1992)
- 303. C.C. Zhou, A.P. Ravn, M.R. Hansen: An Extended Duration Calculus for Real-time Systems. Research Report 9, UNU/IIST, P.O.Box 3058, Macau (1993)

