

Specification and Validation of Telecommunications Systems with Use Case Maps and LOTOS

by

Daniel Amyot

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of
the requirements for the degree of

Ph.D. in Computer Science

under the auspices of the
Ottawa-Carleton Institute for Computer Science

School of Information Technology and Engineering (SITE)
University of Ottawa
Ottawa, Ontario, Canada

© Daniel Amyot, September 2001

Abstract

The functional modeling of telecommunications systems requires an early emphasis on behavioral aspects. In the first stages of common development processes, telecommunications features, services, and functionalities are defined in terms of informal requirements and visual descriptions. As these descriptions grow and evolve, they quickly become error-prone and difficult to understand. Consequently, designs can hardly be checked or validated against such descriptions.

This thesis proposes an innovative methodology named *Specification-Validation Approach with LOTOS and UCMs* (SPEC-VALUE), which tackles these problems using two notations. The first notation, called *Use Case Maps* (UCMs), is used to capture and integrate functional requirements in terms of *causal* scenarios. Scenarios describing system views, uses, and services are becoming a common method of capturing functional requirements of reactive and distributed systems. They are particularly appropriate to represent behavioral aspects so that various stakeholders can understand them. In addition to these general properties of scenarios, UCMs can help reasoning about system-wide functionalities at a high level of abstraction. Integrating UCMs together also helps avoiding many undesirable interactions, usually resulting from the composition of different scenarios, before the generation of prototypes.

The second notation is the formal specification language LOTOS. It will be shown that UCM scenarios bound to architectural components can be translated into high-level LOTOS specifications. In turn, these specifications can be used as prototypes to animate UCMs and to validate high-level designs against requirements systematically through numerous techniques, including functional testing based on UCMs. LOTOS possesses powerful testing concepts and tools that excel at detecting errors and undesirable interactions. LOTOS represents a judicious formalism here because it supports many UCM constructs directly, and it complements most of UCM's weak areas related to the analysis of systems.

SPEC-VALUE introduces theories and techniques for constructing LOTOS specifications from UCMs, for deriving validation test cases from UCMs, and for measuring the structural coverage of LOTOS specifications achieved during validation. An ongoing example is used to illustrate these concepts: the *Tiny Telephony System*. The thesis validates the SPEC-VALUE methodology through its application to various telecommunications systems (Group Communication Server, Group-Call service of GPRS, feature interactions, agent-based simplified basic call, and GSM's MAP protocol), and concludes with an assessment of these experimental results.

Acknowledgements

Many people were involved in the production of this thesis, and I wish to express my gratitude towards them. *Grazie mille* to my supervisor and friend, Professor Luigi Logrippo, who spent countless hours reading my numerous drafts and initiating unforgettable discussions. For the last quarter of my life, I have been most fortunate to enjoy his wisdom, experience, guidance and enthusiasm. For many years, I also had the chance to savor the expertise, unbounded imagination and warm encouragements of Professor Ray Buhr, my co-supervisor. I hope my thanks will reach him under the California sun, where he now enjoys a well-deserved retirement. I also extend my thanks to my committee, Dr. Robert L. Probert, Dr. Michael Weiss, Dr. C. Murray Woodside, and Dr. Richard H. Carver, my external examiner, for gracefully accepting to review and comment this work.

The weekly (and intense) meetings with the LOTOS Group, composed of wonderful people, represented an exquisite arena where many of the ideas presented here were first introduced. I thank the numerous friends I met there over the years. In particular, I am indebted towards Jacques Sinnens for sharing his deep technical knowledge and his humour with me. His constant presence made the past nine years a memorable experience.

Many parts of this thesis, and in particular the case studies, benefited from collaborations with colleagues and co-authors, whom I wish to thank: Rossana Andrade, Natalia Balaba, Hichem Ben Fredj, Francis Bordeleau, Don Cameron, Leïla Charfi, Pascal Forhan, Nicolas Gorse, Tom Gray, Jim Hodges, Serge Mankovski, Andrew Miga, Gunter Mussbacher, Dorin Petriu, John Visser, Tom Ware Alan Williams, and many others who have shared their enthusiasm towards UCMs and LOTOS.

To Michel Racine, Johanne Forgues, Louise Desrosiers, Hasan Ural, and the SITE staff: *merci* for your constant and friendly support. You are the best!

This work was made possible by the financial support of the “Fonds pour la Formation de Chercheurs et l'Aide à la Recherche” (FCAR, Québec), the Natural Sciences and Engineering Research Council of Canada, Communications and Information Technology Ontario, the University of Ottawa, Mitel Corporation, Nortel Networks, and Motorola Canada. A special thank goes to my manager, Daisy Fung, for giving me the time I needed to complete this thesis.

Enfin, mille mercis à mes amis ainsi qu'aux membres de ma famille. À mes parents, Jean-Guy et Madeleine, qui m'ont appris à travailler et à aimer apprendre, tout en m'apportant un support exceptionnel. À mon épouse, Annie Blais, pour son amour, son aide, sa foi en moi et ses encouragements de tous les jours. À nos enfants Sandra, Jean-Luc, et Mireille, pour leur rires et pour me rappeler l'importance du jeu et de l'équilibre. Je vous aime.

Table of Contents

Abstract	i
Acknowledgements	iii
Table of Contents	v
List of Figures	xi
List of Tables	xiii
List of Acronyms	xv
CHAPTER 1	
Introduction	1
1.1 Motivation	1
1.2 Research Hypothesis	5
1.3 New Approach: SPEC-VALUE	7
1.4 Thesis Contributions	10
1.4.1 Contribution 1: SPEC-VALUE Methodology	10
1.4.2 Contribution 2: Theories and Techniques Supporting SPEC-VALUE	11
1.4.3 Contribution 3: Illustrative Experiments Validating SPEC-VALUE	12
1.4.4 Issues Not Addressed in this Thesis	13
1.5 Thesis Outline	13

CHAPTER 2

Basic Definitions and Notations	15
2.1 <i>Basic Definitions</i>	15
2.1.1 Four Engineering Disciplines	16
2.1.2 Processes, Formal Methods, Specifications and Designs	18
2.1.3 Validation and Verification	20
2.2 <i>Introduction to Use Case Maps</i>	22
2.2.1 Philosophy of UCMs	22
2.2.2 Information Needed to Construct UCMs	23
2.2.3 Basic UCM Path Notation	23
2.2.4 UCM Component Notation	25
2.2.5 Advanced UCM Path Notation	26
2.2.6 UCM Tools	28
2.3 <i>Introduction to LOTOS</i>	28
2.3.1 Philosophy of LOTOS	28
2.3.2 Information Needed to Construct LOTOS Specifications	29
2.3.3 LOTOS Operators	30
2.3.4 LOTOS Abstract Data Types	31
2.3.5 Labelled Transitions Systems and Underlying Semantics	33
2.3.6 Equivalences and Other Relations	35
2.3.7 Validation and Verification in LOTOS	41
2.3.8 LOTOS Tools	42
2.3.9 Enhancements to LOTOS	43
2.4 <i>Chapter Summary</i>	43

CHAPTER 3

Literature Survey	45
3.1 <i>Causality</i>	45
3.1.1 Why causality?	46
3.1.2 Concurrency Models and Equivalence Relations	47
3.1.3 Causality and Use Case Maps	49
3.1.4 Causality and LOTOS	50
3.1.5 Summary and Discussion	51
3.2 <i>Specification Techniques</i>	51
3.2.1 Evaluation Criteria for Specification Techniques	52
3.2.2 Overview of Selected Techniques	54
3.2.3 Comparison Between Specification Techniques	59
3.3 <i>Scenarios</i>	63
3.3.1 Why Scenarios?	64
3.3.2 Evaluation Criteria for Scenario Definitions	65
3.3.3 Overview of Selected Scenario Notations	68
3.3.4 Construction Approaches	74
3.4 <i>Validation and Verification</i>	87
3.4.1 Properties	87
3.4.2 General Testing Concepts	90
3.4.3 LOTOS Testing	95
3.4.4 Coverage	101
3.4.5 Testing Patterns	103
3.4.6 Summary and Discussion	106

	3.5 Chapter Summary	107
CHAPTER 4		
	From Requirements to UCMs in SPEC-VALUE	111
	4.1 <i>Return on the SPEC-VALUE Methodology</i>	111
	4.1.1 SPEC-VALUE and Software Development Process Models	112
	4.2 <i>First Steps of the SPEC-VALUE Methodology</i>	114
	4.2.1 From Requirements to UCMs	114
	4.2.2 Style and Content Guidelines for UCMs	115
	4.2.3 Integration of Scenarios	116
	4.3 <i>Ongoing Example: Tiny Telephone System (TTS)</i>	117
	4.3.1 Informal Requirements for TTS	117
	4.3.2 Individual Use Case Maps for TTS	118
	4.3.3 Integrated UCM View	120
	4.4 <i>Chapter Summary</i>	123
CHAPTER 5		
	From Use Case Maps to LOTOS in SPEC-VALUE	125
	5.1 <i>Construction Approach</i>	125
	5.1.1 Appropriateness of LOTOS	127
	5.1.2 Unfitness of TMDL	128
	5.2 <i>Construction Guidelines</i>	129
	5.2.1 Overview	129
	5.2.2 Construction Guidelines for Paths	132
	5.2.3 Construction Guidelines for Structures	142
	5.2.4 Construction Guidelines for Data	154
	5.2.5 Towards Partial Automation	155
	5.3 <i>Applying the Construction Guidelines to TTS</i>	157
	5.3.1 Structure of the TTS Specification	157
	5.3.2 TTS Data Types and Operations	160
	5.3.3 TTS Process Definitions	161
	5.4 <i>Chapter Summary</i>	166
CHAPTER 6		
	UCM-LOTOS Testing Framework	169
	6.1 <i>Testing Approach in SPEC-VALUE</i>	170
	6.1.1 Justification for a Testing Approach	170
	6.1.2 Testing in SPEC-VALUE	171
	6.1.3 Validation Testing and Conformance Testing	173
	6.2 <i>UCM-LOTOS Testing Concepts</i>	174
	6.2.1 Combination of Approaches	175
	6.2.2 Structure of UCM-Based Validation Test Suites	175
	6.2.3 Validity Relation	180
	6.2.4 Comparing Validity and Conformance in the Two Worlds	182

6.3	<i>UCM-Oriented Testing Patterns for Test Goal Selection</i>	184
6.3.1	Introduction to UCM-Oriented Testing Patterns	185
6.3.2	Template for UCM-Oriented Testing Patterns	187
6.3.3	UCM-Oriented Testing Pattern Language	189
6.3.4	Testing Pattern and Strategies for Alternatives	191
6.3.5	Testing Pattern and Strategies for Concurrent Paths	194
6.3.6	Testing Pattern and Strategies for Loops	197
6.3.7	Testing Pattern and Strategies for Multiple Start Points	200
6.3.8	Testing Pattern and Strategies for a Single Stub and its Plug-ins	207
6.3.9	Testing Pattern and Strategies for Causally Linked Stubs	210
6.3.10	Discussion	213
6.3.11	Section Summary	220
6.4	<i>Complementary Strategies and Test Case Generation</i>	221
6.4.1	From Test Goals to Test Cases	221
6.4.2	Strategies for Value Selection	222
6.4.3	Completeness and Determinism Issues	223
6.4.4	Strategies for Rejection Test Cases	224
6.5	<i>Testing the TTS System</i>	226
6.5.1	Test Goals for TTS	226
6.5.2	Further Test Goals for Robustness Testing	231
6.5.3	Test Cases Generation	232
6.5.4	Results from Test Execution	234
6.6	<i>Chapter Summary</i>	235

CHAPTER 7

	Structural Coverage	237
7.1	<i>Structural Coverage in SPEC-VALUE</i>	238
7.2	<i>Issues in the Use of Probes</i>	239
7.3	<i>Probes in Sequential Programs</i>	240
7.4	<i>Probe Insertion in Lotos</i>	241
7.4.1	A Simple Insertion Strategy	242
7.4.2	Improving the Probe Insertion Strategy	245
7.4.3	Interpreting Structural Coverage Results	247
7.4.4	Tool Support	249
7.5	<i>TTS Structural Coverage Results</i>	249
7.5.1	Summary of Coverage Results	250
7.5.2	Comments on LOLA and Missing Probes	252
7.6	<i>Discussion</i>	253
7.6.1	Compositional Coverage of the Structure	253
7.6.2	Specification Styles	254
7.6.3	Test Case Management Based on Structural Coverage	254
7.7	<i>Chapter Summary</i>	255

CHAPTER 8

Experiments with SPEC-VALUE	257
8.1 <i>Group Communication Server (GCS)</i>	257
8.1.1 System Overview and UCM Descriptions	258
8.1.2 Construction of the LOTOS Prototype	259
8.1.3 Test Selection and Execution	260
8.1.4 Structural Coverage	260
8.1.5 Discussion	261
8.2 <i>GPRS Group Call (PTM-G)</i>	262
8.2.1 System Overview and UCM Descriptions	262
8.2.2 Construction of the LOTOS Prototype	264
8.2.3 Test Selection and Execution	265
8.2.4 Structural Coverage	266
8.2.5 Discussion	267
8.3 <i>Feature Interactions (FI)</i>	269
8.3.1 System Overview and UCM Descriptions	271
8.3.2 Construction of the LOTOS Prototype	275
8.3.3 Test Selection and Execution	276
8.3.4 Structural Coverage	281
8.3.5 Discussion	281
8.4 <i>Agent-Based Simplified Basic Call (SBC)</i>	285
8.4.1 System Overview and UCM Descriptions	286
8.4.2 Construction of the LOTOS Prototype	288
8.4.3 Test Selection and Execution	289
8.4.4 Structural Coverage	290
8.4.5 Discussion	291
8.5 <i>Self-Coverage of GSM Mobile Application Part (MAP)</i>	292
8.5.1 Construction of the LOTOS Prototype	292
8.5.2 Test Generation	292
8.5.3 Structural Coverage	293
8.5.4 Discussion	294
8.6 <i>Test Suite Validation Using Mutation Analysis</i>	294
8.6.1 Mutation Analysis and Validation	295
8.6.2 Mutant Generation and SPEC-VALUE	296
8.6.3 Application to Case Studies	301
8.6.4 Discussion	304
8.7 <i>Chapter Summary</i>	305

CHAPTER 9

Conclusions and Future Work	313
9.1 <i>Hypothesis and Contributions</i>	313
9.1.1 Validation of the Research Hypothesis	313
9.1.2 Contributions of the Thesis	314
9.1.3 SPEC-VALUE and the Formal Specifications Maturity Model	318
9.2 <i>SPEC-VALUE and Related Methodologies</i>	319
9.2.1 Comparing SPEC-VALUE to Related Methodologies	320
9.2.2 Integrating SPEC-VALUE to Related Methodologies	324

	9.3 <i>Research Issues</i>	327
	9.3.1 Medium-Term Research Issues	327
	9.3.2 Long-Term Research Issues	328
	References	331
Appendix A:	UCM Quick Reference Guide	357
Appendix B:	LOTOS Specification of TTS	359
Appendix C:	Comparing Val And Conf	379
	Index	385

List of Figures

FIGURE 1.	Evolution Towards Requirements Engineering and System Design	1
FIGURE 2.	Specification-Validation Approach with LOTOS and UCMs (SPEC-VALUE) 8	
FIGURE 3.	Basic Notation and Interpretation	24
FIGURE 4.	Shared Routes and OR-Forks/Joins	24
FIGURE 5.	Concurrent Routes with AND-Forks/Joins, and Some Variations	25
FIGURE 6.	Dynamic Components and Dynamic Responsibilities.	25
FIGURE 7.	Stubs and Plug-ins	26
FIGURE 8.	Path Interactions.	27
FIGURE 9.	Timers, Aborts, Failures, and Shared Responsibilities	27
FIGURE 10.	Representation of a System Specified in LOTOS	30
FIGURE 11.	A Behaviour Expression and its LTS as a Behaviour Tree	35
FIGURE 12.	Connecting Several LOTOS Relations	39
FIGURE 13.	Illustration of Several Relations.	40
FIGURE 14.	Families of Concurrency Models	49
FIGURE 15.	Limit of Testability.	92
FIGURE 16.	Relations and Verdicts for Tests.	98
FIGURE 17.	From Requirements to UCMs with SPEC-VALUE.	113
FIGURE 18.	TTS Basic Call UCM.	119
FIGURE 19.	Individual UCMs for TTS Features	120
FIGURE 20.	Integrated UCM View of TTS	121
FIGURE 21.	From UCMs to LOTOS with SPEC-VALUE	126
FIGURE 22.	Construction of a LOTOS Specification from a UCM	130
FIGURE 23.	Interpreting AND-Joins and OR-Joins.	135

FIGURE 24.	Interpreting Stubs and Plug-Ins.	137
FIGURE 25.	Interpreting Timers	139
FIGURE 26.	Interpreting Aborts	140
FIGURE 27.	Interpreting Dynamic Responsibilities Linked to Pools	141
FIGURE 28.	Interpreting Dynamic Responsibilities Bound to Slots	142
FIGURE 29.	Containment and Plug-Ins.	146
FIGURE 30.	Situations with Unrelated Path Segments	148
FIGURE 31.	Channel Constraints and Valid Message Exchanges.	153
FIGURE 32.	Structure of the LOTOS Specification	160
FIGURE 33.	Construction of Process User	162
FIGURE 34.	Process Calling Tree for the Agent Component	163
FIGURE 35.	Extract from the Terminating Plug-in Process	164
FIGURE 36.	Binding of a Plug-in to a Stub in Process SO	165
FIGURE 37.	UCM-Based Testing with SPEC-VALUE.	172
FIGURE 38.	Partitioning of Acceptance and Rejection Test Groups and Test Cases . .	180
FIGURE 39.	Comparison Between val and conf	183
FIGURE 40.	UCM-Oriented Testing Pattern Language.	189
FIGURE 41.	Plug-in HandleStubs (Step 1)	190
FIGURE 42.	Plug-in HandleStartPoints (Step 2).	190
FIGURE 43.	Plug-in HandleConstructs (Step 3).	191
FIGURE 44.	Reference UCM: Testing Pattern for Alternatives	192
FIGURE 45.	Reference UCM: Testing Pattern for Concurrent Paths	195
FIGURE 46.	Reference UCM: Testing Pattern for Loops	197
FIGURE 47.	Reference UCM: Testing Pattern for Multiple Start Points	200
FIGURE 48.	Reference UCM: Testing Pattern for Single Stubs	207
FIGURE 49.	Reference UCM: Testing Pattern for Causally Linked Stubs	210
FIGURE 50.	Individual UCM in LOTOS with its LTS, Canonical Tester and Test Purposes	215
FIGURE 51.	Structural Coverage with SPEC-VALUE	238
FIGURE 52.	UCM and Responsibilities Information for “Initiate Call” Operation . . .	263
FIGURE 53.	Partial UCM for INTL.	272
FIGURE 54.	Global UCM and INTL Plug-in	273
FIGURE 55.	Construction of the FI Test Suite.	277
FIGURE 56.	A Feature Interaction Between TCS and INFB	280
FIGURE 57.	SimplifiedBasicCall UCM for SBC	287
FIGURE 58.	Structure of the LOTOS Specification	288
FIGURE 59.	Self-Coverage of the MAP Specification	293
FIGURE 60.	val vs. conf: Classification of Criteria and Associated Propositions. . . .	379
FIGURE 61.	Example Requirements, Canonical Tester, SUTs, and Test Suites	380

List of Tables

TABLE 1.	UCMs and LOTOS: Two Related and Complementary Notations.....	6
TABLE 2.	Roles of UCMs and LOTOS in SPEC-VALUE.....	9
TABLE 3.	Summary of LOTOS Syntax and Semantics	31
TABLE 4.	LTS Notation and Definitions	34
TABLE 5.	LOTOS Equivalence Relations	36
TABLE 6.	Other Relations for LOTOS.....	38
TABLE 7.	Evaluation of the Selected Specification Techniques	62
TABLE 8.	Benefits and Drawbacks of Scenarios.....	65
TABLE 9.	Comparison of the Selected Scenario Notations.....	73
TABLE 10.	Benefits and Drawbacks of Protocol Engineering Construction Approaches ..	76
TABLE 11.	Comparison of the Selected Construction Approaches	85
TABLE 12.	Notation for Test Definitions	97
TABLE 13.	Passes, Fails, and Failsall Relations.....	97
TABLE 14.	On the Partial Automation of Construction Guidelines.....	156
TABLE 15.	Notation for Test Purposes.....	177
TABLE 16.	Correspondence Between Templates for Design Patterns and Test Patterns ..	188
TABLE 17.	Truth Table for Multiple Start Points Example (from Figure 47).....	202
TABLE 18.	Notation for UCMs Interpreted in LOTOS.....	214
TABLE 19.	Test Goals Extracted from UCMs Through Testing Patterns	231
TABLE 20.	Further Test Goals for Robustness	232
TABLE 21.	Example of Probe Insertion in Pascal	240
TABLE 22.	Simple Probe Insertion in LOTOS.....	243
TABLE 23.	Underlying LTSs	244

TABLE 24.	TTS Structural Coverage Results	251
TABLE 25.	SBC Test Suite and Verdicts	290
TABLE 26.	TTS Mutants Generated Using Six Categories of Mutation Operators	299
TABLE 27.	Mutation Analysis of TTS and its Test Suite	302
TABLE 28.	Testing Pattern Strategies Used in TTS Test Cases.....	303
TABLE 29.	Mutation Analysis of GCS, PTM-G, and FI	304
TABLE 30.	Summary of Experiments with SPEC-VALUE.....	306
TABLE 31.	Construction Guidelines Usage.....	308
TABLE 32.	Testing Patterns Usage	310
TABLE 33.	Summary of the Formal Specifications Maturity (FSM) Model	319

List of Acronyms

Acronym	Definition	Page
ADT	Abstract Data Type	29
ANSI	American National Standard Institute	2
ASN.1	Abstract Syntax Notation One	55
BBE	Basic Behaviour Expression	242
BC	Basic Call	160
BE	Behaviour Expression	242
CADP	CÆSAR-ALDEBARAN Distribution Platform	42
CCS	Calculus of Communicating Systems	29
CEB	Communication Entity Block	286
CFSM	Communicating Finite State Machine	55
CG	Construction Guideline	125
CMM	Capability Maturity Model	5
CND	Calling Number Delivery	271
CO	Call Object	286
COMET	Concurrent Object Modelling and Architectural Design Method	325
CPN	Coloured Petri Net	57
CRESS	Chisel Representation Employing Systematic Specification	84
CREWS	Cooperative Requirements Engineering With Scenarios	69
CSP	Communicating Sequential Processes	29
CT	Canonical Tester	39
CTMF	Conformance Testing Methodology and Framework	94
DCT	Dynamic Causal Tree	50
DEB	Device Entity Block	286

EFSM	Extended Finite State Machine	55
EIA	Electronic Industry Association	17
E-LOTOS	Enhanced LOTOS	43
ELUDO	Environnement LOTOS de l'Université D'Ottawa	43
ETSI	European Telecommunications Standards Institute	2
FDT	Formal Description Techniques	5
FI	Feature Interaction	12
FIFO	First In First Out	159
FMCT	Formal Methods in Conformance Testing	93
FSM	Finite State Machine	55
FSM	Formal Specifications Maturity model	5
GCS	Group Communication Server	12
GPRS	General Packet Radio Service	12
GSM	Global System for Mobile Communication	13
HMSC	High-level MSC	56
IDL	Interface Description Language	55
IETF	Internet Engineering Task Force	2
IN	Intelligent Networks	3
INFB	IN Freephone Billing	271
INTL	IN Teen Line	271
IOFSM	Input/Output Finite State Machines	91
IP	Internet Protocol	12
ISO	International Organization for Standardization	2
ITU	International Telecommunications Union	2
LEB	Logical Entity Block	286
LIFO	Last In First Out	259
LOLA	LOTOS Laboratory	43
LOTOS	Language of Temporal Ordering Specification	5
LQN	Layered Queuing Network	115
LSC	Life Sequence Chart	71
LTS	Labelled Transition System	9
MAP	Mobile Application Part protocol	13
MS	Mobile Station	263
MS	Mutation Score	300
MSC	Message Sequence Chart	2
OCL	Object Constraint Language	58
OCS	Originating Call Screening	117
ODP	Open Distributed Processing	3
OMG	Object Management Group	52
OMT	Object Modeling Technique	81
OO	Object-Oriented	114
OS	Operating System	272

OSI	Open System Interconnection	18
PBX	Private Branch eXchange	129
PCO	Points of Control and Observations	70
PIN	Personal Identity Code	271
PLMN	Public Land Mobile Network	262
PN	Petri Net	57
POTS	Plain Old Telephone System	129
PTM-G	GPRS Point-to-Multipoint Group Call	12
RATS	Requirements Acquisition and spec. for Telecom. Services	72
ROOM	Real-Time Object-Oriented Modeling	55
RT-TROOP	Real-Time TRaceable Object-Oriented Process	78
SBC	Simplified Basic Call	12
SCP	Service Control Point	272
SDL	Specification and Description Language	55
SECM	Systems Engineering Capability Model	17
SPEC-VALUE	Specification-Validation Approach with LOTOS and UCMs	7
SUM	Synthesized Usage Model	78
SUT	Specification Under Test	93
SUT	System Under Test	39
TCS	Terminating Call Screening	271
TIA	Telecommunications Industry Association	2
TMDL	Timethread Map Description Language	128
TS	Test Suite	94
TTCN	Tree and Tabular Combined Notation	55
TTS	Tiny Telephone System	11
UCM	Use Case Map	5
UCMNav	UCM Navigator	28
UCT	Use Case Trees	70
UML	Unified Modeling Language	58
UORE	Usage Oriented Requirements Engineering	78
URN	User Requirements Notation	23
V&V	Validation and Verification	20
VDM	Vienna Development Method	55
XMI	XML Metadata Interchange	59
XML	eXtensible Markup Language	22

*À mes enfants,
Sandra, Jean-Luc, et Mireille,
qui ont dû partager leur père
avec ce livre
durant toute leur vie...*

CHAPTER 1

Introduction

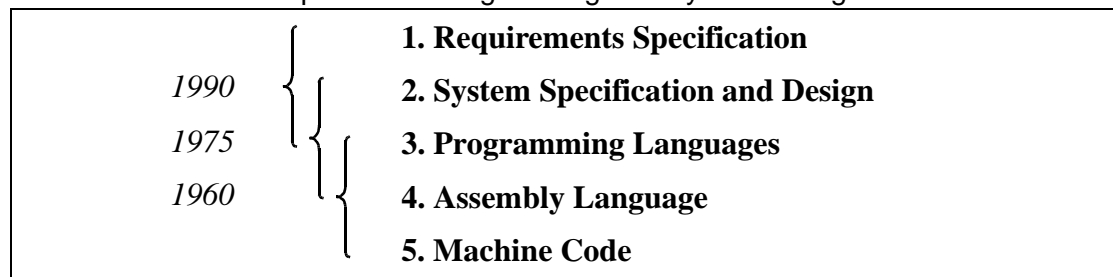
*Research is to see what everybody else has seen,
and to think what nobody else has thought.*

*Albert von Szent-Györgyi,
1937 Nobel Laureate in Medicine*

1.1 Motivation

The last few decades have resulted in an evolution of software design methodologies towards requirements engineering and high-level design, where the errors are the most costly for software producers. This trend was illustrated (see Figure 1) by Piotr Dembinski at FORTE 95 [103].

FIGURE 1. Evolution Towards Requirements Engineering and System Design



Requirements engineering has traditionally been concerned with investigating the goals, functions, and constraints of (software) systems. It can be broken down into four tasks: elicitation of information related to the problem domain; modeling of the problem; analysis of costs, completeness, and consistency; and validation with the customer. These tasks pave the way to the generation of com-

plete, consistent, and unambiguous specifications of system behaviour that are well suited for design and implementation activities [120]. When applied to reactive systems, requirements engineering tasks need to focus on behaviour rather than on input/output functions, the latter being more relevant to sequential systems.

Although several approaches have been suggested to tackle these requirements engineering tasks, many design processes still skip these tasks by jumping from informal requirements directly to component-based specifications and descriptions of systems. For instance, we observed this harmful situation in the telecommunications area, where complex distributed and reactive systems are designed and continuously enhanced with new *services* (often called *features* when packaged as marketable units). New telecommunications software products, involving increasingly complex architectures and protocols, are constantly being designed in industry and in standardization bodies (ANSI, ETSI, ISO, ITU, TIA, IETF, etc.). This is particularly true of new services for mobile, Internet-based, and agent-based communication. In the early stages of many conventional design processes used in industry and in standardization bodies, many features are described using (informal) component-based operational descriptions, tables, and visual notations such as *Message Sequence Charts* (MSCs) [208]. Whereas the focus should be on system and functional views, it is found to be on details belonging to a lower level of abstraction, or to later stages of the design process [20][21]. Hence, many requirements and high-level design decisions are buried in the details, and the understandability of the system goals and functionalities is affected in a way that makes the adaptability of services to particular legacy architectures very difficult [77]. Also, as these descriptions evolve, they quickly become error-prone and difficult to manage. One well-known error that can be detected at the requirements and design stages is the one of undesirable interactions between services (the so-called *feature interaction problem* [65][85][230]). It is well known that the cost of errors is much lower when found at the requirements and design stages and much higher when found during the implementation [290][335]. In the very competitive area of telecommunications, conventional approaches that focus too soon on details without proper description and understanding of requirements and high-level designs delay the introduction of new customer services and they increase the cost of their implementation [178].

There is an urgent need for high-quality documents that are concise, descriptive, maintainable, consistent, and understandable by readers with many different needs and perspectives, also known as the *stakeholders* (architects, engineers, testers, managers, marketing people, etc.). This need for precisely documenting all stages of the design process, which is significant in the industrial environment, becomes critical in the standardization process (e.g. I.130 [200] or Q.65 [204]), where there is international scrutiny for which stages are formalized and must undergo formal review and approval [20].

In this context, the following issues should be addressed:

- While designing systems and services in the initial stages, the discussion must focus on a level of detail that reflects the level of knowledge (about data, messages, components, etc.) available at the time. Irrelevant details tend to obscure the main idea behind a feature/service/functionality, especially when the latter needs further modifications or refinements.
- Several levels of abstraction similar to viewpoints in *Open Distributed Processing* (ODP) [195] and planes in *Intelligent Networks* (IN) [202] are often mixed in a single description.
- A simple visual notation, which abstracts from messages while focusing on the tasks to perform and their cause-to-effect (or *causal*) relations, can help concentrating on the general control flow while providing for more maintainable and reusable scenario descriptions. The Message Sequence Charts notation is very commonly used, but it focuses on components and message exchanges, which come into consideration later at the detailed design stages. Such a focus can be inappropriate while defining the functionalities in the initial stages of the design, when details related to messages and components might be unknown [21].
- There are possibly ambiguities, inconsistencies or undesirable interactions inside or between service descriptions, or between levels of abstraction of a given service. These remain difficult to detect with conventional inspection methods, and often remain hidden until errors are discovered after implementation, at which point corrections can be very costly and system interoperability can be jeopardized. Telecommunication networks are heterogeneous in nature (in age, capabilities, and implementation) and, unfortunately, con-

temporary open standards do not guarantee interoperability between systems developed by different vendors [19][120].

Over the years, several approaches have been used to provide such documents. On one hand, proponents of formal methods have claimed to solve the problem by providing unambiguous and mathematical notations and verification techniques, but the penetration of these methods in industry and in standardization bodies remains, unfortunately, low [20][103]. On the other hand, scenario-driven approaches, although often less formal, have raised a higher level of interest and acceptance, mostly because of their intuitive representation of services [215][368]. Their application to early stages of the design and standardization processes raises new hopes for the availability of concise, descriptive, maintainable, and consistent documents and design specifications that need to be understood by a variety of readers. A more rigorous approach, driven by scenarios and supported by a formal description technique, would allow the design process to focus on the main functional aspects of the system to be specified, and hence better to cope with some of the complex problems related to the design, documentation, validation, and maintenance of systems and standards. However, integrating individual scenarios in different ways may result in different kinds of unexpected or undesirable interactions. Appropriate integration techniques will hopefully lead to fewer such interactions.

The goal of this thesis is to provide techniques to describe and design distributed and telecommunication systems in a better way, through formal prototyping and validation. It also intends to fill the gap between the stage where services are described informally and the first formal specification of the system, which can be validated and then used to generate the kind of message sequence information currently found in component-based descriptions. The thesis presents a methodology where the level of scenario abstraction is different from the one used by most popular techniques. The approach focuses on the very first stage of design and standardization processes, where many information and design decisions are often lost or hidden behind implementation details. Such details should be omitted at this stage, whereas the general *causal flow of responsibilities* should be emphasized. Causality often expresses intentions at the requirements level. A prime goal of this thesis is hence to enable the description, formalization, and validation of telecommunications systems using causal scenarios.

1.2 Research Hypothesis

The process of going from informal functional requirements to a high-level formal specification is a research subject where much work has been done [20][62][39][103]. However, many challenges, such as the issues presented in the previous section, still remain. *Formal Description Techniques* (FDTs), such as LOTOS [191], Estelle [192] and SDL [205], were created in order to formally express functional requirements, and hence to answer some of these challenges [350]. In particular, FDTs are well suited for the precise definition of telecommunication systems. Although they help avoid ambiguities and inconsistencies, FDTs often require an inappropriate level of detail and completeness in the preliminary stages of standards definitions. Furthermore, as Ed Brinksma mentioned in his invited talk at FORTE'96, if we were to fit the current FDTs into a *Capability Maturity Model* (CMM) [175][278] adapted for formal methods, we would still be at the first level (referred to as *initial* and sometimes as *anarchy*). In fact, a concrete *Formal Specifications Maturity model* (called the *FSM* model) was suggested by Fraser and Vaishnavi [138] a year later. This FSM model describes five levels of maturity, identical to those of CMM: 1) initial, 2) repeatable, 3) defined, 4) managed, and 5) optimized. In this model, reviewed in more details in Section 9.1.3, the sole use of an FDT by experts corresponds to the initial (first) level of maturity.

In this thesis, we present an innovative approach where we combine an FDT to a semiformal visual notation for causal scenarios called *Use Case Maps* (UCMs) [74][76]. UCMs use paths that *causally* link activities (called *responsibilities*), which can be bound to underlying organizational structures. Leyton observes that the mind assigns to any shape (including functionary) a causal history explaining how the shape was formed [245], and UCMs capture such shapes visually and help arguing about causality at an abstract level. Based on the literature and on previous experiences with the notation, we assume that UCMs can be used to represent and integrate important aspects of functional requirements for telecommunications services [21][22][24][25][77][78]. Integrating UCMs together can also help avoiding many undesirable interactions between services before any prototype is generated. The selected FDT is the formal specification language LOTOS [191], the *Language of Temporal Ordering Specification*. LOTOS possesses powerful testing concepts and tools that we use for the detection of logical and design errors. LOTOS is an appropriate formalism here because it supports

many UCM concepts directly, and it complements most of UCM’s weak areas related to the analysis of systems.

Table 1 summarizes how UCMs and LOTOS can be seen as complementary notations. At the same time, they share common characteristics which make them a good match. This claim will be supported in the literature review (Chapter 3).

TABLE 1. UCMs and LOTOS: Two Related and Complementary Notations

Use Case Maps	LOTOS
<ul style="list-style-type: none"> • Causal scenario notation (semi-formal) • Readable, graphical • Abstract • Scalable • Loose • Relatively effortless to learn 	<ul style="list-style-type: none"> • Mature formal language • Unambiguous • Good theories and tools for: <ul style="list-style-type: none"> - Consistency and completeness checking - Testing and simulation - Validation and verification
Both Notations	
<ul style="list-style-type: none"> • Focus on ordering of actions • Have similar constructs, which simplifies the mapping of UCMs onto LOTOS • Can handle behaviour descriptions with or without components • Have been used to describe telecommunications systems in the past • Have been used to detect feature interactions in the past 	

The research hypothesis is denoted as follows:

In the process of designing complex telecommunications systems, requirements described using the Use Case Map causal scenario notation can guide the generation of LOTOS specifications useful for validating high-level designs systematically through numerous techniques, including functional testing based on UCMs.

By supporting the engineering of requirements with tools and techniques developed for the engineering of systems (UCMs) and of protocols (LOTOS), this approach aims to help producing better-quality designs and to improve human understanding with reduced time, costs, and efforts. We also believe that this solution can help to reach higher levels on the Formal Specifications Maturity

(FSM) model scale, and this assessment will be concentrated in the conclusion (Section 9.1.3) rather than being spread throughout the core chapters.

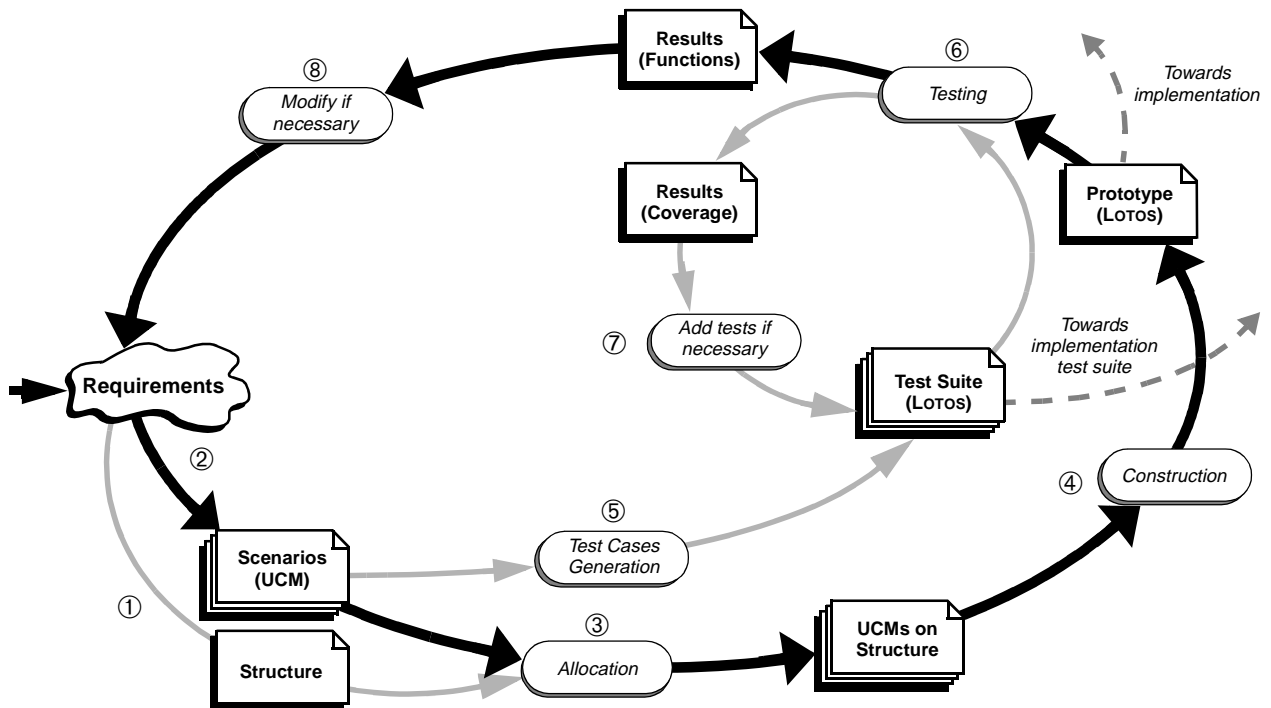
1.3 New Approach: SPEC-VALUE

We believe that using UCMs in a scenario-oriented approach represents a judicious choice for the description of communicating and reactive systems. They fit well in the design approach proposed in this thesis, the *Specification-Validation Approach with LOTOS and UCMS* (or *SPEC-VALUE*) methodology. SPEC-VALUE aims to improve the maturity of design processes based on formal specifications by introducing a semiformal description (UCM) between informal requirements and design-oriented formal specifications (LOTOS). Such an improvement strategy is called “Transitional-Unassisted” in the FSM model (see Table 33 on page 319). We intend to validate the research hypothesis by developing SPEC-VALUE and by successfully applying it to a wide range of telecommunications applications.

Requirements are usually dynamic; they change and are adapted over time. This is why we promote an iterative and incremental approach (in spiral form) that allows rapid prototyping of abstract behaviour and test case generation directly from scenarios. Figure 2 presents such an approach and introduces the main concepts behind SPEC-VALUE. Its main cycle is first concerned with the description of system structures ① and scenarios ②, which can be done independently. A structure, also called component substrate, contains the abstract system components of interest (mostly software, but also hardware), as well as some of their relationships (containment, communication links, etc.). Services and functionalities are captured as Use Case Maps (scenario elicitation). These UCMs represent scenarios emphasizing the causal relationships among the responsibilities that compose services and large-grain functionalities. The responsibilities defined in the UCMs are then allocated to the components in the selected underlying structure ③. Each component will have to perform the responsibilities allocated to it. Next, the scenarios are combined (manually, in this thesis) to synthesize a LOTOS specification ④, which becomes the executable prototype enabling formal validation of the system’s abstract behaviour ⑥.

Concurrently with these steps, validation test cases can be generated from the individual scenarios ⑤ to ensure that the specification conforms to each intended functionality. The test cases are described in the same language as the specification, i.e. LOTOS. These tests check the integration of the functionalities, which is currently done manually in our approach. They also check that the integrated behaviour emerging from the collaboration among the components in the system structure corresponds to the intended behaviour expressed by the UCMs. Although both the tests and the prototype are generated from the same UCMs, discrepancy often result. This is largely due the complexity of scenario integration in a component-based prototype and to the numerous design decisions that are required at that level. Tests are much simpler to derive and are more likely to be correct. Also, because UCMs are close to the functional requirements, verifying that the LOTOS specification conforms to the UCMs is a way of validating the high-level design against the requirements.

FIGURE 2. Specification-Validation Approach with LOTOS and UCMs (SPEC-VALUE)



Probes can be inserted in the specification to measure its structural coverage by the test suite. Once the specification has been tested against all the test cases, results and statistics ⑥ can be

obtained automatically from the execution traces (described as *Labelled Transition Systems*, or LTSs). The test results indicate whether logical and design errors (including undesirable interactions) are present, while the coverage results determine whether some part of the specification is unreachable, or whether the test suite is incomplete ⑦. If no problem is detected, then the specification conforms, according to the test selection strategy, to the UCMs and hence to the functional requirements. Such coherence increases substantially the level of confidence in the UCMs, in the LOTOS specification, and in the validation test suite. Following the verdict, modifications may be required to the UCMs, to the test cases ⑦, to the specification, or even to the requirements ⑧. In fact, SPEC-VALUE is an iterative and incremental approach as new functionalities may be integrated at a later time.

Table 2 summarizes, in the context of SPEC-VALUE, the roles of these two complementary languages. These items will be explained and illustrated throughout the thesis.

TABLE 2. Roles of UCMs and LOTOS in SPEC-VALUE

Use Case Maps	LOTOS
<ul style="list-style-type: none"> • Requirements capture and scenario elicitation in terms of related causal flows • Architectural reasoning • Bridge to design • Feature interaction avoidance • Design documentation • Generation of abstract test cases • Basis for the generation of abstract prototypes 	<ul style="list-style-type: none"> • Executability and increased formality enabling system analysis • Validation of requirements through (functional) testing • Detection of inconsistencies and other logical errors • Detection of incompleteness in the requirements through coverage measurements • Feature interaction detection • Production of validated test suites

Further, some of the products generated by SPEC-VALUE can be used in later steps the development cycle, towards implementation (dashed arrows in Figure 2). In particular, a LOTOS prototype can be refined into a form suitable for code generation using conventional SPEC-VALUE tools, or else be used for conformance testing. Similarly, abstract LOTOS test cases can be transformed into concrete functional test cases for the validation of detailed designs and implementations. These applications are related issues however belong to development processes with a scope wider than SPEC-VALUE's and are therefore not discussed in this thesis.

1.4 Thesis Contributions

This thesis offers three main contributions: the SPEC-VALUE methodology, a set of techniques to support the SPEC-VALUE cycles, and application of SPEC-VALUE to specify and validate several complex telecommunications applications.

1.4.1 Contribution 1: SPEC-VALUE Methodology

We claim that SPEC-VALUE has several benefits, difficult to find all at once in other design and standardization processes:

- **Separation of the functionalities from the underlying structure:** since scenarios are formalized at a level of abstraction higher than message exchanges, different underlying structures or architectures can be evaluated with more flexibility, even before the generation of a prototype. The scenarios then become highly reusable entities; for example they can be used on different equipment and for different products. Senior managers and senior designers can keep control over the general logic of the design without having to know the characteristics of the latest equipment. This separation helps to cope also with the incremental addition of new functionalities that require modifications to the structure.
- **Fast prototyping:** once the structure and the scenarios are selected and documented, and once the responsibilities have been allocated to their respective components, a prototype (the first formal specification of the system abstract behaviour) can be generated rapidly. This is mainly due to the ease with which LOTOS constructs can formalize UCM constructs. Formal prototyping adds rigor to scenario-based requirements and engineering with UCMs because UCMs are semiformal and non-executable.
- **Test case generation:** scenarios guide the generation of test cases, hence allowing the verification of the prototype against the UCMs and its validation against the informal functional requirements. The test suite can itself be validated using structural coverage criteria on the model. It can be reused as a basis for functional or regression test suite in the subsequent steps of the development process.

- **Design documentation:** the documentation of requirements and designs is done as we go along the development cycle. It is also adapted to the expressive needs of the different people involved in the design process. The part related to scenarios should be understandable by marketing people and service operators. These people do not have to know every technical detail described in the subsequent formal specifications (such as message exchanges), since such details may be important only for engineers, implementors, or testers. UCMs allow different specialists to become involved in discussions at different levels while sharing a common language and, hopefully, understanding.

More details on the SPEC-VALUE methodology, as well as UCMs for the ongoing example (the *Tiny Telephone System*, or *TTS*), are provided in Chapter 4.

1.4.2 Contribution 2: Theories and Techniques Supporting SPEC-VALUE

Different theories and techniques are involved in the support of the SPEC-VALUE cycles. Some of them, such as theories and tools for the testing of LOTOS specifications [279] (step © in Figure 2) and for the visual editing of UCMs [257] (steps ①, ②, and ③ in Figure 2), already exist. Others are developed in this thesis:

- **Guidelines for the construction of LOTOS specifications from UCMs:** in his masters thesis, Amyot provided a mapping between UCM paths (called Timethreads at the time) and LOTOS [12]. This mapping is extended in this thesis to cover component-based specifications, which better reflect the design, and to cover the new UCM constructs developed over the last six years. Among others, new construction guidelines are provided for UCMs that have responsibilities bound to components and for UCMs with stubs and plug-ins (sub-UCMs). These construction guidelines, which relate to step ④ in Figure 2, are developed in Chapter 5.
- **UCM-LOTOS testing framework:** test cases can be derived from UCMs and applied to specifications and implementations in order to check their conformance to the UCMs and their validity with respect to the requirements. The thesis presents a new validation rela-

tion (val) accompanied by a set of testing patterns, which contain test selection strategies for UCMs. These patterns serve as a basis for the evaluation of functional coverage in terms of UCMs. This framework, related to step ⑤ in Figure 2, is developed in Chapter 6.

- **Structural coverage for LOTOS:** the thesis presents a new technique for automatically measuring the structural coverage of LOTOS specifications by a test suite. This technique includes a tool-supported theory for the insertion of probes in LOTOS specifications and for coverage measurement. The structural coverage theory, which relates to step ⑦ in Figure 2, is developed in Chapter 7.

1.4.3 Contribution 3: Illustrative Experiments Validating SPEC-VALUE

The SPEC-VALUE approach and its supporting techniques have been validated against a wide range of telecommunications applications. Chapter 8 includes results and lessons learned from six experiments:

- **Group Communication Server (GCS):** first application of the SPEC-VALUE approach and techniques in their entirety [15][17]. The GCS is an academic example of a distributed client/server application.
- **GPRS Group-Call (PTM-G):** application of SPEC-VALUE a mobile communication service of the *General Packet Radio Service* (GPRS) standard [16][24]. This work was done during the first standardization stage of GPRS [128].
- **First Feature Interaction Contest (FI):** application of SPEC-VALUE targeted towards the avoidance and the detection of undesirable interactions between a collection of telephony features described in the 1998 Feature Interaction Contest [18][22][161].
- **Agent-Based Simplified Basic Call (SBC):** application of SPEC-VALUE during a feasibility study for the application of a functional testing process to industrial telephony applications based on agents and IP [25][369].
- **GSM's MAP Protocol (MAP):** application of the probe insertion technique to measure the self-coverage of a conformance test suite generated automatically from a LOTOS speci-

fication. The *Mobile Application Part* (MAP) protocol of the *Global System for Mobile Communication* (GSM) standard is used as the example.

- **Test Suite Validation:** application of mutation analysis to the above specifications in order to validate the various test suites generated using the UCM-oriented testing patterns. This experiment discusses the effectiveness of these test suites.

Most of these experiments were done in collaboration with industrial partners, professors, and other students. The SPEC-VALUE approach is currently being used by other graduate students in other projects and theses [25][32][381].

1.4.4 Issues Not Addressed in this Thesis

The reader should be warned that Use Case Maps and the SPEC-VALUE methodology are still quite young and maturing. A consequence is that many interesting and important issues are not addressed in this thesis:

- The use of UCMs for capturing requirements and eliciting system scenarios is not discussed as such, although many illustrations and guidelines are given in the various experiments.
- The automated synthesis of the LOTOS specification from UCMs, although much desired by many people, is not a goal of this thesis. Often, automated mappings require the restricted use of the two notations involved. We take the point of view that flexibility should be allowed in both notations, at the cost of a manual transformation, with consistency ensured through validation and conformance testing.
- The automated generation of test cases from UCMs is not covered by this thesis either.
- The testing used here is functional (black-box). It is targeted towards the user-system level. Component or unit testing are not addressed in the thesis.

1.5 Thesis Outline

The rest of the thesis is divided into nine chapters:

- Chapter 2 presents general definitions of concepts used throughout the thesis as well as introductions to Use Case Maps and LOTOS.
- Chapter 3 is the literature review that covers the necessary background information on causality, on scenarios, on formal techniques, and on validation and verification.
- Chapter 4 details the first steps of SPEC-VALUE methodology and introduces the UCMs for the *Tiny Telephone System* (TTS), an ongoing example used throughout the thesis.
- Chapter 5 presents the construction guidelines for the generation of LOTOS specifications from UCMs, illustrated with the TTS example.
- Chapter 6 describes the UCM-based testing framework used to validate LOTOS models, illustrated with the TTS example.
- Chapter 7 defines the probe insertion technique used to measure the structural coverage of LOTOS specifications, illustrated with the TTS example.
- Chapter 8 presents six experiments used to validate the SPEC-VALUE methodology and techniques. It contains the lessons learned during the specification and the validation of telecommunications systems of various complexity and natures.
- Chapter 9 recalls the contributions of the thesis, compares the SPEC-VALUE methodology to similar approaches, and attempts to provide new insights in how to integrate SPEC-VALUE to design processes with a wider scope. This chapter concludes with some directions for future research.

These chapters are meant to be read linearly, although the definitions (Chapter 2) and some sections in the literature review (Chapter 3) can be skipped at first and referred to at a later time when necessary. Due to the incredibly large number of acronyms and technical terms found in this thesis (which is typical of telecommunications systems and standards), we included a glossary of acronyms on page xv, together with an index on page 385.

CHAPTER 2

Basic Definitions and Notations

Description is important because it is the clay in which software developers fashion their works. Methods are, above all, about what to describe; about tools and materials and techniques for descriptions; and about imposing a coherent structure on a large description task.

Michael Jackson, 1995

This chapter provides general definitions of concepts used throughout the thesis as well as introductions to Use Case Maps and LOTOS.

2.1 Basic Definitions

The SPEC-VALUE methodology combines ideas from many disciplines with different cultures and, often, different semantics associated to the same terminology. Computer scientists, systems engineers, telecommunications engineers, formalists, and defense organizations often have their own conventions and standards involving different definitions for the same terms. Throughout the thesis, several definitions will be given when specific terminology problems will be encountered. This section however focuses on more basic definitions and concepts, some of which have been used in an intuitive way in the introductory chapter. In particular, four disciplines related to this thesis will be outlined, and definitions will be provided for various terms including requirements, specification, design, process, prototype, validation, and verification.

2.1.1 Four Engineering Disciplines

The SPEC-VALUE methodology involves notations from systems engineering (Use Case Maps) and protocol engineering (LOTOS), which are used for the engineering of requirements, systems, and software in general. The main characteristics of these disciplines are stated in this section.

Requirements Engineering

Requirements Engineering is the development and use of cost-effective technology for the elicitation, specification, and analysis of the stakeholder requirements, which are to be met by software intensive systems [308]. Zave provides an interesting classification of the research effort in this area [384].

A *requirement* is something that states that a product will have a given characteristic or achieve a given purpose, including what, how well, and under what conditions [122]. Requirements can address software and non-software (e.g. hardware) issues. Requirements are usually classified as *functional* (defining functions of the system under development) or as *non-functional* (to characterize expected performance, robustness, usability, maintainability, etc.). For instance, performance requirements describe how well system products must perform certain functions along with the conditions under which the functions are performed. Functional requirements are sometimes seen as operationalizations of non-functional requirements. After decades of focus on functional requirements, non-functional requirements are nowadays the topic of much interest [96]. It is important to note that requirements define the problem to be solved by software, not the software that solves it.

A *stakeholder* is an individual or organization interested in the success of a product or system. Stakeholders include customers, users, developers, engineers, managers, manufacturers, testers, and so on.

The functionalities of telecommunications applications and of reactive systems in general are expressed more often in terms of system and component behaviour than in terms of algorithms or input/output functions. Use Case Maps have proven to be useful for the engineering of telecommunications systems requirements [22]. They help capturing and illustrating behavioural requirements at the system level. To a lesser extent, non-functional issues related to architecture, robustness and per-

formance can also be addressed, quantified and reasoned about [76][324][330]. Representing requirements as visual scenarios, UCMs can also be understood by various stakeholders.

Systems Engineering

Systems Engineering is an interdisciplinary approach enabling the realization of successful systems [122]. The term *system* usually refers to the aggregation of end products/technologies and enabling products/technologies that achieves a given purpose.

Systems Engineering focuses on defining customer needs and required functionality early in the development cycle, on documenting requirements, and then on designing and validating systems. At the same time, system issues like operations, performance, test, manufacturing, cost, schedule, training, support, and disposal are considered. Systems Engineering integrates all the disciplines and speciality groups into a team effort forming a structured development process that proceeds from concept to production to operation. This engineering field considers both the business and the technical needs of all customers with the goal of providing a quality product that meets the user needs. Systems Engineering is still a maturing domain where several standards such as EIA's *Systems Engineering Capability Model* (SECM) [122] are emerging.

Use Case Maps were created as a notation for systems engineering, with a focus on the representation of high-level design decisions [74].

Protocol Engineering

Protocol Engineering is the efficient use of trusted components, (formal) methods and tools to construct an integrated architecture of devices and processes which collaborate to provide desired communications services while satisfying constraints such as cost, time, reliability and safety [296]. Communication *protocols* are the rules that govern the communication between the different components within a distributed computer system [53][180].

The first attempt at defining this discipline was done by Piatkowski in 1980 [285], and Liu provided an early survey in [248]. While the construction of valid, safe, and efficient protocols represents the main goal of protocol engineering, the importance of service concept was recognized and emphasized by Vissers and Logrippo [363]. According to Saleh [316], this engineering field includes many specific areas such as formal specification of protocols and services, protocol validation, protocol synthesis (from service specifications), protocol implementation, protocol conformance testing, protocol conversion, protocol performance analysis, quality assurance, and so on.

Nowadays, the engineering of communication protocols often involves the use of formal description techniques such as LOTOS, which was specifically created for the formalization and validation of protocols in the *Open System Interconnection* (OSI) reference model [194].

Software Engineering

Software Engineering is the study and application of systematic, disciplined, quantifiable approaches to the development, operation, and maintenance of software [187][335]. Software Engineering is not independent from the three other engineering disciplines. All four share many goals related to the quality and the cost of products (usually software) and to the satisfaction of the users. Most aspects of Software Engineering and their research directions are being discussed in [136].

SPEC-VALUE intends to bring these areas closer together by combining notations and techniques (UCMs and LOTOS) from two of these areas and by using them as a bridge between the engineering of requirements, systems, protocols, and software.

2.1.2 Processes, Formal Methods, Specifications and Designs

The thesis makes extensive use of terms like process, formal method, specification, and design. Several definitions, which may vary depending on the context, are provided below.

In a general (system engineering) context, a *process* is a set of interrelated activities that, together, transform inputs into outputs [122]¹. The *process maturity* represents the extent to which a process is explicitly documented, managed, measured, controlled, and continually improved. Models

like CMM [278], SECM [122], and FSM [138] can help measuring this maturity. One of the goals of SPEC-VALUE is to improve the maturity of processes to which it is integrated.

Many protocol engineering and software engineering development processes suggest the use of *formal methods* to improve their maturity. Formal methods are techniques for expressing requirements in a manner that enables the requirements to be studied mathematically. They allow sets of requirements/descriptions to be examined for completeness, consistency, and equivalency to other sets of requirements/descriptions.

Formal methods are used to produce formal specifications. A *specification* is a document that clearly and accurately describes requirements and other characteristics for a product and the procedures to be used to determine that the product satisfies these requirements [122]. A specification is qualified as formal when it is written using a formal language [241]. Specifications are usually split into two categories: *requirements specifications*, which focus on the problem domain (the “what”), and *software specifications*, which focus on the description of the design in conformance with the requirements specification (the “how”). Specifications describe user functionalities, also called *services*. When services are packaged into marketable units, the telecommunication industry calls these services *features*.

Some specifications become *standards*, as they are documents that establish engineering and technical requirements for processes, procedures, practices and methods that have been decreed by authority or adopted by consensus. Guidelines for the creation of software and system specifications are provided by the IEEE in [188][189].

Specifications are used to construct more detailed descriptions called *designs*. Typically, a design includes an operational concept (how users are expected or intended to use the product), components and their relationships, and sometimes decisions about the processes that will produce, deploy, and support it. We often distinguish between *high-level designs*, which focus on system func-

1. However, in terms of LOTOS, a process is a behavioural abstraction that can be instantiated.

functionalities and end-to-end scenarios, and *low-level designs*, where detailed issues related to protocols and algorithms are handled. Designs have to be valid with respect to their specification and requirements. Scenarios can be used to describe system functionalities, often from the user's point of view, and to capture several aspects of requirements and designs.

Designs make reference to the *system architecture*, representing the logical or physical structure that specifies interfaces and services provided by the *system components* used to accomplish system functionality. System components may be personnel, hardware, software, facilities, data, materiel, services, or techniques which satisfy one or more requirements in the lowest levels of the architecture [122].

2.1.3 Validation and Verification

A product needs to be checked against its design, which needs to be checked against the specification, which needs to be checked against the requirements. These activities are divided into two main categories, namely validation and verification (V&V).

Validation is an activity that ensures that the stakeholders' true needs and expectations are met by the end product. In other words, validation is the determination of the correctness of the final product with respect to the user's needs (hopefully captured correctly by the requirements), or "Are we building the right product?". This concept can be extended to the validation of the design and of the specification.

Verification is an activity that ensures that the selected design solution satisfies the specification, and that the end product satisfies the design. Ultimately, verification is about determining whether the product fulfills the requirements established, or "Are we building the system right?".

V&V is usually applied to a product or to a *model*, the latter being a simplified representation of some aspect of the real world. Specifications and designs are models, and so are prototypes. A *prototype* is a model of a product built or constructed for the purpose of: assessing the feasibility of a new or unfamiliar technology; assessing or mitigating technical risk; validating requirements; demonstrat-

ing critical features; qualifying a product; qualifying a process; characterizing performance or product features; or elucidating physical principles [122]. A prototype normally will not be identical to the final product in all its characteristics. Rather, it will correspond to it only for the characteristics that motivated its construction. Executable specifications, designs and other mathematical models can be used to prototype software [275]. This idea was pioneered nearly two decades ago by people like Davis [109] and Balzer *et al* [41]. In this thesis, LOTOS specifications will be used to prototype telecommunications systems.

Testing is one of the most pragmatic and most popular V&V technique. A *test* is an activity in which a system, product, prototype, or a component is used under specified conditions, the results are observed or recorded, and an evaluation is made as to whether it adequately meets some or all of its requirements. Tests are used for different purposes, such as validation testing (the focus of this thesis), unit or component testing, and regression testing, which is used to determine that a change to a system component has not adversely affected functionality, reliability or performance.

One way of reducing the required V&V efforts is to construct a model or a product from a more abstract model. The *synthesis* is the translation of input requirements (including performance, function, and interface) into possible solutions (resources and techniques) satisfying those inputs. Synthesis does not have to be automatic. Interactive and incremental synthesis is also possible.

Transitioning from one model to the other and performing V&V on them demand some way of tracing related elements among the models. This particularity, called *traceability*, is the ability to trace the heritage and lineage of a requirement. Traceability shows upward compliance of derived requirements/specifications/designs/products with higher level parent requirements and downward completeness of derived requirements/specifications/designs/products from higher level parent requirements.

The following sections, together with the other chapters, will make use of all the general definitions discussed so far. Others will be provided when required.

2.2 Introduction to Use Case Maps

This introduction is intended to provide an overview of Use Case Maps to people who want to familiarize themselves with this notation. This section presents the philosophy behind the notation and the information necessary to use it. Then, the basic path notation is introduced, followed by part of the component notation used in this thesis. Advanced notational concepts are then outlined, and finally supporting tools are discussed. A summary of the UCM notation is provided as a quick reference guide in Appendix A:

2.2.1 Philosophy of UCMs

The Use Case Map notation aims to link behaviour and structure in an explicit and visual way. *UCM paths* are first-class architectural entities that describe *causal* relationships between *responsibilities*, which can be bound to underlying *organizational structures* of abstract *components* [76]. These paths represent scenarios that intend to bridge the gap between system requirements and detailed design [30]. The relationships are said to be causal because they involve concurrency and partial ordering of activities and because they link causes (e.g., preconditions and triggering events) to effects (e.g. postconditions and resulting events).

With UCMs, scenarios are represented in terms of abstract responsibilities expressed above the level of messages exchanged between components. Hence, these scenarios are not necessarily bound to a specific organizational structure. This feature promotes the evaluation of architectural alternatives early in the design process. UCMs provide a bird's-eye view of system functionalities, they allow for dynamic behaviour and structures to be represented and evaluated, and they improve the level of reusability of scenarios [24].

Use Case Maps are primarily visual, but a formal textual representation also exists. Based on the *eXtensible Markup Language* (XML) 1.0 standard [379], this representation allows for tools to generate UCMs or use them for further processing and analysis [23].

The UCM notation was developed at Carleton University by Professor Buhr and his team, and it has been used for the description and the understanding of a wide range of complex applications

(including telecommunication systems) since 1992. UCMs have raised a lot of interest in the software community, which led to the creation of a user group in March 1999, with over 200 members from all continents [359]. UCMs are also being considered as a notation for describing functional requirements in ITU-T's upcoming standard, the *User Requirements Notation* (URN) [28][82][84][203].

2.2.2 Information Needed to Construct UCMs

UCMs can be derived from informal requirements, or from use cases [212] when they are available. *Responsibilities* need to be stated or be inferred from these requirements. For illustration purpose, separate UCMs can be created for individual features, or even for individual scenarios. However, the strength of this notation mainly resides in the integration of scenarios.

It is important to clearly define the *interface* between the environment and the system under description. This interface will lead to the start points and end points of the UCM paths, and it also corresponds to the messages exchanged between the system and its environment. These messages are further refined in models for detailed design (e.g. with Message Sequence Charts).

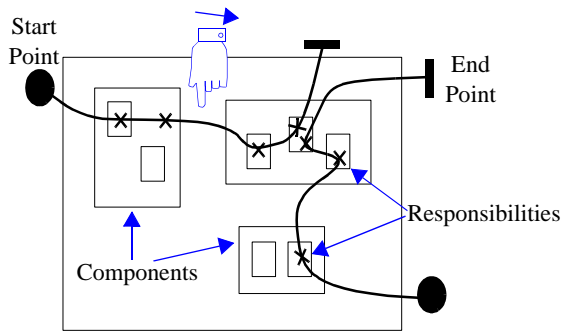
UCM can be composed of paths where responsibilities are not bound to any component. These scenarios, called *unbound UCMs*, are useful as they describe system functionalities independently of the architecture. However, because designers are often the people who create and use UCMs, some design information such as internal components may be relevant. In this case, the description of these components, their nature, and some relationships (e.g. components that include sub-components) are required. Communication links between components are usually not required, but they can be added.

2.2.3 Basic UCM Path Notation

The UCM notation is mainly composed of *path elements*, and also of *components*. The basic path notation addresses simple operators for causally linking responsibilities in sequences, as alternatives, and in parallel. More advanced operators can be used for structuring UCMs hierarchically and for representing exceptional scenarios and dynamic behaviour (Section 2.2.5). Components can be of different natures, allowing for richer description of some entities in a system (Section 2.2.4).

Figure 3 illustrates four basic notation elements of UCMs: *start points*, *responsibilities*, *end points*, and *components*. In this section, simple boxes are used as components.

FIGURE 3. Basic Notation and Interpretation



Imagine tracing a path through a system of objects to explain a causal sequence, leaving behind a visual signature. Use Case Maps capture such sequences. They are composed of:

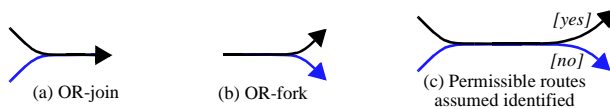
- **start points** (filled circles representing preconditions and/or triggering causes)
- causal chains of **responsibilities** (crosses, representing actions, tasks, or functions to be performed)
- and **end points** (bars representing postconditions and/or resulting effects).

The responsibilities can be bound to **components**, which are the entities or objects composing the system.

The wiggly lines are *paths* that connect start points, responsibilities, and end points. A responsibility is said to be *bound* to a component when the cross is inside the component. In this case, the component is responsible to perform the action, task, or function represented by the responsibility. Start points may have preconditions attached, while responsibilities and end points can have postconditions. We call *route* a scenario that traverses paths and associated responsibilities from a start point to an end point.

Alternatives and shared segments of routes are represented as overlapping paths (Figure 4). An *OR-join* merges two (or more) overlapping paths while an *OR-fork* splits a path into two (or more) alternatives. Alternatives may be guarded by conditions represented as labels between square brackets.

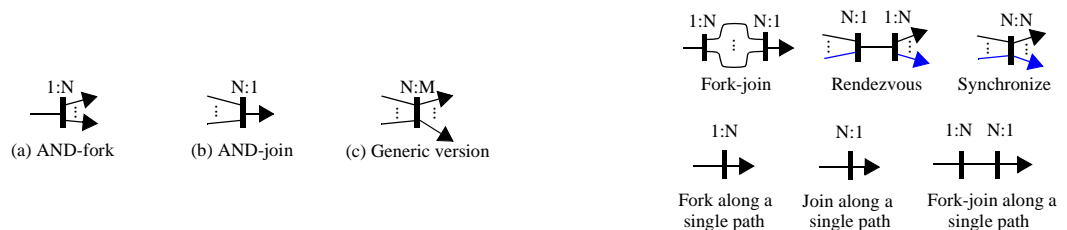
FIGURE 4. Shared Routes and OR-Forks/Joins



Indicate routes that share common causal segments. alternatives may be identified by labels or by conditions (*[guards]*)

Concurrent and synchronized segments of routes are represented through the use of a vertical bar (Figure 5). An *AND-join* synchronizes two (or more) paths together while an *AND-fork* splits a path into two (or more) concurrent segments. Cardinalities ($N:M$) are not required to be written as they usually result from the number of incoming/outgoing path segments.

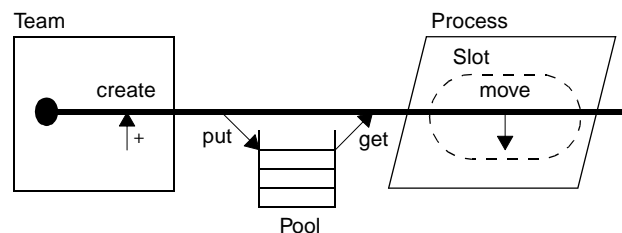
FIGURE 5. Concurrent Routes with AND-Forks/Joins, and Some Variations



2.2.4 UCM Component Notation

Components can be of different *types* and can possess different *attributes*. Although many component notations could be used underneath UCM paths, Buhr suggests several types and attributes relevant for complex systems (real-time, object-oriented, dynamic, agent-based, etc.) [74][76]. The UCM Quick Reference Guide (Appendix A: — A8 and A9) illustrates all the component types and attributes in Buhr's notation. Some of the most interesting ones are illustrated in Figure 6.

FIGURE 6. Dynamic Components and Dynamic Responsibilities



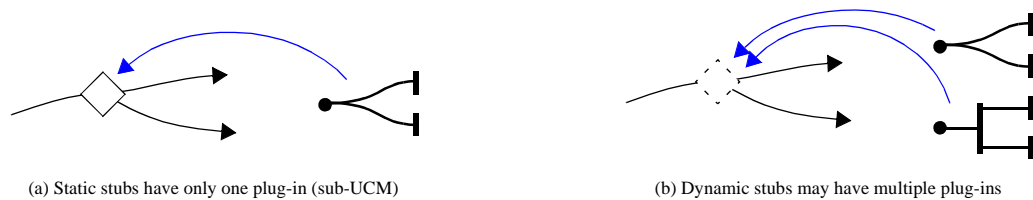
Rectangles are called *teams* and are allowed to contain components of any type. This is a default/generic component used in most UCMs. Parallelograms are active components (*processes*), which usually imply a control thread, whereas rounded rectangles are passive components (*objects*)

which are usually controlled. Dashed components are called *slots* and may be populated with different component instances at different times. Slots are containers for *dynamic components* (DC) in execution, while *pools* are containers for DCs that are not executing (they act as data). Dynamic components can be created, moved, stored, and deleted with *dynamic responsibilities* (see Appendix A: — A10) such as create, put, get, and move in Figure 6.

2.2.5 Advanced UCM Path Notation

When maps become too complex to be represented as one single UCM, a mechanism for defining and structuring sub-maps becomes necessary. Top-level UCMs, called *root maps*, can include containers (called *stubs*) for sub-maps (called *plug-ins*). Stubs are of two kinds (Figure 7):

FIGURE 7. Stubs and Plug-ins

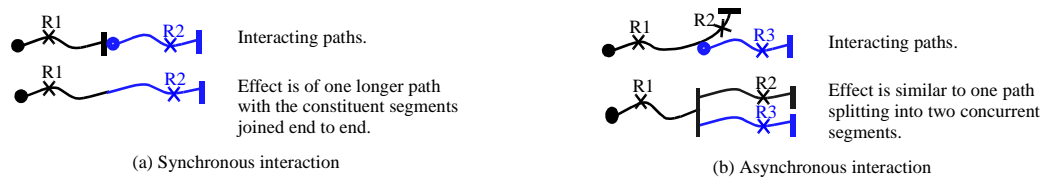


- **Static stubs:** represented as plain diamonds, they contain only one plug-in, hence enabling hierarchical decomposition of complex maps.
- **Dynamic stubs:** represented as dashed diamonds, they may contain several plug-ins, whose selection can be determined at run-time according to a *selection policy* (often described with preconditions). It is also possible to select multiple plug-ins at once, sequentially or in parallel.

Path segments coming in and going out of stubs can be identified on the root map. Although they are not required to be shown visually, their presence helps to achieve unambiguous *bindings* of plug-ins to stubs. A binding is a set of pairs $\langle stub_incoming_segment, plug_in_start_point \rangle$ and $\langle stub_outgoing_segment, plug_in_end_point \rangle$. A dynamic stub has one such binding per plug-in.

Different paths may interact with each other synchronously or asynchronously (see Figure 8). *Synchronous interactions* are shown by having the end point of one path touching the start point (or a waiting place) of another path. A path touching the start point (or a waiting place) represents an *asynchronous interaction*.

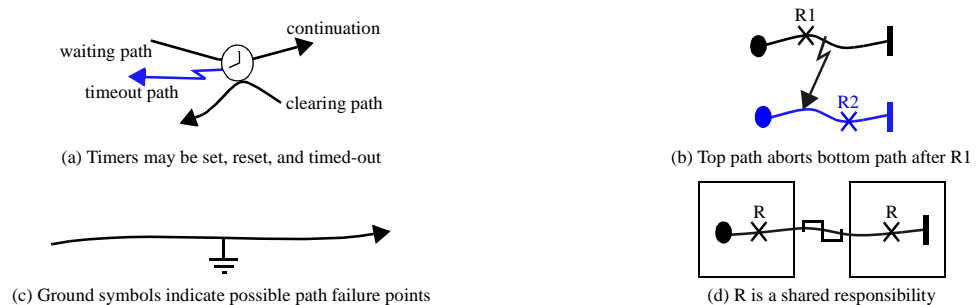
FIGURE 8. Path Interactions



Other notational elements include (Figure 9):

- **Timer:** special waiting place triggered by the timely arrival of a specific event. It can also enable a time-out path when this event does not arrive in time.
- **Abort:** a path can terminate the progression of another causal chain of responsibilities.
- **Failure point:** indicates potential failure points on a path.
- **Shared responsibility:** represents a complex activity that involves negotiation between two or more components.

FIGURE 9. Timers, Aborts, Failures, and Shared Responsibilities



Finally, the notation supports extensions specific to agent systems and to performance modeling [283][324][325]. These extensions are not addressed in this thesis, but their respective path annotations are included in the quick reference guide (Appendix A: — A11).

2.2.6 UCM Tools

There currently exists only one tool that supports the UCM notation and the XML format: the *UCM Navigator* (UCMNAV) [257]. Although still a prototype under development, this tool is already robust enough for the creation, navigation, and maintenance of UCMs. Both the path and component notations are fully supported. UCMNAV ensures the syntactical correctness of the UCMs manipulated, generates XML descriptions, exports UCMs in different formats (e.g. Encapsulated Postscript, Computer Graphics Metafile, Scalable Vector Graphics, and Maker Interchange Format), and generates reports. More recently, support for simple data model and scenario definitions was included. This enables the highlight of specific scenario paths in a collection of UCMs and the automated generation of MSCs [258].

Alternatively, any drawing package or word processors could be used to draw UCMs. However, syntactic errors may be introduced in the UCMs, and no XML code is generated.

2.3 Introduction to LOTOS

This introduction is intended to provide an overview of LOTOS to the reader not familiar with this specification language. This section presents the philosophy behind the language and the information necessary to create specifications. LOTOS operators and abstract data types are shortly reviewed, followed by an overview of LOTOS' underlying model, which will be used to define various relations between specifications. Main validation and verification techniques are then presented, followed by an enumeration of several supporting tools.

2.3.1 Philosophy of LOTOS

LOTOS, the *Language of Temporal Ordering Specification*, is an algebraic specification language standardized by ISO [56][191]. It was especially developed for the formal description of the Open Systems Interconnection (OSI) architecture (interfaces, services, and protocols) [194], although nowadays the

language is used to describe distributed and concurrent systems in general. In LOTOS, the specifier describes a system by defining the temporal relations among the actions that constitute the system's externally observable behaviour. The main influences for the behaviour part of LOTOS were Milner's *Calculus of Communicating Systems* (CCS) [260] and Hoare's *Communicating Sequential Processes* (CSP) [177]. LOTOS *behaviour expressions* are built from elementary actions by using operators such as *action prefix*, *choice*, *parallel composition*, *multiway synchronization*, *hiding*, *process instantiation*, and a few others. Data abstractions are specified with *Abstract Data Types* (ADT), based on Ehrig and Mahr's ACT ONE language [121]. LOTOS is suitable for the integration of behaviour and structure in a unique executable model. LOTOS allows the use of many tool-supported validation and verification techniques such as simulation, testing, expansion, equivalence checking, model checking, and goal-oriented execution [57].

2.3.2 Information Needed to Construct LOTOS Specifications

LOTOS can specify the temporal ordering of system actions from very little information. Assuming that actions and the necessary data types can be determined, a number of *styles* were defined according to the additional information available to the specifier [364]. The constraint-oriented style is useful when local and global behavioural constraints are representable. The resource-oriented style uses the architectural structure as a starting point. The state-oriented style focuses on system and component states, and the monolithic style is used when no information about constraints, architecture, or states is available, resulting in abstract specifications.

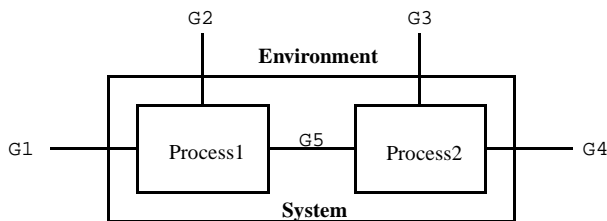
Use Case Maps can represent a good starting point for the generation of a LOTOS specification, because they capture actions and their causal ordering. LOTOS can specify abstract sequences of actions without having to say which entities can generate them. Hence it can be used to express unbound UCMs. However if it is desired to represent bound UCMs faithfully in a LOTOS specification, then information about architectural elements, interfaces, and message parameters is needed. The LOTOS resource-oriented style can support the description of bound UCMs, but inter-component causal flows in the UCMs need to be refined as messages or inter-process synchronizations, and there are usually many ways to do so. This novel aspect will be discussed further in Chapter 5.

2.3.3 LOTOS Operators

In LOTOS, systems and components are described in terms of *processes*. A LOTOS process is viewed as a black box interacting with its *environment* via its observable *gates* (Figure 10). Its internal actions are unobservable by the environment. A *behaviour expression* is built by combining LOTOS actions by means of operators and possibly other behaviour expressions.

The basic element of a behaviour expression is the *action* which represents synchronization between processes, between a process and its environment, or both. An action consists of a gate name, a list (possibly empty) of value *experiment offers* (value offers or interaction parameters), and possibly a *predicate* that imposes conditions on the event to be accepted. Actions are atomic in the sense that they occur instantaneously, without consuming time.

FIGURE 10. Representation of a System Specified in LOTOS



```

specification System [G1, G2, G3, G4] : noexit
behaviour
  hide G5 in
    Process1[G1, G2, G5]
    |[G5]|
    Process2[G3, G4, G5]
where
  process Process1[G1, G2, G5] :noexit :=
    (* ... Behaviour of Process1 *)
  endproc
  process Process2[G3, G4, G5] :noexit :=
    (* ... Behaviour of Process2 *)
  endproc
endspec

```

In Figure 10, the system is composed of two processes that interact with each other on the hidden gate G5 (interaction point). In LOTOS terms, we say that `Process1` is *synchronized* with `Process2` on G5. LOTOS synchronization is based on a multi-way rendezvous concept.

There are three basic behaviour expressions, and more complex expressions can be formed as shown in Table 3, where a is an action, B_i are behaviour expressions, g_i are gates, v_i are values, and P is a predicate.

TABLE 3. Summary of LOTOS Syntax and Semantics

	<i>Name</i>	<i>Behaviour Expression</i>	<i>Definition</i>
<i>Basic Behaviour Expressions</i>	Inaction	stop	Cannot engage in any interaction (deadlock).
	Successful Termination	exit (v_1, \dots, v_n)	Terminates successfully (and produces a δ action). Return values may optionally be specified.
	Process Instantiation	ProcName [g_1, \dots, g_n]	Creates an instance of a process ProcName.
<i>Basic Operators</i>	Action Prefix	$a; B$	Prefixes a behaviour expression B with an action a .
	Choice	$B_1 \parallel B_2$	Offers a choice between two behaviour expressions.
	Enabling	$B_1 \gg B_2$	Sequences two behaviour expressions. B_1 has to exit for B_2 to be executed. Values may be passed through the construct: $B_1 \gg$ accept parameters in B_2
	Disabling	$B_1 \triangleright B_2$	B_1 can be disrupted by B_2 during normal functioning.
<i>Composition</i>	Parallel Composition	$B_1 \parallel [g_1, \dots, g_n] \parallel B_2$	B_1 and B_2 behave independently, except for the gates g_1, \dots, g_n where B_1 and B_2 must synchronize.
	Interleaving	$B_1 \parallel \parallel B_2$	B_1 and B_2 behave independently (the synchronization set is empty).
	Full Synchronization	$B_1 \parallel \parallel B_2$	B_1 and B_2 are synchronized on all their gates.
<i>Other Operators</i>	Hiding	hide g_1, \dots, g_n in B	Hides actions g_1, \dots, g_n , which become internal and can no longer synchronize with the environment.
	Guarded Behaviour	$[P] \rightarrow B$	B can be executed if P is true.
	Local Definition	let $x:s = E$ in B	Substitutes a value expression (E) by a value identifier (x) of sort s in B .
	Process Definition	process ProcName [g_1, \dots, g_n] (parameters) : funct := B endproc	Creates a process definition with formal gates and parameters. The functionality funct indicates whether the process can terminate successfully (exit , optionally with values) or not (noexit). Can be instantiated as a basic behaviour expression.
	Comment	(* <i>This is a comment</i> *)	Comment skipped by the parsers.

More operators exist (e.g. generalized choice (different from \parallel) and `par`) but they can often be avoided and they are not used in this thesis.

2.3.4 LOTOS Abstract Data Types

LOTOS models data by an abstract equational notation. There are no predefined data types, but there is a standardized library of commonly required data types. In LOTOS, every data type is a set of data values and operations that require to be defined.

LOTOS *sorts* are distinct sets of data values. The concept of sort in LOTOS corresponds to the concept of type in many programming languages. LOTOS *operations* correspond to functions and procedures to manipulate objects. By means of operations it is possible to combine values of the same or different sorts into aggregate values (e.g. a record), or establish relations between them. LOTOS *equations* state properties that must be satisfied by (any implementation of) the objects of the type. They are often interpreted as rewrite rules by tools. LOTOS *types* package sorts, operations, and equations together. Types can be aggregated, inherited, renamed, defined formally and instantiated (actualized). As an illustrative example, the type `Boolean` can be defined as:

```

type Boolean is
  sorts Bool
  opns
    true, false :          --> Bool (* Constructors *)
    not :              Bool --> Bool
  eqns
    ofsort Bool
      not (true) = false;
      not (false) = true;
endtype (* Boolean *)

```

In LOTOS, data can be associated with actions in two ways: `!value`, which means *value offer*, and `?variable:type`, meaning *value query*. These can be combined in actions. For example,

```
G5 !3 ?answer:Bool
```

denotes an action where on gate `G5`, the value `3` is offered, and a value for `answer` (of type `Bool`) is queried simultaneously. Offers and queries are both *experiments*. Selection predicates can be optionally added to value queries, as in:

```
G5 ?n:Integer [n > 3]
```

meaning that the acceptable values for the integer `n` are greater than `3`. These examples demonstrate the abstract nature of the language, since it allows to express in a single action system events that could be quite complex to implement.

2.3.5 Labelled Transitions Systems and Underlying Semantics

The underlying model of LOTOS is based on the concept of *labelled transition systems* (LTSs). An LTS is a generalization of a finite state machine that provides a convenient way for expressing the step-by-step operational semantics of behaviour expressions. The latter evolve by executing one action at a time, selected from their alphabet set. The following notations and definitions for LTSs are excerpted from [69], [150], and [242].

Definition 2.1: A **labelled transition system** is a 4-tuple $LTS = \langle S, s_0, L, T \rangle$, where:

- S is a (finite) non-empty set of states;
- $s_0 \in S$ is the initial state;
- L is a (finite) set of observable actions; and
- $T = \{ \text{---}a\text{---} \subseteq S \times S \mid a \in L', \text{ where } L' = L \cup \{i\} \}$, is the set of *transitions*, which are binary relations on S . If $s_1 \text{---}a\text{---} s_2$ such that $s_1, s_2 \in S$ then $\langle s_1, s_2 \rangle \in \text{---}a\text{---}$. i represents a hidden *internal action*.

Note that $\text{---}a\text{---}$ can be interpreted both as a set (over $S \times S$) and as a relationship between two states. The notation and definitions in Table 4 are widely used for interpreting LTSs and for defining different conformance and equivalence relations, some of which will be introduced in the next section and then used in Chapter 6:

TABLE 4. LTS Notation and Definitions

<i>LTS Notation</i>	<i>Definitions</i>
$L = \{a, a_1, a_2, \dots, a_m\}$	The alphabet of observable actions. We define i to be the internal action (often named τ in the literature [153]), and δ to be the successful termination action.
$B \xrightarrow{a} B'$	After executing the observable action a , the behaviour expression B is transformed into another behaviour expression B' .
$B \xrightarrow{i^k} B'$	After executing a sequence of k hidden actions, the behaviour expression B is transformed into another behaviour expression B' .
$B \xrightarrow{a_1 a_2} B'$	$\exists B''$ such that $B \xrightarrow{a_1} B'' \wedge B'' \xrightarrow{a_2} B'$.
$B =_a \Rightarrow B'$	B is transformed into another behaviour expression B' by executing zero or more internal actions, followed by the observable action a , then zero or more internal actions. Formally, $\exists k_0, k_1 \in N / B \xrightarrow{i^{k_0} a i^{k_1}} B'$.
$B =_a \Rightarrow$	B may accept the action a . Formally, $\exists B' / B =_a \Rightarrow B'$.
$B \neq_a \Rightarrow$	$\neg(B =_a \Rightarrow)$, that is, B must refuse the action a .
$B =_\sigma \Rightarrow B'$	B is transformed into another behaviour expression B' by executing a sequence of observable actions. Formally, if $\sigma = a_1, a_2, \dots, a_n$ then $\exists k_0, k_1, \dots, k_n \in N / B \xrightarrow{i^{k_0} a_1 i^{k_1} a_2 \dots a_n i^{k_n}} B'$.
$B =_\sigma \Rightarrow$	$\exists B' / B =_\sigma \Rightarrow B'$.
B after σ	The set $\{B' / B =_\sigma \Rightarrow B'\}$, i.e. the set of all behaviour expressions reachable from B after executing the sequence σ .
$Tr(B)$	The trace set of B , defined as $\{\sigma / B =_\sigma \Rightarrow\}$. Note that $Tr(B) \subseteq L^*$.

The operational semantics of LOTOS is expressed in terms of *inference rules* acting on an underlying LTS. For instance, the simplified inference rules for the choice operator are:

Notation: $\frac{\text{if ...}}{\text{then ...}}$	$\frac{B_1 \xrightarrow{a_1} B_1'}{B_1 [] B_2 \xrightarrow{a_1} B_1'}$	$\frac{B_2 \xrightarrow{a_2} B_2'}{B_1 [] B_2 \xrightarrow{a_2} B_2'}$
---	--	--

The first inference rule means that if the behaviour expression B_1 can perform a_1 and then behave like B_1' , then $B_1 [] B_2$ can add a transition a_1 to the LTS and then behave like B_1' (the B_2 alternative is dropped). The rule on the right is symmetrical. The LOTOS standard provides inference rules for all the operators in the language [191].

LTSs can also be represented visually as graphs. Often, LOTOS behaviour expressions are shown as *behaviour trees*, which represent unfolded LTSs (i.e. where loops are expanded). For example, a behaviour expression for a simple telephone system is given on the left side of Figure 11, with the corresponding behaviour tree (LTS) on the right side.

FIGURE 11. A Behaviour Expression and its LTS as a Behaviour Tree



Labelled transition systems will help illustrating behaviour expressions and reasoning about them throughout the thesis, particularly when the construction and validation of LOTOS specifications from Use Case Maps will be discussed.

2.3.6 Equivalences and Other Relations

Multiple equivalence, ordering, and other relations have been defined for LOTOS. They are usually defined in terms of the underlying semantic model (LTS) rather than with the LOTOS syntax itself. These relations are of the utmost importance because they are at the heart of many validation, verification, simplification, and implementation techniques for LOTOS. In this section, we will distinguish between *equivalence* relations (which are symmetric, reflective, and transitive) and other relations (which are not symmetric and may or may not be transitive).

Equivalence Relations

The LOTOS theory distinguishes between several types of equivalence relations, many of which are inspired from CCS [260]. Depending on the level of details considered (observable or hidden actions,

branching structure, non-determinism, etc.), two specifications (i.e. two LTSs) may or may not be equivalent. Table 5 contains some of the most interesting equivalence relations, from the strongest, which distinguishes the most, to the weakest, which distinguishes the least. Assume that two behaviour expressions $S1$ and $S2$ are compared:

TABLE 5. LOTOS Equivalence Relations

Relations	Definitions
Equality: $S1 = S2$	$S1$ and $S2$ are equal ($S1 = S2$) iff their respective LTSs are isomorphic (i.e. the LTSs are the same).
Strong bisimulation: $S1 \sim S2$	Each immediate successor (next action, visible or not) of $S1$ must be equivalent to some immediate successor of $S2$, and conversely. Formally: If $S1 \sim S2$ then, for all $a \in L \cup \{i, \delta\}$ (i) whenever $S1 \xrightarrow{a} S1'$ then $\exists S2' / S2 \xrightarrow{a} S2'$ and $S1' \sim S2'$ (ii) whenever $S2 \xrightarrow{a} S2'$ then $\exists S1' / S1 \xrightarrow{a} S1'$ and $S1' \sim S2'$
Congruence: $S1 \approx_c S2$	A context $C[\bullet]$ is a behaviour expression with a formal process parameter $[\bullet]$ called a hole. If $C[\bullet]$ is a context and B is a behaviour expression, then $C[B]$ is the behaviour expression that is the result of replacing the \bullet occurrences by B . $S1$ and $S2$ are congruent ($S1 \approx_c S2$) iff, for all context $C[\bullet]$, $S1 \approx_c S2$ implies $C[S1] \approx_c C[S2]$. Congruent behaviour expressions can be interchanged (like substitutable components) in any context and lead to global specifications that are observationally equivalent.
Weak bisimulation: $S1 \approx S2$	Whereas strong bisimulation considers i like an observable action, weak bisimulation abstracts from internal actions, except when they cause non-determinism in alternatives. We say that $S1$ and $S2$ are weak bisimulation equivalent ($S1 \approx S2$) iff for all sequences $\sigma \in L^*$, each σ -descendant of $S1$ is equivalent to some σ -descendant of $S2$, and conversely. Formally: If $S1 \approx S2$ then, for all $a \in L \cup \{\delta\}$ (i) whenever $S1 \xRightarrow{a} S1'$ then $\exists S2' / S2 \xRightarrow{a} S2'$ and $S1' \approx S2'$ (ii) whenever $S2 \xRightarrow{a} S2'$ then $\exists S1' / S1 \xRightarrow{a} S1'$ and $S1' \approx S2'$ This relation is also called <i>observational</i> equivalence.
Testing equivalence: $S1 \underline{te} S2$	$S1$ and $S2$ are testing equivalent ($S1 \underline{te} S2$) iff they cannot be distinguished by any test case. A test case is a behaviour expression which, when composed with a specification, leads to one of three possible verdicts (must pass, may pass, reject). A more formal definition of \underline{te} will be given later.
Trace equivalence: $S1 \underline{tr} S2$	$S1$ and $S2$ are trace equivalent ($S1 \underline{tr} S2$) iff they can produce the same traces of observable actions. Formally, $S1 \underline{tr} S2$ iff $Tr(S1) = Tr(S2)$.

Suppose that **SPECS** is the set of all possible behaviour expressions (or specifications). The relations in Table 5 are all defined over $\mathbf{SPECS} \times \mathbf{SPECS}$, and hence can be compared. A strict ordering exist among these relations: $= \subset \sim \subset \approx_c \subset \approx \subset \underline{\text{te}} \subset \underline{\text{tr}}$. For example, if $S1 \approx_c S2$ holds, then we can conclude that $S1 \approx S2$, $S1 \underline{\text{te}} S2$, and $S1 \underline{\text{tr}} S2$, but $S1 \sim S2$ may not hold (we do not know with certainty).

The strong bisimulation is the strongest meaningful equivalence relation that does not require isomorphism at the LTS level. Many algebraic laws have been defined for strong equivalence [191], and they remain valid for all the weaker equivalences. However, this equivalence relation is somewhat deficient as it treats the internal action i on the same basis as all other actions. Weak bisimulation solves this problem by abstracting from internal actions, while preserving meaningful branching structures in the LTS. Testing equivalence comes from a more pragmatic point of view where equivalence can only be assessed by means of testing. The congruence relation has useful implications for design as it is the largest relation that allows the substitution of a behaviour expression by a congruent one in any LOTOS context. Such behaviour expressions act like pluggable components. Trace equivalence, although it is easy to understand, is not very useful as all the branching structure of the LTS (which includes non-determinism, what can be accepted, and what can be refused) is lost.

Other Relations

Equivalence relations are not the only way to compare behaviour expressions. Sometimes, relations that are not symmetric may be more appropriate to evaluate validity when different levels of abstractions are involved. For instance, a preorder, which is a reflexive and transitive relation, may be used to check the conformance of a protocol specification against a service specification. If $\underline{\mathbf{R}}$ is a preorder, then $\underline{\mathbf{R}} \cap \underline{\mathbf{R}}^{-1}$ becomes symmetric and hence also becomes an equivalence.

Over the years, many non-symmetric relations have been defined by Brinksma [69], Leduc [242][243], and others. Table 6 recalls three basic relations (defined over $\mathbf{SPECS} \times \mathbf{SPECS}$), which will be linked later to the concepts of canonical tester and testing equivalence.

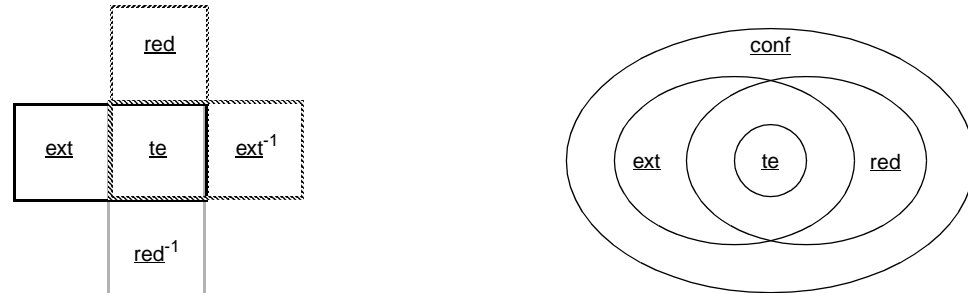
TABLE 6. Other Relations for LOTOS

<i>Relations</i>	<i>Definitions</i>
Conformance: $S1 \underline{\text{conf}} S2$	$S1$ conforms to $S2$ ($S1 \underline{\text{conf}} S2$) expresses that $S2$ deadlocks less often than $S1$ itself when tested against the traces of $S1$. $S2$ may however contain behaviour not present in $S1$. Formally: $S1 \underline{\text{conf}} S2$ iff $\forall \sigma \in Tr(S2), \forall A \subseteq L,$ if $\exists S1' / \forall a \in A, S1 = \sigma \Rightarrow S1' \wedge S1' \neq a \Rightarrow$ then $\exists S2' / \forall a \in A, S2 = \sigma \Rightarrow S2' \wedge S2' \neq a \Rightarrow$
Reduction: $S1 \underline{\text{red}} S2$	The reduction relation states that $S1 \underline{\text{red}} S2$ if $S1$ can only execute actions that $S2$ can execute, and $S1$ can only refuse actions that can be refused by $S2$. In other words, <u>red</u> takes away unnecessary options. $S1$ has fewer traces than $S2$ yet $S1$ deadlocks less often in an environment limited to the traces of $S1$. Formally: $S1 \underline{\text{red}} S2$ iff $S1 \underline{\text{conf}} S2 \wedge Tr(S1) \subseteq Tr(S2)$.
Extension: $S1 \underline{\text{ext}} S2$	We say that $S1$ extends $S2$ ($S1 \underline{\text{ext}} S2$) when $S1$ has more traces than $S2$, but in an environment whose traces are limited to those of $S2$, $S1$ deadlocks less often. Formally: $S1 \underline{\text{ext}} S2$ iff $S1 \underline{\text{conf}} S2 \wedge Tr(S2) \subseteq Tr(S1)$.

Brinksma demonstrated that red and ext are both preorders, whereas conf is not because this relation is not transitive (i.e. $S1 \underline{\text{conf}} S2 \wedge S2 \underline{\text{conf}} S3$ does not imply $S1 \underline{\text{conf}} S3$) [69]. Leduc suggested another conformance relation (conf-eq) which is a preorder [242], but its treatment is outside the scope of the thesis.

Testing Equivalence and Canonical Testers

According to the LOTOS testing theory presented by Brinksma [69], test cases are behaviour expressions that are composed with the specification. Two specifications are testing equivalent ($S1 \underline{\text{te}} S2$) if they cannot be distinguished by any test case. An interesting property of te is that it can be expressed formally in terms of red or ext: $S1 \underline{\text{te}} S2 \Leftrightarrow S1 \underline{\text{red}} S2 \wedge S2 \underline{\text{red}} S1 \Leftrightarrow S1 \underline{\text{ext}} S2 \wedge S2 \underline{\text{ext}} S1$. This is illustrated in the left half of Figure 12. The testing equivalence is both a reduction and an extension and, according to their definitions, reduction and extension are both conformance relations (see the right part of Figure 12).

FIGURE 12. Connecting Several LOTOS Relations

Still, equivalences are not always the best way to establish the validity of implementations. For instance, the conformance relation is often used as a criterion to test an implementation (System Under Test — *SUT*) against its specification (*S*). When $SUT \text{ conf } S$, *SUT* is allowed to be more deterministic and to contain more alternative behaviour than *S*.

Every specification *S* has a *canonical tester* ($CT(S)$), which is a behaviour expression (with the same traces as *S*) that tests *S* completely according to conf [69]. Many such testers exist for a given specification, and they are all testing equivalent with each other. $CT(S)$ represents the only test case necessary to check that a *SUT* conforms to *S*. An interesting property is that $CT(CT(S)) \text{ te } S$.

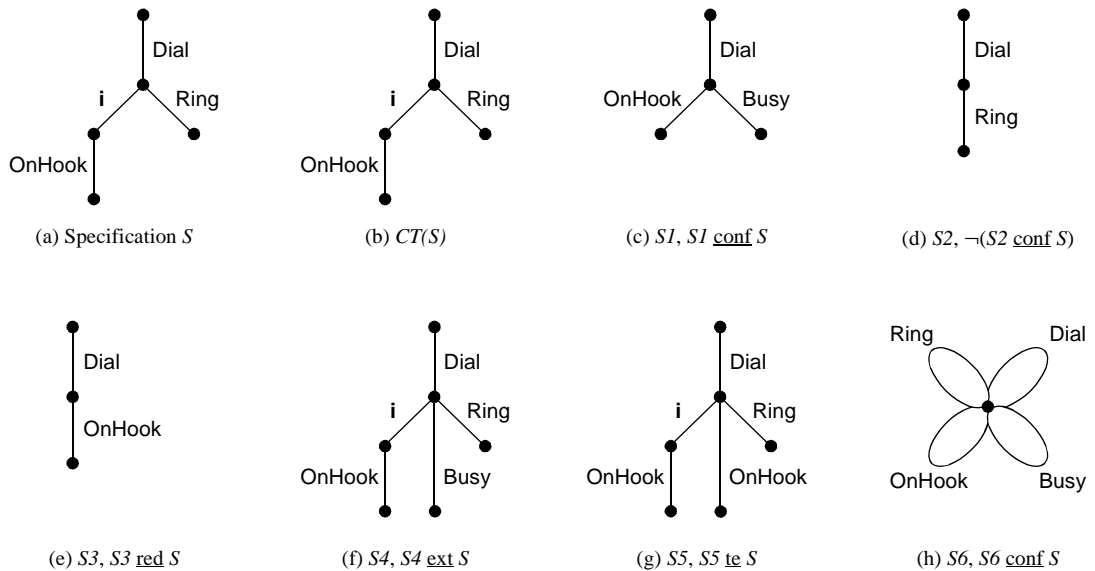
For most realistic specifications, a canonical tester cannot be directly generated as it may be infinite, especially when data values or recursive processes are involved. If such a tester can be generated, then it can be applied to test another (e.g. refined) LOTOS specification. However, since these testers are usually non-deterministic, the use of canonical testers on real implementations does not guarantee that an error will be highlighted (by an unexpected deadlock). Consequently, $CT(S)$ should not be used for testing conformance of real implementations directly, but only used to guide the generation of an adequate test suite (with deterministic test cases) from it [69]. Tretmans suggests an algorithm, based on this idea, to generate test cases from a LOTOS specification [345]. Leduc presented simplified canonical testers for the conf-eq relation [242] and others that handle divergent behaviour (LTS with loops of internal actions) [243], but these developments will not be discussed any further in the thesis.

Behaviour expressions that are reductions of $CT(S)$ are sound test cases for any implementation of S . We say that $S2$ is *irreducible* if $S1 \underline{\text{red}} S2 \Rightarrow S1 \underline{\text{te}} S2$. Suppose two test cases (T_x and T_y) where $T_x \underline{\text{red}} CT(S)$ and $T_y \underline{\text{red}} CT(S)$. These two test cases have the same detectability power if they are testing equivalent, i.e. when $T_x \underline{\text{te}} T_y$. If T_x is irreducible with the same detectability power as T_y , then T_x is a better test case than T_y because T_x is simpler and yet as powerful as T_y . Irreducible test cases usually make excellent tests.

Example

Figure 13 illustrates some of the relations covered so far. Assume an alphabet of observable actions $L = \{\text{Busy, Dial, OnHook, Ring}\}$.

FIGURE 13. Illustration of Several Relations



- (a) The specification S . OnHook is considered mandatory whereas Ring is optional.
- (b) The canonical tester of S . It happens here that S is a self-tester ($CT(S) = S$).
- (c) $S1$ conforms to S , even if Ring is absent and Busy is added.
- (d) $S2$ does not conform to S because OnHook is not available after Dial.

- (e) $S3$ reduces S . Note also that $S3$ red $CT(S)$ and that $S3$ cannot be reduced further. Therefore $S3$ is an irreducible test case for S .
- (f) $S4$ extends S by adding an observable action that was not part of S .
- (g) $S5$ is testing equivalent to S .

Note that testing $S1$, $S3$, $S4$, $S5$, and $S6$ against $CT(S)$ does not result in any premature deadlock. This is the case for all specifications conforming to S . Testing $S2$ against $CT(S)$ results in a deadlock when the $\langle \text{Dial}, \mathbf{i}, \text{OffHook} \rangle$ branch is chosen in the canonical tester, hence $S2$ does not conform to S .

One weakness of the conf relation is that it is always possible to build a trivial implementation that conforms to the specification. For instance, $S6$ in Figure 13(h) accepts every action of the alphabet L and hence can never deadlock. This is where rejection test cases (to be seen in Section 3.4.3) can help establishing the validity of such implementations.

2.3.7 Validation and Verification in LOTOS

The algebraic nature of LOTOS enables a multitude of techniques to become applicable for the validation and verification of specifications [57]. Some of the most popular techniques are:

- **Step-by-step execution** (or interactive simulation), in which the specifier takes the role of the environment by providing events to the specification and by observing the results (i.e. the next possible events) [135][162]. Although useful for debugging, step-by-step execution is probably the simplest and weakest validation technique available for LOTOS.
- **Equivalence checking**, used to check the conformity or the equivalence of one specification against another (usually after some refinement or modifications) [153][242]. In LOTOS, equivalence checking is usually done using the underlying LTS model, but it can sometimes be done algebraically through the use of equivalence rules.
- **Model checking** aims to check a specification against safety, liveness, or responsiveness properties (often derived from the requirements) [149][150]. These properties can be

expressed, among other languages, in terms of temporal logic or μ -calculus formulas. In the LOTOS world, this technique usually requires that the specification be expanded into a corresponding model, which is some graph representation (labeled transition system, finite state machine, or Kripke structure) of the specification's semantics. On-the-fly model checking techniques, where the whole model does not have to be generated *a priori*, exist as well [135].

- **Testing** is concerned with the existence (or the absence) of traces, trees, use cases, or more generally scenarios in the specification. These scenarios reflect system functionalities and are transformed into black-box test cases that can be composed with the specification [69][70][243][279][345]. Test cases are usually less powerful and expressive than properties expressed in logic. However, test cases are often more manageable and understandable than properties and they relate more closely to (informal) operational requirements and semantics.

Other validation and verification techniques such as random walks [135], goal-oriented execution [165], symbolic execution [40], symbolic equivalence and model checking [327], and observers [115][142] can also be applied to LOTOS specifications, but they are not as commonly used. This thesis focuses mostly on testing as a validation technique. Note that all these techniques are supported (to various degrees) by tools, some of which are presented in the next section.

2.3.8 LOTOS Tools

LOTOS being a well-established standard, a number of solid tools have been developed for it around the world. We shall mention three of the most popular ones:

- **CADP** (CÆSAR-ALDEBARAN Distribution Platform): developed at INRIA in Grenoble, France [135]. It was built for extensive state exploration of LOTOS specifications. This tool provides a variety of searching strategies to detect conditions of interest in the execution of a protocol or a feature, including step-by-step execution, random walks, equivalence checking, and (on-the-fly) model checking.

- **ELUDO** (Environnement LOTOS de l'Université D'Ottawa): developed at the University of Ottawa, Canada. It is mostly useful at the initial stages of the development of the LOTOS specification, since it has an effective step-by-step execution option, graphically attractive and user-friendly. It also supports symbolic expansion, model checking, and goal-oriented execution. ELUDO, together with CADP and other tools, is now part of the EUCALYPTUS Toolbox [145].
- **LOLA** (LOtos LABoratory): developed at the Universidad Politécnica de Madrid, Spain [279][280][301]. It is a state exploration tool for the simulation and testing of LOTOS specifications. Test cases are specified as LOTOS processes, and they can be composed with the specification to detect possible errors. LOLA analyses the test terminations for all possible evolutions. If the number of test runs is too large or even infinite, LOLA can use equivalence relations and coverage heuristics to check a representative subset of the possible evolutions. Verdicts such as Must pass, May pass and Reject are provided by the tool for each test case.

In this thesis, we mainly use LOLA because of its ability to test LOTOS specifications.

2.3.9 Enhancements to LOTOS

The International Organization for Standardization has recently the LOTOS language to produce *Enhanced LOTOS* (or E-LOTOS) [198]. This new language is backward compatible with LOTOS. E-LOTOS includes new operators and semantics for handling time and modules, and a new functional language (à la ML) replaces ADTs for representing data types.

E-LOTOS is not being used in this thesis because execution and validation tools are not currently available.

2.4 Chapter Summary

This chapter reviews general definitions that will be used throughout the thesis. Section 2.1.1 shows how SPEC-VALUE is related to the engineering of requirements, systems, protocols, and software,

while the rest of Section 2.1 establishes basic terminology for processes, formal methods, specifications, designs, validation, verification, prototypes, and many other related concepts.

Sections 2.2 and 2.3 provide tutorial material for readers who want to familiarize themselves with UCMs and LOTOS. They both cover the philosophy behind each notation, the information needed to use them, elements of the notation (paths and components for UCMs; operators, ADTs, LTSs and relations for LOTOS), and tool support.

Contributions

The following items are original contributions of this chapter:

- Quick tutorial on the Use Case Maps notation.
- Quick tutorial on the formal description technique LOTOS.

CHAPTER 3

Literature Survey

“Post hoc, ergo propter hoc”

Unknown source

“After this, therefore because of this”. This logical fallacy is committed whenever someone implies that an event that occurred before another event must have caused this event.

This chapter surveys existing work and building blocks in four major areas closely related to SPEC-VALUE. We first recall several models that support *causality* (the focus of our scenarios), as opposed to a plain temporal ordering (Section 3.1). Then, Section 3.2 compares Use Case Maps and LOTOS with several other notations and *specification techniques*. Because SPEC-VALUE uses *scenarios* as building blocks for the specification and validation of telecommunications systems, we devote Section 3.3 to the introduction of several scenario notations and related approaches, including some techniques for the *construction* of communicating and distributed entities from scenarios. We also include a fourth section on the *verification and validation* of distributed systems, with a special emphasis on the techniques relevant to the notations and concepts used in the thesis (Section 3.4). A summary follows in Section 3.5.

3.1 Causality

Causality is a relation that connects causes to their effects. In concurrency theory, establishing causality is useful as this helps distinguish events that are caused by other events from events that are simply observed one after another without one affecting the other. This is in fact the main concern of the incorrect inference cited above: *post hoc, ergo propter hoc* (after this, therefore because of this). Cau-

sality is also one important criterion used for evaluating scenario notations in Section 3.3.2 (where it is called *ordering*). Since the representation of causality is a feature that distinguishes UCMs from many other scenario notations, this section discusses different semantics and representations for causality. In particular, Section 3.1.1 introduces important benefits of causality while Section 3.1.2 summarizes some of the major causal models available. Causality is also briefly discussed in the context of UCMs and LOTOS in Section 3.1.3 and Section 3.1.4 respectively.

3.1.1 Why causality?

We see four main reasons why it can be beneficial to capture causality in models:

To Capture Intentions

Causality helps expressing intentions at an abstract level as well as focusing on them. Leyton observes that the mind assigns to any shape a causal history explaining how the shape was formed [245]. Such causal history can become a valuable and long-lived artefact in a development process. UCMs capture existing or desired functionalities (shapes) visually and help arguing about causality at a level close to requirements and high-level designs.

To Distinguish the Type of Ordering

Causal ordering and temporal ordering are almost indistinguishable for sequential processes. However, when concurrency is involved (e.g. in communicating and distributed systems), interpreting a temporal ordering as a causal relationship can be misleading. For instance, in a simplified telephone system, the following scenario could be observed: `<OffHook, Dial, Ring, RingBack>`. We might conclude that `RingBack` is caused by its prefix, i.e. `<OffHook, Dial, Ring>`. However, this interpretation might be wrong. `Ring` and `RingBack` may both be caused independently by `<OffHook, Dial>`, and there might not necessarily be any causal dependency between them. An early focus on causal relations between actions helps to avoid several misunderstandings related to temporal ordering. The causal dependence between events should be documented in the early stages of the design process, before this information gets lost among the details of linear sequences or in the behaviour of individual components. UCMs are very helpful in this context.

To Generate Smaller Models

Many specification languages, including LOTOS, have underlying semantics based on models that do not support causality. For instance, labelled transition systems describe temporal ordering exclusively. LTSs interpret concurrency through the *interleaving* of parallel actions, which often leads to very large models. Using causality at the description level (e.g. with UCM scenarios) leads to smaller descriptions and enable the preservation of causality in the underlying model (when causality is supported). This could in turn help cope with combinatorial explosions of system states.

To Allow the Refinement of Actions

Refinement of actions is an important research topic in concurrency theory. The general problem is defined as follows: actions at a given level of abstraction are replaced by more complicated processes on a lower level of abstraction. The behaviour of the refined system is intended to be inferred compositionally from the behaviour of the original system and from the behaviour of the processes substituted for actions. Action refinement promotes the design of systems in a modular and hierarchical way. Many authors, including van Glabbeek and Goltz [153], have shown that interleaving models of concurrent systems (e.g. LTSs) are not suited for defining action refinement in its general form, and that causal models are more appropriate. In SPEC-VALUE, such refinement should hence be done at the UCM level, not at the underlying LTS level.

3.1.2 Concurrency Models and Equivalence Relations

Specification techniques such as Petri Nets, SDL, LOTOS and UCMs can all describe concurrency in various forms. However, the semantic models associated to these languages may or may not support causality very well. The models of concurrency found in the literature usually fall into one of these two categories:

- ***Interleaving semantics***: the independent progression of two processes is modelled by specifying the possible interleaving of their (atomic) actions.
- ***Causal semantics***: the causal relations between the actions of a system are represented explicitly. This is often referred to as *true concurrency* or *partial order* semantics.

Equivalence relations, which establish whether a model is equivalent to another model according to some criteria, can be defined over both types of semantics. Such relations are often used to establish the correctness of refinements and implementations with respect to specifications of concurrent systems.

In a recent survey on action refinement [153], van Glabbeek and Goltz indicate that the preserved level of detail in system runs (interleaving versus causal) is not the only aspect to be considered when describing concurrency models and relations. Two other important aspects include:

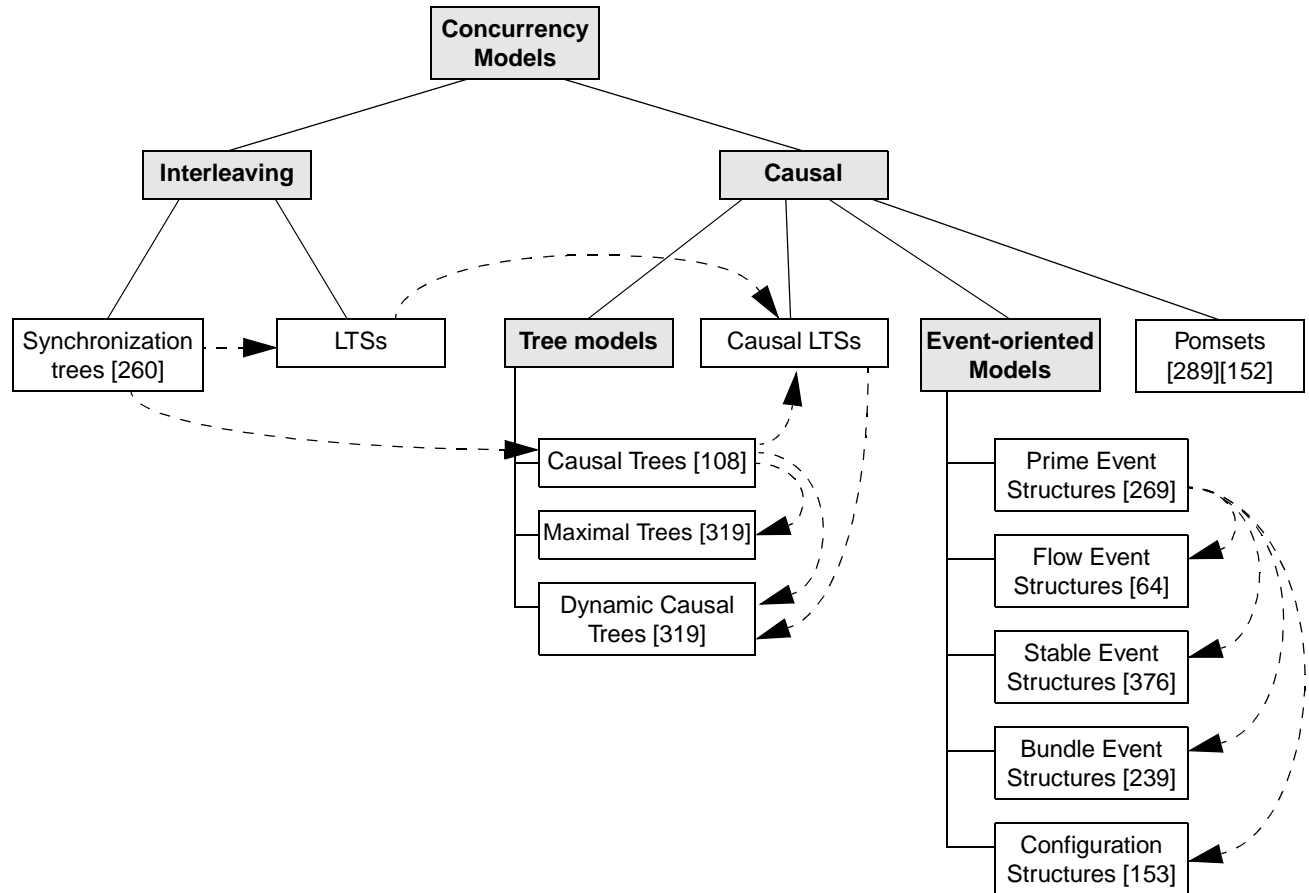
- Preserved level of detail of the choice structure between system runs: this goes from *trace semantics* (linear time), where the choice structure is completely neglected, to *decorated trace semantics*, where part of the choice structure is taken into account, and finally to *bisimulation semantics* (branching time), which preserves the information where two different runs diverge.
- Treatment of internal or invisible actions: this goes from relations where internal actions are treated like *visible* actions (strong bisimulation) to relations where internal actions are *invisible* and can be, to a certain extent, abstracted (weak bisimulation).

Families of Concurrency Models

Several important families of concurrency models are classified in Figure 14. The leaves of this tree (white boxes) are fine-grained families for which references to existing work are provided. Grey boxes indicate related sets of families. Dotted arrows indicate which family (tail) influenced the creation of another, more general and more expressive family (head).

Several people argue that Petri Nets can be used as a causal semantic model. However, Petri Nets usually rely on translations to interleaving models, e.g. LTSs, for the verification of properties such as bisimulations [319].

FIGURE 14. Families of Concurrency Models



3.1.3 Causality and Use Case Maps

Using the four factors discussed in Section 3.1.1, we observe that UCMs describe *causal* relationships between responsibilities, which may be allocated to components. In order to do so, UCMs offer a variety of constructs such as sequence, AND-fork, AND-join, and (a)synchronous interactions between UCM paths.

UCMs support alternatives (OR-fork), which are closer to the *branching structure* in LTSs and trees than to linear-time models such as *pomsets* (partial order multisets). However, UCMs do not support conflict/exclusion relations (e.g. the non-occurrence of action a is condition for the occur-

rence of action b) as most event-oriented models would. Hence, UCMs have more affinities with trees and LTSs than with event structures.

Action refinement could also be useful to UCMs. A stub is essentially a coarse-grained action refined by a plug-in, and some properties might be interesting to preserve under such refinement. Causal models support action refinement much better than plain interleaving models would. However, formal action refinement will not be pursued in the thesis and will therefore remain a research topic.

3.1.4 Causality and LOTOS

Many semantic models can be used underneath a given specification language, which often focuses on the syntactic level. For instance, standard LOTOS is based on an interleaving semantic model (LTSs), which takes into consideration internal actions and offers a wide range of equivalence relations, including bisimulation. However, LTSs are weak at representing causality, even if the syntax of LOTOS can express this concurrency concept (e.g. with the interleaving operator).

Various substitute semantics for LOTOS have been suggested in the literature: bundle event structures by Langerak [239] and Brinksma *et al.* [71], event-oriented models expressed in terms of a causal algebra by Pires [286] and Quartel [300], causal LTSs by Coelho da Costa [99][100], and maximal trees and dynamic causal trees by Saïdouni [319] (DCTs). Among all these semantics, DCTs are probably the most interesting as they are more general than causal trees (see Figure 14), and they are closer to the current LOTOS semantic model than those based on event-oriented models. DCTs also correspond to unfolded causal LTSs. DCTs cannot represent infinite behaviour as easily as causal LTSs, nor do they solve the state explosion problem. However, in a verification context, DCTs do not suffer from undecidability problems whereas causal LTSs do.

When choosing a causality model, a trade-off is usually required between expressiveness and ease of verification, which are two opposite forces. Causality leads to equivalence relations that are more complex and harder to define than relations based on an interleaving semantics, especially in a context where internal actions are considered [153]. Correct and efficient verification algorithms are hard to define for expressive causal semantics, and currently tool support is very weak.

3.1.5 Summary and Discussion

This section discusses several results from the concurrency theory in general, with a special emphasis on causality. It explains why causality is an interesting property of semantic models. Causality allows for a better understanding of intentions and ordering, may result in smaller models, and enables the refinement of actions. Important families of concurrency models based on interleaving and causal semantics were briefly discussed and classified (Figure 14).

Both Use Case Maps and LOTOS can express causality at a syntactic level, which is beneficial to SPEC-VALUE. However, causality is missing from the standard LOTOS semantic model (LTSs). Although many alternative semantics exist, SPEC-VALUE, with its emphasis on validation, is still using LTSs. Causal semantic models are difficult to validate and verify formally, equivalence relations are complex, and tool support for causal V&V is still sparse and experimental. LTSs are the standard semantics for LOTOS, which is well supported by many validation and verification tools. Hence, LTSs still represent the most pragmatic avenue for validation in SPEC-VALUE., even at the cost of losing causal relationships and generating larger models.

3.2 Specification Techniques

Nowadays, specification techniques are widely applied to many software-related engineering fields. Formal methods have particularly raised much hopes over the last two decades for the specification of requirements and designs, and for their validation and verification [104][176][220]. Formal methods are mathematical specification languages with formal syntax and semantics, which offer rigorous support of system development [351]. Despite several successful applications to real systems, formal methods have been the target of myriads of criticisms over the years because they have not really met the initial optimistic expectations of their users [20][250]. In order to answer these critics and to explain the real strengths and weaknesses of formal methods, several myths have been described and explained to the software community by Hall [166] and Bowen [67]. Le Charlier and Flener have replied with additional myths on the usefulness of formal methods, the main one being that specifications are necessarily informal [241]. Against all odds, formal methods have shown notable resiliency. They are still in use nowadays and are the subject of research and development. New tools have been

developed for them and applied in areas at times quite remote from the ones envisaged by their initial designers [20].

It is recognized that formal methods and specification techniques in general are not universal. For instance, some methods handle concurrency better than others, whereas the latter might be more appropriate for the description of sequential algorithms. Jackson even claims that universal methods cannot be effective because they cannot take advantage of any particular features of the problem at hand, and they must abstract from any feature whose universal treatment is simply too hard [211]. Most formal methods therefore focus on particular problem domains (they were named *domain-specific formal methods* in [250]), one of which being telecommunications. But even for one such domain, multiple techniques are often used together to specify various facets of the problem. Several advocates of multi-method solutions include Holzmann [181], Zave and Jackson [382], as well as standardization bodies such as the ITU-T (with SDL/MS/ASN.1/TTCN) and the OMG (with UML). SPEC-VALUE is not different in that respect as it is based on two specification techniques. However, in order to justify that these techniques represent a potentially useful combination, we need to evaluate them against a number of criteria and compare them to other techniques.

This section presents and comments on six formal and semi-formal specification techniques used for telecommunications systems. Already, two of them have been introduced in Section 2.2 (Use Case Maps) and Section 2.3 (LOTOS). Section 3.2.2 gives an overview of all these techniques, and then a comparison is provided (Section 3.2.3) according to criteria presented in Section 3.2.1.

3.2.1 Evaluation Criteria for Specification Techniques

In a recent survey [20], we have selected a wide range of evaluation criteria on the basis of our own experiences with specification techniques and requirements engineering, and of existing surveys from Ardis *et al.* [39], Craigen *et al.* [104], and Weidenhaupt *et al.* [368]. In order to evaluate and compare specification techniques for telecommunications systems, we selected a total of thirteen criteria grouped in four categories, namely usability, validation and verification, tool support, and training. We see them as being all fundamental, and therefore we do not try to prioritize them at this point.

Usability

- **Readability:** specifications need to be readable by domain experts (and not only by experts in the specification technique). There is a strong emphasis here in human understanding, and in common understanding amongst different stakeholders, including the client.
- **Modularity:** composition operators are needed to allow large specifications to be easily written and understood by decomposing them into smaller parts.
- **Abstraction:** this criterion is concerned with the level of detail that needs to be addressed, and with separation of concerns. An abstraction mechanism that supports two-way traceability allows to go from complex and high-level viewpoints to simple and low-level viewpoints and vice versa.
- **Scalability:** we say that a technique is scalable if it allows the specification of complex and simple systems in a similar way.
- **Maintenance and Evolution:** we are interested in techniques that allow for the reuse of old parts of a specification in the creation of new parts, for the addition of new details, and for the modification of existing parts. Frequent changes in a distributed, iterative, and evolving drafting process need to be supported with minimal effort. Ripple effects on the document consistency, caused by the impact of a modification on other parts of the documents, need to be minimized.
- **Looseness:** in the early stages of the specification/design process, few details are available, and a specification technique should permit some level of incompleteness and non-determinism in a specification.
- **Maturity:** a technique has a high level of maturity mainly if it has undergone some certification process and if it has a history of use in various applications.

Validation and Verification (V&V)

- **Completeness and Consistency:** techniques should offer ways of checking completeness and consistency between partial functionalities, scenarios, and levels of abstraction.

- **Testing and Simulation:** V&V is greatly improved when specifications can be executed, animated, simulated, and tested. Also one should be able to obtain test cases from specifications.
- **Verifiability and Correctness:** verification of a model against requirement properties. Verification approaches are usually stronger than testing and simulation as they intend to prove that a property holds in general, but they are also harder and more costly to perform.

Tool Support

We are interested in techniques that are supported by tools for the capture, the editing, the maintenance, the animation, the testing, and the verification of specifications. We are especially looking for multi-platform, industrial-strength and quality tools, where support and training is available.

Training

- **Learning Curve:** we are interested in how quickly a new user can learn the concepts, theories, techniques, and tools to make useful application of the specification technique, and in how different is this technique from the current practice.
- **Tutorials and Documentation:** good tutorials and documentation are necessary for a good training. Courses, case studies, and other technology transfer activities are important as well.

These criteria will be used to compare the specification techniques introduced in the next section.

3.2.2 Overview of Selected Techniques

In this section, we give a short overview of six specification techniques (UCM, LOTOS, SDL, MSC, Petri Nets, and UML) particularly relevant to the scenario-based description of high-level specifications and designs of distributed systems and for the documentation of telecommunication standards. The selected techniques have all been used to describe real world problems and solutions.

Other specification and description notations, such as the Abstract Syntax Notation One (ASN.1) [201][337], Estelle [192], plain/extended/communicating Finite State Machines (FSM, EFSM, CFSM) [143], the Interface Description Language (IDL) [273], Larch [163], the Real-Time Object-Oriented Modeling (ROOM) [326], the Tree and Tabular Combined Notation (TTCN) [197][209], the Vienna Development Method (VDM) [219], and Z [336], are not discussed in this thesis. Although most of them have reached good levels of recognition in different areas (including telecommunications and distributed systems), and have been standardized in some cases, we believe they are less appropriate for the scenario-based specification and validation of telecommunications systems than the six techniques we selected. A minimum number of techniques is surveyed in order for this chapter to remain concise.

A few other formalisms are covered in a recent survey of specifications techniques for wireless standards [20], which served as a basis for this section. We also invite the interested reader to look at other studies from Ardis *et al.* [39] on specification methods for reactive systems, from Clarke *et al.* [97] on the state of in art in formal methods, and from Craigen *et al.* [104] on industrial application of these methods.

Because UCMs and LOTOS were already presented in Chapter 2, only the four remaining notations (SDL, MSC, Petri Nets and UML) are introduced in this section.

Specification and Description Language (SDL)

SDL [205] is an FDT designed for reactive, concurrent, real-time, distributed, and heterogeneous systems. The basic SDL model consists of extended finite state machines communicating by means of message queues. Notions of types and inheritance make SDL an object-based language. SDL is a language used to support human understanding of system descriptions, formal analysis and comparison of behaviours, in an implementation independent way. SDL is suitable for international standards in the telecommunication area, for systems in development, and for verification and validation of the system behaviour. SDL has two concrete syntaxes: the graphic representation called SDL/GR and the

textual representation called SDL/PR. The graphic form is more intuitive and displays relationships more clearly than the textual form. The language has two major features:

- An SDL system describes the application in the sense that many aspects (structure, interfaces, and behaviour) of the application are described.
- SDL is a high-level language. The extended finite state machine paradigm gives the designer a possibility to concentrate on the application problem and not to deal with low-level programming issues.

SDL is also being integrated, to some extent, to UML through a *profile* [207].

Message Sequence Charts (MSC)

The MSC notation, standardized by ITU-T [208][313], is a graphical and textual language for the description and specification of the interactions between system components. The main area of application for Message Sequence Charts is as an overview specification of the communication behaviour of real-time systems, in particular telecommunication switching systems. MSCs may be used for requirement specification, simulation and validation, test case specification and documentation of distributed systems.

MSCs focus on the communication behaviour of system components and their environment by means of message exchanges. A set of MSCs usually covers a partial system behaviour only since each MSC represents one scenario or several closely related scenarios. The main focus of MSCs is not on complete system descriptions but rather on the specification of special system properties or functions (i.e. scenarios). MSCs can also represent test purposes for the automatic generation of test cases. MSCs can be used as complement to SDL. Similarly to SDL, the MSC language has graphical (MSC/GR) and textual (MSC/PR) syntax forms. A recent enhancement, *High-level MSCs* (HMSC), includes control structures that can combine several MSCs. Under an apparent simplicity, MSCs can nevertheless lead to many subtleties and misinterpretations [236].

Petri Nets

Petri Nets (PNs) [281] are abstract machines used to describe system behaviour visually with a directed graph containing two types of nodes: places and transitions. *Places*, represented by circles, contain *tokens* whereas *transitions*, represented by lines, allow tokens to move between places. An event usually corresponds to the *firing* of a transition, which is allowed when all arrows entering the transition originate from places with tokens. PNs can be represented graphically, and they are usually formalized with simple mathematical arrays and functions. They can specify the logic of distributed systems at different levels of abstraction, and multiple techniques and tools can be used to verify them.

A problem with plain Petri Nets is the explosion of the number of elements of their graphical form when they are used to describe complex systems, hence they are seldom used nowadays in this form. However, numerous extensions have been suggested over the years to cope with this problem, many of which are supported by tools. For instance, *Design/CPN* is a widely used tool within the Petri Net community and has been developed for more than 10 years [95]. *Design/CPN* supports *Coloured Petri Nets* (CPNs) [216], an extension with complex data types (colour sets for tokens) and complex data manipulations (arc expressions and guards), both specified in the functional programming language ML. This tool also supports *Hierarchical CPNs*, i.e. net diagrams that consist of a set of separate modules (subnets) with well-defined interfaces. Other extensions to Petri nets include time, probabilities, communication, and even object orientation. Among them, *Object CPNs* support class nets, inheritance, (a)synchronous communication, and dynamic creation and destruction of nets [253].

Petri Nets and their variants offer both graphical and textual representations. CPNs are now being standardized in a superset called *High-Level Petri Nets* [199], which also include a textual format described in SGML [251].

Unified Modeling Language (UML)

The Unified Modeling Language (UML) is a general-purpose modelling language for specifying, visualizing, constructing and documenting the artifacts of software systems (in particular object-oriented and component-based systems), as well as for business modelling and other non-software systems. It includes many concepts and notations useful for the description and documentation of multiple models, and it enjoys a strong support from academic and industrial communities. UML 1.3 is the latest version of this OMG standard [274][358].

UML has a semi-formal semantic meta-model which defines basic modelling concepts such as objects and classes. This meta-model includes well-formedness rules expressed as formal constraints in the *Object Constraint Language* (OCL). UML is graphical notation that supports nine different diagram types, whose static semantics are defined in terms of the meta-model. These diagrams can be categorized into two sets. The first set, called *behavioural diagrams*, focuses mainly of functional and dynamic aspects of systems. It is comprised of five types of UML diagrams:

- **Use case diagrams:** Show actors and use cases together with their relationships. They describe system functionalities from the user's point of view.
- **Sequence diagrams:** Describe patterns of interaction among objects, arranged in a chronological order. They originate from Message Sequence Charts.
- **Collaboration diagrams:** Show the generic structure and interaction behaviour of systems.
- **Statechart diagrams:** Show the system in terms of a hierarchical state machine, with the events that cause the transitions of one state to another and the actions that result. They are based on Harel's notation [168].
- **Activity diagrams:** Capture the dynamic behaviour of a system in terms of operations. They focus on flows driven by internal processing. Activity diagrams share many characteristics with UCMs: focus on sequences of actions, guarded alternatives, and concur-

rency; start and end points have similar purpose; complex activities can be refined; and simple mapping to components can be achieved through *swimlanes*.

The second set, called *structural diagrams*, describe components and static characteristics of systems. It includes these four types of UML diagrams:

- **Class diagrams:** Capture the vocabulary of a system. They show the entities in a system and their general relationships.
- **Object diagrams:** Snapshots of a running system. They show object instances (with data values) and their relationships at some point in time.
- **Component diagrams:** Show the dependencies among software components.
- **Deployment diagrams:** Show the configuration of run-time processing elements and the software components, processes, and objects that live on them.

A textual representation of UML models and meta-models, the *XML Metadata Interchange* (XMI), has been included in the latest version of the standard [186].

3.2.3 Comparison Between Specification Techniques

Comparing such complex techniques represents a major challenge, and this section does not pretend to cover everything there is to say about them. However, the criteria presented in Section 3.2.1 will help to emphasize some of the main points of interest related to this thesis. The major strengths and weaknesses of each technique will be enumerated for each category of evaluation criteria.

Usability

UCMs, MSCs, and most UML diagrams can be read and understood by a wide variety of stakeholders, which is not always the case for the three other techniques, especially LOTOS which lacks a usable graphical representation. In terms of modularity, SDL has already reached a good level of maturity, while the others are still catching up: UCMs with libraries of plug-ins and patterns, LOTOS with E-LOTOS, MSCs with HMSCs, PNs with High-Level PNs, and UML with improved packages and profiles.

Abstraction is certainly a strength of UCMs, LOTOS and UML (e.g. activity diagrams), whereas MSCs and SDL require a commitment to fine-grained details (messages, entities, parameters, etc.). This is also true of scalability, where different mechanisms proved their usefulness in the past (plug-ins and stubs for UCMs, processes for LOTOS, and many other such strategies for UML and SDL).

Maintenance and evolution appear to be an issue for all of these techniques, but it is worse for MSCs and PNs due to the nature of these notations (arrows and lines/nodes) and to the use of disjoint scenario descriptions in plain MSCs. Improving the maintenance and evolution capabilities of such languages would contribute in a positive way to the handling of requirements changes. Looseness is best supported by UCMs, because useful descriptions can be achieved in early design stages when details are not always available. The other techniques need more details, and the FDTs (LOTOS and SDL) are especially demanding in terms of precision. Overall, most techniques are fairly mature and standardized (PNs are undergoing standardization), but the UCM notation is still under development and will hopefully be standardized within the next few years (work in this direction has already started in ITU-T Study Group 10 and in the OMG).

Validation and Verification

LOTOS, SDL and Petri Nets are well-suited for V&V and, by using them, many types of design errors and inconsistency and incompleteness problems have to be resolved at the time the specification is written. This is not the case for the other techniques, because many disjoint and loose descriptions can be created, which are prone to being inconsistent and incomplete.

Testing and simulation are well supported within the two FDTs and PNs, since they are executable languages. This is far from being the case for many UML diagrams (the main exception being Statechart diagrams, whose dynamic semantics in UML is still ambiguous). The new generation of tools based on the UML/SDL standard Z.109 however improves this situation by combining SDL simulation/testing capabilities to UML design [207]. UCMs are not executable as is, but the thesis intends to provide an alternative by mapping UCMs to an executable formalism.

Many techniques exist for verification and correctness checking of LOTOS, SDL, and PN specifications, and there is considerable experience in using them in specifications of real-life systems. However, UCMs and UML lack established verification frameworks.

Tool Support

Several industrial-strength tools are available for SDL and MSCs (e.g. Telelogic's Tau) and for UML (e.g. Rational Rose), and they have been used in telecommunications companies for a number of years. The three other techniques are also supported, to a lesser extent, by (university) tools. These tools are routinely used in research environments, so they are fairly robust, although they are not industrially supported.

Training

Because of the intuitive nature of UCMs, MSCs, and PNs, the learning curve is excellent and the techniques are easily accepted by many practitioners. It is possible to use these techniques at different levels of competence, and books and tutorials are available. LOTOS and its related methodology are well documented in books, tutorials, and papers. However, experience shows that the language is not easy to learn, although its order of difficulty does not exceed the one of many "unconventional" programming languages. To learn and use SDL and UML effectively, books, papers, technical reports, and a number of courses as well as commercial toolsets are available.

Summary and Discussion

According to our evaluation criteria, Table 7 summarizes the strengths and weaknesses of the selected specification techniques.

TABLE 7. Evaluation of the Selected Specification Techniques

Technique	Readability	Modularity	Abstraction	Scalability	Maintenance & Evolution	Looseness	Maturity	Completeness & Consistency	Testing & Simulation	Verifiability & Correctness	Tool Support	Learning Curve	Tutorials & Doc
UCM	+	0	+	+	0	+	-	-	-	-	0	+	0
LOTOS	-	0	+	+	0	-	+	+	+	+	0	-	+
MSC	+	0	-	-	-	0	+	-	0	0	+	+	+
SDL	0	+	-	+	0	-	+	+	+	+	+	0	+
Petri Nets	0	0	0	0	-	0	0	+	+	+	0	+	+
UML	+	0	+	+	0	0	+	-	-	-	+	0	+

Legend: + = Strength; 0 = Adequate; - = Weakness.

Note that UCMs have several properties suitable for the representation of requirements and high-level designs (readability, abstraction, scalability, looseness and ease of learning), while LOTOS complements most of UCMs weak areas related to the analysis of requirements (maturity, completeness & consistency, testing & simulation, verifiability & correctness). Hence, we have reasons to believe these two complementary notations to be a particularly good match. Moreover, UCMs and LOTOS offer similar constructs (such as sequence, alternative, parallelism, hiding, and structure, i.e. stubs in UCMs and processes in LOTOS), which result in simpler mapping and traceability relations. In particular, both UCMs and LOTOS aim to represent the ordering of abstract events in a system. These are two of the main reasons why SPEC-VALUE uses those notations.

The complementarity of LOTOS and system behaviour paths has also been observed by Vigder [362]. In his thesis, Vigder used a very preliminary path notation called *slices*, which guided the

design of component-based concurrent systems. These designs were translated (manually) to LOTOS models, which provided formal semantics and enabled analysis. No validation strategy was proposed, and the practicality of the work was somewhat limited. Nevertheless, Vigder's work provided some inspiration for the creation of Use Case Maps and for SPEC-VALUE.

UCMs and LOTOS being mutually complementary does not mean that UCMs could not be combined to other techniques, on the contrary. We believe most design methodologies could benefit from the combined use of different specification techniques, which often bring different viewpoints and complementary strengths. For instance, SDL and UCMs also complement each other, and this research direction is being studied by Sales and Probert [317][318]. However, the UCM-LOTOS combination is the only avenue pursued here, other combinations being beyond the scope of this thesis.

3.3 Scenarios

Over the last few years, there has been a strong interest, in both academia and industry, in the use of *scenarios* for requirements engineering and system design, testing, and evolution [107][185][368]. Scenarios are known to help describing functional requirements, uncovering hidden requirements and trade-offs, as well as validating and verifying requirements. The introduction of *use cases* in the object-oriented world confirmed this trend almost a decade ago [212].

The exact definition of a scenario may vary depending on used semantics and notations, but most definitions include the notion of a *partial* description of system usage as seen by its users or by related systems [305]. There is no clear separation between the meanings of use case and scenario. In UML, use cases are defined as sequences of actions a system performs that yield observable results of value to a particular user (actor) [274]. In the object-oriented community, use cases are interpreted as classes of related scenarios, where scenarios are sequential and where use case parameters are instantiated with concrete values. Hence, a scenario is a specific realization of a use case [274][304]. However, the requirements engineering community sometimes sees multiple use cases as being contained in a scenario. In this thesis, the terms “use cases” and “scenarios” are used interchangeably, although *sequential scenarios* will refer to instantiated sequences of events or actions.

Many scenario-driven methodologies are now available and they often have a high degree of acceptance because of the intuitive and linear nature of scenarios [215][368]. Scenarios can be related to traces (of internal actions and external events), to message exchanges between components, to interaction sequences between a system and its user, to a more or less generic collection of such traces, etc. Numerous notations are also used to describe scenarios: semi-formal pictures [212], natural language or structured text [126][274][304], grammars or automata [183], boolean or logic expressions [340], tables [98], and message exchange diagrams similar to MSCs [212][274][304][333], to mention but a few. The approaches available differ on many aspects, depending on the definition and the notation used. Scenarios are used not only to elicit requirements and produce specifications, but also to drive the design, the testing, the overall validation, and the evolution of the system. This section introduces several scenario notations and gives a short comparison based on eight criteria. It also discusses several design processes that make use of scenarios, as well as techniques for the analytic and synthetic construction of component behaviour from scenarios.

3.3.1 Why Scenarios?

One frequent problem requirements engineers are faced with is that stakeholders may have difficulties expressing goals and requirements in an abstract way [238]. Typical usage scenarios for an hypothetical system may be easier to obtain than goals or properties when the system understanding is in its infancy. This fact has been recognized in cognitive studies on human problem solving [47] and in research on inquiry-based requirements engineering [288].

The use of scenarios for requirement engineering and system design bears benefits and drawbacks. A non-exhaustive list of the most relevant ones follows in Table 8.

TABLE 8. Benefits and Drawbacks of Scenarios

Benefits	Drawbacks
<ul style="list-style-type: none"> • Scenarios are intuitive and relate closely to the requirements. Different stakeholders, such as designers and users, can understand them. They are particularly well suited for operational descriptions of reactive systems. • They can be introduced in iterative and incremental design processes. • They can abstract from the underlying system structure, if necessary. • They are most useful for documentation and communication. • They can guide the generation of requirements-based tests used for validation at different levels (specification, design and implementation). • They can guide the construction of more detailed models and implementations. 	<ul style="list-style-type: none"> • Since scenarios are partial representations, completeness and consistency of a set of scenarios are difficult to assess, especially when the scenarios are not described at a uniform abstraction level. • Scenarios are not able to express most non-functional requirements. • Scenarios often leave required properties about the intended system implicit. • The synthesis of components behaviour, from a collection of scenarios, remains a complex problem. • The use of scenarios leads to the usual problems related to traceability with other models used in the design process. • Getting and maintaining the right granularity for the scenarios can be a real challenge. • Design approaches based on scenarios are rather recent and seldom possess a high level of maturity. Scalability and maintainability represent notably important issues.

The increasing popularity of scenarios makes us believe that their benefits outweigh their drawbacks. Further, these drawbacks can often be cured by using scenarios in conjunction with other techniques. In spite of the fact that this thesis intends to emphasize the benefits of scenarios, many issues related to scalability, maintainability, completeness, consistency, synthesis, and traceability will be addressed as well.

3.3.2 Evaluation Criteria for Scenario Definitions

Many definitions of the term “scenario” exist, and it would be impossible to enumerate them all. However, the following collection of eight important criteria will help to categorize and compare many scenario notations [29]:

- **Component-centered:** Scenarios can be described in terms of communication events between system components only (i.e. component-centered), or else independently from components, in a pure functional style (end-to-end). This is a very important criterion as many notations focus solely on interactions between components, while in our view these

interactions are secondary and result from the need to implement the causality relationships linking responsibilities (processing activities) in different components. An early focus on messages may lead to system overspecification and may prune out other appropriate options.

- **Hiding:** Scenarios could describe system behaviour with respect to their environment only (black-box), or it could include internal (hidden) information as well (grey-box). According to Chandrasekaran [90], the most important reason that impeded the progress of various large projects he studied is the lack of internal details in scenarios. Essentially, treating the system like a black box in a scenario model means that there shall be no consideration of implementation constraints while describing scenarios. It does not mean that a scenario shall not delve into details of requirements on internal system functionality. Zave and Jackson present a different viewpoint and claim that when it comes to requirements, the environment is not the most important thing — it is the only thing [383] (this view is shared by Probert and Wei in [295]). They suggest to avoid any implementation bias on the basis that requirements are supposed to describe what is observable at the interface between the environment and the system, and nothing else about the system. Our opinion is more in line with Chandrasekaran's: shared events, whether they are controlled by the system or by the environment, are insufficient. Many implementation constraints are not necessarily premature design decisions, but in fact non-functional requirements. Additionally, there comes a point where the gap between requirements and high-level designs or implementations needs to be filled (an important topic of this thesis), and descriptions of activities performed internally by the system can then be of tremendous help.
- **Representation:** Scenarios can be described in various ways, for instance with semi-formal pictures, natural language, structured text, grammars, trees, state machines, tables, and sequence diagrams. Graphical representations are often better understood by a wide range of stakeholders, whereas structured textual languages are often less constrained in terms of expressiveness. The level of formality has also an impact on the usefulness of a

- notation: less formality is better for requirements, but more formality is desirable for detailed design and automated model transformations or code generation.
- **Ordering:** Scenarios represent a collection of events ordered according to *time* only or to *causality*. Causal ordering is very important when concurrency is involved, otherwise concurrent actions expressed with a time ordering might result in logical fallacies in the requirements (see the explanation of *Post hoc, ergo propter hoc* below the title of this chapter). Causality is further discussed in Section 3.1.
 - **Multiplicity:** We can either have one *single* trace only (i.e. a sequential scenario) or possibly *multiple* related traces per scenario. Having multiple scenarios linked together leads to more concise descriptions and to a better understanding of the integration of scenarios, whereas the availability of individual scenarios eases the construction of traceable links across design models. But obviously, a notation that can support multiple scenarios can support single scenarios as well.
 - **Abstraction:** An *abstract* scenario is generic, with formal parameters, whereas a *concrete* scenario focuses on one specific instance, with concrete values. Abstraction is beneficial in the early stages of design (e.g. requirements capture) and for capturing families of scenarios that differ only by their concrete values. Notations that focus on concrete scenarios however ease the transition towards detailed models (e.g. state machines), test cases, and implementations.
 - **Identity:** Scenarios can focus on *one actor* or target *many* actors at once. The later is seen as a major benefit in terms of expressiveness.
 - **Dynamicity:** A scenario notation is *dynamic* when it enables the description of behaviour that modifies itself at run-time, otherwise it is said to be *static*. Emerging telecommunication services enabled by IP networks, agent systems, and negotiation mechanisms can benefit from notations that can express dynamicity.

Obviously, other sets of criteria could be defined. For instance, Cockburn uses four dimensions to use case descriptions, namely *purpose*, *content*, *plurality*, and *structure* [98]. Purpose can be

either for stories (explanations) or for requirements. Content can be either contradicting, consistent prose, or formal content. Plurality is either 1 or multiple, similar to our multiplicity. Structure can be unstructured, semi-formal, or formal. This dimension shares some common characteristics with our representation criterion. According to this classification, Use Case Maps' purpose could be both stories and requirements, whereas the other criteria would be evaluated respectively to consistent prose, multiple, and semi-formal. Rolland *et al.* suggest yet another set of criteria in [311], but without a real emphasis on specific needs of telecommunication systems.

The next section does not attempt to provide a single scenario definition. Instead, it presents and compares different notations according to the selected criteria.

3.3.3 Overview of Selected Scenario Notations

There are dozens of scenario notations used for the description of system usage, goals, and business logic. For example, Hurlbut's thesis surveyed and compared nearly sixty different scenario, use case, and policy formalisms and models [184][185], and others are likely to emerge in the upcoming years. This section focuses on selected scenario notations particularly relevant to the telecommunications domain, and it provides a concise comparison in terms of the criteria seen in Section 3.3.2.

Message Sequence Charts

The scenario notation the most commonly used by telecommunications companies and standards bodies is undoubtedly Message Sequence Charts. MSCs are essentially graphical (although a textual machine-processable format exists), composed of concrete events (messages), and centered towards components. MSCs can represent internal actions and multiple actors. While conventional MSCs mostly use time ordering and single traces (MSC'2000 now enables multiple traces), High-Level MSCs focus more on multiple structured scenarios and also on causality.

MSCs have been used by many people to formalize scenarios. Kimbler *et al.* use them to create Service Usage Models, which describe the dynamic behaviour of the system services from the user's perspective [229][304]. Andersson and Bergstrand also present a method to formalize use cases that introduces an unambiguous syntax through MSCs [31].

Use Cases

Jacobson's use cases are prose descriptions of behaviour from the user's perspective [212]. They are mostly black-box, i.e. they focus on the interactions between actors and systems. Use case diagrams offer a graphical means by which use cases can be related to each other. They offer relations such as *uses* and *extends*, which allow for uses cases to reuse (part of) other scenarios. Use cases can be of two kinds: basic courses, for normal scenarios, and alternative courses, which include fault-handling scenarios. Use cases are mostly based on a time ordering, they represent multiple abstract scenarios, and they may involve many actors.

CREWS-L'Ecritoire

CREWS, the European ESPRIT project on *Cooperative Requirements Engineering With Scenarios* [126], proposes structured narrative text for capturing requirements scenarios, together with a set of style and content guidelines [46]. These are supported by a tool called *L'Ecritoire* [312] and, to some extent, by the SAVRE tool [252].

In a way similar to Jacobson's use cases, these scenarios are divided into two main categories: normal scenarios and extension scenarios. The latter can be either normal (alternatives) or exceptional, depending on whether they allow to reach the associated goal or not. This notation supports multiple actors and abstract scenarios, focuses on external events, is centered towards components, and uses time ordering.

Scenario Trees

Hsia *et al.* suggest the use of scenario trees that represent all scenarios for a particular user [183]. Similarly to LTSs, scenario trees are composed of nodes, which capture system states, and of arcs representing events that allow the passage from one state to the next. They also focus on interactions between actors and the system, they use time ordering, and they can be abstract.

This notation is best suited for a single thread of control and well defined state transition sequences that have few alternative courses of action and no concurrency, which is seldom the case in

real telecommunications systems. Regular expressions are used to formally express the user scenario that results in a deterministic finite state machine.

Use Case Trees

Boni Bangari proposes Use Case Trees (UCTs) as a text-based notation for describing scenarios related to one entity [59]. This notation, inspired from TTCN [197][294], captures sequential and alternative scenarios in terms of messages. These messages are sent and received through points of control and observations (PCOs) belonging to an actor under test. The grammar-like representation allows for sub-trees, timer events and data parameters (assignments, operations and qualifiers) to be defined and used. An interesting property of UCTs is that sequential scenarios can be automatically derived (usually as Message Sequence Charts) and characterized as normal, low risk, or high-risk scenarios. This notation is potentially useful for defining compact validation test suites targeted towards the system as a whole or towards single components. However, the lack of support for concurrency, multiple entities and hiding limits its usefulness as a requirements notation.

Chisel Diagrams

Aho *et al.* have performed empirical studies with telecommunication engineers to create the Chisel notation [4]. The graphical language Chisel is used for defining requirements of telecommunication services. Chisel diagrams are trees whose branches represent sequences of (synchronous) events taking place on component interfaces. Nodes describe these events (multiple concurrent events can take place in one node) and arcs, which can be guarded by conditions, link the events in causal sequences. Multiple abstract scenarios and actors can be involved, but internal actions are not covered. The interested reader can find further information on the transition from Chisel to UCMs in [18], and from Chisel to LOTOS in [354][357].

Statechart Diagrams

Glantz uses Harel's Statechart notation [168], now part of the Unified Modeling Language, as a way of capturing scenarios [154]. This results in a formal notation for validating and simulating a behav-

journal model representing the external view of a system. Scenarios must be structured such that they are all disjoint. Any overlapping scenarios must be either merged into a single scenario or partitioned into several disjoint ones. Such structuring allows for each scenario to be modelled by a closed State-chart, i.e. a single initial state and a single terminal state, with other states in between. Composition of scenarios is performed through sequence, alternative, iteration, or concurrency declarations. These scenarios support causal ordering, multiple actors, and multiple abstract scenario sequences.

Life Sequence Charts

Damm and Harel propose Life Sequence Charts (LSCs) [106], which enrich MSCs with a concept called *liveness*. Liveness enables one to specify mandatory scenarios as well as forbidden scenarios (e.g. to capture safety requirements) through the same representation. Although the liveness concept is certainly useful and leads to more accurate component descriptions, LSCs satisfy essentially the same criteria as HMSCs.

Somé's Scenarios

Somé *et al.* represent timed scenarios with structured text, but also with a formal interpretation where preconditions, triggers, sequence of actions, reactions and delays are specified [332][333][334]. Scenarios are interpreted as timed sequences of events, which make them appropriate for real-time systems. External events represent interactions between components, including actors, whereas actions can be internal. These textual scenarios can also be represented graphically. Somé extended the MSC notation to support additional scenario elements such as conditions and expiration delays (now covered to some extent by HMSCs).

Multiple abstract scenarios and actors can be considered by these component-based notations. They are ordered according to time, although non-linear causality appears when composing the scenarios together to form an automaton.

RATS

In his RATS (*Requirements Acquisition and specification for Telecommunication Services*) methodology [120], Eberlein uses three different scenario representations: textual (natural language), structured (in text, but with pre/post/flow conditions) and formalized (structured text, more component-centered). The aim of having these three notations is to allow a smooth and gradual transition from a service description in natural language to a formal specification in SDL. Scenarios are divided into normal, parallel/alternative, and exceptional behavior, in order to help the developer to first focus on the most common behavior and then later on the less common system functionality. The use cases can be structured hierarchically in overall use cases of higher abstraction. Most scenarios are abstract and linear, although *overall scenarios* capture multiple scenarios, with a causal ordering. The methodology has been implemented in a prototype of the RATS tool, a client-server-based expert system.

UML Activity Diagrams

All UML behavioural diagrams can be used to describe scenarios. Four of them have already been discussed in some form in this section: Jacobson's use cases and use case diagrams, sequence diagrams (similar to MSCs, although less expressive than Z.120), collaboration diagrams (same information as MSCs, but with a two-dimensional view of the component architecture), and Statechart diagrams. The last type, activity diagrams, stands out as an interesting way of capturing scenarios. Activity diagrams capture the dynamic behavior of a system in terms of operations. They focus on end-to-end flows driven by internal processing. Activity diagrams share many characteristics with UCMs: focus on sequences of actions, guarded alternatives, and concurrency; complex activities can be refined; and simple mapping of behavior to components can be achieved through vertical *swimlanes*. However, activity diagrams do not capture dynamicity well, and the binding of actions to "components" is semantically weak in the current UML standard.

Use Case Maps

UCMs are discussed thoroughly in this document, but let us recall that they represent multiple abstract scenarios through visual paths linking responsibilities. The latter can be allocated to compo-

nents or users (actors), yet interactions between component, which implement the causal flow of responsibilities, are left to a more detailed stage of the design process. UCMs can also capture run-time self-modifying behaviour through dynamic stubs and dynamic responsibilities.

Summary and Discussion

The thirteen scenario notations compared in this paper are summarized in Table 9. Due to some major differences, HMSCs are considered separately from basic MSCs in this table.

TABLE 9. Comparison of the Selected Scenario Notations

Scenario Notation	Comp.-centered or end-to-end	Hiding	Representation	Ordering	Multiplicity	Abstraction	Identity	Dynamicity
MSC	Comp.-centered	Yes	Sequence Diagram	Time	Single	Concrete	Many	Static
HMSC	Comp.-centered	Yes	Sequence Diagram	Causal	Multiple	Concrete	Many	Static
Use Case	Comp.-centered	No	Text	Time	Multiple	Abstract	Many	Static
CREWS'	Comp.-centered	No	Structured Text	Time	Multiple	Abstract	Many	Static
Scen. Tree	Comp.-centered	No	Tree & Grammar	Time	Multiple	Abstract	One actor	Static
UCT	Comp.-centered	No	Text & Grammar	Time	Multiple	Concrete	One actor	Static
Chisel	Comp.-centered	No	Tree	Causal	Multiple	Abstract	Many	Static
Statechart	Comp.-centered	No	State Machine	Causal	Multiple	Abstract	Many	Static
LSC	Comp.-centered	Yes	Sequence Diagram	Causal	Multiple	Concrete	Many	Static
Somé's	Comp.-centered	Yes	Structured Text & Sequence Diagram	Time	Multiple	Abstract	Many	Static
RATS	Either type	Yes	Structured Text	Causal	Multiple	Abstract	Many	Static
UML Act-ivity Diag.	End-to-end	Yes	Paths on Swimlanes	Causal	Multiple	Abstract	Many	Static
UCM	End-to-end	Yes ^a	Paths on Components	Causal	Multiple	Abstract	Many	Dynamic

a. In bound UCMs, what is inside components is usually assumed to be hidden.

MSCs are most useful for single scenarios, especially when expressing lengthy black-box interactions between actors and a given system (something that UML activity diagrams and UCMs do not do well). However, MSCs are not appealing for structuring related scenarios. HMSCs and LSCs are more powerful and expressive, but they still require an early commitment to components. Use cases and UCTs are generally not used to describe internal responsibilities and they do not support causal ordering. CREWS' scenarios improve on use cases by using structured text and guidelines, yet

they have essentially the same limitations. Scenario trees and UCTs focus on only one actor at a time, which is often not desirable when describing telecommunications systems requirements. Chisel diagrams represent a good alternative to scenario trees, but they still focus on interactions between components. Somé's scenarios lack the causal ordering that only appears when scenarios are transformed into component automata.

UCMs, RATS, and UML activity diagrams stand up as being the only surveyed scenario notations that are not component-centered but define end-to-end behavior. This is useful for early descriptions of requirements and helps to avoid overspecification. Also, UCMs stress causality relationships that can span many components. UML activity diagrams can, to some extent, present a similar view with swimlanes, but swimlanes are semantically weak and they cannot represent the architecture in two dimensions (swimlanes show components as columns). RATS scenarios can capture non-functional information, unlike most other notations. They have many of UCMs' characteristics, but UCMs have only one type of scenarios (not three as in RATS), and they are graphical, a property that makes them appealing to a variety of stakeholders. UCMs can also capture dynamicity through dynamic stubs (with multiple sub-maps selected at run-time) and dynamic responsibilities (which can move sub-maps around and store them in pools of sub-maps). This useful feature, fairly unique to UCMs, enables the description of emerging telecommunication services based on agents and dynamic selection of negotiation mechanisms.

Overall, we believe the UCM notation to have very good features for the capture of requirements and the description of high-level designs. Moreover, LOTOS is one of the few formal technique surveyed that can specify scenarios that are component-centered as well as those that are not (SDL requires components), hence it can support a progression from system requirements descriptions to component-centered high-level designs. SPEC-VALUE can therefore take advantage of this added value provided by the UCM-LOTOS combination.

3.3.4 Construction Approaches

In the scenario-driven development of telecommunication systems and services, it is important to leverage the investment in scenarios in order to generate systems rapidly, at low cost, and with a high

quality. To support the progression from scenarios capturing requirements and high-level functionalities to detailed designs and implementations based on communicating entities, we can learn much by examining different construction approaches used in the protocol engineering discipline, where the construction of a model based on another model is a concept supported by many techniques. The construction of models from scenarios is nowadays getting a lot of attention from academia and industry [341]. This section introduces and compares many construction approaches.

Protocol Engineering Approaches

In the field of protocol engineering, the construction of a model based on another model is a concept supported by many approaches. In [293], Probert and Saleh present two categories of construction approaches for communication protocols that can be generalized to most reactive and distributed systems:

- **Analytic approach:** this is a build-and-test approach where the designer iteratively produces versions of the model by defining messages and their effect on the entities. Due to the manual nature of this construction approach, which often results in incomplete and erroneous model, an extra step is required for the analysis, verification (testing), and correction of errors.
- **Synthetic approach:** a partially specified model is constructed or completed such that the interactions between its entities proceed without manifesting any error and (ideally) provide the set of specified services. For properties preserved by such approaches, no verification is needed as the correctness is insured by construction.

In particular, Saleh surveyed multiple synthesis techniques applied to two protocol engineering domains:

- Synthesis of protocol specifications from service specifications [293][315]. In a layered reference model like OSI, this problem relates to the design of the protocol specification of layer N from the service specifications of layers N and N-1. The usefulness of service specifications is emphasized in [363].

- Synthesis of protocol converters [316]. This problem is formulated as the design of a converter for the interworking between two incompatible protocols, at layers N and M, given the formal specification of these protocols and/or the services they provide.

TABLE 10. Benefits and Drawbacks of Protocol Engineering Construction Approaches

Construction Approach	Benefits	Drawbacks
Analytic	<ul style="list-style-type: none"> • No formal source model required. • Both the source and target models can exploit the richness of their respective modelling language to their full extent. • The constructed model can more easily take into consideration design or implementation constraints (e.g. to reflect the high-level design), and be optimized accordingly. • Non-functional requirements (e.g. performance and robustness) can more easily be taken into consideration. 	<ul style="list-style-type: none"> • Transformation mostly manual. • Errors may result from the construction. • Verification is required. • Many iterations may be required to fix the errors detected during verification. • Time-consuming.
Synthetic, interactive	<ul style="list-style-type: none"> • Improper synthetic constructions can be avoided by interacting with the designer. • Correctness “ensured” by construction (under certain assumptions). Many faults are therefore avoided. • Verification theoretically not required. • Only one iteration required. • Quick construction. 	<ul style="list-style-type: none"> • Not fully automated. • Requires a formal source model. • May require a partially constructed model to be available. • Both source and target modelling languages are usually restricted in style and content. • Requires more details in the source model than non-automated approaches. • Difficult to take into consideration design/implementation constraints, optimizations, and non-functional requirements. • Resulting model usually hard to understand, maintain and extend.
Synthetic, automated	<ul style="list-style-type: none"> • Fully automated. • Correctness “ensured” by construction (under certain assumptions). Many faults are therefore avoided. • Verification theoretically not required. • Only one iteration required. • Very quick construction. 	<ul style="list-style-type: none"> • Requires a formal source model. • Both source and target modelling languages are usually restricted in style and content. • May result in improper synthetic constructions in ambiguous cases (the algorithm makes the decisions, not the designer). • Requires more details in the source model than non-automated approaches. • Resulting model usually hard to understand, maintain and extend.

Synthetic approaches may or may not be fully automated. Sometimes, they require the *interactive* participation of the designer as some decisions need to be taken along the way. In both cases, synthetic approaches require the source model to be described formally (usually with some automata model or with FDTs), whereas analytic approaches may start with semi-formal or informal models. Analytic and (automated) synthesis approaches have many other benefits and drawbacks, some of which are summarized in Table 10.

We have no intention of surveying the myriad of approaches for the synthesis of protocols or converters. However, we can build on the benefits and drawbacks presented here to evaluate construction techniques based on scenarios that are applicable to telecommunications systems in general.

Comparison Criteria for Model Construction

The construction of models that integrate scenarios represents a problem similar to those faced by the protocol engineering community. A collection of scenarios often needs to be checked for completeness, consistency, and absence of undesirable interactions. To do so, most V&V techniques require that a model which integrates these scenarios be available. Also, it is often desirable to map the scenarios onto a component architecture at design time in order to enable the generation of component behavior in distributed applications (e.g. telecommunication systems). These two construction levels are of particular interest for this thesis:

- P1) Integration of a collection of requirements scenarios in an abstract model used for the analysis of requirements. No components are required here.
- P2) Integration of a collection of scenarios in a component-based model used not only for the analysis of the requirements, but also as a high-level design which considers some implementation issues.

Different approaches targeting these two levels are already available, twenty of which are reviewed next. Additional evaluation criteria include:

- Type of construction approach: analytic, synthetic non-automated, or synthetic automated.

- Source scenario notation, such as the ones overviewed in Section 3.3.3.
- Target construction model (SDL, UML Statecharts, automata, LOTOS, etc.).
- Whether the scenario model requires explicit components and messages.

Non-Automated Analytic Approaches

The *Usage Oriented Requirements Engineering* (UORE) approach proposed by Regnell *et al.* [304][306] builds on the Objectory method [212] and adds a construction phase (unfortunately called synthesis in their work) where use cases are integrated manually into a *Synthesized Usage Model* (SUM). This “synthesis”, which addresses level *P1*, is composed of three activities: formalization of use cases (using an extended MSC notation), integration of use cases (which produces usage views, one for each actor/component), and verification (through inspection and testing). The resulting SUM is a set of automata whose purpose is to serve as a reference model for design and V&V, including Cleanroom’s statistical usage testing [259][307] and dynamic testing [305]. No automated support is provided yet.

In RATS, Eberlein provides informal guidelines [120]. Non-functional requirements have to be refined into either functional requirements or implementation constraints. The functional requirements have to be expressed in textual use cases. The user then has to define states in the system behavior. Adding pre-, flow- and post-conditions results in structured use cases. The most formal use-case notation uses atomic actions, which still contain textual descriptions. These formalized use cases are then mapped to SDL flowchart constructs in order to address level *P2*. The approach does not go deeply into the construction of the SDL model as RATS focuses more on the acquisition and the specification of requirements (including non-functional ones).

In his thesis, Bordeleau addresses *P2* by defining the *Real-Time TRaceable Object-Oriented Process* (RT-TROOP), which combines the use of scenario textual descriptions (use cases), UCMs, MSCs, and ROOM (UML-RT) [62]. Included is an approach where UCM scenarios are first transformed into HMSCs, and then into hierarchical CFSMs (ROOMCharts) [61]. No construction algorithm is proposed, but the use of transformation patterns is suggested instead. Several such patterns

are provided for the UCM-HMSC mapping [63], and for the construction of ROOMCharts from HMSCs. HMSCs are used to fill the gap between UCMs, which abstract from message exchanges, and the state machines, which describe the behaviour of the actors/components involved. Traceability relationships are also defined in this process. RT-TROOP focuses more on design than on requirements validation because verification of the ROOM model is limited (especially when compared to FDTs). ObjecTime, ROOM's tool, supports animation and a limited form of testing based on MSCs, but at the same time it supports automatic code generation.

Krüger *et al.* present a related technique for the transformation of a set of MSCs to a State-chart model [235], hence addressing *P2*. The construction takes into consideration the type of semantics associated to MSCs, e.g. whether there are fewer, more, or the same number of components in the system than what is found in the MSCs, or whether additional messages (from another scenario) are allowed or forbidden between two messages in a component, etc. This technique is however very immature at this point and it is not supported by algorithms or tools.

According to Lamsweerde and Willemet, a drawback of scenarios is that system properties are often left implicit. If these properties were explicit (e.g. in declarative terms), then consistency/completeness analysis would be much easier to carry out. Lamsweerde and Willemet address *P1* by exploring the process of inferring (by induction) formal specifications of such properties (goals) from scenario descriptions [238]. Their scenarios are sequential and synchronous interaction diagrams whereas their goals are linear temporal logic properties expressed in the KAOS language. The scenarios can either be positive (must be covered) or negative (must be excluded). Their technique represents a novel and promising contribution, but it remains analytic as it requires validation to be performed because inductive inference is not sound. This approach is not yet supported by tools.

Yee and Woodside developed a transformational approach to process partitioning using timed Petri Nets [380], which addresses *P2*. An abstract scenario model combining both the system and its environment (*Process Specification of Requirements* — PSR) is partitioned, using a collection of correctness preserving transformations (abstraction, refinement, sequentialization, partitioning, and

resource access control), into a collection of communicating processes that can represent system components (*proto-design*). Both the source and the target models are described using timed Petri Net, and the transformations ensure their behavioural equivalence from the environment viewpoint. Being executable, the target model can be used for analysis and for performance evaluation of alternative architectures. The source model does not require any component, but the selection and application of the transformations are manual.

Non-Automated Synthesis Approaches

Desharnais *et al.* propose a synthesis approach for the integration of sequential scenarios represented in state-based relational algebra [114]. The initial scenarios involve the system and a single actor (concurrency is not involved), and the result is one large scenario represented again in relational algebra. As a result, level *P1* is addressed. Although the authors claim that data and complex conditions being incorporated in the formalism represent an advantage over other approaches, their technique seems somewhat limited in terms of usability and scalability for realistic telecommunication systems.

In his thesis [333][334], Somé proposes a composition algorithm that transforms his scenarios into Alur's timed automata [7], one for each component (hence addressing *P2*). This synthesis algorithm is implemented in a prototype tool, where consistency and completeness issues in the scenarios are resolved through the interactive assistance of the requirements engineer. The synthesis is based on the common preconditions rather than on the sequences of actions. Super-states are used when the preconditions of one scenario are included in that of a second scenario. The algorithm preserves the temporal constraints associated to the scenarios, which is seldom the case of other (semi-)automated synthesis techniques.

Harel and Kugler propose an algorithm for the synthesis of Statecharts from a subset of the Life Sequence Charts (LSCs — [106]) notation, without data or conditions [169]. This algorithm decides the satisfiability and consistency of a set of LSCs, something that is harder to do than for MSCs due to the possibility of expressing forbidden scenarios. The algorithm then produces a global system automaton. In order to address *P2*, this global automaton can be distributed (as Statecharts)

over the set of components involved in the LSCs. These components share all their information with each other, which simplifies the synthesis algorithm. This work is promising but it is not yet supported by tools.

Alur *et al.* have an algorithm which transforms a set of stateless basic MSCs into communicating state machines of various types (level *P2*) [9]. This technique supports the detection of implied scenarios resulting from the composition of multiple MSCs. Alur's algorithm uses a language-theoretic framework with closure conditions. Its emphasis is on safety and on efficiency (it executes in polynomial time), and it can generate counter-examples for non-realizable sets of MSCs. The detection is based on previous work done in collaboration with Holzmann and Peled [8], who extended this work in another direction to support HMSCs during requirements analysis with the tool UBET [182][249].

Mäkinen and Systä developed an approach and tool to synthesize UML Statechart diagrams from a set of UML sequence diagrams [254], hence addressing *P2*. Since fully automated synthesis may overgeneralize the Statechart and may introduce more scenarios than described in the sequence diagrams, the MAS (*Minimally Adequate Synthesizer*) approach is interactive. MAS models the synthesis process as a language inference problem and translates sequence diagrams first into traces, then into finite state automata, and finally into Statechart diagrams. The interactive part of the tool asks membership queries visualized as sequence diagrams (in a nutshell: "Is this sequence diagram acceptable?"), which allow the derivation of a consistent and deterministic Statechart diagram. Counter-examples can be provided when appropriate.

Automated Synthesis Approaches

With their SCED methodology [232], Koskimies *et al.* propose a synthesis algorithm based on that of Biermann and Lrishnaswamy [49], the latter being available since the mid-70's. SCED's synthesis algorithm integrates *scenario diagrams*, an extension of the basic MSC'92 notation with iterations, conditions, and sub-scenarios (thus more in line with MSC 2000), and outputs OMT state diagrams [314], which are based on Harel's Statecharts. The synthesis is supported by the SCED tool [233],

which also contains visual editors for scenario diagrams and state diagrams. The state machine generated by the tool is minimal with respect to the number of states necessary to support the scenarios. The authors claim that their approach is not tied to the OMT methodology, and hence can be reused in other contexts to address the level *P2*.

Schönberger *et al.* have developed another algorithm based on a similar idea [322], only this time they start with another type of scenario notation: UML collaboration diagrams. Their synthesis procedure addresses *P2* by generating UML Statecharts, which make extensive use of concurrency constructs to satisfy the inherent concurrency found in collaboration diagrams (but absent from Koskimies' scenario diagrams). Although their algorithm does not output a minimized state machine, the authors provide several state diagram compression techniques. This procedure has a polynomial complexity and is not incremental, whereas Koskimies' approach is incremental but with an exponential complexity. A prototype tool implements this algorithm, and it can be used to generate graphical user interfaces automatically, provided that the initial collaboration diagrams include appropriate additional information [125].

Whittle and Shumann [374] propose an algorithm for the generation of UML Statecharts from a collection of UML sequence diagrams (addresses *P2*). It allows for conflicts to be detected and resolved through UML's Object Constraint Language (OCL) and global state variables. These Statecharts can be non-deterministic. The target Statechart model is intended to be highly structured (hierarchical) and readable in order to be modified and refined by designers. This algorithm shares similarities with the work of Schönberger [322] and Somé [333] as the hierarchical nature of the states is inferred. However, the synthesis is also influenced by structure elements found in other types of UML diagrams such as class diagrams. The approach is supported by a prototype tool written in Java.

Leue *et al.* have developed two algorithms for the automated synthesis of Real-Time Object-Oriented Modeling (ROOM) models [326] from standard HMSC scenarios [244]. Essentially, ROOM-Charts (hierarchical state machines similar to Harel's) are generated for each actor in the HMSCs,

hence addressing *P2*. One major assumption is that the basic MSCs referenced by the HMSC are mutually exclusive, i.e. unlike SCED, only one scenario is active at any time. This results in simpler synthesis algorithms. The first algorithm, called *maximum traceability*, preserves the HMSC structure in the synthesized model. The second one, called *maximum progress*, generates smaller state machines but sacrifices traceability with respect to HMSCs. The properties preserved by these algorithms are still under investigation. Both algorithms are implemented in the MESA toolset [45], and their authors claim that their work can be adapted to support SDL and UML.

Mansurov and Zhukov address *P2* and target the automated generation of SDL models from HMSCs [255]. The scenarios are first sliced by actor, then communicating finite state machines are generated for each actor. These FSMs are made deterministic and minimal, and then transformed into SDL processes. The resulting SDL system usually allows more traces than those defined by the HMSCs. Very little is said about the synthesis algorithm itself, and the levels of detail and consistency required by the MSCs is relatively high. This technique is implemented in MOST, the Moscow Synthesizer Tool.

Li and Horgan target the architectural analysis of telecommunications systems with an algorithm for the semi-automated synthesis of SDL models from architectures described using component, links, and *archflows* [246]. Archflows are sequential workflows where the steps are observable events, internal events, or sending/reception of messages performed by the components (hence addressing *P2*). The resulting SDL model is complete and assumed to be valid when it contains all the archflow traces, in a way similar to a *May Pass* verdict in the LOTOS testing theory (Section 3.4.3). Workflows are assumed not to conflict with each other, hence they should be consistent and have no undesirable interaction, which is of limited use for early validation. Non-determinism is allowed, and the model can be supplemented with performance information for performance prediction evaluations. The method is supported by a toolset, the WORKFLOW-TO-SDL-DIRECT-SIMULATION.

Khendek and Vincent propose an approach for the construction of an SDL model given an existing SDL model, whose properties need to be preserved (an extension relation is provided), and a

set of new MSC scenarios [228]. The synthesis algorithm considers only input/output signals, not the actions in the transitions. The semi-automated construction is done in three steps: add new components if necessary (manually), synthesize the new architecture behaviour from MSCs using the MSC2SDL tool [1], and then merge the behaviour descriptions of the old SDL with the increment SDL, on a per process basis. If non-determinism that violates the extension relation is added along the way, then the tool reports the problem (error detection only). If an MSC description of the old SDL specification is available, then the approach can be simplified to adding new MSCs to the old MSCs and regenerate the new specification using the MSC2SDL tool. However, the extension relation may also be violated by this approach.

Turner presents an approach called CRESS (Chisel Representation Employing Systematic Specification), which defines tightly defined rules for the syntax and static semantics of an enhanced version of Chisel diagrams [354]. This improved notation has formal denotations in both LOTOS and SDL, hence enabling the synthesis of formal models in order to support the rapid creation, specification, analysis and development of features. Although CRESS often represents scenarios as trees (more precisely as directed acyclic graphs), the tree nodes represent interactions between components. Hence, this approach is roughly comparable to the ones starting from HMSCs (although CRESS' interactions are synchronous and directionless) and it also addresses *P2*. CRESS is supported by a set of tools for parsing, checking and translating diagrams. However, the synthesis algorithm remains undocumented and hence little is known about the design decisions taken by the translation tools.

Dulz *et al.* present an approach where performance prediction models (in SDL) are also automatically synthesized from MSC scenarios, but this time supplemented with performance annotations [118]. Their goal is to obtain performance estimates early in the design process (other techniques for the construction of performance models from UML and SDL are reported in [377]). The synthetic SDL model is intended to be a throw-away prototype (level *P1*), but it is nonetheless used to generate the code for the target system whose performance is evaluated. The approach is supported by a prototype tool (LISA), however the algorithm remains obscure. It is not even clear whether two MSCs that start with a similar transition should be composed as alternatives, as sequences, or in parallel.

Summary and Discussion

Several aspects of the reviewed construction approaches are summarized in Table 11:

TABLE 11. Comparison of the Selected Construction Approaches

Approach	Level	Type of Approach	Scenario Models	Construction Models	Comp?
Regnell <i>et al.</i> (UORE)	$P1^a$	Analytic	Extended MSC	Automata	Y
Eberlein (RATS)	$P2$	Analytic	Structured text	SDL	N
Bordeleau (RT-TROOP)	$P2$	Analytic	UCMs, HMSCs	ROOMCharts	N
Krüger <i>et al.</i>	$P2$	Analytic	MSCs	Statecharts	Y
Lamsweerde and Willemet	$P1$	Analytic	Sequential and synchronous MSCs	LTL properties in KAOS	Y
Yee and Woodside	$P2$	Analytic	Timed Petri Net	Timed Petri Net	N
Desharnais <i>et al.</i>	$P1$	Synthetic, non-automated	State-based relational algebra	State-based relational algebra	Y^b
Somé	$P2$	Synthetic, non-automated	Structured text, extended MSCs	Timed automata	Y
Harel and Kugler	$P2$	Synthetic, non-automated	LSCs	Statecharts	Y
Alur <i>et al.</i>	$P2$	Synthetic, non-automated	Basic MSCs	CFSMs	Y
Mäkinen and Systä (MAS)	$P2$	Synthetic, non-automated	UML sequence diagrams	UML Statecharts	Y
Koskimies <i>et al.</i> (SCED)	$P2$	Synthetic, automated	Extended MSCs	OMT state diagrams	Y
Schönberger <i>et al.</i>	$P2$	Synthetic, automated	UML collaboration diagrams	UML Statecharts	Y
Whittle and Schumann	$P2$	Synthetic, automated	UML sequence diagrams	UML Statecharts	Y
Leue <i>et al.</i>	$P2$	Synthetic, automated	HMSCs	ROOMCharts	Y
Mansurov and Zhukov	$P2$	Synthetic, automated	HMSCs	SDL	Y
Li & Horgan	$P2$	Synthetic, automated	Archflows	SDL	Y
Khendek and Vincent	$P2$	Synthetic, automated	MSCs, SDL	SDL	Y
Turner (CRESS)	$P2$	Synthetic, automated	Extended Chisel diagrams	SDL or LOTOS	Y
Dulz <i>et al.</i>	$P1$	Synthetic, automated	Extended MSCs	SDL	Y

a. The model is component-based, but mostly used as a reference model for requirements validation.

b. Interactions between a user and the system in terms of a relational algebra.

Most of the techniques surveyed here require the use of scenario notations based on messages exchanged between communicating entities (see rightmost column). MSC-like notations such as basic MSCs, extended MSCs, HMSCs, LSCs, and UML interaction diagrams are especially common as source scenario models for construction approaches. Techniques based on HMSCs can further benefit from recent theoretical results on necessary conditions for the synthesis of communicating automata from HMSCs [173]. For target construction models, communicating finite state machines, whether they are hierarchical (ROOMCHARTS, (UML) Statecharts, or OMT state diagrams) or not (SDL'96 or plain CFSMs) are very common. It is difficult to evaluate approaches for component-based scenarios as they use varying source and target models, they are still under heavy development, and they are not supported by commercial tools. Synthesis approaches also have different sets of constraints and design decisions embedded in their algorithms. Only three of the techniques surveyed (RATS, RT-TROOP, and Yee&Woodside) do not start from scenarios expressed in terms of components and messages, and they are only used in analytic construction approaches.

In SPEC-VALUE, UCMs abstract from the communication aspect between the components, although interaction with the environment could be attributed to start points and end points along UCM paths. UCMs could also be unbound, meaning that no component would be involved. Hence, most of the synthesis algorithms surveyed are of little use for the construction of formal specifications from UCMs. Furthermore, any attempt to automate the synthesis of such specification, even partially, would require further formalization of UCMs, which are currently semi-formal. In general, one can't go from the informal to the formal by formal means. An analytic approach to the construction of LOTOS specifications from UCMs therefore seems to be, at this time, the most appropriate avenue.

An interesting characteristic of these two languages is that they can both address levels *P1* and *P2*. Amyot's master thesis partially addressed level *P1* by providing several mapping rules between unbound UCMs and LOTOS [12]. In the current thesis, we extend this work to address level *P2*, where components are considered in order to produce high-level designs. Additionally, since SPEC-VALUE rests on an analytic transformation from UCMs to LOTOS, there will be much emphasis on the verification aspects required to gain a high degree of confidence in the resulting specification.

3.4 Validation and Verification

In general, *validation* refers to activities that ensure that the right product has been designed while *verification* refers to activities that ensure that the product is designed correctly. Validation often involves user requirements and scenarios whereas verification usually makes use of formal models and coarse-grain properties (e.g., absence of deadlocks). The distinction between validation and verification is at times blurred by the same techniques being applicable to both activities, and this is especially true of formal methods. The distinction is more a state of mind than mutually exclusive sets of techniques.

When constructing an initial formal specification from informal requirements, as it is the case in SPEC-VALUE, Brinksmas points out that the resulting models cannot be demonstrated by formal means, hence experimentation becomes necessary [73]. Experimental validation constitutes an essential methodological ingredient for the analysis of telecommunications systems. SPEC-VALUE intends to introduce such a validation framework in Chapter 6.

Many concepts surrounding V&V have already been presented in Section 2.1.3, with an emphasis on LOTOS techniques in Section 2.3.6 and Section 2.3.7. Concepts related to construction approaches have also been introduced in Section 3.3.4. These notions will not be repeated as such. Instead, the current section will complement these concepts with additional background on properties, general testing concepts, LOTOS testing, coverage, and testing patterns. These concepts will help defining and understanding the validation framework based on UCMs and LOTOS, which is an expected contribution of the thesis.

3.4.1 Properties

The verification of a system under design usually involves checking a formal model against another. Equivalence relations verify that the two models, which are usually represented using the same language, are equivalent under some criteria. This requires a similar level of completeness from both models. However, sometimes designers want to verify their complete formal model against partial and more manageable models that we call *properties*. Properties are often described in a language differ-

ent from the model being verified. This section presents some of the main concepts and results surrounding properties and their use in a LOTOS-based approach.

Classifying Properties

In reactive systems, and especially in telecommunications systems, properties can be classified in three categories [316]:

- **Safety properties**: something bad never happens. In the realm of protocol engineering, these properties ensure the absence of deadlocks, unspecified reception errors, buffer or channel overflow, and other errors in the system.
- **Liveness properties**: something good will eventually happen, i.e. the system performs its intended functions.
- **Responsiveness properties**: the system respects the response time requirements (timeliness, performance) and it has the possibility of recovering in the case of transient failures (robustness and fault-tolerance).

Using different techniques, these properties can be usually guaranteed by verifying the absence of syntactic and semantic design errors [293]:

- **Syntactical or logical design errors**: are related to the logical structure of the exchange of messages among entities. These errors are usually independent of the service or functionality: deadlocks, unspecified receptions, instabilities, livelocks, overspecification, and channel overflow. The absence of such design errors often guarantees safety properties.
- **Semantic design errors**: are related to the functionalities to be provided by the system. Such errors are manifested by the abnormal functioning of the system and its inability to meet its intended purpose. These errors usually cause liveness and responsiveness properties to be unsatisfied.

To some extent, verification is more concerned with the detection of syntactical errors (in safety properties) whereas validation focuses more on the detection of semantic errors (in liveness and

responsiveness properties). Note however that validation can be performed by “verifying” liveness and responsiveness properties! Again, both concepts overlap on many occasions, and there is no clear-cut separation.

Properties and LOTOS

In the LOTOS world, verification is usually achieved through techniques such as theorem proving, reachability analysis, model checking, equivalence checking, and testing. Theorem proving handles systems with an infinite number of states, but it usually cannot be completely automated and hence requires human assistance. Reachability analysis and model checking, which are based on models or state exploration, require a finite number of states, but they are fully automated. Recent research on symbolic model checking [327] and on-the-fly model checking [135] provides some relief for large or infinite state spaces, but available tools are still limited and they impose many constraints and simplifying assumptions on the models. Equivalences can also be used to verify properties. Chehaibar *et al.* [93] express their properties as graphs (Finite State Machines) that are checked, through branch equivalence and bisimulation equivalence, against the specification. However, even this approach needs a finite representation of the specification, which can hardly be generated from the complex and dynamic telecommunications systems on which the thesis focuses.

Properties expressed in temporal logic (for model checking) are usually large-grained whereas test cases can be considered as small-grained properties, because the latter are more constrained and they usually cover fewer states in the model. However, temporal logic properties are often more difficult to create and to use than test cases, which are more linear. Tools recently started to address this issue by providing graphical means of developing temporal logic properties [331]. In any case, measuring the completeness and consistency of a set of properties remains a complex issue [135].

Conformance relations (conf) and canonical testers ($CT(S)$), as introduced in Section 2.3.6, mainly target the verification of liveness properties. Whereas $CT(S)$ verifies all liveness properties at once, test cases verify small-grain properties. Test cases that are reductions of the canonical tester of a specification (T_x such that $T_x \underline{\text{red}} CT(S)$) are called *acceptance* tests. Their counterpart, called *rejec-*

tion tests in this document, are expected to be rejected by the implementation. Essentially, rejection test cases are used to check small-grained safety properties. They can also help cope with a weakness of the conf relation mentioned at the end of Section 2.3.6. For instance, in Figure 13 on page 40, stating that an implementation should refuse the sequence <Dial, Dial> would show that $S6$ is not valid with respect to its specification S . The LOTOS theory does not address the derivation of rejection test cases because many arbitrary decisions can be taken during their creation. Responsiveness properties are also difficult to verify in LOTOS because this language lacks quantitative notions of time and probabilities. However, robustness and fault-tolerance can be checked to some extent through the use of temporal logic properties and tests corresponding to exceptional scenarios.

We use testing as the main validation technique in this thesis. For complex and realistic telecommunications systems, testing is simpler, more pragmatic, and better supported along the whole design process than any other technique discussed so far. Moreover, even the most formal verifier admits that a formally verified system should still be tested (who verified the compiler? and the operating system? and who verified the verifier? and so on) [347]. The next section focuses on general testing concepts, followed by a more detailed presentation of LOTOS testing.

3.4.2 General Testing Concepts

The main goal of testing is to detect errors. Research and experience have shown that the cost of finding an error gets much higher the closer we get to the implementation [267][290][287][335]. Therefore, testing should be used as soon as possible, even at the specification level. A good test case is a test that highlights a fault in the specification. A good *test suite* is a set of test cases that covers, under some hypotheses and assumptions, critical aspects, if not all aspects, of a specification.

This section briefly covers general testing concepts such as test selection, test hypotheses, testability, conformance testing, test suites and test architectures.

Test Selection and Hypotheses

When testing complex systems, one of the main problems faced by engineers is the selection of an appropriate test suite. Tretmans suggests four approaches that facilitate this selection [346]:

- Select goals for individual tests.
- Weaken the specification, which then allows more correct implementations and requires fewer properties to check.
- Weaken the implementation relation. For instance, conf, being weaker than te, will allow more correct implementation and will require fewer properties to check than te. This approach is seldom used because the implementation relation is often selected before the specification is created (as it can influence the way the specification is built).
- Improve the test hypotheses.

Phalippou has discussed the last option extensively, especially in the context of synchronous testing for a class of input/output finite state machines (IOFSMs) [284]. He defines several test hypotheses and their impact:

- **Regularity:** the number of next states is limited for each state in the implementation. This allows for infinite test cases to be reduced to finite test cases.
- **Independence:** the actions or functions are projected to independent sets. This enables a divide-and-conquer approach to testing.
- **Uniformity:** the value domain are partitioned (e.g. according to some congruence rule), and then one test for each partition is used.
- **Fairness:** the (non-deterministic) behaviour of the implementation can be covered in a finite and computable number of attempts.

Also, many test selection strategies imply a *reset hypothesis*, which requires the implementation to possess a working reset feature to be used before each test case.

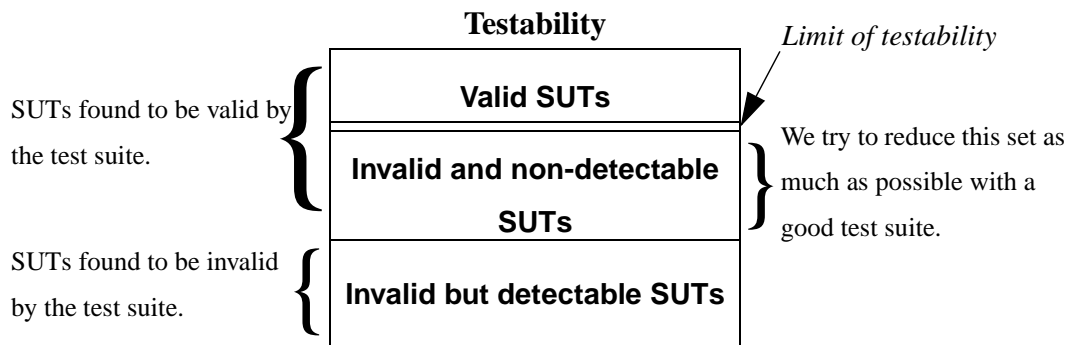
Rather than working on hypotheses or relations, the thesis focuses on Tretmans' first option and use UCMs to define appropriate goals which will lead to the generation of test cases.

Testability

Validation test cases can be derived according to many strategies. What is most desirable however is a test suite that will detect invalid SUTs with the most success and the least cost. Figure 15, inspired from Drira and Azéma [116], illustrates one important test selection goal. In this diagram, the notion of *detectability* means that a test suite detects the invalidity of a specification with respect to the requirements. For example, Section 2.3.6 states that two LOTOS test cases have the same detectability if they are testing equivalent. *Testability* exposes some limits caused by constraints on the accessibility, observability, and controllability of the SUT, and of the automatability of the testing process. Other limits also relate to the fact that the behaviour may be infinite. In a recent paper, Baumgarten and Wiland discuss many definitions of testability and provide an interesting framework where qualitative notions of testability can be evaluated [43].

LOTOS specifications are highly testable (in opposition to conventional software) and those constraints are much weaker for LOTOS than for real implementations, but they are nonetheless present.

FIGURE 15. Limit of Testability



There is a limit of testability beyond which invalid SUTs are not detected by a finite (and incomplete) test suite. This set of invalid SUTs has to be reduced as much as possible. The test case derivation and selection strategy has a direct impact on the size of this set. Of course, a good strategy leads to a good detectability and to a lower testability limit, but also to higher costs of derivation and/or execution.

Conformance Testing

Specification-based testing has been used with different specification languages over the last few decades (for instance, see [10][88][213][309]). In the context of (formal) *conformance testing*, a specification-based technique verifying that the implementation under test conforms to its specification by attempting to detect conformance errors, most methods assume that both the specification and the implementation can be modelled in the same (formal) language. According to the *Formal Methods in Conformance Testing* (FMCT) framework [196], a test suite can be:

- **Exhaustive:** all passing implementations are compliant to the specification.
- **Sound:** all implementations that do not pass are not compliant.
- **Complete:** the test suite is both sound and exhaustive,

The validation context proposed in this thesis is different from traditional conformance testing. In particular, the term specification can be misleading as our LOTOS specifications really are high-level design prototypes. Therefore, in the context of SPEC-VALUE, the term “specification” is replaced by “*requirements*”, and “implementation” becomes “*specification under test* (SUT)”. To validate the SUT, we plan to use functional (black-box) test cases derived from user requirements captured as UCMs. Conformance test suites are usually abstract and they target artificial coverage criteria in terms of another previously defined model. Nevertheless, many ideas and techniques developed for conformance testing can be applied in our specific validation context.

If a test suite is neither sound nor exhaustive, then nothing concerning conformance or validity can be concluded by means of testing. Pragmatically, it is almost never possible to construct a finite exhaustive test suite for real-life systems. Consequently, test suites are usually sound, but still incomplete. Any error detected by a sound test suite proves that the SUT is incorrect, but not finding an error does not mean that the SUT is without errors. Optimizations of such test suites target the minimization of the number of test cases and their complexity/length/cost, and the maximization of the

discriminatory power of the tests. A test suite TS_1 is said to discriminate more than another one (TS_2) if TS_1 finds faults in more specifications than TS_2 .

Note that in the current practice, conformance does not imply *interoperability* and interoperability does not imply conformance. Two systems are interoperable if they can communicate and work together to achieve a common goal. Two different implementations may conform to the same requirements or standards and yet they might not be completely interoperable. Interoperability testing is more costly than conformance testing. The cost of conformance testing increases linearly with the number of products to test (each one is tested against the specification) whereas the cost of interoperability testing increases with the number of possible combinations of these products, which leads to many more configurations to check. However, if the specification (or standard) is formal enough, then conformance could potentially imply some level of interoperability. This is another motivation for the creation of formal specifications from requirements.

Test Suites and Test Architecture

The *Conformance Testing Methodology and Framework* (CTMF) [193] details the definition of an *abstract test suite* as being composed of *test groups*. Each group consists of several *test cases* according to a logical ordering of execution. A test case contains *test steps*, each of which consists of several *test events*, i.e. the atomic interactions between the tester and the implementation or SUT. Test steps can be shared by many test cases.

A test case is often composed of several parts:

- **Test purpose:** describes the objective of the test case (expected behaviour, verification goal, etc.).
- **Test preamble:** contains the necessary steps to bring the SUT into the desired starting state.
- **Test body:** defines the test steps needed to achieve the test purpose.
- **Test postamble:** used to put the SUT into a stable state after a test body is executed.

Conformance test cases for finite state machines usually have a test body that contains one transition followed by a **test verification** step (checking sequence, unique input/output, distinguishing sequence, etc.), which identifies the target state [248][360]. A preamble may also contain a verification sequence that checks the initial state. However, in many test suites, the initial state resulting from the preamble has already been checked as a target state in a previous test case. Often test cases are designed to be mutually execution-independent. Test cases can also be described in TTCN-3 [209], the latest ITU-T standard notation for the specification of abstract and retargetable tests.

Note that CTMF also defines multiple test architectures (local, distributed, coordinated, and remote methods) but they will not be used in the thesis. The SPEC-VALUE approach focuses on high-level functional testing at the specification level. Therefore, the only test architecture that is intended to be used is based on LOTOS synchronous testing between the tester and the specification, where the points of control and observation (PCO) are represented as observable LOTOS gates.

3.4.3 LOTOS Testing

LOTOS exhibits interesting static semantics features that are supported by many tools. The successful compilation of a LOTOS specification ensures that several data-flow anomalies, such as the use of an undefined or unassigned value identifier (variable), cannot occur. Since many of these problems are automatically avoided or can be detected using existing techniques [323], they will not receive much attention in the thesis.

Dynamic behaviour, however, is a totally different story. This is where testing can help. Section 2.3.6 already provided an overview of basic concepts of the LOTOS testing theory (testing equivalence, conformance relation, canonical testers, and tests cases). This section provides additional definitions for the concepts of test suites and verdicts.

Test Suites and Relations

The LOTOS testing theory has a test assumption stating that the implementation (the SUT in our case), modelled as a LTS, communicates in a symmetric and synchronous way with external observers, the

test processes. There is no notion of initiative of actions, and no direction can be associated to a communication.

A correct test case is a reduction of the specification's canonical tester ($T_x \text{ red } CT(S)$). To verify the successful execution of a test case, such a test process T_x and the specification under test SUT are composed in parallel, synchronizing on all gates but one (a *Success* event, added at the end of each test case). If the composed behaviour expression deadlock occurs prematurely, i.e. if *Success* is not always reached at the end of each branch of the LTS resulting from this composition, then the SUT fails this test. If this is not the case, then it must have passed the test¹.

Table 12 and Table 13 present formal definitions of notations and relations that will help characterizing the (un)successful execution of test cases in LOTOS. Many of these definitions are inspired from previous work by Hennessy and De Nicola [113][174] and Brinksma *et al.* [70], and from the FMCT framework [196]. They are used mostly in Chapter 6 where a theory for the derivation of validation test cases from UCMs is proposed.

In Table 13, the relations are not only defined for individual test cases, but also for entire test suites. Although the same names are used, their signatures (domains) are different.

1. It has passed unless the SUT exhibits divergent behaviour, such as an infinite loop of internal events, a case that is outside the scope of this thesis.

TABLE 12. Notation for Test Definitions

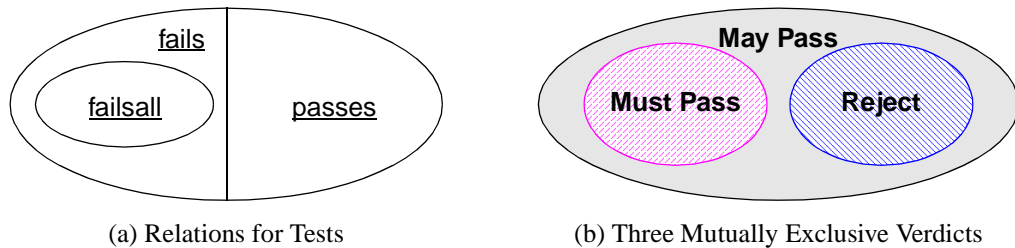
<i>Notations</i>	<i>Definitions</i>
SPECS	Universe of all possible behaviour expressions.
$S, S1, S2...$	Specifications. $S \in \mathbf{SPECS}, S1 \in \mathbf{SPECS}, S2 \in \mathbf{SPECS}, \dots$
SUT	Specification Under Test. $SUT \in \mathbf{SPECS}$.
TESTS	Universe of test cases. In LOTOS, tests are also behaviour expressions (processes): TESTS = SPECS .
$CT(S)$	Canonical tester of specification S . $CT(S) \in \mathbf{TESTS}$.
TS	Test suite (set of test cases) for testing specification SUT . $TS \subseteq \mathbf{TESTS}$.
TG_n	Test group n . The test suite contains all test groups. $TS = \bigcup_{g=1}^n TG_g \wedge TG_g \subseteq \mathbf{TESTS}$.
T_x	Test case x , which belongs to a test group. $\forall T_x, \exists TG_g \mid T_x \in TG_g \wedge TG_g \subseteq TS$.
$ACCEPT(TS)$	Set of acceptance test cases found in TS (Must tests). $ACCEPT(TS) \subseteq TS$.
$REJECT(TS)$	Set of rejection test cases found in TS (Reject tests). $REJECT(TS) \subseteq TS$.

TABLE 13. Passes, Fails, and Failsall Relations

<i>Relation</i>	<i>Definitions</i>
$SUT \underline{\text{passes}} T_x$	Pass relation for one test case: $\underline{\text{passes}} \subseteq \mathbf{SPECS} \times \mathbf{TESTS}$. $SUT \underline{\text{passes}} T_x \Leftrightarrow \forall t \in Tr(SUT \parallel [\text{all gates but } Success] \parallel T_x), t \text{ reaches } Success$.
$SUT \underline{\text{passes}} TS$	Pass relation for a test suite: $\underline{\text{passes}} \subseteq \mathbf{SPECS} \times \text{PowerSet}(\mathbf{TESTS})$. $SUT \underline{\text{passes}} TS \Leftrightarrow \forall T_x \in TS, SUT \underline{\text{passes}} T_x$.
$SUT \underline{\text{fails}} T_x$	Failure relation for one test case: $\underline{\text{fails}} \subseteq \mathbf{SPECS} \times \mathbf{TESTS}$. $SUT \underline{\text{fails}} T_x \Leftrightarrow \neg(SUT \underline{\text{passes}} T_x)$
$SUT \underline{\text{fails}} TS$	Failure relation for a test suite: $\underline{\text{fails}} \subseteq \mathbf{SPECS} \times \text{PowerSet}(\mathbf{TESTS})$. $SUT \underline{\text{fails}} TS \Leftrightarrow \neg(SUT \underline{\text{passes}} TS) \Leftrightarrow \exists T_x \in TS, SUT \underline{\text{fails}} T_x$.
$SUT \underline{\text{failsall}} T_x$	Failure relation for one test case: $\underline{\text{failsall}} \subseteq \mathbf{SPECS} \times \mathbf{TESTS}$. $SUT \underline{\text{failsall}} T_x \Leftrightarrow \forall t \in Tr(SUT \parallel [\text{all gates but } Success] \parallel T_x), t \text{ does not reach } Success$.
$SUT \underline{\text{failsall}} TS$	Failure relation for a test suite: $\underline{\text{failsall}} \subseteq \mathbf{SPECS} \times \text{PowerSet}(\mathbf{TESTS})$. $SUT \underline{\text{failsall}} TS \Leftrightarrow \forall T_x \in TS, SUT \underline{\text{failsall}} T_x$.

The difference between fails and failsall is that the *Success* event is never reached in failsall, while it may be so for some test runs in fails as long as at least one test run leads to a premature deadlock (or to an infinite loop). Hence, fails is implied by failsall (see Figure 16(a)).

FIGURE 16. Relations and Verdicts for Tests



Verdicts and Types of Tests

This testing theory is supported by the tool LOLA [301], which expands the composition of a test and a specification to determine whether the executions reach the *Success* event or not. Three *verdicts* can occur after the execution of one test case (Figure 16(b)):

- **Must pass:** all the possible executions (called *test runs*) were successful (they reached the *Success* event for every trace). Formally: $SUT \text{ passes } T_x \Rightarrow \text{Must pass}$.
- **May pass:** some executions were successful, some unsuccessful (or inconclusive according to a depth limit). Formally: $\neg(SUT \text{ passes } T_x) \wedge \neg(SUT \text{ failsall } T_x) \Rightarrow \text{May pass}$.
- **Reject:** all executions failed (they deadlocked prematurely or were inconclusive). Formally: $SUT \text{ failsall } T_x \Rightarrow \text{Reject}$.

With real implementations, test cases often must be executed more than once in the presence of non-determinism in either the test or the implementation (under some fairness assumption). Deterministic testing, as defined by Calder *et al.*, addresses this issue to some extent for concrete distributed programs [86][87]. At the specification level, LOLA avoids this problem altogether because it determines the response of a specification to a test by a complete state exploration of their composi-

tion [280]. For tests that do not contain **exit**, LOLA uses the composition on the left, whereas the composition on the right is for tests that do contain **exit**:

$$\begin{array}{ll}
 SUT[\{EventSUT\}] & (SUT[\{EventSUT\}] \\
 |[\{EventSUT\} \cup \{EventTx\}]| & |[\{EventSUT\} \cup \{EventTx\}]| \\
 T_x[\{EventTx\} \cup \{Success\}] & T_x[\{EventTx\} \cup \{Success\}] \\
 &) \gg Success; \mathbf{stop}
 \end{array}$$

LOLA analyzes all the test terminations for all possible evolutions (the test runs). The successful termination of a test run consists in reaching a state where the termination event (*Success*) is offered. A test run does not terminate if a deadlock or internal livelock¹ is reached.

The LOTOS theory differentiates three types of intent for tests submitted to a SUT:

- **Must test:** T_x is a “must test” of SUT if it is intended to terminate for every test run when applied to SUT (SUT passes T_x). A “must test” corresponds to a mandatory scenario, and the expected verdict is a Must pass.
- **May test:** T_x is a “may test” of SUT if it is intended to terminate for at least one test run when applied to SUT (\exists trace in SUT $[[$ all gates but $Success]$ T_x that leads to a $Success$). A “may test” corresponds to an optional scenario, and the expected verdict is either a May pass or a Must pass.
- **Reject test:** T_x is a “reject test” of SUT if it is intended not to terminate successfully for any test run when applied to SUT (SUT failsall T_x). A “reject test” corresponds to a forbidden scenario, and the expected verdict is a Reject.

These types relate to what we call *acceptance/rejection testing*. An acceptance test is a “must test” in the set $ACCEPT$ which checks that a functionality is present or that an expected result is effectively output. A failure in that case is seen as catastrophic, because the underlying liveness property is

1. There is no notion of fairness in this theory. Whenever there is a loop of internal events (τ -loop) which is not under the control of the test process, then the test run has to be truncated. We try to avoid these loops as much as possible in our specifications. Although some theories and simplifications (through weak bisimulation) exist, they are not implemented in LOLA.

violated. A rejection test (a “reject test” in *REJECT*) checks that the SUT rejects one or many events after a given sequence of events. A success in that case is catastrophic because the underlying safety property is violated. For a given test suite *TS*, *REJECT(TS)* and *ACCEPT(TS)* represent partitions, i.e. they are mutually exclusive ($REJECT(TS) \cap ACCEPT(TS) = \emptyset$) and together they constitute the whole test suite ($REJECT(TS) \cup ACCEPT(TS) = TS$). Rejection test cases can lower the testability limit of test suites conventionally composed solely of acceptance test cases. In Section 6.2.2, we will see that the test type (acceptance or rejection) is, together with the test goal, part of our definition of test purpose.

“May tests” will not be used in the SPEC-VALUE approach as the interpretation of the May pass verdict, composed of successful and unsuccessful traces, usually requires human intervention. Although canonical testers can be reduced to sets of deterministic test cases [69], if a SUT happens to be non-deterministic (i.e. for a same input event the SUT may offer different resulting events on different occasions), then an acceptance test could also result in a May pass verdict. In this case, the test case has to be augmented with the necessary alternatives (present in the canonical tester) so that it results in a Must pass verdict. As a result, the test would no longer be a sequence of events but a tree of events.

Evolution Towards Input/Output LTSs

Brinksma and Tretmans surveyed many extensions to LTS with applications to test frameworks, formal test generation (with tools), and asynchronous test contexts [72]. Several of these extensions have been used as enhanced semantic models for LOTOS. In particular, Tretmans partitions the actions on a LTS into inputs and outputs [347]. This enables the application of the conformance relation *ioco*, which is more appropriate for testing real implementations than a directionless conformance relation like *conf*. The relation *ioco* also helps to alleviate the need for a category of rejection tests based on non-deterministic outputs of data on a gate, which is otherwise required when *conf* is used in a validation context. This concept is extended by Heerink to multiple channels in the *mioco* relation [171]. Although such enhanced semantics is attractive from a conformance testing perspective, where it has been used so far [117], it does not seem to apply directly to the validation of a LOTOS specification, which is the focus of SPEC-VALUE. Moreover, current tool support targets the automated generation

and execution of test cases from LOTOS specification, but nothing is available for the testing of the specification itself. Again, a major assumption behind these techniques is that the formal specification is correct and valid with respect to the requirements. Chapter 6 proposes scenario-based techniques that help increasing the level of confidence in such a specification.

3.4.4 Coverage

“When to stop testing?” is and will remain an important problem for communications software validation and verification. Communications software is often tested until the probability of failure is believed to be small, or until the deadline for the product release is reached (whichever comes first) [328]. Statistical models can also be helpful [265]. Lai [237] mentions that knowing how much of the application source code has been covered by a test suite can help estimate the risk of releasing the software product to users, and discover new tests necessary to achieve a better coverage. Inexperienced testers tend to execute down the same path of a program, which is not an efficient testing technique.

Coverage measures are considered to be a key element in deciding when to stop testing. Coverage analysis of code is a common approach to measure the quality and the adequacy of a test suite [371][386]. Coverage criteria can guide the selection of test cases (*a priori*, i.e. before the execution of the tests) and be used as metrics for measuring the quality of an existing test suite (*a posteriori*, i.e. after the execution of the tests). Many methods are available, and several criteria are well established [92]:

- **Statement coverage:** checks which statements or operations are executed. Also called structural coverage.
- **Branch coverage:** checks whether all possible outcomes of a branch are executed. This is particularly relevant to structured programming languages.
- **Data-flow coverage:** measure of executing paths between creations, modifications, and uses of data values.
- **Path coverage:** checks the execution of syntactically- or semantically-defined paths.

- **Mutation adequacy:** checks whether the tests kill all non-equivalent mutants of a program. Mutation testing is a white-box method for creating test cases which are sensitive to small syntactic changes to the structure of a program or of a specification. If a test can distinguish a valid program from an invalid variation (a *mutant*, which is the valid program/specification plus one modification to one operator or construct done according to a fault model), then this is a good test and should be part of the test suite [112][292]. If a test suite cannot detect an invalid mutant, then it needs to be augmented with a suitable test case.

This thesis covers a different angle of the same question, relating to specification coverage. Specifications, just like programs, can be covered for several reasons and according to several criteria [10]. For example, we want to cover a specification in the generation of conformance test cases for an implementation, or in order to check whether a specification satisfies abstract requirements. These processes can also gain in quality from the use of coverage measurements. Many formal specification languages already benefit from tool-supported coverage metrics, including SDL with Telelogic's *Tau* [342], which measures the coverage of symbols like states and transitions, and VDM with IFAD's *VDMTools* [190]. Such tools have started to appear for design modelling languages as well, e.g. for UML collaboration diagrams [2]. Even hardware description languages now benefit from coverage analysis. For instance, Joyce uses probe-based instrumentation of Verilog descriptions for measuring the coverage of a test suite at simulation time [221]. Unfortunately, no such tools are currently available for LOTOS.

Still, several coverage criteria have been defined for LOTOS specifications. For instance, van der Schoot and Ural developed a technique for static data-flow analysis [323], Carver and Tai defined a sequencing constraint coverage criterion [86], and Cheung and Ren proposed an operational coverage criterion [94]. These three techniques are used mostly for guiding, *a priori*, the generation of test cases from the specification. The first one is based on data usage, the second on the satisfaction and non-satisfaction of constraints, and the third one is based on the semantics of LOTOS operators.

Instrumentation

A posteriori metrics of coverage often require the code to be instrumented in order for relevant data to be collected and coverage results to be computed. *Probe insertion* is a well-known white-box technique for monitoring software in order to identify portions of code that has not been yet exercised, or to collect information for performance analysis. A program is instrumented with probes (generally counters) without any modification of its functionality. When executed, test cases trigger these probes, and counters are incremented accordingly. Probes that have not been “visited” indicate that part of the code is not reachable with the tests in consideration. Obvious reasons include that the test suite is incomplete, that the implementation is not deterministic enough, or that this part of the code is reachable under no circumstance.

For well-delimited programs, Probert suggests a technique for inserting the minimal number of statement probes necessary to cover all branches [291]. Minimizing the number of probes is important because instrumentation usually has an impact on the speed of test execution. This idea will be adapted to LOTOS in Chapter 7 in order to measure the structural coverage of specifications by a test suite. UCM path coverage will be used in Chapter 6 as an *a priori* test selection criteria, with the assumption that this corresponds to the coverage of functional requirements.

3.4.5 Testing Patterns

This section provides a short overview of patterns in general, with an emphasis on design and testing patterns. Section 6.3 intends to develop testing patterns for the selection of test cases from requirements and high-level designs based on the coverage of UCM paths.

Patterns

Nearly a decade ago, patterns have emerged from the object-oriented community as a new software engineering problem-solving discipline. Although multiple definitions exist [101], a *pattern* remains essentially a proven and reusable solution to a recurring problem in a specific context. It also describes the relevant forces, which may be present in varying degrees in a context, as well as relations among them. A pattern explains insights that have led to generally recognized good practices.

Patterns have roots in many disciplines, most notably in Alexander's work on urban planning and building architecture [5][6]. Alexander used patterns to describe what he called a “quality without a name” in architectural solutions, where patterns focus on good design culture and on documentation rather than on technology.

Software patterns truly became popular with the publication of a design pattern book written by Gamma *et al.* [144]. Software patterns can be defined at several levels, including (from the more general to the more detailed):

- **Process patterns:** express problems and solutions at a methodological level.
- **Architectural patterns:** express a fundamental structural organization for software systems.
- **Design patterns:** provide a scheme for refining the subsystems or components of a software system, or the relationships between them.
- **Idioms:** low-level patterns specific to a programming language.

Content of a Pattern

Software patterns can be described according to different formats or *templates* [6][51][81][144]. However, most patterns contain five core elements [240][256][361]:

- **Name:** A short familiar, descriptive name or phrase, usually more indicative of the solution than of the problem or context.
- **Problem:** The challenge to be addressed.
- **Context:** The situations under which the pattern applies. Often includes background, requirements, and discussions of why this pattern exists.
- **Forces:** A description of the relevant factors, constraints and compromises that contribute to the problem and/or its solution. The interactions between the forces may also be included.

- **Solution:** how to address the problem in order to balance forces and to construct solution artifacts. Solutions often include several variants and/or ways to adjust to circumstances.

Additional elements can also be found, including rationales, resulting contexts, consequences, examples, related patterns, and known uses.

Patterns can be regrouped into *pattern catalogs* [81], which are collections of related patterns divided into categories, or as *pattern languages* [299], which are collections of patterns that work together to solve problems in a specific domain. In a pattern language, a resulting context of one pattern becomes the context of its successor patterns.

Design Patterns and Testing Patterns

Patterns can be used for designing systems, which is their traditional use, but also for testing them. *Testing patterns* can provide established solutions for designing tests or for supporting the testing process. This section gives a brief review of existing pattern-oriented work relevant to the areas of interest to this thesis, i.e. telecommunications systems, scenarios, and testing.

In the telecommunications area, most patterns that currently exist target the design level. Rising collected many recent such patterns in her book [310]. Adams *et al.* focus on fault-tolerance systems [3], whereas Utas proposes a pattern language for handling and avoiding undesirable interactions between telephony features [361]. Andrade *et al.* recently presented a pattern language for mobility management adapted to second generation wireless communication systems [34][35]. In the area of scenarios, Buhr used UCMs for describing and understanding macroscopic behaviour patterns in object-oriented frameworks [80]. Jacobson suggested abstract use cases as a rigorous expression of the problem part of a pattern [139]. Bordeleau proposed scenario composition patterns for the construction of hierarchical finite state machines from UCMs [62]. Mussbacher and Amyot defined several UCM patterns for describing functionalities of complex reactive and distributed systems according to various styles [266]. In the area of testing, DeLano and Rising created a pattern language, mostly at the process level, for testing large software systems [111]. Binder provides a com-

prehensive set of patterns for testing object-oriented systems in a very recent book [51]. Other authors, notably Beizer [44] and Siegel [329], propose testing solutions in the general form of patterns, although they are not called patterns explicitly in these publications.

The most interesting patterns however cover many of the areas of interest at the same time. For instance, Andrade uses UCMs to describe requirements, *design* and analysis patterns for mobile wireless communications systems [32][33][35]. These generic patterns express functions that are common to many existing mobile systems and can be used in the early steps of design. Although the design pattern community is not used to seeing patterns described with scenarios (in UCMs or any other form), this work shows much promise. Mussbacher and Amyot also illustrated their patterns for UCMs using various telecommunications systems [266]. In his collection of patterns, Binder suggests the use of *test design* patterns for UML-based scenarios, in the context of OO systems testing [50][51]. Some of them are related to the patterns developed in Section 6.3.

There is a real interest in patterns from the software community, and the need for testing patterns adapted to telecommunications and reactive systems is still crying. We see in this an opportunity to provide testing patterns adapted to telecommunications systems and integrated to SPEC-VALUE as a means to develop suitable validation test suites from requirements and designs expressed with UCM scenarios. Testing patterns can be seen as a semi-formal way of selecting test cases, and they represent a good match for a semi-formal notation like UCMs.

3.4.6 Summary and Discussion

This section complements many validation and verification concepts introduced in Chapter 2. It provides additional background on some relevant work about properties, general testing concepts, LOTOS testing, coverage, and testing patterns.

Among the conclusions, we observe that testing is one of the most pragmatic approaches for validating and verifying complex telecommunications systems, even at the specification level. Acceptance test cases, which are reductions of canonical testers in LOTOS, can describe fine-grained liveness properties and be used to assess conformance. The need for rejection test cases, which are

discussed superficially in the LOTOS theory, is emphasized. Rejection tests can describe fine-grained safety properties, lead to improved validity relations when used jointly with acceptance tests, and lower the limit of testability. In the context of LOTOS testing, this section also defines notations and relations (inspired from the literature) formalizing verdicts and types of tests.

Test selection is and will remain a major V&V issue. In this thesis, test hypotheses are not intended to guide the test selection process. Instead, coverage-based criteria based on UCM paths, i.e. testing patterns, will be used to derive test goals. Additional coverage metrics, based on the structural coverage of LOTOS specifications, will be measured *a posteriori*. Such metrics are still lacking in the literature, but there is an opportunity to define an approach based on probe insertion to tackle this issue.

3.5 Chapter Summary

This chapter reviews existing work and concepts in four areas of interest to SPEC-VALUE. Section 3.1 addresses many issues related to the concept of causality. It explains how causality can be beneficial when describing concurrent systems. A discussion on concurrency models for interleaving and causal semantics follows, and different families are briefly introduced and classified. Although UCMs and LOTOS are able to capture causality at a syntactic level, the LOTOS semantic model, based on LTSs, does not capture causality as such. Nevertheless, LTSs are still used in SPEC-VALUE because they offer simple verification algorithms and good tool support for validating system specifications.

Section 3.2 focuses on specification techniques and emphasizes the similarities and differences between Use Case Maps, LOTOS, and four other techniques (MSCs, SDL, Petri Nets, and UML). These techniques are compared against thirteen criteria regrouped under four categories (usability, V&V, tool support, and training). UCMs appear particularly suitable for the representation of functional requirements and of high-level designs. LOTOS complements most of UCMs' weak areas related to the analysis of systems. Both languages also make use of similar constructs and support system descriptions with and without components. These characteristics, introduced in Table 1 on page 6, facilitate the mapping of UCMs to LOTOS, as suggested in SPEC-VALUE.

Scenarios are the main topic of Section 3.3. Their benefits and drawbacks are presented in general terms. Then, thirteen scenario notations relevant to telecommunications systems are compared against eight evaluation criteria. Again, the UCM notation proves to be an interesting alternative because it is not component-centered, it supports causality, and it supports dynamicity, three features that help describing telecommunication systems requirements and early designs.

Section 3.3.4 covers construction approaches, where individual scenarios (closer to the requirements) are integrated to form a composite view of all scenarios (closer to the high-level design). Benefits and drawbacks of analytic and synthetic (interactive and automated) approaches are presented. Then, twenty construction approaches are introduced and briefly compared. Most of the synthetic approaches require the presence of messages or interactions between components. Since UCMs abstract from this kind of communication, the existing algorithms for the synthesis of models from scenarios are of little use. In the context of SPEC-VALUE, an analytic approach (manual transformation followed by a verification step) appears to be more appropriate. This will be one of the main topic discussed in Chapters 4 and 5.

As for validation and verification, Section 3.4 presents several necessary concepts related to properties, testing in general, LOTOS testing, coverage, and testing patterns. The validation problem addressed by SPEC-VALUE is different from that of conventional conformance testing because the latter checks a (formal) implementation against a formal specification whereas SPEC-VALUE suggests the validation of a first formal specification against informal requirements and semi-formal UCMs. The LOTOS testing theory can still be used, but several needs are identified in order to adapt it to the context of SPEC-VALUE. These needs include the use of rejection test cases, the definition of a validation relation more discriminative than *conf*, the selection of tests using UCM-based testing patterns, and the need to measure the coverage of LOTOS specifications by a test suite. These issues are intended to be addressed in Chapter 6 and Chapter 7.

This literature review shows that UCMs and LOTOS represent a good match with much potential in an approach like SPEC-VALUE. The remaining chapters of this thesis will discuss and illustrate how this potential can be exploited.

Contributions

The following items are original contributions of this chapter:

- Evaluation of six specification techniques.
- Evaluation of thirteen scenario notations.
- Survey and brief comparison of twenty analytic and synthetic construction approaches.
- Argument showing that UCMs and LOTOS are compatible and complementary techniques.

CHAPTER 4

From Requirements to UCMs in SPEC-VALUE

Right now it's only a notion but I think I can get money to make it into a concept and then later change it into an idea.

Woody Allen (Annie Hall, 1977)

This chapter presents the first steps of the SPEC-VALUE methodology, which was introduced in Chapter 1, together with a recapitulation of the main motivations behind the existence of the approach. These first steps are illustrated in Section 4.3 with an ongoing example that will be developed throughout the thesis. This example is the *Tiny Telephone System* (TTS), for which the informal requirements, the structure and the UCM scenarios are provided.

4.1 Return on the SPEC-VALUE Methodology

The *Specification-Validation Approach with LOTOS and UCMs* (SPEC-VALUE) aims to produce validated and executable specifications of system requirements and high-level designs, together with validated functional test cases and documentation. One of the main assumptions behind SPEC-VALUE is that the system functionalities to be designed can be described in terms of Use Case Maps. This is usually true of reactive, concurrent, and distributed systems, which focus on behaviour. For detailed sequential systems that focus on input/output functions (e.g., a sorting algorithm), other notations and approaches are more appropriate. The introduction of a semi-formal representation of system functionalities is in line with the second level of the Formal Specifications Maturity model scale, and hence improves upon the sole use of formal languages (Section 9.1.3).

As explained in the literature review, Use Case Maps and LOTOS complement each other in many ways, the gap between the two notations is small, and a translation from UCMs to LOTOS appears straightforward (Table 1 and Section 3.2). UCMs are close to the functional requirements, which represent the starting point of SPEC-VALUE, they handle causality, dynamicity, and optional component structures, and they help reasoning about requirements without having to commit to details that belong to a lower level of abstraction. LOTOS specifications, which are executable prototypes, can formalize requirements and high-level design in terms of abstract sequences of events and, when appropriate, component-based behaviour. LOTOS enables formal analysis and early validation of the UCMs and the requirements, as well as the generation of functional test cases that are reusable down the road towards the implementation (Section 3.4).

SPEC-VALUE uses an analytic approach for the construction of prototypes from scenarios. According to the analysis in Section 3.3.4, the differences in the levels of looseness, completeness and details between UCMs and LOTOS suggest that analytic approaches (which are manual and require a verification step) are more appropriate than synthetic approaches (which are automatable but require strict, formal, and often restrictive semantics), especially for complex systems.

4.1.1 SPEC-VALUE and Software Development Process Models

SPEC-VALUE, whose steps are recalled in Figure 17, is more limited in scope than traditional software development process models (waterfall, prototyping, spiral, object-oriented, etc.) [335]. This methodology focuses on how to bridge the gap between requirements and the first high-level design, and it is not concerned with detailed design, implementation, and maintenance of software.

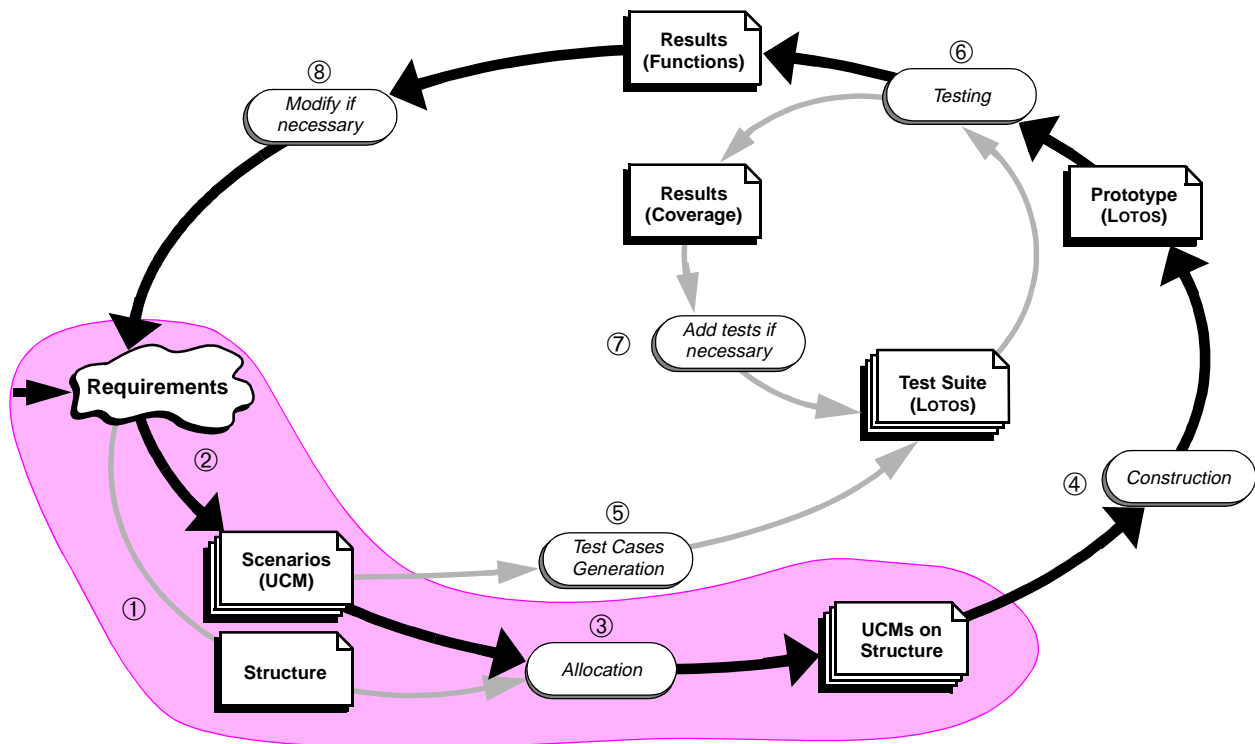
Still, SPEC-VALUE shares many similarities with prototyping approaches [172]. Prototypes focus on the aspects of the system that are most important to the customer, and they provide means for early validation. Prototypes are well suited in environments where requirements are yet to be fully determined, a place where UCMs also proved to be useful. Allowing the stakeholders to play with a prototype can give invaluable insight into the feasibility or correctness of solutions under investigation [120]. Often, prototypes demand much effort and investment, and yet they are thrown away because they implement a subset of the requirements only. However, very abstract and high-level lan-

guages such as LOTOS require less effort for the generation of prototypes, and the latter exhibit a high reusability when new system functionalities become needed.

SPEC-VALUE also resembles Boehm's spiral model [54]. Both approaches are iterative and they enable the integration of new scenarios and functionalities, as well as the modification of existing ones. Iterations also help to understand the problem and to cope with it in small chunks. Risk analysis is at the basis of each iteration [55]. It is used to determine and evaluate the alternatives as to what to integrate and validate next.

The spiral model contains macro-iterations for requirements capture and analysis, design, implementation, and so on. This model is general enough for other process models to be used within a macro-iteration [290]. For instance, SPEC-VALUE could find its place nicely as the requirements capture and analysis macro-iteration for processes targeting complex telecommunications systems.

FIGURE 17. From Requirements to UCMs with SPEC-VALUE



4.2 First Steps of the SPEC-VALUE Methodology

The three first steps of SPEC-VALUE, where UCMs are used to capture the informal operational requirements, are highlighted in Figure 17. The elicitation and representation of causal scenarios will be briefly introduced in this section, and then illustrated in Section 4.3. Several guidelines for the use of the UCM notation and the integration of scenarios are given in Section 4.2.2 and Section 4.2.3 respectively.

4.2.1 From Requirements to UCMs

One interesting contribution of SPEC-VALUE, which is inherited directly from UCMs, is the separation of concerns between system functionalities and underlying structure. A structure contains the abstract system components of interest as well as some of their relationships (containment, communication links, etc.). Step ① is the description of the system structure, which are represented in this thesis using Buhr's component notation (Appendix A: — A8 and A9). The components represent coarse-grained entities of relevance to the requirements engineers and designers. They can be extracted directly from the requirements or environmental constraints, or discovered during an iteration in the approach. They are different from classes in the OO world. A component could be represented by aggregating many class instances (objects), or a class could represent many different components or roles. Hence, there is not necessarily a one to one mapping between OO classes and components [74].

Step ② is a scenario elicitation phase where system services and large-grained functionalities are captured as UCM paths. These paths represent scenarios whose emphasis is on the causal relationships among the responsibilities that compose the functionalities. The elicitation can be done from informal requirements, business goals, interviews, existing documentation, designs, code, test cases, and so on.

The responsibilities defined in the UCMs can be allocated to the components in the selected underlying structure (step ③). Each component will have to perform the responsibilities allocated to it. The double binding of responsibilities (to paths and to components) is what links behaviour and structure.

Since scenarios are formalized at a level of abstraction higher than message exchanges, different underlying structures or architectures can be evaluated with more flexibility, even before the generation of a prototype.

4.2.2 Style and Content Guidelines for UCMs

The Use Case Map notation is flexible and can be used across a wide range of domains. However, the notation does not come with a predefined set of style and content guidelines. UCMs have been used in diverse ways, and with various levels of looseness (a classification is given in [266]). Constructing UCMs with the goal of generating LOTOS specifications [24], SDL specifications [317], ROOM models [62], agent systems [77][124] or Layered Queuing Networks (LQNs) [283][324][325] affects the style in which the paths are drawn and the supplemental information that needs to be attached to the responsibilities and other path elements.

In this thesis, several guidelines will be applied in order to facilitate the generation of LOTOS specifications from UCMs:

- G1.** Start points going towards the components representing the system under design and end points coming out of them will represent interaction points with the environment, i.e. the users.
- G2.** If data values or variables need to appear on the UCM on start and end points, they can be added to the labels using the syntax *!value* or *?variable*.
- G3.** Labels will be used for all responsibilities, start points, end points, timers and waiting places. Labels should be valid LOTOS identifiers.
- G4.** Responsibilities, start points, end points, timers and waiting places located inside a system component will be hidden from their environment (may not be the case for actors, see **G7**). This is only an assumption in the UCM domain because UCMs do not support interfaces yet, but this will be made concrete in the LOTOS domain.
- G5.** Guards on path alternatives will be identified by italicized conditions between square brackets. Responsibilities are not to be used as conditions.

- G6.** A causal flow between two responsibilities in two different components implies the need for message exchanges. However, this does not imply that these components can communicate directly; intermediate components may be involved.
- G7.** Users (and their roles) may be represented as components. Start/end points, waiting places and responsibilities associated to the users are visible. Multiple concurrent users are represented as stacks of components.
- G8.** Users could have different roles (e.g. originator and terminator). These roles are best represented as two different components in a UCM (for better understandability), but they will be reunited as one component in the formal specification. The labels used will be of the form `ComponentName:Role`.

This set of guidelines is provided as is, without any intention to validate or complete it in the thesis.

4.2.3 Integration of Scenarios

Often, system functionalities will be described as individual UCMs, and they need to be integrated together in the UCMs and/or the LOTOS prototype. Also, as requirements are dynamic, new functionalities may become necessary, and new scenarios will have to be integrated in the old set.

Not selecting a good mix of system functionalities for the initial (and subsequent) increment ranks in third place among the problems of use case modelling identified by Chandrasekeran [90], therefore this is an important problem. Karlsson *et al.* already evaluated six methods for prioritizing software requirements [224]. Unfortunately, their results are not really useful here. None of the prioritizing methods described in this article provides means for handling interdependence, and the study focuses on non-functional requirements. Use Case Maps emphasize functional requirements, and often scenarios are interdependent.

When integrating UCMs or constructing a model (in LOTOS or any other language), one should try to sort scenarios. In order to reduce integration risks, priority should be given to the most important scenarios, i.e. the ones with the most impact on the system, and to the ones that are the least likely to change.

If functionalities can be integrated in a hierarchical way (through stubs and plug-ins), then the top-level maps should be worked out first. The presence or absence of stubs in these maps has a big impact on how easily new UCMs can be integrated, so the need for stubs should be anticipated. Dynamic stubs also allow more flexibility than static stubs (although a static stub could be transformed into a dynamic one when necessary).

Again, these few integration guidelines result from the author's experience and the thesis does not aim to validate them explicitly.

4.3 Ongoing Example: Tiny Telephone System (TTS)

The goal of the Tiny Telephone System is to illustrate several steps that allow requirements engineers and designers to bridge the gap between requirements and UCMs. First, individual requirements will be provided, then individual UCMs will be constructed for each feature of TTS, and finally an integrated view of the functionalities will be given as one global UCM.

4.3.1 Informal Requirements for TTS

TTS is used to establish a telephone connection between two parties. The originating party (or *caller*) is the user who initiates the call, and the terminating party (or *callee*) is the one who receives the call. The basic call service is as follows. The caller, who is initially busy but not involved in a call connection, initiates a call request (req) to the system. If the callee is idle, then the callee's phone will ring (ring) after some internal update (to reflect that the callee is now busy) while the caller's phone will emit a ringback signal. If the callee is already busy in another phone session, then the caller's phone shall emit a busy signal.

TTS users may also have subscribed to additional features. The first one is Originating Call Screening (OCS), which will deny the call if the callee is in the caller's screening list (and emit the appropriate signal to the caller). The second is Call Number Display (CND), which displays the caller's number on the callee's phone (disp) while the latter is ringing.

Each user has an agent that takes care of the handling of internal databases and updates. To simplify the design, TTS assumes that users cannot register to or unregister from a feature. Users are initially subscribed to a list of features, which may be empty.

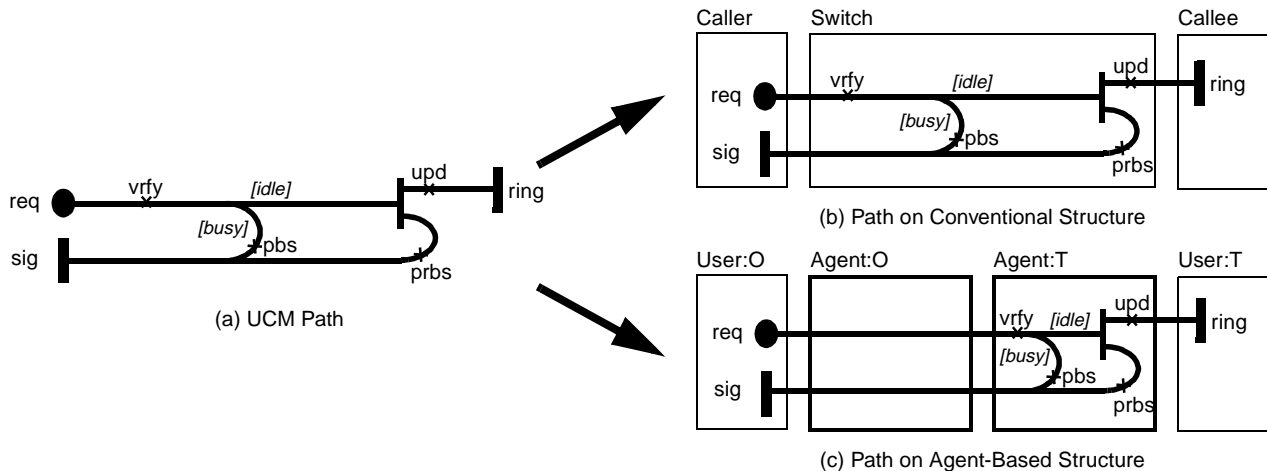
4.3.2 Individual Use Case Maps for TTS

Typical telephone system documentation will first describe the basic call, and then the basic call combined to one feature. This section follows the same idea by presenting UCMs for TTS' basic call, OCS, and CND.

TTS Basic Call

Starting from the requirements, the designer can draw a UCM path like the one in Figure 18(a). This corresponds to step ① in SPEC-VALUE. The req and ring signals, which are start and end points, were mentioned explicitly in the requirements. However, this UCM exhibits additional responsibilities that were left implicit: vrfy verifies whether the called party is idle or busy (conditions are between square brackets) and upd updates the callee's status. Instead of having a multitude of call progression signals on the caller's side, a single end point (sig) is used to propagate the appropriate signal or announcement from the system to the user. Additional signals and announcements are expected to be carried in the same way, so this simplifies the UCM and the overall design. A consequence is that appropriate signals need to be prepared by the system: pbs prepares a busy signal whereas prbs prepares a ring-back signal. As explained in guideline **G2**, input variables and output values could be associated to start and end points when necessary (e.g. req?Callee and sig!Signal). However, these will not be shown for the UCMs to remain simple.

The underlying structure can be of various natures, and this is the focus of step ② in SPEC-VALUE. For instance, Figure 18(b) shows the UCM path bound to a conventional switch-based structure. However, the requirements suggest a less centralized architecture where each user has an agent. Figure 18(c) presents a possible agent-based structure. The agents are all alike, but they have different roles: originating (O) or terminating (T). The same idea applies to the users as well.

FIGURE 18. TTS Basic Call UCM

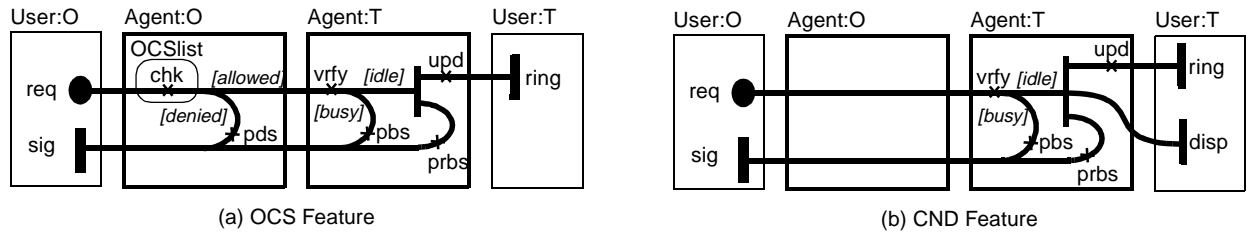
Under an apparent simplicity, UCMs such as Figure 18(c) convey a lot of information in a compact form, and they allow requirements engineers and designers to use two dimensions (structure and behaviour) to evaluate architectural alternatives for their system. Once both views are satisfactory, then they are combined to form a bound UCM, as indicated by step ③ in SPEC-VALUE. The binding is done by allocating the UCM responsibilities (and optionally start points, end points, and other path elements) to the components in the structure. Figure 18(c) represents the result of the allocation.

OCS and CND Features

Individual UCMs for OCS and CND, based on the structure of agents, are given in Figure 19. OCS requires a passive object (e.g. a database) which represents the list of screened numbers that the caller is forbidden to contact (OCSlist). This new component can be checked (chk) to determine whether the call should be allowed or denied at the originating side. Denied calls cause some update and the preparation of an appropriate signal (pds) in the originating agent.

CND extends TTS' basic call with a new result which displays the number of the caller (disp). This display is concurrent with the ringing of the phone at the terminating end.

FIGURE 19. Individual UCMs for TTS Features



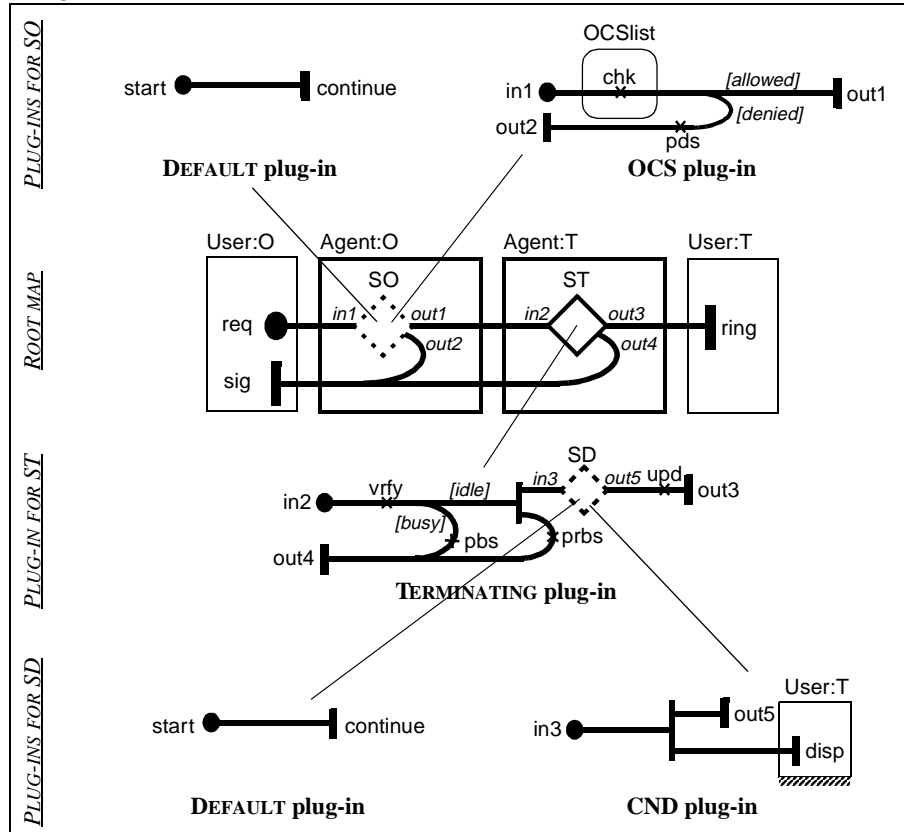
4.3.3 Integrated UCM View

Refinement of designs can be obtained in UCMs by the use of stubs and plug-ins. The root map in the middle of Figure 20 shows an enhanced version of the UCM from Figure 18. Through the use of stubs, this root map enables the integration of many scenarios coming from different features. The OCS and CND features described in Figure 19 are integrated to the root map as plug-ins for the stubs. The five different UCMs (the DEFAULT plug-in is used twice) shown in Figure 20 represent only one way to integrate the individual UCMs seen so far; other possibilities exist but they will not be explored here. Generating integrated views is also part of step ③ in SPEC-VALUE. Although such a view is not mandatory, it usually helps structuring the scenarios together, ensuring their consistency, and avoiding undesirable behaviours or side effects.

The originating dynamic stub SO has two plug-ins (DEFAULT and OCS). The start point of the DEFAULT plug-in (start) is bound to the incoming path segment in1, and the end point continue is bound to the outgoing segment out1. The OCS plug-in includes the OCSlist component, which is then considered to be inside the Agent component. If the caller subscribes to the Originating Call Screening service, then the originating agent will select the OCS plug-in instead of the DEFAULT plug-in. This is the *selection policy* of the dynamic stub SO.

The terminating static stub ST contains one plug-in only (TERMINATING), hence selection policies are not necessary here. Static stubs enable modular and stepwise decomposition of functionality. Their plug-ins act as refinements and they also prevent the calling maps from being cluttered with too many details.

FIGURE 20. Integrated UCM View of TTS



TERMINATING includes the dynamic stub SD, which handles the display of information to the terminating user. SD’s selection policy states that the CND plug-in is selected when the terminating party is a Call Number Delivery subscriber. Otherwise, the DEFAULT plug-in is selected.

Note that the DEFAULT plug-in is reused in two different stubs. However, the bindings are different. For SD, the start point of DEFAULT (start) is bound to the incoming path segment in3, and the end point continue is bound to the outgoing segment out5. The binding relationships of the other plug-ins is defined in this example by start points being bound to incoming path segments of the same name and by end points being bound to outgoing path segments of the same name.

The CND plug-in has some interesting characteristics. First, it leaves an end point dangling; disp is not bound to any output segment of the stub, but it becomes a new observable event at the ter-

minating end. Not all plug-in start points and end points need to be explicitly bound to stub segments. However, this flexibility needs to be used with moderation otherwise parent maps (where such plug-ins are called) will no longer represent the big picture clearly. Second, the CND plug-in includes a reference to an existing component (User:T), defined in the root map. Referenced components, also known as *anchored* components, are shaded in the UCM notation. Such component is interpreted as being declared outside the component that contains the calling stub. This means that User:T in CND is not a sub-component of Agent:T (obviously, User:T is defined at the top level).

A plug-in that uses anchored components is said to be in an *unconstrained style* [76]. This style enables parent (root) maps to be simplified by showing only the main paths through a set of components, treating meandering across components as details deferred to plug-ins. Another benefit is that considerable flexibility in filling in details is provided. Also, sub-components can be declared or referenced in the appropriate maps, when required by the causal scenarios that need to be supported. However, this style breaks the component containment intuition shown in UCM structures and it leaves the big picture somewhat incomplete, requiring it to be mentally pieced together from different maps. Therefore, the unconstrained style and anchored components should be used with special care.

Once stubs are defined at key points on a path, it becomes easy to add new plug-ins, which could represent new features in the TTS example. Existing maps and plug-ins can further be decomposed or extended (e.g. when a radically different service is added) with new paths and new stubs. Experienced requirements engineers and designers may even prefer to skip the individual flat UCMs (e.g. Figure 18(c) and Figure 19) and work directly with an integrated UCM view, where all the UCMs are connected through stub/plugin bindings (e.g. Figure 20).

By selecting plug-ins for the stubs in the integrated UCM view, one can obtain a flattened map, which may still contain multiple end-to-end scenarios. For instance, by selecting the DEFAULT plug-in in stubs SO and SD, the resulting map becomes the same as the original basic call in Figure 18(c). The OCS and CND individual UCMs can also be derived in the same way. The inte-

grated view contains however more scenarios, resulting from the combination of individual features. For instance, a totally new UCM would result from the case where both OCS and CND are active. This is what enables designers to reason about undesirable interactions between features at a high-level of abstraction (to be discussed further in Section 8.3.1).

4.4 Chapter Summary

This chapter recalls some of the main motivations behind the use of Use Case Maps and LOTOS in the SPEC-VALUE methodology, many of which resulted from the literature review in Chapter 3. The methodology is also briefly compared to common software design processes such as the prototyping model and the spiral model.

Section 4.2 expands on the three first steps of the SPEC-VALUE methodology, which are concerned with the capture of functional requirements in terms of UCMs. A particular attention is devoted to style and content guidelines that increase the completeness of UCM descriptions as well as the compatibility and traceability between UCMs and subsequent models (such as LOTOS prototypes). The integration of scenarios is also briefly discussed along with a few integration guidelines resulting from the author's experience.

These steps are illustrated in Section 4.3 with an example intended to be used throughout the thesis: the *Tiny Telephone System* (TTS). The basic call responsibilities and causal flows are derived from informal requirements (step ①), together with an appropriate structure of components (step ②). Responsibilities are allocated to components, resulting in bound UCMs (step ③). The OCS and CND features are also represented as individual UCMs. Then, the features and the basic call are merged to form an integrated UCM view of TTS, where stubs and plug-ins are heavily used. This section also discusses the appropriateness of selection policies, of the unconstrained style with anchored components, and of the flattening of integrated UCMs

Contributions

The following items are original contributions of this chapter:

- Partial illustration of Contribution 1 (Section 1.4.1) regarding the separation of the functionalities from the underlying structure and the design documentation in SPEC-VALUE.
- Illustration of the first steps of SPEC-VALUE, i.e. from requirements to UCMs.
- Style, content, and integration guidelines for Use Case Maps.
- Informal requirements and UCMs for the Tiny Telephone System.

CHAPTER 5

From Use Case Maps to LOTOS in SPEC-VALUE

*Don't worry about that specification paperwork.
We'd better hurry up and start coding, because
we're going to have a whole lot of debugging to
do...*

Barry Boehm, 1984

This chapter presents an analytic approach for the construction of LOTOS specifications from Use Case Maps. The core of this approach is found in Section 5.2, where construction guidelines (CG) are introduced and several are illustrated individually. These guidelines are then applied to the Tiny Telephone System example, and the resulting LOTOS specification is discussed in Section 5.3.

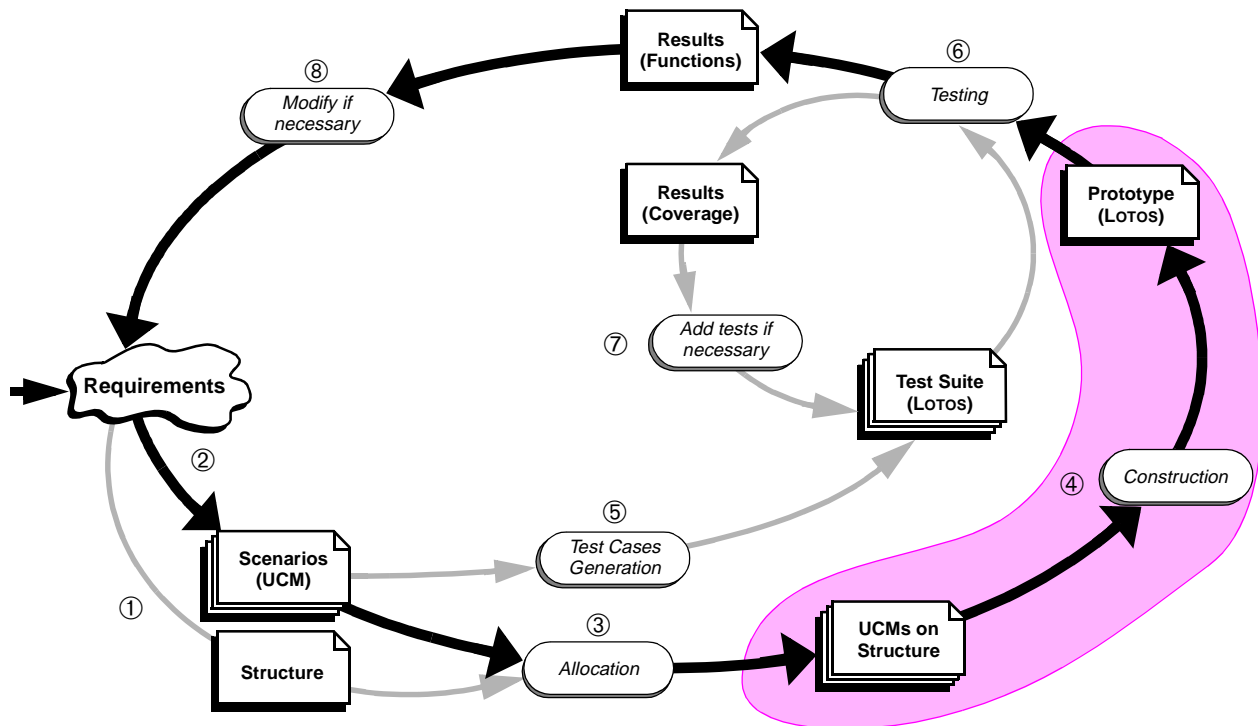
5.1 Construction Approach

The construction of a LOTOS prototype from UCMs corresponds to step ④ in the SPEC-VALUE methodology and is highlighted in Figure 21. The prerequisite is a collection of UCMs (which could be integrated or not), similar to what was obtained for the Tiny Telephone System in Chapter 4. The output is a specification that captures the functional requirements and the high-level design, where the structure of components may or may not be considered.

Several methods for the integration of scenarios and the construction of models were reviewed in Section 3.3.4. Most of the synthetic and analytic approaches surveyed are based on various interaction diagram notations and they involve components and messages. UCMs are defined above the level of messages, and they may or may not include components. Algorithms for the automated and inter-

active synthesis of integrated models from message-oriented scenarios hence are not well suited for translating UCMs fully and automatically.

FIGURE 21. From UCMs to LOTOS with SPEC-VALUE



An analytic approach, where the model is generated manually and then verified, appears to be the most pragmatic way of capturing UCMs directly in LOTOS. As shown in Table 10, analytic approaches have some advantages over synthetic ones:

- They do not require a formal representation of the scenarios (this is particularly relevant to UCMs, which are semi-formal).
- The source and target modelling languages do not need to be restricted, i.e. their richness can be exploited to their fullest extent.
- The target model can additionally take into consideration further design constraints and non-functional requirements, which are not necessarily captured by the scenario model.

5.1.1 Appropriateness of LOTOS

The choice of LOTOS as a target modelling language is motivated by several factors, many of which were mentioned in Section 3.2, Section 3.4, and Table 1 on page 6:

- This language is capable of expressing behaviour at several stages of design or levels of abstraction, including the initial ones where it is not yet known what the components of the system are, what are their states, and what are the messages exchanged between them (this is normally the situation at the early stages of design).
- LOTOS is mature in the sense that it is an established international standard, around which much useful theory and a number of useful tools have been developed.
- UCMs and LOTOS both focus on the ordering of actions and they share many constructs that have similar semantics (such as sequence, alternative, parallelism, hierarchical design and structure), which result in simpler mapping and traceability relations.
- LOTOS is capable of specifying UCM with and without components.
- LOTOS specifications are executable prototypes that can be formally analyzed and validated against the intended functionalities of informal requirements and individual scenarios. Other types of liveness and safety properties can be verified as well. These capabilities complement most of UCMs' weak areas related to the analysis of requirements (in terms of maturity, completeness & consistency, testing & simulation, verifiability & correctness).
- LOTOS enables the automated generation of diagrams such as Message Sequence Charts. Sequences produced by a LOTOS prototype can be translated into MSCs, which are often more suitable than UCMs for the visualization of detailed scenarios, for diagnostics, and for providing scenario information to programmers, testers, and automated tools.

Use Case Maps and LOTOS therefore represent a good match with much potential for specifying and validating telecommunications systems.

5.1.2 Unfitness of TMDL

In his masters thesis [12], Amyot presented a methodology for the semi-automated generation of LOTOS specifications from unbound UCMs. The maps were manually described using the *Timethread Map Description Language* (TMDL)¹, and then a compiler (`tmdl2lot`) would generate the specification automatically [13]. Although this approach has been successfully used for a simple telephony system [14], TMDL lacks three major features that are necessary for the modelling of realistic telecommunications systems:

- **Components:** TMDL does not consider any structural artifact. Use case paths are the only type of object described (unbound maps). The resulting specification becomes consequently purely functional in nature, like a service specification, without any message passing. However, there comes a point in the design cycle where components cannot be avoided, especially when multiple instances of a particular component (e.g. telephones or agents) need to be considered. The distribution of behaviour over a topology of components is challenging and difficult to automate.
- **Data types:** TMDL does not have data types, yet many complex telecommunications systems rely heavily on a data model for databases, conditions, and parameters.
- **Composition:** Since TMDL does not support hierarchical design with stubs and plug-ins, the designer has to provide a single global UCM where all scenarios are correctly composed. Unfortunately, as the system complexity increases, this approach becomes quickly unpractical for most realistic systems. Generating such a global map would result in a very large picture, difficult to understand, maintain, and analyze, hence defeating the purpose of UCMs.

TMDL is essentially an example of the unfitness of an automated synthesis approach that excessively constrains the source modelling language. In the thesis, TMDL is put aside in favor of an analytic approach.

1. Use Case Maps were previously called *Timethread Maps*.

Analytic approaches suggest the use of *guidelines* for generating target models from source models. Over the last seven years, many LOTOS specifications have been produced out of systems designed with UCMs: a Telepresence system [12], a simplified Plain Old Telephone System (POTS) [14], a Group Communication Server [15][17], the Group-Call service of the GPRS mobile telephony system [16][24], a feature-rich telephony system [18][22], an agent-based PBX [25], a Wireless Mobile ATM Network [32], and the Call Name Presentation service of the Wireless Intelligent Network [381]. The author has written or collaborated to the generation of most of these specifications. The guidelines enumerated in the next section result from lessons learned during these experiments (some of which will be further explored in Chapter 8).

5.2 Construction Guidelines

This section presents guidelines for the construction of LOTOS specifications from UCMs. SPEC-VALUE is limited to the provision of guidelines, which are informal mapping rules that have proven to be useful but which may not necessarily be followed exactly by users at a given point. Rigorous construction rules would essentially lead to the creation of synthesis algorithms, hence diminishing the flexibility targeted by analytic approaches. In this context, the guidelines are more in line with design patterns [101][144] than with formal transformation rules, although guidelines have a simpler structure and are less detailed than conventional design patterns.

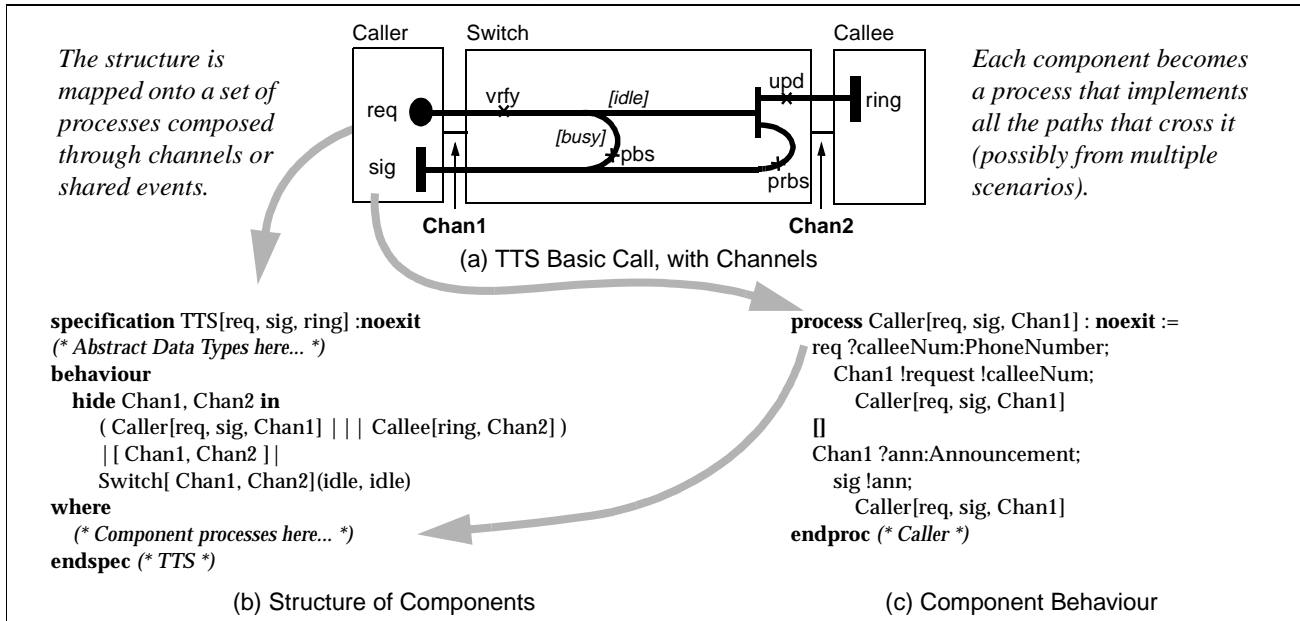
An overview of the main construction principles is given first, followed by an enumeration of several guidelines illustrated with short examples inspired from the Tiny Telephone System. Additional comments on the partial automation of such guidelines conclude the section.

5.2.1 Overview

In a nutshell, the construction approach (step ④ in Figure 21) consists in translating each UCM component into a LOTOS process that preserves the internal causality relationships between the responsibilities and events that are part of path segments crossing this component. This idea is illustrated with the TTS Basic Call UCM, originally found in Figure 18(b), where the Caller component is mapped onto a LOTOS process (Figure 22(b)). The structure itself is converted to a set of processes composed

through shared communication channels (LOTOS gates), as in Figure 22(c). The causal relationships between the components are also considered during the construction of the processes. Decisions related to the nature of the message exchanges must then be made and documented.

FIGURE 22. Construction of a LOTOS Specification from a UCM



The structure of the target model (a LOTOS specification in our case) and the selected specification style will also influence the ease with which new scenarios can be introduced. A style that reflects both the UCM structure of components and the use of responsibilities, stubs and plug-ins is likely to be flexible and easy to handle when the time comes to specify changes or additions in a UCM. This style is inspired from the conventional *resource-oriented style* [363], where components and media are specified as communicating processes. However, the processes developed using the guidelines of the next sections are not limited to the use of sequences, alternatives, and process instantiations as in the conventional resource-oriented style; concurrency will also be used internally.

Eight guidelines are developed in this chapter. Their main goal is to provide a traceable path from UCMs to LOTOS (and often in the other direction as well) where the intended behaviour of the UCMs is formalized. Sub-guidelines are also defined for different aspects of the general guidelines

and for special cases. The given guidelines should provide useful hints and guidance to beginners and intermediate users of SPEC-VALUE. However, experienced users may deviate occasionally from these guidelines for various reasons (examples will be given in Chapter 8), sometimes at the cost of reduced traceability.

The construction guidelines are structured into three families. Although they intend to be as orthogonal as possible, many of the more detailed guidelines refer to other families. These guidelines are meant to be used iteratively and no pre-defined order is suggested.

- **Paths:** focus on the construction of LOTOS behaviour expressions from UCM paths and their elements (Section 5.2.2).
 - ◆ Construction Guideline 1: Interpreting Interaction Points and Responsibilities
 - ◆ Construction Guideline 2: Representing Causal Paths
 - ◆ Construction Guideline 3: Interpreting Stubs and Plug-Ins
 - ◆ Construction Guideline 4: Other Path Elements
- **Structure:** focus on the construction of LOTOS behaviour expressions from UCM components, the paths they contain, and the inter-component paths that link them (Section 5.2.3). This is where most challenges reside in terms of automation as many design decisions are required.
 - ◆ Construction Guideline 5: Interpreting the Structure
 - ◆ Construction Guideline 6: Integrating Multiple Unrelated Path Segments in a Component
 - ◆ Construction Guideline 7: Refining Inter-Component Causality
- **Data:** focus on the construction of a LOTOS data model and supporting infrastructure elements such as databases (Section 5.2.4).
 - ◆ Construction Guideline 8: Representing Data

5.2.2 Construction Guidelines for Paths

Construction Guideline 1: Interpreting Interaction Points and Responsibilities

Interaction points include start points, waiting places, and end points. Together with responsibilities, they are captured as LOTOS gates. Guideline CG-1 captures the general case whereas Guideline CG-1.a considers how responsibilities and end points may affect value identifiers.

CG-1) Interaction points (start points, waiting places, and end points) and responsibilities are specified as LOTOS gates.

These elements capture interactions with the environment: start points and waiting places can be controlled during the validation whereas end points and responsibilities can be observed. Preconditions associated with start points and waiting places can additionally be translated to selection predicates. Interaction points *req*, *sig*, and *ring* in Figure 22(b) were transformed to gates according to this guideline. Likewise for responsibilities *vrly*, *upd*, *prbs*, and *pbs*. Similar names are given to improve two-way traceability and understanding. If a name is not a valid LOTOS name (e.g. LOTOS keyword or forbidden character), then it will need to be changed at the UCM level or to be made valid through some documented translation. The visibility of these elements and their relationship to component interfaces are further discussed in Guideline CG-5.c and Guideline CG-5.b respectively.

CG-1.a) UCM responsibilities and end points may affect the content of value identifiers.

The effects of a responsibility's computation or of an end point's postcondition, if any, can be represented as modifications of variables (called *value identifiers* in LOTOS) in the behaviour expression that follows the corresponding action. For instance, if *upd* updates the callee's status to *busy*, then this can be represented as:

```
upd; (let calleeStatus:UserState = busy in B)
```

where *B* is the behaviour expression that follows *upd*.

An interesting case is when `B` is a process instantiation (as it is often the case when capturing an end point or when the path exits the component, see Guideline CG-2): the values can be used directly as process parameters. For example:

```
upd; Switch[Chan1, Chan2](callerStatus, busy)
```

where the switch maintains the status of the caller (first parameter) and of the callee (second parameter, here updated to the value `busy`).

Value identifiers can also be modified through a database, as suggested in Guideline CG-8.

Construction Guideline 2: Representing Causal Paths

Causal paths inside a component are represented through the use of appropriate LOTOS constructs. Sequences (possibly with recursion) represent the simplest way of capturing causality (Guideline CG-2). Various UCM constructs enable simple aggregations of sequential paths, and they are handled directly by more specific guidelines: OR-forks with Guideline CG-2.a, AND-forks with Guideline CG-2.b, OR-joins and AND-joins with Guideline CG-2.c. Path interactions represent another special case and they are treated by Guideline CG-2.d. Most of these guidelines were introduced in [12].

CG-2) Linear causal paths are represented as sequences using the action prefix operator (possibly with recursion).

UCM path segments represent the linear progression of causality. This can obviously be captured using the LOTOS action prefix operator. Ends of path segment in a component can also be supplemented by a reinstantiation of the component process. Such recursive calls enable component persistence, i.e. a component can “execute” the causal paths multiple times, a quality that is often required of reactive systems components.

For example, the Caller process contains this partial behaviour:

```
req ?calleeNum:PhoneNumber; (* get a connection request *)
  Chan1 !request !calleeNum; (* send it to the switch *)
  Caller[req, sig, Chan1] (* get ready for the next request *)
```

CG-2.a) The choice operator is used to represent alternatives (OR-forks).

This requires little explanation as both operators denote similar concepts. Choices may be guarded when conditions are attached to paths leaving the OR-fork.

For example, the Switch process could contain the partial behaviour:

```
vrfy;
( [idle] -> ...
  []
  [busy] -> pbs; ... )
```

The multiple use of the choice operator can represent OR-forks with more than two branches.

CG-2.b) The interleave operator is used to represent concurrent paths (AND-forks).

Again, both concepts are very similar. The Switch process hence could contain:

```
[idle] -> ( upd; ...
           |||
           prbs; ... )
```

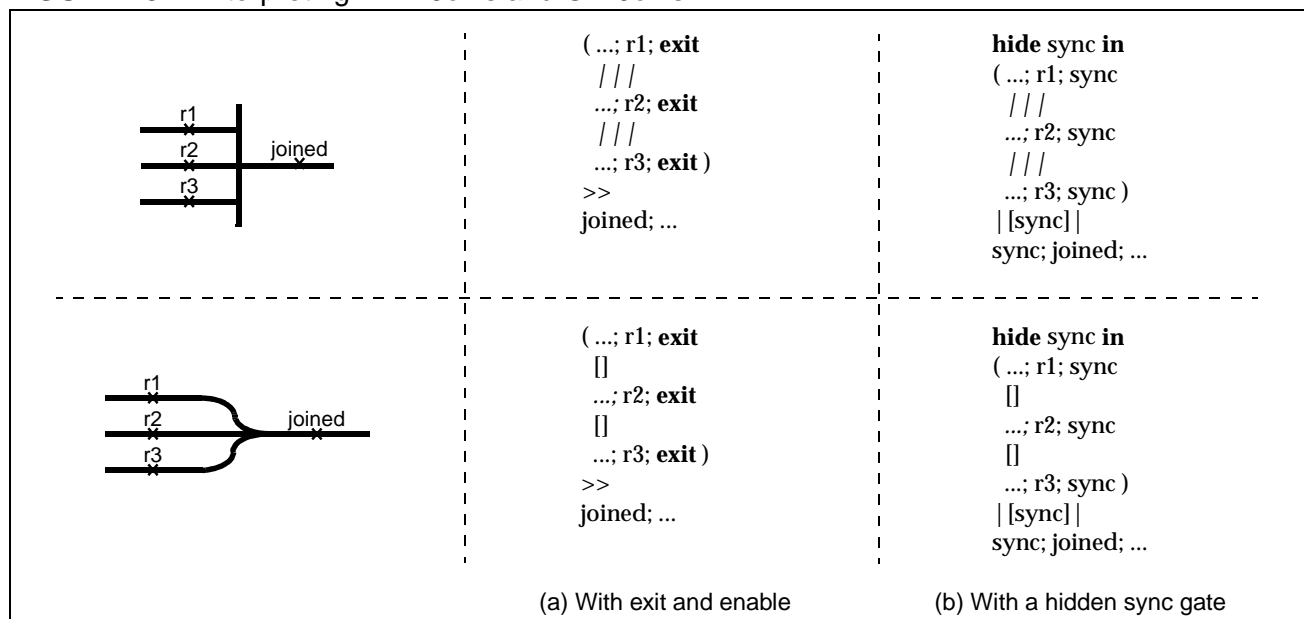
The multiple use of the interleave operator can represent AND-forks with more than two branches.

CG-2.c) AND-joins and OR-joins are specified with the enable operator or with hidden gates.

Concurrent paths and alternative paths entering an AND-join or OR-join need to **exit** (on compatible values, if any). The enable operator (>>) is then used to capture AND-joins and OR-joins.

This leads to the enabled segment (behaviour expression) that follows the joined paths (Figure 23(a)). Alternatively, the **exit** can be replaced with a hidden synchronization gate, and the enable by the generalized synchronization operator. In this case, the segment following the joined path has to start with the synchronization gate, which is also used in the synchronization operator (Figure 23(b)).

FIGURE 23. Interpreting AND-Joins and OR-Joins



OR-joins may also be represented with duplicated behaviour expressions (one for each joined path), particularly when the next UCM path element is an end point. If an OR-join causes a path to loop on itself, then a sub-process, which specifies the loop path, is instantiated. It reinstantiates itself recursively for each iteration, and it exits to terminate the loop. The enable operator then captures this **exit** and continues with the rest of the UCM path.

CG-2.d) Synchronous and asynchronous interactions between UCM paths are specified using the generalized synchronization operator.

Start points and waiting places may be triggered either synchronously by an end point coming from another UCM path, or asynchronously by an empty path segment coming from another UCM

(see Figure A3 in Appendix A:). The UCM paths involved in the interaction are specified in LOTOS as concurrent behaviour synchronized on the gate name of the start point or waiting place. The triggered path awaits this event. The triggering path will also wait for this event in case of a synchronized interaction. However, in case of an asynchronous interaction, the triggering path will provide this event concurrently ($| | |$) with the behaviour representing the rest of its path. Different types of extended waiting places are defined in [12], but they are not considered here as they do not belong the standard UCM notation.

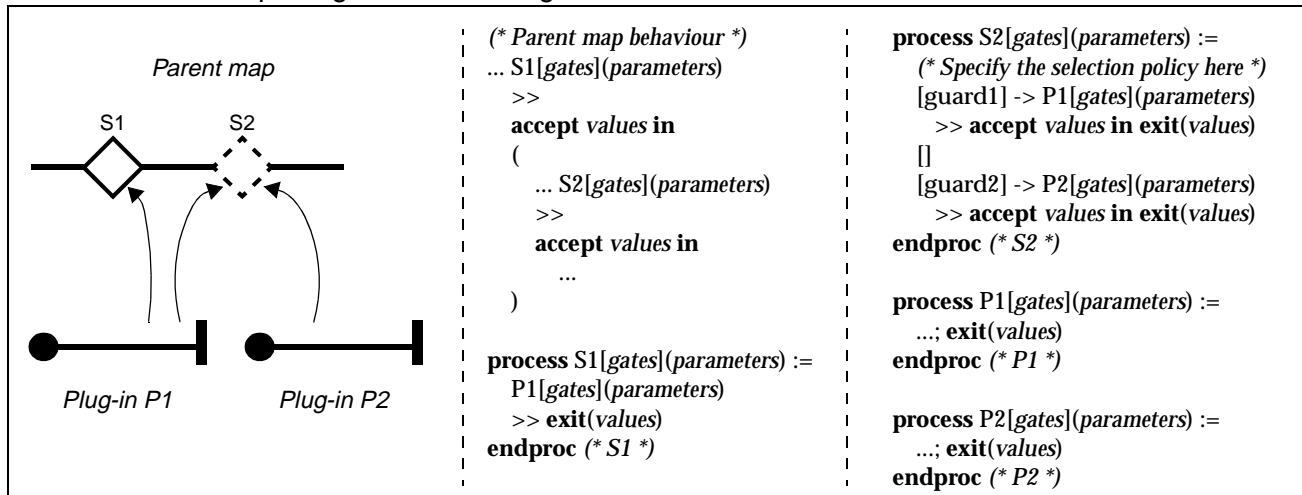
Construction Guideline 3: Interpreting Stubs and Plug-Ins

Stubs and plug-ins are also represented with the versatile LOTOS process. These guidelines will be explained in general terms only, and they will be illustrated more completely with the TTS example in Section 5.3.3. The generic guideline is presented in Guideline CG-3, and further details are provided for plug-ins (Guideline CG-3.a), stubs and selection policies (Guideline CG-3.b), and binding relationships (Guideline CG-3.c).

CG-3) Stubs and plug-ins are processes linked through instantiations.

A plug-in can be used in multiple stubs, hence it is best represented as an independent process definition which can be instantiated at will. Stubs are also represented as processes so they can describe selection policies and be instantiated from multiple input segments. Stubs instantiate the plug-ins bound to them.

Since Figure 22 does not contain any stub, Figure 24 will be used to illustrate the essence of this guideline. The static stub S1 contains one plug-in and no selection policy, whereas the dynamic stub S2 contains two mutually exclusive plug-ins (whose selection could be guarded). All stubs and plug-ins have their own process definitions.

FIGURE 24. Interpreting Stubs and Plug-Ins

CG-3.a) Plug-ins have parameter lists for input points and values, and for output points and values.

In addition to data parameters needed by the plug-in, the process definition requires a list of input points that represents the list of plug-in start points triggered by the calling stub (there may be more than one). In return, the process definition exits with resulting values and with another list of points, which represents the resulting end points that were reached by the plug-in.

CG-3.b) The stub processes specify the selection policy, i.e. the type of composition between the possible plug-ins.

Static stubs have no selection policy per se, hence their behaviour is reduced to instantiating the plug-in process. Dynamic stubs however have to choose among potentially many plug-ins, hence the selection policy is captured as a behaviour expression inside the process definition. Most of the time, the selection policy will result in the deterministic instantiation of only one plug-in process. Therefore, selection policies in LOTOS often make use of guarded process instantiations combined with the choice operator ($[]$). However, more complex selection policies may require the instantia-

tion of multiple plug-in processes, e.g. in sequence (with the enable operator \gg) and/or in parallel (with the interleave operator $||$).

CG-3.c) A stub process specifies the binding relation between a stub and its plug-in(s).

Stub processes receive a list of entry points as input and then output a list of exit points upon termination. These lists correspond to the path segments enabled before and after a stub. The entry points are used to determine the list of start points (Guideline CG-3.a) that need to be triggered in a plug-in, according to the binding relation between the stub and its plug-in(s). The list of exit points is generated according to the end points resulting from the plug-in(s), again according to the binding relation. See Figure 36 for an example.

Though they have only one plug-in, static stubs still need to implement the binding relation in this way because the bound plug-in may be reused somewhere else.

Construction Guideline 4: Other Path Elements

The following elements are discussed in order to complete the coverage of the UCM path elements. However, they are not illustrated with the TTS example. More complex case studies, found in Chapter 8, will make use of these guidelines.

There is no generic guideline here, only four independent cases: timers (Guideline CG-4.a), aborts (Guideline CG-4.b), failure points (Guideline CG-4.c), and dynamic responsibilities (Guideline CG-4.d).

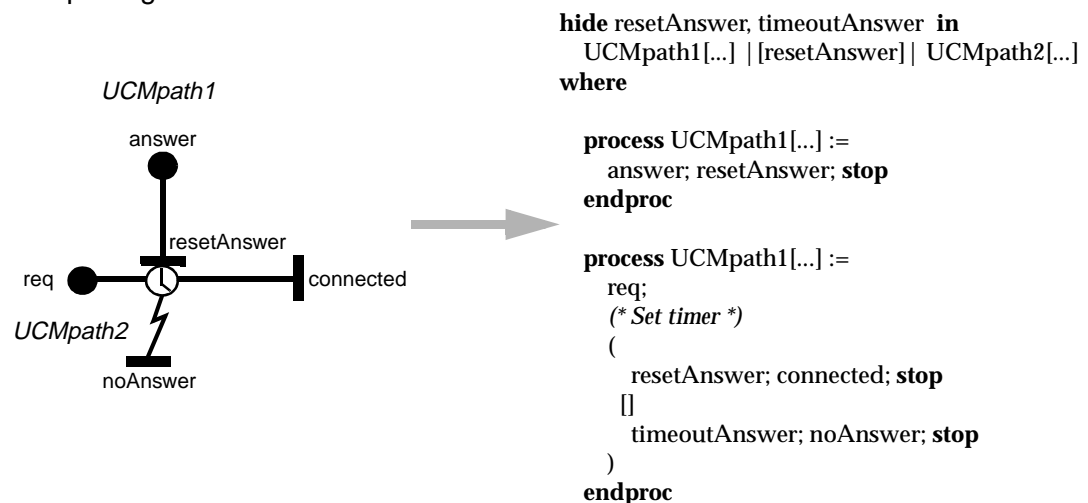
Other elements related to performance annotations (timestamp points, data stores, devices, service requests, response time requirements, etc.) and goal annotations (goal tags, goals, etc.) are not covered here. These are extensions to the UCM notation targeted towards specialized domains with objectives different than those addressed in the thesis, such as performance analysis and generation of goal-based implementations for agent systems.

CG-4.a) Timers are specified using a reset event and a timeout event.

LOTOS does not provide support for representing quantitative time. However, timers can still be specified in an abstract way. They act like regular waiting places, except that they do not wait their triggering event (reset) forever. A reset is caused (a)synchronously by a triggering path, in a way similar to the interactions discussed in Guideline CG-2.d. A timeout is represented by a LOTOS action, which can be internal or not. If hidden, this action may occur at any point. It is suggested to use a hidden gate name instead of the internal \mathbf{i} action in order to improve traceability. If observable, this action becomes under the control of the environment, which improves the overall ease with which validation can be performed. Once a timer is reached (i.e. set), there is a choice between the reset event followed by the rest of the path behaviour and the timeout event followed by the behaviour of the timeout path. This choice is non-deterministic unless the reset and the timeout events are both made observable.

Figure 25 illustrates a timer which is reset synchronously by another path. In the corresponding LOTOS code, the events `resetAnswer` and `timeoutAnswer` are hidden, but each of them could also be made observable.

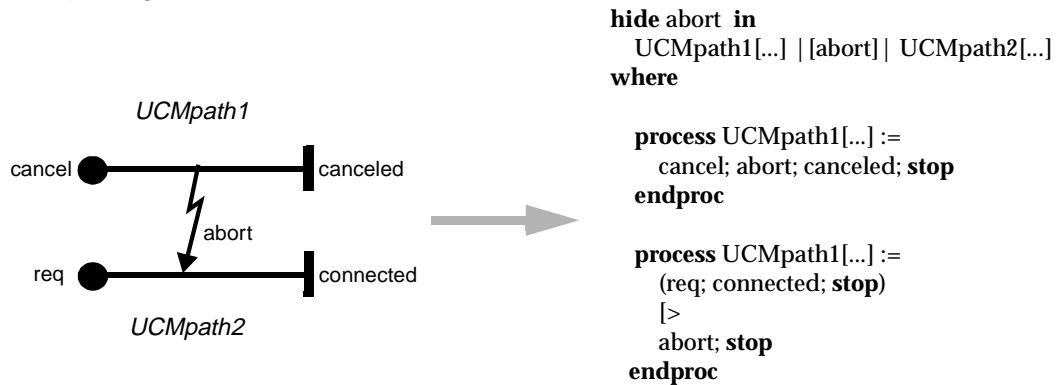
FIGURE 25. Interpreting Timers



CG-4.b) Aborts (exceptions) are specified using the disable operator.

A path may abort the progression of causality in another path. As discussed in [12], this is specified as one process disabling the other one through a shared gate (different from the channels defined in the structure). The disabling process just performs the action and then carries on. The disabled process performs the same action and then stops or reinstatiates itself.

Figure 26 illustrates a path which aborted by another path, and abort is used as a shared event.

FIGURE 26. Interpreting Aborts

In LOTOS, the disable operator exhibits a controversial behaviour: there is no way to enforce the choice of a disabling action over other ones. This means that if a component behaviour always has a choice between the disabling action and other actions, then the disabling action could potentially be delayed forever. However, one good thing about this type of behaviour is that it helps to expose potential race conditions.

CG-4.c) Failure points are specified using failure events.

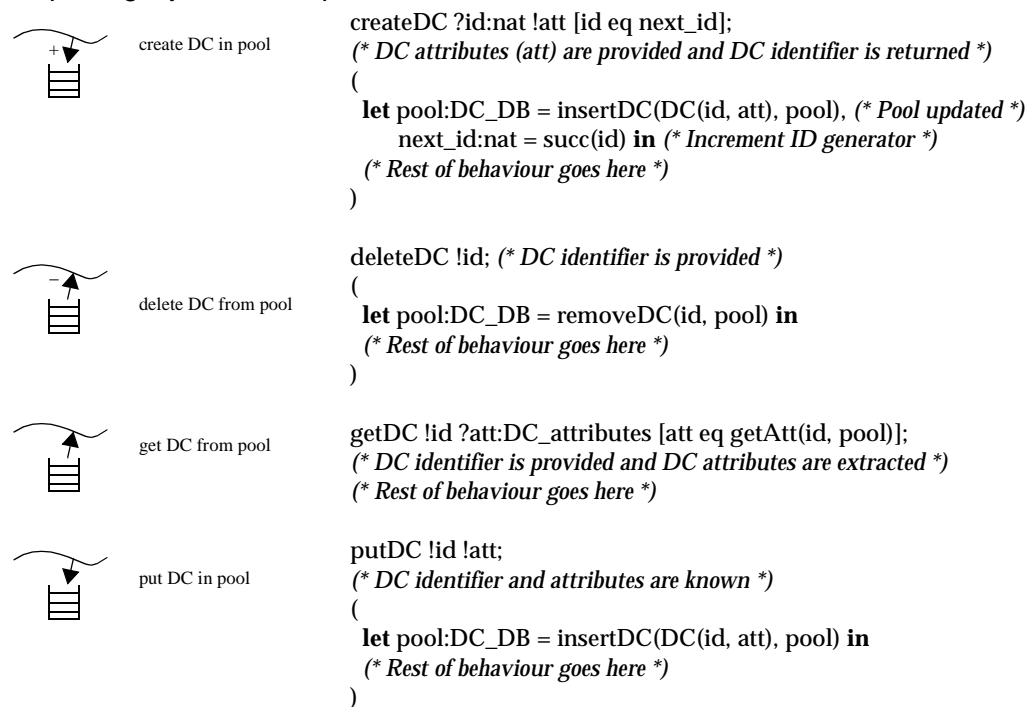
Failure points (Figure A7(c) in Appendix A:) are specified as a choice between the continuation of the path behaviour and a failure event that leads to a **stop** behaviour. Like timeout events, failure events can be hidden or visible, depending on the validation goals.

CG-4.d) Dynamic responsibilities are represented as gates or as process instantiations.

Dynamic responsibilities used on a pool of components are specified as gates. These gates have experiments that modify the content of a database that represents the pool of dynamic components (DC). Modifications are done according to the nature of the dynamic responsibility involved (create, delete, get, put).

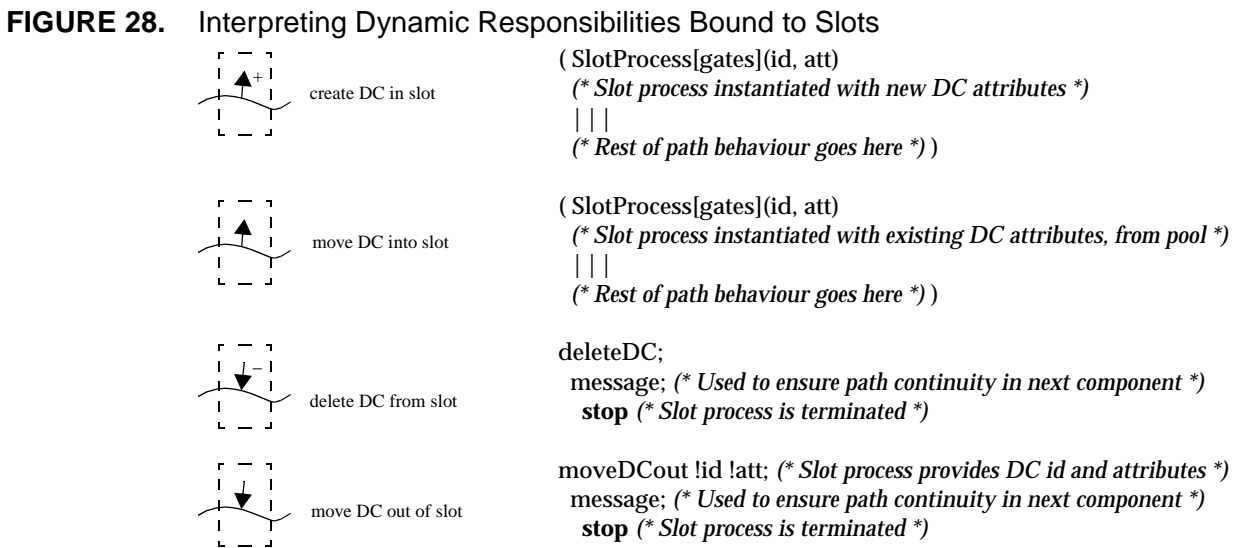
Figure 27 presents an example of how such gates could be used. The assumption here is that there exist an abstract data type for describing pools as DC databases (DC_DB), with operations for adding DCs (insertDC), removing DCs (removeDC), or getting the attributes of a particular DC (getAtt). Dynamic components also have identifiers (id, of type nat in the example), as suggested in Guideline CG-8.

FIGURE 27. Interpreting Dynamic Responsibilities Linked to Pools



As for operation on slots, the creation of a component or a move into a slot is specified as an instantiation of the relevant process. The assumption is that there exists a process definition representing the component (Guideline CG-5). The deletion of a component or a move out of a slot is specified as a gate, followed potentially by a message (to ensure the continuation of the causal path in another component) and by the termination of the process representing the component (**exit** or **stop**).

Figure 28 present short LOTOS examples illustrating how dynamic responsibilities could be specified when bound to slots.



5.2.3 Construction Guidelines for Structures

Construction Guideline 5: Interpreting the Structure

The UCM structure of components is captured as a collection of synchronized processes, where some actions may be hidden depending on the nature of the component. Guideline CG-5 represents the general case. Guideline CG-5.a considers component roles as a special case. Component interfaces are handled by Guideline CG-5.b, Guideline CG-5.c further considers hiding of path elements, and Guideline CG-5.d discusses the preservation of the hierarchical component structure.

CG-5) Components are specified as processes synchronized on their shared channels/gates.

LOTOS has a powerful concept, the process, that is general enough to represent active objects, passive objects, agents, groups of interacting objects, and so on. Therefore, any component can potentially be mapped directly to a LOTOS process with the same name. This is the case of processes Caller, Callee, and Switch in Figure 22.

When they are not provided, communication channels need to be added between communicating entities, which will then synchronize on these channels through the synchronization ($| [\dots] |$) operator. For instance, the Switch synchronizes with Caller on Chan1 and with Callee on Chan2. The Caller and the Callee do not communicate directly, hence they interleave ($| | |$). Although synchronization is achieved in LOTOS through a binary operator, its multiway rendezvous mechanism enables complex topologies to be represented. A characterization of these topologies is provided in [12][60][140]. In fact, the structure is often specified in a resource-oriented style [364] using generalized parallel composition and interleaving operators.

CG-5.a) Roles of a component are merged before being mapped to a process.

UCMs may include multiple components that illustrate different roles of a single entity. For instance, a user can assume the role of a caller or of a callee at different times. This concept is used in Figure 20, where a user can be originator (User:O) or terminator (User:T). In LOTOS, there would be only one process type (`User`) that would integrate both roles. As a consequence, the paths allocated to these roles also need to be integrated in this process (see Guideline CG-6.d)

CG-5.b) If a component has a predetermined interface to the external world, then interaction points and responsibilities are transformed into experiments (values) attached to a gate representing the interface.

The merging of several gates into one, to which experiments are added in order to distinguish between the former gates, is called *gate splitting* [58]. This technique is a well known LOTOS correctness preserving transformation that allows to satisfy both the spirit of Guideline CG-1 as well as constraints imposed by predetermined interfaces. Gate splitting also helps to cope with the addition or the removal of start and end points because it simplifies the modifications to be done on the specification.

As an example, suppose that the Caller component from Figure 22(a) provides a unique interface to the user (environment) called GUI. Then, the interaction points req and sig can be represented by GUI!req and GUI!sig respectively, where the gate corresponds to the name of the interface or communication channel, and where experiment values represent the names of the start and end points.

CG-5.c) LOTOS gates representing UCM interaction points, responsibilities and channels that are not observable by the environment are made internal through the `hide` operator.

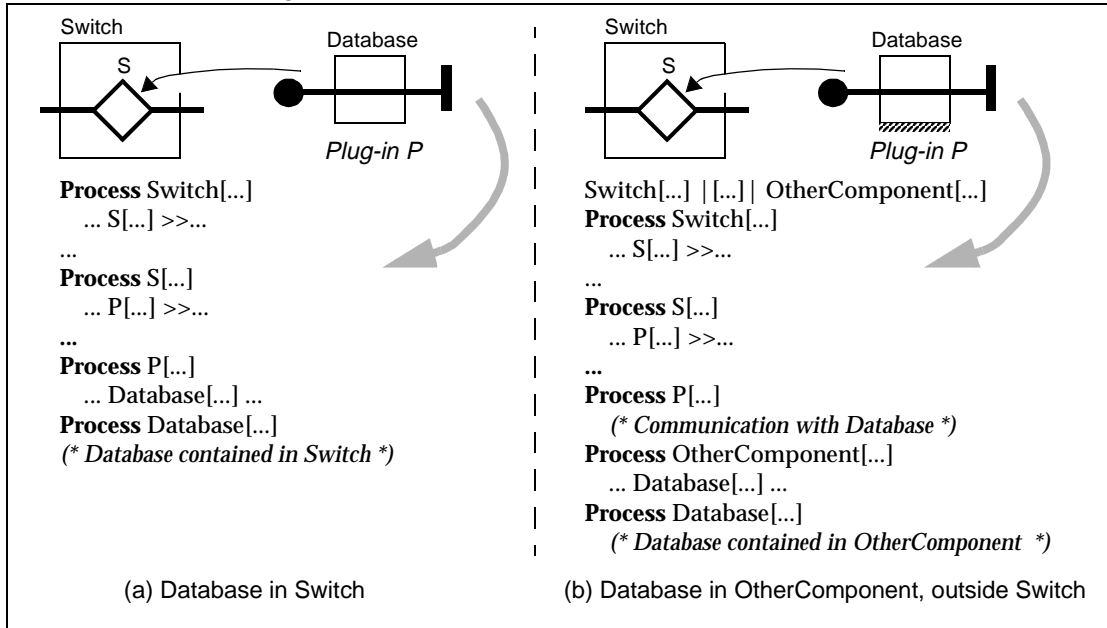
Although the UCM notation does not specify explicitly what is observable and what is not, we can establish conventions or provide supplemental information to make this distinction. One such convention can be inferred from the style guidelines G4 and G7 presented in Section 4.2.2: system start/end points, responsibilities, and waiting places are internal, and so are the channels between communicating entities. Start/end points, responsibilities, and waiting places allocated to actor components (such as Caller and the Callee) are observable. For example, req, sig, and ring belong to actor components and are therefore observable, i.e. they are part of the specification's gate list, whereas vrfy, upd, prbs, and pbs are hidden locally inside the Switch process (Figure 22). Channels Chan1 and Chan2 are hidden at the structure level (this still allows the processes to use them for communication).

CG-5.d) Containment of components is maintained.

If a component, implemented by a process, contains sub-components, then their associated processes are instantiated (and possibly defined) within the former process. If the Switch of Figure 22(a) had sub-components, then their respective processes would be instantiated inside the Switch process. They could also be defined locally inside the Switch process with the LOTOS **where** clause. This would improve separation of concerns and modularity. However, the scoping rules of LOTOS would also lead to the non-availability of the sub-processes in processes representing other components.

Containment can be specified indirectly through stubs and plug-ins. For instance, assuming that the Switch contains a stub S with a plug-in P that contains a component Database (Figure 29(a)), then the corresponding Switch process would instantiate process S, which in turn would instantiate P, which in turn would instantiate the Database process, hence satisfying containment. However, the situation is slightly different for plug-ins that contain *anchored* components, which are declared outside the component that contains the parent stub. Anchored components are already contained elsewhere and would possibly require the use of some communication mechanism to be accessed. Figure 29(b) shows an example where Database is contained in OtherComponent, which communicates with Switch. In all cases, the component structure needs to be consistent, i.e. a component cannot be contained in two disjoint parent components. This needs to be checked at the UCM level.

FIGURE 29. Containment and Plug-Ins



Construction Guideline 6: Integrating Multiple Unrelated Path Segments in a Component

Path segments are *unrelated* when they are not combined explicitly through UCM operators (OR/ AND fork/join, or (a)synchronous path interactions). When combining such segments in a component process, several decisions need to be taken and they must be documented in order to improve traceability and test case selection. The generic case is presented in Guideline CG-6, whereas Guidelines 6.a to 6.d illustrate specific treatments of four families of situations. Note that these situations are not mutually exclusive and hence several may have to be considered simultaneously.

CG-6) If multiple unrelated path segments (possibly from different UCM scenarios, maps or roles) cross one component, then they are integrated together in the corresponding LOTOS process.

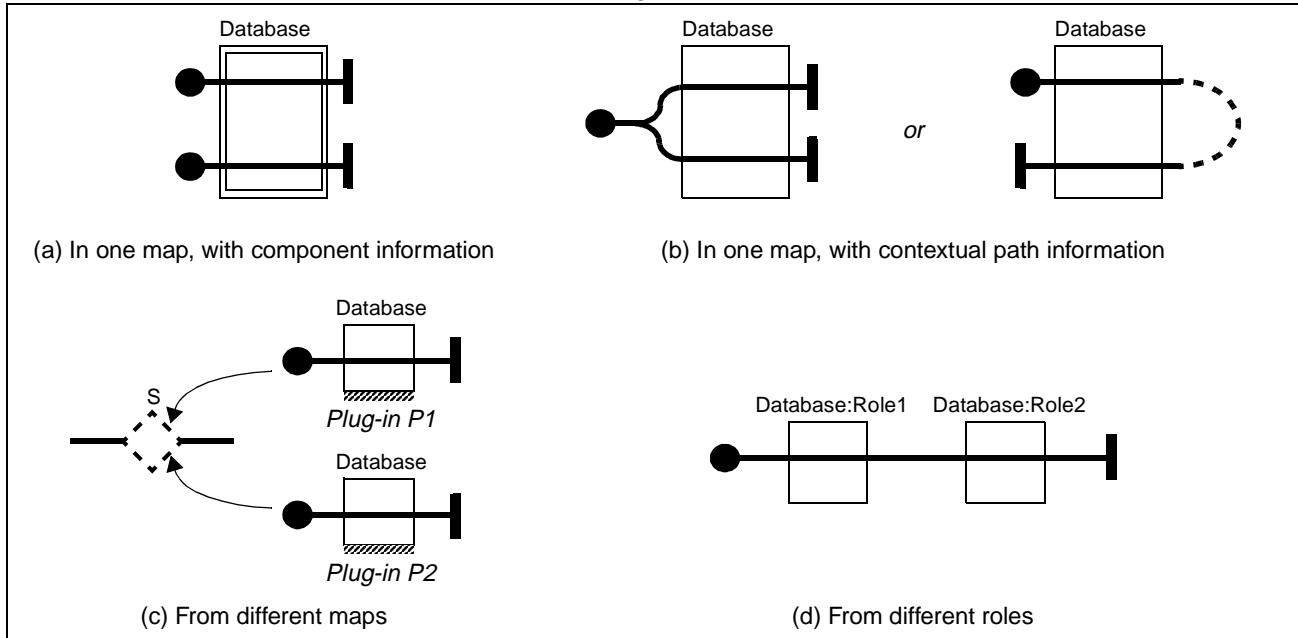
This guideline addresses multiple situations where unrelated path segments need to be composed to form the behaviour of a component process. In general, the integration with the interleave operator ($| | |$) is the most permissive option, but it might however lead to undesirable behaviour (e.g.

global sequences that should be refused) or to issues with respect to non-functional requirements (e.g. resource consumption, or security) which could be addressed in the high-level design. The most restrictive option would be a sequential integration through the action prefix operator ($;$). Again, problems could result due to undesirable behaviour (e.g. unexpected deadlock at validation time) or to issues with non-functional requirements (e.g. performance). Often, the most appropriate level of integration lies in between, using choices ($[]$), generalized synchronizations ($([\dots] |)$), or a combination of all the operators seen so far.

Each of the synthetic construction approaches surveyed in Section 3.3.4 commits to one particular integration solution. The analytic approach used in SPEC-VALUE offers more flexibility for dealing with various situations, at the cost of a manual integration. Four main families of situations are identified in Figure 30, where multiple unrelated path segments cross the Database component:

- Unrelated path segments in one map, with component information: Figure 30(a)
- Unrelated path segments in one map, with contextual path information: Figure 30(b)
- Unrelated path segments from different maps: Figure 30(c)
- Unrelated path segments from different roles: Figure 30(d)

Each of these situations serves as a basis for a more specific construction guideline. In all cases, appropriate documentation, including comments added to the component behaviour, should be provided in order to trace the behaviour constituents to the path segments from which they originate as well as the rationale behind that specific integration. This information can be used later during the selection of validation test cases.

FIGURE 30. Situations with Unrelated Path Segments

CG-6.a) The nature of the component can constrain the potential types of integration.

In real-time systems, components that are passive or that enforce mutual exclusion often have specific limitations, such as not allowing internal concurrency. This can be reflected in a UCM description (see Appendix A: Figures A8 and A9). Components may be protected (enforcing mutual exclusion), declared as passive objects, or declared as stacks to possibly limit the number of concurrent instances or threads.

For example, because they are bound to a protected component (shown with double lines), the two unrelated path segments from Figure 30(a) should not be allowed to evolve concurrently. Moreover, multiple concurrent instances of these path segments should be disallowed as well. One possible solution could be to use the choice operator ($[]$) to integrate these two segments. The Database component could be reinstated upon the termination of any segment, but not before.

CG-6.b) Contextual path information can help to determine an appropriate integration, but it should not be seen as a guarantee.

Contextual path information includes UCM constructs used before (AND-fork, OR-fork, etc.) or after (OR-join, AND-join, sequence, etc.) path segments that are unrelated from the perspective of one particular component. That information can guide the designer in the choice of an appropriate integration. However, in many cases, other concerns or non-functional requirements may dictate the use of a different option.

Figure 30(b) shows two examples where the two path segments crossing the Database are related outside the boundaries of that component. The first example uses an OR-fork, and hence one might conclude that integrating the segments sequentially ($;$) is not judicious, and that an alternative ($[]$) is the best option. In the second example, the bottom path segment follows the top one after an arbitrarily long causal sequence of intermediate responsibilities somewhere outside the Database. One could conclude that a sequential integration ($;$) is sufficient in this case.

However, in both examples, if the Database is intended to handle multiple concurrent requests (performance requirement), then the path segments should probably be integrated using the interleave operator ($| |$). This shows that although contextual information can be of some help, it is not sufficient for deciding the best integration.

CG-6.c) The integration of unrelated path segments from different maps can be influenced by the global context, including selection policies.

Different maps may include path segments crossing one same component. This situation is illustrated in Figure 30(c), where two plug-ins include a reference to the same Database. The path segments inside both anchored components need to be integrated. The way these two maps interrelate may have an impact on the appropriateness of the integration.

Relevant contextual information in this example includes the stub selection policy. For instance, if both plug-ins are intended to evolve concurrently in the stub, then it might be sensible to allow these two segments to evolve concurrently ($| | |$). If the plug-ins are mutually exclusive, then the segments could potentially be integrated using the choice operator ($[]$), and so on.

CG-6.d) The integration of unrelated path segments from different roles need be considered.

Roles represent yet another dimension that needs to be considered when integrating unrelated path segments. Figure 30(d) shows an example where the Database can assume various roles. Whether the component can assume many roles simultaneously, exclusively, or sequentially will again have an impact on the potentially appropriate integrations (e.g. $| | |$ vs. $[]$ vs. $;$). The decision is not necessarily done at the level of the whole component; the integration could be more fine grained. For instance, two roles may be mutually exclusive for one scenario, and they could be assumed simultaneously for another scenario. Another interesting situation is when roles A and B are mutually exclusive whereas roles A and C can be assumed simultaneously.

Construction Guideline 7: Refining Inter-Component Causality

Causality between responsibilities or events located in different components is represented by means of messages, i.e. events resulting from the synchronization of two LOTOS processes representing these components. This guideline is at the core of the problem of MSC generation from UCMs. Due to the complexity of the said problem, these guidelines do not discuss the additional information (e.g. annotations to the responsibilities and inter-component path segments) that UCMs would require in order to automatically derive message patterns from inter-component causality. They rather focus on generic constraints that should be satisfied by any message-based refinement.

Guideline CG-7 captures the essence of the general case, whereas Guideline CG-7.a and Guideline CG-7.b focus respectively on shared responsibilities and direction of messages. The satisfaction of channel constraints are further illustrated in Guideline CG-7.c.

CG-7) Causal paths that span two components may be refined through exchanges of messages.

This guideline implies many design decisions that need to be taken during the construction of the LOTOS prototype. To ensure the causality between the responsibilities and events performed by two different components, message-like interactions are essentially required.

The simplest solution is for the first process to send a single notification message to the second process. The nature of this message and of its value experiments (parameters) depends on the information available to the designers. For instance, if the protocol to be used is already known, then the message can be concrete, otherwise it will be abstract or synthetic.

More complex exchanges of messages could be used if the protocol requires it (e.g. three-way handshake, remote procedure call, etc.). The specification could abstract from the details of the processing implied by such complex protocols while retaining the message patterns.

Sent messages are represented as gate experiments, similarly to Guideline CG-5.b. In Figure 22(c), the Caller sends a request message to the Switch (`Chan1 !request !calleeNum`) in order to refine the `<req, vrfy>` inter-component causal sequence. It also receives an announcement (`Chan1 ?ann:Announcement`) that refines the `<pbrs, sig>` and `<pbrs, sig>` causal sequences.

Symmetry is enforced in synchronized actions. For instance, in Figure 22(c), all the actions involving `Chan1` in the Switch process must be reflected in the Caller process, with value experiments on which both parties can agree.

CG-7.a) Shared responsibilities are refined through negotiations (exchanges of messages).

Shared responsibilities (Figure A7(d) in Appendix A:) act like inter-component causal paths, but they explicitly imply the existence of a complex communication mechanism (e.g. multiple

exchanges of messages satisfying some constraints, protocols, or negotiation rules) that goes beyond simple notification messages. This mechanism may be part of the documentation that accompanies the UCM.

CG-7.b) Direction of messages is made explicit when necessary.

LOTOS synchronization is directionless, hence interpreting events as being directional requires contextual or additional information. This is essential for the generation of MSCs from UCMs. Some gates may be interpreted as linking only two components in a specific direction, hence solving the issue. Ambiguity in the direction of a message may also be solved by analyzing the message sent through a bidirectional gate. For instance, the `request` message sent via `Chan1` in Figure 22(c) could imply that the sender is the `Caller` and the receiver is the `Switch`.

However, as the specification gets more complex, a more explicit definition of the direction of a message becomes unavoidable. The source and the destination of a message are then added to the message itself as new experiments. This solution also helps coping with issues related to multiple instances of components. For example, if many callers and switches are involved in a system, then the different types of message could be:

```
Channel ?source:CallerId ?dest:SwitchId ?msg:Message
```

and

```
Channel ?source:SwitchId ?dest:CallerId ?msg:Message
```

The assumption here is that channel names are unique, and that we know the types of components they are linking.

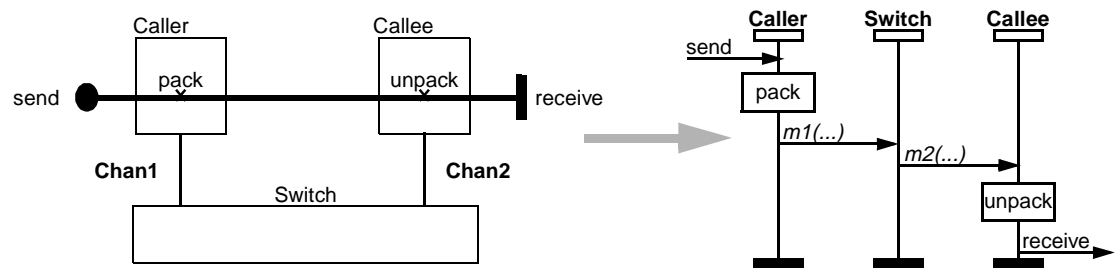
CG-7.c) Communication constraints imposed by pre-determined channels need to be satisfied.

Component architectures may contain pre-determined communication channels that act as constraints on the valid message exchanges. Such constraints have three implications:

- Inter-component causality between components A and B that are not linked by a channel will require the participation of intermediate components (e.g. C, or C and D).
- The communication will be established between the entities connected by channels (e.g. A with C, and C with B).
- The forwarding of messages in these intermediate components need to be integrated to the rest of their behaviour (in a way similar to Guideline CG-6).

Figure 31 illustrates such a situation. The Caller and the Callee are not allowed to exchange messages directly because they are not connected by a communication channel. Even if the path goes directly from the Caller to the Callee, the Switch needs to be involved in any communication between these two entities. One potential message exchange that is valid is shown on the right in the form of an MSC. The messages $m1$ and $m2$ may be supplemented by information related to the sender, the receiver, and other relevant parameters.

FIGURE 31. Channel Constraints and Valid Message Exchanges



In this particular example, the behaviour of Caller needs to include the sending of $m1$ over Chan1, and that of Callee should accept $m2$ from Chan2. The Switch is required to receive $m1$ from Chan1 and then to send $m2$ over Chan2, even if no path segment crosses that component.

Again, this guideline is not self-contained and it can be coupled to the other guidelines seen in this section (e.g. to handle more complex protocols or direction of messages).

5.2.4 Construction Guidelines for Data

Construction Guideline 8: Representing Data

Although seldom defined explicitly in UCMs, data types and related operations need to be defined in LOTOS.

CG-8) Abstract data types are used to describe identifiers, operations, conditions, and databases.

UCMs often assume the existence of a data model that needs formalization in order to represent the following elements:

- **Identifiers:** enumerated types (explicit or based on the *natural number* data type) are used to describe simple message names, to specify the direction of messages, and to distinguish between multiple instances of a component. The LOTOS standard already provides basic types (booleans, natural numbers, sets, etc.) and mechanisms to manipulate and extend them.
- **Operations:** operations with parameters can be used to specify tuples (e.g. messages with parameters, user profiles, etc.) and rewrite rules to manipulate them or to extract information from them.
- **Conditions:** conditions, used in LOTOS in guards and selection predicates, make use of comparison operators (based on the *boolean* data type) which evaluate to true or false.
- **Databases:** passive objects and pools are often represented as databases, which are specified using tuples, sets of tuples, sets of sets of tuples, and so on. LOTOS offers a *set* data type with many predefined, extensible and modifiable operations.

In the example of Figure 22, messages are identified by values on gates Chan1 and sig, and the Switch process would require the use of conditions to select the appropriate branch of the OR-fork. The required abstract data types would be defined before the **behaviour** clause.

5.2.5 Towards Partial Automation

Automatically generating a complete LOTOS model from UCMs is a very difficult task, as was explained in the discussion on synthetic construction approaches (Section 3.3.4). This is why an analytic approach is used in SPEC-VALUE. However, even an analytic approach does not prevent the partial automation of some of the guidelines explored here. Such partial automation would be a way to improve the maturity of SPEC-VALUE to the third level (Defined, or Transitional-Assisted) on the Formal Specifications Maturity model scale (Section 9.1.3)

Table 14 provides an overview of the potential for automation of the eight major guidelines defined in this chapter. This brief evaluation is based on the experience of the author with a prototype compiler (unbound UCMs to LOTOS [13]) and with the manual generation of various specifications from UCMs, accompanied by numerous discussions on this topic. Given the current status of the UCM notation and supporting tools, a degree of automation difficulty is provided for each guideline, from simple (1) to difficult (5). Additional information elements that would be required to improve the situation are enumerated, but the format in which this information could be provided remains outside the scope of this thesis.

TABLE 14. On the Partial Automation of Construction Guidelines

	Construction Guideline	Automatable?
Paths	1. Interaction points and responsibilities	<ul style="list-style-type: none"> • Degree of difficulty: 2 (all guidelines). • More precise information on the visibility (what is observable) and on interfaces is required. • Pre/post conditions and value identifiers need to be expressed in a way that is mappable to LOTOS abstract data types and expressions.
	2. Causal paths	<ul style="list-style-type: none"> • Degree of difficulty: 2 (all guidelines). • The level of recursion and of concurrency of scenario paths needs to be described. • Loop detection can be a problem, but an explicit UCM loop construct would alleviate it (such a construct is now being supported by the UCM Navigator tool).
	3. Stubs and plug-ins	<ul style="list-style-type: none"> • Degree of difficulty: 3 (all guidelines). • Skeletons can be generated for stub and plug-in processes, instantiations and bindings. • Selection policies however need to be described in a precise language from which corresponding LOTOS behaviour expressions can be generated.
	4. Other path elements	<ul style="list-style-type: none"> • Degree of difficulty: 2 (Guidelines 4.a, 4.b, and 4.c). • More precise information on the visibility of timeout, abort, and failure events is required. • Degree of difficulty: 4 (Guideline 4.d). • The handling of dynamic responsibilities, slots, and pools is quite remote from LOTOS operators. Precise information on the exact nature and impact of the dynamic responsibilities is required.
Structure	5. Structure	<ul style="list-style-type: none"> • Degree of difficulty: 3 (all guidelines). • More precise information on channels (interfaces) is required. • Consistent containment relationship needs to be ensured. • Arbitrary structures are sometimes impossible to capture directly with the LOTOS binary synchronization operator, and additional transformations are then required.
	6. Unrelated path segments	<ul style="list-style-type: none"> • Degree of difficulty: 5 (all guidelines). • Precise definitions of role attributes, map compositions and role compositions are required. • Consistency with contextual path information and selection policies needs to be enforced. • The integration can be influenced by requirements outside the scope of UCMs.
	7. Inter-component causality	<ul style="list-style-type: none"> • Degree of difficulty: 4 (all guidelines). • Precise definitions of channels, protocols and negotiation patterns (with their message names), and of their parameters are required. • Data flows need to be defined (possibly attached to UCM paths). • Routing information is required when causality can be refined through several routes.
Data	8. Data	<ul style="list-style-type: none"> • Degree of difficulty: 4 (all guidelines). • A precise data model, compatible with LOTOS abstract data types, is required. • Most definitions for databases, pools, and message encoding would need to be provided by designers and requirements engineers.

5.3 Applying the Construction Guidelines to TTS

The construction process used here starts by analyzing the TTS UCM model and its components to determine an appropriate specification structure. Then, data structures and operations for identifiers, messages, and databases are defined using ADTs. Finally, the behaviour of each component is described in a process. Sub-processes for stubs and plug-ins may be needed along the way. These three steps are usually interleaved as the description of component processes may unveil the need for new or different ADTs.

While constructing the prototype, the LOTOS designer makes choices concerning the mapping of components, stubs, plug-ins, causal paths, path elements, messages, etc., according to the construction guidelines. Traceability links between the UCM model and the LOTOS model are established when these design decisions are taken. Since UCMs are used at a higher level of abstraction than LOTOS, this analysis gives an opportunity to inspect the UCM documentation and to detect missing parts, contradictions, or other such problems.

The complete specification of TTS can be found in Appendix B. Several parts will be presented or referenced in the following sections.

5.3.1 Structure of the TTS Specification

The integrated TTS UCM (Figure 20) shows that there are three types of components required: User, Agent, and OCSlist. Different instances of User and Agent, for originating and terminating roles, are also involved.

According to Guideline CG-5, process definitions are needed for User and Agent. Two process definitions are sufficient because originating and terminating roles will be merged (Guideline CG-5.a). Since the OCSlist passive object is included in Agent and since it essentially represents a database, the OCSlist will be specified as a process parameter (an ADT) rather than as an independent process (Guideline CG-8). Guideline CG-5.d does not apply because there is no longer any embedded component.

User and Agent do not have a predetermined interface to the external world, hence Guideline CG-5.b is not used. The visible actions will include the start points and end points, namely req, ring, sig, and disp. Gates are defined for these points (Guideline CG-1), whereas the other start points and end points will be used internally for the binding of plug-ins to stubs. Responsibilities are assumed to be internal in this model, so they will not be part of component interfaces (i.e. gate parameters).

Guideline CG-5 suggests the creation of communication channels between communicating entities in the absence of explicit ones. A specific user will always communicate with the same agent, so using the users interface gates as channels is appropriate (in this system, the modelling of agents is more important than the modelling of users). However, an agent may communicate with any other agent. A new communication channel is hence required. The corresponding gate, named `agent2agent`, will be hidden (Guideline CG-5.c). In order to allow any agent to communicate to any other agent, a communication medium (the process `Medium`) is added.

To make the prototype more flexible, the specification will allow the dynamic instantiation of user-agent pairs. These instances will possess their own identifiers and internal data, which will be initialized at the beginning through an additional gate: `init`. As a result, each test scenario will be able to provide an initial configuration composed of users and agents, with appropriate values for their parameters (e.g. subscribed features). This leads to a less rigid approach to validation than having a fixed configuration, which often results in the creation of multiple specifications for the coverage of many variations and cases.

According to the decisions taken so far, the structure can be expressed by the following process definitions and compositions (lines 414 to 448 in Appendix B):

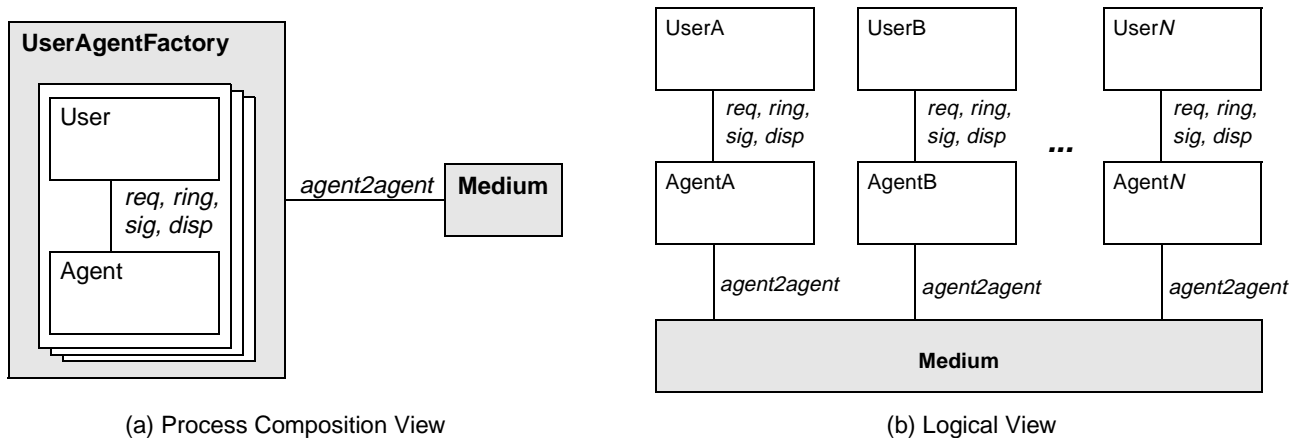
```

414  behaviour
415
416      (* Gates not visible to the users are set to be internal. *)
417  hide
418      agent2agent  (* Inter-agent communication channel *)
419  in
420      (
421          (* We create as many user/agent pairs as necessary. *)
422          UserAgentFactory [req, ring, sig, disp, init, agent2agent]
423          |[agent2agent]|
424          (* Agents communicate through some medium. *)
425          Medium[agent2agent]
426      )
427
428  where
429
430      (*****
431      (* Process UserAgentFactory: To create and initialize users and agents. *)
432      (*****
433
434      process UserAgentFactory [req, ring, sig, disp, init, agent2agent]:noexit :=
435          init ?userId:User ?userFeatures:FList ?OCSlist:UserList ?state:UserState;
436          (
437              (* Create the user and its associated agent *)
438              (
439                  User [req, ring, sig, disp] (userId)
440                  |[req, ring, sig, disp]|
441                  Agent [req, ring, sig, disp, agent2agent]
442                      (info(userId, userFeatures, OCSlist), state)
443              )
444              |||
445              (* Prepare to accept new creation requests *)
446              UserAgentFactory [req, ring, sig, disp, init, agent2agent]
447          )
448      endproc (* UserAgentFactory *)

```

The `UserAgentFactory` process enables the instantiation of user-agent pairs in a recursive way. Each user can communicate with its agent, but agents can communicate with each other only through the medium. The nature of the medium can be of many kinds. In the TTS example, the medium is specified as a bidirectional FIFO channel of length 2 (lines 451 to 473 in Appendix B).

The process composition is illustrated in Figure 32(a). Note that some groupings of components in the specification structure differ from those of the abstract component structure found in the integrated UCM. This is because LOTOS processes are composed using *binary* operators. However, the resulting groupings and possible ways of establishing communications are semantically equivalent (Guideline CG-5). The logical view of how the components communicate is best shown in Figure 32(b).

FIGURE 32. Structure of the LOTOS Specification

(a) Process Composition View

(b) Logical View

5.3.2 TTS Data Types and Operations

The above component structure required different parameters for the processes: user identifiers, lists of features, OCS screening lists, states, etc. Responsibilities are also likely to modify these values. Appropriate abstract data types are needed to support these data structures and their operations, as suggested by Guideline CG-8.

The LOTOS standard provides a library of common ADTs, some of which were included and adapted in the specification of TTS (lines 19 to 169 in Appendix B). Booleans, natural numbers, elements, and sets are at the basis of all other ADTs defined for the support of TTS data structures and operations. This is achieved through mechanisms like inheritance, renaming, and actualization.

The ADTs specific to TTS (lines 170 to 361 in Appendix B) include:

- **Identifiers:** State (busy or idle), Announcement (callDenied, busySig, ringBack, etc.), User (userA, userB, userC), and Feature (BC, CND, OCS). The type `Direction` (line 352) is also utilized to identify the direction of a message sent to or received from the medium (Guideline CG-7.b). Note how this type uses the renaming capabilities of LOTOS to simplify its definition.

- **Operations:** Most types include operations for comparing values such as `eq` (is equal to) and `ne` (is not equal to). For enumerated types with many elements (e.g. `Feature`), the operation `map` is used to map elements to natural numbers. This enables the reuse of operations defined for natural numbers as well as additional flexibility in adding elements to the type or removing elements from the type. The ADT `UInfo` (line 317) uses operations to represent records and to extract information for these records.
- **Conditions:** Throughout the specification, conditions make use of comparison operators in guards (e.g. lines 519 and 528) and selection predicates (e.g. line 539).
- **Databases:** the `OCSlist` passive object is represented as a set of users (type `UserList`, line 251), to which additional operations were added (type `OCScheck`, line 260). Another database is used by each agent to maintain the list of subscribed features (type `FList`, line 308). Many operations on databases are automatically inherited from the standard `Set` ADT.

ADTs are also defined for stub entry points and exit points (lines 362 to 407 in Appendix B). Identifiers (type `StubPath`) and databases (type `SPList`) are used in the binding of plug-ins to stubs, as suggested by Guideline CG-3.c.

5.3.3 TTS Process Definitions

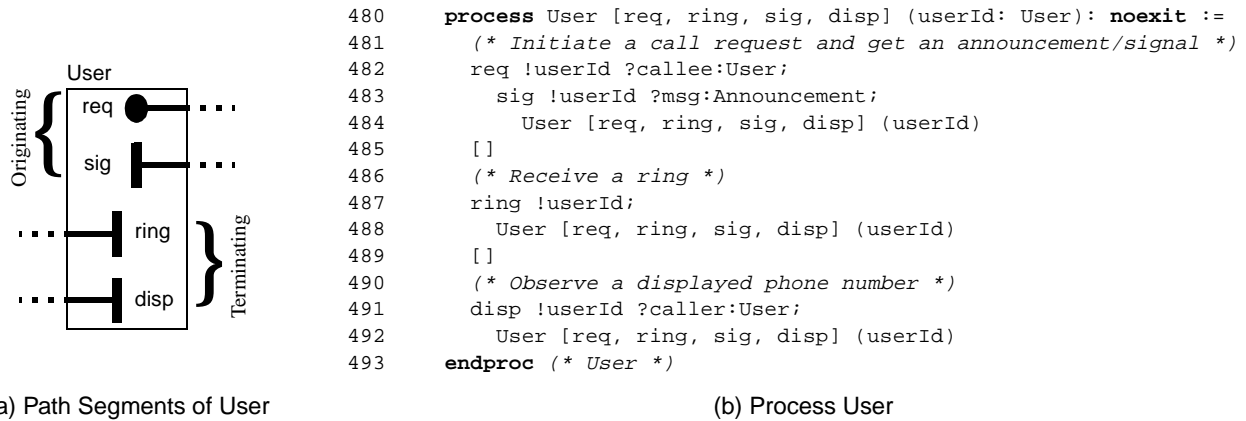
In this section, the construction guidelines are used to build process definitions for the User and Agent components.

Process User

The process `User` is relatively simple as it considers only four short segments (Figure 33(a)) coming from the originating and terminating roles found in the root map and in the `CND` plug-in. Both roles are merged according to Guideline CG-5.a, and their respective path segments are integrated as explained in Guideline CG-6 and Guideline CG-6.d. Figure 33(b) shows the resulting process definition: the `req` and `sig` points are integrated sequentially (as implied by the context, see Guideline CG-6.b) whereas the `ring` and `disp` segments are integrated as alternatives. The `userId` process parameter

enables the instantiation of multiple users that are unique and that can realistically assume both roles at the same time. The process reinstatiates itself recursively in order to handle the next events.

FIGURE 33. Construction of Process User



(a) Path Segments of User

(b) Process User

Process Agent

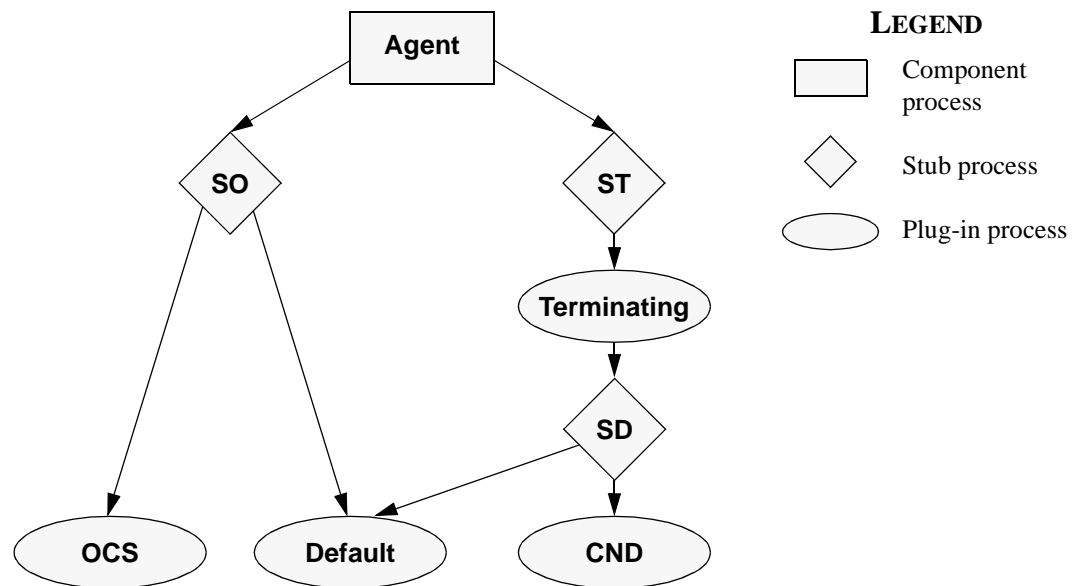
The process `Agent` is more complex as it considers more path segments as well as inter-component causality relationships. This process can be found between lines 496 and 646 in Appendix B. `Agent` includes the `OCSlist`, as suggested by Guideline CG-5.d, but as a process parameter instead of as a sub-process. The six responsibilities bound to the `Agent` component (`chk`, `pds`, `vrfy`, `pbs`, `prbs` and `upd`) are represented as LOTOS gates (Guideline CG-1) and are locally hidden (Guideline CG-5.c). They come from both the originating and terminating roles, which are again integrated in one process definition (Guideline CG-5.a). Several responsibilities, such as `prbs`, `pbs`, and `pds`, affect the value of the `sgnal` returned to the originator, while `upd` updates the terminator's status (Guideline CG-1.a).

One interesting difference between the `User` component and the `Agent` component is that agents contain stubs, which in turns contain plug-ins. According to Guideline CG-3, stubs are defined locally as sub-processes. This is effectively the case for processes `SO` and `ST` (lines 582 to 644). Representing a dynamic stub, the stub process `SO` also specifies the selection policy (Guideline CG-3.b). Conditions (guards) are used to provide priority to `OCS` over the default behaviour; if the user has

subscribed to OCS, then the OCS plug-in process is instantiated, otherwise the default plug-in process is instantiated.

Guideline CG-3 also mentions that plug-ins are mapped to independent processes, therefore process definitions are provided for `Default`, `Terminating`, `OCS`, and `CND` (lines 648 to 776). Note that the `Terminating` plug-in also contains a stub (`SD`) specified as a sub-process in `Terminating`. The resulting process calling tree for the agent component is illustrated in Figure 34. The `Default` plug-in process can be instantiated by two stub processes, namely `SO` and `SD`, so the consistency and reusability found at the UCM level are reflected in the LOTOS prototype.

FIGURE 34. Process Calling Tree for the Agent Component



Having processes for the plug-ins simplifies the integration of path segments in the `Agent` process. The latter only has to cover the path segments found in the root map. These segments are integrated according to Guideline CG-6.c. Alternatives are used to integrate most segments, except for the two segments coming out of stub `ST` found in the terminating role: its plug-in (`Terminating`) contains an AND-fork which can lead to a behaviour where both segments are active simultaneously. Therefore, the interleave operator (`| |`) is used to integrate these two segments (line 563).

The process `Terminating` (Figure 35) represents a good illustration of how causal paths are represented inside a component. The OR-fork found in the `Terminating` plug-in (Figure 20) is mapped to the LOTOS choice operator (line 694), as prescribed by Guideline CG-2.a. Both alternatives are guarded by conditions representing those found on the UCM path segments. The AND-fork found in the same plug-in is represented by the `interleave` operator (line 689), as suggested by Guideline CG-2.b. The OR-join in the `Terminating` plug-in is captured by the `exit` operator (lines 686 and 698), which outputs `out4` as the end point to be connected to an outgoing segment in the calling stub. This case of duplicated behaviour concords with what was explained in Guideline CG-2.c.

FIGURE 35. Extract from the `Terminating` Plug-in Process

```

677 [state eq idle] ->
678   (
679     (
680       SD[disp](Insert(in3, EmptySPList), ui, state, user0)
681       >>
682       accept ui:UInfo, state:UserState, user0:User, piep:SPList
683       in
684         [out5 IsIn piep] ->
685         upd; (* Updates the busy status *)
Guideline CG-2.c → 686         exit (ui, busy, any Announcement, user0,
687             Insert(out3, Insert(out4, EmptySPList)))
688     )
Guideline CG-2.b → 689   |||
690   prbs; (* Prepares the ringBack signal *)
691     exit (ui, any UserState, ringBack, user0,
692         Insert(out3, Insert(out4, EmptySPList)))
693   )
Guideline CG-2.a → 694 []
695 [state eq busy] ->
696   (
697     pbs; (* Prepares the busy signal *)
Guideline CG-2.c → 698     exit (ui, state, busySig, user0, Insert(out4, EmptySPList))
699   )

```

Binding Plug-ins to Stubs

Each stub process is responsible for calling plug-ins with parameters such that the stub/plug-in binding relationship is preserved. This is achieved with the help of input/output segments specified by the `StubPath` and `SPList` abstract data types. Figure 36 illustrates this idea, corresponding to Guideline CG-3.c, by showing how the stub process `SO` uses the plug-in process `OCS`.

The first parameter of OCS is `in1` (line 595), which corresponds to the plug-in start point triggered by the stub incoming segment of the same name (Guideline CG-3.a). The remaining parameters are internal values of the `Agent` process, which can be used and modified by the plug-in process. OCS exits with another list of parameters whose last element, `piep` (stands for *plug-in end points*, line 599), is the list of end points reached by the plug-in. The elements of this list are then bound to their respective stub output segment (lines 602 to 606), hence completing the implementation of the binding relationship.

FIGURE 36. Binding of a Plug-in to a Stub in Process SO

```

594  (* First in1 parameter is the plug-in start point *)
595  OCS[chk, pds](in1, ui, state, userT)
596  >>
597  (* Connect the resulting end points to the stub exit paths *)
598  accept ui:UInfo, state:UserState, msg:Announcement, userT:User,
599         piep:SPList
600         (* piep is the resulting list of plug-in end points *)
601  in
602    [out1 IsIn piep] ->
603      exit (ui, state, msg, userT, Insert(out1, EmptySPList))
604    []
605    [out2 IsIn piep] ->
606      exit (ui, state, msg, userT, Insert(out2, EmptySPList))

```

For plug-ins that contain multiple start points that can be activated simultaneously, a list of start point names (type `SPList`) should be used instead of a single start point name (type `Stub-Path`).

Inter-Component Causality

The last topic to be addressed for the construction of the `Agent` process is inter-component causality. Because there are paths crossing the originating and terminating roles, agents need to communicate with each other in order to support this causality.

The path that goes from the `Agent:O` to `Agent:T` in Figure 20 is used here as an example. Guideline CG-7 states that inter-component causality needs to be refined through exchanges of messages. The path of interest can be refined as a message sent from the originating agent to the terminating

agent. This message essentially forwards the call request, hence it will be named `request`. An agent communicates with the medium through gate `agent2agent`. One possible message is:

```
522 agent2agent !toMedium !uid(ui) !userT !request;
```

The direction of the message (`toMedium`) is made explicit through gate splitting, as prescribed by Guideline CG-7.b. The source (`uid(ui)`) and destination (`userT`) are also included as identifiers of sort `User`.

The terminating agent receives a request in the same way. Note that the originating user is not known in advance, therefore a question mark is used to get its identifier. Note also that the direction of the message has been changed to the opposite direction, i.e. `fromMedium`.

```
546 agent2agent !fromMedium ?userO:User !uid(ui) !request;
```

The medium receives the first message from the originating agent and forwards it to the appropriate terminating agent. Guideline CG-7 mentions that symmetry should be enforced in synchronized actions. Since both agents are synchronized with the medium, the latter needs to support both types of events, with parameters of the same sorts and in the same order. Indeed, the process `Medium` possesses such events (e.g. lines 459 and 461 in Appendix B):

```
459 agent2agent !toMedium ?from:User ?to:User ?msg:Announcement;  
461 agent2agent !fromMedium !from !to !msg;
```

The behaviour of the medium is also inferred by the forwarding of messages between agents, as suggested in Guideline CG-7.c. Other types of messages are used to implement the inter-component relationship from terminating agents to originating agents.

5.4 Chapter Summary

This chapter proposes an analytic approach for the construction of LOTOS specifications from Use Case Maps. Section 5.1 provides an overview of the approach in the context of the SPEC-VALUE methodology. It recalls why LOTOS is appropriate as a specification language for the prototyping of telecommunications systems described with UCMs. This section also discussed the reasons why TMDL is unfit as a language for the synthesis of specifications for complex telecommunications sys-

tems. An analytical approach to the construction of specifications appears to be a more practical solution.

The core of this chapter is found in Section 5.2, where three families of construction guidelines are presented (for paths, structures, and data) and where several are illustrated individually. A total of 8 general guidelines and 22 small-grained guidelines cover many important topics such as: interaction points and responsibilities (2), causal paths (5), stubs and plug-ins (4), other path elements (4), structure and components (5), unrelated path segments (5), inter-component causality (4), and data (1). The application of these guidelines results in a LOTOS specification that is close to the UCM model. Using this construction approach, the translation of telecommunication features described with UCMs is systematic.

As discussed in Section 5.2.5, construction guidelines for UCM structures are the most complex to automate because designers and requirements engineers have to choose among numerous possible solutions. Some of these decisions could be embedded in the UCM description, but this would require further formalization of the notation and of annotations. How best to capture these decisions in UCM terms is outside the scope of this thesis and is left as a topic for future work.

The guidelines are applied to the Tiny Telephone System example, and the resulting LOTOS specification (found in Appendix B) is discussed in Section 5.3. Particular emphasis is put on the structure, on data types and operations, and on the definition of processes for users and agents. The complexity of the `Agent` process requires the use of many guidelines related to stubs, plug-ins, and inter-component causality.

The next chapter will discuss the validation of this prototype against the UCMs and the requirements, as well as a testing framework based on UCMs and LOTOS.

Contributions

The following items are original contributions of this chapter:

- Partial illustration of Contribution 1 (Section 1.4.1) regarding fast prototyping in SPEC-VALUE.
- Partial illustration of Contribution 2 (Section 1.4.2) regarding the guidelines for the construction of LOTOS specifications from UCMs in SPEC-VALUE.
- Illustration of step ④ in SPEC-VALUE, i.e. from UCMs to LOTOS.
- 8 general construction guidelines and 22 small-grained guidelines for the generation of LOTOS prototypes from UCMs.
- Brief evaluation of the potential for automation of these guidelines.
- Construction of a LOTOS prototype for the Tiny Telephone System.

CHAPTER 6

UCM-LOTOS Testing Framework

Examining a program to see if it does not do what it is supposed to do is only half of the battle. The other half is seeing whether the program does what it is not supposed to do.

Glenford J. Myers, 1979

LOTOS prototypes can be constructed from Use Case Maps, as seen in the previous chapter. Because an analytic construction approach represents a major step where important design decisions are made, some of which are prone to mistakes, it is essential to verify the target LOTOS model against the source UCM model. The current chapter addresses this issue by presenting a novel framework for validating requirements and designs described with Use Case Maps and prototyped in LOTOS. The concepts behind this framework are founded on the LOTOS testing theory and on new test selection strategies based on UCMs.

The testing approach is first placed in the proper context with respect to the SPEC-VALUE methodology (Section 6.1). Basic concepts related to the UCM-LOTOS testing framework, including a new validation relation, are introduced in Section 6.2. Test goal selection techniques based on a UCM-oriented testing pattern language are developed in Section 6.3, and Section 6.4 complements the testing patterns with additional strategies for the generation of test cases. These techniques are applied to the TTS system example, and the results are presented in Section 6.5. A summary of this important chapter is found in Section 6.6.

6.1 Testing Approach in SPEC-VALUE

Testing is one technique among many others used for the validation and verification of systems. The main motivations behind this strategic choice for V&V are recalled in Section 6.1.1, whereas Section 6.1.2 presents the context in which testing is used in the SPEC-VALUE methodology. Section 6.1.3 contrasts the concepts of validation testing (used in SPEC-VALUE) and conformance testing.

6.1.1 Justification for a Testing Approach

Many validation and verification concepts and techniques, with a special focus on those related to LOTOS, were introduced in Section 2.3.7 and Section 3.4. A particular attention was given to some of the most popular ones, namely step-by-step execution, equivalence checking, model checking, and testing. When compared to the three other approaches, testing appears to be the most practical and suitable technique for verifying that a LOTOS specification respects the intent of the UCMs from which it was constructed:

- With appropriate tool support, checking a LOTOS specification against a test case requires less effort than covering the same traces with step-by-step execution. Many of these traces are caused by the interleaving semantics of LOTOS combined with non-determinism and internal actions in the specification under test. For one test case, a tool such as LOLA automatically covers all the non-deterministic traces where internal actions are interleaved. Moreover, a test suite can be checked in batch, a useful feature for verifying the specification each time it is modified. Step-by-step execution is laborious to use for verifying realistic telecommunications systems due to the numerous and lengthy traces to cover.
- Testing requires the presence of only one formal model to test, and test cases can be derived or generated systematically from requirements or scenarios, whereas equivalence checking usually requires two formal models to be compared. Equivalence checking is hardly useful in SPEC-VALUE (and in the early steps of the design process in general) because the construction approach aims to produce a first high-level specification from informal requirements and semi-formal scenarios. Hence, testing appears to be more appropriate.

- The requirements addressed in this thesis are expressed and captured mostly operationally as (UCM) scenarios. Test cases are small-grained properties defined at a level similar to the one offered by scenarios, whereas large-grained liveness and safety properties, for which model checking is most useful, are often difficult to extract or infer from scenario descriptions (Section 3.4.1). Again, testing appears to be the most appropriate approach in our context.

Note however that these approaches are not mutually exclusive but complementary. SPEC-VALUE focuses on testing because testing is unavoidable in any design process [51][68], but this does not prevent the use of additional V&V techniques. Provided sufficient resources, the other techniques can be used beyond testing:

- Step-by-step execution can be useful to debug a specification declared faulty, possibly by an unexpected verdict resulting from the execution of a test.
- Equivalence checking can be used when refinements or improvements are done at the LOTOS level only.
- Model checking can be used to cover additional properties, when they are available.

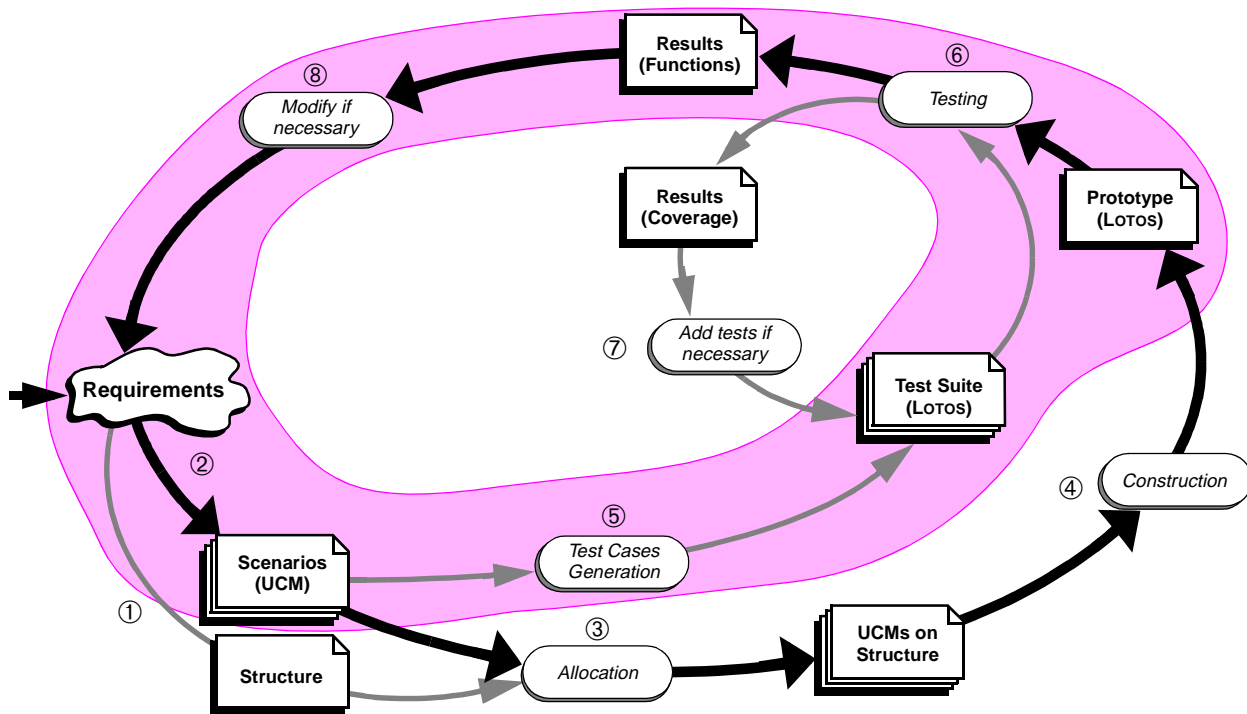
Despite the potential usefulness of a multi-technique approach to validation and verification, this thesis limits its scope to the testing approach about to be presented in this chapter.

6.1.2 Testing in SPEC-VALUE

SPEC-VALUE's testing framework is illustrated in Figure 37. In a nutshell, LOTOS test cases are generated manually from UCMs (step ⑤). Test selection strategies make use of (unbound) UCMs to extract abstract sequences, and the presence of an underlying component structure is optional. These abstract sequences are transformed into LOTOS test processes. The testing itself is performed by composing the test cases with the LOTOS prototype (step ⑥). This operation is performed automatically with LOLA, which then outputs the resulting verdict for each test. If a verdict is not satisfactory, then appropriate modifications might be brought to the requirements (step ⑧), which may result in cascading modifications to the scenarios, the tests and the prototype. In many cases however, the required modi-

fications will be more localized (e.g. to the prototype or to the tests), without affecting the requirements. Coverage measurements (step ⑦) will be the topic of the next chapter.

FIGURE 37. UCM-Based Testing with SPEC-VALUE



Although the term *testing* is used most commonly in the sense of implementation testing, executable specifications and formal prototypes can also be tested in order to see whether they satisfy requirements. The latter is essentially a validation activity, yet many of the methods and concepts used in implementation testing also apply. In SPEC-VALUE, testing the prototype/specification is really intended to be *validation testing*. In this context, the methodology helps to validate a LOTOS prototype against its requirements by verifying (through testing) that this prototype corresponds to the UCMs from which it was derived. This is essentially the research hypothesis discussed in Section 1.2.

An important assumption here is that UCMs capture the functional requirements correctly. Users, requirements engineers and designers can all *inspect* the UCMs derived from informal require-

ments, thus establishing their validity. UCMs are defined at a level of abstraction that is efficient for early inspection of the system design. Inspection is known to be a very cost-effective quality improvement technique, especially for requirements documents [218]. Unlike inspection, testing requires an executable or formally defined artifact, which is, in SPEC-VALUE, the LOTOS prototype resulting from the construction step. Testing acts as an essential supplement to inspection for detecting behavioural problems. It also improves the level of confidence in the validity and the consistency of the different models involved.

There is an apparent circularity issue in the testing cycle of Figure 37. If the LOTOS prototype and test cases are both correctly generated from the same UCMs, then one could think that the test suite should not detect any problem. However, we believe this is seldom the case (especially as the complexity of the system increases) and several experimentations in Chapter 8 show that problems can indeed be detected. Constructing a component-based LOTOS specification from UCM involves several design decisions caused by a more detailed level of description and by the transformation of end-to-end scenarios into component behaviour. These decisions may introduce unexpected errors in the specification. However, when deriving functional test cases from end-to-end scenarios, many of these decisions need not be taken. Hence, functional tests are closer to the UCMs and more likely to be correct than a component-based specification that integrates all scenarios. Testing the specification against these functional test cases can therefore lead to the detection of errors and to a higher level of confidence when errors are no longer detected. Also, since we assume that UCMs capture the requirements correctly, functional testing based on UCMs becomes a validation technique applicable to LOTOS prototypes as well as detailed design models and, possibly in the future, implementations.

6.1.3 Validation Testing and Conformance Testing

The validation testing offered by SPEC-VALUE is different from the conventional conformance testing introduced in Section 3.4.2 and standardized in the Conformance Testing Methodology and Framework (CTMF) [193]. Conformance testing usually requires a model (e.g. a formal protocol specification) from which a black-box test suite is derived and then used to test an implementation or another model. Test suites are generally abstract (e.g. not necessarily defined in terms of user requirements),

they are defined in a specification language with operational semantics (e.g. TTCN), and they target artificial coverage criteria defined in terms of the source model. In SPEC-VALUE, the test suite is derived from informal requirements and semi-formal scenarios (the UCMs), and the goal is to create and check the first formal model (the LOTOS specification). This test suite, composed of test cases whose purposes can be related to the requirements, is used to validate the model against the requirements, hence the term validation. Conformance testing can be used at a later stage of the design cycle, when an implementation is required to be declared conformant to the formal model.

The Formal Methods in Conformance Testing (FMCT) framework [196] was an effort to integrate new techniques based on several formal methods (SDL, Estelle and LOTOS) to the context of conformance testing. Among many others, concepts and techniques adapted to LOTOS were proposed by Brinksma [69], Carver and Chen [87], Tretmans [346], and van der Schoot and Ural [323]. Cavalli *et al.* also included a conformance testing methodology for all specifications that can be transformed into a finite state machine [89]. This particular methodology will be discussed further in one of the case studies (Section 8.5).

Although the means and the intent of validation testing are different from those of conformance testing, many general concepts and techniques are common. To some extent, this chapter instantiates the FMCT framework in the context of UCMs and LOTOS, as suggested by Hogrefe *et al.* in their final report [179]. Consequently, most of the terminology used here can be related to that of FMCT.

6.2 UCM-LOTOS Testing Concepts

Several concepts for UCM-LOTOS testing need to be introduced before tackling the problem of test selection, which is addressed in Section 6.3. First, Section 6.2.1 presents how different approaches to testing are combined in SPEC-VALUE. Then, a general structure for UCM-based test suites is given in Section 6.2.2. Section 6.2.3 defines the validation relation that will be used for determining whether the LOTOS prototype verifies the UCMs and hence validates the requirements. This relation is then compared to the conventional LOTOS conformance relation in Section 6.2.4.

6.2.1 Combination of Approaches

There exists an enormous number of testing techniques for formal methods. A classification that has been proposed by Poston identifies the three following categories [287]:

- **Black box:** testing based on externally visible behaviour of a specification or program [44].
- **White box:** testing based on the internal structure of a specification or program.
- **Grey box:** testing based on the design.

The UCM-LOTOS testing framework suggested in SPEC-VALUE uses ideas from all these categories. It focuses mainly on causal sequences at the design level as defined by UCMs (grey-box), on the generation of functional tests in LOTOS (black-box), on structural coverage measurement techniques for LOTOS (white-box), and on the use of relevant data values (boundary analysis and equivalence classes, i.e. black-box testing).

SPEC-VALUE makes use of several guidelines and assumptions related to testing. Often, UCMs are generic enough to provide for implicit equivalence classes of data and behaviour, and we intend to take advantage of this characteristic during the selection of test cases. The focus will also be on deterministic test cases (as sequences of events) whenever possible, i.e. when the specification under test is deterministic. Such tests are usually faster to check and their results simpler to understand. They can also be reused more easily in the stages of the design cycle closer to implementation. Finally, recursion will not be given much attention, except for critical system functionalities.

6.2.2 Structure of UCM-Based Validation Test Suites

Test cases have to reflect the UCMs in order for them to be traceable to the requirements and, obviously, to the UCMs themselves. This is particularly important when faults are detected at testing time; traceability to the UCMs and to the requirements helps assessing where the problems are located and what modifications should be made. The structure of test suites defined in CTMF and presented in Section 3.4.2 can help in the implementation of such a traceability relationship. This structure is flexible and can be tailored to the context of SPEC-VALUE.

A UCM-based *validation test suite* is a collection of test groups, where each group is linked to one UCM (in the case where multiple non-integrated UCMs are available) or to a particular set of functionally related root maps and plug-ins (in the case where only the integrated UCM is available). A test group may contain multiple (LOTOS) test cases, which are composed of the following parts:

- **Test purpose:** a pair $\langle type, goal \rangle$ where the test *type* is either acceptance (must test) or rejection (reject test). As explained at the end of Section 3.4.3, *may tests* are not really helpful here because they require too much effort for the interpretation of verdicts. The test purpose also contains a *goal*, which is the specific UCM route that is covered by the test. This route is usually defined as an abstract sequence of events corresponding to UCM start points, responsibilities, waiting places, timers and end points.
- **Test preamble:** test events needed to bring the *Specification Under Test* (SUT¹) in a state that satisfies the preconditions attached to the UCM route corresponding to the test *goal*.
- **Test body:** test events corresponding to the selected goal, instantiated with appropriate data values. Reusable test steps, representing fragments of routes, may optionally be defined and built upon. This can help to define test suites that are more consistent and easier to extend.
- **Test verification** (optional): test events used to check that the SUT has reached the post-condition attached to the UCM route corresponding to the *goal*. The verification is not based on FSM techniques, such as unique input/output sequences, because FSMs are usually not available at requirement time.

The individual test cases that constitute the test suite are LOTOS processes where test events, which belong to preambles, bodies, and verification steps, are transformed into LOTOS events. Table 15, which supplements the definitions found in Table 12, formalizes the notation for test purposes. Note that this representation is significantly simpler than a full-fledge standard testing lan-

1. In the thesis, SUT refers to a *specification* under test, not a *system* under test (implementation)

guage such as TTCN. TTCN is not used here because it uses a format and a level of detail that would lead to unnecessarily long and verbose descriptions of test purposes.

TABLE 15. Notation for Test Purposes

<i>Notations</i>	<i>Definitions</i>
ABSSEQ	Universe of all possible abstract causal sequences (linear LTSs). $\mathbf{ABSSEQ} \subseteq \mathbf{SPECS}$.
TESTTYPE	Acceptance or rejection test case. $\mathbf{TESTTYPE} = \{Accept, Reject\}$
$TP(T_x)$	Test purpose of test case T_x , composed of a test type and of a goal (abstract causal sequence). $TP(T_x) \subseteq \mathbf{TESTTYPE} \times \mathbf{ABSSEQ}$.
$Type(TP(T_x))$	Projection of $TP(T_x)$: type of test case T_x . $Type(TP(T_x)) \in \mathbf{TESTTYPE}$.
$Goal(TP(T_x))$	Projection of $TP(T_x)$: goal of test case T_x . $Goal(TP(T_x)) \in \mathbf{ABSSEQ}$.

A test suite TS can be partitioned into two mutually exclusive subsets according to the type of test cases, i.e. acceptance and rejection test cases:

$$\begin{aligned} \mathbf{Definition 6.1:} \quad & ACCEPT(TS) = \{ T_x / T_x \in TS \wedge Type(TP(T_x)) = Accept \} \\ & REJECT(TS) = \{ T_x / T_x \in TS \wedge Type(TP(T_x)) = Reject \} \end{aligned}$$

Definition 6.1 respects the two properties described in Section 3.4.3: the two sets do not overlap ($REJECT(TS) \cap ACCEPT(TS) = \emptyset$) and they are complete ($REJECT(TS) \cup ACCEPT(TS) = TS$).

Defining Tests Groups

For each individual UCM or functionally-related set of root maps and plug-ins in an integrated UCM, it is desirable to create at least two test groups: one for acceptance test cases and another one for rejection test cases. Groups can be described as a collection of sequential test processes, one for each test case. This way of representing test groups has the benefit of establishing clear traceability relationships between test cases and requirements UCMs. They also simplify diagnostics and debugging when unexpected verdicts are encountered, and they require a minimum quantity of memory

For complex test cases and complex specifications, the creation of one process for each test case is recommended. The state space resulting from the testing composition will be more manageable by the tools, and the overall test cases traceability and reusability improved.

Towards Implementation Test Suites

For LOTOS prototypes, test postambles are seldom used because the execution of each test case can easily start with the initial state of the LOTOS specification (i.e. resetting a LOTOS SUT is not an issue). Moreover, the selection of data values is eased by the fact that we only have constraints and conditions associated to one path to satisfy, starting from a well-known initial state. However, if the test suite is meant to be refined as an *implementation* test suite, then postambles become most relevant because the cost of resetting a real machine after each test might be too high. In this case, we need to give much attention to four points:

- **Ordering:** the order in which test groups and test cases within test groups are executed becomes relevant. An ordering strategy is needed for reducing the cost of executing the test suite.
- **Postambles:** they become necessary for bringing the SUT back to an acceptable initial state, where the preamble of the next test case can satisfy its precondition. A merging of postambles and preambles can be performed in some cases, but at the cost of an increased coupling between test cases.
- **Data values:** their selection becomes critical as they have more constraints to satisfy across many test cases. They need to be carefully chosen and be consistent within a test group.
- **Target:** tests need to be retargetable and readable by test equipment. Currently, LOTOS processes are not used in a standard way to describe test cases, and hence they might have to be translated into a more suitable representation. TTCN-3, the latest ITU-T standard notation for the specification of abstract test cases, is such a representation [209]. TTCN-3

provides sufficient support for LOTOS constructs used in most test cases (sequence, choice, interleave, process instantiation, value passing, etc.).

These issues will not be investigated further in the thesis as it focuses on specification testing rather than on implementation testing.

6.2.3 Validity Relation

Checking the SUT against the test cases must help to establish the validity of the specification against the UCMs and the requirements. At this point, we assume that the test cases are LOTOS processes derived from UCMs capturing the requirements faithfully. Several equivalence and implementation relations for LOTOS have been presented in Section 2.3.6. Among them, the conf relation is the most common for establishing the conformance of a model with respect to another. One weakness of the conf relation is that it focuses exclusively on acceptance testing, and it is always possible to build a trivial model that conforms to another (e.g. a process that accepts everything, as in Figure 13(h)). To solve this weakness, rejection testing can be used in addition to acceptance testing (Section 3.4.3). This solution is at the basis of a new relation called val, which is used to determine the *validity* of a SUT against a validation test suite.

Using the notations and definitions found in Table 12 and Table 13, suppose that TS is a validation test suite generated from informal requirements and/or a collection of UCMs ($TS \subseteq \mathbf{TESTS}$). TS is composed of finite sets of finite acceptance test cases ($ACCEPT(TS)$, see Definition 6.1) and rejection test cases ($REJECT(TS)$), as shown in Figure 38.

FIGURE 38. Partitioning of Acceptance and Rejection Test Groups and Test Cases

		Validation Test Suite TS					
		$ACCEPT(TS)$			$REJECT(TS)$		
Test Groups		TG_{A1}	TG_{A2}	...	TG_{R1}	TG_{R2}	...
Test Cases		$T_{A1.1},$ $T_{A1.2}, \dots$	$T_{A2.1},$ $T_{A2.2}, \dots$...	$T_{R1.1},$ $T_{R1.2}, \dots$	$T_{R2.1},$ $T_{R2.2}, \dots$...

Intuitively, a *SUT* is valid with respect to a validation test suite *TS* if and only if all acceptance test cases in *TS* pass successfully and all rejection test cases in *TS* are rejected. Formally:

Definition 6.2: $\underline{\text{val}}$: Validity relation. $\underline{\text{val}} \subseteq \mathbf{SPECS} \times \text{PowerSet}(\mathbf{TESTS})$.
 $SUT \underline{\text{val}} TS \Leftrightarrow SUT \underline{\text{passes}} ACCEPT(TS) \wedge SUT \underline{\text{failsall}} REJECT(TS)$

Conformance test suites can be sound, exhaustive or complete (Section 3.4.2), and this concept is also applicable to a validation context. Assuming an idealized variant of $\underline{\text{val}}$ called $\underline{\text{val-id}}$, which could handle very large or even infinite sets of acceptance and rejection test cases, *TS* could be characterized in the following way:

- Necessary condition for validity: *TS* is **sound** \Leftrightarrow
 $(\forall SUT \in \mathbf{SPECS}, SUT \underline{\text{val-id}} TS \Rightarrow SUT \underline{\text{passes}} ACCEPT(TS) \wedge SUT \underline{\text{failsall}} REJECT(TS))$
- Sufficient condition for validity: *TS* is **exhaustive** \Leftrightarrow
 $(\forall SUT \in \mathbf{SPECS}, SUT \underline{\text{passes}} ACCEPT(TS) \wedge SUT \underline{\text{failsall}} REJECT(TS) \Rightarrow SUT \underline{\text{val-id}} TS)$
- *TS* is **complete** $\Leftrightarrow TS$ is **sound** $\wedge TS$ is **exhaustive**.

For most realistic telecommunications systems, validation test suites have to be truncated to finite and manageable sets of acceptance and rejection test cases. Consequently, *TS* can be sound but cannot be exhaustive (nor complete). In SPEC-VALUE, the soundness of *TS* will result from its derivation from UCMs (or from the requirements in some cases) interpreted in LOTOS. In practice, the lack of completeness caused by testing limitations forces us to accept a more pragmatic interpretation of the validity relation: if a SUT is not shown to be valid by a test suite, then it is considered invalid (Proposition 1).

$$\neg(SUT \underline{\text{passes}} ACCEPT(TS) \wedge SUT \underline{\text{failsall}} REJECT(TS)) \Rightarrow \neg(SUT \underline{\text{val}} TS) \quad \textbf{(PROP. 1)}$$

Proposition 1 is a direct result of the logical implication defining the concept of soundness. This pragmatic interpretation can take other equivalent forms, one of which (Proposition 2) makes use of the $\underline{\text{fails}}$ relation defined in Table 13.

$$SUT \underline{\text{fails}} ACCEPT(TS) \vee \neg(SUT \underline{\text{failsall}} REJECT(TS)) \Rightarrow \neg(SUT \underline{\text{val}} TS) \quad \textbf{(PROP. 2)}$$

These two pragmatic propositions are insufficient to prove formally that $SUT \text{ val } TS$, but this was expected: it is not possible to prove formally that a formal model validates an informal one. This issue is also connected to the *limit of testability* illustrated in Figure 15. This is a reality with which testers are already familiar, especially in the telecommunication field where complex systems exhibit infinite behaviours. To cope with this limitation, testers need to increase the quality of the test suite by improving its detectability, which will lower the testability limit (Section 3.4.2). This will help to detect more invalid SUTs and to increase the level of confidence in the system. However, an equilibrium between the quality of the test suite and the costs of its derivation and/or execution is also required. Practical and efficient test selection techniques can help reaching such a balance, and this issue will be addressed in Section 6.3.

6.2.4 Comparing Validity and Conformance in the Two Worlds

In a “perfect world”, requirements would already be formalized from the beginning. Assuming that the requirements can be represented as an LTS enables an interesting comparison between the relations val and conf. This comparison becomes possible by creating a test suite equivalent to the canonical tester of the requirements. In the “real world” however, requirements are usually informal and such canonical tester cannot be defined.

Suppose that Req represents the formalized requirements ($Req \in \mathbf{SPECS}$) and that it is possible to derive its canonical tester ($CT(Req)$) or a test suite with the same discriminatory power (denoted by $TS \cong \{CT(Req)\}$, or $TS \cong CT(Req)$ for short). Definition 6.3 says that two test suites have the same discriminatory power (\cong) if and only if any SUT passes them both or fails them both.

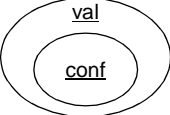
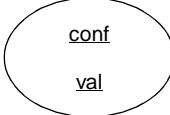
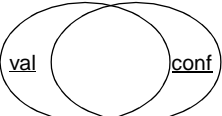
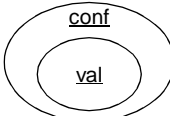
Definition 6.3: $\forall SUT \in \mathbf{SPECS}, TS1 \subseteq \mathbf{TESTS}, TS2 \subseteq \mathbf{TESTS},$
 $TS1 \cong TS2 \Leftrightarrow (SUT \text{ passes } TS1 \Leftrightarrow SUT \text{ passes } TS2)$

Proposition 3 suggests that an SUT passes the canonical tester $CT(Req)$ (or the equivalent test suite TS) if and only if this SUT conforms to the requirements.

$$SUT \text{ passes } CT(Req) \Leftrightarrow SUT \text{ conf } Req \quad \text{(PROP. 3)}$$

Proposition 3, which is proven in [69][242], is at the basis of the classification illustrated in Figure 39. Note that $CT(Req)$ and the tests derived directly from this canonical tester are all acceptance test cases. Intuitively, the expression $SUT \text{ val } ACCEPT(TS)$ means that each sequence or trace refused by SUT is also refused by Req . Hence, similarly to the conformance relation, SUT is allowed to be more deterministic and to have more behaviour than Req . However, the additional behaviour in SUT must not be rejected by any rejection test case. Therefore, the set $REJECT(TS)$ must also be a criterion for comparing val and conf. Figure 39 classifies test suites according to these two criteria. Although the signatures of these relations are not exactly the same, the mapping of conf's signature to val's is made implicit by Proposition 3, which enables a more direct comparison. This comparison is detailed through various propositions and illustrated with an example in Appendix C:

FIGURE 39. Comparison Between val and conf

	Real world $\neg(ACCEPT(TS) \cong CT(Req))$	Perfect world $ACCEPT(TS) \cong CT(Req)$
Without Rejection Tests $REJECT(TS) = \emptyset$		
With Rejection Tests $REJECT(TS) \neq \emptyset$		

The key points resulting from this comparison are:

- In the real world, conf is more powerful than val when there are no rejection test cases involved. However, they are incomparable when there are rejection tests cases.
- In a perfect world, both relations are equivalent in the absence of rejection test cases. However, val becomes more powerful than conf when rejection test cases are considered.

- The use of rejection test cases permits the detection of invalid yet conforming SUTs, even in the presence of incomplete acceptance test suites. Therefore, rejection tests improve detectability over conventional conformance testing for LOTOS, and they lower the limit of testability, as suggested in Section 3.4.2.

This theoretical result does not solve the problem of test selection and generation, which are the topics of the next two sections. However, as suggested by Myers' citation (found below the title of this chapter) and by Harel's attention to rejection scenarios at the requirements level [167], this result emphasizes the need to produce both acceptance and rejection tests when validating a specification against requirements.

6.3 UCM-Oriented Testing Patterns for Test Goal Selection

The relation *val* presupposes the existence of a sound test suite used to establish the validity of the LOTOS prototype, which integrates all the scenarios generated from the requirements, against the intended functional requirements. Such test cases can be generated in numerous ways. This section introduces a novel approach where system UCMs are used in combination with *testing patterns* for the selection of goals for test purposes in a test plan. These goals take the form of abstract causal sequences suitable for the generation of LOTOS test suites composed of acceptance and rejection test cases. The selection and generation of test cases corresponds to step ⑤ in the SPEC-VALUE methodology (Figure 37).

Section 3.4.5 already provided an overview of patterns in general, with an emphasis on design and testing patterns. Section 6.3.1 discusses how the pattern concept can be applied to UCM-based testing, and a template is tailored accordingly in Section 6.3.2. A testing pattern language is defined in Section 6.3.3 in order to explain how to apply individual UCM-oriented testing patterns on a complex UCM. Sections 6.3.4 to 6.3.9 present the individual testing patterns with strategies for the selection of test goals based on alternatives, concurrent paths, loops, multiple start points, single stubs, and causally linked stubs. These testing patterns, which capture the author's experience in UCM-based test selection, represent an important constituent of the UCM-LOTOS testing framework. Section 6.3.10

attempts to relate UCM-oriented testing patterns to the conventional LOTOS testing theory. A summary of the main points and a brief discussion follow in Section 6.3.11.

6.3.1 Introduction to UCM-Oriented Testing Patterns

UCM scenarios are constructed from requirements. Not only can they be used to construct the first design, but they also can be reused to guide the generation of functional test cases. To exploit this idea, testing patterns will be defined based on the nature and target coverage of UCM paths. These paths capture causal flows and they essentially become visual templates for the selection of test goals at a development stage close to requirements definition.

This novel approach to test selection shares many concepts with white-box testing [267][290]. However, the selection is done at a much higher level abstraction, and the focus is on the structure of a UCM (paths and constructs) rather than on the structure of a program. Each UCM route, where data parameters are instantiated, is a candidate for becoming a functional test case applicable to the LOTOS prototype for validation purpose. Tretmans suggested the use of goals as a means to select tests for complex systems [346] (see Section 3.4.2), and UCM routes can indeed be interpreted as test goals to be fulfilled.

Testing Patterns as a Semi-Formal Approach to Test Selection

Testing patterns represent a trade-off between intuitive test case generation, still heavily used nowadays, and formal test case generation, more rarely used, even in the telecommunications industry. For instance, in many LOTOS-based techniques, test selection is done either informally, or formally through a model like finite state machines (e.g. transition tours [143]) or LTSs (e.g. using a canonical tester [69]). SPEC-VALUE targets the generation of the first system formal model from requirements and scenarios that are not necessarily formalized, therefore a formal model-based strategy cannot be used here. Testing patterns happen to be rather useful in this particular context. Their users can potentially benefit from the existence of visual requirements and design information (the UCMs) together with suggestions of mappings onto test cases. Testing patterns can be seen as a *semi-formal approach*

to test selection that fits nicely with the level of abstraction targeted by a semi-formal notation like Use Case Maps.

UCM paths and constructs represent *forces* (introduced in Section 3.4.5) related to the functional coverage of the requirements, the quality of the tests, the number of tests to generate, the complexity of each test, and the overall cost of testing. The higher the number of routes or end-to-end paths covered in a UCM, the higher the coverage and the confidence in the validity of the system, but the longer the test suite and the higher the costs of its derivation and execution. Testing patterns can help achieving good compromise solutions between cost and effectiveness by balancing these forces.

A UCM may include many possible routes, some of which might not be necessary for testing purposes. In this context, the traditional question “*how much testing is enough testing?*” becomes “*what are the routes to be tested?*”. There is no unique answer to this question. Achieving the coverage of all UCM path segments and constructs is certainly a sensible testing objective. However, depending on how critical, important, or relevant are the routes, a UCM may be tested more or less thoroughly. The assumption here is that UCM constructs are where such decision can be taken. Testing patterns targeting UCM constructs hence helps concretizing test plans.

Naturally, the use of testing patterns is not the only way to derive test cases from requirements and UCMs. For instance, UCMs could be transformed into an intermediate model (e.g. FSM or spanning tree) from which test cases could be generated using conventional techniques. However, such test cases would be rather synthetic, whereas the application of testing patterns to the UCM scenarios will lead to test cases that are closer to the requirements and the initial intent of the scenarios. Moreover, we believe such patterns to provide more flexibility to the tester in the selection of appropriate test cases; not all the information necessary to the generation of effective test suites is found in the UCMs.

The eight UCM-oriented testing patterns defined in this chapter record the experience gained during the validation of various specifications by the author (Chapter 8). They focus on how abstract causal sequences, used as goals in test purposes, can be generated from UCMs containing different constructs. Because of the abstract nature of UCMs and because these testing patterns are indepen-

dent of underlying structures of components, the patterns can be applied to a multitude of contexts, and can be combined together (e.g. in a pattern language) for dealing with complex UCMs.

Targeted Coverage

The testing patterns we develop here target the coverage of scenarios described in UCM terms. These patterns aim to cover functional scenarios at various levels of completeness: all results, all causes and all results, all path segments, all end-to-end paths, all plug-ins, and so on. The rationale is that covering UCM paths leads to the coverage of the associated events and responsibilities (and of their relative ordering) forming the requirements scenarios. The patterns are inspired partly from various existing test selection strategies for implementation languages constructs such as branching conditions and loops [44][267][290], or for cause-effect graphs [267][270]. The contribution of these patterns is in their application to UCM scenarios at a level close to requirements.

6.3.2 Template for UCM-Oriented Testing Patterns

The UCM-oriented testing patterns are formatted according to a template inspired from Binder's [51] and from generic templates used in the design pattern community (Section 3.4.5). The former is specifically tailored for test patterns in general whereas the latter suggest useful fields such as forces and examples. In his book, Binder regrouped most test selection and generation techniques available nowadays for object-oriented and procedural models and programs, and he describes them using his pattern template. Table 16 presents the template used in the thesis and how it relates to the templates mentioned above.

TABLE 16. Correspondence Between Templates for Design Patterns and Test Patterns

<i>Design Patterns (in general)</i>	<i>Binder's Test Design Patterns</i>	<i>UCM-Based Testing Patterns</i>	<i>Definitions for UCM-Based Testing Patterns</i>
Name	Name	Name	Descriptive name that identifies the pattern
Problem	Intent	Intent	Type of tests produced by the pattern (test goals)
	Fault Model	Fault Model	Type of faults targeted by the pattern
Context	Context	Context	Situations under which the pattern applies
Forces		Forces	Relevant factors contributing to the problem and the solution/strategy, and their interactions
Solutions	Strategy	Strategy	How the test goals are constructed
	Entry/Exit Criteria		
Example	—	Example	Illustration of UCM with selected test goals
Consequences	Consequences	Consequences	Benefits and drawbacks of using this strategy
Known Uses	Known Uses	Known Uses	Related uses in terms of other UCM constructs
Related Patterns	Related Patterns	Related Patterns	Related testing patterns (see also Section 6.3.3)

Problems are split into two fields: *intent* and *fault model*. Binder's *context* field already contains *forces*, but our template makes them more explicit, as suggested by the design pattern community [256]. The entry and exit criteria used by Binder are useful for classifying very different approaches to test selection and generation, but they are rather useless in our context where UCMs are always involved. A solution, called *strategy* in our template, takes certain forces into account and resolves some of them at the expense of others. Related solutions will be described under the same pattern for the sake of conciseness. Consequently, a testing pattern can include multiple triples $\langle \textit{Strategy}, \textit{Example}, \textit{Consequences} \rangle$, accompanied by a unique strategy name for future reference and for traceability. As a visual convention, horizontal dashed lines will separate these triples.

In the following testing patterns, a UCM route (delimited by angle brackets) is an abstract causal sequence that represents the *goal* part of a test purpose (Section 6.2.2). A test goal establishes the traceability between the UCM and a test case, and it is independent of the *type* of test (accept or reject). The type will be taken into consideration only when transforming a test goal into an acceptance or rejection LOTOS test process.

6.3.3 UCM-Oriented Testing Pattern Language

A pattern language is a set of cooperating patterns that combine to provide solution guidelines for a problem in a particular context. Each of the testing patterns developed in this chapter focuses on one particular type of UCM construct (alternative, loop, stub, and so on.). These patterns can cooperate to help solution more complex UCMs where multiple constructs of various type are used. However, The connections between patterns (how they relate with one another) are not always obvious.

The following *UCM-oriented testing pattern language* explains how the individual testing patterns can be connected together in order to derive test goals from UCMs with multiple constructs. This language itself is expressed as a UCM that shows the various steps (static stubs) and patterns (dynamic stubs) involved (Figure 40). A similar way of expressing pattern languages has already been used successfully by Andrade and Logrippo [33][35], and it seems appropriate in our context. The proposed combination of testing patterns should be seen as a general recommendation rather than as a strict procedure to be followed blindly. The testing pattern language is a guideline and is no substitution for the test engineer's judgement and experience.

FIGURE 40. UCM-Oriented Testing Pattern Language

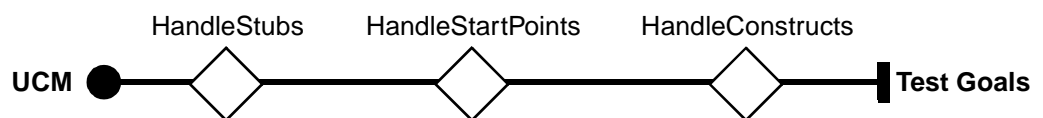
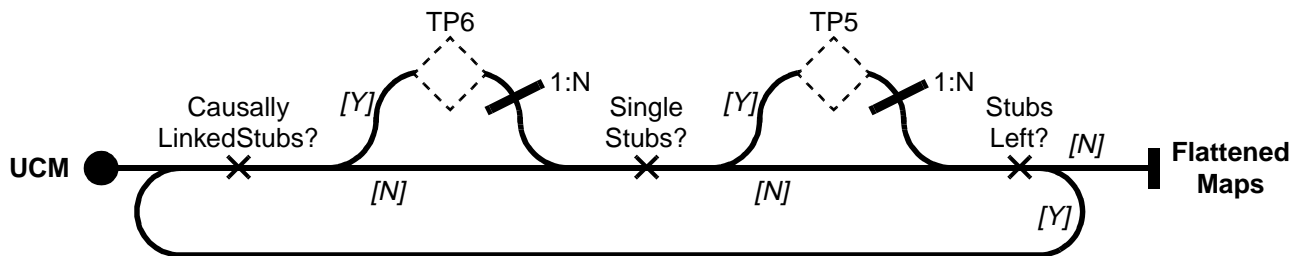


Figure 40 shows the progression from UCMs to test goals through three steps. Stubs can first be substituted by their plug-ins in order to produce a collection of flattened maps (HandleStubs). Each flattened map may contain multiple start points, and various subsets of these start points may be enabled in order to generate test goals. HandleStartPoint shows how to select such subsets, which lead to further flattened maps where disabled start points are essentially considered absent. The final step (HandleConstructs) consists in generating test goals from flattened maps that contain alternatives, concurrent segments, and loops. These three static stubs are refined by three plug-ins with the same names.

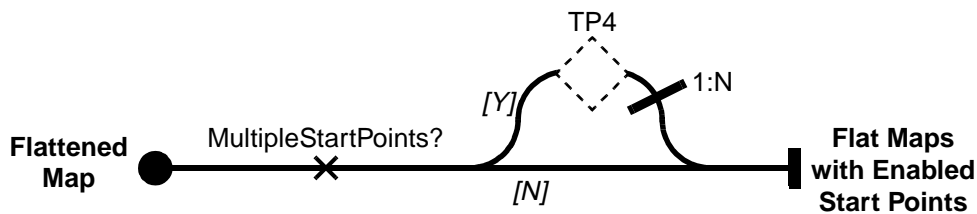
The first step, captured by the HandleStubs plug-in in Figure 41, flattens the stubs using two testing patterns. First, if there are stubs that are causally linked (e.g. in sequence), then Testing Pattern 6—CAUSALLY LINKED STUBS is used (TP6 stub), for which three strategies are defined. Many flattened maps could result from these strategies (shown with the 1:N AND-fork). They may however still contain single stubs (TP5), which are flattened using one of the three strategies found in Testing Pattern 5—SINGLE STUB. This pattern will generate multiple flattened maps for dynamic stubs (the second 1:N AND-fork). In turn, if there are nested stubs in the plug-ins, they would appear in the flattened map and both testing patterns could be applied iteratively.

FIGURE 41. Plug-in HandleStubs (Step 1)



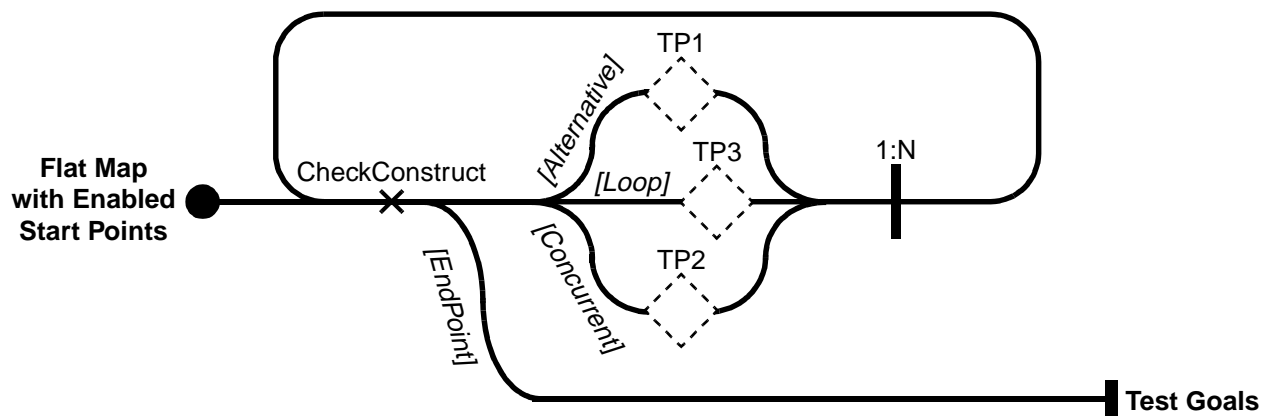
Flattened maps may contain multiple start points. If so, then they should be handled by the Testing Pattern 4—MULTIPLE START POINTS as shown in Figure 42. This pattern leads to flat maps where different subsets of the start points involved are enabled and the other removed according to one of the eight defined strategies. Again, some of them may lead to a multiplication of such flat maps (1:N). Maps with a single start points are assumed to have this start point enabled by default.

FIGURE 42. Plug-in HandleStartPoints (Step 2)



Finally, the last step iteratively handles remaining UCM constructs for alternatives, concurrent segments, and loops until an end point is reached (Figure 43). These constructs can be nested or in sequence. Testing Pattern 1—ALTERNATIVE has 4 strategies, whereas Testing Pattern 2—CONCURRENT has 3 strategies and Testing Pattern 3—LOOP has 4 strategies. Multiple partial test goals are constructed along the way (1:N), until they are finalized when end points are reached.

FIGURE 43. Plug-in HandleConstructs (Step 3)



This UCM-oriented testing pattern language helps to identify the paths to test from the original UCM by using combinations of patterns. The strategies described in these patterns are coverage-driven and aim to balance the various forces involved in order to come up with cost-effective test goals. The next five sub-sections define and illustrate the various testing patterns and strategies used in our testing pattern language.

6.3.4 Testing Pattern and Strategies for Alternatives

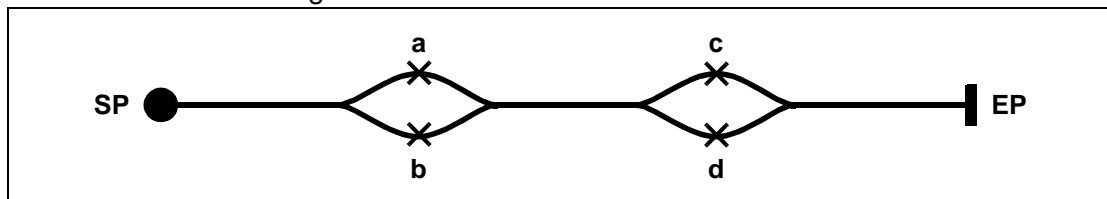
NAME: TESTING PATTERN 1—ALTERNATIVE

INTENT

To generate, for alternative UCM routes, test goals expressed in terms of sequentially linked start points, responsibilities, waiting places, end points, and other such events.

CONTEXT

The functionality under test is captured as a UCM path that contains multiple routes connected through OR-forks. Some of these routes may also merge at a later point through OR-joins. Figure 44 shows a UCM used to illustrate strategies for alternatives. Four responsibilities (**a** to **d**) identify different path segments found between the UCM start point (**SP**) and the end point (**EP**). Branches in OR-forks may also be guarded by boolean conditions.

FIGURE 44. Reference UCM: Testing Pattern for Alternatives**FAULT MODEL**

A test goal is a route from a start point to an end point. The fault model assumes that faults can hide in UCM path segments not traversed by any test goal. These faults are independent of their allocation to components, if any. The coverage of all path segments would ensure that these faults are detected.

FORCES

Alternatives that join and then fork again lead to multiple end-to-end combinations (routes) for covering the path segments. Generating test goals for all combinations leads to more thorough test suites, but at a higher cost.

The following four strategies are inspired from control flow testing. They are ordered according to the completeness of their route coverage, from the least complete to the most complete.

STRATEGY 1.A: ALTERNATIVE — ALL RESULTS

Each end point (result) is covered. There could be many end points connected by an OR-fork, leading to the same number of goals.

EXAMPLE: {<SP, a, c, EP>}

CONSEQUENCE

Minimal coverage for all results. Very low cost, but some path segments might not be covered.

STRATEGY 1.B: ALTERNATIVE — ALL PATHS

All decisions (e.g. true or false) of conditions are exercised. Also referred to as “All branches”.

EXAMPLE: {<SP, a, c, EP>, <SP, b, d, EP>}

CONSEQUENCE

Minimal coverage for all segments. Some end-to-end paths might not be covered.

STRATEGY 1.C: ALTERNATIVE — ALL PATH COMBINATIONS

All combinations of conditions (e.g. True-True, True-False, False-True, False-False) are explored. Also referred to as “All branch combinations” or “All decision combinations”.

EXAMPLE: {<SP, a, c, EP>, <SP, a, d, EP>, <SP, b, c, EP>, <SP, b, d, EP>}

CONSEQUENCE

Minimal coverage for all end-to-end paths. There might be various ways of satisfying (or not) one condition, and only one possibility is explored. This strategy may result in abstract causal sequences that are not feasible according to the conditions attached to the branches. For instance, if two branches from two consecutive OR-forks respectively have the conditions $[c1]$ and $[\text{not}(c1)]$, then an abstract sequence covering these two branches cannot be used to create an acceptance test case. Such sequences however are good candidates for rejection test cases.

STRATEGY 1.D: ALTERNATIVE — ALL COMBINATIONS OF SUB-CONDITIONS

Complex conditions include more than one operator, and all combinations of basic boolean expressions can be explored. This strategy can further be applied to multiple conditions when necessary.

EXAMPLE

Observe the following LOTOS guard: $[(c1 \text{ AND } c2) \text{ OR } (c3 \text{ AND } c4)]$. Since $c1$, $c2$, $c3$, and $c4$ can all evaluate to True or False, there are up to 16 possible combinations (2^4) for this condition only¹.

CONSEQUENCE

Results in thorough test suites when combined to Strategy 1.C, but at a very high price.

KNOWN USES

This testing pattern can also be applied to timers with time-out paths, which represent a point where two alternatives are possible (time-out and no time-out). Also applicable to failure points, which represent yet another type of alternative.

RELATED PATTERNS

Alternative paths may further contain concurrent segments and loops, to be handled respectively with Testing Pattern 2—CONCURRENT and Testing Pattern 3—LOOP. UCMs with stubs and multiple start points are assumed to have been flattened during a previous step.

6.3.5 Testing Pattern and Strategies for Concurrent Paths

NAME: TESTING PATTERN 2—CONCURRENT

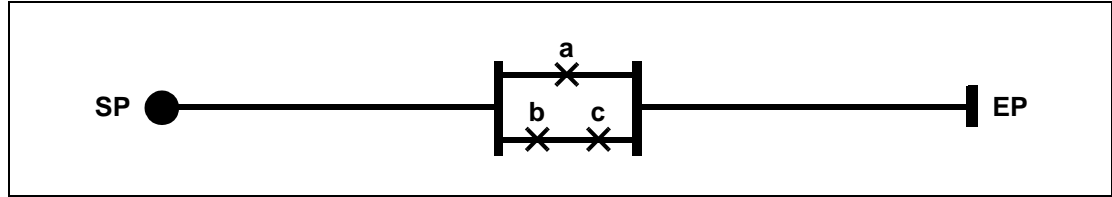
INTENT

To generate, for concurrent UCM path segments, test goals expressed in terms of sequentially linked start points, responsibilities, waiting places, end points, and other such events.

CONTEXT

The functionality under test is captured as a UCM path that contains multiple segments connected through AND-forks. Some of these segments may also merge at a later point through AND-joins. Figure 45 shows a UCM used to illustrate strategies for path segments that run concurrently due the presence of AND-forks.

1. If sub-conditions are not independent, some combinations might be impossible to satisfy. For instance, in the predicate $[x < 3 \text{ OR } x > 5]$, there is no solution such that $x < 3$ is true and $x > 5$ is true. See Section 6.4.3.

FIGURE 45. Reference UCM: Testing Pattern for Concurrent Paths**FAULT MODEL**

A test goal is a route from a start point to one or multiple end points. The fault model assumes an interleaving semantics à la LOTOS for the concurrent path segments. Faults can result from some interleaved combinations that are prevented undesirably by the prototype specification. The coverage of all interleaved combinations would ensure that these faults are detected.

FORCES

The number of possible combinations in an interleaving semantics depends on the number of concurrent path segments (k) and the number of responsibilities and events along these segments (n_k). This number can be computed by the function *InterComb* (interleaved combination — Definition 6.4). For example, the concurrent segments in Figure 45 lead to $InterComb(1, 2) = (1+2)!/(1!*2!) = 3$ possible combinations ($\langle \mathbf{a}, \mathbf{b}, \mathbf{c} \rangle$, $\langle \mathbf{b}, \mathbf{a}, \mathbf{c} \rangle$, $\langle \mathbf{b}, \mathbf{c}, \mathbf{a} \rangle$).

Definition 6.4: $InterComb(n_1, n_2, \dots, n_k) = \frac{(n_1 + n_2 + \dots + n_k)!}{n_1! \times n_2! \times \dots \times n_k!} = \frac{\left(\sum_{i=1}^k n_i\right)!}{\prod_{i=1}^k n_i!}$

The interleaving semantics quickly produces a high number of possible combinations, and generating test goals for all combinations leads to more thorough test suites, but at a higher cost. However, from a functional testing viewpoint, this number can be reduced when concurrent responsibilities and events are hidden. For instance, if responsibilities **a**, **b**, and **c** in Figure 45 are bound to a component, then the only visible events in the corresponding LOTOS prototype would be **SP** and **EP**. With a tool such as LOLA, only one functional test case is sufficient to cover all combinations. Hence, we can abstract from hidden concurrent responsibilities and events when selecting test goals that are intended to be used for the generation of LOTOS test cases.

Nested AND-forks and sequences of AND-forks separated by AND-joins can be handled independently.

The following three strategies are ordered according to the completeness of their combination coverage, from the least complete to the most complete.

STRATEGY 2.A: CONCURRENT — ONE COMBINATION

One combination is chosen. This simple strategy is to be used when the verification of concurrency is not critical, i.e. any ordering of the responsibilities or events along the concurrent paths will suffice.

EXAMPLE: { <SP, a, b, c, EP> }

CONSEQUENCE

Minimal coverage. Very low cost, but many combinations (and potential faults) might not be covered.

STRATEGY 2.B: CONCURRENT — SOME COMBINATIONS

Several combinations are chosen. This strategy is to be used when concurrency is important, but when the total number of possible combinations is too high. The higher the number of goals generated, the higher becomes the level of confidence.

EXAMPLE: { <SP, a, b, c, EP>, <SP, b, a, c, EP> }

CONSEQUENCE

Affordable cost, but some combinations (and potential faults) might not be covered.

STRATEGY 2.C: CONCURRENT — ALL COMBINATIONS

All combinations are generated. This simple strategy is to be used only when concurrency is critical and when the number of combinations is practical.

EXAMPLE: { <SP, a, b, c, EP>, <SP, b, a, c, EP>, <SP, b, c, a, EP> }

CONSEQUENCE

Total coverage, but potentially very high cost.

KNOWN USES

This testing pattern can also be applied to asynchronous interactions between two UCM paths. The UCM path triggered in passing evolves concurrently with the rest of the other UCM path, with a causal behaviour similar to an AND-fork.

RELATED PATTERNS

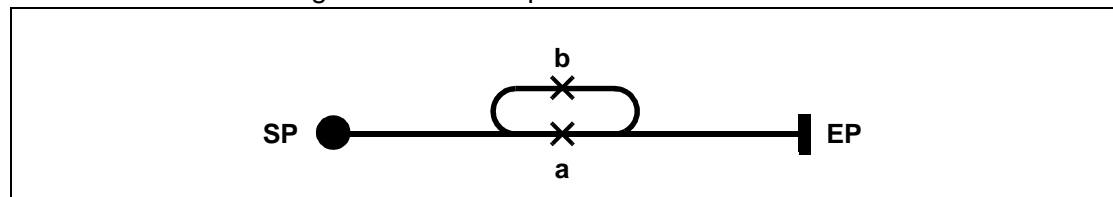
Concurrent paths may further contain alternatives and loops, to be handled respectively with Testing Pattern 1—ALTERNATIVE and Testing Pattern 3—LOOP. UCMs with stubs and multiple start points are assumed to have been flattened during a previous step.

6.3.6 Testing Pattern and Strategies for Loops**NAME: TESTING PATTERN 3—LOOP****INTENT**

To generate, for looping UCM path segments, test goals expressed in terms of sequentially linked start points, responsibilities, waiting places, end points, and other such events.

CONTEXT

The functionality under test is captured as a UCM path that contains loops. Figure 46 shows a UCM used to illustrate relevant strategies.

FIGURE 46. Reference UCM: Testing Pattern for Loops**FAULT MODEL**

A test goal is a route from a start point to an end point. The fault model assumes that faults can result from the prototype specification allowing an incorrect number of iterations in a loop. The coverage of all numbers of iterations would ensure that these faults are detected.

FORCES

Loops (specified using recursion in LOTOS) may have a minimal number of iterations (m , which could be 0) and a maximal number of iterations (n). In this case, the overall number of test goals is bounded by $n-m$. However, this number may be too large for practical testing. In the absence of an upper bound, this number may even be infinite. Compromise solutions are needed for such cases.

In some cases, it may not be sufficient to check that the prototype specification accepts valid numbers of iteration. Some invalid number of iterations (e.g. $<m$ or $>n$) should also be rejected. Also, different strategies can be used on nested loops and series of loops independently.

The following four strategies, inspired from [44][267], are ordered according to the completeness of their iteration coverage, from the least complete to the most complete.

STRATEGY 3.A: LOOP — ALL SEGMENTS

A minimal number of iterations that covers all path segments ($\text{MAX}(1, m)$ iterations) is selected. This strategy generates one test goal for the loop and should be used when testing the loop is not utterly important, for instance when focusing on some other aspect or construct of the UCM path under test.

EXAMPLE: {<SP, a, b, a, EP>} (if $m = 0$)

CONSEQUENCE

The looping path is tested at a minimal cost, but some numbers of iterations (and potential faults) might not be covered.

STRATEGY 3.B: LOOP — AT MOST k ITERATIONS

A maximal number of iterations k is selected ($m \leq k \leq n$). This strategy generates $k-m+1$ test goals for the loop and should be used when $n-m$ is very large or when n is undetermined. The closer k gets to n , the better the coverage but the higher the cost.

EXAMPLE: {<SP, a, EP>, <SP, a, b, a, EP>, <SP, a, b, a, b, a, EP>} (if $m = 0$ and $k = 2$)

CONSEQUENCE

The looping path is tested at a pragmatic cost, but some numbers of iterations (and potential faults) might not be covered.

STRATEGY 3.C: LOOP — VALID BOUNDARIES

The valid boundaries in terms of iterations are selected: m , $m+1$, $n-1$, and n . This strategy generates at most 4 test goals for the loop and should be used when $n-m$ is practical.

EXAMPLE: (if $m = 0$ and $n = 4$)

{<SP, a, EP>, <SP, a, b, a, EP>, <SP, a, b, a, b, a, b, a, EP>, <SP, a, b, a, b, a, b, a, b, a, EP>}

CONSEQUENCE

The looping path is tested at a pragmatic cost, with an emphasis on the lower and upper bounds (where many faults usually occur). Some numbers of iterations (and potential faults), however unlikely, may still not be covered. Also, invalid boundaries are not checked.

STRATEGY 3.D: LOOP — ALL BOUNDARIES

The valid boundaries in terms of iterations (m , $m+1$, $n-1$, and n) together with the invalid boundaries ($m-1$ and $n+1$) are selected. This strategy generates at most 6 test goals for the loop and should be used when $n-m$ is practical. Note that the invalid boundaries target the generation of rejection test cases.

EXAMPLE: (if $m = 1$ and $n = 5$)

{<SP, a, EP>, <SP, a, b, a, EP>, <SP, a, b, a, b, a, EP>, <SP, a, b, a, b, a, b, a, EP>, <SP, a, b, a, b, a, b, a, b, a, EP>, <SP, a, b, a, b, a, b, a, b, a, b, a, EP>}

CONSEQUENCE

The looping path is tested at a pragmatic cost, with an emphasis on the lower and upper bounds (where many faults usually occur). Invalid abstract sequences are also checked for both boundaries, i.e. if m or n is incorrectly specified in the loop conditions, then the problem can be detected. How-

ever, some numbers of iterations (and potential faults) might not be covered, although this is often unlikely.

KNOWN USES

This testing pattern can also be applied to loops attached to stubs, e.g. when a path segment exits a stub and enters it again.

RELATED PATTERNS

Looping paths may further contain alternatives and concurrent segments, to be handled respectively with Testing Pattern 1—ALTERNATIVE and Testing Pattern 2—CONCURRENT. UCMs with stubs and multiple start points are assumed to have been flattened during a previous step.

6.3.7 Testing Pattern and Strategies for Multiple Start Points

NAME: TESTING PATTERN 4—MULTIPLE START POINTS

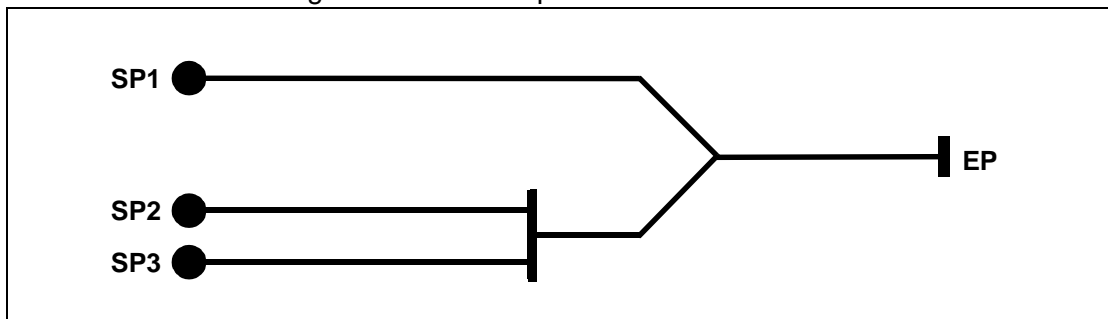
INTENT

To generate, for UCM with multiple start points, test goals expressed in terms of sequentially linked start points, responsibilities, waiting places, end points, and other such events.

CONTEXT

The functionality under test is captured as a UCM that contains multiple start points. These start points can be triggered concurrently, representing multiple potential causes for a resulting event. Figure 47 shows a UCM used to illustrate relevant strategies.

FIGURE 47. Reference UCM: Testing Pattern for Multiple Start Points



In Figure 47, **EP** can be caused either by **SP1** alone, or by **SP2** in conjunction with **SP3** (responsibilities could be added, but they have no impact on the rationale behind the strategies developed here). This example is generic enough to capture all cases and derive general strategies while avoiding superfluous constructs.

FAULT MODEL

A test goal is a route (or many route segments that join) from one or many start points to an end point. The fault model assumes that faults can result from a subset of the start points triggered simultaneously. The coverage of all such subsets would ensure that these faults are detected.

FORCES

There could be numerous subsets to check (2^n , where n is the number of start points). Generating test goals for all subsets leads to more thorough test suites, but at a higher cost. Some of these subsets could provide insufficient stimuli for the observation of the expected result. Some subsets could provide redundant stimuli, whereas others could provide just the right set of necessary stimuli. Additionally, stimuli could be given in many sequential orders. These combinations would again multiply the number of test goals.

STRATEGY OVERVIEW

In order to determine which start points need to be triggered for reaching a specific end point, we can use *path sensitization* algorithms such as the ones suggested for cause-effect graphing by Myers [267] and by Nursimulu and Probert [270]. Starting from an end point (EP), we follow the path backward to find the causes (SP_n) that need to be triggered. Weyuker *et al.* [373] also suggested test selection strategies for boolean specifications (without behaviour however) from which the representation of several ideas discussed in this section were inspired. Different strategies can be defined based on the coverage of the start points and on how easy diagnostics can be established in case of an unexpected verdict.

In logical terms, considering start points and end points only, Figure 47 translates to the following boolean expression: $EP \Leftrightarrow SP1 \vee (SP2 \wedge SP3)$. The corresponding truth table is shown in Table 17, where T and F denote the presence and absence of a triggering event. Cases 1 to 7 are to be addressed by the following eight strategies. Case 0 is not really interesting because no behaviour is

initiated. Although in this case the goal $\langle \mathbf{EP} \rangle$ could be used for the generation of a rejection test case, the latter is unlikely to be effective (very low yield).

TABLE 17. Truth Table for Multiple Start Points Example (from Figure 47)

Case #	SP1	SP2	SP3	$\text{SP1} \vee (\text{SP2} \wedge \text{SP3})$	Subset
0	F	F	F	F	Insufficient stimuli. Not interesting.
1	F	F	T	F	Insufficient stimuli
2	F	T	F	F	Insufficient stimuli
3	F	T	T	T	Necessary stimuli
4	T	F	F	T	Necessary stimuli
5	T	F	T	T	Redundant stimuli
6	T	T	F	T	Redundant stimuli
7	T	T	T	T	Racing stimuli

When \mathbf{EP} is expressed as a minimal sum of product terms ($\text{term}_1 \vee \text{term}_2 \vee \dots \vee \text{term}_n$), then each term_i taken individually characterizes a subset of *necessary* stimuli (start points). Moreover, each subset of stimuli that falsifies \mathbf{EP} is qualified as *insufficient*. *Racing* stimuli are subsets that can cause two or more resulting events \mathbf{EP} (i.e. two or more product terms evaluate to true), whereas *redundant* stimuli are the subsets that do not belong to any of the other categories. These four categories are at the basis of eight strategies for multiple start points:

- **Necessary subsets:** Strategies 4.A, 4.B, and 4.C
- **Redundant subsets:** Strategies 4.D and 4.E
- **Insufficient subsets:** Strategies 4.F and 4.G
- **Racing subsets:** Strategy 4.H

Note that some of these strategies may collapse (i.e. become indistinguishable) for simple maps. Note also that some test goals may not be feasible due to contradicting preconditions attached to different start points. These sequences need to be filtered out or be used for the generation of rejection test cases. The following strategies also consider whether one, some, or all possible stimuli orderings in a subset should be selected for the generation of test goals. The combinations that appeared to

be most pragmatic are covered here. Finally, strategies from different subsets are *not* mutually exclusive and can be used in combination.

STRATEGY 4.A: MULTIPLE START POINTS — ONE NECESSARY SUBSET, ONE GOAL

When multiple necessary subsets can cause the end point (case 3 or case 4), then select one of these subsets and generate one goal accordingly. This strategy targets the minimal set of causes that can lead to the end result. It produces test goals useful for the generation of acceptance test cases.

EXAMPLE: {<SP2, SP3, EP>} (if case 3 is selected)

CONSEQUENCE

The coverage of end points is insured, but not that of start points (for instance, **SP1** is not covered). In case of an unexpected verdict, the diagnostic is simple because the end-to-end path to be followed is known.

STRATEGY 4.B: MULTIPLE START POINTS — ALL NECESSARY SUBSETS, ONE GOAL

Select one test goal for each necessary subset (cases 3 and 4). This strategy targets the minimal set of causes that can lead to the end result, and it targets the generation of acceptance test cases.

EXAMPLE: {<SP2, SP3, EP>, <SP1, EP>}

CONSEQUENCE

The coverage of start points linked to the target end point is complete, but not that of the interleaving start points. In case of an unexpected verdict, the diagnostic is simple because the end-to-end path to be followed is known.

STRATEGY 4.C: MULTIPLE START POINTS — ALL NECESSARY SUBSETS, ALL GOALS

Select all possible test goals for each necessary subset (cases 3 and 4). This strategy targets the minimal set of causes that can lead to the end result, and it targets the generation of acceptance test cases.

EXAMPLE: {<SP2, SP3, EP>, <SP3, SP2, EP>, <SP1, EP>}

CONSEQUENCE

The coverage of start points linked to the target end point is complete, as well as the interleaving start points (two such situations for case 3). Diagnostics remain simple.

STRATEGY 4.D: MULTIPLE START POINTS — ONE REDUNDANT SUBSET, ONE GOAL

The necessary causes are present, plus some redundant (but insufficient) causes. One test goal is selected for one redundant subset. This strategy is non-minimal with respect to the causal relationship. It is useful for the generation of acceptance test cases for testing robustness and partial race conditions on top of expected functionalities. It can be used in a context where the start points are connected to OR-joins and AND-joins.

EXAMPLE: For case 6, **SP1** can cause **EP** by itself, and this should remain the case in the presence of **SP2**: {<SP1, SP2, EP>}

CONSEQUENCE

Start point coverage is partial, and diagnostics are difficult due to the presence of irrelevant and possibly interfering events and responsibilities.

STRATEGY 4.E: MULTIPLE START POINTS — ALL REDUNDANT SUBSETS, ONE GOAL

The necessary causes are present, plus some redundant (but insufficient) causes. One test goal is selected for *each* redundant subset. This strategy is non-minimal with respect to the causal relationship. It is useful for the generation of acceptance test cases for testing robustness and partial race conditions on top of expected functionalities. It can be used in a context where the start points are connected to OR-joins and AND-joins.

EXAMPLE: **SP1** can cause **EP** by itself, and this should remain the case in the presence of either **SP2** (case 6) or **SP3** (case 5): {<**SP1, SP2, EP**>, <**SP3, SP1, EP**>}

CONSEQUENCE

Start point coverage is total, but diagnostics are difficult due to the presence of irrelevant and possibly interfering events and responsibilities. Rejection test cases could also be derived from abstract sequences to which a second **EP** (which should not be observed) is added at the end.

STRATEGY 4.F: MULTIPLE START POINTS — ONE INSUFFICIENT SUBSET, ONE GOAL

Select a test goal for one subset with insufficient stimuli (cases 1 or 2). This strategy checks that the end result cannot be reached, and it targets the generation of rejection test cases.

EXAMPLE: {<**SP2, EP**>} (if case 2 is selected)

CONSEQUENCE

Incomplete coverage of the start points, but simple diagnostics.

STRATEGY 4.G: MULTIPLE START POINTS — ALL INSUFFICIENT SUBSETS, ONE GOAL

Select a test goal for each subset with insufficient stimuli (cases 1 and 2). This strategy checks that the end result cannot be reached, and it targets the generation of rejection test cases.

EXAMPLE: {<**SP3, EP**>, <**SP2, EP**>}

CONSEQUENCE

Still an incomplete coverage of the start points (if some start points are necessary stimuli all by themselves, like **SP1** for instance), but simple diagnostics.

Strategies where all test goals for all insufficient subsets are considered may lead to a higher cost with limited gain in effectiveness (if any). Unless the scenario or application is highly critical, such strategies will produce numerous rejection test cases that are unlikely to be useful. Hence, such strategies are not discussed here.

STRATEGY 4.H: MULTIPLE START POINTS — SOME RACING SUBSETS, SOME GOALS

This strategy targets the testing of race conditions with acceptance test cases (case 7). It is left very loose in terms of path coverage and interleaving of stimuli because many test goals could exist, and selecting the most appropriate ones should be based on the functionality under test.

EXAMPLE: { <SP1, SP3, SP2, EP, EP>, <SP2, SP3, SP1, EP, EP> }

The example includes sequences where the stimuli associated to each product term (**SP1** and (**SP2** \wedge **SP3**)) are used sequentially. Note that two resulting events are generated, and they could be distinguished if necessary (e.g. **EP**_{SP1} and **EP**_{SP2 \wedge SP3}).

CONSEQUENCE

Start point coverage is partial (in general), and diagnostics are difficult due to the presence of possibly interfering events and responsibilities. However, the resulting goals are of interest because they can detect race conditions that would not be detectable otherwise (high-yield test goals).

KNOWN USES

This testing pattern can also be applied to (a)synchronous interactions involving multiple start points. The waiting places or timers involved then have a behaviour similar to an AND-join. When a waiting place or timer is triggered by many alternative end points, then this testing pattern needs to be applied to each alternative end point separately.

RELATED PATTERNS

Sensitized paths may further contain alternatives (OR-forks), concurrent segments (AND-forks) and loops, to be handled respectively with Testing Pattern 1—ALTERNATIVE, Testing Pattern 2—CONCURRENT, and Testing Pattern 3—LOOP. UCMs with stubs are assumed to have been flattened during a previous step.

6.3.8 Testing Pattern and Strategies for a Single Stub and its Plug-ins

NAME: TESTING PATTERN 5—SINGLE STUB

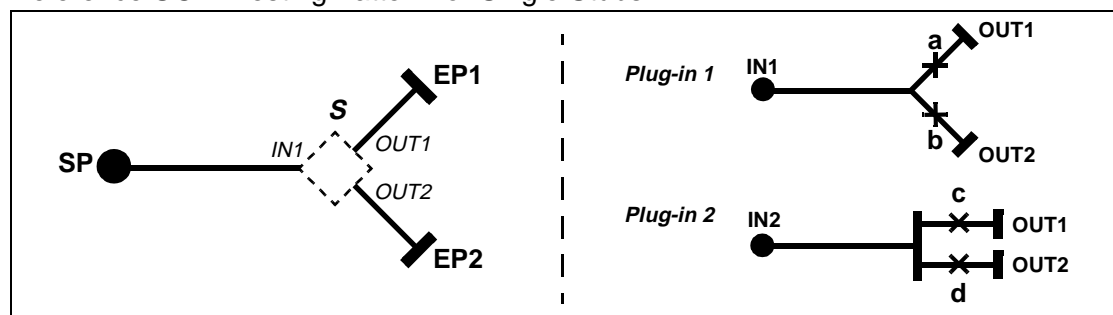
INTENT

To generate, for UCM paths that contain a single stub with plug-ins, test goals expressed in terms of sequentially linked start points, responsibilities, waiting places, end points, and other such events.

CONTEXT

The functionality under test is captured as a UCM path that contains a single stub. Various plug-ins may be included in dynamic stubs, and stubs can be nested. Figure 48 is used to illustrate relevant strategies. It shows a UCM containing a stub (left side) together with a set of plug-ins (right side).

FIGURE 48. Reference UCM: Testing Pattern for Single Stubs



FAULT MODEL

A test goal is a route from a start point to one or multiple end points. The fault model assumes that faults hide in a plug-in or in the selection of appropriate plug-ins in one stub (e.g. there could be a non-deterministic choice between two plug-ins). The coverage of all plug-ins in flattened maps, where the stub is substituted with appropriate plug-ins according to the binding relationships, would ensure that these faults are detected.

FORCES

By flattening a UCM that contains stubs and plug-ins, the resulting UCM no longer contains stubs or plug-ins, but it may contain alternatives, concurrent paths, loops, and multiple start points. Therefore, the testing patterns previously presented can be applied. However, a plug-in can be used in different

stubs (contexts). For dynamic stubs, the use of the selection policy in the flattening procedure can help reducing the number of valid combinations of plug-ins in a stub.

The following three strategies are ordered according to the number of plug-ins covered.

STRATEGY 5.A: SINGLE STUB — STATIC FLATTENING

The stub **S** in Figure 48 is assumed to be static (would be shown as a diamond without the dashed lines), hence it contains only one plug-in without any selection policy. The selection of test goals is based on the simple substitution of the stub by its plug-in. The testing patterns seen so far (alternative, concurrent, loop, and multiple start points) can then be used on the flattened map.

EXAMPLE 1: if *Plug-in 1* is used in **S**, together with Strategy 1.B: Alternative — All paths:

{<SP, a, EP1>, <SP, b, EP2>}

EXAMPLE 2: if *Plug-in 2* is used in **S**, together with Strategy 2.A: Concurrent — One combination:

{<SP, c, d, EP2, EP1>}

CONSEQUENCE

For static stubs, all the plug-ins are obviously covered by the strategy (since there is only one plug-in). The resulting path coverage is as good as that offered by the strategies used on the flattened map.

STRATEGY 5.B: SINGLE STUB — DYNAMIC FLATTENING, SOME PLUG-INS

The stub **S** in Figure 48 is dynamic, hence it contains many plug-ins and a selection policy. In this strategy, the selection of test goals is based on the substitution of the stub by *a subset* of the plug-ins bound to that stub. Multiple flattened maps may result from this procedure. The testing patterns seen so far (alternative, concurrent, loop, and multiple start points) can then be used on each of the flattened maps. This pattern is useful when the same plug-in is bound to many stubs. If this plug-in is already tested in another stub, then the tester might wish not to cover it again, even if the context is different.

EXAMPLE: Assume this selection policy, where both plug-ins are bound to the stub **S**:

if (condition==true) **then** use *Plug-in 1* **else** use *Plug-in 2*.

Now, suppose that *Plug-in 1* is already tested elsewhere in another stub. This strategy

would suggest that *Plug-in 2* be used in **S**, but not *Plug-in 1*. One flattened map would result, on which Strategy 2.A: Concurrent — One combination can be used:

{<SP, c, d, EP2, EP1>}

CONSEQUENCE

All plug-ins are tested, but possibly in different stubs. The plug-ins are not tested in all the contexts where they belong. The resulting path coverage is as good as that offered by the strategies used on the flattened maps. This strategy leads to fewer test goals than Strategy 5.C because some combinations of plug-ins in stubs are not covered.

STRATEGY 5.C: SINGLE STUB — DYNAMIC FLATTENING, ALL PLUG-INS

The stub **S** in Figure 48 is dynamic, hence it contains many plug-ins and a selection policy. In this strategy, the selection of test goals is based on the substitution of the stub by *all* the plug-ins bound to that stub. Multiple flattened maps may result from this procedure. The testing patterns seen so far (alternative, concurrent, loop, and multiple start points) can then be used on each of the flattened maps. This pattern is useful when a high coverage of the plug-ins in all their potential contexts (stubs) is required.

EXAMPLE: Assume this selection policy, where both plug-ins are bound to the stub **S**:

if (condition==true) **then** use *Plug-in 1* **else** use *Plug-in 2*.

A first flattened map would result from the use of *Plug-in 1* in **S**, and then Strategy 1.B: Alternative — All paths can be used. A second flattened map would result from the use of *Plug-in 2* in **S**, and then Strategy 2.A: Concurrent — One combination can be used. The final collection of goals is the union of those selected for each flattened map: {<SP, a, EP1>, <SP, b, EP2>, <SP, c, d, EP1, EP2>}

CONSEQUENCE

All plug-ins are tested in each of the stubs where they are bound (contexts). The resulting path coverage is as good as that offered by the strategies used on the flattened maps.

KNOWN USES

No other uses at this point.

RELATED PATTERNS

Flattened maps may contain multiple start points, in which case Testing Pattern 4—MULTIPLE START POINTS should be used. UCMs with causally linked dynamic stubs are assumed to have been flattened during a previous step using Testing Pattern 6—CAUSALLY LINKED STUBS.

6.3.9 Testing Pattern and Strategies for Causally Linked Stubs

NAME: TESTING PATTERN 6—CAUSALLY LINKED STUBS

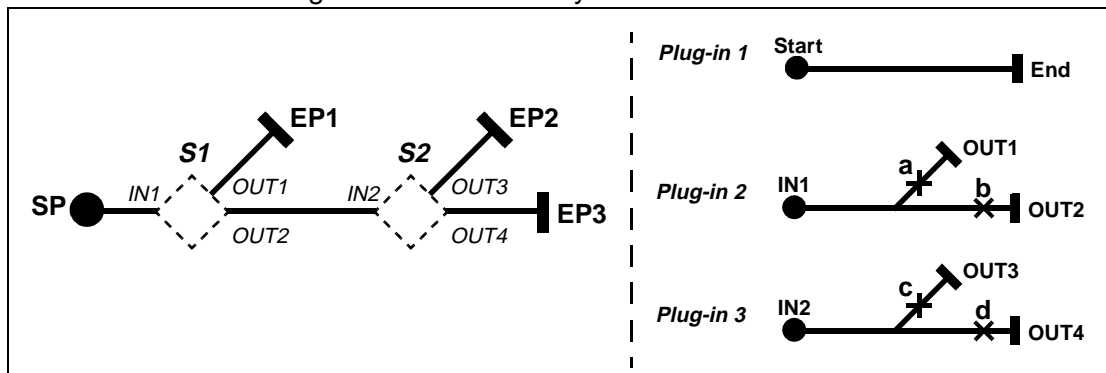
INTENT

To generate, for UCM paths that contain causally linked dynamic stubs (e.g. in sequence), test goals expressed in terms of sequentially linked start points, responsibilities, waiting places, end points, and other such events.

CONTEXT

The functionality under test is captured as a UCM path that contains multiple causally linked dynamic stubs. Various plug-ins may be included, and stubs can be nested. Stubs are assumed to have a *default* plug-in representing the absence of specific features at this location. Plug-ins are used to capture *features* that deviate from the basic behaviour. Figure 48 is used to illustrate relevant strategies. It shows a UCM containing two stubs (left side) together with their plug-ins (right side). *Plug-in 1* is the default behaviour for both stubs **S1** (End is bound to OUT2) and **S2** (End is bound to OUT4). *Plug-in 2* belongs to **S1** whereas *Plug-in 3* is used by **S2**.

FIGURE 49. Reference UCM: Testing Pattern for Causally Linked Stubs



FAULT MODEL

A test goal is a route from a start point to one or multiple end points. The fault model assumes that faults result from combinations of plug-ins bound to causally linked stubs (potentially a feature interaction). The coverage of all combinations of plug-ins in flattened maps, where all stubs are substituted with appropriate plug-ins according to the binding relationships, would ensure that these faults are detected.

FORCES

By flattening a UCM that contains stubs and plug-ins, the resulting UCM no longer contains stubs or plug-ins, but it may contain alternatives, concurrent paths, and loops. Therefore, the testing patterns previously presented can be applied. However, there might be many possible combinations, especially in situations where a UCM has many levels of nested stubs and plug-ins or where stubs have numerous plug-ins. Generating test goals for all combinations leads to more thorough test suites, but at a higher cost. For dynamic stubs, the use of the selection policy in the flattening procedure can help reducing the number of combinations of plug-ins in a stub and across causally linked stubs.

The following three strategies are ordered according to the likelihood of finding undesirable interactions between plug-ins (from low-yield test goals to high-yield test goals). Note that these strategies are *not* mutually exclusive and can be used in combination.

STRATEGY 6.A: CAUSALLY LINKED STUBS — DEFAULT BEHAVIOUR

The default plug-ins are used in all the causally linked stubs, and the selection of test goals is performed based on the resulting flattened map, where the testing patterns seen so far (alternative, concurrent, loop, and multiple start points) can be used.

EXAMPLE: *Plug-in 1* is used in **S1** and in **S2**. With Strategy 1.B: Alternative — All paths:
{<SP, EP3>}

CONSEQUENCE

Targets the validation of default behaviour, in the absence of features captured by untested plug-ins. The resulting path coverage is as good as that offered by the strategies used on the flattened map. The coverage of plug-in combinations is very weak (low-yield test goals).

STRATEGY 6.B: CAUSALLY LINKED STUBS — INDIVIDUAL FEATURES

One feature (plug-in) is used in one stub, while the default plug-ins are used in all the other causally linked stubs. Multiple flattened maps may result from this procedure, one for each feature. The selection of test goals is performed based on the resulting flattened maps, where the testing patterns seen so far (alternative, concurrent, loop, and multiple start points) can be used.

EXAMPLE: A first flattened map results from *Plug-in 1* being used in **S1** and *Plug-in 3* in **S2**. The second map results from *Plug-in 2* being used in **S1** and *Plug-in 1* in **S2**. With Strategy 1.B: Alternative — All paths, the overall set of test goals is:
{<SP, a, EP1>, <SP, b, EP3>, <SP, c, EP2>, <SP, d, EP3>}

CONSEQUENCE

Targets the validation of individual feature behaviour, in the absence of other features. The resulting path coverage is as good as that offered by the strategies used on the flattened maps. The coverage of plug-in combinations is still weak (low-yield test goals).

STRATEGY 6.C: CAUSALLY LINKED STUBS — FEATURE COMBINATIONS

All combinations of two or more features (plug-ins) are used in causally linked stubs. Multiple flattened maps may result from this procedure. The selection of test goals is performed based on the resulting flattened maps, where the testing patterns seen so far (alternative, concurrent, loop, and multiple start points) can be used.

EXAMPLE: The flattened map results from *Plug-in 2* being used in **S1** and *Plug-in 3* in **S2**. With Strategy 1.B: Alternative — All paths, the set of test goals becomes:

{<SP, a, EP1>, <SP, b, c, EP2>, <SP, b, d, EP3>}

CONSEQUENCE

Targets the validation of feature interactions resulting from combination of plug-ins in different dynamic stubs. The resulting path coverage is as good as that offered by the strategies used on the flattened maps. The coverage of plug-in combinations is good (high-yield test goals). The most interesting test goals are those different from the goals generated by Strategy 6.A and Strategy 6.B, i.e. {<SP, b, c, EP2>, <SP, b, d, EP3>} in the example above. Some of these interactions might be classified as *undesirable* by designers and requirements engineers; they should be prevented by the use of appropriate conditions and selection policies at the UCM level, and the corresponding test goals should be used as a basis for the generation of rejection test cases.

KNOWN USES

No other uses at this point.

RELATED PATTERNS

Flattened maps may still contain individual stubs, to be handled by Testing Pattern 5—SINGLE STUB. Rather than covering all possible combinations of features, checking all pair-wise combinations, as suggested by Williams [375], could represent a sensible and cost-effective solution when more than two stubs are causally linked.

6.3.10 DISCUSSION

This section discusses three topics related to UCM-oriented testing patterns. First, it provides a link between testing patterns and the LOTOS testing theory. Second, it briefly compares these patterns to closely related test patterns written by Binder. Last, the test purposes generated through our testing patterns are compared to other types of test purposes found in the literature.

Relating Testing Patterns to the LOTOS Testing Theory

Test goals are selected by applying testing patterns on a given UCM. If individual UCMs were formal models, then our testing patterns could be linked to the LOTOS testing theory. Under the assumption of the existence of a LOTOS interpretation for each UCM obtained from the functional requirements, which could be constructed according to the guidelines described in the previous chapter, abstract causal sequences could be compared to reductions of canonical testers. Table 18 gives additional definitions for unbound UCM paths (without components) that can be described in terms of LTSs:

TABLE 18. Notation for UCMs Interpreted in LOTOS

<i>Notations</i>	<i>Definitions</i>
LOTUCMS	Set of LOTOS interpretations (behaviour expressions) of UCM paths used for the construction of a specification under test. $\text{LOTUCMS} \subseteq \text{SPECS}$
LU_n	LOTOS interpretations of UCM_n . $LU_n \in \text{LOTUCMS}$.

There may be many UCMs involved in the construction of a LOTOS prototype. Each of them could be transformed, while abstracting from underlying components, into a separate LOTOS model LU_n . The generation of sound test cases for the integrated prototype model can be reinforced by checking that test goals (abstract sequences) selected from UCM_n respect the behaviour described by LU_n . In other words, test goals used in acceptance test cases should be reductions (red) of the canonical tester of the corresponding individual model LU_n . We suggest this as a *soundness property* that must be satisfied by our goals. This is described by Property 1, where UCM_n is used for the construction of LU_n and for the selection of test goals in a test group TG_n (see Table 15).

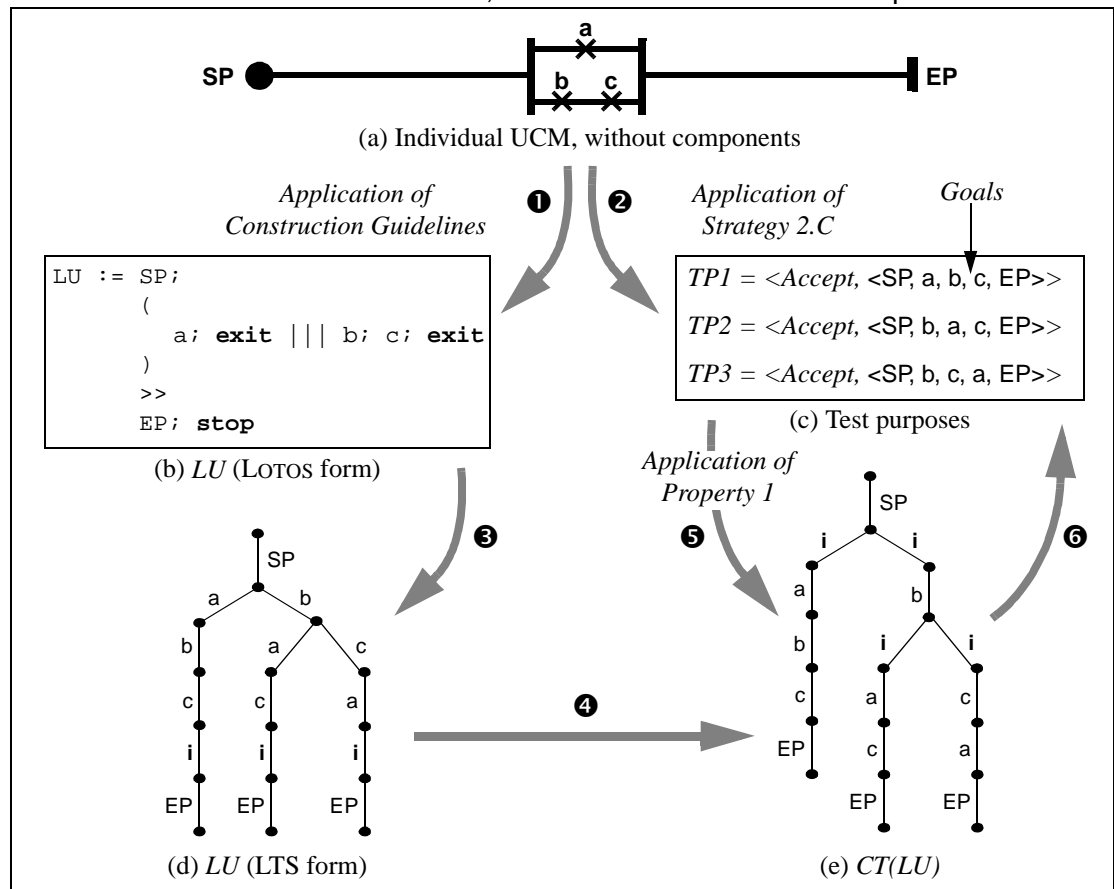
$$\forall LU_n, \forall T_x, T_x \in \text{ACCEPT}(TG_n) \Rightarrow \text{Goal}(TP(T_x)) \text{ red } CT(LU_n) \quad \text{(PROPERTY 1)}$$

As for rejection test cases in the test suite TS , their goals must not be reductions of the canonical tester of any LU_n (Property 2).

$$\forall LU_n, \forall T_x, T_x \in \text{REJECT}(TS) \Rightarrow \neg(\text{Goal}(TP(T_x)) \text{ red } CT(LU_n)) \quad \text{(PROPERTY 2)}$$

To illustrate the effect of Property 1, the UCM of Figure 45 is reused as an example in Figure 50, where several steps are identified:

FIGURE 50. Individual UCM in LOTOS with its LTS, Canonical Tester and Test Purposes



- ❶ A LOTOS behaviour expression LU (b) is constructed from an individual UCM without components (a) according to the guidelines in Section 5.2.
- ❷ According to Section 6.3.5, Strategy 2.C can be applied to the same UCM (a) in order to generate goals for test purposes (the latter also include test types, i.e. accept or reject) (c).
- ❸ LU (b) can be transformed into a labelled transition system (d) (see Section 2.3.5).
- ❹ A canonical tester $CT(LU)$ (e) is generated from the LTS (d) (discussed in Section 2.3.6).

- ⑤ According to Property 1, the goal of each test purpose in the (acceptance) test group generated from the UCM needs to be a reduction of $CT(LU)$. Indeed:

$Goal(TP1) \underline{\text{red}} CT(LU)$, $Goal(TP2) \underline{\text{red}} CT(LU)$, and $Goal(TP3) \underline{\text{red}} CT(LU)$.

Step ⑤ shows that the test goals are sound, and hence the confidence in the soundness of the LOTOS test cases to be generated is increased. The soundness of the test cases is a property that is required for the val relation to hold. Property 1 can therefore help ensuring that testing patterns and their strategies generate sound test purposes.

Applications of Property 2 are of a similar nature. If a test goal is intended to be used for generating a rejection test case, then this goal must not be a reduction of the canonical tester of any UCM considered in the requirements.

The existence of a canonical tester for an individual UCM, which is more manageable in terms of complexity and size than the canonical tester of the whole integrated set of UCMs, also enables another strategy to be used on top of testing patterns for the selection of test purposes. Step ⑥ suggests that test purposes can be derived directly from $CT(LU)$, provided that they are irreducible reductions of this canonical tester. Conventional LOTOS test selection techniques could therefore potentially be used at this point, but these techniques are outside the scope of this thesis. Alternatively, more detailed test purposes (represented as trees) could be generated from a goal combined to LU through techniques like Probert and Wei's Non-deterministic Ripple Sets [295]. However, such test purposes would no longer be sequential and could include inconclusive verdicts, two things that augment the difficulty of assessing validity and of establishing simple diagnostics.

Note that for non-deterministic choices, the application of some testing patterns may lead to test goals that violate Property 1. For example, if a UCM OR-fork is guarded by two overlapping conditions ($X > 3$ and $X < 5$), selecting $X = 4$ for any of the two path would result in a test goal that is *not* a reduction of the canonical tester of the UCM interpreted in LOTOS, and to a *may pass* verdict at testing time. This could be detected by step ⑤ before applying the test case to the prototype. Solutions

could involve modifying the conditions in the UCM (see Section 6.4.3) or using step ⑥ to augment the test case until it satisfies Property 1. The resulting test would become a tree with alternatives and would no longer be linear.

Comparison with Binder's Test Patterns

In his book, Binder presented test patterns for object-oriented systems [51]. To the knowledge of the author (and to mine), this is the most comprehensive collection of such patterns. Multiple levels of details are addressed, from classes to applications to regression testing.

Two of his test patterns stand out as being related to the ones defined in this chapter:

- *Round-trip Scenario Test* [50]: its intent is to extract a control flow model from a UML sequence diagram and then develop a path set that provides minimal branch and loop coverage (similar to Testing Patterns 1 and 3). The proposed solution, based on heuristics, leads to fairly synthetic tests. Our testing patterns are more flexible and better tailored to UCMs, and they lead to test goals and test cases of higher quality and closer to requirements. Binder's solution does not consider concurrency (partial orders) and sub-models (e.g. plug-ins). The UCM-oriented test patterns handle such constructs and provide strategies for coping with related issues such as scalability and state explosion which are avoided altogether by Binder. Since his book focus on OO models and programs that are mostly sequential in nature, topics relevant to concurrent and distributed systems (such as telecommunications systems) are covered only to a very limited extent.
- *Extended Use Case Test*: its intent is to develop a system-level test suite by modeling essential capabilities as extended use cases. Binder cites many benefits related to the use of use cases: close to requirements, developed by various stakeholders, leverage design information, capture main functionalities and relationships, and so on. UCMs provide similar advantages, together with a better management of several disadvantages enumerated in this pattern: difficult to find the right level of abstraction (UCMs provide an appropriate level of abstraction for early design stages), performance is usually not specified with use

cases (it can be with UCMs), and UML *extend* and *include* relationships for use cases are difficult to flatten (flattening is simpler with the UCM stub/plugin mechanism). The solution proposed is very generic and is not of much help for the generation of concrete test goals, whereas the UCM-oriented testing pattern language offers a more systematic way of generating test goals.

UCM-oriented testing patterns, language and strategies hence address some of the weaknesses of Binder's patterns related to the testing of telecommunications systems.

Comparison with Related Types of Test Purposes

In this thesis, a test purpose is a pair $\langle type, goal \rangle$ where the test type is either acceptance (must test) or rejection (reject test). The testing patterns target the selection of functional test goals from UCMs capturing requirements or high-level designs. These test goals are abstract sequences of events corresponding to UCM start points, responsibilities, waiting places/timers, and end points. The test cases generated from such test purposes aim to validate formal specifications (prototypes) that integrate functionalities of complex telecommunications systems expressed as Use Case Maps.

Test goals and test purposes have often been used in various contexts. In general, they are combined to formal specifications (assumed to be correct and valid) in order to generate conformance test cases. Here is a brief comparison between our test purposes and related ones:

- The **Conformance Testing Methodology and Framework** (CTMF) [193] use test purposes to describe well-defined test objectives, focusing on a single conformance requirements or a set of related conformance requirements, in a rather informal way using prose descriptions. How to generate these test purposes is outside the scope of this framework.
- The **Formal Methods in Conformance Testing** (FMCT) framework [196] defines a generic and formal test purpose as an abstract requirement or behaviour accompanied by an implementation or satisfaction relation. Such test purpose hence describes the set of implementation models that should contain this behaviour or requirement. Our test pur-

- pose is essentially an instantiation of this concept, where the requirement is the test goal and the satisfaction relation is val combined with the test type.
- **Grabowski et al.** [158], **Ek et al.** [123], and more recently **Probert et al.** [297][298] specify test purposes as MSCs and use them to guide the generation of TTCN test cases from an SDL specification. This is one of the techniques currently supported by SDL tools (e.g. Telelogic's). Due to their graphical nature, MSCs represent an attractive way of capturing test goals. Unfortunately, how these MSCs are generated is not discussed (they are assumed to be given or validated by customers). Moreover, these test purposes target *May* tests as they describe successful sequential traces, and the SDL specification helps computing other branches that fail or that are inconclusive in the corresponding test case. The test purposes emphasized in this thesis lead to *Must* tests and *Reject* tests, which lead to simpler diagnostics and clearer assessments.
 - **Probert and Wei** [295] use prose test purposes in an algebraic semantic context (*Non-deterministic Ripple Sets* — NRS) to guide the generation of test cases for the validation and conformance checking of implementations. Much in line with test goals derived from UCMs, their concept of test purpose implies a specific, biased set of paths (specified using *choice patterns*) that have the potential to be traversed. Again, a correct and possibly non-deterministic service specification (LTS) is required, and the origin of the test purposes is not discussed (assumed to come from the requirements). The leaves of the resulting test case (trees) also have to be tagged manually with a verdict (pass, fail, or inconclusive).
 - **Bertolino et al.** [48] use “architectural descriptions” of systems in abstract LTS form and derive test purposes directly from them in order to test conformance of implementations against software architectures. The difficulty here resides in the complexity of the LTS, which becomes very large (or even infinite) for any realistic telecommunications system. Also, the LTS is assumed to be correct, whereas our test goals aim to validate the LOTOS/LTS specification that integrates all the functionalities expressed with UCMs.
 - **Jard et al.** [214][217] use graphs (partial LTSs) as test goals and use them to generate or validate test cases using a formal specification (in SDL or LOTOS) and the TGV toolkit. In

general, how these test goals are generated remains unclear (although model checking is presented as a candidate) and the formal specification is again assumed to be correct.

- **Charfi** uses a path traversal algorithm adapted from Miga's [258] to traverse a UCM model (augmented with key annotations in LOTOS) and extract test goals [91]. These goals are then used, in combination with a LOTOS specification and the TGV toolkit, to generate acceptance test cases. Although the level of automation is rather high, the quantity of test goals generated becomes difficult to manage for any complex collection of UCMs.

Note that a recent addition to the UCM language called *scenario definitions* [84][258] offers another alternative to UCM designers for defining test goals using an initialization of global path variables. Applications to the generation of tests is however future work.

6.3.11 Section Summary

UCM scenarios that describe the requirements or the design should also be used for the selection of appropriate test goals. This section argues that testing patterns can be used as a sensible and semi-formal approach to test selection that fits the level of abstraction targeted by a semi-formal notation like Use Case Maps.

This section provides a UCM-oriented testing pattern language and an appropriate template to support this selection. Six testing patterns and a total of twenty-five coverage-directed strategies based on UCM constructs were defined, motivated, and illustrated. The patterns represent a traceable link between functional requirements and test cases. They produce abstract causal sequences (test goals) which in turn can be used for the generation of sound LOTOS test processes (acceptance and rejection). These patterns are independent of any underlying structure of components, hence they apply to a wide variety of systems.

A relation between testing patterns and the LOTOS testing theory was established. The testing patterns were compared to those of Binder's, and the resulting test goals and test purposes were briefly compared with relevant techniques that also use test purposes in the telecommunications system domain.

Although UCM-based testing patterns represent a major step in the selection of appropriate test cases, additional information still needs to be considered for the generation of detailed LOTOS processes from abstract sequences. This topic is addressed in the next section.

6.4 Complementary Strategies and Test Case Generation

In order to generate LOTOS test cases from Use Case Maps, test goals first need to be selected using the testing patterns defined in the previous section. However, several problems remain, such as linking an abstract sequence to the design decisions made during the construction of the LOTOS prototype (Section 6.4.1), selecting appropriate values (Section 6.4.2), dealing with incompleteness and non-determinism (Section 6.4.3), and generating rejection tests (Section 6.4.4). The current section proposes several complementary strategies as elements of solution for the UCM-LOTOS testing framework.

6.4.1 From Test Goals to Test Cases

Dealing with telecommunications systems often involves the use of components and data. The construction of a LOTOS specification from UCMs is based on several guidelines (Section 5.2), but also on design decisions related to the definition of data, messages, parameters, visibility of responsibilities and events, etc. Beside the obvious consistency required between the gate names used in the test cases and in the specification (CG-1), these design decisions influence the generation of a LOTOS test case from a test purpose. More precisely, the following elements need to be taken into consideration (the relevant construction guidelines are cited):

- **Visibility:** due to the presence of components in the original set of UCMs, some responsibilities and events from the abstract sequence may be hidden at some level. Common situations involve responsibilities and events located inside a component or messages/interactions between two given components (CG-5.c). Timers (CG-4.a) and failure points (CG-4.c) may also have visible events in order to improve controllability and testability. Since a LOTOS process can only synchronize on visible gates, hidden responsibilities and events should not be part of the test cases.

- **Parameters:** when a responsibility or an event is specified with parameters in the LOTOS prototype, these parameters need to be included and instantiated in the test case. Common cases include the use of dynamic responsibilities (CG-4.d) and component interfaces (CG-5.b). Parameter values need to conform to the type definitions (CG-8).
- **Messages:** if visible messages are used to specify the causality relationship between two responsibilities in the abstract sequence under consideration (CG-7), then these messages need to be included in the test case as well. Shared responsibilities (CG-7.a) and direction of messages also need to be considered (CG-7.b).
- **Preambles:** if a precondition is attached to the UCM from which a test goal was selected, then this precondition needs to be satisfied by the test case. To bring the system from an initial state to a state that satisfies this precondition, an appropriate preamble might be necessary.
- **Verification steps:** if a postcondition is attached to the UCM from which a test goal was selected, then this postcondition needs to be tested by the test case. To do so, an appropriate sequence of verification steps might be necessary.

Note that stubs (CG-3) and the integration of path segments (CG-2 and CG-6) have little impact on how to go from test goals to test cases because most of the related decisions are already embedded in the goals generated from the testing patterns.

Additional selection strategies, discussed in the next two sections, can guide the selection of suitable data values for parameters.

6.4.2 Strategies for Value Selection

When parameters need to be instantiated, the values must satisfy the guards and selection predicates accompanying the LOTOS events that correspond to the selected goal. However, a possibly large number of such values might exist, and selecting one combination might not be sufficient to ensure a good coverage of parameters. If some data-oriented coverage is required, multiple test cases could be created for each test goals. Conventional strategies related to traditional black-box testing can be used at

this point [44]. Two of the best-known strategies are equivalence classes and boundary interior analysis [267]. Note that the UCM-LOTOS testing framework does not focus on data-oriented coverage.

6.4.3 Completeness and Determinism Issues

UCMs may contain some non-deterministic behaviour due to overlapping conditions. For example, suppose a two-branch OR-fork where two conditions $C1$ and $C2$ are located, one per branch. These conditions use variables whose type is a subset of natural numbers ($[0..5]$). For these conditions, there are four cases where the generation of test cases can be influenced by the lack of completeness and/or determinism:

- **Complete and disjoint conditions:** assume that $C1$ is $X > 3$ and $C2$ is $X \leq 3$
The simplest case. Any value will lead to the selection of one specific alternative.
- **Complete conditions with conjunction:** assume that $C1$ is $X > 3$ and $C2$ is $X < 5$
 $X = 4$ is a value that will result in a non-deterministic execution (and possibly to a may pass verdict).
- **Incomplete and disjoint conditions:** assume that $C1$ is $X > 3$ and $C2$ is $X < 3$
 $X = 3$ is a value that will cause a deadlock.
- **Incomplete conditions with conjunction:** assume that $C1$ is $0 < X < 3$ and $C2$ is $1 < X < 5$
 $X = 2$ is a value that will result in a non-deterministic execution (and possibly to a may pass verdict). $X = 5$ is a value that will cause a deadlock.

The choice of a specific value in a test case can influence the resulting verdict for a given test goal. The second case indicates a UCM where refinement of conditions may be needed in order to get a deterministic specification. The last two cases are symptoms of a problematic UCM. For OR-forks in general and for selection policies in dynamic stubs, Parnas tables can help to assess, at specification time, that a collection of conditions is deterministic and complete [276]. Such tables could be used in combination with testing patterns for alternatives and for stubs/plugin-ins (Sections 6.3.4 and 6.3.8).

6.4.4 Strategies for Rejection Test Cases

Deriving rejection test cases from requirements is a challenging task. If some scenarios or properties are explicitly forbidden (e.g. as suggested by Harel and his play-in scenarios [167]), then UCMs can be used to capture them, and then the proposed testing patterns and strategies, which mainly target acceptance test cases, could be used to derive rejection test cases. However, such forbidden scenarios are usually missing from the requirements. Often in telecommunications systems, anything can go wrong and the number of potential rejection test cases is unlimited. Consequently, selection strategies for rejection test cases appear just as necessary as for acceptance test cases.

Several rejection test case selection strategies are introduced in this section. They are inspired by various sources and techniques, including: invalid output, invalid input resulting in a valid output (in black-box testing [267]), use of explicit safety properties (i.e., something bad should not happen), and “dirty” testing [44], which is automatable to some extent for testing robustness of implementations [234]. Our strategies adapt some of these techniques and ideas to the LOTOS-UCM testing framework. These strategies focus on high-level specifications and may not all be relevant for testing real implementations. For instance, implementations are usually required to handle errors with detection routines and exceptions (hence, acceptance test cases should cover these situations), whereas high-level specifications (such as the ones we use in this thesis) may defer such treatment to a later stage in the design process.

We suggest the following non-exhaustive list of five strategies:

R1. Forbidden scenarios from requirements: several rejection test cases can be generated directly from requirements or safety properties. The generation is usually straightforward when forbidden scenarios are defined as explicit requirements.

R2. Use of testing patterns: several goals generated from testing patterns and their strategies can only be used for rejection test cases. The most notable ones are:

- Unsatisfiable set of conditions in successive alternatives found in OR-forks (Strategy 1.C).
- $m-1$ iterations in a loop, where m is the minimal number of iterations, or $n+1$ iterations in a loop, where n is the maximal number of iterations (Strategy 3.D).

- Redundant stimuli that cause two instances of the resulting event (Strategies 4.D and 4.E).
- Insufficient stimuli that still cause the resulting event (Strategies 4.F and 4.G).
- Unsatisfiable set of conditions attached to multiple start points (Strategies 4.A to 4.H).
- Unsatisfiable set of conditions found in selection policies of nested dynamic stubs (Strategies 5.B and 5.C)
- Unsatisfiable set of conditions found in selection policies of causally linked dynamic stubs, e.g. undesirable feature interactions (Strategy 6.C).

R3. Incomplete conditions: several collections of conditions can be incomplete. Selecting a value that is not covered by any of these conditions, as seen in Section 6.4.3, leads to a rejection test case. This however is usually a symptom of a problem in the UCMs.

R4. Off-by-one value: an invalid output for a given set of valid inputs translates, in UCM terms, in an incorrect resulting event. Since LOTOS allows non-deterministic behaviour, valid and invalid resulting events could be offered simultaneously to a test case. Invalid resulting events can be detected with a rejection test case by using the acceptance test case and by complementing its resulting values. The same LOTOS gate is used, but the accepted values are totally disjoint. For instance, using appropriate predicates, the resulting event `EP !Display !3` could be complemented into `EP !Display ?n:number [n ne 3]` or into `EP ?msg:MsgType ?n:number [(msg ne Display) or (n ne 3)]`. This fault model seems rather simple yet, in the absence of explicit forbidden scenarios, it increases the confidence that the expected result, usually found in a corresponding acceptance test, is the only one the system can offer. Specification languages such as SDL have explicit catch-all constructs (e.g. `OTHERWISE` clause) that can be used in similar situations.

R5. Off-by-one gate: similar to the off-by-one value strategy, only this time the gate name is modified. This gate mutation is particularly useful when values and gate splitting (interfaces) are not used, i.e. only the gate name represents the expected result. This strategy is not mutually exclusive with the off-by-one value strategy.

Some of these strategies will be further illustrated with the TTS example and with the experiments discussed in Chapter 8.

Rejection Test Cases and Incomplete Requirements

A difficulty appears when requirements are incomplete, which is to be expected especially in the initial stages of system development. An incomplete set of UCMs and validation test cases usually results from such situations. Some rejection test cases constructed following strategies such as **R2** to **R5** (described above) can be problematic as they might imply the rejection of a valid requirement that has not been specified explicitly. Accordingly, rejection test cases should be inspected by requirements engineers in order to assess their correctness and to confirm that they do not correspond to implicit and valid requirements. Incorrect rejection test cases can motivate modifications to the requirements, and hence to the specification.

Also, as requirements evolve, several constraints expressed as forbidden situations in the requirements may be relaxed to allow the creation of new functionalities and features. The list of rejection test cases needs to be revised accordingly and some of them might have to be removed.

6.5 Testing the TTS System

The TTS Use Case Maps, together with the defined testing patterns, enable the selection and generation of functional test cases for verifying that the prototype satisfies the UCM scenarios and for validating scenario integration. First, abstract causal sequences are selected from the UCMs (Section 6.5.1). Additional abstract sequences aiming to test the robustness of the prototype are selected in Section 6.5.2. All these abstract sequences are test goals, which are then transformed into LOTOS test processes in Section 6.5.3. The results of their execution are presented in Section 6.5.4.

6.5.1 Test Goals for TTS

Test purposes are composed of test goals extracted from the UCMs and of test types (accept or reject). Since many test goals may exist, guidance is required for selecting the ones most appropriate according to the test plan. Testing patterns can help in this selection process.

The integrated UCM view of the Tiny Telephone System (Figure 20) is the starting point where testing patterns can be applied. The three steps suggested in the UCM-oriented testing pattern language (Section 6.3.3) are used here.

The first step (Figure 41) consists in flattening the stubs contained in the TTS UCM. There are no causally linked stubs at the beginning, so Strategy 5.A is applied to stub ST, which is therefore replaced by its plug-in (TERMINATING). This results in an intermediate flattened map where two dynamic stubs (SO and SD) are causally linked. Since the number of plug-ins is small for both stubs, all three strategies in Testing Pattern 6 are applied. As a result, four combinations of stubs lead to four flat UCMs (i.e. without stubs).

The second step (Figure 42) handles flat maps that have multiple start points. This is not the case for any of the maps here, so Testing Pattern 4 is not used.

The third step (Figure 43) consists in handling the remaining UCM constructs (alternatives, concurrent segments, loops) for each flat map. Four test groups are created for the four combinations of stubs (flat maps), and each group will contain test goals that can be used to generate acceptance and/or rejection test cases.

Combination 1: SO = DEFAULT, ST = TERMINATING, SD = DEFAULT

This combination, resulting from Strategy 6.A applied to stubs SO and SD, corresponds to the basic call UCM of Figure 18. Strategy 1.B (Alternative — All paths) is applied to the OR-fork and Strategy 2.B (Concurrent — Some combinations) to the AND-fork. This leads to the following three test goals, where hidden responsibilities are marked by a star (*). Conditions found along the selected paths and in the selection policies are also enumerated (*A* is the originating party, *B* is the terminating party, and *C* is a third party). For all these sequences, *A* and *B* are not subscribed to any feature. Note that *A* is originally busy at the beginning (see the informal requirements in Section 4.3.1); *A* is not yet involved in any call, but *A* cannot receive any call request (this is similar to a situation where a user's phone is initially off hook).

- Abstract sequence: <req, vrfy*, prbs*, upd*, sig, ring>
 Constraints: $[not(has(A, OCS))]$, $[busy(A)]$, $[not(has(B, CND))]$, $[idle(B)]$
 Informally: *A* calls *B* (idle), ringback first.
- Abstract sequence: <req, vrfy*, prbs*, upd*, ring, sig>
 Constraints: $[not(has(A, OCS))]$, $[busy(A)]$, $[not(has(B, CND))]$, $[idle(B)]$
 Informally: *A* calls *B* (idle), ring first.
- Abstract sequence: <req, vrfy*, pbs*, sig>
 Constraints: $[not(has(A, OCS))]$, $[busy(A)]$, $[not(has(B, CND))]$, $[busy(B)]$
 Informally: *A* calls *B* (busy).

In these constraints, $has(A, OCS)$ means that *A* has subscribed to OCS, $busy(A)$ means *A* is busy, $idle(B)$ stands for *B* is idle, and so on.

Note that the application of Strategy 2.B to the AND-fork resulted in the coverage of all combinations of visible events (sig and ring), just as Strategy 2.C would have because hidden responsibilities are not used in test cases.

Combination 2: SO = DEFAULT, ST = TERMINATING, SD = CND

This combination, resulting from Strategy 6.B applied to stubs SO and SD, corresponds to the CND UCM of Figure 19(b). Again, Strategy 1.B is applied to the OR-fork and Strategy 2.B to the AND-fork. There are three test goals:

- Abstract sequence: <req, vrfy*, prbs*, disp, upd*, ring, sig>
 Constraints: $[not(has(A, OCS))]$, $[busy(A)]$, $[has(B, CND)]$, $[idle(B)]$
 Informally: *A* calls *B* (idle), displays, ring first.
- Abstract sequence: <req, vrfy*, prbs*, disp, upd*, sig, ring>
 Constraints: $[not(has(A, OCS))]$, $[busy(A)]$, $[has(B, CND)]$, $[idle(B)]$
 Informally: *A* calls *B* (idle), displays, ringback first.

- Abstract sequence: $\langle \text{req}, \text{vrfy}^*, \text{pbs}^*, \text{sig} \rangle$
 Constraints: $[\text{not}(\text{has}(A, \text{OCS}))], [\text{busy}(A)], [\text{has}(B, \text{CND})], [\text{busy}(B)]$
 Informally: A calls B (busy), no display.

Note that the last test goal is the same as the last one from the previous combination. Whether B has subscribed to CND or not is of no consequence because the selected path does not go through stub SD . Consequently, only the two first sequences will be used in the test suite.

Combination 3: $\text{SO} = \text{OCS}$, $\text{ST} = \text{TERMINATING}$, $\text{SD} = \text{DEFAULT}$

This combination, also resulting from Strategy 6.B applied to stubs SO and SD , corresponds to the OCS UCM of Figure 19(a). This time, Strategy 1.C (Alternative — All path combinations) is used on the two consecutive OR -forks, and Strategy 2.A (Concurrent — One combination) is applied to the AND -fork because the ordering between ring and sig is not critical and because the intermediate responsibilities involved are hidden. There are three new test goals:

- Abstract sequence: $\langle \text{req}, \text{chk}^*, \text{vrfy}^*, \text{prbs}^*, \text{upd}^*, \text{ring}, \text{sig} \rangle$
 Constraints: $[\text{has}(A, \text{OCS})], [\text{busy}(A)], [\text{allowed}(B)], [\text{not}(\text{has}(B, \text{CND}))], [\text{idle}(B)]$
 Informally: A calls B (idle), allowed.
- Abstract sequence: $\langle \text{req}, \text{chk}^*, \text{vrfy}^*, \text{pbs}^*, \text{sig} \rangle$
 Constraints: $[\text{has}(A, \text{OCS})], [\text{busy}(A)], [\text{allowed}(B)], [\text{not}(\text{has}(B, \text{CND}))], [\text{busy}(B)]$
 Informally: A calls B (busy), allowed but busy.
- Abstract sequence: $\langle \text{req}, \text{chk}^*, \text{pds}^*, \text{sig} \rangle$
 Constraints: $[\text{has}(A, \text{OCS})], [\text{busy}(A)], [\text{denied}(B)], [\text{not}(\text{has}(B, \text{CND}))], [\text{busy}(B)]$
 Informally: A calls B , denied.

The condition $\text{allowed}(B)$ implies that B is not part of A 's OCS list, whereas $\text{denied}(B)$ means the opposite.

Combination 4: SO = OCS, ST = TERMINATING, SD = CND

This last combination, resulting from Strategy 6.C applied to stubs SO and SD, allows for the detection of undesired behaviour when both features are active, an interesting case that was not explicitly considered when capturing the requirements with individual UCMs. The following test goals represent desired fine-grained liveness properties which might be violated by the prototype where both features are integrated. Strategy 1.B is used for both OR-forks and Strategy 2.A for the AND-fork. Three test goals are hence sufficient, but two of them are similar to sequences defined for previous combinations:

- Abstract sequence: <req, chk*, vrfy*, prbs*, disp, upd*, ring, sig>
 Constraints: $[has(A, OCS)], [busy(A)], [allowed(B)], [has(B, CND)], [idle(B)]$
 Informally: A calls B (idle), allowed, displays.
- Abstract sequence: <req, chk*, vrfy*, pbs*, sig>
 Constraints: $[has(A, OCS)], [busy(A)], [allowed(B)], [has(B, CND)], [busy(B)]$
 Informally: A calls B (busy), allowed but busy. Already covered.
- Abstract sequence: <req, chk*, pds*, sig>
 Constraints: $[has(A, OCS)], [busy(A)], [denied(B)], [has(B, CND)], [busy(B)]$
 Informally: A calls B, denied. Already covered.

Combining the Test Goals

A total of nine original test goals were extracted from the integrated UCM (Table 19). The use of testing patterns led to three duplicate goals which will not be included in the test suite. Because test goals are described at a high level of abstraction, duplicate test goals are easier to detect and eliminate than duplicate test cases and this elimination results in a more optimized test suite.

TABLE 19. Test Goals Extracted from UCMs Through Testing Patterns

<i>Test Group #</i>	<i>Stub SO</i>	<i>Stub ST</i>	<i>Stub SD</i>	<i>Test Goal #</i>	<i>Abstract Sequence</i>
1	DEF.	TERM.	DEF.	1	<req, vrfy*, prbs*, upd*, sig, ring>
				2	<req, vrfy*, prbs*, upd*, ring, sig>
				3	<req, vrfy*, pbs*, sig>
2	DEF.	TERM.	CND	4	<req, vrfy*, prbs*, disp, upd*, ring, sig>
				5	<req, vrfy*, prbs*, disp, upd*, sig, ring>
3	OCS	TERM.	DEF.	6	<req, chk*, vrfy*, prbs*, upd*, ring, sig>
				7	<req, chk*, vrfy*, pbs*, sig>
				8	<req, chk*, pds*, sig>
4	OCS	TERM.	CND	9	<req, chk*,vrfy*,prbs*, disp, upd*,ring, sig>

Test purposes include these goals together with test types, i.e. acceptance or rejection. All the test goals seen here will be used to create test cases of both types.

6.5.2 Further Test Goals for Robustness Testing

It should be emphasized that one should not limit the testing to the goals set through testing patterns. Further goals can be defined for other types of requirements (e.g. robustness), concurrent scenarios along the same paths, or data values prone to errors (e.g. using boundary analysis). Risk analysis [55], business goals and non-functional requirements can be used to suggest scenarios useful as robustness tests. The following five test goals are used as an illustration of such cases:

- Abstract sequence: <req, vrfy*, pbs*, sig>
Constraints: $[not(has(A, OCS))]$, $[busy(A)]$
Informally: *A* calls *A* (busy), should return a busy signal.
- Abstract sequence: <req, vrfy*, upd*, ring, sig, req, vrfy*, pbs*, sig>
Constraints: $[not(has(A, OCS))]$, $[busy(A)]$, $[not(has(B, CND))]$, $[idle(B)]$, $[busy(C)]$
Informally: *A* calls *B* (idle), then *C* calls *B* (busy). The busy state of *B* should be set.

- Abstract sequence: <req, req, vrfy*, vrfy*, pbs*, pbs*, sig, sig>
 Constraints: [not(has(A, OCS)), [busy(A)], [busy(B)]
 Informally: A calls B (busy) while B calls A (busy).
- Abstract sequence: <req, chk*, pds*, sig>
 Constraints: [has(A, OCS)], [busy(A)], [denied(A)]
 Informally: A calls A, denied (A is on its own OCS list).
- Abstract sequence: <req, chk*, pds*, sig, req, chk*, pds*, sig>
 Constraints: [has(A, OCS)], [busy(A)], [denied(B)], [denied(C)]
 Informally: A calls B, denied, then A calls C, denied (tests OCS list longer than 1 element)

These test goals are summarized in Table 20, where the “-” symbol is used as a don’t care value, i.e. any plug-in could be used.

TABLE 20. Further Test Goals for Robustness

Test Group #	Stub SO	Stub ST	Stub SD	Test Goal #	Abstract Sequence
5	DEF.	TERM.	-	10	<req, vrfy*, pbs*, sig>
	DEF.	TERM.	DEF.	11	<req, vrfy*, upd*, ring, sig, req, vrfy*, pbs*, sig>
	DEF.	TERM.	-	12	<req, req, vrfy*, vrfy*, pbs*, pbs*, sig, sig>
	OCS	-	-	13	<req, chk*, pds*, sig>
	OCS	-	-	14	<req, chk*, pds*, sig, req, chk*, pds*, sig>

6.5.3 Test Cases Generation

Now that the test goals are available, LOTOS test cases can be generated by considering the points cited in Section 6.4.1. Multiple acceptance and rejection test cases can be generated for each goal. However, for the sake of simplicity and because many decisions related to the choice of values are implicit, only one test case of each type will be generated for each test goal.

The test goal #1 (Table 19), composed of the abstract causal sequence <req, vrfy*, prbs*, upd*, sig, ring>, will be used as an example. Since the actions annotated by a star (*) are hidden, they will not

be part of the test case. The constraints associated to this test goal ($[not(has(A, OCS))]$, $[busy(A)]$, $[not(has(B, CND))]$, $[idle(B)]$) require a preamble that will initialize the SUT with users A and B , their subscribed features, and their initial state. In terms of the current prototype, this translates to:

```
init !userA !Insert(BC, EmptyFList) ?dummy:UserList !busy;
init !userB !Insert(BC, EmptyFList) ?dummy:UserList !idle;
```

The dummy value identifier used here for the OCS screening list is a don't care value because these users are not subscribed to OCS.

The test body is composed of three events corresponding to req, sig, and ring:

```
req !userA !userB;
sig !userA !ringBack;
ring !userB;
```

These five LOTOS actions represent an acceptance test case. A rejection test case can be generated by choosing one of the strategies discussed in Section 6.4.4. For instance, the application of the “Off-by-one value” strategy leads to a mutation of the last action into:

```
ring ?dummy:User [dummy ne userB]
```

This action means that a phone other than B 's is ringing. Because the acceptance test case and the rejection test case differ only by their last event, they can be regrouped into a single test process. As discussed in Section 6.2.2 (page 178), such a combined test process becomes an acceptance test:

```
process t1 [req,ring,sig,disp,init,reject,success]: noexit :=
  init !userA !Insert(BC, EmptyFList) ?dummy:UserList !busy;
  init !userB !Insert(BC, EmptyFList) ?dummy:UserList !idle;
  req !userA !userB;
  sig !userA !ringBack;
  (
    ring !userB; success; stop
    []
    ring ?dummy:User [dummy ne userB]; reject; stop
  )
endproc (* t1 *)
```

This process contains a `success` event (for the acceptance part) and a `reject` event (for the rejection part). The testing tool LOLA will look for `success` as the indication of a successful execution, and a must pass verdict is expected.

A total of 14 test processes (see Appendix B, lines 778 to 947) were created in a similar way. Each process uses one of the 14 test goals (Table 19 and Table 20) as the basis for an acceptance test case and a rejection test case. The rejection parts are all generated according to the “Off-by-one value” strategy.

6.5.4 Results from Test Execution

The *TestExpand* command of LOLA allows for the execution of LOTOS test cases [279]. This command has parameters for limiting the depth of the expansion resulting from the composition of a test process and the specification, for maintaining internal events or for simplifying them according to testing equivalence, for specifying the expected verdict (from the test purpose), for generating diagnostic traces, and for doing partial expansions according to state space and memory usage heuristics [280].

LOLA was used to check the 14 test cases composing the TTS test suite (*TS*). The resulting verdicts were all as expected, i.e. must pass for each test case. Formally, this means that the LOTOS prototype (*TTS*) constructed from the UCMs passed all the acceptance test cases and failed all the rejection test cases ($TTS \text{ passes } ACCEPT(TS) \wedge TTS \text{ failsall } REJECT(TS)$). Note that there were no rejection test processes as such in the test suite, because rejection test cases were integrated to acceptance test cases. According to Definition 6.2, the conclusion is that the TTS specification is valid with respect to the UCMs and the requirements ($TTS \text{ val } TS$).

Naturally, several small defects and discrepancies between the specification and the tests have been found during the construction and the validation of this prototype, but they were easily fixed because the TTS system is rather simple (more interesting problems are discussed in Chapter 8). LOLA allows for the tester to look at execution traces ending with an unexpected result, which eases diagnostics and debugging. Also, LOLA allows for batch testing. All test groups or test cases can be

executed in sequence, and their individual expected result can be checked. With the help of simple shell scripts, any unexpected result of a test can be discovered very quickly. This approach becomes very useful for regression testing. A change to the specification or to the test suite can be validated within a few seconds by re-checking the whole test suite¹.

6.6 Chapter Summary

This chapter presented a novel framework for the validation of high-level system specifications based on UCMs and on LOTOS testing theory and tools. This testing approach to validation, integrated to SPEC-VALUE and motivated in Section 6.1, is different from conventional conformance testing, although many of the terms and concepts used here are instantiated from CTMF.

In Section 6.2, test purposes are defined by a type (acceptance or rejection) and an abstract causal sequence known as the test goal. A new validity relation (val) was defined in terms of sound acceptance/rejection test cases. Differences between val and the well-known LOTOS conformance relation conf were discussed and they are illustrated thoroughly in Appendix C:

UCM scenarios describing requirements or designs are used in Section 6.3 for the selection of appropriate test goals. Six testing patterns regrouping 25 selection strategies, used to extract abstract causal sequences from UCM structures, are defined and illustrated in conformance with an appropriate template. These patterns, which suit the level of abstraction and formality addressed by UCMs, are connected in a UCM-oriented testing pattern language. The patterns are independent of any underlying structure of components, hence they apply to a wide variety of systems. They are also linked to the LOTOS testing theory, and were compared to Binder's test patterns (which do not really address complex issues such as concurrency and nesting of sub-paths) and to other test purpose notations.

The generation of LOTOS processes from abstract sequences is addressed in Section 6.4. Consideration needs to be given to the design decisions made during the construction of the LOTOS proto-

1. Compiling the TTS specification and checking the 14 test cases takes less than three seconds on a Pentium II (Celeron) 300MHz with 64MB RAM and Windows 98. This is fast enough to be used in an iterative and incremental design cycle.

type and the selection of appropriate values. The generation of rejection test cases, which is missing from most LOTOS test derivation techniques, is explored through the application of multiple strategies based on requirements, conditions, testing patterns, and the structure of the UCMs.

The testing patterns were applied to the TTS UCMs in order to select appropriate test goals and to generate acceptance and rejection test cases (Section 6.5). A test suite composed of 14 test processes was described and executed. Following the successful execution of the test suite according to the expected verdict of each test case, it was concluded that the LOTOS prototype was a valid representation of the UCMs and the requirements. The testing patterns were used rather precisely in this section, but experienced testers are likely to use them more informally in practice.

Contributions

The following items are original contributions of this chapter:

- Partial illustration of Contribution 1 (Section 1.4.1) regarding test case generation in SPEC-VALUE.
- Partial illustration of Contribution 2 (Section 1.4.2) regarding a UCM-LOTOS testing framework integrated to the SPEC-VALUE methodology.
- Illustration of step © in SPEC-VALUE, i.e. from UCMs to LOTOS test cases.
- The validation relation val, which is well suited for validation testing and which is more discriminating than the conf relation when the latter is evaluated through a finite test suite (Section 6.2.3).
- A UCM-oriented testing pattern language that explains how 25 coverage-driven strategies regrouped under six testing patterns can collaborate to select test goals from systems specified by UCMs (Section 6.3).
- Motivation and strategies for generating rejection test cases in LOTOS (Section 6.4.4).
- Application of the framework (test case selection, generation, and execution) to validate the Tiny Telephone System example (Section 6.5).

CHAPTER 7

Structural Coverage

Le doute sage et vraiment philosophique (s'il existait) consisterait donc à éteindre (ou plutôt à voiler) les lumières qui nous éblouissent, pour juger par un autre organe de l'esprit que celui de sa vue.

Joseph Joubert, 1808

The use of testing patterns for the selection of test cases ensures an appropriate coverage of system functionalities according to the UCM scenarios and the test plan. However, one can doubt that this coverage measure is sufficient when the LOTOS prototype is taken into consideration. As implied by Joubert in his citation (found below this chapter's title), a different point of view on the coverage could be wise and beneficial. This new viewpoint will be based on the *structure* of the prototype, not on its *functionalities*, the latter being already covered by the test suite.

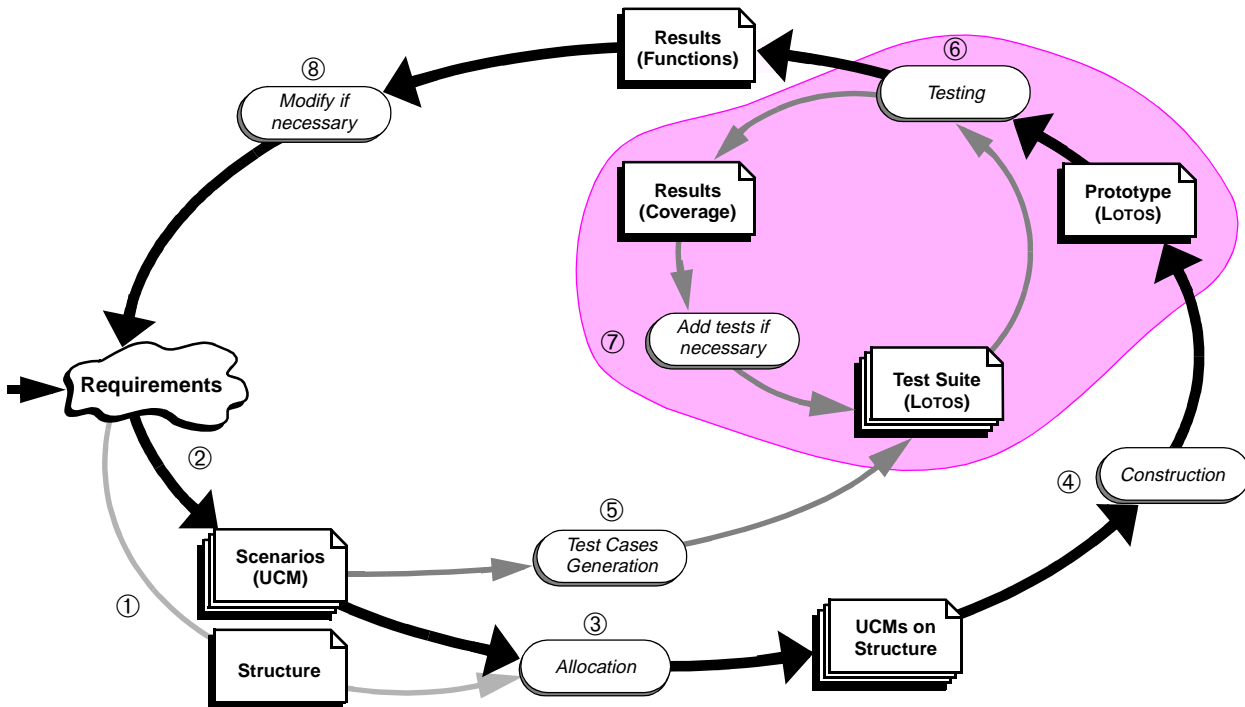
This chapter, which uses coverage concepts introduced in Section 3.4.4, presents a new technique for measuring the structural coverage of LOTOS specifications. This technique is first placed in the proper context with respect to the SPEC-VALUE methodology (Section 7.1). Because the technique uses instrumentation with probes, some issues are raised in Section 7.2. Section 7.3 illustrates how probes are used in sequential programs, whereas Section 7.4 adapts this idea to the context of LOTOS specifications. The technique is used to measure the structural coverage of the TTS system in Section 7.5. A brief discussion of other potential applications is included (Section 7.6), and then the chapter summary follows.

7.1 Structural Coverage in SPEC-VALUE

The generation of test cases from scenarios or from requirements is an *a priori* approach to validation. Such test cases can be constructed in parallel with the specification, or even before the specification is written. In SPEC-VALUE, the *functional coverage* is achieved, according to the test plan based on testing patterns and related strategies, when the test suite is executed successfully (i.e. *SUT val TS*). This functional coverage is a *semantic* measure of the correctness of a specification.

Observing the structure of the specification, composed of branches, events, and other such constructs can further enhance the quality of the test suite. The *structural coverage* of a test suite relates to the parts of the specification that have been visited by test cases. When this coverage is unsatisfactory, new test cases can be added *a posteriori* (step ⑦ in Figure 51). New types of faults or defects can be uncovered along the way. Under the assumption of a complete functional coverage, the structural coverage can be used as a basis for test suite completeness.

FIGURE 51. Structural Coverage with SPEC-VALUE



This chapter is concerned with the *syntactic* measure of the coverage of a formal specification by a validation test suite. This is different from implementation coverage by a conformance test suite derived (often automatically) from a formal specification. In our case, the validation test suite is obtained manually, hence coverage measures become necessary at the specification level.

The focus is on the structural coverage of LOTOS specifications using *probe insertion* [26]. We can instrument a specification and then assess that the structural coverage is achieved when all probes are visited by at least one test case. The main goals are to provide hints and assistance in the detection of unreachable portions of real-size specifications, and to measure the completeness of the test suite with respect to the syntactic structure of the specification under test, and not its underlying semantics or functionalities. Another goal is to cast these ideas in a pragmatic environment where the necessary steps for coverage measurement are automated as much as possible.

7.2 Issues in the Use of Probes

Probe insertion is a well-known white-box technique for monitoring software in order to identify portions of code that has not been yet exercised (Section 3.4.4). A program is instrumented with probes (counters) without any modification of its functionality. When executed, test cases trigger these probes, and counters are incremented accordingly. Probes that have not been “visited” indicate that part of the code is not reachable with the tests in consideration.

The following four points are notable software engineering issues related to approaches based on probe instrumentation of implementation code or of executable specifications:

1. *Preservation of the original behaviour.* New instructions shall not interfere with the intended functionalities of the original program or specification, otherwise tests that ran successfully on the original behaviour may no longer do so.
2. *Type of coverage.* Because probes are generally implemented as counters, it is easier to measure the coverage in terms of control flow rather than in terms of data flow or in terms of faults. Other techniques, summarized by Charles in [92], are more suitable for the two last categories of coverage criteria.

3. *Optimization.* In order to minimize the performance and behavioral impact of the instrumentation, the number of probes shall be kept to a minimum, and the probes need to be inserted at the most appropriate locations in the specification or in the program.
4. *Assessment.* What is assessable from the data collected during the coverage measurement represents another issue that needs to be addressed. Questions such as “Are there redundant test cases?” and “Why hasn’t this probe been visited by the test suite?” are especially relevant in the context of SPEC-VALUE.

These issues will be discussed for sequential program in the next section, and for LOTOS specifications in Section 7.4.

7.3 Probes in Sequential Programs

For well-delimited sequential programs, Probert suggests a technique for inserting the minimal number of *statement probes* necessary to cover all branches [291]. Table 21 illustrates this concept with a short Pascal program (a) and an array of counters named `Probe[]`. The counters indicate the number of times each probe has been reached.

TABLE 21. Example of Probe Insertion in Pascal

a) Original Pascal code	b) Three probes inserted	c) Optimal number of probes
<pre>statement1; if (condition) then begin statement2 end else begin statement3 end {end if};</pre>	<pre>statement1; inc(Probe[1]); if (condition) then begin inc(Probe[2]); statement2 end else begin inc(Probe[3]); statement3 end {end if};</pre>	<pre>statement1; inc(Probe[1]); if (condition) then begin inc(Probe[2]); statement2 end else begin {No probe here!} statement3 end {end if};</pre>

Intuitively, (b) shows three statement probes being inserted on the three branches of the program. In (c), the same result can be achieved with two probes only. Using control flow information, the number of times that `statement3` is executed is computed from `Probe[1]-Probe[2]`. After the execution of the test suite, if `Probe[2]` is equal to `Probe[1]`, then the conclusion is that the ‘else’ branch that includes `statement3` has not been covered.

It has been proved in [291] that the optimal number of statement probes necessary to cover all branches in a well-delimited sequential program is $|E| - |V| + 2$, where $|E|$ and $|V|$ are respectively the number of edges and the number of vertices of the underlying extended delimited Böhm-Jacopini flowgraph of the program.

The four issues raised in Section 7.2 are covered as follow:

1. *Preservation of the original behaviour*: if the probe counters are variables that do not already exist in the program, then the original functionalities are preserved.
2. *Type of coverage*: the coverage is related to the control flow of the program.
3. *Optimization*: there exists a way to minimize the number of statement probes so it can be smaller than the number of statements.
4. *Assessment*: this technique covers all branches in a well-delimited sequential program.

7.4 Probe Insertion in LOTOS

Similarly to probe insertion in well-delimited sequential programs, LOTOS constructs could be used to instrument a specification at precise locations while preserving its general structure and its externally observational behaviour. Although the execution of test cases can be slowed down by this instrumentation, it is not desirable to affect the functionality of the specification or the results of the validation process.

Among the LOTOS constructs, the most likely candidate for incarnating a probe is an internal event with a unique identifier. Such event can be composed of a hidden gate name that is not part of

any original process in the specification (e.g. *Probe*), followed by a unique value of some new enumerated abstract data type (e.g. P_0, P_1, P_2, P_3 , etc.).

A simple probe insertion strategy for LOTOS is presented in Section 7.4.1, followed by a more optimized version in Section 7.4.2. The interpretation of coverage results and tool support are discussed respectively in Sections 7.4.3 and 7.4.4.

7.4.1 A Simple Insertion Strategy

According to Table 3 on page 31, a *basic behaviour expression* (BBE) is either the inaction **stop**, the successful termination **exit**, or a process instantiation ($P[\dots]$). In LOTOS, a *behaviour expression* (BE) can be one of the following¹:

- A BBE.
- A BE prefixed by a unary operator, such as the action prefix ($;$), a **hide**, a **let**, or a guard ($[\text{predicate}] \rightarrow$).
- Two BEs composed through a binary operator, such as a choice ($[\]$), an enable (\gg), a disable (\gt), or one of the parallel composition operators ($[\dots] \mid$, \parallel , or $\parallel \parallel$).
- A BE within parentheses.

In this chapter, a *sequence* is defined as a BBE preceded by one or more events, separated by the action prefix operator ($e_1; e_2; \dots e_n; BBE$). A BBE that is not preceded by any event is called a *single BBE*.

Probes enable the measure of the coverage of every event in a behaviour expression, and therefore in a whole specification. The simplest and most straightforward strategy consists in adding a probe after each event at the syntactic level. For each event e and each behaviour expression B , the expression $e; B$ is transformed into $e; Probe!P_id; B$ where *Probe* is a hidden gate and P_id a unique identifier. A probe that is visited guarantees, by the action prefix inference rule, that the prefixed event

1. We consider a very common subset of LOTOS where there are no generalized **Par** or **Choice** operators.

has been performed. In this case, if all the probes are visited by at least one test case in the validation test suite, then the test suite has achieved a total *event coverage*, i.e. the coverage of all the events in the specification (modulo the value parameters associated to these events).

Table 22 illustrates this strategy on a very simple specification *S1* (a). Since there are three occurrences of events in the behaviour, three probes, implemented as hidden gates with unique value identifiers, are added to *S1* to form *S2* (b). The validation test suite is somehow derived from scenarios or requirements according to some test plan or functional coverage criteria (e.g. UCMs and testing patterns). In this example, it is composed of two test cases (*Test1* and *Test2*), which remain unchanged during the transformation. The third specification (c) will be discussed in Section 7.4.2.

TABLE 22. Simple Probe Insertion in LOTOS

a) Original LOTOS specification (<i>S1</i>)	b) 3 probes inserted in the specification (<i>S2</i>)	c) 2 probes inserted, using the improved strategy (<i>S3</i>)
<pre> specification S1[a,b,c]: exit ... (* ADTs *) behaviour a; exit [] b; c; stop where process Test1[a]:exit:= a; exit endproc (* Test1 *) process Test2[...]:noexit:= b; c; Success; stop endproc (* Test2 *) endspec (* S1 *) </pre>	<pre> specification S2[a,b,c]: exit ... (* ADTs *) behaviour hide Probe in (a; Probe!P_1; exit [] b; Probe!P_2; c; Probe!P_3; stop) where ... (* Test1 and Test2 *) endspec (* S2 *) </pre>	<pre> specification S3[a,b,c]: exit ... (* ADTs *) behaviour hide Probe in (a; Probe!P_1; exit [] b; c; Probe!P_2; stop) where ... (* Test1 and Test2 *) endspec (* S3 *) </pre>

Probe insertion is a syntactic transformation that also has an impact on the underlying semantic model. Table 23 presents the LTSs resulting from the expansion of *S1* and *S2*. A LOTOS **exit** is represented by δ at the LTS level. When a test case ending by **exit** is checked (e.g. *Test1*), LOLA automatically transforms such δ into **i** followed by *Success*. Although the LTSs (a) and (b) are not equal as trees, they are observationally equivalent. Therefore, the tests that are accepted and refused by *S1* will be the same as those of *S2*.

TABLE 23. Underlying LTSs

a) Original LOTOS specification (S1)	b) 3 probes inserted in the specification (S2)	c) Composition of S2 with two test cases: Test1 & Test2

Table 23(c) presents two traces, resulting from the composition of each test process found in Table 22(a) with $S2$, that cover the events and probes of $S2$. Test1 covers P_1 in the left branch of (c) whereas Test2 covers P_2 and P_3 in the right branch. Neither of these tests covers all probes, but together they cover all three probes, and therefore the event coverage is achieved, as expected from such a validation test suite. The fact that the entire LTS is covered here is purely coincidental, as it is usually not the case for complex specifications.

Going back to the four issues enumerated in Section 7.2, the following observations are made:

1. *Preservation of the original behaviour*: probes are unique internal events inserted *after* each event (internal or observable) of a sequence. They do not affect the observable behaviour of the specification. This insertion can be summarized by Proposition 4, which coincides with one of the LOTOS congruence rules found in the standard [191] (congruence rules preserve observational and testing equivalences in any context):

$$e; B \approx_c \mathbf{hide\ Probe\ in\ } (e; \mathbf{Probe!P_id}; B) = e; \mathbf{i}; B \quad \text{(PROP. 4)}$$

2. *Type of coverage*: the coverage is concerned with the structure of the specification, not with its data flow or with fault models. The resulting *event coverage* makes abstraction of the semantic values in the events (e.g. the expression dial?n:nat abstracts from any natural number n).

3. *Optimization*: the total number of probes equals the number of occurrences of events in the specification. Reducing the number of probes is the focus of the next section.
4. *Assessment*: this strategy covers all events syntactically present in a specification. Single basic behaviour expressions are not covered.

7.4.2 Improving the Probe Insertion Strategy

The simple insertion strategy leads to interesting results, yet two problems remain. First, the number of probes required can be very high. The composition of a test case and a specification where multiple probes were inserted (and transformed into internal events) can easily result in a state explosion problem. Second, this approach does not cover single BBEs as such, because they are not prefixed by any event. Single BBEs may represent a sensible portion of the structure of a specification that needs to be covered as well. This section presents four optimizations that help solving these two problems, followed by an assessment of this improved strategy.

First Optimization: Sequences

In a sequence of events, the number of probes can be reduced to one probe, which is inserted just before the ending BBE. If such a probe is visited, then LOTOS' action prefix inference rule leads to the conclusion that all the events preceding the probe in the sequence were performed. The longer the sequence, the better this optimization becomes. Table 22(c) shows specification *S3* where two probes are used instead of three as in *S2*. This *sequence coverage* implies the coverage of events with fewer probes or the same number in the worst case.

Second Optimization: Parentheses

The second optimization concerns the use of parenthesis in $e; (B)$, where B is not a single BBE. In this case, no probe is required before (B) . The behaviour expression B will most certainly contain probes itself, and a visit to any of these probes ensures that event e is covered (again, by the prefix inference rule).

Third Optimization: Single BBEs

For the structural coverage of single BBEs (without any action prefix), there are some subtle issues that first need to be explored. Suppose that $*$ is one of the LOTOS binary operators enumerated at the beginning of Section 7.4.1 ($[]$, \gg , $[>$, $[[\dots]]$, $[|]$, $[| |]$). If a single BBE is prefixed with a probe in the generic patterns $BBE * BE$ and $BE * BBE$, then care is required in order not to introduce any new non-determinism. Additional non-determinism could lead some test cases to fail. A probe can safely be inserted before the single BBE unless one of the following situations occurs:

- BBE is **stop**: This is the inaction. No probe is required on that side of the binary operator ($*$) simply because there is nothing to cover. This syntactical pattern is useless and should be avoided in the specification.
- BBE is a process instantiation $P[\dots]$: A probe before the BBE can be safely used except when $*$ is the choice operator ($[]$), or when $*$ is the disable operator ($[>$) with the BBE on its right side. In these cases, a probe would introduce undesirable non-determinism that might cause some test cases to fail partially: LOLA would return a *may pass* verdict instead of a *must pass*. A solution would be to guard the process instantiation. One way of doing so in many cases would be to partially expand process P with the expansion theorem so an action prefix would appear. Another solution is presented below, in the last optimization.
- BBE is **exit**: The constraints are the same as for the process instantiation. The solution is also to prefix this **exit** with some event.

Fourth Optimization: Process Instantiations as BBEs

When a process P is not defined as a single BBE, then the necessary number of probes can be further reduced when P is instantiated in only one place in the specification (except for recursion in P itself). In this case, a probe before P is not necessary because probes inserted within P will ensure that the instantiation of P is covered. This is especially useful when facing a process instantiation as a single BBE. For example, suppose a process Q that instantiates P in one place only, where P is not a BBE and P is not instantiated in any process other than Q and P itself:

$$Q[\dots] := e1; e2; e3; \mathbf{stop} \quad [] \quad P[\dots]$$

A probe inserted before P would make the choice non-deterministic, which could lead to undesirable verdicts during the testing. However, if P is not a single BBE and if it is not instantiated anywhere else, then no probe is required before P in this expression. Any probe covered in P would ensure that the right part of the choice operator in Q has been covered. This situation often happens in processes that act as containers for aggregating and handling other process instances. For instance, the process representing a UCM stub would instantiate plug-in processes. If a plug-in is used only in this stub, then the stub process does not need any probe in front of the instantiation of this plug-in process.

Comments on Probe Insertion Issues

Regarding the four issues enumerated in Section 7.2, the improved strategy achieves a coverage of the specification that is larger than the simple strategy, and it takes fewer probes to do so.

1. *Preservation of the original behaviour*: probes are unique internal events inserted before each BBE. When such BBE is prefixed by an event, then the probe does not affect the observable behaviour of the specification (Proposition 4). When the BBE is not prefixed, a case not addressed by the simple strategy, then special care must be taken in order not to introduce new non-determinism.
2. *Type of coverage*: the *sequence and single BBE coverage*, which implies the event coverage of the simple strategy, is also concerned with the structure of the specification.
3. *Optimization*: the total number of probes is less or equal to the total number of sequences and BBEs in the specification.
4. *Assessment*: this strategy covers all events syntactically present in a specification, as well as single BBEs other than **stop** (which should not be found in a LOTOS specification anyway).

7.4.3 Interpreting Structural Coverage Results

Several problem sources can be associated to probes that are not visited by a test suite. They usually fall into one of the following categories:

- *Incorrect specification.* In particular, the specification could include unreachable code caused by processes that cannot synchronize properly or by guards that can never be satisfied. Note that there is a well-known theoretical result that shows there can be no algorithm to determine whether or not a particular statement of a program is reachable [110]. Similarly, one cannot determine automatically that a probe in a specification is unreachable.
- *Incorrect test case.* This is usually detected before probes are inserted, during the verification of the functional coverage of the specification.
- *Incomplete test suite.* Caused by an untested part (an event or a single BBE) of the specification (e.g. a feature of the specification that is not part of the original requirements).
- *Discrepancy.* Due to the manual nature of the construction of the specification from the UCMs, there could be some discrepancy between a test and the specification caused by ADTs, guards, the choice ([]) operator, or other such constructs.

Code inspection and step-by-step execution of the specification can help diagnosing the source of the problem highlighted by a missed probe. Although unreachable code cannot be detected automatically, practical experience and various empirical experiments have shown that human beings are, in fact, quite good at determining whether or not code is reachable [371][372].

LOLA's *FreeExpand* could also be used to expand the specification in order to check that all probes are in the underlying LTS. This would ensure that no part of the code is unreachable. However, for most real-size specifications, this approach is not likely to work because of the state explosion problem and incomplete evaluation of guards and predicates. Using on-the-fly model checking, the verification of an appropriate property, which would state that a particular probe could be eventually reached, seems a more practical solution. Goal-oriented execution, a technique based on LOTOS' static semantics proposed by Haj-Hussein *et al.* in [165], represents a promising approach to the determination of the reachability of a uniquely identified probe. However, this technique would first have to be extended in order to allow specific internal events (the probes) to be used as goals.

7.4.4 Tool Support

A filter tool called LOT2PROBE, written in LEX, was built for the automated translation of special comments manually inserted in the original specification (e.g. (`*_PROBE_*`)) into internal probes with unique identifiers (e.g. `Probe!P_0;`). A new abstract data type that enumerates all the unique probe identifiers is also added to the specification. Care is taken not to add any new line to the original specification, in order to preserve two-way traceability between the transformed specification and the original one.

Since most LOTOS prototypes specified in a resource-oriented style (including those constructed from UCMs) do not contain any full synchronization operator, the `Probe` gate was hidden at the topmost level of the specification (the `behaviour` section), and was added to the list of gate parameters of all process definitions and instantiations. Where a full synchronization operator (`| |`) is used, the tool suggests the use of the generalized synchronization operator (`[[. . .] |`) to avoid unexpected deadlocks.

Batch testing under LOLA can then be used for the execution of the validation test suite against the transformed specification. Several scripts, written in PERL and LEX, compute probe counts for each test and output textual and HTML summaries of the probes visited, with a highlight on probes that were not covered by any test.

Though full automation of probe insertion is possible, the solution developed so far is still semi-automatic because of some special cases (i.e. with problematic BBEs) that are not trivial to handle. However, the manual insertion of these probe comments has the benefit of being more flexible, and it can be done at specification time or after the initial validation.

7.5 TTS Structural Coverage Results

The improved strategy was used to insert probes in the TTS specification (see the (`*_PROBE_*`) comments in Appendix B). This specification contains 25 events, 18 sequences, and 22 single BBEs. The simple insertion strategy of Section 7.4.1 would have required 25 probes, but no single BBE would have been covered. To cover all events and single BBEs straightforwardly, 44 probes (25+22)

are required. However, by using the optimizations discussed in Section 7.4.2, this number can be reduced to 26, which represents a very good improvement. This reduction is explained in part by the presence of sequences and by the high number of stub processes and plug-in processes that are instantiated only once.

The TTS specification was transformed by LOT2PROBE into a new but equivalent specification where the (**_PROBE_**) comments are translated to `Probe` gates with a unique identifier, from `P_0` to `P_25`. The test suite was executed against this new specification, and the tests resulted in the same verdicts as with the original specification, so no new non-determinism had been added, as expected. It took 140 seconds to create the new specification, to compile it, to execute the tests and to compile the test results automatically through scripts and batch files¹. Most of this time (>95%) was spent on the testing itself, the latter is a laborious task for LOLA because internal actions can no longer be abstracted using simplifications based on testing equivalence. This is caused by the approach which requires the probe (internal) events to be output in the execution traces in order to measure the structural coverage. If these probes are not explicitly part of the traces, then no measure can be done. Although this coverage measurement takes a few minutes to be computed instead of a few seconds for a simple validation of the specification against the test suite, the impact is minimal for SPEC-VALUE because structural coverage measurements are done sporadically, once the functional coverage is achieved.

7.5.1 Summary of Coverage Results

The results of this experiment are summarized in Table 24, with 26 probe identifiers shown on the left (together with their respective line number) and 14 test case identifiers at the top.

1. On a Celeron 300MHz, 64 MB RAM, running LOLA on Windows 98.

TABLE 24. TTS Structural Coverage Results

Test # → Probe #, line ↓	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Test Suite
P_0, line 462	9	16	8	16	9	16	8		16	2	73	87			260
P_1, line 467												21			21
P_2, line 484	1				1			0*			2	28		1	33
P_3, line 488		1		1		1			1		1				5
P_4, line 492				1	1				1						3
P_5, line 523	1	1	1	1	1	1	1		1	1	17	9			35
P_6, line 532								0*						1	1
P_7, line 541	2				2						4	56			64
P_8, line 559	4	4	2	4	4	4	2		4	1	24	30			83
P_9, line 569		2		2		2			2		2				10
P_10, line 574			1				1			1	9	8			20
P_11, line 577		2	1	2		2	1		2	1	16	20			47
P_12, line 603						1	1		1						3
P_13, line 606								1					1	2	4
P_14, line 616	1	1	1	1	1					1	9	7			22
P_15, line 638			1				1			1	8	8			19
P_16, line 641	1	1		1	1	1			1		1				7
P_17, line 661	2	2	1	1	1	1				1	10	7			26
P_18, line 686	1	1		1	1	1			1		1				7
P_19, line 691	1	1		1	1	1			1		1				7
P_20, line 698			1				1			1	8	8			19
P_21, line 721				1	1				1						3
P_22, line 731	1	1				1					1				4
P_23, line 751						1	1		1						3
P_24, line 756								1					1	2	4
P_25, line 774				1	1				1						3
Traces no probe	1	3	1	6	2	3	1	1	6	1	3	127	1	1	157
Traces, probes	16	24	12	24	16	24	12	1	24	2	95	223	1	1	475

The test suite covers all 26 probes, therefore the structural coverage of the TTS specification is complete. All the specification code is reachable and no additional test case is required.

7.5.2 Comments on LOLA and Missing Probes

For the TTS specifications (with and without probes), the number of traces generated by LOLA for the test cases can be found at the bottom of each column. Because probes add new internal actions to a specification, the number of possible traces gets higher in the presence of interleaving. This explains why many more end-to-end sequences are executed for a same test case in the presence of probes. When a specification gets complex and when many probes are inserted, this interleaving of new internal actions can result in an explosion of the number of states. To avoid this problem, the testing functionalities of LOLA come with optional heuristics for the partial expansion instead of the default exhaustive expansion. In our experience (six specifications with probes in the last three years), the probe coverage given by a heuristic expansion is the same as with an exhaustive expansion. However, heuristic expansions lead to an important reduction (nearly 99%) of the size of the resulting LTS and of the time required for the expansion. This option was used in the structural coverage measurement of the TTS specification.

Table 24 also shows that some robustness test cases (#10 to #14) covered probes that were not visited by the UCM-based validation test suite (#1 to #9). This is the case for P_1 , visited only by test #12, and for P_6 visited only by test #14 (grey cells in Table 24):

- Probe P_1 (line 467 in Appendix B): this probe belongs to the process definition of the medium used by agents to communicate with each other. This medium is not part of the UCMs, hence no specific validation test case was generated using testing patterns. Test #12 is covering this probe because this medium represents a FIFO buffer of size two, and it takes two simultaneous calls for reaching this probe. Only test #12 initiates two simultaneous call requests.
- Probe P_6 (line 532 in Appendix B): the *TestExpand* command of LOLA indicates that this probe is not covered by tests #1 to #9. However, using step-by-step execution or free expansion of the test #8, the resulting traces indicate that probes P_6 and P_2 (see the black cells in Table 24) are visited on top of those discovered by *TestExpand*. For instance:

```

init ! usera ! ...;
req ! usera ! userb;
i; (* chk *)
i; (* pds *)
i; (* probe ! p_24 *)
i; (* exit (...) *)
i; (* probe ! p_13 *)
i; (* exit (...) *)
sig ! usera ! calldenied;
i; (* probe ! p_2 *)
i; (* probe ! p_6 *)
success;

```

This discrepancy is caused by a bug in the *TestExpand* command which stops when the success event is reached, even if further internal events can be executed and added to the trace. In fact, test #14 could have given the same inexact result. This problem can be solved using step-by-step execution or free expansion (*FreeExpand*) on the test cases which are suspected to cover probes that are not visited using *TestExpand*.

Using LOLA, if a probe is not covered, then the bug in *TestExpand* might just be the real cause. Else, the four possible explanations described in Section 7.4.3 must be considered. In any case, the use of a heuristic expansion is more efficient and does not affect the coverage result.

7.6 Discussion

Section 7.6.1 comments on the compositional coverage of the structure in the presence of many probes whereas Section 7.6.2 addresses the impact of the specification style on the coverage measurement. Section 7.6.3 gives a brief overview of how coverage measurements could be used for test case management.

7.6.1 Compositional Coverage of the Structure

A full structural coverage of a LOTOS specification does not imply that inserted probes need to be covered all at once. Since probes do not affect the observable behaviour of the specification, a *compositional* coverage of the structure becomes possible. Probes can be covered independently, and one could even do this one probe at a time. This reduces the size of the resulting LTSs to a minimum, and

thus helps avoiding the state explosion problem. The LOT2PROBE filter allows different variations of the probe comment in the specification (e.g. (**_PROBE_A_**), (**_PROBE_B_**), etc.). Different groups of probes can therefore be transformed into hidden gates, one group at a time. This approach is applied to another case study, where there are too many probes to handle all at the same time (Section 8.2).

7.6.2 Specification Styles

Two equivalent specifications written using different styles might lead to different coverage measures for the same test suite. The way a LOTOS specification is structured usually reflects more than its underlying LTS model. For instance, in a resource-oriented style, the structure can be interpreted as the architecture of the system to be implemented [364]. In a constraint-oriented style, processes impose local or end-to-end constraints on the system behaviour.

This problem is also true of programs in general, as observed in Weyuker's *anti-extensionality property*, which states that the semantic equality of two programs is not sufficient to imply that they should necessarily be tested the same way [371]. In any situation, the important thing is to achieve the target functional and structural criteria.

7.6.3 Test Case Management Based on Structural Coverage

Structural coverage results could be used for test case management in at least two ways. First, these results could help detecting redundant test cases by providing useful hints in terms of probes visited. A test case whose visited probes are all already visited by another test case (or by a set of test cases) is an indication of redundancy from a structural perspective. If this test case is not motivated by a relevant rationale (functionality, robustness, etc.), then it could be removed from the test suite. Test suites could be reduced in size while retaining the same structural coverage.

Second, the optimization of the order of passage of the test cases could also be determined structurally according to similar criteria. For instance, the tests that visit the highest number of probes, i.e. the largest probe coverage, could be executed first. If a larger structural coverage at the specification level implies a larger coverage at the implementation level, then such criteria could help

sort test cases with the hope of finding faults and errors earlier when validating an implementation with the test suite.

7.7 Chapter Summary

This chapter presented a new theory and a semi-automated technique for the measure of structural coverage of LOTOS specification against test suites. This coverage can improve the quality and consistency of both the specification and the validation test cases, hence resulting in a higher degree of confidence in the system's description. This technique provides an assessment of how well a given test suite has covered a LOTOS specification rather than providing a guideline on how the specification is to be covered for testing implementations.

Section 7.1 motivated the need for a syntactic measure that complements the semantic (functional) coverage achieved when a validation test suite is successfully executed. This section also positioned the structural coverage approach in the SPEC-VALUE methodology. Section 7.2 presented four general issues related to the instrumentation of programs and specifications with probes. The theory behind an existing probe insertion technique for sequential programs was reviewed in Section 7.3. In Section 7.4, a similar theory is tailored for a process algebra with concurrency, namely LOTOS. This section presented a probe insertion strategy that covers events and basic behaviour expressions while maintaining observational equivalence and minimizing the number of probes.

The coverage results provided by this technique can help detect incomplete test suites, discrepancy between a specification and its test suite, and unreachable parts of the specification, with respect to the requirements and UCMs in consideration. A tool set composed of LOT2PROBE and other scripts supports this technique. Although LOLA is conveniently used in this thesis, the technique and tool set are independent from the LOTOS execution environment. Section 7.5 applied the technique to the TTS example and discussed the coverage results. Section 7.6 discussed the compositional coverage of specifications, the impact of specification styles over coverage results, and the potential use of structural coverage for test case management and optimizations of test suites.

Contributions

The following items are original contributions of this chapter:

- Partial illustration of Contribution 2 (Section 1.4.2) regarding a new theory and a new technique for the structural coverage of LOTOS specifications, integrated to SPEC-VALUE.
- Illustration of step ⑦ in SPEC-VALUE, i.e. coverage measures.
- Partial tool support: from manually inserted (`*_PROBE_*`) comments to coverage reports. Probe insertion is not automated yet.
- Application of the technique and tools to measure the structural coverage of the Tiny Telephone System example.

CHAPTER 8

Experiments with SPEC-VALUE

*Que dites-vous?... C'est inutile?... Je le sais!
Mais on ne se bat pas dans l'espoir du succès!
Non! non, c'est bien plus beau lorsque c'est
inutile!*

*Cyrano de Bergerac
Edmond Rostand, 1897*

This chapter introduces six experiments based on real-life and hypothetical telecommunications systems used to validate the SPEC-VALUE methodology and techniques. Five of them contain technical descriptions as well as lessons learned during the specification and the validation of the aforementioned communicating systems (Sections 8.1 to 8.5). Most of these case studies include sections that provide overviews of the system and the UCM descriptions, of the resulting LOTOS specification, of the selection and execution of test cases, of the structural coverage achieved, and general discussions. A sixth experiment uses mutation analysis to study the effectiveness of validation test cases generated from testing patterns in finding faults (Section 8.6). A global summary follows in Section 8.7.

8.1 Group Communication Server (GCS)

The *Group Communication Server* (GCS) was an academic exercise used to demonstrate the applicability of multiple specification, validation, and performance evaluation techniques developed at Carleton University and the University of Ottawa, two of which were UCMs and LOTOS. The SPEC-VALUE methodology was mainly developed at the same time as this experiment, whose details can be found in a technical report [17] and a publication [15]. Also, UCM-based performance analysis of the GCS is studied by Scratchley and Woodside in [324][325].

8.1.1 System Overview and UCM Descriptions

A GCS allows the multicasting of messages to members of a group. Groups are created and destroyed dynamically as the need arises. A GCS offers the core services required for the implementation of the server side of systems such as mailing lists, Internet Relay Chat (IRC), videoconferences, and publish-and-subscribe systems. Users are permitted to join and quit one or many groups. Messages consist in a variety of types (voice, video, data, etc.) and are multicast to the members of the group via different communication channels, selected to suit the requirements of the group. A group may have an administrator whose tasks include registration management and group deletion. A group may also have a moderator whose task is to approve or reject messages sent to the group. A group is defined according to different parameters, some of which can be changed dynamically: administered or not, moderated or not, public subscription or private subscription (i.e. performed by the administrator), and open multicasting or closed multicasting (i.e. posting by group member only).

Twelve individual scenarios, described as UCMs, were extracted from the twelve GCS functionalities drafted in the informal requirements: group creation, list groups, get attributes, group deletion, member registration, list group members, member deregistration, multicast, change administrator, change open attribute, change private attribute, and change moderator. Tables describing the responsibilities and conditions supplemented the UCMs. Different potential structures were defined and evaluated, and the selected one includes a dozen components, including senders, receivers, database objects, and group and multicast processes. Different communication channels are also identified. Group teams and multicast processes are created and destroyed when required through the use of dynamic responsibilities.

This collection of UCMs does not use any stubs or plug-ins; the integration of the scenarios was done at the LOTOS level by using preconditions attached to UCM start points and postconditions attached to end points.

8.1.2 Construction of the LOTOS Prototype

The prototype was constructed by integrating the GCS functionalities (UCM scenarios) directly into the LOTOS specification. The twelve functionalities were integrated one by one, in the order given in the previous section.

The structure of the GCS specification is intended to be flexible and dynamic. The process definition of a group contains sub-processes (one per functionality), one of which is instantiated upon the arrival of the corresponding service request from the sender. This type of structure was very useful for the incremental integration and validation of the required functionalities. Also, groups can be created and destroyed dynamically. In fact, no predefined set of users or groups needed to be hard-coded in the specification. This flexibility enables the use of different system configurations at testing time, at the cost of having a specification that cannot be represented as a finite LTS or state machine (as required by some LOTOS tools like CÆSAR [146]). Another level of flexibility is found in the modularity of the multicast process. The requirements and the UCMs were not specific about the way a message was to be multicast by the group communication server to the receivers. Hence, the specification includes a generic solution where the multicast protocol can be changed without affecting the rest of the specification. In this experiment, three such protocols were included as examples:

- **Sequential Multicast:** instead of sending the messages to receivers concurrently, the sending is done sequentially in a LIFO order.
- **Best Effort Sequential Multicast:** as for Sequential Multicast, the sending is done sequentially in a LIFO order. However, problems may occur during the sending of messages, or during their reception if the sending is synchronous. This protocol includes a timeout mechanism to ensure that such failures do not block the protocol. The number of successful messages sent is also counted.
- **Broadcast:** instead of using point-to-point communication, some underlying broadcast mechanism (such as IP broadcast) is used to send a message to all group members at once. Receivers are responsible for the filtering of relevant messages based on their belonging to specific groups.

The resulting GCS specification contains 29 abstract data type definitions (800 lines of commented LOTOS code) and 19 process definitions (750 lines). Passive components, i.e. databases, were specified as process parameters. Since the focus was on the server side, the clients (i.e. senders and receivers) were not specified.

8.1.3 Test Selection and Execution

The application of testing patterns to the 12 UCMs lead to the definition of 12 acceptance test groups and 12 rejection test groups (one of each type per UCM, to improve traceability), for a total of 56 acceptance test cases and 51 rejection test cases. Most rejection test cases were created by mutating the last action in an abstract sequence, or by mutating the values accepted on this last action (strategies *off-by-one gate* and *off-by-one value* in Section 6.4.4). Two additional acceptance test cases were added to illustrate other testing possibilities, such as the use of abstract timers in test cases or the use of generic test process definitions that make use of verification steps (in order to reduce the need for rejection tests). The number of lines of commented LOTOS code used to describe these 109 tests is close to 1600. LOLA executed all the tests properly in less than 5 seconds, including the time for compiling the specification¹.

8.1.4 Structural Coverage

The GCS specification contains a total of 57 LOTOS events, as well as 35 simple basic behaviour expressions (BBEs). The events are structured in 40 different sequences. Using the improved strategy and its optimizations, a total of 54 probes were inserted in the prototype. This strategy and the optimizations hence resulted in a 41% reduction of the number of probes necessary to cover all events and BBEs (i.e. $(57 + 35 - 54) / (57 + 35)$). Owing to the presence of more internal actions, the time taken to measure the coverage was higher, but still manageable in an iterative process: 3 minutes 55 seconds. Five probes were missed during the first measure. Two of these were caused by the bug in LOLA's *TestExpand* command (see Section 7.5.2) and they could be reached through simulation, so they are not really problematic. The three other probes were missed for the following reasons:

1. In this chapter, all times are measured on a Celeron 300MHz, 64MB RAM, running Windows 98.

- Two probes were related to a feature that was not part of the requirements or the UCMs, but that was specified in LOTOS anyway (a group is deleted when there is no member left). As such, relevant test cases could not have been derived from the UCMs. This problem was solved by documenting the UCM appropriately and by adding two test cases, obtained from a step-by-step simulation of the specification, which cover these probes.
- One probe was not covered because a UCM path had been specified as a choice between two guarded behaviour expressions with different values. It seemed easier to code in such a way this particular UCM path in LOTOS. However, the test case derived from the UCM covered one alternative only. Another test case, which covers the probe found on the other alternative, was added to the corresponding acceptance test group.

The three additional test cases are already included in the numbers given in the previous section.

8.1.5 Discussion

This specification went through several small iterations as the SPEC-VALUE methodology gained in maturity. A large proportion of the construction guidelines, the testing patterns, and the structural coverage techniques were developed during this experiment.

One lesson learned during this experiment is that modifications to the scenarios at the LOTOS level may have an impact on the Use Case Maps and on the requirements, and hence could lead to modifications at these levels of abstraction. Deviations may be caught by the test suite and/or the structural coverage measurement (e.g. the missing probes in this GCS example), but there is no guarantee that they will. It is the responsibility of the LOTOS specifier to evaluate the impact of deviations from UCM paths and construction guidelines and to report to the UCM designers and/or the requirements engineers in order to determine what needs to be updated.

8.2 GPRS Group Call (PTM-G)

The *General Packet Radio Service* (GPRS) is a mobile telephony service standardized by the European Telecommunications Standards Institute (ETSI) [128]. It allows its subscribers to send and receive data in an end-to-end packet transfer mode. Built on top of the concepts and technologies of *Global System for Mobile Communications* (GSM) [127][264], a connection-oriented service for mobile telephony, GPRS provides connectionless packet transfer within the *Public Land Mobile Network* (PLMN) in interworking with external networks (such as X.25 and TCP/IP). The focus of the experiment was on one specific GPRS service: the *Point-To-Multipoint-Group Call* (PTM-G). This service allows transmissions to specific groups of users in specific geographical areas. At any point in time, the network has the knowledge of the number of users and their location.

This experiment was done in collaboration with an industrial partner and a student (“stagiaire”) from France during the first standardization stage of GPRS. This evolving draft standard represented a great opportunity for assessing the usefulness and adequacy of SPEC-VALUE in a standardization context. Details on this experiment can be found in two publications [16][24]. Other papers presenting LOTOS specifications of GSM and GPRS include contributions from Tuok and Logrippo [348][349] and from Ghribi *et al.* [36][151].

8.2.1 System Overview and UCM Descriptions

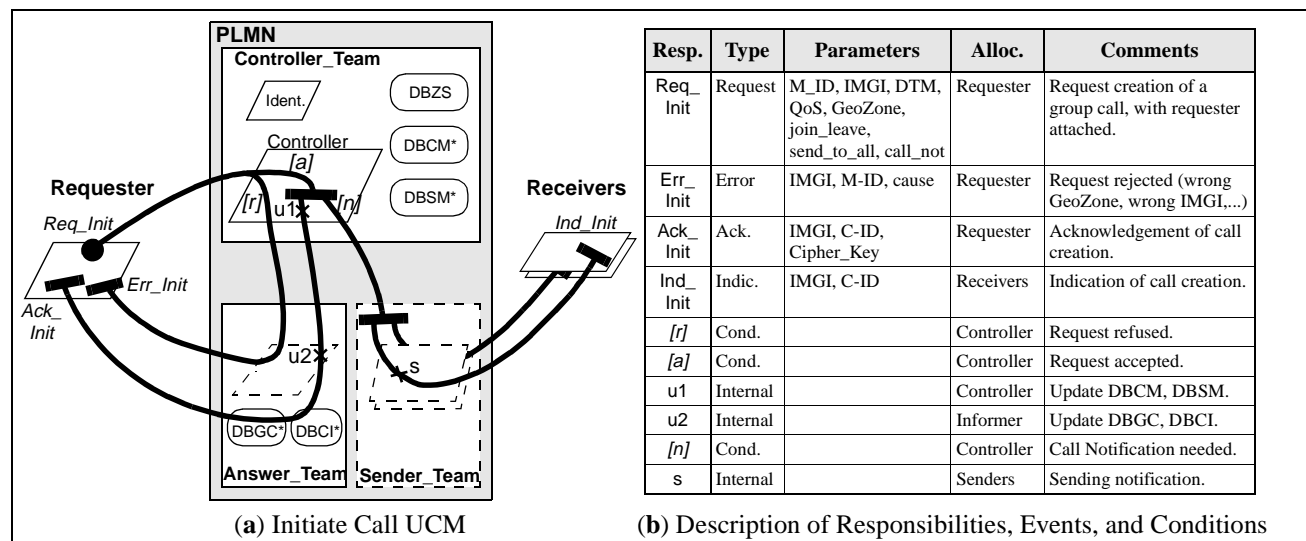
The PTM-G service is composed of six operations: *Initiate Call*, to create a group call; *Terminate Call*, to delete a group call; *Call Status*, to get the attributes of a group call; *Join Call*, to join an existing group call; *Leave Call*, to quit a joined group call; and *Data Transfer*, to send messages and data. In order to model the Terminate Call and Leave Call operations invoked by the network, three artificial operations were also defined. Two of them are located in the underlying services: *Attach GPRS* and *Detach GPRS*. The third one, *Change Zone*, emulates the routing operation triggered by the physical layer.

As no concrete structure or architecture was imposed in the preliminary version of the standard, we defined an abstract structure (using logical entities) independent of the physical components

of GPRS. Figure 52(a) presents the 15 components included in this structure. The PTM-G service is transparent to the user, so it is located inside the PLMN component. The requesters and receivers are roles that can be assumed by the users, identified by their *mobile stations* (MS).

Nine UCMs were created, one for each operation. The first six UCMs were obtained easily since PTM-G operations are described rather operationally, although very informally, in the draft standard [128]. The three others were created according to our general understanding of GSM and GPRS. All nine UCMs start with a single start point, leading to one or possibly many end points. Figure 52(a) shows the UCM for the Initiate Call operation, which is triggered by a requester (a mobile station). Without getting into the details of the example, this operation updates several databases (represented as objects), returns an acknowledgement to the requester, and may propagate notifications concurrently to the receivers who are members of the group. Tables were used to provide details related to the UCM responsibilities and conditions (Figure 52(b)).

FIGURE 52. UCM and Responsibilities Information for “Initiate Call” Operation



The PTM-G scenarios are in essence quite similar to those of the GCS, i.e. the same basic behaviour patterns are found in both systems. The UCMs themselves were also constructed in a simi-

lar way (no integration with stubs and plug-ins, structure that involves the same types of components and channels, use of tables for detailed descriptions, etc.).

8.2.2 Construction of the LOTOS Prototype

Building on the experience gained with the GCS system (Section 8.1), the construction guidelines (Section 5.2) were applied to the nine UCMs without any major problem. Again, the scenarios were integrated at the LOTOS level rather than at the UCM level. The core of the logic is found in the process Controller (see Figure 52(a)), which receives all requests and select the appropriate operation, all of which are specified as individual LOTOS processes for improved modularity.

The construction complexity resulted mainly from the large number of abstract data types required to represent the various databases involved in this system. LOTOS ADTs are cumbersome when used to describe complex data structures such as lists of lists of lists of records, which were used on several occasions in this experiment.

Whereas only the server side was specified in the GCS experiment, the PTM-G specification includes both the server side (PLMN) and the client side (MS). The PLMN and MS process definitions were constructed in a robust way; if one side does not work properly, then additional conditions and behaviour would ensure that the other side would go back to a stable state. Such robustness, although claimed to be desirable in the draft standard requirements [128], was described in an obscure, ambiguous, and incomplete way. Hence, several design decisions were taken at the specification level.

The resulting specification is composed of 53 ADT definitions (1125 lines of LOTOS code), 7 processes for the MS (300 lines), and 23 processes for the PLMN, where the group call service was defined (1100 lines). To date, this is one of the most complex specifications constructed using SPEC-VALUE.

8.2.3 Test Selection and Execution

Testing patterns were used on each individual UCM in order to generate test goals. The resulting test suite was limited mainly to acceptance tests (35 test cases divided into 9 test groups). Only one rejection test was created, essentially for illustrative purpose, because the robust PLMN server accepts almost any operation at any time and always replies with acknowledgements or error messages. The absence of rejection tests was partially compensated by the use of verification steps in the test cases, which are more suited to this type of robust specification.

Many abstract sequences derived from the UCMs were reused as preambles and verification steps for other tests. Preambles are sequences of events that satisfy the preconditions of a scenario under test. For instance, to test the Initiate Call operation (Figure 52(a)), the preamble could include abstract sequences from the Attach GPRS and Join Call UCMs to ensure that the requester gets properly attached to the network. Verification steps are sequences of events that check some aspect of the system's current state. At the end of a test for Initiate Call, an abstract sequence based on the Call Status UCM could be invoked to verify that the group call was correctly initiated in the PLMN. Abstract sequences were very reusable in the context of this experiment, and they helped improving the consistency among the test cases.

The PTM-G specification was initialized with a static configuration of users and databases used as the main context for the execution of the test suite. This configuration was defined in way that enables the satisfaction of the preconditions of all the UCM paths selected as test purposes. Examination of these test preconditions revealed that a single configuration with six mobile stations is sufficient for allowing all the 36 test purposes to be fulfilled. These six mobile stations have different identities, locations, and privileges, they belong to diverse groups, and they are all initially detached from the GPRS network. The five databases are initialized with sufficient information to eventually satisfy all the preconditions associated to the UCMs.

The test suite was checked first against the composition of the servers (PLMN) and the clients (MS), and then against the server alone. The rationale for testing the server alone is that the server is no longer constrained by well-behaved clients, and robustness can be checked more easily. The ver-

dicts provided by LOLA were however the same in both cases because the validation test suite did not explicitly test robustness.

As expected, the test suite led to incorrect traces that were used to diagnose bugs in the specification (due to the ADTs, to the guards, or to unfeasible synchronizations between processes). After several small iterations, the 36 test cases (800 lines of LOTOS code) were successfully executed in 2 minutes. While the length of these test cases varied between 2 and 31 observable events, the length of some execution traces (test runs) reached 155 events with the internal events. The verification of such large 155-event deep LTS explains the time taken by LOLA to validate the specification.

8.2.4 Structural Coverage

The improved insertion technique was used to instrument the MS and PLMN process definitions respectively with 30 and 69 probes. This addition of 99 new internal event to an already-large specification resulted in a state explosion problem, and LOLA's *TestExpand* command could not be used. In deed, the length of some traces exceeded 185 events. However, since probes can be covered in smaller groups by checking the test suite against each sub-group (as suggested in Section 7.6.1), the structural coverage could be measured compositionally. Coverage measurements were first collected for the specification with probes in the MS (client), and then for the specification with probes in the PLMN (server). LOLA was able to cope with this state space and it returned results after several hours.

The long time taken to measure the structural coverage in this experiment was not entirely satisfactory. An alternative solution came up through the use of LOLA's *OneExpand* command. Rather than attempting to do a full exploration of the state space, this command allows for the generation of a given number of random traces (partial exploration) reachable from the composition of a test and the specification. Moreover, *OneExpand* does not suffer from the bug found in *TestExpand* (Section 7.5.2). A partial exploration is more than adequate to show that a probe can be reached, but it cannot ensure that a probe is unreachable by a test. Nevertheless, by extracting 5 traces for each test, the coverage results provided by *OneExpand* were the same as those provided by *TestExpand*, only this time 81 seconds were necessary to generate them. This pragmatic and efficient solution was also used on all the other specifications discussed in this chapter (see row *z* in Table 30).

Since the robustness of the PLMN and the MS was defined at the LOTOS level but not in the UCMs, ten probes were expected to remain unvisited as a result of events that should not occur in the normal use of the system. Indeed, these ten probes were missed by the validation test suite. Another probe was however missed in the PLMN; this unvisited probe highlighted a portion of the code that was useless. This code and its probe were removed from the specification.

The validation test suite was meant to be used on the composition of the PLMN and several MS. It could have been completed with robustness test cases for the PLMN process alone and for the MS process alone. LOLA demonstrated that it was possible to create traces that visit these “robustness” probes using step-by-step execution of the corresponding processes. These traces can serve as a basis for the generation of additional robustness test cases for the PLMN and the MS processes. However, such tests were not added as part of the experiment.

8.2.5 Discussion

The application of SPEC-VALUE to PTM-G service raised many questions about the GPRS standard. Multiple errors, inconsistencies, and ambiguities were unveiled as a result of the application of SPEC-VALUE. Some of the problems that were uncovered include:

- *Sending of indications*: for the successful execution of operations Join Call and Leave Call, it is not clear whether an indication is sent to all participants or not.
- *Rejection cause*: when a rejection is provided (e.g. for a Call Initiation request), the end-user receives an ambiguous answer that can be interpreted in many ways, making it difficult to diagnose the reason of the rejection.
- *Restrictions of joining calls*: in a Join Call operation, there is no restriction to the calls which a subscriber could ask to join. This potential flaw in the design raises many security and privacy issues. It was decided to constrain the use of this operation in the LOTOS prototype.

In most cases, the descriptions are operational and supported by informal figures in the standard. Nevertheless, they represent only partial scenarios, and no system view of the functionalities is

provided. Improving these descriptions with UCMs would represent a good step towards avoiding problems similar to those enumerated above. Moreover, such standards could gain in quality, consistency and completeness if rigorously tested and validated using an approach like SPEC-VALUE.

The first versions of the PTM-G UCMs, prototype and test suite were created mainly by an undergraduate student (P. Forhan), who initially was not familiar with any of GPRS, UCMs, LOTOS and its tools, or SPEC-VALUE. Nevertheless, it took him less than 5 months to gain an understanding of this mobile group call service and to produce useful documentation, concise and descriptive scenarios, a validated specification, and a test suite in which we have a high level of confidence. He based his work on an earlier study of the GCS system, for which the UCM scenarios were generic enough to be reused for the design of the PTM-G service. Since the structures of these two systems are not alike, this reuse of scenarios would have been more difficult with component-based scenarios such as Message Sequence Charts.

This experiment also showed that functionalities can be added incrementally to the system. For PTM-G, the integration and validation started with the operations that were independent of the others, followed by the operations whose dependencies were already specified (in order: *Attach GPRS*, then *Detach GPRS*, then *Initiate Call*, then *Call Status*, then *Join Call*, *Data Transfer* and *Change Zone*, then *Leave Call* and *Terminate Call*). With UCMs, adding new functionalities when the structure is stable is no more difficult than doing it with scenarios based on components and message exchanges. However, when a new functionality requires modifications to the structure, the impact appears to be softened when UCMs are involved. The scenario paths can be easily adapted to the new structure by reallocating the responsibilities, hence providing a traceable link to the components/processes in the specification whose behaviour needs to be adapted. Doing this remodeling requires more efforts with scenarios based on MSCs or the like.

The limitations of LOLA in terms of handling large state spaces for structural coverage measurement were reached with this example. Fortunately, relief strategies based on compositional cover-

age of probes and on the use of *OneExpand* for partial exploration of the state space have proven to be effective and pragmatic.

Another lesson relates to specification structures. In client-server systems, not only should the specification be suitable for system-level validation (e.g. testing clients composed with servers), but it should also allow for unit-level robustness testing (e.g. testing clients and servers as standalone entities). The PTM-G specification was created and tested in this way.

Finally, when robustness is involved in the specification but not in the requirements or in the UCM scenarios, then verification steps appear to be more useful than rejection test cases based on mutation of acceptance test cases.

8.3 Feature Interactions (FI)

This section discusses the specification and validation of a set of telephony features described in the *First Feature Interaction Contest* [161]. As explained in Section 2.1.2, the telecommunication industry usually understands features as customer services packaged into marketable units. In this experiment, a special emphasis is put on the avoidance and the detection of undesirable interactions between features. Such interactions still represent nowadays a complex problem that telecommunication systems designers must face [65][85][159][230][385], and this situation is likely to remain challenging in the future.

By definition, features interact with each other and with the basic system services, which is in many cases the so-called *Plain Old Telephone System* (POTS). However, a feature might be prevented from working according to its intent because of some unexpected interactions with other features in the system. This is at the heart of the *feature interaction* (FI) problem. Similar challenges can be found in the agent community where agent goals might be conflicting and impossible to fulfil simultaneously [42][160]. For the last decade, many partial solutions have been suggested to avoid, detect, analyze, and solve feature interactions at design time and run time. Keck provides an interesting survey on the FI problem and related solutions in [227].

The approach taken in this experiment aims to facilitate the creation of an interaction-avoidant design, and to detect remaining interactions at design time with the help of an executable prototype. Avoidance of interactions between operational requirements is achieved through the visual integration of scenarios expressed with UCMs, and the detection of remaining interactions is achieved through testing of LOTOS prototypes.

LOTOS has been used for years for the specification and validation of telephony systems. Boumezbear, Faci, Logrippo and Stépien have pioneered many approaches based on LOTOS [66][129][130][133]. The availability of executable LOTOS prototypes enabled the detection of feature interactions. Different such techniques were developed by Logrippo and his collaborators (Faci [131][132], Fu *et al.* [141][142], Kamoun [222][223], and Stépien [338][339]), by Turner [352][353][354][355], and by Thomas [343]. One of the challenges in using this language consists in writing the first system specification from informal requirements. However, once a specification is available, rigorous methods can be used to validate and verify the specification.

Use cases have been utilized for the analysis of interactions in telephony systems by Kimbler and Søbirk [229]. More recently, Buhr *et al.* used UCMs to tackle the problems of feature interactions and resolution of conflicts in plain telephone systems [79] and in multi-agent systems [77][78]. The UCM notation helps designers with the visualization of problematic situations and their avoidance at a high level of abstraction. However, UCMs do not support formal validation and verification directly.

With such knowledge and experience available, a methodology that would use the best characteristics of UCMs (e.g. visual description and integration of features) and LOTOS (e.g. powerful theory and tools for validation and verification) for the avoidance and detection of interactions in telecommunications systems seems like a natural evolution. SPEC-VALUE is therefore suited as a candidate for demonstrating that integrating UCMs together helps avoiding many interactions before any prototype is generated, and that LOTOS prototypes can be used to detect remaining interactions. Detailed descriptions of this experiment can be found in a technical report [18] and a publication [22]. Additional insights on the use of UCMs as a feature description notation can be found in [28].

8.3.1 System Overview and UCM Descriptions

The first FI detection contest, organized by Griffeth *et al.* in [161], describes twelve features whose choice has been dictated by the need for them to interact. The network is modeled as a collection of black boxes communicating with each other via defined interfaces. Definitions for POTS and the features are given as informal requirements (in English) and as Chisel diagrams [4].

The graphical language Chisel is used for defining requirements of communication services. Chisel diagrams are trees whose branches represent sequences of (synchronous) events taking place on component interfaces, whereas UCMs are described in terms of responsibilities performed by components. In essence, the Chisel language is at the level of synchronous messages, and so UCM scenarios have to abstract from this level. Although it is possible to use Chisel diagrams directly to generate LOTOS specifications and test cases, this is not the purpose of this experiment. Since the focus of SPEC-VALUE is on the capture of informal requirements, the given Chisel diagrams are considered as such. Further information on the transition from Chisel to UCMs is given in [18], whereas Turner discusses the direct translation of Chisel into LOTOS [354].

Features

The twelve features include switch-based services as well as services based on the *Intelligent Networks* (IN) reference model [202]. On top of POTS, the first phase of the contest described ten features, but this experiment mainly focuses on four of them:

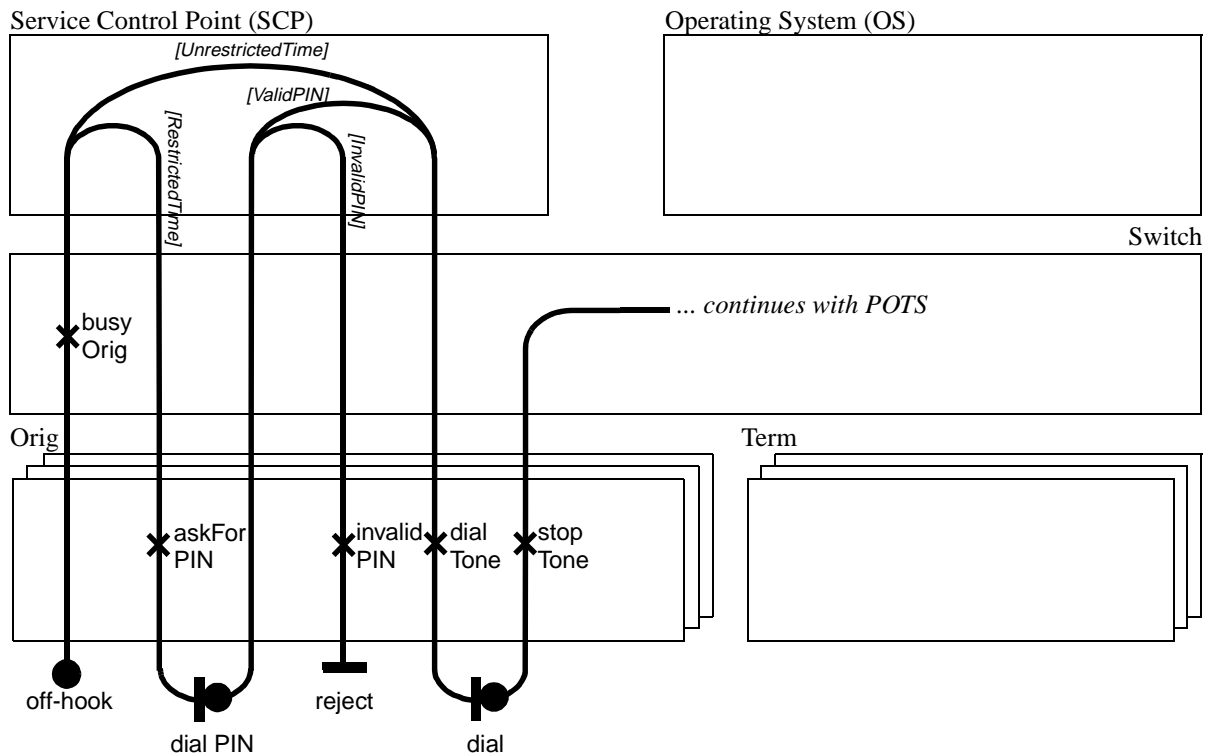
- *IN Teen Line* (INTL): restricts outgoing calls based on the time of day (i.e. hours when homework should be the primary activity). This can be overridden on a per-call basis by anyone with the proper personal identity number (PIN).
- *Calling Number Delivery* (CND): allows the called telephone to receive a calling party's directory number and the date and time.
- *IN Freephone Billing* (INFB): allows the subscriber to pay for incoming calls.
- *Terminating Call Screening* (TCS): restricts incoming calls from lines that appear on a screening list.

The six remaining features were *IN Freephone Routing*, *Call Forwarding Busy Line*, *Three-way Calling*, *IN Call Forwarding*, *Call Waiting*, and *Charge Call*. The second phase contained two additional features, namely *Cellular* (air-time fees) and *Return Call*. The UCMs developed in this phase considered a third additional feature as well, namely *Automatic Call Back*.

Use Case Maps

As an example, Figure 53 shows a partial UCM for the INTL feature. Components like the Switch, the users (Originator role and Terminator role), the Service Control Point (SCP), and the Operating System (OS) are all described in the original requirements. Some events in the requirements become responsibilities local to the switch (like setting the *busy* status of the originator), others become responsibilities that the user can observe (like getting an announcement *askForPIN*), and others remain events that the user can trigger (like *off-hook*).

FIGURE 53. Partial UCM for INTL



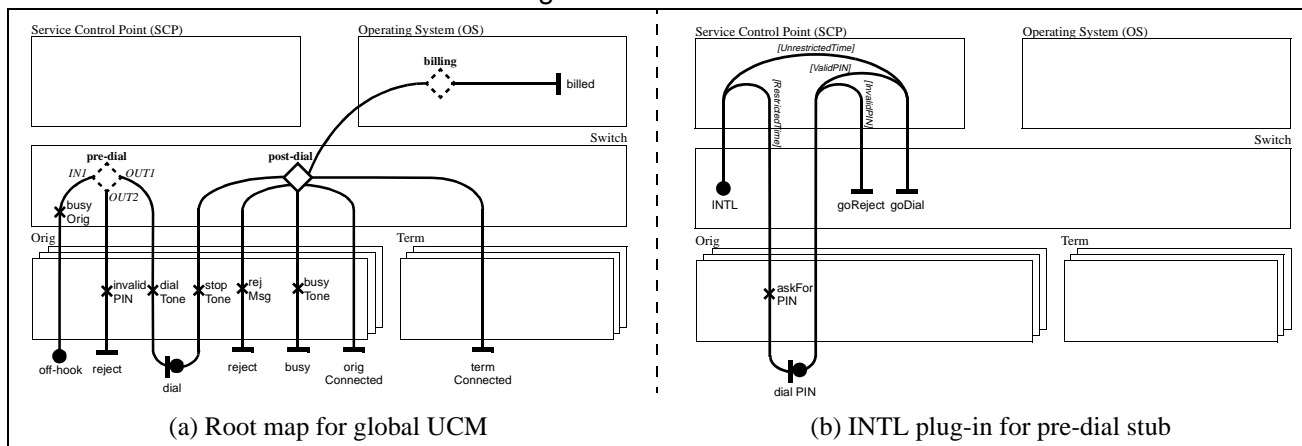
Similar individual scenarios were defined for POTS and for all thirteen features.

Integration of UCM Scenarios

Individual scenarios are useful for understanding the behaviour of one feature, but they can also be integrated together to form a *global UCM*. The assumption here is that performing the integration at this level of abstraction provides early insights in possible conflicts between features expressed as scenarios. Integration helps to ensure early consistency between individual maps. For instance, events and responsibilities that are labeled incorrectly, omitted, at different levels of abstraction, or in different orders become hard to integrate. Integration also helps to avoid ambiguous situations, the most common being non-determinism. A path segment that is a prefix to two different scenarios might imply the need for a way to decide which alternative to take in a global scenario. Many such design decisions can be made during the integration.

The following root map results from the integration of the twelve features of the contest. This integration was done using the *UCM Navigator* tool [257]. The root map in Figure 54(a) represents the global context in which sub-maps representing the features are plugged in.

FIGURE 54. Global UCM and INTL Plug-in



One constructs a complete scenario by selecting appropriate plug-ins for the stubs. The first stub in the root map, pre-dial, has one entry point (*INI*), and two exit points (*OUT1*, *OUT2*). The pre-

dial stub has two plug-ins, one of which (INTL) is illustrated in Figure 54(b). The binding of the INTL plug-in is $\{(INTL, INI), (goDial, OUT1), (goReject, OUT2)\}$.

The other (default) plug-in for the pre-dial stub is an empty path that links *INI* to *OUTI*. The preconditions attached to these two plug-ins are disjoint, i.e. the user must be subscribed to INTL for this plug-in to be selectable, and the user must not be subscribed to INTL for the default plug-in to be selectable. Hence, the two plug-ins can never be active simultaneously. This alternate composition within the stub results from the nature of the individual features and from how they were integrated together. In essence, INTL is the only feature that deviates from all the others between the update of the busy status (*busyOrig*) and the dial tone (*dialTone*), and it overrides POTS according to the intent of INTL.

When an originator user is subscribed to INTL only, the flattening of the root map with the INTL plug-in in the pre-dial stub and default plug-ins in the other stubs results in the individual UCM of Figure 53. As for the other features of interest, the post-dial static stub contains only one default plug-in in which two other stubs are composed in sequence. The process-call stub contains plug-ins for TCS, INFB, and default behaviour, while the display stub contains a CND plug-in that can override the default plug-in.

A total of 23 plug-ins are used to describe the 13 features. A second root map was also created to take care of the disconnection phase in the system. The latter was not explicitly integrated to the other maps as the disconnection was easier to represent independently. These UCMs were extracted from the contest description mainly by D. Petriu, a master's student collaborating to this experiment.

Avoiding Feature Interactions

Integrating scenarios together at the level of UCMs promotes high-level reasoning about the overall system and helps avoid many interactions between features. For instance, many potential interactions between INTL, INFB (or TCS), and CND are avoided because the features in each possible pairwise

combination are allowed to proceed independently in the map. They are integrated using a sequence of three different stubs that encapsulate the features from their environment.

Important design decisions still need to be made at integration and composition time, something that cannot be easily automated. For example, interactions between features in one stub (e.g. INFB and TCS in process-call) are still possible, depending on the selection policy used within this stub. Maps with stubs show how localized the impact of a feature can be. This helps focusing on issues related to how a plug-in (i.e. dynamic behaviour) is selected in one or more dynamic stubs. Since only a limited number of smaller UCMs have to be considered in a stub, it becomes easier to check that they have disjoint and complete preconditions (to avoid non-determinism and unspecified behaviour), or that priorities need to be established. Hence, the design decisions are simpler. The integration then becomes a useful step in a design process that includes UCMs. The UCM notation helps us to reason about architectures and behaviour in order to create systems in which undesirable interactions are avoided early in the development cycle, rather than merely detected at a later stage.

8.3.2 Construction of the LOTOS Prototype

The construction guidelines have been used to build appropriate abstract data types, a structure of components representing the architecture, and internal process behaviour. Although only four features are fully described and validated (INTL, CND, TCS and INFB), the resulting specification supports the representation and the manipulation of information for the thirteen features described as UCMs. Also, the structure of the UCMs (with recursive definitions of stubs and plug-ins) is fully represented in LOTOS.

The abstract data types are mostly derived from the tabular representation of the feature parameters found in the contest description. Simple enumerated types describe time, user addresses, personal identification numbers, announcements, trigger names, response types, and feature names. More complex ADTs handle a database of subscriber information in the switch, log records for the billing in the OS, lists of features, feature parameters in the SCP, and a database of status items in the switch.

LOTOS gates are used to represent individual events shared between the five network components (see Figure 53). These components are represented as LOTOS processes and are synchronized on common gates. Each event in the Chisel diagrams of the contest description (i.e. each responsibility in the UCMs) is mapped onto a unique gate. Therefore, instead of using gate splitting for representing the on-hook and off-hook events on the user/switch interface (as in `user2switch!onHook` and `user2switch!offHook`), we have two individual gates (`onHook` and `offHook`). Having individual gates permits more specific compositions between processes and, more importantly, between the specification and the test processes.

Since this system is designed from the user's viewpoint, some events are observable while others remain hidden within the system. The observable events are the ones on the switch-to-user and user-to-switch interfaces, whereas the hidden events are those on the switch-to-SCP, SCP-to-switch, and to-OS interfaces. Three additional visible events are also created to improve the controllability and testability of the prototype. `Init` allows the initialization of all the databases used by the network components with users' values (likely to come from a test case). `CreateUser` is used to create users (originators and terminators) and specify their initial state. Finally, `Query`'s purpose is to allow a test case to verify the billing log at the end of the test.

The resulting specification is composed of 39 data types (750 lines of LOTOS code) and 13 process definitions (800 lines).

8.3.3 Test Selection and Execution

The test selection strategy used in this experiment differs slightly from that of a plain application of testing patterns to the set of UCMs. The availability of a semi-formal representation of the requirements (in the form of Chisel diagrams) enable the use of a more rigorous technique for the validation of the functional behaviour of the basic call model (POTS) and of individual features. Testing patterns were mostly used for validating the expected interactions between pairs of features.

Test Suite Structure

Often, LOTOS test processes are sequential, monolithic, and deterministic in nature. However, through process instantiation, LOTOS test processes can be built on top of each other, hence reusing part of previous test processes in new ones. Shared processes, which represent sub-sequences of test cases corresponding to common UCM abstract sequences, are used to structure the test suite. These processes are called *common behaviours* and they act in a way similar to test steps in the conformance testing methodology and framework [193].

Figure 55(b) shows the bottom level of LOTOS test processes, composed solely of common behaviour processes for POTS. They are reused by the POTS test cases, and also by common behaviour processes for individual features. On top of the latter, test processes for individual features, and also for each pair of features, are constructed. Common behaviour processes are reused by many test cases, which simplifies the generation of test suites and increases the consistency among test cases.

FIGURE 55. Construction of the FI Test Suite

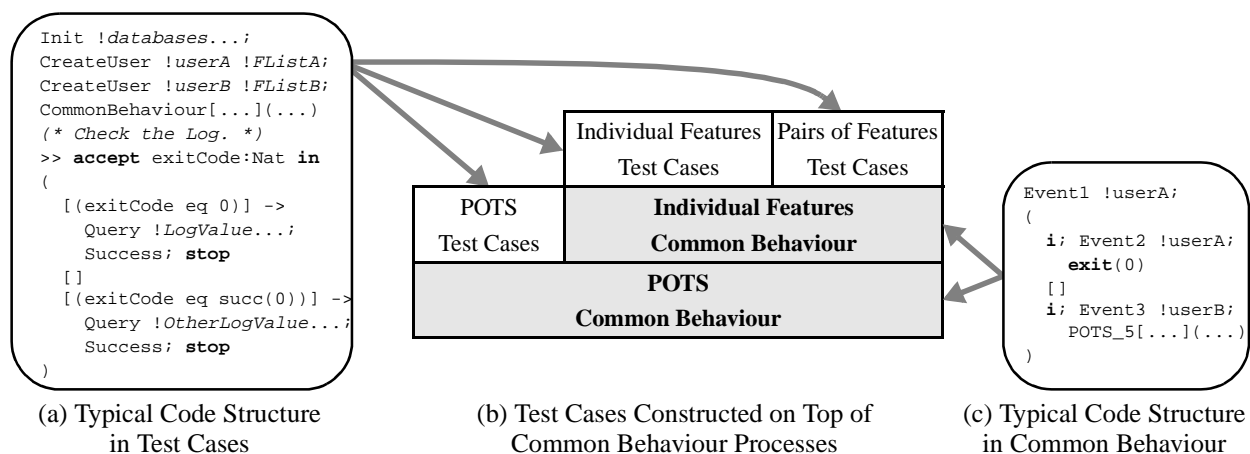


Figure 55(c) presents the typical code structure in common behaviour processes. They are mainly composed of simple expressions that terminate with an *exit code* (**exit**(*n*)). With LOLA, test cases do not need to be sequential or deterministic, so alternatives and explicit non-determinism are allowed in common behaviour processes. Note that many alternatives are preceded by the internal

action **i**. This non-determinism (motivated by the “canonical” nature of the testers) ensures, under LOLA, that all branches in the test case will be selected and covered at testing time.

In Figure 55(a), the typical code structure illustrates that a test case provides the system configuration and verifies the exit codes. More specifically, the system is first initialized (with `Init`), users are created dynamically (with `CreateUser`), and the test cases themselves are performed by instantiating common behaviour processes. The exit code is then captured and used to validate the billing log against its predicted value. This verification step is performed on the `Query` gate.

During the testing, a deadlock in a test case for POTS or for an individual feature indicates an error that needs to be fixed. When all these test cases pass successfully, a deadlock in a test case for a pair of features indicates an unexpected interaction.

Testing POTS and Individual Features

Test cases generated from POTS and individual features check that each feature acts properly when being the only one active (Strategy 6.B), and that POTS acts properly in the absence of any active feature (Strategy 6.A). Often, more than one test case is required to cover one requirement, because initial states and conditions are necessary. POTS has only one *precondition*: whether or not the terminator side is busy. Hence, two test processes, one for each initial configuration, are necessary (200 lines of LOTOS code). They make use of six POTS common behaviour processes, corresponding to POTS states referred to by the Chisel diagrams of other features. This is one of the reasons why such common behaviour can be so easily reused. The two POTS test cases cover all the sequences found in the corresponding Chisel diagram.

Ten test processes are used to validate the features INTL, CND, INFB and TCS (400 lines of LOTOS code). They make use of six additional common behaviour processes. Each test is executed by providing an initial configuration according to the conditions included in the individual UCMs. For instance, three configurations are required for INTL: *UnrestrictedTime*, *RestrictedTime* \wedge *ValidPIN*,

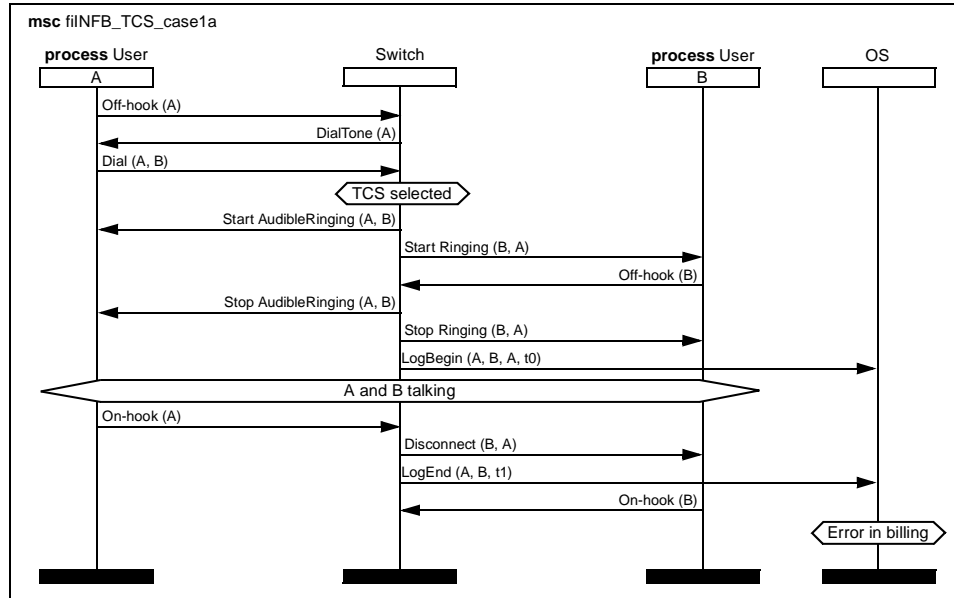
$RestrictedTime \wedge InvalidPIN$. They cover the three partial UCM routes (abstract sequences) found in Figure 53. In effect, this corresponds to the application of Strategy 1.B (Alternative — All paths).

Detecting Unexpected Interactions

In theory, if the same type of integration used for merging the individual UCMs is used again during the generation of the test cases for pairs of features, then there should not be any problem, and perhaps not a single unexpected interaction. In practice however, integrating the expected behaviour of two features in a test sequence is much easier than integrating n features in a system (where $n > 2$). This is one of the main reasons why tests for pairs of features are necessary. Although they cannot cover everything there is to check, they represent a pragmatic and efficient way of attacking the problems of conformance to the requirements and interoperability between features.

A set of 25 test cases (725 lines) was generated by application of Strategy 6.C to the UCM paths where plug-ins for pairs of features are selected. The meaningful combinations of preconditions also helped selecting these test cases, which aimed to validate the features according to the intent expressed by the global UCM. In the first iteration after the successful execution of test cases for POTS and individual features, all the FI test cases passed successfully, except the test for the pair INFB-TCS. LOLA returned three traces that led to unexpected deadlocks. One such trace is illustrated as an MSC in Figure 56: the idle terminator (B) has subscribed to INFB and TCS, and the originator is not on the screening list. In this scenario, the originator (A) on-hooks first, but it is also billed instead of the terminator. The error in the billing was detected when the test case queried the log from the OS and could not synchronize on the expected value. The problem here is that the TCS plug-in was selected, but not INFB. Hence, the person to be billed was the default one, i.e. the originator.

FIGURE 56. A Feature Interaction Between TCS and INFB



The second interaction trace is similar, but the originator on-hooks first. In the third trace, the originator is on the terminator’s screening list. The call should be blocked by TCS, but it goes through because INFB was selected while TCS was not.

The choice between the TCS plug-in and the INFB plug-in in stub process-call, which both override the default plug-in, was left open (i.e. non-deterministic). When integrating the UCMs, it was not clear whether other types of constraints were necessary for these two features to work properly together. Even from a UCM perspective, a mutual exclusion would cause problems, but this is a detail that was buried down in the selection policy within the stub. This is why a more precise and rigorous detection technique appears necessary once the integration is completed.

A sensible solution to this problem would be to give a sequential priority to TCS over INFB in the stub, i.e. INFB would be selected only if TCS allows it. This new composition was specified and the FI test cases adapted according to this new composition. In the end all the test cases (POTS, individual features, and pairs of features) passed successfully. No expected interactions seemed to remain in the global specification.

8.3.4 Structural Coverage

This specification was instrumented with 55 probes in order to cover 49 sequences (94 LOTOS events) and 27 simple BBEs. Structural coverage measures were taken using LOLA's *TestExpand* and *OneExpand* commands. Both approaches provided the same results, but *TestExpand* required 165 seconds whereas *OneExpand* only took 37 seconds.

The testing performed in Section 8.3.3 being quite exhaustive, no probe was expected to be missed. However, four probes were not covered by the test suite. This partial coverage is caused by the specification of only four features out of the thirteen described in the UCMs rather than by the incompleteness of the test suite. The specification includes paths corresponding to stub outgoing segments (hooks) that are not used by the four selected features. The specification of the remaining features and the generation of their validation test cases would undoubtedly lead to a full structural coverage.

8.3.5 Discussion

This FI experiment leads to the discussion of four new topics: the incremental addition of new features, the adequacy of the underlying call structure, some limitations of the UCM notation, and the benefits of this approach when compared to related techniques for feature interactions.

Incremental Addition of New Features

Adding new features has a direct impact on the global UCM, the specification, and the test suite. In this experiment, the integration of three new features (the second phase of the FI contest) to the first ten ones, which were already integrated together, did not have a major impact on the global UCM. The root map did not have to change, but a new stub and a new output path had to be added to a plug-in. The impact has shown to be proportional to how coupled the features are in a map. The more a map is modularized (by using stubs), the less it is likely that major modifications will be necessary. Since the LOTOS specification reflects the global UCM, the impact on the prototype is basically the same. A few new gates and appropriate data structures, together with new processes for specifying the new plug-ins, are required.

The addition of a feature however has a profound impact on the test suite. The number of pairs among n features is $n*(n+1)/2$ (if we assume that a feature may interact with itself), and for each pair the number of test cases grows exponentially with the number of distinct conditions to be covered. The impact of the integration of a new feature will be higher if new types of conditions (e.g. preconditions attached to start points or OR-forks) have to be accounted for in the input domain. However, UCMs can help determine some unnecessary combinations of conditions by following the paths and their associated conditions. For instance, when *UnrestrictedTime* is true in INTL (Figure 53), whether the PIN is valid or not has no impact, hence one of these two combinations can be dropped. Nevertheless, more scenarios need to be pruned out for combinations of features.

Structure Adequacy

The abstract underlying structure in the UCMs appears to be insufficient for the specification of several remaining features. The current behaviour of the switch component (Figure 53) is tightly coupled to the progression of one unique call session. For call sessions involving more than two parties (e.g. for the three-way calling and call waiting features), the current call structure needs to be improved. Call sessions need to be instantiated upon request, and the user status database needs to be decoupled from the switch process in order to be accessible to these sessions. Such improved structure would be similar to those used in many LOTOS specifications for telephony systems [129][133]. The UCM structure, derived from the network architecture given in the requirements, did not specify anything about the internals of the switch, and it was specified as is in the prototype. However, for the sake of extensibility of the specification, new components are needed at the specification level, and they probably need to be mirrored at the UCM level. Such improved structure was not used in this experiment, but the lesson here is that complex scenarios might lead to the discovery of new components at the UCM and LOTOS levels.

Limitations of Plug-ins, Bindings, and Compositions

The following limitations of the UCM notation were observed while integrating the scenarios:

- Although the stub/plugin mechanism is useful for abstraction, modularity, and dynamic behaviour, its use in a global map makes the end-to-end scenarios more difficult to visualize at a first glance. Often, the reader has to mentally flatten the global map to get a better understanding of these scenarios.
- The binding of a plug-in to a stub is done through an external mechanism (the binding relation, together with an optional selection policy), which is not visual.

Care has to be taken when using stubs and plug-ins, otherwise their use might defeat one of the intents of UCMs which is to provide a useful bird's eye view of the system.

Comparison with Related Techniques

In [132], Faci presents a FI detection technique also based on the integration of scenarios and the use of the LOTOS testing theory. This approach makes a distinction between the concepts of composition and integration. *Composition*, noted $f_1|[]|f_2$, expresses the synchronization of features on their common actions with POTS and their interleaving on their independent actions. *Integration*, noted f_1*f_2 , expresses the extension of POTS with n features (two in his examples), such that each feature is able to execute all of its actions which are allowed in the context of POTS, when the other features are disabled. Features are captured as labeled transition systems instead of as UCMs. Integration relates very well to the UCM integration, whereas the composition simply represents the generalized synchronization operator and does not relate to anything specific in SPEC-VALUE. Faci's approach states that an interaction exists between n features if their *integration* does not *conform* to their *composition*.

Conformance is checked through validation test cases, from the user's point of view, similarly to what was done in this FI experiment. Test cases are derived manually (using "knowledge and experience") from the composition specification, and then they are applied to the integration specification. When a deadlock occurs between a test case and the integration specification, an interaction is said to be detected. This last specification is generated manually at the LTS level, which is far less scalable and modular than generating specifications from global UCMs. Indeed, all the examples provided in this thesis contained only two features integrated together, for obvious complexity reasons. UCMs are

a means to integrate scenarios while avoiding some interactions, and they allow for multiple complex features to be considered.

Faci's approach leads to multiple feature interactions that are somewhat spurious or easily avoidable. Indeed, any integration operator (*) other than the generalized synchronization ([[]]) is very likely to cause deadlock situations. The test suite, although it could be generated almost automatically from the composition, is of low quality as it does not consider the way the features were integrated together. Many test cases are *may tests* and would deadlock when composed with the original behaviour. The test suites generated from UCMs are much more representative of the intended system behaviour, they lead to simpler diagnostics (due to their acceptance/rejection nature), and they are more likely to be reusable down the road towards the implementation.

In [141][142], Fu *et al.* applied two different FI detection techniques to the set of features described in the FI detection contest and prototyped in LOTOS. The first one uses observer processes that check, at run-time, whether individual feature properties hold in the presence of other features. The second technique uses the billing information (log) to catch interactions both during and after the execution of test scenarios. These scenarios are very loose and cover large portions of the state space. In an integrated approach to the specification and verification of features, both these FI detection techniques could potentially be used in addition to the UCM-based prototyping and validation.

Turner applied CRESS, reviewed in Section 3.3.4, to the same set of features [354][357]. Using the CRESS toolset, LOTOS models are constructed automatically from formalized Chisel diagrams capturing the individual features. Several types of interactions (e.g. related to consistency or non-determinism) are detected at integration time using static semantic rules. Additional interactions are detected using conventional LOTOS techniques, including testing with LOLA. The author claims that the specifications produced by this toolset are smaller better structured than similar ones written manually in [142]. However, these specifications suffer from a pre-determined composition mechanism and target mainly the detection of interactions, not necessarily an integration that would solve or

avoid them as it is the case with UCMs and selection policies. There is no strategy for the selection and generation of test cases, especially for the ones targeting desirable feature interactions.

Nakamura *et al.* [268] and Hassine [170] recently proposed filtering approaches that help extracting interaction-prone scenarios based on the configuration of UCM stubs and plug-ins. These techniques are based on an assumption used in this experiment, i.e. features that override default plug-ins in different stubs are not likely to interact whereas features that attempt to override default plug-ins in the same stubs are likely to interact unless appropriate measures (selection policies) are taken. Hassine's approach further validates these scenarios using a LOTOS specification. Gorse uses another FI filtering technique in combination with UCMs, only this time it is based on a more abstract representation of the requirements (preconditions, input events, and resulting events/postconditions only) [156][157]. For each suspected interaction, scenarios can be generated directly and mapped onto UCMs or LOTOS test cases. Other filtering techniques (e.g. Keck's [226]) could also be adapted to the UCM context. All of these can be combined to SPEC-VALUE to improve the detection of potential interactions and the selection of suitable validation test cases.

8.4 Agent-Based Simplified Basic Call (SBC)

This experiment is part of a proof-of-concept study for an industrial project. The industrial partner provided a collection of integrated UCMs representing the high-level design of an agent-based private branch exchange (PBX). These UCMs include many features structured with stubs and plug-ins in a way similar to the FI specification of Section 8.3 and to the TTS example. In this experiment, the basic call model was extracted from these UCMs and then simplified. The initial set of more than 100 UCMs was reduced to a collection of 4 UCMs which no longer contained any stub; the handling of stubs and plug-ins was postponed to a later phase of this project. These UCMs were used as an example to demonstrate the potential of the SPEC-VALUE approach in an industrial context.

This section reports on the prototyping and validation of this agent-based Simplified Basic Call (SBC) model. The application of SPEC-VALUE to the whole set of UCMs and features (the second phase of this industrial project), together with the generation of functional test cases and the

detection of undesirable interactions, is discussed further in [25]. The results of this second phase are not included in this thesis because the work was mainly done by other students as part of their own theses.

8.4.1 System Overview and UCM Descriptions

The system in consideration uses a new agent-based architecture being designed by the industrial partners [369]. Call processing goes through four types of components:

- *Device Entity Blocks (DEBs)*: Device agents handling the devices, i.e. the physical end-point of a call (e.g. a telephone, a computer capable of voice over IP, a diary, etc.).
- *Communication Entity Blocks (CEBs)*: Personal agents aware of the restrictions and privileges given to specific users. They also know about the preference relationships between a user and his/her devices. CEBs communicate with DEBs and LEBs.
- *Logical Entity Blocks (LEBs)*: Agents representing the functional role of the end point of a call (e.g. president, director, secretary, etc.).
- *Call Objects (COs)*: Dynamically instantiated objects that handle calls by binding LEBs together in a call session.

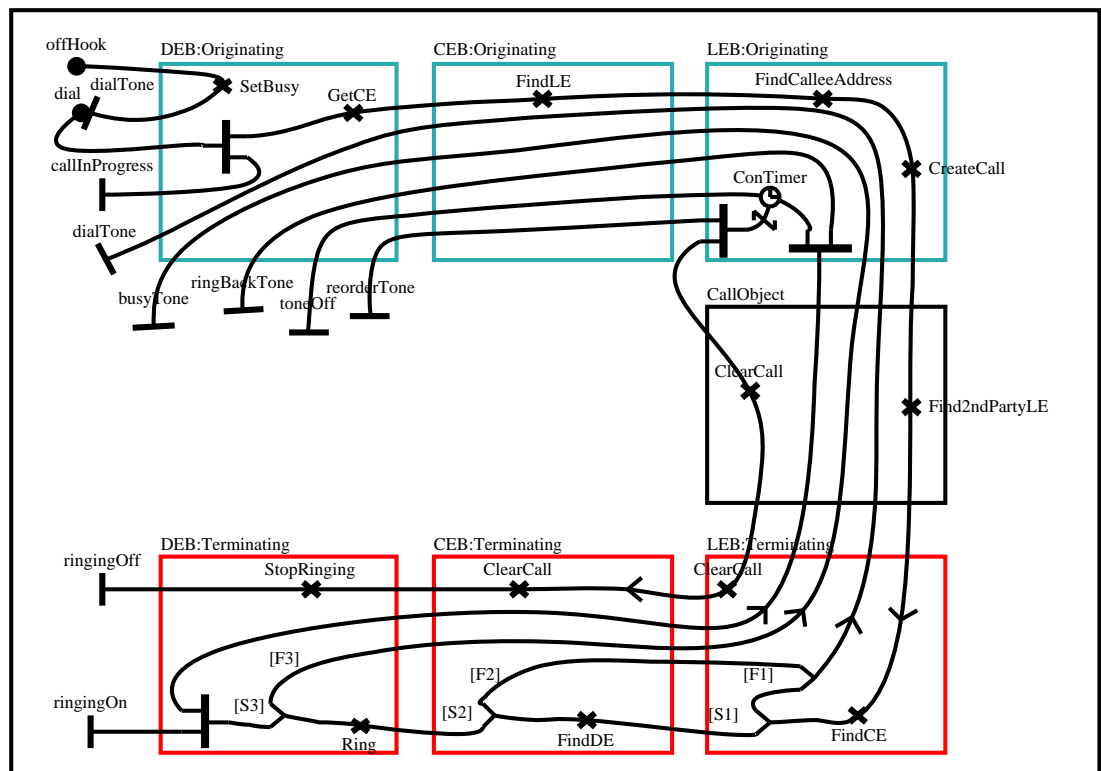
Many instances of these components can be active simultaneously for one user. From a scenario perspective, it is also helpful to distinguish between the roles played by DEBs, CEBs, and LEBs (e.g. originating or terminating). Call objects are similar to the call sessions suggested as a structural improvement for the FI specification in Section 8.3.5.

SBC is modeled using four UCMs, related through their preconditions and postconditions:

- *SimplifiedBasicCall*, the main UCM, where the originating party attempts to initiate a call. This UCM, created with the UCM Navigator [257], is showed in Figure 57.
- *Answer*: this UCM follows *SimplifiedBasicCall* when the phone is ringing on the terminating side. If the terminating party answers, then the phone stops ringing, the connection timer is released, and the connection is established.

- *HangUpOrig*: UCM that represents the termination of a call or of an attempt to make a call by the originating party.
- *HangUpTerm*: UCM symmetrical to *HangUpOrig*, but involving the terminating party.

FIGURE 57. SimplifiedBasicCall UCM for SBC



The three last UCMs, although not shown here, are rather simple. The apparent complexity of the SimplifiedBasicCall UCM results from the flattening of numerous stubs with default plug-ins in the initial collection of UCMs. The use of stubs and plug-ins in the latter makes it more structured and readable.

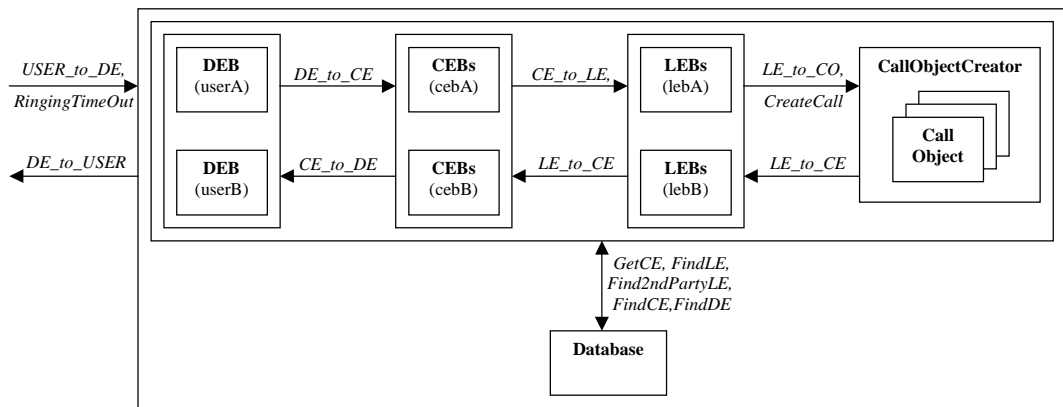
In Figure 57, originating and terminating roles are used for DEBs, CEBs and LEBs. The start points leading to the DEBs and the end points coming out of the DEBs represent user interactions. All the responsibilities are hidden in their respective components.

8.4.2 Construction of the LOTOS Prototype

The construction guidelines and the experience gained with the FI experiment helped determining an appropriate structure for the LOTOS prototype. The integration of the four UCMs cited in the previous section is done directly at the LOTOS level, in a way similar to the GCS and PTM-G examples.

The DEBs, CEBs, LEBs and COs are represented as LOTOS processes, as shown in Figure 58. Multiple instances of these components are used to represent the different users such as *A* (originating party) and *B* (terminating party). These components communicate through unidirectional channels specified as gates (arrows in Figure 58). Also, individual component instances, uniquely identified, use gate splitting to communicate.

FIGURE 58. Structure of the LOTOS Specification



A single Database process is used to simulate all the data repositories assumed to exist in the components but which are not defined explicitly in the UCMs.

In this structure, DEBs are instantiated statically: for any particular DEB, there exists one and only one instance. Each DEB process is instantiated with values corresponding to its identifier and the user with whom it is associated. Other Agents (CEBs, LEBs and Call Objects) are managed in a more dynamic fashion. CEBs and LEBs can be involved in several communication scenarios simultaneously. To permit this, CEB and LEB processes are designed such that there may be many instances of them at the same time (up to three in this specification). Call Objects (COs) are created dynami-

cally by LEBs through the CallObjectCreator process. The COs may be used by any number of LEBs, so this structure enables multi-party calls, something that was possible to do in the FI specification.

Simple abstract data types are defined to describe, compare and manipulate messages and component identifiers (8 ADTs, 200 lines of LOTOS code). Nine process definitions are used to describe the components and their associated factories (750 lines of LOTOS code).

One particularity of this prototype is that it includes a timeout mechanism, resulting from the construction guideline CG-4.a, used to specify the timer found in the SimplifiedBasicCall UCM. The timeout event is made visible in order to improve the controllability at testing time.

8.4.3 Test Selection and Execution

For the validation, the structure was initialized with two users (*A* and *B*), as shown in Figure 58. Simple testing patterns were used to guide the construction of nine acceptance test cases. However, the goal was to produce examples of functional scenarios, without much concern for the complete coverage of the UCM paths. Two rejection test cases were also added to this collection.

The first four test cases (A1 to A4 in Table 25) target the normal and expectable usage of the system, whereas tests A5 to A9 represent unsuccessful communications or exceptional scenarios which highlighted several problems with the prototype. In a sense, these last five test cases checked the robustness of the specification.

Tests R1 and R2 are two rejection test cases whose expected verdict is Reject. They are based on informal requirements rather than on the UCMs themselves.

Three problems were detected. Test A6 shows that the communication mechanism used inside CEBs and LEBs cannot always handle simultaneous hang-ups from both parties because of a race condition. Test A7 indicates that the situation is even worse when *B* hangs up at the same time as the timeout occurs in the LEB. Both these unexpected verdicts imply deficiencies in the way the SBC specification handles these two race conditions. This needs to be fixed at the LOTOS level since UCMs abstract from such details.

Test R2 indicates a problem with the disabling of the CallInProgress signal when a party goes on hook. Not only this needs to be fixed at the LOTOS level, but such a problem may also require remodeling or more details at the UCM level.

TABLE 25. SBC Test Suite and Verdicts

	Test	Description	Verdict	Problem
Acceptance	A1	A calls B, A hangs up first.	MUST	
	A2	A calls B, B hangs up first.	MUST	
	A3	A calls B, B is busy.	MUST	
	A4	A calls B, no answer (timeout).	MUST	
	A5	A calls A, gets busyTone.	MUST	
	A6	A talks to B, and they hang up at the same time.	MAY	CEBs and LEBs were unable to propagate internal messages in both directions at the same time, hence resulting in some unexpected deadlocks.
	A7	A calls B, and B off-hooks at the same time as the ringing timeout occurs.	REJECT	Race condition causing unexpected internal deadlocks (LEBs) in all test executions.
	A8	A calls B, B is busy. A hangs up, tries again, and gets busyTone.	MUST	
	A9	Tests A1 and A2 in sequence (2 calls).	MUST	
Rejection	R1	B off-hooks twice before hanging up.	REJECT	
	R2	CallInProgress signal should not occur after onHook/toneOff.	MUST	Test case that illustrates a problem with the UCMs, which was coded as is in the prototype. Since the callInProgress tone is sent to the originator concurrently with the rest of the call (see Figure 57), it is possible for this event to be observed even after the originator has gone on-hook.

These problems were not fixed in this experiment as the goal was to show problematic situations that validation test cases could uncover.

8.4.4 Structural Coverage

Though the SBC specification is not valid according to Definition 6.2 (*val* relation), the structural coverage was measured in order to determine additional potential problems. Using the improved insertion strategy, 64 probes were inserted in the specification, which represents only 29% of the total number of LOTOS events (204) and simple BBEs (20).

Using LOLA's OneExpand command, the resulting coverage measurements emphasized 17 probes missed by the validation test suite. Two main reasons explain this surprising result:

- Failure conditions are handled by several UCM paths (e.g. [F1], [F2] and [F3] in Figure 57) and the specification detects and reports them as appropriate. However, these conditions are not checked by the validation test suite. 12 unvisited probes fall in this category. Additional test cases are necessary to cover these UCM paths and hence check the robustness of the SBC design.
- As explained in Section 8.4.2, several instances (up to 3 in the specification) of CEBs, LEBs, and COs can be active simultaneously for one user. This type of structure is necessary to support several multi-party features designed on top of this SBC. However, the current validation test suite does not make use of all these instances at once. No third instance is ever created for any component, and hence 5 probes are not covered. This is another example of a specification structure that is intended to be extended with additional behaviour (features in this case).

Again, the validation test suite was not completed in this experiment.

8.4.5 Discussion

The SBC case study illustrated several types of problems that could be detected in an agent-based system using SPEC-VALUE. One lesson learned during this experiment is that if test cases focus too much on the paths related to expected user functionalities, then the robustness of the design might be forgotten along the way, even if this robustness is explicitly defined at the UCM level. The test plan should ensure that, through the use of testing patterns, all UCMs paths are covered. If this is not the case, then it is likely that the functional and structural coverages of the prototype will remain incomplete.

In the second phase of this project, SBC was extended to support the original set of UCMs (including stubs and plug-ins) [25]. The features considered so far are of different types (Outgoing Call Screening, Call Forward Always, Call Forward on Busy, Call Hold, Recall, Call Pickup and Call

Transfer), and new ones are to be added (Auto-Recall, Timed Reminder, and Call Waiting). The approach for FI avoidance and detection used in Section 8.3 is being applied. The main result of this experiment is to have shown, to the satisfaction of the industrial partner, that a design methodology for telephony switches based on the combined use of UCMs and LOTOS is feasible and effective in practice. Part of this methodology is also being integrated to the industrial partner's design process.

8.5 Self-Coverage of GSM Mobile Application Part (MAP)

SPEC-VALUE's structural coverage technique is applicable to specifications developed in various contexts. This experiment involves GSM's *Mobile Application Part* (MAP) [127], an ETSI standard which maintains consistency among databases frequently modified by mobile telephone users. MAP is composed of nine protocols for message exchanges between specific pairs of GSM components. The MAP specification and the test suite used here were not developed using SPEC-VALUE. This experiment discusses the use of the structural coverage technique presented in Chapter 7 for establishing the *self-coverage* of a specification by a test suite automatically generated from this specification.

8.5.1 Construction of the LOTOS Prototype

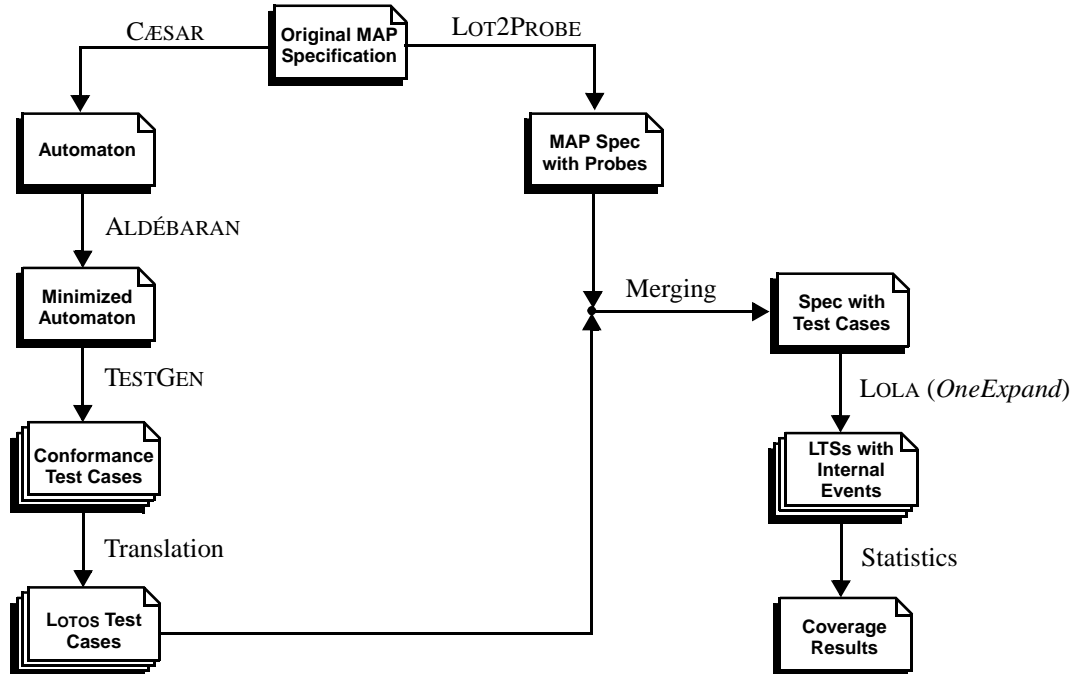
A student (H. Ben Fredj) derived a MAP specification in LOTOS directly and manually from the standard [127]. Scenarios and UCMs were not involved in the construction of this specification. The specifier validated the prototype manually through simulations and step-by-step execution. The resulting specification contains 14 process definitions (850 lines of LOTOS code) and 22 ADT definitions (375 lines). This specification is written in a fairly static style, i.e. processes are not instantiated dynamically. This style was chosen in order for tools such as CÆSAR to be able to extract a finite automaton from the specification [135][146].

8.5.2 Test Generation

The project aimed to generate an abstract conformance test suite *automatically* from the MAP specification in LOTOS and to compare it to a similar test suite generated from a MAP specification written in SDL. TESTGEN is the tool used to generate both abstract conformance test suites. The results of this comparison are outside the scope of the thesis, and hence are not discussed further.

As shown in Figure 59, the specification is first translated into an automaton by CÆSAR, and the resulting graph is minimized by ALDÉBARAN [135]. TESTGEN then generates a conformance test suite by doing a tour of the graph and by using Cavalli's *Unique Event* sequences [89][196], a unique identification of each state adapted to LOTOS from the concept of Unique Input/Output (UIO) sequences [360]. This test suite is converted back to LOTOS test processes in order to check whether or not the structural coverage of the "validated" MAP specification is achieved (hence the name *self-coverage*). A 100% coverage is obviously expected.

FIGURE 59. Self-Coverage of the MAP Specification



8.5.3 Structural Coverage

Three major iterations were needed to achieve a satisfactory structural coverage and a validated specification. In the first one, less than half of the probes were visited by the test suite (417 tests) because of a problem with the data types and guards which caused about half of the specification not to be reachable. This problem was totally missed during the initial validation.

The second iteration fixed this bug and resulted in an improved and longer test suite (603 tests), whose results are shown in Table 30. Several verdicts were wrong because of remaining non-determinism in the specification. This also caused problems with the generated test suite, which couldn't cover 17 probes in the specification.

A third version (not shown here) fixed this non-determinism problem and led to the generation of 684 test cases. Full structural coverage of the specification was achieved with this test suite.

Owing to the finite and fairly deterministic nature of the MAP specification, LOLA took between 10 and 18 seconds to validate the numerous tests associated with the different versions mentioned above. However, because of a very high number of test cases, almost 17 minutes (1000 seconds) were required by *OneExpand* to measure the structural coverage.

8.5.4 Discussion

The use of the structural coverage technique helped preventing the generation of a faulty conformance test suite from an incorrect specification constructed and validated manually. As shown by this MAP experiment, tests derived automatically from a LOTOS specification can be checked against this same specification in order to help detecting unreachable code and non-determinism. Such a self-coverage approach to conformance testing is an interesting by-product of the technique and provides added value to specifications and test suites at a very low price. It also shows that the coverage technique has its place even in the absence of validation test suites.

8.6 Test Suite Validation Using Mutation Analysis

In SPEC-VALUE, test suites generated from UCMs help validating LOTOS prototypes against those same UCMs and against functional requirements. The structural coverage approach helps ensuring the completeness of the test suite in terms of the events and basic behaviour expressions found in the specification. In a sense, this is one way to “validate” the validation test suite. However, the effectiveness of the test suite at finding faults is yet to be determined. So far, this chapter provided anecdotal evidence of the usefulness of these test suites through various applications of the methodology. This

section provides an additional experiment that uses *mutation analysis* to better measure the effectiveness of test suites at detecting faults in *mutants*, which are minor variations of the original specifications.

Section 8.6.1 introduces the concept of mutation in a validation context. The generation of LOTOS mutants in SPEC-VALUE is presented in Section 8.6.2, with an application to TTS. Mutation analysis is performed on TTS and three other specifications in Section 8.6.3, and the main observations and lessons are discussed in Section 8.6.4.

8.6.1 Mutation Analysis and Validation

Mutation analysis is a fault-based technique inspired by *mutation testing*, which was developed more than two decades ago for testing software in general [112]. Mutation testing is traditionally used to create test cases that are sensitive to small changes to the (syntactic) structure of a program, e.g. a modification to an operator, a condition, a variable, or a value. Such a change applied to a valid program results in a *mutant*, and tests are constructed in a way to detect (or *kill*) invalid mutants.

Mutation testing is based on two major assumptions:

- **Competent programmer hypothesis:** Competent programmers tend to write “nearly correct” programs, i.e. if they are not correct they will differ from the corrected version by a few relatively simple faults.
- **Coupling effect:** A test suite sensible enough to uncover all simple faults in a program is implicitly capable of detecting more complex faults. This assumption is more controversial in nature, but it has been justified on numerous occasions in empirical and experimental studies [271].

The mutation analysis used in this thesis aims mainly to measure the effectiveness of a test suite at killing mutant specifications rather than to generate test suites that can kill all mutants. Mutation analysis represents an opportunity to validate the various test suites constructed so far and to discuss the adequacy of the testing patterns and strategies defined in Chapter 6.

8.6.2 Mutant Generation and SPEC-VALUE

Mutation analysis is applicable to executable specifications just as it is applicable to software programs. The assumptions are adapted accordingly, i.e. “programmer” and “program” are substituted with “specifier” and “specification”. Notable examples include Probert and Guo [292] with a mutation testing technique for Estelle, Woodward [378] with a technique for algebraic specifications, and Amman and Black [11][52] with another technique based on temporal logic (CTL) and model checking. To our knowledge, mutation analysis has never been applied to LOTOS specifications.

There exist various strategies for the generation of mutants from programs and specifications. The basic strategy is called *strong mutation* and consists in generating all mutants for pre-determined types of operators and faults. Variations of this approach include *weak mutation* and *firm mutation* [231]. Unfortunately, these strategies lead to an explosion in the number of mutants for all but simple specifications. Since the testing of mutants is a computationally intensive task, minimizing the number of mutants becomes rapidly a necessity. *Selective mutation* was presented as a potential solution to this problem [272], but this technique is still rather synthetic and would produce numerous mutants for large specifications such as the ones studied in this chapter. Furthermore, many of the mutants produced with these techniques would be in violation of the strong static semantics of LOTOS and would be caught at compilation time. For the purpose of the current experiment, which is to observe the effectiveness of test suites on real-size specifications, and because the applicability of these mutant generation strategies to LOTOS specifications is beyond the scope of this thesis, a limited collection of mutants will be selected and generated manually.

Mutation Operators

The generation of mutants requires some specific constructs to be changed. *Mutation operators* are at the heart of mutation analysis as they identify the syntactic modifications responsible for mutant specifications. These operators should model plausible errors and faults while satisfying syntax and static semantics rules checked by compilers.

In the context of SPEC-VALUE, mutation operators should target the design decisions that need be taken during the construction of the specification from the UCMs. Several construction guidelines presented in Section 5.2 require many such decisions and are hence prone to the generation of erroneous specifications. Mutation operators should also relate to common mistakes in LOTOS specifications and to other details that are generally overlooked. In this experiment, six categories of mutation operators are used:

- **Structure:** Construction Guideline 5 requires decisions as to how the LOTOS processes capturing the components should be represented and connected. Gates may be incorrectly hidden or observable, and process synchronization (one fewer/more gate) and composition ($| | | \leftrightarrow []$) can be erroneous.
- **Segment integration:** Construction Guideline 6 suggests that unrelated path segments can be integrated in many ways. Consequently, the integration can be made more permissive (from sequential to alternative to concurrent) or restrictive (from concurrent to alternative to sequential).
- **Terminations:** Construction Guidelines 1 and 2 indicate that sequences of events resulting from linear causal paths can end with the inaction (**stop**) or with process instantiation (e.g. for recursion). AND-joins and OR-joins may also lead to the use of **exit**. Problems may result from the wrong type of termination (**stop** \leftrightarrow **exit** \leftrightarrow process instantiation).
- **Messages:** Construction Guideline 7 promotes the use of messages to refine inter-component causality. Errors may be introduced easily in these messages, e.g. incorrect or non-deterministic ordering, source/destination, type, values, etc.
- **Plug-in bindings:** Plug-ins and stubs are connected through binding relationships, as mentioned in Construction Guideline 3. These bindings span the process definitions of stubs and plug-ins, and errors may be introduced in input/output segments and parameters.
- **Selection policies:** Stub processes specify selection policies (Construction Guideline 3). These policies may be mutated by changing selected plug-ins or conditions.

The constructs cited in these categories can be mutated to simulate plausible errors occurring during the construction of the prototype specification. Note that the definition of ADTs (covered in Construction Guideline 8) is not covered here as it is not the focus of UCMs and validation test cases as used in SPEC-VALUE. Note also that simple construction guidelines (e.g. CG-2.a, CG-2.b, and CG-4) are not covered because they are seldom the subject of interpretation errors.

TTS Mutants

The mutation operators are applied to the TTS specification and 30 mutants are presented in Table 26. These 30 mutants were selected manually. Care was taken to prevent the generation of mutants that are syntactically incorrect or that do not satisfy LOTOS' static semantics. Such mutants would be declared incorrect during their compilation by tools such as LOLA.

Additionally, mutants that are *equivalent* to the original specification are also not selected. Equivalent specifications (according to testing equivalence in this case) would be undetectable from a testing perspective and would also be valid according to val. Common instances include equivalent conditions (e.g. $[n > 3]$ is the same as $[\text{not}(n \leq 3)]$) and behaviours (e.g. `a; stop ||| b; stop` is the same as `a; stop |[c]| b; stop`).

The limited number of mutants will be useful for our goal of providing general observations, but more mutants would be required for a full-fledge empirical study.

TABLE 26. TTS Mutants Generated Using Six Categories of Mutation Operators

	Name	Line	Mutation	Comment
Structure	m1	418	hide disp gate	Hiding of an extra responsibility
	m2	508	hide sig gate	Hiding of an extra responsibility
	m3	423	add ring in synchronization set	Decreased concurrency
	m4	440	remove disp from synchronization set	Increased concurrency
	m5	463	[] →	Increased concurrency
	m6	466	msg2 → msg	Erroneous communication channel
Seg. Integration	m7	527	[] →	More permissive integration
	m8	535	[] →	More permissive integration
	m9	535	[] → exit >>	Less permissive integration
	m10	543	[] →	More permissive integration
	m11	563	→ []	Less permissive integration
	m12	482	sequence → recursion followed by choice	More permissive integration
	m13	489	recursion followed by choice → sequence	Less permissive integration
Term.	m14	577	recursion → stop	Recursion removed
	m15	532	recursion → stop	Recursion removed
	m16	686	exit (...) → stop	Exit (and parameters) removed
	m17	756	exit (...) → stop	Exit (and parameters) removed
Msg.	m18	531	!msg → ?msg:Announcement	Increased non-determinism
	m19	568	!uid → ?user:User	Increased non-determinism
	m20	540	uid(ui) → userA	Fixed message parameter
Plug-in binding	m21	548	in2 → in1	Erroneous input segment (stub)
	m22	595	in1 → in2	Erroneous input segment (plug-in)
	m23	726	in1 → in2	Erroneous input segment (plug-in)
	m24	717	user0 → userA	Fixed plug-in parameter
	m25	717	user0 → userB	Fixed plug-in parameter
	m26	640	out3 → out2	Erroneous output segment (stub)
	m27	642	out4 → out2	Erroneous output segment (stub)
Sel. pol.	m28	609	OCS → CND	Erroneous condition
	m29	724	[CND NotIn fl(ui)] → [true]	Erroneous condition
	m30	717	CND → Default	Erroneous plug-in selection

Effectiveness and Mutation Scores

According to Weyuker and Ostrand's definition [370], the effectiveness of a test suite can be measured by computing *mutation scores*. Traditionally, a mutation score is defined as:

$$MS = \frac{M_{killed}}{M_{gen} - M_{eq}}$$

where M_{killed} is the number of mutants killed by the test suite, M_{gen} is the total number of mutants generated, and M_{eq} is the number of equivalent mutants generated. Equivalent mutants cannot be killed by any test case. If $MS < 1$, then the remaining *live* mutants require the addition or improvement of test cases in order to get killed.

In this experiment, equivalent mutants are prevented from being generated right from the start. However, some mutants might still be *appropriate*, even when not equivalent to the original specification. Appropriate mutants in our case represent alternate specifications that could have been generated from the same set of UCMs and that would be valid according to val. Unlike equivalent mutants, appropriate mutants could be distinguished by test cases. A common situation is a specification that allows for more behaviour, which is not forbidden by rejection test cases. A live mutant could be killed by the addition of a rejection test case, or be declared appropriate. It is up to the requirements engineers and designers to make such decision.

This more relaxed interpretation of mutation score lead to the following definition:

Definition 8.1: The *mutation score* (MS), where M_{app} is the number of appropriate mutants, is computed as follows: $MS = \frac{M_{killed}}{M_{gen} - M_{app}}$

This definition will be used to measure the effectiveness of test suites and to determine potential holes in test selection.

8.6.3 Application to Case Studies

The results of the mutation analysis of the TTS specification and its test suite are presented in Table 27. The Xs represent mutants that have been killed by individual test cases.

Tests 1 to 9 are those generated through the exclusive use of testing patterns whereas tests 10 to 14 are additional robustness test cases. In the *Killed* column, (Y) represents a mutant killed by robustness tests only. Mutants m6 is killed only by test 12 because this test targets mainly the messaging system used, which was not specified at the UCM level. Mutants m14, m15, and m20 were killed by tests that involve users A and B in roles other than originator and terminator respectively.

Overall, 23 mutants were killed by the test suite. Out of the 7 surviving ones, 5 are considered acceptable. Mutants m7, m8, m10 and m12 are specifications that have more permissive integration of path segments. These mutants support all the functionalities expected from the system, and more. The additional behaviour could be declared forbidden and rejection test cases could be constructed accordingly, but this does not seem necessary here. Mutant m21 is also acceptable as the input segment identifier, even if modified, is not checked later on because there is only one such segment in the stub. The use of this identifier could be removed altogether, but it is left there to improve traceability to the UCMs.

Two mutants are still considered unacceptable. The first (m19) sends a message non-deterministically and hence would require an appropriate rejection test case to get killed. The second (m24) has a variable that is incorrectly set to a fixed value. One of the acceptance test case (e.g. test 4) could use different users for the originating and terminating roles in order to detect this problem.

TABLE 27. Mutation Analysis of TTS and its Test Suite

	Mutant	TTS Test Cases														Killed	Acceptable mutant?
		1	2	3	4	5	6	7	8	9	10	11	12	13	14		
Structure	m1				X	X				X						Y	
	m2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	Y	
	m3	X	X		X	X	X			X		X				Y	
	m4				X	X				X						Y	
	m5	X	X		X	X	X			X		X	X			Y	
	m6												X			(Y)	
Seg. Integration	m7														N	Y	
	m8														N	Y	
	m9	X	X	X	X	X	X	X		X	X	X		X	Y		
	m10														N	Y	
	m11	X	X	X	X	X	X	X		X	X	X			Y		
	m12														N	Y	
	m13				X	X				X					Y		
Term.	m14									X	X	X			(Y)		
	m15													X	(Y)		
	m16	X	X		X	X	X			X	X				Y		
	m17								X				X	X	Y		
Msg.	m18								X				X	X	Y		
	m19														N	N	
	m20											X	X		(Y)		
Plug-in binding	m21														N	Y	
	m22						X	X	X	X			X	X	Y		
	m23	X	X				X				X				Y		
	m24														N	N	
	m25				X	X				X					Y		
	m26	X	X		X	X	X			X	X				Y		
	m27	X	X		X	X	X			X	X				Y		
Sel. pol.	m28								X				X	X	Y		
	m29				X	X				X					Y		
	m30				X	X				X					Y		
TOTAL:		9	9	3	14	14	10	4	5	15	4	10	7	6	7	23/30	23/25
Effectiveness		.36	.36	.12	.56	.56	.40	.16	.20	.60	.16	.40	.28	.24	.28	0.92 (i.e. 92%)	

The effectiveness of the TTS test suite is its mutation score: $MS = 23/(30-5) = 23/25 = 0.92$

Effectiveness of Testing Patterns

The testing patterns and strategies used in the selection of test cases from UCMs are summarized in Table 28. This information can be combined to Table 27 in order to provide additional observations in the context of the mutation analysis of TTS.

TABLE 28. Testing Pattern Strategies Used in TTS Test Cases

Testing Pattern	Strategy	TTS Test Cases								
		1	2	3	4	5	6	7	8	9
1. ALTERNATIVE	1.A—ALL RESULTS									
	1.B—ALL PATHS	X	X	X	X	X				X
	1.C—ALL PATH COMBINATIONS						X	X	X	
	1.D—ALL COMBINATIONS OF SUB-CONDITIONS									
2. CONCURRENT	2.A—ONE COMBINATION						X			X
	2.B—SOME COMBINATIONS				X	X				
	2.C—ALL COMBINATIONS	X	X							
3. LOOP	None applicable									
4. MULTIPLE START POINTS	None applicable									
5. SINGLE STUB	5.A—STATIC FLATTENING	X	X	X	X	X	X	X	X	X
	5.B—DYNAMIC FLATTENING, SOME PLUG-INS									
	5.C—DYNAMIC FLATTENING, ALL PLUG-INS									
6. CAUSALLY LINKED STUBS	6.A—DEFAULT BEHAVIOUR	X	X	X						
	6.B—INDIVIDUAL FEATURES				X	X	X	X	X	
	6.C—FEATURE COMBINATIONS									X

In Chapter 6, Strategy 6C was suspected to lead to high-yield test cases. This is supported here by the number of mutants killed by test 9, which has the highest effectiveness (60%).

Also observe that the tests involving concurrent segments (Testing Pattern 2) have, on average, a much higher effectiveness (47%) than those without concurrent segments (16%). As for causally linked stubs (Testing Pattern 6), tests targeting the default behaviour have an effectiveness (28%) lower than those targeting individual features (38%) or combinations of features (60%).

Mutation Analysis of Three Other Specifications

Results of the mutation analysis of the GCS, PTM-G, and FI specifications are reported in Table 29. The SBC specification was not used because its test suite is too incomplete, and the MAP specification and test cases were not generated from UCMs so they cannot be used either. Mutants were generated manually for each of the mutation operator categories, except for GCS and PTM-G whose UCMs do not use stubs and plug-ins. Two segment integration mutants remained alive but were declared appropriate: one for GCS and another one for PTM-G. The effectiveness of each test suite is presented at the bottom of the table.

TABLE 29. Mutation Analysis of GCS, PTM-G, and FI

Mutation Operator Category	GCS	PTM-G	FI	Total
Structure (3 mutants each)	3	3	3	9 / 9
Segment Integration (3 mutants each)	2 (-1)	2 (-1)	2	6 / (9 - 2)
Termination (3 mutants each)	2	2	2	6 / 9
Messages (3 mutants each)	2	3	3	8 / 9
Plug-in Bindings (3 mutants for FI only)	-	-	1	1 / 3
Selection Policies (3 mutants for FI only)	-	-	3	3 / 3
Total	9 / (12 - 1)	10 / (12 - 1)	14 / 18	33 / (42 - 2)
Effectiveness (Definition 8.1)	82%	91%	78%	83%

8.6.4 Discussion

Mutation analysis enabled us to take a hard look at the test suites generated using SPEC-VALUE. Overall, the effectiveness of the test selection and generation strategies are rather good given that mutation coverage was not a priority and that the mutants generated here are of a fair quality and representative of plausible problems. Nevertheless, this experiment resulted in interesting lessons:

- Errors in a specification may be undiscovered by a test suite generated from UCMs.
- The use of testing patterns and rejection testing strategies is not sufficient to ensure validity. Robustness test cases have shown their usefulness in detecting additional errors.

- There may be many appropriate specifications for a given set of UCMs, even if they are not testing equivalent.
- Mutation analysis of four specifications suggest the use of longer test cases for checking terminations, of additional rejection test cases for killing unacceptable mutants with behaviour that should be forbidden (due to concurrency level or non-determinism), and of variations in the data values used in the test cases.
- It is still premature to suggest general conclusions regarding the selection strategies that lead to the highest effectiveness. Such conclusions may not even be possible at all. However, this experiment supports the idea that a test goal that covers more path segments will generally be more effective, but obviously at a higher cost.

8.7 Chapter Summary

The characteristics of the different UCMs, specifications, test suites, and coverage results discussed in this chapter are all summarized in Table 30. The TTS case study is also included in this summary. Several results and lessons related to these four main parts of SPEC-VALUE (Use Case Maps—Chapter 4, LOTOS prototypes—Chapter 5, test suites—Chapter 6, and structural coverage—Chapter 7) are recalled below.

Note that Table 30 follows the general structure used in this Chapter. Columns represent the sections (GCS, PTM-G, FI, SBC, and MAP), and groups of rows refer to the sub-sections (UCMs, LOTOS, tests, and coverage). To give a better idea of the testing complexity involved, rows n and o indicate the maximum numbers of states and transitions resulting from the composition of a single test and the specification (taking into account LOLA's graph minimization based on testing equivalence). This measure is more useful than the plain size of a specification's LTS, which is often infinite in our approach. Rows u and v , which are not discussed in the previous sections, will be explained later. The legend for row x respects the interpretation of coverage results discussed in Section 7.4.3:

- ❶ Unreachable code or error in the LOTOS specification.
- ❷ Incomplete test suite.

- ③ Discrepancies between the LOTOS specification and the test suite or the UCMs from which these tests were derived.

TABLE 30. Summary of Experiments with SPEC-VALUE

	System	GCS	PTM-G	FI	SBC	MAP	TTS
UCMs	a) # Root UCMs	12	9	2	4	-	1
	b) # Plug-in UCMs	0	0	23	0	-	4
	c) # UCM components	12	15	5	7	-	6
LOTOS	d) # Process definitions	19	30	13	9	14	11
	e) # Lines of behaviour	750	1400	800	750	850	375
	f) # Abstract data types	29	53	39	8	22	19
	g) # Lines of ADTs	800	1125	750	200	375	400
	h) # Lines of tests	1600	800	1325	300	7725	375
	i) Total number of lines	3150	3325	2875	1250	8950	1050
Tests	j) # Acceptance functional tests	56	35	37	4	603	14
	k) # Rejection functional tests	51	1	0	2	0	14
	l) # Other tests (e.g. robustness)	2	0	0	5	0	5
	m) # Unexpected verdicts	0	0	1	3	6	0
	n) Max # states by one test	124	2961	850	4277	21	393
	o) Max # transitions by one test	142	3248	1019	4278	21	519
	p) Time to compile & test (sec.)	5	120	11	64	16	5
Coverage	q) # LOTOS events	57	126	94	204	156	25
	r) # LOTOS single BBEs	35	86	27	20	46	22
	s) # Sequences	40	74	49	60	67	18
	t) # Probes inserted	54	99	55	64	83	26
	u) Optimizations reduction	28%	38%	28%	20%	27%	35%
	v) Overall reduction	41%	53%	55%	71%	59%	47%
	w) # Missed probes	3	11	4	17	17	0
	x) Reasons for missed probes	③	①, ③	③	②, ③	①	-
	y) Time to measure, <i>TestExpand</i>	235	-	165	-	-	140
	z) Time to measure, <i>OneExpand</i>	31	81	37	18	1000	9

Use Case Maps

The UCM style and content guidelines (Section 4.2.2) were developed at the same time as the GCS, PTM-G, and FI experiments. They were used on the UCMs for TTS and SBC and they improved the understandability of the scenarios as well as the traceability between UCMs and other models (e.g.

LOTOS, execution traces, and message sequence charts). They also proved to be compatible with the construction guidelines of Section 5.2.

UCMs were useful to evaluate architectural alternatives in numerous occasions, especially in the GCS and PTM-G experiments.

The scenarios for GCS and PTM-G were not integrated at the UCM level, because the stub/plugin concept was not well established at the time these experiments were done. However, as suggested by Section 4.2.3 and as shown by the FI and TTS experiments, integrating UCMs through stubs and plug-ins offers interesting benefits:

- UCM behaviour patterns can be extracted as plug-ins and made more consistent and reusable across different stubs in a design and across designs. End-to-end individual scenarios proved to be reusable across designs in the GCS and PTM-G experiments, but using stubs and plug-ins would have improved this reusability even more.
- Integration helps making decisions that can avoid undesirable interactions among different scenarios and features. Inspection for interactions can be localized to stubs and selection policies early in the development cycle.
- Stubs and plug-ins help evaluating the impact of the incremental addition of new scenarios and features to an existing system.

The SBC UCMs from Section 8.4.1 are not integrated, but an improved version of this system description, where UCMs are integrated and several features are included, also lead to similar conclusions for the use of stubs and plug-ins [25].

LOTOS Prototypes

Most of the construction guidelines introduced in Section 5.2 were applied to one or many of these systems, as shown in Table 31.

TABLE 31. Construction Guidelines Usage

Construction Guideline		GCS	PTM-G	FI	SBC	TTS		
Paths	1. Interaction points and responsibilities	CG-1	X	X	X	X	X	
		CG-1.a	X	X	X		X	
	2. Causal paths	CG-2	X	X	X	X	X	
		CG-2.a	X	X	X	X	X	
		CG-2.b	X	X	X	X	X	
		CG-2.c	X	X	X		X	
		CG-2.d	X	X	X			
	3. Stubs and plug-ins	CG-3			X		X	
		CG-3.a			X		X	
		CG-3.b			X		X	
		CG-3.c			X		X	
	4. Other path elements	CG-4.a				X		
		CG-4.b						
		CG-4.c						
		CG-4.d	X	X				
	Structure	5. Structure	CG-5	X	X	X	X	X
CG-5.a				X	X	X	X	
CG-5.b			X	X		X		
CG-5.c			X	X	X	X	X	
CG-5.d			X	X			X	
6. Unrelated path segments		CG-6	X	X	X	X	X	
		CG-6.a	X	X				
		CG-6.b		X	X	X	X	
		CG-6.c	X	X	X	X	X	
		CG-6.d	X	X	X	X	X	
7. Inter-component causality		CG-7	X	X	X	X	X	
		CG-7.a			X			
		CG-7.b	X	X		X	X	
		CG-7.c	X	X	X	X	X	
Data		8. Data	CG-8	X	X	X	X	X

Specification of path behaviour in component processes appears simpler when UCMs are structured with stubs and plug-ins because a large part of the scenario integration is then performed at the UCM level. A direct mapping from stubs and plug-ins to LOTOS processes modularizes the inte-

gration of path behaviour. When the burden of the integration is pushed down to the level of LOTOS, designers not familiar with this language may have a hard time coping with the construction of the specification. Moreover, other people not involved in the LOTOS part (clients, marketing, management, etc.) would not know anything about how the individual UCMs fit together. For these two reasons, although the GCS and GPRS experiments were successful in the sense that moving to LOTOS directly also resulted in correct specifications and validated test suites, integrating the scenarios at the UCM level seems a better alternative.

UCMs ease architectural reasoning and the discovery of appropriate structures. However, new system components may be also discovered at the LOTOS level. For instance, the FI experiment showed that call session objects are required in the LOTOS prototype and should probably be included in the UCMs as well. These component corresponds to the Call Objects in the SBC experiment.

Deviations from the original scenarios at LOTOS level may be captured by validation and coverage measurements, but it is better when they are reported directly by specifiers to UCM designers for an appropriate update of the UCMs.

Test Suites

Validation test suites were helpful in finding ambiguities, errors and undesirable interactions in the different specifications, UCMs, and informal requirements or standards involved in these experiments. The most important points are:

- Testing patterns help covering UCM paths in a cost-effective way. UCMs already abstract from many details, so it is important to cover all their paths with the test cases. When strategies other than testing patterns are used (e.g. for SBC and FI), then additional effort is required to ensure the coverage of UCM paths. Most of the testing patterns have been used in the case studies (see Table 32). Testing pattern 3 has not been exercised on these case studies because their UCMs do not contain any loop. This pattern is present in our pattern language in order to improve the coverage of the UCM notation.

TABLE 32. Testing Patterns Usage

Testing Pattern	Strategy	GCS	PTM-G	FI	SBC	TTS
1. ALTERNATIVE	1.A—ALL RESULTS	X	X		X	
	1.B—ALL PATHS	X	X	X		X
	1.C—ALL PATH COMBINATIONS			X		X
	1.D—ALL COMBINATIONS OF SUB-CONDITIONS	X	X			
2. CONCURRENT	2.A—ONE COMBINATION	X	X	X	X	X
	2.B—SOME COMBINATIONS	X		X		X
	2.C—ALL COMBINATIONS					X
3. LOOP	3.A—ALL SEGMENTS					
	3.B—AT MOST <i>k</i> ITERATIONS					
	3.C—VALID BOUNDARIES					
	3.D—ALL BOUNDARIES					
4. MULTIPLE START POINTS	4.A—ONE NECESSARY SUBSET, ONE GOAL	X	X		X	
	4.B—ALL NECESSARY SUBSETS, ONE GOAL					
	4.C—ALL NECESSARY SUBSETS, ALL GOALS					
	4.D—ONE REDUNDANT SUBSET, ONE GOAL					
	4.E—ALL REDUNDANT SUBSETS, ONE GOAL					
	4.F—ONE INSUFFICIENT SUBSET, ONE GOAL					
	4.G—ALL INSUFFICIENT SUBSETS, ONE GOAL					
	4.H—SOME RACING SUBSETS, SOME GOALS				X	
5. SINGLE STUB	5.A—STATIC FLATTENING			X		X
	5.B—DYNAMIC FLATTENING, SOME PLUG-INS			X		
	5.C—DYNAMIC FLATTENING, ALL PLUG-INS					
6. CAUSALLY LINKED STUBS	6.A—DEFAULT BEHAVIOUR			X		X
	6.B—INDIVIDUAL FEATURES			X		X
	6.C—FEATURE COMBINATIONS			X		X

- Test suites based on acceptance and rejection test cases proved to be practical and helpful in eliminating problems in the prototype (GCS and TTS). Acceptance tests with verification steps are also very useful for checking robust specifications (PTM-G) or expected interactions among features (FI).
- Test suites do not have to be composed of sequential test cases. The FI experiment showed that they can be structured to promote consistency and reuse of partial sequences (test steps).

- For telecommunications systems with multiple features, a test suite should contain test cases for the basic system without features, for the basic system with features taken individually, and for the basic system with multiple features (usually pairs). In the last category, the number of possible configurations can grow very quickly, and appropriate methods need to be used to filter out the configurations that are unlikely to highlight a problem. The FI experiment used a very simple one, but other approaches are being developed [268].
- Mutation analysis demonstrated that errors may still remain in validated specifications, and that various non-equivalent but appropriate specifications may result from the application of the construction guidelines. Also, the effectiveness of test cases seems to be linked to the length of test goals rather than to the use of a particular testing pattern.

Structural Coverage

Ensuring the structural coverage can improve the quality and consistency of both the specification and the tests, hence resulting in a higher degree of confidence in the system's description. The experiments presented in this chapter demonstrate that coverage results can help detecting unreachable parts or errors in specifications (❶ — PTM-G and MAP), incomplete test suites (❷ — SBC), and discrepancies between specifications and their tests (❸ — GCS, PTM-G, FI and SBC).

Results can also be output quickly and at low cost. When complex specifications are involved, the structural coverage can be measured compositionally (PTM-G and MAP). Probes can be covered independently, even one at a time, through multiple executions of the test suite. The LOT2PROBE filter allows different variations of the probe comment in the specification (e.g. `(*_PROBE_A_*)`, `(*_PROBE_B_*)`, etc.), which represent different groups of probes. Having fewer probes inserted at once reduces the number of internal actions and helps avoiding the state explosion problem. Another alternative is to use LOLA's *OneExpand* command, which provides faster results than *TestExpand* but with a potentially incomplete coverage.

The reduction in the number of probes also helps coping with complexity. By looking at the number of events, simple basic behaviour expressions and sequences, it is possible to evaluate the reduction achieved by the probe insertion strategy presented in Section 7.4. Row u in Table 30 shows the reduction achieved by the improved probe insertion strategy over the number of sequences and single BBEs ($u = (r+s-t)/(r+s)$) whereas row v shows the overall reduction over the number of events and single BBEs ($v = (q+r-t)/(q+r)$). Considering all the experiments (662 events, 236 BBEs, 308 sequences, and 381 probes), the reduction amounts roughly to 30% (sequences and BBEs) and 58% (events and BBEs).

This technique is valuable for scenario-based approaches such as SPEC-VALUE and also for checking, through self-coverage, the quality of conformance test suites generated directly from LOTOS specifications (MAP experiment).

Contributions

The following items are original contributions of this chapter:

- Illustration of Contribution 3 (Section 1.4.3) regarding the validation of the SPEC-VALUE methodology and main constituents (UCM style and content guidelines, prototype construction guidelines, testing patterns, testing theory, and structural coverage) through the study of two hypothetical communicating systems (GCS and FI), a draft standard (PTM-G) and an industrial system (SBC).
- Application of SPEC-VALUE to different telecommunications areas such as network servers, mobile communication, switch-based telephony, and agent-based telephony.
- Additional study of structural coverage technique and tools to the self-coverage of conformance test suites generated automatically from LOTOS specifications (MAP).
- Development of mutation analysis and definition of mutation operators for LOTOS, and application to the validation and effectiveness measurement of validation test suites.
- Development of a novel framework for the avoidance and the detection of undesirable feature interactions.

CHAPTER 9

Conclusions and Future Work

校塔に
鳩多き日や
卒業す

*Nuée de colombes
Au-dessus du lycée;
C'est la fin des classes...*

(Koutou Ni Hato Ooki Hi Ya Sotugyou Su)

*Japanese Haiku by Kusatao Nakamura,
Haijin poet.*

In this chapter, we first review the main contributions of the thesis and relate them to the research hypothesis, to the expected contributions, and to the Formal Specifications Maturity Model (Section 9.1). In Section 9.2, we briefly compare SPEC-VALUE to related methodologies and we provide insights on how it can be integrated to design processes with a wider scope. Finally, we discuss several research issues in Section 9.3.

9.1 Hypothesis and Contributions

9.1.1 Validation of the Research Hypothesis

This thesis presents SPEC-VALUE, which is a methodology for the specification and early validation of telecommunications systems. Requirements are captured with Use Case Maps, which visually describe causal scenarios bound to component structures. The research hypothesis (Section 1.2) is:

In the process of designing complex telecommunications systems, requirements described using the Use Case Map causal scenario notation can guide the generation of LOTOS specifications useful for validating high-level designs systematically through numerous techniques, including functional testing based on UCMs.

Chapter 5 shows that UCMs can guide the construction of high-level designs and prototypes in the form of formal LOTOS specifications. The availability of a LOTOS specification enables the application of numerous validation and verification techniques, many of which were introduced in Section 2.3.7 and in Section 3.4.3. In particular, Chapter 6 describes how functional test cases derived from UCMs help to gain confidence in the conformance of a LOTOS prototype to the UCMs it was generated from. UCM scenarios and their corresponding tests are more likely to be faithful and correct representations of the requirements than a specification that integrates all the requirements into a more detailed and complex description based on components. Hence, testing the LOTOS specification against the UCMs becomes an indirect means of validating the high-level design against the (possibly informal) requirements.

This research hypothesis was validated through the theoretical framework supporting SPEC-VALUE and through the successful application of this methodology to the specification and validation of telecommunications systems of various complexities and nature (Chapter 8). Many problems, related to these systems' functional requirements and to their integration in high-level designs, were uncovered while creating the specifications and test suites, and when checking the latter against the former. In Chapter 6, we also suggested that the UCM-based testing technique be supplemented by other forms of validation, for instance robustness testing based on requirements expressed in a form other than UCMs. Being scenarios, UCMs are necessarily incomplete and only provide a partial view of the overall set of requirements. Nevertheless, UCMs represent a very important piece of this set to which users and other stakeholders can relate, hence their usefulness for validation.

9.1.2 Contributions of the Thesis

Three major contributions were announced in Section 1.4. This section details how this thesis fulfills these expected contributions.

Contribution 1: SPEC-VALUE Methodology

SPEC-VALUE has several benefits, difficult to find all at once in other design and standardization methodologies (Chapter 4):

- **Separation of functionalities from the underlying structure:** this is a characteristic of UCMs that can be reflected in the prototype specification as well since it is possible, in LOTOS, to specify behaviour with or without components. This separation of concerns enables designers and requirements engineers first to focus on the desired functionalities, and then to map them to a suitable underlying structure. Architectural reasoning at a high level of abstraction hence becomes possible. This separation also improves the reusability of scenarios across architectures and across different versions of an evolving architecture as scenarios and components are modified or added incrementally.
- **Fast prototyping:** formal prototypes can be generated in LOTOS from UCMs (with or without components). LOTOS already supports most UCM constructs, and the construction guidelines provided in this thesis bring experience and guidance in the actualization of the UCM-to-LOTOS mapping at a system level. Formal prototyping adds rigor and executability to scenario-based design with UCMs, the latter being semiformal and non-executable.
- **Test case generation:** UCM scenario paths are used to guide, through the application of testing patterns and complementary strategies, the selection of test goals used for the generation of abstract functional test cases. This enables the verification of the prototype against the UCMs and its validation against the informal functional requirements. The test suite can itself be validated using structural coverage criteria on the model, or using mutation analysis. It can be reused as a basis for functional or regression test suite in the subsequent steps of the development process.
- **Design documentation:** the documentation of requirements and designs is done as we go along the development cycle. UCMs focus on system-level functionalities and can be used as a common language between various stakeholders, from marketing people to system architects, designers, and testers. The LOTOS specification and the abstract test cases also provide useful and traceable views of the system design and should be part of its documentation.

Contribution 2: Theories and Techniques Supporting SPEC-VALUE

In order to support the SPEC-VALUE methodology and complement existing theories and tools for the testing of LOTOS specifications (step ⑥ in Figure 2) and for the visual editing of UCMs (steps ①, ②, and ③ in Figure 2), several theories and techniques were developed in this thesis. They were all illustrated with an ongoing example, the Tiny Telephone System (TTS).

- **Guidelines for the construction of LOTOS specifications from UCMs:** Chapter 5 provides a total of 8 general construction guidelines and 22 small-grained guidelines, which relate to step ④ in Figure 2.
- **UCM-LOTOS testing framework:** Chapter 6 defines a testing framework based on UCMs and LOTOS in order to support step ⑤ in Figure 2. The validation relation val is defined and compared to the conventional conformance relation conf. The framework contains UCM-oriented testing pattern language that explains how 25 coverage-driven strategies regrouped under six testing patterns can collaborate to select test goals from systems specified with UCMs. The framework also includes motivations and strategies for generating rejection test cases in LOTOS.
- **Structural coverage for LOTOS:** In relation with step ⑦ in Figure 2, Chapter 7 presents a new technique for automatically measuring the structural coverage of LOTOS specifications by a test suite. This technique includes a theory for the insertion of probes in LOTOS specifications and for coverage measurement together with partial tool support.

Contribution 3: Illustrative Experiments Validating SPEC-VALUE

Chapter 8 validates the SPEC-VALUE methodology and its supporting techniques against five telecommunications systems of various complexity. These experiments include hypothetical communicating systems (GCS and FI), a draft standard (PTM-G) and an industrial system (SBC). They cover a wide range of areas including network servers, mobile communication, group communication, switch-based telephony, and agent-based telephony. The structural coverage technique and tools are also applied to a context different from validation, i.e. to the self-coverage of conformance test suites

generated automatically from LOTOS specifications (MAP example). Most of these experiments were done in collaboration with industrial partners, professors, and other students. The specification of these systems with UCMs and LOTOS improved the understanding of these systems, while their validation improved their quality and the overall confidence in their description. The effectiveness of test cases is further studied through mutation analysis, adapted to LOTOS for the purpose of this experiment.

Additional Contributions

The development of SPEC-VALUE required the study of many different domains, which resulted in additional contributions of this thesis:

- Quick tutorial on the Use Case Maps notation (Section 2.2).
- Quick tutorial on the formal description technique LOTOS (Section 2.3).
- Evaluation of six specification techniques against thirteen comparison criteria (Section 3.2).
- Evaluation of thirteen scenario notations against eight comparison criteria (Section 3.3).
- Survey and brief comparison of twenty analytic and synthetic construction approaches (Section 3.3.4).
- Argument supporting that UCMs and LOTOS are compatible and complementary notations (Chapters 3 and 4, summarized in Section 4.1 and in Table 1).
- Style, content, and integration guidelines for Use Case Maps (Section 4.2).
- Development of a framework for the avoidance and the detection of undesirable feature interactions. (Section 8.3).
- Definition of mutation operators for LOTOS specifications constructed with SPEC-VALUE (Section 8.6.2).

9.1.3 SPEC-VALUE and the Formal Specifications Maturity Model

In addition to the research hypothesis, Section 1.2 suggests that SPEC-VALUE could improve the maturity of design processes based on formal specifications. To support this claim, our methodology can be measured against the *Formal Specifications Maturity model* (or *FSM* model), based on the CMM and defined by Fraser and Vaishnavi [138]. This FSM model, summarized in Table 33, is composed of five levels of maturity, the first one being the lowest level. Each level possesses *generic strategies* describing typical operations performed to develop formal specifications [137]. According to this model, the sole use of formal specification languages does not guarantee maturity levels beyond the first one (*initial*).

The need for semiformal specifications and notations, which is at the basis of levels 2 and 3 in the FSM model, is motivated among other things by the following two points. First, formal methods have an implicit assumption: they are useful for detecting detailed problems when the model is close to being complete and correct. Hence, this requires spadework at another level, closer to the human way of thinking. Second, as suggested in the recommendations made by Craigen *et al.* [104], there is a clear need for improved integration of formal methods techniques with other software engineering practices, and emphasis should be put on developing notations more suitable for use by individuals not expert in formal methods or mathematical logic. Semiformal notations like UCMs can be quite useful in this context, and they could potentially broaden the accessibility to formal methods and lead to their wider acceptance by the software community [105][277].

SPEC-VALUE proposes the introduction of a semiformal description (UCM) between informal requirements and design-oriented formal specifications (LOTOS). The methodology also supports stepwise refinement and incremental construction of specifications. However, this construction is still manual; it is supported by guidelines rather than by synthesis tools. These characteristics fit the “Transitional-Unassisted” generic strategy described in Table 33. Therefore, SPEC-VALUE can improve design processes based on formal specifications by moving them from an **initial** level of maturity (level 1) to a **repeatable** level of maturity (level 2) on the FSM scale. This represents another contribution of the methodology.

TABLE 33. Summary of the Formal Specifications Maturity (FSM) Model (Extracted from [138])

<i>Level</i>	<i>Generic Strategies in FSM</i>
1- Initial <i>analyst dependent</i>	Direct-Unassisted: <ul style="list-style-type: none"> • Direct formalization process relying on problem elicitation, structuring, and requirements specification skills of the requirements engineer. • Requirements engineer has thorough knowledge of the application domain. • Requirements engineer can grasp and formalize the whole problem in its entirety.
2- Repeatable <i>process management</i>	Transitional-Unassisted: <ul style="list-style-type: none"> • One or more semiformal specifications provide mediating increments or formality between informal natural language specifications (i.e. informal requirements) and formal specifications. • Translation from semiformal to formal specifications performed by the requirements engineer without computer assistance. • Can guide stepwise refinement of formal specifications.
3- Defined <i>process defined with computerized rules</i>	Transitional-Assisted: <ul style="list-style-type: none"> • Use of semiformal specifications to mediate between informal natural language specifications and formal specifications. • Computer assistance is available to move between semiformal and formal specifications to replace human labor and to reduce errors in writing formal specifications.
4- Managed <i>incorporate measure into process by computerization</i>	Direct-Assisted: <ul style="list-style-type: none"> • Relies on computer-based support to develop formal specifications directly from informal natural language specifications or requirements. • Computer assistance via knowledge-based support for eliciting, discovering, and creating formal specifications.
5- Optimizing <i>continuous improvement and optimization</i>	Advanced-Direct-Assisted: <ul style="list-style-type: none"> • Advanced version of direct-assisted strategy that uses domain-dependent analogies. • A learning system with capability to maintain and increase a rich repository of analogies and to provide commonly occurring operations for which schemas are available at high syntactic levels. • Capable of use on increasingly complex systems.

9.2 SPEC-VALUE and Related Methodologies

This section briefly compares SPEC-VALUE to related methodologies. It also provides additional insights on how SPEC-VALUE can be integrated to more general methodologies with a wider scope.

9.2.1 Comparing SPEC-VALUE to Related Methodologies

The following methodologies, many of which are introduced in Chapter 3, are compared to SPEC-VALUE because they support the design of telecommunications or reactive systems with scenarios and/or formal methods.

Timethreads-LOTOS

SPEC-VALUE builds on previous work on the formalization of *Timethreads* (an earlier version of UCMs) in LOTOS. The Timethreads-LOTOS technique addresses the specification of UCM paths only [12][13][14], and the system end-to-end functionalities are not allocated to component structures, hence resulting specifications are simpler and less complete. Issues related to the implementation of causal relationships across components are not addressed. SPEC-VALUE addresses these issues, and it also provides a comprehensive testing framework for validation. Additional UCM constructs, such as stubs and plug-ins, are also supported by SPEC-VALUE.

LOTOSphere

The LOTOSphere project [57][302] was an international effort to formulate an integrated methodology for the development of communications software. LOTOSphere based itself on LOTOS and correctness-preserving algebraic transformations. Requirements were to be written in LOTOS constraint-oriented style, which subsequently would be transformed into state-oriented and resource-oriented styles for implementation [364]. Other transformations of the specifications would provide test cases. The obvious advantage of this approach is that one could hope that these transformations would be implemented in tools, thus guaranteeing conformance throughout the whole process. Unfortunately, it turned out that constraint-oriented specifications are difficult to write, especially because most designers seem to be more used to think in terms of scenarios than in terms of constraints. The algebraic transformation approach could not be practically applied to realistic specifications, and algebraic transformation tools were not made available. We believe that SPEC-VALUE, even if limited to the first stages of development cycles, has a better chance to be applied in an industrial environment.

SDL and MSCs

This is one of the most popular approaches, for which industrial support exists and powerful tools are available [20]. MSCs represent scenarios visually in terms of sequences of messages exchanged between system components. They guide the generation of component behaviour, which can be formally specified and validated with SDL, and they can also be used to test SDL models. Multiple methodologies based on the combined use of SDL and MSCs (or similar types of scenarios) exist, and many were described in Chapter 3 (e.g. Eberlein's *RATS service development methodology* [120], Regnell's *Usage Oriented Requirements Engineering* [304], Mansurov and Zhukov's [255], Khendek and Vincent's [228], Probert *et al.*'s [297][298], and Dulz's [118]). Emerging tools for the synthesis of SDL models from MSCs have started to appear and they promote this approach to the third maturity level on the FSM scale. However, components and messages need to be identified early in the design process for both MSCs and SDL. SDL also requires the definition of states for the components. Premature decisions often need to be taken on the sole basis of informal requirements. Moreover, the system view of functionalities and causal relationships between activities tends to be hidden behind clouds of details, especially as the scale of the system increases. Sales *et al.* have started working on the transformation of UCMs to SDL in order to address several of these issues [262][318][317].

Unified Modeling Language

UML includes sophisticated diagram notations [274], among which we find *use case diagrams* [212], which show relationships among prose descriptions of behaviour. These diagrams present the interactions between actors and the system in a black-box fashion, and they do not help much in visualizing system behaviour or causality relationships. *Sequence diagrams* (similar to MSCs) and *statechart diagrams* do focus on behaviour, but at a detailed design level that include objects, states, and messages. Unfortunately, these component-based diagrams suffer from problems similar to those of SDL and MSCs. *Activity diagrams* can illustrate causality between events, but they are usually unrelated to components (*swimlanes* provide functional grouping only), and their sub-diagrams are less flexible and less powerful than UCM stubs [27]. UML, as used in the *Rational Unified Process (RUP)* [303], is a modelling notation and is not intrinsically concerned with formal validation of prototypes and

generation of test suites, whereas SPEC-VALUE focuses on these two aspects. *UML-RT*, a variant of UML adapted to real-time systems, enables validation through limited testing. Requirements MSCs are then compared to MSCs generated from a statechart model describing component behaviour. Telelogic *Tau/UML* improves validation for UML designs via a conversion to SDL. Unfortunately, these two tool-supported methodologies do not really enforce or even suggest the use of activity diagrams to capture requirements scenarios.

RT-TROOP

The *Real-Time TRaceable Object-Oriented Process* (RT-TROOP), defined by Bordeleau [61][62], uses scenario textual descriptions (use cases), UCMs, (H)MSCs, and ROOM (UML-RT). In this approach, UCM scenarios (extracted from use cases) are first transformed into HMSCs, refined by MSCs, and then transformed into hierarchical CFSMs. Patterns, which are fewer in number but which are more elaborated than SPEC-VALUE's construction guidelines, provide partial guidance for these transformations. Traceability relationships are also defined in this process. RT-TROOP focuses more on design than on requirements validation because verification of the ROOM/UML-RT model is limited. Unlike SPEC-VALUE, RT-TROOP does not really develop any validation or testing strategies.

Rigorous Object-Oriented Analysis

In their *Rigorous Object-Oriented Analysis* (ROOA) method [263], Moreira and Clark integrate formal techniques with object-oriented analysis methods (i.e. OMT [314]) in order to generate executable prototypes (in LOTOS) and validate them against the requirements. ROOA considers the system as a set of concurrent objects, modeled by LOTOS processes. The method starts by identifying the object model, and then scenarios are created to model the system dynamics. Scenarios are, again, sequences of interactions between the objects, just like the SDL/MSC approach. Validation is done through simulation, testing, and symbolic execution. However, regrouping OMT classes into system components for the specification (i.e. going from a static model to a more dynamic and functional one) remains difficult. Unlike SPEC-VALUE, ROOA does not really develop any strategy for the validation or the derivation of test cases.

OMT and LOTOS

Wang and Cheng [365][366][367] propose an approach where requirements are transformed into a LOTOS formal specification through the semiformal graphical modelling notation OMT [314]. OMT offers three types of models, which are integrated during the formalization. Class diagrams (static object model) are mapped to abstract data types. Statechart diagrams (dynamic model) have their states transformed into processes and their transitions into process instantiations. Dataflow diagrams (functional model), which share some characteristics with UCMs but with a heavy focus on data, have their data objects mapped to sorts and operations, while their services are translated to gates with experiments. Similar to SPEC-VALUE, formalization guidelines are provided to support these transformations. However, SPEC-VALUE produces LOTOS specifications earlier in the development cycle than this OMT-LOTOS approach. Abstract causal scenarios and (optionally) component structures are all that is required, whereas OMT uses class diagrams and definitions of object behaviour based on state machines, which are at a lower and more detailed level of abstraction and demand much commitment. Also, the OMT-LOTOS approach does not use scenarios. As for validation, this approach uses static checking of ADTs provided by LOTOS and its tools (like SPEC-VALUE), together with additional checking of algebraic assertions on these ADTs via Larch tools [163]. It is suggested that the dynamic behaviour be validated with LOTOS testing and tools such as LOLA. However, this validation framework provides no guidance on what should be tested to get a high confidence in the conformance of the LOTOS model to the OMT model, in the correctness of these two models, and in their validity with respect to the requirements. The UCM-LOTOS testing framework in SPEC-VALUE addresses such issues.

CRESS and ANISE

Both CRESS (*Chisel Representation Employing Structured Specifications*) [354] and ANISE (*Architectural Notions In Service Engineering*) [355], developed by Turner, use LOTOS as the underlying language for the formal description and validation of telecommunication features [356][357]. Whereas CRESS uses a formalized version of the Chisel scenario notation to describe features [4], ANISE consists in an architectural language that includes collaboration patterns and feature combination opera-

tors tailored for telecommunications systems. Both languages support the static detection of feature interactions and can be translated automatically to LOTOS for dynamic detection and validation. ANISE also hides LOTOS to a large extent as test cases are described with a companion language (ANTEST) using the vocabulary found in the feature description language. Errors are also reported in ANTEST terms, which improves overall understandability. Both approaches however suffer from a lack of strategy for the derivation of validation test cases and from several limitations in how features can be composed. For instance, ANISE descriptions can only be used to modify a base system, not another feature. Given this limitation, a feature like Call Display Blocking cannot be described in terms of modifications to a Call Display feature. SPEC-VALUE is not as automated as these techniques and does not support static verification mechanisms yet, but it enables features to be composed in more ways, and it includes validation test selection strategies (testing patterns) together with coverage metrics at the LOTOS level. ANISE is also a textual language and is less appealing than a visual scenario notation like UCMs.

9.2.2 Integrating SPEC-VALUE to Related Methodologies

SPEC-VALUE is meant to be integrated to other methodologies rather than replacing them. Many existing and forthcoming methodologies have a scope much wider than what is addressed by SPEC-VALUE. However, when the capture, specification, and validation of requirements and high-level designs becomes a concern, these other methodologies could potentially benefit from integrating SPEC-VALUE. For instance, such integration can be envisaged for the UML-based methodologies, for standardization methodologies, and for test-oriented methodologies such as SPEC-TO-TEST.

UML-Based Methodologies

UML activity diagrams are conceptually very similar to UCM paths [27]. However, UCMs also capture dynamic run-time behaviour (with dynamic stubs) and allocation of responsibilities (activities) to components. SPEC-VALUE could be used to derive formal specifications from enhanced UML activity diagrams and to provide a suitable validation framework. Another possibility would be to use SPEC-VALUE (with UCMs and LOTOS) between informal requirements (or use cases) and detailed-

design notations such as sequence diagrams, collaboration diagrams, and statechart diagrams. This is in a way similar to the process suggested by Bordeleau [62], but with an early emphasis on validation.

As an example, Gomaa's COMET (*Concurrent Object Modelling and Architectural Design Method*) [155] could benefit from the inclusion of UCMs and early validation. COMET is an iterative, use-case driven methodology that uses UML stereotypes extensively to provide a real-time flavour to various UML diagram elements and concepts. COMET's process starts with rather informal use case diagrams (requirements model) and then jumps to collaboration diagrams for the analysis model. Collaborations involve components and sequences of messages, just like MSCs. There is a big gap to fill here, and many design decisions need to be taken along the way. COMET provides limited means of ensuring consistency between the two models. UCMs could lead to a smoother transition between COMET's requirements and analysis models by helping to identify and structure components and objects, to discover appropriate messages and negotiation mechanisms, and to avoid complex and costly iterations in the evaluation of alternative architectures where components and messages are required to be identified. To some extent, UCMs could even replace use case diagrams in this context. Furthermore, the UCM-LOTOS testing framework provides a means to validate this intermediate model and to produce test cases applicable to the analysis model.

Standardization Methodologies

Telecommunication standards (from ANSI, ETSI, TIA, ISO, etc.) are often developed following a three-stage methodology first developed by ITU-T to describe services and protocols for ISDN [134][200][204]. Services are first described from the user's point of view in prose form and with tables (stage 1), then with sequences of messages between the different functional entities involved (stage 2), and finally with specifications of protocols and procedures (stage 3).

It is being recognized that formal description techniques and tools could help to cope with the increasing complexity of the services being standardized, to shorten the standards development cycle, to introduce formal test methodologies, and to assist in rapid validation and verification, harmonization, and evolution of standards. MSCs are now often used at stage 2 whereas SDL has started to

replace the traditional and informal pseudo-code used at stage 3. Recently, Monkewich proposed a quality assurance methodology (to be supported by ITU-T) that emphasizes the use of FDTs and the generation of tests [261].

SPEC-VALUE fits nicely in the first stage of this generic standardization methodology, as well as in the transition between stages 1 and 2. It uses notations adapted to the description of complex services in the early stages, while providing support for formalization, rapid prototyping, validation, and test case generation (in line with [261]). For wireless mobile telephony standards, Hodges and Visser already suggested UCMs for stage 1 descriptions and MSCs for stage 2 descriptions [178]. Amyot *et al.* also agree on UCMs for stage 1, but they propose the use of LOTOS to support the generation of consistent sets of MSCs from UCMs in stage 2 [20].

SPEC-VALUE has already been used in part by Yi [381], who constructed and validated a LOTOS prototype based on UCM descriptions of a *Wireless Intelligent Network* (WIN) feature. WIN is a standard being developed by the Telecommunication Industry Association (TIA) [38] to drive Intelligent Network (IN) capability into wireless networks based on the North-American standard ANSI-41 [37]. With the help of validation test cases, Yi also generated MSC scenarios similar to those found in the draft standard [344]. UCMs are now being used to capture stage 1 descriptions of numerous WIN features as well as new IMT-2000 features in the 3GPP2 (*3rd Generation Partnership Program*) standardization effort. Andrade also used SPEC-VALUE to describe and validate requirements and analysis patterns for mobile wireless systems and standards [33][35].

ITU-T Study Group 17 has also initiated work towards the standardization of a *User Requirements Notation* [82][203], which targets the representation of requirements for future telecommunication systems and services. At this early stage of its development, URN contains one candidate notation for the representation of non-functional requirements (GRL — *Goal-oriented Requirements Language* [83][247]) and a complementary scenario notation for functional requirements: the Use Case Map notation (Z.152) [84]. Relations between URN and other formal languages (including

transformations) will be defined in the Z.153 companion standard. A mapping from UCMs to LOTOS is likely to be considered for standardization, and SPEC-VALUE may represent the starting point.

SPEC-TO-TEST

This methodology, which is developed at the University of Ottawa, aims to develop functional test cases in TTCN from telecommunications systems informal requirements captured as Use Case Maps. Many paths are being investigated. The first path uses LOTOS for prototyping and validating the UCMs quickly, together with feature interaction detection. This specification uses functional test cases to generate MSCs. An SDL specification is then constructed from these MSCs and from the original UCMs. Industrial-strength SDL tools can finally generate TTCN test cases automatically. The second path investigates the direct generation of TTCN test cases from LOTOS specifications, whereas the third path bypasses the LOTOS step and targets the generation of SDL specifications directly from UCMs. In this process, SPEC-VALUE can be used in its entirety in the two first paths, and the third path can still benefit from the generation of test goals from UCMs in order to validate the SDL specification and to generate TTCN test cases from it.

9.3 Research Issues

Many items left for future work are distributed among the previous chapters. The following list recalls the most important ones, which target the automation and generalization of this work, and categorize them into medium-term and long-term research issues.

9.3.1 Medium-Term Research Issues

The following issues relate to how to *improve* computer-based support of SPEC-VALUE in order to reach the **defined** level of maturity (level 3) in the FSM model (Table 33):

- Definition of a data dictionary for early consistency and completeness verification of UCM scenarios, with tool support. This tool, which could be part of the existing UCM Navigator tool, would also check compliance to the style and content guidelines presented in Section 4.2.

- Automated generation of LOTOS *skeleton* specifications that would cover the most automatable construction rules as discussed in Section 5.2.5. These skeleton specifications would still require to be completed by specifiers for the most complex construction rules.
- Automated selection of test goals from UCMs according to preset strategies as defined in the testing patterns of Section 6.3.
- Automated insertion of probes (without the (*_Probe_*) comments) in LOTOS specifications according to the optimized strategy presented in Section 7.4.2. This would lead to fully automated structural coverage measurements and fewer errors in instrumentation.

9.3.2 Long-Term Research Issues

These issues are concerned with how to widen the scope of SPEC-VALUE and how to automate this methodology in order to reach the **managed** and **optimizing** levels of maturity (levels 4 and 5) in the FSM model (Table 33). Solving these issues would pave the way to rapid and *automated* prototyping and validation of high-level designs of telecommunications systems.

The first set of issues is concerned mainly with the semantics of the UCM notation:

- Definition of a data model for UCMs in order to define precisely data types, identifiers, operations, conditions, databases, etc. Whether a limited data model or a full-fledge (and standard) language such as ASN.1 or ADTs is necessary remains an important research question.
- Definition of a flexible component model with concepts such as actors, ports/interfaces, channels, etc. Having such a model would enable one to use UCM paths on top of component notations other than Buhr's, including ROOMcharts, UML collaborations and packages, SDL blocks and processes, and Architecture Description Languages such as ACME [147][148]. A flexible component model would enable better description and analysis of software architectures and improve the adaptability of the notation to the culture, needs and knowledge of specific design teams.
- More precise definition of the dynamic semantics of UCMs.

- Distinction between normal and exceptional scenarios at the UCM level. The concept of scenario definition can be used in this context [258].
- All these semantic precisions, which could be standardized for instance in [84], would enable better and more automatable integrations of UCMs to other languages and notations, including UML, MSCs, and SDL.

The second set of issues relates to the application of SPEC-VALUE:

- Definition of libraries of reusable and instantiable UCM behaviour patterns of feature definitions and other generic scenarios, with tool support.
- Definition of libraries of message exchange patterns for inter-component causality relationships, with tool support.
- Automated *synthesis* of LOTOS specifications from these improved UCMs. This would require additional details (design decisions) to be provided at the UCM level.
- Algorithms for the generation of rejection test goals from UCMs, with tool support.
- Automated generation of specification test cases (in LOTOS) from UCM-based test goals.
- Automated generation of implementation test cases (e.g. in TTCN) from UCM-based test goals.
- Automated generation of LOTOS mutants for test suite evaluation.
- Optimization of test suites and test case management through metrics based on structural coverage, as suggested in Section 7.6.3.
- Use of underlying semantic models that capture causality better than LOTOS's LTSs, for instance causal trees [108], dynamic causal trees [319], or bundle event structures [239] (discussed in Section 3.1). Tool-supported causal testing based on such models could potentially lead to a reduced number of testing patterns and test goals, and to better diagnostics.

References

-
- [1] Abdalla, M.M., Khendek, F. and Butler, G. (1999) “New Results on Deriving SDL Specifications from MSCs”. In: *SDL'99, Proceedings of the Ninth SDL Forum*, Montréal, Canada. Elsevier.
 - [2] Abdurazik, A. and Offut, J. (2000) “Using UML Collaboration Diagrams for Static Checking and Test Generation”. In: <<UML>>2000, *3rd International Conference on the Unified Modeling Language*, York, UK, October 2000. LNCS 1939, 383-395.
 - [3] Adams, M., Coplien, J., Gamoke, R., Hanmer, R., Keeve, F., and Nicodemus, K. (1998) “Fault-Tolerant Telecommunication System Patterns”. In: L. Rising (ed.), *The Patterns Handbook: Techniques, Strategies, and Applications*, Cambridge University Press, New York, 189-202.
http://www.bell-labs.com/~cope/patterns/telecom/PLoP95_telecom.html
 - [4] Aho, A., Gallagher, S., Griffeth, N., Scheel, C., and Swayne, D. (1998) “Sculptor with Chisel: Requirements Engineering for Communications Services”. In: K. Kimbler and L. G. Bouma (Eds), *Fifth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'98)*, Lund, Sweden, September 1998. IOS Press, 45-63.
<http://www-db.research.bell-labs.com/user/nancyg/sculptor.ps>
 - [5] Alexander, C., Ishikawa, S., and Silverstein, M. (1977) *A Pattern Language*. Oxford University Press, New York, USA
 - [6] Alexander, C. (1979) *The Timeless Way of Building*. Oxford University Press, New York, USA.
 - [7] Alur, R. and Dill D. (1994) “A theory of timed automata”. In: *Theoretical Computer Science* (126), 183-235.
 - [8] Alur, R. Holzmann, G. and Peled, D. (1996) “An Analyzer for Message Sequence Charts”. In: *Software Concepts and Tools*, 17(2):70-77. <http://cm.bell-labs.com/cm/cs/what/ubet/papers/aAfMSCs.ps.gz>

- [9] Alur, R., Etessami, K., and Yannakakis, M. (2000) "Inference of Message Sequence Charts". In: *22th International Conference on Software Engineering (ICSE 2000)*, Limerick, Ireland, ACM Press, 304-313.
- [10] Ammann, P.E., Black, P.E., and Majurski, W. (1998) "Using Model Checking to Generate Tests from Specifications". In: *Proceedings of 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, Brisbane, Australia, December. <http://hissa.ncsl.nist.gov/~black/Papers/icfem98.ps>
- [11] Ammann, P.E. and Black, P.E. (2000) "A Specification-Based Coverage Metric to Evaluate Test Sets". In: *International Journal of Reliability, Quality and Safety Engineering*, World Scientific Publishing, Singapore (December 2000). <http://hissa.ncsl.nist.gov/~black/Papers/ijrqse.ps>.
- [12] Amyot, D. (1994) *Formalization of Timethreads Using LOTOS*. M.Sc. thesis, Dept. of Computer Science, University of Ottawa, Ottawa, Canada. <http://www.site.uottawa.ca/~damyot/phd/msctheses.pdf>
- [13] Amyot, D. (1994) *LOTOS Generation from Timethread Maps: A Language and a Tool*. CSI 5900 project report, Dept. of Computer Science, University of Ottawa, Canada. <http://www.site.uottawa.ca/~damyot/ucm/tmdl/tmdl.pdf>
- [14] Amyot, D., Bordeleau, F., Buhr, R.J. A., and Logrippo, L. (1995) "Formal support for design techniques: a Timethreads-LOTOS approach". In: G. von Bochman, R. Dssouli, O. Rafiq (Eds.), *FORTE VIII, 8th International Conference on Formal Description Techniques*, Montréal, Canada, October 1995. Chapman & Hall, 57-72. <http://www.site.uottawa.ca/~damyot/phd/forte95/forte95.pdf>
- [15] Amyot, D., Logrippo, L., and Buhr, R.J.A. (1997) "Spécification et conception de systèmes communicants : une approche rigoureuse basée sur des scénarios d'usage". In: G. Leduc (Ed.), *CFIP 97, Ingénierie des protocoles*, Liège, Belgium, September 1997. Hermès, 159-174. <http://www.site.uottawa.ca/~damyot/cfip97/cfip97.pdf>
- [16] Amyot, D., Hart, N., Logrippo, L., and Forhan, P. (1998) "Formal Specification and Validation using a Scenario-Based Approach: The GPRS Group-Call Example". In: Selic, B. (Ed.), *ObjecTime Workshop on Research in OO Real-Time Modeling*, Ottawa, Canada, January 1998. <http://www.site.uottawa.ca/~damyot/wrroom98/wrroom98.pdf>
- [17] Amyot, D. (1998) *Group Communication Server: A Scenario-Based Design Exercise*. CITO report #1388, Ottawa, Canada, June 1998. <http://www.site.uottawa.ca/~damyot/gcs/>
- [18] Amyot D. (1998) *Use Case Maps for the Design and the Validation of Interaction-Free Telephony Features*. CITO report #1430, Ottawa, Canada. <http://www.site.uottawa.ca/~damyot/FI/>
- [19] Amyot, D. (1999) "Éditorial : Problèmes (ré)partis?". In: APIIQ, *L'Expertise informatique*, Vol. 4, n° 1, 18-22, winter 1999, p.2. <http://www.apiiq.qc.ca/expertise/>
- [20] Amyot, D., Andrade, R., Logrippo, L., Sincennes, J., and Yi, Z. (1999) "Formal Methods for Mobility Standards". In: *IEEE 1999 Emerging Technology Symposium on Wireless Communications & Systems*, Richardson, Texas, USA, April 1999. <http://www.UseCaseMaps.org/pub/ets99.pdf>
- [21] Amyot, D. and Andrade, R. (1999) "Description of Wireless Intelligent Network Services with Use Case Maps". In: *SBRC'99, 17° Simpósio Brasileiro de Redes de Computadores*, Salvador, Brazil, May 1999, 418-433. <http://www.UseCaseMaps.org/pub/sbrc99.pdf>

-
- [22] Amyot, D., Buhr, R.J.A., Gray, T., and Logrippo, L. (1999) "Use Case Maps for the Capture and Validation of Distributed Systems Requirements". In: *RE'99, Fourth IEEE International Symposium on Requirements Engineering*, Limerick, Ireland, June 1999, 44-53. <http://www.UseCaseMaps.org/pub/re99.pdf>
- [23] Amyot, D. and Miga, A. (2000) *Use Case Maps Linear Form in XML, version 0.20*, July 2000. <http://www.UseCaseMaps.org/xml/>
- [24] Amyot, D. and Logrippo, L. (2000) "Use Case Maps and LOTOS for the Prototyping and Validation of a Mobile Group Call System". In: *Computer Communication*, 23(12), 1135-1157. <http://www.UseCaseMaps.org/pub/cc99.pdf>
- [25] Amyot, D., Charfi, L., Gorse, N., Gray, T., Logrippo, L., Sincennes, J., Stepien, B., and Ware, T. (2000) "Feature description and feature interaction analysis with Use Case Maps and LOTOS". In: *Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'00)*, Glasgow, Scotland, UK, May 2000. <http://www.UseCaseMaps.org/pub/fiw00lotos.pdf>
- [26] Amyot, D., and Logrippo, L. (2000) "Structural Coverage for LOTOS—A Probe Insertion Technique". In: H. Ural, R.L. Probert and G.v. Bochmann (eds) *Testing of Communicating Systems: Tools and Techniques (TestCom 2000)*. Kluwer Academic Publishers, 19-34.
- [27] Amyot, D. and Mussbacher, G. (2000) "On the Extension of UML with Use Case Maps Concepts". In: *UML2000, 3rd International Conference on the Unified Modeling Language*, York, UK, October 2000. LNCS 1939, 16-31. <http://www.UseCaseMaps.org/pub/uml2000.pdf>
- [28] Amyot, D. (2000) "Use Case Maps as a Feature Description Language". In: S. Gilmore and M. Ryan (Eds), *Language Constructs for Designing Features*. Springer-Verlag. 27-44. <http://www.UseCaseMaps.org/pub/fireworks2000.pdf>
- [29] Amyot, D. and Eberlein, A. (2001) "An Evaluation of Scenario Notations for Telecommunication Systems Development". In: *9th International Conference on Telecommunications Systems (ICTS'01)*, Dallas, USA, March 2001. <http://www.UseCaseMaps.org/pub/icts01.pdf>
- [30] Amyot, D. and Mussbacher, G. (2001) "Bridging the Requirements/Design Gap in Dynamic Systems with Use Case Maps (UCMs)". Tutorial in: *23rd International Conference on Software Engineering (ICSE'01)*, Toronto, Canada, May 2001. <http://www.UseCaseMaps.org/pub/icse01.pdf>
- [31] Andersson, M. and Bergstrand, J. (1995) *Formalizing Use Cases with Message Sequence Charts*. Master thesis, Department of Communication Systems, Lund Institute of Technology, Sweden, May 1995. http://www.efd.lth.se/~d87man/EXJOB/Title_Abstract_Preface.html
- [32] Andrade, R. (2000) "Applying Use Case Maps and Formal Methods to the Development of Wireless Mobile ATM Networks". In: *Lfm2000, The Fifth NASA Langley Formal Methods Workshop*, Williamsburg, Virginia, USA, June 2000. <http://www.UseCaseMaps.org/pub/lfm2000.pdf>
- [33] Andrade, R. and Logrippo, L. (2000) "Reusability at the Early Development Stages of the Mobile Wireless Communication Systems". In: *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2000)*, Vol. VII, Computer Science and Engineering: Part I, Orlando, Florida, July 2000, 11-16. <http://www.UseCaseMaps.org/pub/sci2000.pdf>

- [34] Andrade, R., Bottomley, M., Logrippo, L., and Coram, T. (2000) "A Pattern Language for Mobility Management". In: *Proc. of the 7th Conference on the Pattern Languages of Programs (PLoP 2000)*, Monticello, Illinois, August. <http://lotos.site.uottawa.ca/ftp/pub/Lotos/Papers/plop2000>
- [35] Andrade, R. (2001) *Capture, Reuse, and Validation of Requirements and Analysis Patterns for Mobile Systems*. Ph.D. thesis, SITE, University of Ottawa, Canada, May 2001. http://lotos.site.uottawa.ca/ftp/pub/Lotos/Theses/ra_phd.pdf
- [36] Andriantsiferana, L. Ghribi, B., and Logrippo, L. (1999) "Prototyping and Formal Requirement Validation of GPRS: A Mobile Data Packet Radio Service for GSM". In: *Proceedings of the 7th International Working Conference on Dependable Computing For Critical Applications (DCCA-7)*, San Jose, CA, USA, 99-118. <http://lotos.site.uottawa.ca/ftp/pub/Lotos/Papers/dcca7.ps.gz>
- [37] ANSI/TIA/EIA (1997) *ANSI-41-D, Cellular Radiotelecommunications Intersystem Operations*.
- [38] ANSI/TIA/EIA (1998) *ANSI 771, Wireless Intelligent Networks (WIN). Additions and modifications to ANSI-41 (Phase 1)*. TR-45.2.2.4, December 1998.
- [39] Ardis, M.A., Chaves, J.A., Jagadeesan, L. J., Mataga, P., Puchol, C., Staskauskas, M.G., and Olnhausen, J.V. (1996) "A Framework for Evaluating Specification Methods for Reactive Systems — Experience Report". In: *IEEE Transactions on Software Engineering*, 22 (6), 378-389.
- [40] Ashkar, P. (1992) *Symbolic Execution of LOTOS Specifications (SELA)*. M.Sc. thesis, Dept. of Computer Science, University of Ottawa, Ottawa, Canada. http://lotos.site.uottawa.ca/ftp/pub/Lotos/Theses/ashkar_msc.pub.ps.gz
- [41] Balzer, R.M., Goldman, N.M., and Wile, D.S. (1982) "Operational Specification as the Basis for Rapid Prototyping". In: *ACM SIGSOFT Software Engineering Notes*, vol. 7, no. 5, December 1982, 3-16.
- [42] Barbuceanu, M., Gray, T., and Mankovski, S. (1998) "How To Make Your Agents Fulfil Their Obligations". In: H.S. Nwana and D.T. Ndumu (Eds), *PAAM'98, Third Conference on Practical Application of Intelligent Agents and Multi-Agents*, London, UK, March 1998, 255-276.
- [43] Baumgarten, B. and Wiland, H. (1998) "Qualitative Notions of Testability". In: *11th International Workshop on Testing of Communicating Systems (IWTCS'98)*, Tomsk, Russia.
- [44] Beizer, B. (1995) *Black box testing*. John Wiley & Sons.
- [45] Ben-Abdallah, H. and Leue, S. (1997) *MESA: Support for scenario-based design of concurrent systems*. Technical Report 97-12, Department of Electrical & Computer Engineering, University of Waterloo, Canada, October.
- [46] Ben Achour, C., Rolland, C., Maiden, N.A.M., and Souveyet, C. (1999) "Guiding Use Case Authoring: Results of an Empirical Study". In: *RE'99, Fourth IEEE International Symposium on Requirements Engineering*, Limerick, Ireland, June 1999, 36-43.
- [47] Benner, K.M., Feather, M.S., Johnson, W.L., and Zorman, L.A. (1993) "Utilizing Scenarios in the Software Development Process". In: *Information System Development Process*, Elsevier Science, B.V. North-Holland, 117-134.

-
- [48] Bertolino, A., Corradini, F., Inverardi, P. and Muccini, H. (2000) "Deriving Test Plans from Architectural Descriptions". In: *22th International Conference on Software Engineering (ICSE 2000)*, Limerick, Ireland, ACM Press, 220-229.
- [49] Biermann, A.W. and Lrishnaswamy, R. (1976) "Constructing Programs from Example Computations". In: *IEEE Transactions on Software Engineering*, SE-2, 141-153.
- [50] Binder, R.V. (1998) "How to test UML Sequence Diagram scenarios". In: *Object Magazine*, 7(9), March 1998, 16-19.
- [51] Binder, R.V. (2000) *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison Wesley object technology series.
- [52] Black, P.E., Okun, V. and Yesha, Y. (2000) "Mutation Operators for Specifications". In: *15th Automated Software Engineering Conference (ASE2000)*, Grenoble, France, September, IEEE Computer Society, pages 81-88. <http://hissa.ncsl.nist.gov/~black/Papers/opers.ps>
- [53] Bochmann, G.v. (1993) "Specification languages for communication protocols". In: *CHDL'93, Conference Proceedings of the IFIP Conference on Hardware Description Languages and their Applications*, OCRI Publications, Ottawa, Canada, 365-382.
- [54] Boehm, B. (1988) "A Spiral Model of Software Development and Enhancement". In: *IEEE Computer*, May, 61-72.
- [55] Boehm, B. (1991) "Software Risk Management: Principles and Practices". In: *IEEE Software*, January, 32-41.
- [56] Bolognesi, T. and Brinksma, E. (1987) "Introduction to the ISO Specification Language LOTOS". In: *Computer Networks and ISDN Systems*, vol. 14, no. 1, 25-59.
- [57] Bolognesi, T., van de Lagemaat, J., and Vissers, C. (1995) *LOTOSphere: Software Development with LOTOS*. Kluwer Academic Publishers, The Netherlands.
- [58] Bolognesi, T., De Frutos, D., Langerak, R., and Latella, D. (1995) "Correctness Preserving Transformations for the Early Phases of Software Development". In: Bolognesi, T., van de Lagemaat, J., and Visser, C., *LOTOSphere: Software Development with LOTOS*. Kluwer Academic, The Netherlands.
- [59] Boni Bangari, A. (1997) A Use Case Driven Validation Framework and Case Study. M.Sc. thesis, SITE, University of Ottawa, Ottawa, Canada.
- [60] Bordeleau, F. (1993) *Visual Descriptions, Formalisms and the Design Process*. M.Sc. thesis, School of Computer Science, TR-SCE-93-35, Carleton University, Ottawa, Canada.
- [61] Bordeleau, F. and Buhr, R.J.A. (1997) "The UCM-ROOM Design Method: from Use Case Maps to Communicating State Machines". In: *Conference on the Engineering of Computer-Based Systems*, Monterey, USA, March 1997. <http://www.UseCaseMaps.org/pub/UCM-ROOM.pdf>
- [62] Bordeleau, F. (1999) *A Systematic and Traceable Progression from Scenario Models to Communicating Hierarchical Finite State Machines*. Ph.D. thesis, School of Computer Science, Carleton University, Ottawa, Canada. http://www.UseCaseMaps.org/pub/fb_phdthesis.pdf

- [63] Bordeleau, F. and Cameron, D. (2000) "On the Relationship between Use Case Maps and Message Sequence Charts". In: *2nd Workshop of the SDL Forum Society on SDL and MSC (SAM2000)*, Grenoble, France, June 2000. <http://www.UseCaseMaps.org/pub/sam2000.pdf>
- [64] Boudol, G. and Castellani, I. (1990) "Three equivalent semantics for CCS". In: *LNCS*, vol 469, Springer-Verlag, 96-141.
- [65] Bouma, L.G., and H. Velthuisen (eds), *Second International Workshop on Feature Interactions in Telecommunications Systems*, IOS Press, Amsterdam, Netherlands, 1994.
- [66] Boumezbeur, R. and Logrippo, L. (1993) "Specifying telephone systems in LOTOS". *IEEE Communications Magazine*, 31(8), August, 38-45. <http://lotos.site.uottawa.ca/ftp/pub/Lotos/Papers/svtsl.ps.Z>
- [67] Bowen, J. and Hinchley, M. (1995) "Seven More Myths of Formal Methods". In: *IEEE Software*, July, 34-41.
- [68] Bowen, J. (2000) "The Ethics of Safety-Critical Systems". In: *Communications of the ACM*, 43(4), April, 91-97
- [69] Brinksma, E. (1988) "A theory for the derivation of tests". In: S. Aggarwal and K. Sabnani (Eds), *Protocol Specification, Testing and Verification VIII*, North-Holland, 63-74, June 1988.
- [70] Brinksma, E., Tretmans, J., and Verhaard, L. (1991) "A Framework for Test Selection". In: B. Jonsson, J. Parrow, and B. Pehrson (Eds.), *Protocol Specification, Testing and Verification XI*, Elsevier Science Publishers B.V.
- [71] Brinksma, E., Kaoten, J.-P., Langerak, R. and Latella, D. (1998) "Partial order models for quantitative extensions of LOTOS". In: *Computer Networks and ISDN Systems*, 30, Elsevier Science B.V., 925-950.
- [72] Brinksma, E. and Tretmans, J. (2000) "Testing transition systems: An annotated bibliography". In F. Cassez, C. Jard, B. Rozoy, and M. Ryan (eds) *Proceedings of Summer School MOVEP'2k Modelling and Verification of Parallel Processes*, Nantes, July 2000, 44-50. <http://fmt.cs.utwente.nl/publications/deposit/BrTr00.ps.gz>
- [73] Brinksma, E. (2000) "Verification is Experimentation!". In: *11th International Conference on Concurrency Theory (CONCUR'2000)*, Pennsylvania, USA, August 2000.
- [74] Buhr, R.J.A. and Casselman, R.S. (1996) *Use Case Maps for Object-Oriented Systems*, Prentice-Hall, USA. http://www.UseCaseMaps.org/pub/UCM_book95.pdf
- [75] Buhr, R.J.A. (1998) *High Level Design and Prototyping of Dynamic Agencies*, research project description. <http://www.sce.carleton.ca/rads/agents/>
- [76] Buhr, R.J.A. (1998) "Use Case Maps as Architectural Entities for Complex Systems". In: *IEEE Transactions on Software Engineering, Special Issue on Scenario Management*. Vol. 24, No. 12, December 1998, 1131-1155. <http://www.UseCaseMaps.org/pub/tse98final.pdf>
- [77] Buhr, R.J.A., Amyot, D., Elammari, M., Quesnel, D., Gray, T., and Mankovski, S. (1998) "High Level, Multi-agent Prototypes from a Scenario-Path Notation: A Feature-Interaction Example". In: H.S. Nwana and D.T. Ndumu (Eds), *PAAM'98, Third Conference on Practical Application of Intelligent Agents and Multi-Agents*, London, UK, March 1998, 277-295. <http://www.UseCaseMaps.org/pub/4paam98.pdf>

-
- [78] Buhr, R.J.A., Amyot, D., Elammari, M., Quesnel, D., Gray, T., and Mankovski, S. (1998) "Feature-Interaction Visualization and Resolution in an Agent Environment". In: K. Kimbler and L. G. Bouma (Eds), *Fifth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'98)*, Lund, Sweden, September 1998. IOS Press, 135-149. <http://www.UseCaseMaps.org/pub/fiw98.pdf>.
- [79] Buhr, R.J.A., Elammari, M., Gray, T., and Mankovski, S. (1998) "Applying Use Case Maps to Multi-agent Systems: A Feature Interaction Example", *Hawaii International Conference on System Sciences (HICSS'98)*, Hawaii, January 1998. <http://www.useCaseMaps.org/pub/hiccs98.pdf>
- [80] Buhr, R.J.A. (1999), "Understanding Macroscopic Behaviour Patterns in Object-Oriented Frameworks, with Use Case Maps". In: Fayad, M.E., Schmidt, D.C., and Johnson, R.E. (eds) *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. John Wiley & Sons. <http://www.UseCaseMaps.org/pub/uof.pdf>
- [81] Buschmann, F. Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996) *Pattern-Oriented Software Architecture — A System of Patterns*. John Wiley & Sons.
- [82] Cameron, D. et al. (2001) *Draft Recommendation Z.150 — User Requirements Notation (URN)*. Canadian Contribution to ITU-T Study Group 10, September 2001. <http://www.UseCaseMaps.org/urn>
- [83] Cameron, D. et al. (2001) *Draft Recommendation Z.151 — URN-NFR: Goal-oriented Requirements Language (GRL)*. Canadian Contribution to ITU-T Study Group 10, September 2001. <http://www.UseCaseMaps.org/urn>
- [84] Cameron, D. et al. (2001) *Draft Recommendation Z.152 — URN-FR: Use Case Maps Notation (UCM)*. Canadian Contribution to ITU-T Study Group 10, September 2001. <http://www.UseCaseMaps.org/urn>
- [85] Cameron, E.J., Griffeth, N., Linand, Y.-J., Nilson, Y.-J., Schnure, W.K. and Velthuijsen, H. (1994) "A Feature Interaction Benchmark for IN and Beyond". In: L. G. Bouma and H. Velthuijsen (eds), *Feature Interactions in Telecommunications Systems*, Amsterdam, The Netherlands, May 1994. IOS Press, 1-23. <http://www-db.research.bell-labs.com/user/nancyg/benchmark.ps>
- [86] Carver, R.H. and Tai, K.C. (1995) "Test Sequence Generation from Formal Specifications of Distributed Programs". In: *15th International Conference on Distributed Computing Systems*, May, 360-367.
- [87] Carver, R.H. and Chen, J. (1996) "Incremental Conformance Testing using LOTOS Specifications". In: *Proc. 5th International Conference on Computer Communications and Networks (IC3N'96)*, October.
- [88] Carver, R.H. and Tai, K.C. (1998) "Use of Sequencing Constraints for Specification-Based Testing of Concurrent Programs". In: *IEEE Transactions on Software Engineering*, Vol. 24, No. 6, June, 471-490.
- [89] Cavalli, A., Kim, S., and Maigron, P. (1993) "Improving Conformance Testing for LOTOS". In: R.L. Tenney, P.D. Amer and M.Ü. Uyar (Eds), *FORTE VI, 6th International Conference on Formal Description Techniques*, North-Holland, 367-381, October 1993.
- [90] Chandrasekarn, P. (1997) "How Use Case Modeling Policies Have Affected the Success of Various Projects (Or How to Improve Use Case Modeling)". In: *Addendum to the 1997 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97)*. 6-9.

- [91] Charfi, L. (2001) *Formal Modeling and Test Generation Automation with Use Case Maps and LOTOS*. M.Sc. thesis, SITE, University of Ottawa, Canada, 2001.
- [92] Charles, Olivier. (1997) *Application des hypothèses de test à une définition de la couverture*. Ph.D. thesis, Université Henri Poincaré — Nancy 1, Nancy, France, October 1997.
- [93] Chehaibar, G., Garavel, H., Mounier, L., Tawbi, N., and Zulian, F. (1996) “Specification and Verification of the PowerScale™ Bus Arbitration Protocol: An industrial Experiment with LOTOS”. In: R. Gotzhein and J. Brederke (Eds), *Proceedings of FORTE/PSTV'96*, Kaiserslautern, Germany, 435-450, October 1996.
- [94] Cheung, T. Y. and Ren, S. (1992) *Operational Coverage and Selective Test Sequence Generation for LOTOS Specification*. TR-92-07, Dept. of Computer Science, University of Ottawa, Canada, January 1992.
- [95] Christensen, S., Jørgensen, J.B., and Kristensen, L.M. (1997) “Design/CPN — A Computer Tool for Coloured Petri Nets”. In: E. Brinksma (ed.), *TACAS'97, Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, Twente, The Netherlands, volume 1217 of *Lecture Notes in Computer Science*, Springer-Verlag, 209-223.
- [96] Chung, L., Nixon, B.A., Yu, E. and Mylopoulos, J. (2000) *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers.
- [97] Clarke, E.M., Wing, J.M. *et al.* (1996) “Formal Methods: State of the Art and Future Directions”. In: *ACM Computing Surveys*, Vol. 28, No. 4, December 1996.
- [98] Cockburn, A. (1997) “Structuring Use cases with goals”. In: *Journal of Object-Oriented Programming (JOOP/ROAD)*, 10(5), September 1997, 56-62.
<http://members.aol.com/acockburn/papers/usecases.htm>
- [99] Coelho da Costa, R.J. and Courtiat, J.-P. (1993) “A true concurrency semantics for LOTOS”. In: M. Diaz and R. Groz (Eds), *Formal Description Techniques, V*, North-Holland.
- [100] Coelho da Costa, R.J. (1993) *Systèmes de transitions étiquetés causaux: une nouvelle approche pour la description du comportement événementiel de systèmes concurrents*. Ph.D. thesis, report 93179, LAAS. Université Paul Sabatier, Toulouse, France.
- [101] Coplien, J.O. (1997) *A Pattern Definition*. <http://hillside.net/patterns/definition.html>
- [102] Corriveau, J.-P. (1996) “Retraçage et processus de développement pour des projets industriels orientés objet”. In: APIIQ, *L'Expertise informatique*, Vol. 3, n° 1, 18-22, summer 1996.
<http://www.apiiq.qc.ca/expertise/>
- [103] Courtiat, J.-P., Dembinski, P., Holzmann, G.J., Logrippo, L., Rudin, H. and Zave, P. (1996) “Formal methods after 15 years: Status and trends — A paper based on contributions of the panelists at the FORmal TEchnique '95 Conference, Montreal, October 1995”. In: *Computer Networks and ISDN Systems*, 28, Elsevier Science B.V., 1845-1855.
- [104] Craigen, D., Gerhart, S., and Ralston, T. (1994) *Industrial applications of formal methods to model, design, and analyze computer systems: an international survey*. Noyes Data Corporation (Publisher), USA.

-
- [105] Craigen, D., Gerhart, S., and Ralston, T. (1995) "Formal Methods Technology Transfer: Impediments and Innovation". In M. G. Hinchey and J. P. Bowen (eds): *Applications of Formal Methods*. Prentice-Hall International Series in Computer Science, September 1995, 399-419.
- [106] Damm, W. and Harel, D. (1999) "LCSs: Breathing Life into Message Sequence Charts". In: *3rd IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'99)*, Kluwer Academic Publishers, 293-312.
- [107] Dardenne, A., van Lamsweerde, A. and Fickas, S. (1993) "Goal-Directed Requirements Acquisition". In: *Science of Computer Programming*, 20, 3-50.
- [108] Darondeau, P. and Deganeau, P. (1989) "Causal Trees". In: *ICALP'89*, LNCS vol. 372, Springer-Verlag, 234-248.
- [109] Davis, A.M. (1982) "Rapid Prototyping Using Executable Requirements Specifications". In: *ACM SIGSOFT Software Engineering Notes*, vol. 7, no. 5, December 1982, 39-44.
- [110] Davis, M.D. and Weyuker, E.J (1983) *Computability, Complexity, and Languages*. Academic Press, New York, USA.
- [111] DeLano, D., and Rising, L. (1996) "System Test Pattern Language". In: *Pattern Languages of Programs (PLoP'96)*, Allerton Park, Illinois, USA. <http://www.agcs.com/patterns/papers/systestp.htm>
- [112] DeMillo, R.A., Lipton, R.J., and Sayward, F.G. (1978) "Hints on test data selection: Help for the practicing programmer". In: *IEEE Computer*, 11(4), April, 34-41.
- [113] De Nicola, R. and Hennessey, M.C.B. (1984) "Testing equivalences for processes". In: *Theoretical Computer Science*, 34:83-133.
- [114] Desharnais, J., Frappier, M., Khédri, R., and Mili, A. (1997). "Integration of Sequential Scenarios". In: *ESEC'97, Sixth European Engineering Conference*, LNCS 1301, Springer-Verlag, 310-326.
- [115] Diaz, M., Juanole, G., and Courtiat, J.-P. (1994) "Observer — A Concept for Formal on-line Validation of Distributed Systems". In: *IEEE Transactions on Software Engineering*, Vol. 20, No. 12, December 1994, 900-913.
- [116] Drira, K. and Azéma, P. (1995) "Les graphes de refus pour la vérification de conformité et l'analyse de testabilité des protocoles de communication". In: *Electronic Journal on Networks and Distributed Processing*, No. 1, April 1995, 27-47.
- [117] Du Bousquet, L., Ramangalahy, S., Simon, S. and Viho, C. (2000) "Formal Test Automation: The conference Protocol with TGV/TORX". In: H. Ural, R.L. Probert and G.v. Bochmann (eds) *Testing of Communicating Systems: Tools and Techniques (TestCom 2000)*. Kluwer Academic Publishers, 220-228.
- [118] Dulz, W., Gruhl, S., Lambert, L., and Söllner, M. (1999) "Early performance prediction of SDL/MSD specified systems by automated synthetic code generation". In: *SDL'99, Proceedings of the Ninth SDL Forum*, Montréal, Canada. Elsevier.
- [119] Easterbrook, S. and Nuseibeh, B. (1995) "Managing Inconsistencies in an Evolving Specification". In: *Proceedings of the Second International Symposium on Requirements Engineering*, York, UK, March.
- [120] Eberlein, A. (1997) *Requirements Acquisition and Specification for Telecommunication Services*. PhD thesis, University of Wales, Swansea, UK. <http://kona.swan.ac.uk/~eeberle/Publications/thesis.tar.gz>

- [121] Ehrig, H. and Mahr, B. (1985) *Fundamentals of Algebraic Specification 1 (Equations and Initial Semantics)*. Volume 6 of EATCS Monographs on Theoretical Computer Science, Springer-Verlag, Berlin, Germany.
- [122] EIA (1999) *Systems Engineering Capability Model (SECM)*. Interim Standard EIA/IS 731-1.
- [123] Ek, A., Ellsberger, J. and Wiles, A. (1993) *Computer Supported Test Generation from SDL Specifications*. Technical Report, Telia Research, Sweden.
- [124] Elammari, M. and Lalonde, W. (1999) "An Agent-Oriented Methodology: High-Level and Intermediate Models". In: *Proc. of the 1st Int. Workshop on Agent-Oriented Information Systems (AOIS'99)*, Heidelberg, Germany, June 1999. <http://www.UseCaseMaps.org/pub/aom-aois99.pdf>
- [125] Elkoutbi, M., Khriiss, I., and Keller, R.K. (1999) "Generating User Interface Prototypes from Scenarios". In: *RE'99, Fourth IEEE International Symposium on Requirements Engineering*, Limerick, Ireland, June 1999, 150-158. <ftp://ftp.iro.umontreal.ca/pub/gelo/Publications/Papers/isre99.pdf>
- [126] ESPRIT (1996) *Cooperative Requirements Engineering With Scenarios (CREWS)*. Project 21.903. <http://sunsite.informatik.rwth-aachen.de/CREWS/>
- [127] ETSI (1992) Digital Cellular Telecommunication System (Phase 2). *Mobility Application Part (GSM 09.02), Version 4.0.0* (June 1992).
- [128] ETSI (1996) Digital Cellular Telecommunications system (Phase 2+); *General Packet Radio Service (GPRS); Service Description Stage 1 (GEM 02.60), Version 2.0.0* (November 1996).
- [129] Faci, M., Logrippo, L. and Stépien, B. (1989) "Formal Specification of telephone systems in LOTOS", *Protocol Specification, Verification and Testing*, IX, North-Holland.
- [130] Faci, M., Logrippo, L., and Stépien, B. (1991) "Formal Specification of Telephone Systems in LOTOS: The Constraint-Oriented Approach". *Computer Networks and ISDN Systems*, 21 (1991) 53-67. <http://lotos.site.uottawa.ca/ftp/pub/Lotos/Papers/telephone.CNIS9007.ps.Z>
- [131] Faci, M. and Logrippo, L. (1994) "Specifying Features and Analysing their Interactions in a LOTOS Environment". In: L. G. Bouma and H. Velthuijsen (eds), *Second International Workshop on Feature Interactions in Telecommunications Software Systems*, IOS Press, 136-151. <http://lotos.site.uottawa.ca/ftp/pub/Lotos/Papers/Fits94.CameraReady.ps.gz>
- [132] Faci, M. (1995) *Detecting Feature Interaction in Telecommunications Systems Designs*. Ph.D. thesis, Department of Computer Science, University of Ottawa, November 1995. http://lotos.site.uottawa.ca/ftp/pub/Lotos/Theses/mf_phd.ps.gz
- [133] Faci, M., Logrippo, L., and Stépien, B (1997) "Structural Models for Telephone Specifications". In: *Computer Network & ISDN Systems*, 29, 501-528. <http://lotos.site.uottawa.ca/ftp/pub/Lotos/Papers/isdn95.ps.gz>
- [134] Faynberg, I., Gabuzda, L.R., and Jacobson, T. (1997) "The Development of the Wireless Intelligent Network (WIN) and its Relation to the International Intelligent Network Standards". In: *Bell Labs Technical Journal*, Vol. 2, No. 3, summer 1997, 76-86.

-
- [135] Fernandez, J.-C., Garavel, H., Kerbrat, A., Mateescu, R., Mounier, L., and Sighireanu, M. (1996) "CADP: A Protocol Validation and Verification Toolbox". In: *Proceedings of the 8th Conference on Computer-Aided Verification*, New Brunswick, NJ, USA, 437-440.
- [136] Finkelstein, A. (ed.) (2000) *The Future of Software Engineering*. Special track of the *22th International Conference on Software Engineering*, ACM Press.
- [137] Fraser, M.D., Kumar, K. and Vaishnavi, V.K. (1994) "Strategies for incorporating formal specifications in software development". In: *Communications of the ACM*, Vol. 37, No. 10, October 1994, 74-86
- [138] Fraser, M.D. and Vaishnavi, V.K. (1997) "A Formal Specifications Maturity Model". In: *Communications of the ACM*, Vol. 40, No. 12, December 1997, 95-103.
- [139] Fraser, S., Booch, G., Buschmann, F., Coplien, J., Kerth, N., Jacobson, I., and Rosson M.B. (1996). "Patterns: Cult to Culture?". In: *OOPS Messenger*, 6(4):85-88, March 1996.
- [140] de Frutos-Eserig, D. (1993) "A Characterization of LOTOS Representable Networks of Parallel Processes". In: G. Scollo (Ed), *Proceedings of AMAST'93*.
- [141] Fu, Q. (2000) *Feature Interaction Detection in a Telephony Network Integrated with Switch-based Features and IN Features*. M.Sc. thesis, SITE, University of Ottawa, Canada.
http://lotos.site.uottawa.ca/ftp/pub/Lotos/Theses/qf_msc.pdf
- [142] Fu, Q., Harnois, L. Logrippo, L., and Sincennes, J. (2000) "Feature Interaction Detection: a LOTOS-Based Approach". In: *Computer Networks*, 32(4), 433-448.
- [143] Fujiwara, S., v. Bochmann, G., Khendek, F., Amalou, M., and Ghedamsi, A. (1991) "Test Selection Based on Finite State Models". In: *IEEE Transactions on Software Engineering*, 17 (6), June, 591-603.
- [144] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1997) *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- [145] Garavel, H. (1996) "An overview of the Eucalyptus Toolbox". In: *Proceedings of the COST 247 International Workshop on Applied Formal Methods in System Design*, Maribor, Slovenia, 76-88.
- [146] Garavel, H (1998) "OPEN/CAESAR: An Open Software Architecture for Verification, Simulation, and Testing". In: B. Steffen (ed) *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*.
- [147] Garlan, D., Monroe, R. T., and Wile, D. (1997) "ACME: an Architectural Interchange Language". In: *Proceedings of ICSE'97, 19th IEEE International Conference on Software Engineering*.
- [148] Garlan, D. (2000) "Software Architecture: a Roadmap". In: *22nd International Conference on Software Engineering (ICSE2000): Future of Software Engineering*, Limerick, Ireland, 93-101
- [149] Ghribi, B. (1992) *A Model Checker for LOTOS*. M.Sc. thesis, Dept. of Computer Science, University of Ottawa, Canada. <http://lotos.site.uottawa.ca/ftp/pub/Lotos/Theses/>
- [150] Ghribi, B. and Logrippo, L. (1993) "A Validation Environment for LOTOS". In: A. Danthine, G. Leduc, and P. Wolper (Eds), *Protocol Specification, Testing and Verification, XIII*, North-Holland.

- [151] Ghribi, B. and Logrippo, L. (1999) "Prototyping and Formal Requirement Validation of GPRS: A Mobile Data Packet Radio Service for GSM". In: *Seventh International Conference on Dependable Computing for Critical Applications (IFIP/IEEE)*, San Jose, CA, 99- 118.
- [152] Gischer, J. (1989) "The equational theory of pomsets". In: *Theoretical Computer Science*, 61, 199-224.
- [153] van Glabbeek, R. and Goltz, U. (1998) *Refinement of Actions and Equivalence Notions for Concurrent Systems*. Hildesheimer Informatik Bericht 6/98, Germany.
<http://theory.stanford.edu/~rvg/abstracts.html#41>
- [154] Glinz, M. (1995) "An Integrated Formal Model of Scenarios Based on Statecharts". In: *Proceedings of the 5th European Software Engineering Conference (ESEC 1995)*, Sitges, Spain.
- [155] Gomma, H. (2000) *Designing Concurrent, Distributed, and Real-Time Applications with UML*. UML Series, Addison Wesley.
- [156] Gorse, N. (2000) *The Feature Interaction Problem: Automatic Filtering of Incoherences & Generation of Validation Test Suites at the Design Stage*. M.Sc. thesis, SITE, University of Ottawa, Canada, September 2000. <http://www.UseCaseMaps.org/pub/ng-thesis.zip>
- [157] Gorse, N., Logrippo, L. and Sincennes, J. (2001) "The Feature Interaction Problem: Automatic Filtering of Incoherences & Generation of Validation Test Suites at the Design Stage". In: *Proceedings of the 6th Mitel Conference (MICON 2001)*, Ottawa, Canada, August 2001.
http://micmac.mitel.com/micon/Proceedings/Luigi_Logrippo_MICON_Proceedings.pdf
- [158] Grabowski, J., Hogrefe, D. and Nahm, R. (1993) "Test Case Generation with Test Purpose Specification by MSCs". In: O. Faergemand and A. Sarma (eds), *SDL'93 - Using Objects*, North-Holland, October 1993. http://www.itm.mu-luebeck.de/english/publications/Abstract_SDL93-SAMSTAG.html
- [159] Grégoire, J-C. and Ferguson, M.J. (1997) "Neglected Topics of Feature Interactions: Mechanisms, Architectures, Requirements". In: P. Dini *et al.*, *Feature Interactions in Telecommunications and Distributed Systems IV*, IOS Press, 3-12.
- [160] Griffeth, N.D. and Velthuijsen, N.D. (1994) "The Negotiating Agents Approach to Runtime Feature Interaction Resolution". In: L. G. Bouma and H. Velthuijsen (eds), *Second International Workshop on Feature Interactions in Telecommunications Software Systems*, IOS press, 217-235.
<http://www-db.research.bell-labs.com/user/nancyg/fiw94.ps>
- [161] Griffeth, N.D., Tadashi, O., Grégoire, J.-C. and Blumenthal, R. (1998) "First Feature Interaction Detection Contest". In: K. Kimbler and W. Bouma (eds.), *Fifth International Workshop on Feature Interactions in Telecommunications Software Systems*, IOS Press, 327-359.
<http://www.tts.lth.se:80/FIW98/contest.html>
- [162] Guillemot, R., and Logrippo, L. (1989) "Derivation of Useful Execution Trees from LOTOS Specifications by Using an Interpreter." In: K.J. Turner (Ed.) *Formal Description Techniques*. North-Holland (Proc. of the 1st FORmal TEchniques International Conference, Stirling, UK., 1988) 311-325.
<http://lotos.site.uottawa.ca/ftp/pub/Lotos/Papers/forte88.am.ps.Z>
- [163] Guttag, J.V. and Horning, J.J. (1993) *Larch: Languages and Tools for Formal Specification*. Springer-Verlag.
- [164] Hackathorn, R. (1997) "Data Delivery When You Want It". In: *BYTE*, vol. 22, no. 6, June 1997.

-
- [165] Haj-Hussein, M., Logrippo, L. and Sincennes, J. (1993) "Goal Oriented Execution for LOTOS". In: M. Diaz and R. Groz (Eds), *Formal Description Techniques*, V, North-Holland, 311-327.
- [166] Hall, A. (1990) "Seven Myths of Formal Methods". In: *IEEE Software*, 7(5), September, 11-19.
- [167] Harel, D. (2000) "From Play-In Scenarios To Code: An Achievable Dream". In: *Fundamental Approaches to Software Engineering (FASE2000)*, LNCS 1783, Springer-Verlag, 22-34.
http://www.wisdom.weizmann.ac.il:81/Dienst/UI/2.0/Describe/ncstrl.weizmann_il/MCS00-06
- [168] Harel, D. and Gery, E. (1996) "Executable Object Modeling with Statecharts". In: *Proceedings of the 18th International Conference on Software Engineering*, Berlin, IEEE Press, March 1996, 246-257.
- [169] Harel, D. and Kugler, H. (2000) "Synthesizing State-Based Object Systems from LSC Specifications". In: *Fifth International Conference on Implementation and Application of Automata (CIAA 2000)*, LNCS, Springer-Verlag.
- [170] Hassine, J. (2001) *Feature Interaction Filtering and Detection with Use Case Maps and LOTOS*, M.Sc. thesis, University of Ottawa, Canada, February 2001.
http://lotos.site.uottawa.ca/ftp/pub/Lotos/Theses/jh_msc.pdf
- [171] Heerink, L. (1998) *Ins and Outs in Refusal Testing*. Ph.D. thesis, University of Twente, Enschede, The Netherlands.
- [172] Hekmatpour, S. and Ince, D. (1988) *Software Prototyping, Formal Methods and VDM*. Addison Wesley.
- [173] H elou et, L. and Jard, C. "Conditions for synthesis of communicating automata from HMSCs". In: *5th International Workshop on Formal Methods for Industrial Critical Systems*, Berlin, April 2000.
<http://www.fokus.gmd.de/research/cc/tip/fmics/abstracts/helouet.html>
- [174] Hennessy, M. (1988) *Algebraic Theory of Processes*. Foundations of Computing, MIT Press, Cambridge, USA.
- [175] Herbsleb, J., Zubrow, D., Goldenson, D., Hayes, W., and Paulk, M. (1997) "Software Quality and the Capability Maturity Model". In: *Communications of the ACM*, Vol. 40, No. 6, June 1997, 31-40.
- [176] Hinchey, M.G. and Bowen, J.P. (1995) "Applications of Formal Methods FAQ". In M. G. Hinchey and J. P. Bowen (eds): *Applications of Formal Methods*. Prentice-Hall International Series in Computer Science, September 1995, 1-15.
- [177] Hoare, C. A. R. (1985) *Communicating Sequential Processes*. Prentice-Hall International, U.K.
- [178] Hodges, J. and Visser, J. (1999) "Accelerating Wireless Intelligent Network Standards Through Formal Techniques". In: *IEEE 1999 Vehicular Technology Conference (VTC'99)*, Houston (TX), USA.
<http://www.UseCaseMaps.org/pub/vtc99.pdf>
- [179] Hogrefe, D., Heymer, S., and Tretmans, J. (1996) "Report on the standardization project 'Formal methods in conformance testing'". In: *9th International Workshop on Testing of Communicating Systems (IWTC'S'96)*, Darmstadt, Germany. Chapman & Hall, 289-298.
- [180] Holzmann, G.J. (1991) *Design and Validation of Computer Protocols*, Prentice Hall.

- [181] Holzmann, G.J. (1994) "The theory and practice of a formal method: NEWCORE". In: *Proceedings of the 13th IFIP World Computer Congress*, Hamburg, Germany.
- [182] Holzmann, G.J., Peled, D., and Redberg, M. (1997) "Design tools for requirements engineering". In: *Bell Labs Technical Journal*, 2(1):86-95.
http://www.lucent.com/minds/techjournal/winter_97/pdf/paper07.pdf
- [183] Hsia, P., Samuel, J., Gao, J., Kung, D., Toyoshima, Y. and Chen, C. (1994) "Formal Approach to Scenario Analysis". *IEEE Software*, 1994, 33-40.
- [184] Hurlbut, R. (1997) *A Survey of Approaches for Describing and Formalizing Use Cases*. Technical Report 97-03, Department of Computer Science, Illinois Institute of Technology, USA.
<http://www.iit.edu/~rhurlbut/xpt-tr-97-03.html>
- [185] Hurlbut, R. R. (1998) *Managing Domain Architecture Evolution Through Adaptive Use Case and Business Rule Models*. Ph.D. thesis, Illinois Institute of Technology, Chigago, USA.
<http://www.iit.edu/~rhurlbut/hurl98.pdf>
- [186] IBM, Unisys *et al.* (1998) *XMI (XML Metadata Interchange) Proposal*. OMG document ad/98-10-05, October 1998. <http://www.software.ibm.com/ad/features/xmi.html>
- [187] IEEE (1990) *Standard Glossary of Software Engineering Terminology*. IEEE Std 610.12-1990, New York, USA.
- [188] IEEE (1993) *Recommended Practice for Software Requirements Specifications*. IEEE Std 830-1993, New York, USA.
- [189] IEEE (1995) *Guide for Developing System Requirements Specifications*. IEEE Std P1233/D3, New York, USA.
- [190] IFAD (1999) *VDMTools*. <http://www.ifad.dk/Products/VDMTools>
- [191] ISO (1989), Information Processing Systems, Open Systems Interconnection, *LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, IS 8807, Geneva.
- [192] ISO (1989), Information Processing Systems, Open Systems Interconnection, *Estelle — A Formal Description Technique Based on an Extended State Transition Model*, IS 9074, Geneva.
- [193] ISO/EIC (1991) Information Technology, Open Systems Interconnection, *Conformance Testing Methodology and Framework (CTMF)*, IS 9646, Geneva. Also: CCITT X.290-X.294.
- [194] ISO/IEC (1994) Information Technology, Open Systems Interconnection, *Basic Reference Model. The Basic Model*. IS 7498-1: 1994, Geneva
- [195] ISO/ITU-T (1995) *Open Distributed Processing, Reference Model*, ISO 10746, ITU Recommendation X.901-904, Geneva.
- [196] ISO/EIC (1996) *Proposed ITU-T Z.500 and Committee Draft on "Formal Methods in Conformance Testing" (FMCT)*. ISO/EIC JTC1/SC21/WG7, ITU-T SG 10/Q.8, CD-13245-1, Geneva.
- [197] ISO/EIC (1997) *OSI CTMF Part 3: The Tree and Tabular Combined Notation — Second Edition*, IS 9646-3: 1997, Geneva.
- [198] ISO/IEC (1998) *Enhancements to LOTOS (E-LOTOS)*. FCD 1998-10-03, DIS 15437, Geneva.

-
- [199] ISO/IEC (1999) *High Level Petri Net Standard*, DIS 15909, JTC 1/SC 7, Geneva.
- [200] ITU (1988) *Recommendation I.130, Method for the characterization of telecommunication services supported by an ISDN and network capabilities of ISDN*. CCITT, Geneva.
- [201] ITU (1994) *Recommendation X.680-683, Abstract Syntax Notation One (ASN.1)*. Geneva.
- [202] ITU (1995) *Q.1200 General Series, Intelligent Networks Recommendation Structure*. Geneva.
- [203] ITU-T, Study Group 10 (1999) *Proposal for a new question to define a notation for user requirements*. Canadian contribution, COM10-D56, November 1999.
http://www.UseCaseMaps.org/pub/Q12_URN.pdf
- [204] ITU (2000) *Recommendation Q.65, The unified functional methodology for the characterization of services and network capabilities including alternative object-oriented techniques*. Geneva.
- [205] ITU (2000) *Recommendation Z.100, Specification and Description Language (SDL)*. Geneva.
- [206] ITU (2000) *Recommendation Z.105, SDL Combined with ASN.1 (SDL/ASN.1)*. Geneva.
- [207] ITU (2000) *Recommendation Z.109, SDL combined with UML*. Geneva.
- [208] ITU (2000) *Recommendation Z. 120: Message Sequence Chart (MSC)*. ITU, Geneva.
- [209] ITU (2001) *Recommendation Z. 140: The Tree and Tabular Combined Notation version 3 (TTCN-3): Core language*. ITU, Geneva.
- [210] Jackson, M. (1995) *Software Requirements & Specifications — a lexicon of practice, principles and prejudices*. Addison-Wesley, ACM Press.
- [211] Jackson, M. (1998) “Formal Methods and Traditional Engineering”. In: *J. Systems Software*, 40, 191-194
- [212] Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. (1993) *Object-Oriented Software Engineering, A Use Case Driven Approach*. Addison-Wesley, ACM Press.
- [213] Jagadeesan, L.J., Votta, L.G., Porter, A., Puchol, C., and Ramming, J.C. (1998) “Specification-based Testing of Reactive Software: A Case Study in Technology Transfer”. In: *Journal of Systems Software*, 40, 249-262
- [214] Jard, C., Jéron, T. and Morel, P. (2000) “Verification of Test Suites”. In: H. Ural, R.L. Probert and G.v. Bochmann (eds) *Testing of Communicating Systems: Tools and Techniques (TestCom 2000)*. Kluwer Academic Publishers, 3-18.
- [215] Jarke, M. and Kurki-Suonio, R., editors. (1998) *IEEE Transactions on Software Engineering, Special Issue on Scenario Management*. Vol. 24, No. 12, December 1998.
- [216] Jensen, K. (1992) *Coloured Petri Nets — Basic Concepts, analysis Methods and Practical Use — Volume 1: Basic Concepts*. Monographs in Theoretical Computer Science, Springer-Verlag, Berlin.
- [217] Jéron, T. and Morel, P. (1999) “Test Generation Derived from Model-Checking”. In: N. Halbwachs and D. Peled (eds) *CAV’99*. LNCS 1633, Springer, 108-122.

- [218] Johnson, P. M. (1998) "Reengineering inspection". In: *Communications of the ACM*, Vol. 41, No. 2, February 1998, 49-52.
- [219] Jones, C.B. (1986) *Systematic Software Development Using VDM*. Prentice-Hall, London, U.K.
- [220] Jones, S., Till, D., and Wrightson, A.M. (1998) "Formal Methods and Requirements Engineering: Challenges and Synergies". In: *J. Systems Software*, 40, 263-273.
- [221] Joyce, D. (1997) "Code Coverage Analysis Works In Hardware Design". In: *Integrated System Design*, January. <http://www.isdmag.com/editorial/1997/edafeature9701.html>.
- [222] Kamoun, J. (1996) *Formal Specification and Feature Interaction Detection in the Intelligent Network*. M.Sc. thesis, SITE, University of Ottawa, Canada. http://lotos.site.uottawa.ca/ftp/pub/Lotos/Theses/jk_msc.ps.gz
- [223] Kamoun, J., and Logrippo, L. (1998) "Goal-Oriented Feature Interaction Detection in the Intelligent Network Model". In K. Kimbler and L. G. Bouma (Eds), *Fifth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'98)*, Lund, Sweden, September 1998. IOS Press, 172-186.
- [224] Karlsson, J., Wohlin, C., and Regnell, B. (1998) "An evaluation of methods for prioritizing software requirements". In: *Information and Software Technology*, 39, 939-947. <http://www.tts.lth.se/Personal/bjornr/Papers/IST98.pdf>
- [225] Katoen, J.-P. and Lambert, L. (1998) "Pomsets for Message Sequence Charts". In: *First Workshop of the SDL Forum Society on SDL and MSC*, Berlin, Germany, June, 291-300.
- [226] Keck, D.O. (1998) "A Tool for the Identification of Interaction-Prone Call Scenarios". In: K. Kimbler and L. G. Bouma (Eds), *Fifth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'98)*, Lund, Sweden, September 1998. IOS Press, 276-290.
- [227] Keck, D.O. and Kuehn, P.J. (1998) "The Feature and Service Interaction Problem in Telecommunications Systems: A Survey". In: *IEEE Transactions on Software Engineering*, Vol. 24, No. 10, October 1998, pp. 779-795.
- [228] Khendek, F. and Vincent, D. (2000) "Enriching SDL Specifications with MSCs". In: *2nd Workshop of the SDL Forum Society on SDL and MSC (SAM2000)*, Grenoble, France, June 2000.
- [229] Kimbler, K. and Søbirk, D. (1994) "Use case driven analysis of feature interactions". In: L. G. Bouma and H. Velthuisen (eds), *Feature Interactions in Telecommunications Systems*, Amsterdam, The Netherlands, May 1994. IOS Press, 167-177.
- [230] Kimbler, K. and Bouma, L. G. (1998) *Fifth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'98)*, Lund, Sweden, September 1998. IOS Press.
- [231] Koo, I. (1996) *Mutation Testing and Three Variations*. Technical Report, University of British Columbia, Canada. <http://www.ee.ubc.ca/home/comlab1/irenek/etc/www/techpaps/mutate/mutation.html>
- [232] Koskimies, K. and Mäkinen, E. (1994) "Automatic Synthesis of State Machines from Trace Diagrams". In: *Software Practice and Experience*, 24(7), July 1994, 643-658.

-
- [233] Koskimies, K., Männistö, T., Systä, T., and Tuomi, J. (1996) *SCED: A tool for dynamic modelling of object systems*. University of Tampere, Department of Computer Science, Report A-1996-4, July.
<ftp://cs.uta.fi/pub/reports/A-1996-4.ps.Z>
- [234] Kropp, N.P., Koopman, P.J., and Siewiorek D.P. (1998) "Automated Robustness Testing of Off-the-Shelf Software Components". In: *Proceedings of FTCS'98*, Munich, Germany.
<http://www.cs.cmu.edu/~koopman/ballista/ftcs98/ftcs98.pdf>
- [235] Krüger, I., Grosu, R., Scholz, P. and Broy, M. (1999) "From MSCs to Statecharts". In: *Distributed and Parallel Embedded Systems*, Kluwer Academic Publishers.
<http://www4.informatik.tu-muenchen.de/papers/KGSB99.html>
- [236] Ladkin, P.B., and Leue, S. (1995) "Four issues concerning the semantics of Message Flow Graphs". In: D. Hogrefe and S. Leue (Eds), *Formal Description Techniques, VII, Proceedings of the Seventh IFIP International Conference on Formal Description Techniques FORTE'94*, Chapman & Hall.
<http://sven.uwaterloo.ca:80/~sleue/publications.files/forte94.ps.Z>
- [237] Lai, R. (1996) "How could research on testing of communicating systems become more industrially relevant?". In: *9th International Workshop on Testing of Communicating Systems (IWTCS'96)*, Darmstadt, Germany, 3-13.
- [238] van Lamsweerde, A. and Willemet, L. (1998) "Inferring Declarative Requirements Specifications from Operational Scenarios". In: *IEEE Transactions on Software Engineering, Special Issue on Scenario Management*. Vol. 24, No. 12, December 1998, 1089-1114.
- [239] Langerak, R. (1992) *Transformations and Semantics for LOTOS*. Ph.D. thesis, Department of Computer Science, University of Twente, The Netherlands.
- [240] Lea, D. (1994) "Christopher Alexander: An Introduction for Object-Oriented Designers". In: *Software Engineering Notes*, January 1994. <http://gee.cs.oswego.edu/dl/ca/ca/ca.html>.
- [241] Le Charnier, B. and Flener, P. (1998) "Specifications Are Necessarily Informal or: Some More Myths of Formal Methods". In: *Journal of Systems Software*, 40, 275-296
- [242] Leduc, G. (1991) "Conformance relation, associated equivalence, and minimum canonical tester in LOTOS". In: B. Jonsson, J. Parrow, and B. Pehrson (Eds.), *Protocol Specification, Testing and Verification XI*, Elsevier Science Publishers B.V., 249-264.
- [243] Leduc, G. (1994) "Failure-based Congruences, Unfair Divergences and New Testing Theory". In: S.T. Vuong and S.T. Chanson (Eds), *Protocol Specification, Testing, and Verification, XIV*, Vancouver, Canada. Chapman & Hall, 252-267.
- [244] Leue, S., Mehrmann, L. and Rezai, M. (1998) "Synthesizing ROOM Models from Message Sequence Chart Specifications". Technical Report 98-06, Department of Electrical and Computer Engineering, University of Waterloo, Canada, April 1998. Short paper version in: *13th IEEE Conference on Automated Software Engineering*, Honolulu, Hawaii, October 1998.
<http://sven.uwaterloo.ca:80/~sleue/publications.files/tr98-06.ps.gz>
- [245] Leyton, M. (1999) *Symmetry, Causality, Mind*. MIT Press.
- [246] Li, J.J. and Horgan, J.R. (2000) "Applying formal description techniques to software architectural design". In: *Computer Communications*, 23(12), 1169-1178.

- [247] Liu, L. and Yu, E. (2001) "From Requirements to Architectural Design - Using Goals and Scenarios". In: *From Software Requirements to Architectures Workshop (STRAW 2001)*, Toronto, Canada, May 2001. <http://www.UseCaseMaps.org/pub/straw01.pdf>
- [248] Liu, M.T. (1989) "Protocol Engineering". In: *Advances in Computers*, Vol. 29, 79-195.
- [249] Lucent Technologies (1999) *uBET — Lucent Behavior Engineering Toolset*. <http://cm.bell-labs.com/cm/cs/what/ubet/>
- [250] Luqi and Goguen, J.A. (1997) "Formal Methods: Promises and Problems". In: *IEEE Software*, January 1997, 73-85.
- [251] Lyngsø, R.B. and Mailund, T. (1998) "Textual Interchange Format for High-level Petri Nets". In: *Proceedings of the 1998 Workshop on Practical Use of Coloured Petri Nets and Design/CPN*, June 8-12, Aarhus, Denmark. See also <http://www.daimi.aau.dk/designCPN/man/Misc/textformat.pdf>
- [252] Maiden, N.A.M. (1998) "SAVRE: Scenarios for Acquiring and Validating Requirements". In: *Journal of Automated Software Engineering*, 5, 419-446.
- [253] Maier, C. and Maximilians, L. (1997) "Object Coloured Petri Nets — a Formal Technique for Object Oriented Modelling". In: *Workshop PNSE'97 Petri Nets in System Engineering Modelling, Verification, and Validation*, Hamburg, Germany, September 25-26.
- [254] Mäkinen, E. and Systä, T. (2001) "MAS – An Interactive Synthesizer to Support Behavioral Modeling in UML". In: *23rd International Conference on Software Engineering (ICSE'01)*, Toronto, Canada, May 2001.
- [255] Mansurov, N. and Zhukov, D. (1999) "Automatic synthesis of SDL models in use case methodology". In: *SDL'99, Proceedings of the Ninth SDL Forum*, Montréal, Canada. Elsevier.
- [256] Meszaros, G. and Doble, J. (1998) "A Pattern Language for Pattern Writing". In: *Pattern Languages of Program Design 3*, Addison-Wesley, 529-574. <http://hillside.net/patterns/Writing/patterns.html>
- [257] Miga, A. (1998) *Application of Use Case Maps to System Design with Tool Support*. M.Eng. thesis, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada. http://www.UseCaseMaps.org/pub/am_thesis.pdf
- [258] Miga, A., Amyot, D., Bordeleau, F., Cameron, C. and Woodside, M. (2001) "Deriving Message Sequence Charts from Use Case Maps Scenario Specifications". In: *Tenth SDL Forum (SDL'01)*, Copenhagen, Denmark, June 2001. <http://www.UseCaseMaps.org/pub/sdl01-miga.pdf>
- [259] Mills, H.D., Dyer, M., and Linger, R.C., (1987) "Cleanroom Software Engineering". In: *IEEE Software*, September 1987, 19-24.
- [260] Milner, R. (1989) *Communication and Concurrency*. Addison-Wesley, Reading, Massachusetts, USA.
- [261] Monkewich, O. (2001) *ITU-T Draft Recommendation A.3 Supplement 1: Guidelines on the quality aspects of Protocol related Recommendations*. Canadian Contribution to ITU-T Study Group 10, April 2001.
- [262] Monkewich, O., Sales, I. and Probert, R. (2001) "OSPF Efficient LSA Refreshment Function in SDL". In: *Tenth SDL Forum (SDL'01)*, Copenhagen, Denmark, June 2001. <http://www.UseCaseMaps.org/pub/sdl01-sales.pdf>

-
- [263] Moreira, A. M. D., and Clark, R. G. (1996) "Adding rigour to object-oriented analysis". In: *Software Engineering Journal*, IEE, 11(5), September 1996, 270-280.
- [264] Mouly, M. and Pautet, M.-B (1992) *The GSM System for Mobile Communications*. Cell & Sys.
- [265] Musa, J.D., and Ackerman, A.F. (1989) "Quantifying Software Validation: When to Stop Testing?". In: *IEEE Software*, May 1989.
- [266] Mussbacher, G. and Amyot, D. (2001) "A Collection of Patterns for Use Case Maps". In: *First Latin American Conference on Pattern Languages of Programming (SugarLoafPLoP 2001)*, Rio de Janeiro, Brazil, October 2001.
- [267] Myers, G. J. (1979) *The Art of Software Testing*. Wiley-Interscience, New-York.
- [268] Nakamura, M., Kikuno, T., Hassine, J., and Logrippo, L. (2000). "Feature Interaction Filtering with Use Case Maps at Requirements Stage". In: *Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'00)*, Glasgow, Scotland, UK, May 2000. <http://www.UseCaseMaps.org/pub/fiw00filter.pdf>
- [269] Nielsen, M., Plotkin, G.D., and Winskel, G. (1981) "Petri nets, event structures and domains, part I". In: *Theoretical Computer Science*, 13(1), 85-108.
- [270] Nursimulu, K. and Probert, R. (1995) "Cause-Effect Graphing Analysis and Validation Requirements". Department of Computer Science, University of Ottawa, Canada, TR-95-14 (June).
- [271] Offutt, A.J. (1992) "Investigations of the Software Testing Coupling Effect". In: *ACM Transactions on Software Engineering and Methodology*, Vol. 1, No. 1, January, 5-20.
- [272] Offutt, A.J., Rothermel, G., and Zapf, C. (1993) "An Experimental Evaluation of Selective Mutation". In: *15th International Conference on Software Engineering (ICSE'93)*, 100-107.
- [273] OMG (1995) *The Common Object Request Broker: Architecture and Specification, Version 2.0*. <http://www.omg.org>
- [274] OMG (1999) *Unified Modeling Language Specification, Version 1.3*. June 1999. <http://www.omg.org>
- [275] Parnas, D.L. (1994) "Using Mathematical Models in the Inspection of Critical Software". In: Craigen, D., Gerhart, S., and Ralston, T. (eds) *Industrial applications of formal methods to model, design, and analyze computer systems: an international survey*. Noyes Data Corporation, USA, 17-31.
- [276] Parnas, D. L., Madey, J., and Iglewski, M. (1994) "Precise Documentation of Well-Structured Programs". In: *IEEE Transactions on Software Engineering*, Volume 20 Number 12 (December), 948-976.
- [277] Parnas, D.L. (1998) "Formal Methods Technology Transfer Will Fail". In: *Journal of Systems Software*, 40, 195-198.
- [278] Paulk, M., Curtis, B., Chrissis, M.B., and Weber, C. (1993) *Software Capacity Maturity Model, Version 1.1*. Software Engineering Institute, CMU/SEI-93-TR-25 (February).
- [279] Pavón, S. and Llamas, M. (1991) "The testing Functionalities of LOLA". In: J. Quemada, J.A. Mañas, and E. Vázquez (Eds), *Formal Description Techniques, III*, IFIP/North-Holland, 559-562.
- [280] Pavón, S., Larrabeiti, D., and Rabay, G. (1995) *LOLA—User Manual, version 3.6*. DIT, Universidad Politécnica de Madrid, Spain, LOLA/N5/V10 (February).

- [281] Peterson J. (1977) "Petri Nets". In: *ACM Computing Surveys*, 9(3), September, 223-252.
- [282] Petrenko, A. (1998) "Modeling Faults in Object State Machines". In: *ObjecTime Workshop on Research in OO Real-Time Modeling*, Ottawa, Canada, January 1998.
- [283] Petriu, D.B. (2001) *Layered Software Performance Models Constructed from Use Case Map Specifications*. M.Eng. thesis, Dept. of Systems and Computer Eng., Carleton University, Ottawa, Canada.
- [284] Phalippou, M. (1994) *Relations d'implantation et hypothèses de test sur des automates à entrées et sorties*. Ph.D. thesis, Université de Bordeaux I, France.
- [285] Piatkowski, T.F. (1980) "An engineering discipline for distributed protocol systems". In: *Proceedings of the NATO Advanced Study Institute: New Concepts in Multi-User Communication*, Norwich, UK, August 1980.
- [286] Pires, L.F. (1994) Architectural Notes: a Framework for Distributed Systems Development. CTIT Ph.D. thesis 94-01, Twente University, The Netherlands.
<http://wwwhome.cs.utwente.nl/~pires/thesis/index.html>
- [287] Poston, R.M. (1996) *Automating specification-based software testing*. IEEE Computer Society Press, Los Alamitos, CA, USA.
- [288] Potts, C., Takahashi, K., and Antòn, A.I. (1994) "Inquiry-Based Requirements Analysis". In: *IEEE Software*, March 1994, 21-32.
- [289] Pratt, V. (1986) "Modeling concurrency with partial orders". In: *International Journal of Parallel Programming*, 15, 33-71.
- [290] Pressman, R. S. (1997) *Software Engineering — A Practitioner's Approach*. Fourth edition. McGraw-Hill, USA.
- [291] Probert, R.L. (1982) "Optimal Insertion of Software Probes in Well-Delimited Programs", *IEEE Transactions on Software Engineering*, Vol 8, No 1, January 1982, 34-42.
- [292] Probert, R.L. and Guo, F. (1991) "Mutation Testing of Protocols: Principles and Preliminary Experimental Results". In I. Davidson and D.W. Litwack (eds), *Protocol Test Systems III (IWPTS'91)*, North-Holland, 57-76.
- [293] Probert, R.L. and Saleh, J. (1991) "Synthesis of communications protocols: survey and assessment". In: *IEEE Transactions on Computers*, Vol. 40, No. 4, April 1991, 468-476.
- [294] Probert, R.L. and Monkewich, O. (1992) "TTCN: the international notation for specifying tests of communications systems". In: *Computer Networks and ISDN Systems*, 23 (05), 417-438.
- [295] Probert, R.L. and Wei, L. (1995) "Towards a 'Practical Formal Method' for Test Derivation". In: *Protocol Test Systems VIII (IWPTS'95)*, Evry, France. 433-448.
- [296] Probert, R.L. and Lew, N. (1996) "Protocol quality engineering: addressing industry concerns about formal methods". *Computer Communications* 19, 1258-1267.

-
- [297] Probert, R.L. and Williams, A.W. (1999) "Fast Functional Test Generation using an SDL model". In: *Proceedings of the 12th annual International Workshop on the Testing of Communicating Systems (IWTCS'99)*, Budapest, Hungary, September 1999, 299-315.
<http://www.site.uottawa.ca/~awilliam/papers/iwtcs.ps>
- [298] Probert, R.L, Ural, H., and Williams, A.W. (2001) "Rapid generation of functional tests using MSCs, SDL and TTCN". In: *Computer Communications*, Vol. 24, No. 3-4, February 15, 2001, 374-393.
- [299] Rising, L. (1999) "Patterns: A Way to Reuse Expertise". In: *IEEE Communications Magazine*, Vol. 37, No. 4, April 1999.
- [300] Quartel, D.A.C. (1998) *Action relations. Basic design concepts for behaviour modelling and refinement*. CTIT Ph.D thesis series, no. 98-18, University of Twente, Enschede, The Netherlands.
<http://wwwhome.cs.utwente.nl/~quartel/publications/PhD/index.html>
- [301] Quemada, J., Pavón, S. and Fernández, A. (1988) "Transforming LOTOS Specifications with LOLA: The Parametrized Expansion". In: K. J. Turner (Ed), *Formal Description Techniques, I*, IFIP/North-Holland, 45-54.
- [302] Quemada, J., Azcorra, A., and Pavón, S. (1995) "The LOTOSphere design methodology". In [57], 29-58.
- [303] Rational Software (1998) *Rational Unified Process 5.0*, Cupertino, CA, USA.
- [304] Regnell, B., Kimbler, K., and Wesslén, A. (1995) "Improving the Use Case Driven Approach to Requirements Engineering". In: *Proceedings of Second IEEE International Symposium on Requirements Engineering*, York, U.K., March 1995, 40-47.
<http://www.tts.lth.se/Personal/bjornr/Papers/tts-94-24.ps>
- [305] Regnell, B., and Runeson, P. (1998) "Combining Scenario-based Requirements with Static Verification and Dynamic Testing". In: E. Dubois, A. L. Opdahl, and K. Pohl (Eds.), *Proceedings of the Fourth International Workshop on Requirements Engineering — Foundations for Software Quality (REFSQ'98)*, Pisa, Italy, June 1998. <http://www.tts.lth.se/Personal/bjornr/Papers/REFSQ98.pdf>
- [306] Regnell, B. (1999) *Requirements Engineering with Use Cases — a Basis for Software Development*. Ph.D. Thesis, Department of Communication Systems, Lund Institute of Technology, Sweden.
<http://www.tts.lth.se/Personal/bjornr/thesis/>
- [307] Regnell, B., Runeson, P., and Wohlin, C. (1999) "Towards Integration of Use Case Modelling and Usage-Based Testing". *Journal of Systems and Software*, Elsevier.
<http://www.tts.lth.se/Personal/bjornr/thesis/JSS99.pdf>
- [308] RENOIR (1996) *Requirements Engineering Network Of International cooperating Research groups — a network of excellence*. ESPRIT project 20.800, <http://www.cs.ucl.ac.uk/research/renoir/>
- [309] Richardson, D.J., O'Malley, O, and Tottle, C. (1989) "Approaches to Specification-Based Testing". In: R.A. kemmerer (Ed), *Software Engineering Notes*, Vol. 14, No. 8, 86-96, December 1989.
- [310] Rising, L. (ed.) (2001) *Design patterns in communications software*. Cambridge University Press.

- [311] Rolland, C., Ben Achour, C., Cauvet, C., Ralyte, J., Sutcliffe, A.G., Maiden, N.A.M., Jarke, M., Haumer, P., Pohl, K., Dubois, E., and Heymas, P. (1998) "A proposal for a Scenario Classification Framework". In: *Requirements Engineering Journal*, 3(1), 23-47.
- [312] Rolland, C., Souveyet, C. and Ben Achour, C. (1998) "Guiding Goal Modelling using Scenarios". In: *IEEE Transactions on Software Engineering, Special Issue on Scenario Management*. Vol. 24, No. 12, December 1998.
- [313] Rudolph, E., Graubmann, P., and Grabowski, J. (1996) "Tutorial on Message Sequence Charts". In: *Computer Networks and ISDN Systems*, 28, 1629-1641.
- [314] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorenson, W. (1991) *Object-Oriented Modeling and Design*. Prentice-Hall.
- [315] Saleh, K. (1996) "Synthesis of communications protocols: an annotated bibliography". In: *ACM SIGCOMM Computer Communications Review*, Vol.26 , No.5, October, 40-59.
- [316] Saleh, K. (1998) "Synthesis of protocol converters: an annotated bibliography". In: *Computer Standards & Interfaces*, 19, 105-117.
- [317] Sales, I. and Probert, R.(2000) "From High-Level Behaviour to High-Level Design: Use Case Maps to Specification and Description Language". In: *SBRC'2000, 18^o Simpósio Brasileiro de Redes de Computadores*, Belo Horizonte, Brazil, May 2000. <http://www.UseCaseMaps.org/pub/sbr00.pdf>
- [318] Sales, I. (2001) *A Bridging Methodology for Internet Protocols Standards Development*. M.Sc. thesis, SITE, University of Ottawa, Canada, 2001. <http://www.UseCaseMaps.org/pub/is-thesis.zip>
- [319] Saïdouni, D.-E. (1996) *Sémantique de maximalité : Application au raffinement d'actions dans LOTOS*. Ph.D. thesis, LAAS, report 96098. Université Paul Sabatier, Toulouse, France.
- [320] Schmidt, D.C. (1994) "The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software". In: *Sun User Group conference*, 1994.
- [321] Schmidt, D.C. (1996) "Object-Oriented Design Patterns for Concurrent, Parallel, and Distributed Systems". Invited talk, University of Carleton, march 1996.
- [322] Schönberger, S., Keller, R.K., and Khriiss, I. (1999) "Algorithmic Support for Model Transformation in Object-Oriented Software Development". In: *Theory and Practice of Object Systems (TAPOS)*. John Wiley and Sons. To appear. <ftp://ftp.iro.umontreal.ca/pub/gelo/Publications/Papers/tapos99.pdf>
- [323] van der Schoot, H. and Ural, H. (1997) "Data Flow Analysis of System Specifications in LOTOS". In: *International Journal of Software Engineering and Knowledge Engineering*, Vol.7, No. 1, 43-68.
- [324] Scratchley, W.C. and Woodside, C.M. (1999) "Evaluating Concurrency Options in Software Specifications". In: *MASCOTS'99, Seventh International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, College Park, MD, USA, October 1999, 330-338. <http://www.UseCaseMaps.org/pub/mascots99.pdf>
- [325] Scratchley, W.C. (2000) *Evaluation and Diagnosis of Concurrency Architectures*. Ph.D. thesis, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada, June 2000. <http://www.UseCaseMaps.org/pub/scratchley-thesis.pdf>

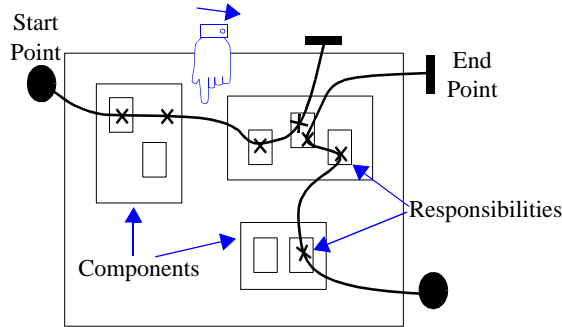
-
- [326] Selic, B., Gullekson, G., and Ward, P.T. (1994) *Real-Time Object-Oriented Modeling*, Wiley & Sons.
- [327] Shankland, C., Thomas, M., and Brinksma, E. (1997) "Symbolic Bisimulation for Full LOTOS". In: *Lecture Notes in Computer Science*, 1349, Springer-Verlag, 479-493.
<http://www.cs.stir.ac.uk/~ces/Papers/AMAST97.ps>
- [328] Sherer, S.A. (1991) "A Cost-Effective Approach to Testing". In: *IEEE Software*, March 1991.
- [329] Siegel, S. (1996) *Object-oriented software testing: a hierarchical approach*. John Wiley & Sons.
- [330] Smith, C.U. and Woodside, C.M. (1999) "Performance Validation at Early Stages of Development". In: *Performance 99*, Istanbul, Turkey, October.
- [331] Smith, M.H., Holzmann, G.J., and Etesami, K. (2001) "Events and Constraints: A Graphical Editor for Capturing Logic Requirements of Programs". In *RE'01, Fifth IEEE Int. Symposium on Requirements Engineering*, Toronto, Canada, August 2001, 14-22.
- [332] Somé, S., Dssouli, R., and Vaucher, J. (1996) "Un cadre pour l'ingénierie des exigences avec des scénarios". In: Bennani, A., Dssouli, R., Benkiran, A., and Rafiq, O. (Eds), *CFIP 96, Ingénierie des protocoles*, ENSIAS, Rabat, Maroc.
- [333] Somé, S., Dssouli, R., and Vaucher J. (1996) "Toward an Automation of Requirements Engineering using Scenarios". In: *Journal of Computing and Information*, 2(1), 1110-1132.
- [334] Somé, S. (1997) *Dérivation de Spécifications à partir de Scénarios d'interaction*. Ph.D. thesis, Département d'IRO, Université de Montréal, Canada.
- [335] Sommerville, I. (1992) *Software Engineering, Fourth Edition*. Addison-Wesley
- [336] Spivey, J.M. (1992) *The Z Notation: A Reference Manual*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, second edition, 1992.
- [337] Steedman, D (1990). *Abstract Syntax Notation One ASN.1: The Tutorial & Reference*. Technology Appraisals, Twickenham, UK.
- [338] Stépien, B. and Logrippo, L. (1995) "Feature Interaction Detection using Backward Reasoning with LOTOS". In: S. Vuong (ed.), *Protocol Specification, Testing and Verification XIV*, Vancouver, 71-86.
<http://lotos.site.uottawa.ca/ftp/pub/Lotos/Papers/pstv.94.book.ps.Z>
- [339] Stépien, B. and Logrippo, L. (1995) "Representing and Verifying Intentions in Telephony Features using Abstract Data Types". In: K. E. Cheng and T. Ohta (eds.), *Third International Workshop on Feature Interactions in Telecommunications Software Systems*, IOS Press, 141-155.
<http://lotos.site.uottawa.ca/~bernard/intention.ps.Z>
- [340] Sutcliffe, A.G., Maiden, N.A.M., Minocha, S., and Manuel, D. (1998) "Supporting Scenario-Based Requirements Engineering". In: *IEEE Transactions on Software Engineering, Special Issue on Scenario Management*. Vol. 24, No. 12, December 1998, 1072-1088.
- [341] Systä, T. Keller, R. and Koskimies, K. (2001) "Scenario-based Round-trip Engineering, OOPSLA 2000 Workshop summary". In: *SIGSOFT Software Engineering Notes*, March 2001, vol 26, no 2.
- [342] Telelogic (1999) *Tau Tool*. <http://www.telelogic.com/solution/tools/tau.asp>

- [343] Thomas, M. (1997) “Modelling and Analysing User Views of Telecommunications Services”. In: Dini, P., Boutaba, R., and Logrippo, L. (Eds.) *Feature Interactions in Telecommunications and Distributed Systems IV*, IOS Press.
- [344] TIA/EIA (1998) *Wireless Intelligent Networks (WIN). Additions and modifications to ANSI-41 (Phase I)*. TR-45.2.2.4, PN-3661 Ballot Version, May 1998.
- [345] Tretmans, J. (1989) “Test Case Derivation from LOTOS Specifications”. In S. T. Vuong (Ed), *Formal Description Techniques II*. North-Holland, 345-360, December 1989.
- [346] Tretmans, J. (1993) “A formal approach to conformance testing”. In: *Proceedings of the 6th International Workshop on Protocol Test Systems*, Pau, France, September 1993.
- [347] Tretmans, J. (1999) “Testing Concurrent Systems: A Formal Approach”. In: *10th Int. Conference on Concurrency Theory (CONCUR'99)*. LNCS 1664, Springer-Verlag, 46-65.
- [348] Tuok, R. (1996) *Modeling and Derivation of Scenarios for a Mobile Telephony System in LOTOS*. M.Sc. thesis, Dept. of Computer Science, University of Ottawa, Ottawa, Canada.
http://lotos.site.uottawa.ca/ftp/pub/Lotos/Theses/rt_msc.ps.gz
- [349] Tuok, R. and Logrippo, L. (1998) “Formal Specification and Use Case Generation for a Mobile Telephony System”. In: *Computer Networks and ISDN Systems*, (30) 11, 1045-1063.
- [350] Turner, K.J. (1992) *Using Formal Description Techniques; An Introduction to ESTELLE, LOTOS and SDL*, Wiley Publishers, U.K.
- [351] Turner, K.J. (1993) “An Engineering Approach to Formal Methods”. In: A. Danthine, G. Leduc, and P. Wolper (Eds), *Protocol Specification, Testing and Verification, XIII*, North-Holland, 357-380.
<ftp://ftp.cs.stir.ac.uk/pub/staff/kjt/research/pubs/engg-form.pdf>
- [352] Turner, K.J. (1998) “Validating Architectural Feature Descriptions using LOTOS”. In: K. Kimbler and W. Bouma (eds.), *Fifth International Workshop on Feature Interactions in Telecommunications Software Systems*, IOS Press. <ftp://ftp.cs.stir.ac.uk/pub/staff/kjt/research/pubs/val-feat.pdf>
- [353] Turner, K.J. (1998) “An architectural description of intelligent network features and their interactions”. In: *Computer Networks and ISDN Systems*, vol. 30, no. 15, September, 1389-1419.
<ftp://ftp.cs.stir.ac.uk/pub/staff/kjt/research/pubs/arch-feat.pdf>
- [354] Turner, K.J. (2000) “Formalising the Chisel Feature Notation”. In: *Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'00)*, Glasgow, Scotland, UK, May 2000. IOS Press, Amsterdam, 241-256.
<ftp://ftp.cs.stir.ac.uk/pub/staff/kjt/research/pubs/form-chis.pdf>
- [355] Turner, K.J. (2000). “Realising Architectural Feature Descriptions using LOTOS”. In: *Parallel Computers, Networks and Distributed Systems (Calculateurs Parallèles, Réseaux et Systèmes Répartis)*, Editions Hermès, Paris, August 2000. <ftp://ftp.cs.stir.ac.uk/pub/staff/kjt/research/pubs/feat-lot.pdf>
- [356] Turner, K.J. (2000). “Structuring Telecommunication Features”. To appear in: S. Gilmore and M. Ryan (Eds), *Language Constructs for Designing Features*. Springer-Verlag. 1-10.

-
- [357] Turner, K. (2001) "Modular Feature Specification". In: *Proceedings of the 6th Mitel Conference (MICON 2001)*, Ottawa, Canada, August 2001.
http://micmac.mitel.com/micon/Proceedings/Ken_Turner_MICON_Proceedings.pdf
- [358] UML Revision Task Force (1999) *OMG Unified Modeling Language Specification, version 1.3*, June 1999. <http://uml.shl.com/artifacts.htm>
- [359] *Use Case Maps Web Page* and *UCM User Group* (1999). <http://www.UseCaseMaps.org>
- [360] Ural, H. (1992) "Formal methods for test sequence generation". In: *Computer Communications*, 15, 311-325.
- [361] Utas, G. (1998) "A Pattern Language of Feature Interaction". In: K. Kimbler and L. G. Bouma (Eds), *Fifth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'98)*, Lund, Sweden, September 1998. IOS Press, 98-114
- [362] Vigder, M. (1992) *Integrating Formal Techniques into the Design of Concurrent Systems*. Ph.D. thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada.
- [363] Vissers, C. and Logrippo, L. (1986) "The importance of the service concept in the design of data communications protocols". In: *Proceedings of the 6th IFIP Workshop on Protocol Specification, Testing and Verification*, June 1986, 3-17.
- [364] Vissers, C.A., Scollo, G., van Sinderen, M., Brinksma, E. (1991) "Specification Styles in Distributed Systems Design and Verification", *Theoretical Computer Science '89*, 179-206.
- [365] Wang, E.Y. and Cheng, B.H.C. (1997) "Formalizing and Integrating the Dynamic Model within OMT". In: *IEEE Proc. of International Conference on Software Engineering (ICSE'97)*, Boston, USA, May 1997. <ftp://ftp.cse.msu.edu/pub/serg/requirements/icse97.ps.gz>
- [366] Wang, E.Y. and Cheng, B.H.C. (1998) "A Rigorous Object-Oriented Design Process". In: *Proc. of International Conference on Software Process*, Naperville, USA, June 1998.
<ftp://ftp.cse.msu.edu/pub/serg/requirements/icsp5.ps.gz>
- [367] Wang, E.Y. and Cheng, B.H.C. (1998) "Formalizing and Integrating the Functional Model into Object-Oriented Design". In: *Proc. of the 10th International Conf. on Software Engineering and Knowledge Engineering*, San Francisco, USA, June 1998.
<ftp://ftp.cse.msu.edu/pub/serg/requirements/seke98.ps.gz>
- [368] Weidenhaupt, K., Pohl, K., Jarke, M., and Haumer, P. (1998) "Scenarios in System Development: Current Practice". In: *IEEE Software*, March/April 1998, 34-45.
- [369] Weiss, M., Gray, T., and Diaz, A. (1997) "Experiences with a Service Environment for Distributed Multimedia Applications". In: Dini, P., Boutaba, R., and Logrippo, L. (Eds.) *Feature Interactions in Telecommunications and Distributed Systems IV*, IOS Press.
- [370] Weyuker, E.J. and Ostrand, T.J. (1980) "Theories of Program Testing and the Application of Revealing Subdomains". In: *IEEE Transactions on Software Engineering*, Vol. 6.
- [371] Weyuker, E.J. (1988) "The evaluation of program-based software test data adequacy criteria". In: *Communications of the ACM*, 31(6), June, 668-675.

- [372] Weyuker, E.J. (1988) "An empirical study of the complexity of data flow testing". In: *Proc. Second Workshop on Testing, Verification and Analysis*, Banff, Canada, July 1988.
- [373] Weyuker, E.J., Goradia, T., and Singh, A. (1994) "Automatically Generating Test Data from a Boolean Specification". In: *IEEE Transaction on Software Engineering*, Vol. 20, No. 5, 353-363. Also in [287].
- [374] Whittle, J. and Shumann, J. (2000) "Generating Statechart Designs From Scenarios". In: *22th International Conference on Software Engineering (ICSE 2000)*, Limerick, Ireland, ACM Press, 314-323.
- [375] Williams, A.W. (2000) "Determination of Test Configurations for Pair-Wise Interaction Coverage". In: H. Ural, R.L. Probert and G.v. Bochmann (eds) *Testing of Communicating Systems: Tools and Techniques (TestCom 2000)*. Kluwer Academic Publishers, 59-74.
- [376] Winskel, G. (1987) "Event structures". In: W. Brauer, W. Reisig, and G. Rosenberg (eds) *APN'86, Advances in Petri Nets*. LNCS, vol. 255, Springer-Verlag, 325-392.
- [377] Woodside, C.M., Menascé, D. and Gomaa, H., eds. (2000) *Proceedings of the Second International Workshop on Software and Performance*, Ottawa, Canada, September 2000.
<http://www.sce.carleton.ca/wosp2000/>
- [378] Woodward, M.R. (1993) "Errors in Algebraic Specifications and an Experimental Mutation testing Tool". In: *Software Engineering Journal*, July 1993, 211-224.
- [379] W3 Consortium (1998) *Extensible Markup Language (XML) 1.0*. W3C Recommendation, 10 February 1998. <http://www.w3.org/TR/REC-xml>
- [380] Yee, G.M. and Woodside, C.M.. (1990) "A Transformational Approach to Process Partitioning Using Timed Petri Nets". In: *Proc. Int. Computer Symposium 90 (ICS90)*, Taiwan, December, 395-401.
- [381] Yi, Z. (2000) *CNAP Specification and Validation: A Design Methodology Using LOTOS and UCM*. M.Sc. thesis, SITE, University of Ottawa, Canada. <http://www.UseCaseMaps.org/pub/yi-thesis.pdf>
- [382] Zave, P. and Jackson, M. (1996) "Where do operations come from? A multiparadigm specification technique". In: *IEEE Transactions on Software Engineering*, XXII(7), July, 508-528.
- [383] Zave, P. and Jackson, M. (1997) "Four dark corners of requirements engineering". In: *ACM Transactions on Software Engineering and Methodology* VI(1), January 1997, 1-30.
<http://www.research.att.com/~pamela/4dc.ps>
- [384] Zave, P. (1997) "Classification of research efforts in requirements engineering". In: *ACM Computing Surveys* 29(4), December, 315-321. <http://www.research.att.com:80/orgs/ssr/people/pamela/re.ps.gz>
- [385] Zave, P. (2001) "Requirements for Evolving Systems: A Telecommunications Perspective". In: *RE'01, Fifth IEEE Int. Symposium on Requirements Engineering*, Toronto, Canada, August 2001, 2-9.
- [386] Zhu, H., Hall, P.A.V., and May, J.H.R. (1997) "Software unit test coverage and adequacy". In: *ACM Computing Surveys*, 29(4), December, 366-427

Appendix A: UCM Quick Reference Guide

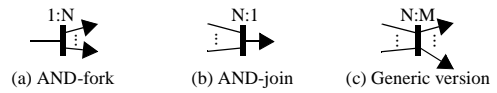
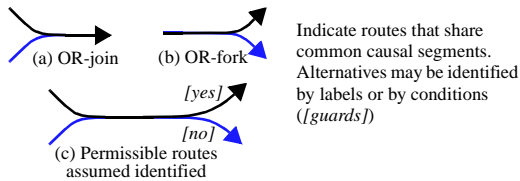


Imagine tracing a path through a system of objects to explain a causal sequence, leaving behind a visual signature. Use Case Maps capture such sequences. They are composed of:

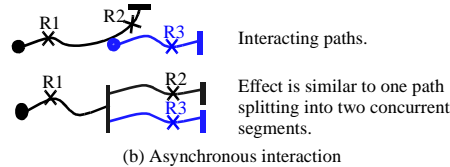
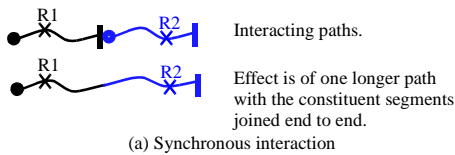
- **start points** (filled circles representing preconditions and/or triggering causes)
- causal chains of **responsibilities** (crosses, representing actions, tasks, or functions to be performed)
- and **end points** (bars representing postconditions and/or resulting effects).

The responsibilities can be bound to **components**, which are the entities or objects composing the system.

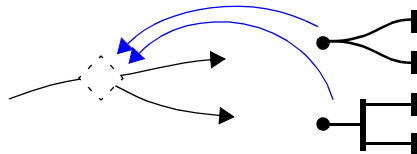
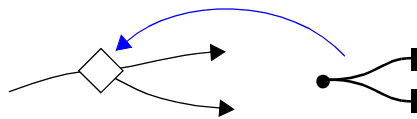
A1. Basic notation and interpretation



A2. Shared routes and OR-forks/joins.

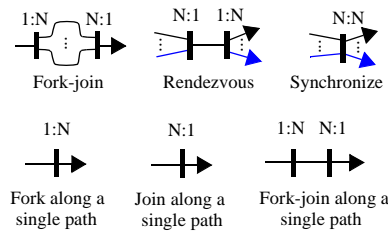


A3. Path interactions.

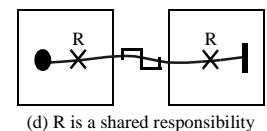
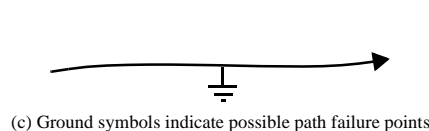
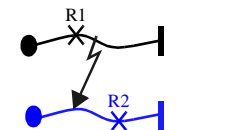
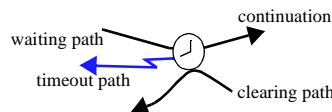


A6. Stubs and plug-ins.

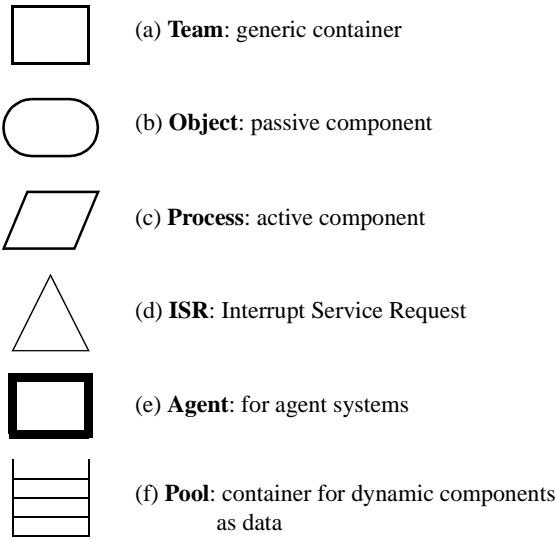
A4. Concurrent routes with AND-forks/joins.



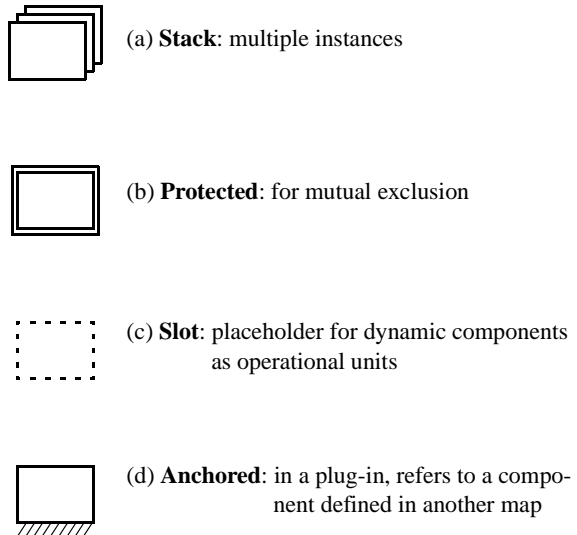
A5. Variations on AND-forks/joins.



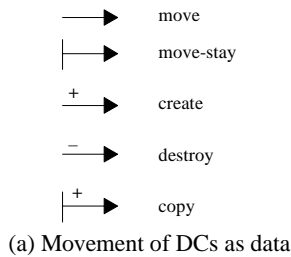
A7. Timers, aborts, failures, and shared responsibilities.



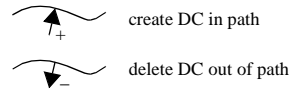
A8. Component types.



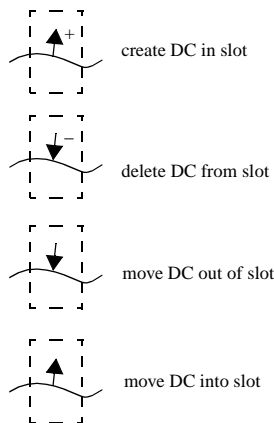
A9. Component attributes.



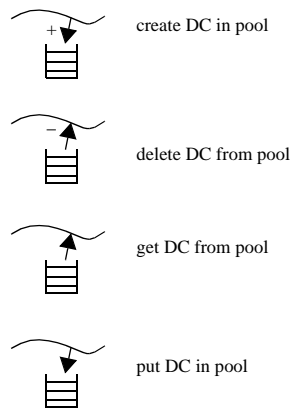
(a) Movement of DCs as data



(b) Directly into or out of paths

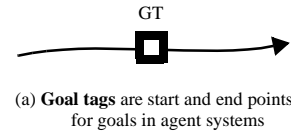


(c) Into or out of slots

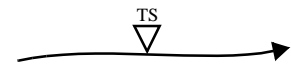


(d) Into or out of pools

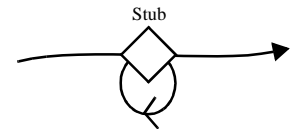
A10. Movement notation for **dynamic components** (DCs).



(a) **Goal tags** are start and end points for goals in agent systems



(b) **Timestamps** are start and end points for response time requirements



(c) **Direction arrows** can be used when path direction is ambiguous

A11. Notation extensions

Appendix B: LOTOS Specification of TTS

Here is the fully commented LOTOS specification derived from the Use Case Maps for the *Tiny Telephone System* (TTS) given in Figure 20. It contains the following elements:

- Identification and modification history: lines 1 to 11.
- Definition of observable gates and events: lines 12 to 18.
- Basic data structures and operations (simplified version of the International Standard ADTs): lines 19 to 169.
- Data structures and operations specific to TTS: lines 170 to 408.
- Component structure, with process definitions for components and stubs: lines 409 to 647
- Process definitions for the plug-ins: lines 648 to 777.
- Validation test processes for the basic call: lines 778 to 834.
- Validation test processes for individual features: lines 835 to 926.
- Validation test processes for pairs of features: lines 927 to 947.
- Robustness test cases: lines 948 to 1044.
- 26 probe comments (**_PROBE_**).

```

1  (*****
2  (* Tiny Telephone System (TTS) *)
3  (* Goal : Ongoing example for Ph.D. thesis proposal *)
4  (* Version: 1.0 *)
5  (* Date : February 15, 2000 *)
6  (* Authors: Daniel Amyot (damyot@site.uottawa.ca) *)
7  (* SITE, University of Ottawa, Canada *)
8  (* History: February 15, 2000 : Added 26 probe comments. *)
9  (* December 10, 1999 : First version (~12h work) & 14 test cases. *)
10 (*****
11
12 specification TTS [req, (* Request start point, from User to Agent *)
13 ring, (* Ring end point, from Agent to User *)
14 sig, (* Signal end point, from Agent to User *)
15 disp, (* Display end point, from Agent to User *)
16 init (* Extra gate for initializing configurations. *)
17 ]:noexit
18

```

```

19  (*=====*)
20  (*           Modified IS8807 ADT definitions           *)
21  (*=====*)
22
23  (* Types FBoolean, Element, and Set contain corrections *)
24  (* to the library from the International Standard 8870. *)
25  (* Type Boolean remains the same, but NaturalNumber was *)
26  (* simplified by removing unnecessary arithmetic and *)
27  (* comparison operators. *)
28  (* Further operators in Set have been commented out as *)
29  (* they are not used in this specification. *)
30
31  type Boolean is
32  sorts
33      Bool
34  opns
35      true, false: -> Bool
36      not: Bool -> Bool
37      _ and _, _ or _, _ eq _, _ ne _: Bool, Bool -> Bool
38      _ xor _, _ implies _, _ iff _: Bool, Bool -> Bool
39  eqns
40      forall x, y: Bool
41      ofsort Bool
42          not (true) = false ;
43          not (false) = true ;
44          x and true = x ;
45          x and false = false ;
46          x or true = true ;
47          x or false = x ;
48          x xor y = x and not (y) or (y and not (x)) ;
49          x implies y = y or not (x) ;
50          x iff y = x implies y and (y implies x) ;
51          x eq y = x iff y ;
52          x ne y = x xor y ;
53  endtype (* Boolean *)
54
55  (*****
56
57  type NaturalNumber is Boolean
58  sorts
59      Nat
60  opns
61      0: -> Nat
62      Succ: Nat -> Nat
63      _ + _: Nat, Nat -> Nat
64      _ eq _, _ ne _: Nat, Nat -> Bool
65  eqns
66      forall m, n: Nat
67      ofsort Nat
68          m + 0 = m ;
69          m + Succ (n) = Succ (m) + n ;
70      ofsort Bool
71          0 eq 0 = true ;
72          0 eq Succ (m) = false ;
73          Succ (m) eq 0 = false ;
74          Succ (m) eq Succ (n) = m eq n ;
75          m ne n = not (m eq n) ;
76  endtype (* NaturalNumber *)
77
78  (*****

```

```

79
80  type      FBoolean is
81  formalops FBool
82  formalops true      : -> FBool
83          not        : FBool -> FBool
84  formaleqns
85      forall x : FBool
86      ofsort FBool
87          not(not(x)) = x;
88  endtype    (* FBoolean *)
89
90  (*****)
91
92  type      Element is FBoolean
93  formalops Element
94  formalops _ eq _, _ ne _      : Element, Element -> FBool
95  formaleqns
96      forall x, y, z : Element
97      ofsort Element
98          x eq y = true =>
99          x      = y      ;
100
101      ofsort FBool
102          x = y =>
103          x eq y = true      ;
104          x eq y = true , y eq z = true =>
105          x eq z = true      ;
106
107          x ne y = not(x eq y) ;
108  endtype    (* Element *)
109
110  (*****)
111
112  type      Set is Element, Boolean, NaturalNumber
113  sorts     Set
114  opns      {} : -> Set
115          Insert, Remove : Element, Set -> Set
116          _IsIn_, _NotIn_ : Element, Set -> Bool
117          _Includes_, _eq_, _ne_ : Set, Set -> Bool
118          (*_Union_, _Ints_, _Minus_ : Set, Set -> Set
119          _IsSubsetOf_ : Set, Set -> Bool
120          Card : Set -> Nat *)
121
122  eqns      forall x, y : Element,
123          s, t : Set
124  ofsort Set
125
126          x IsIn Insert(y,s) =>
127          Insert(x, Insert(y,s)) = Insert(y,s) ;
128          Remove(x, {}) = {} ;
129          Remove(x, Insert(x,s)) = s ;
130          x ne y = true of FBool =>
131          Remove(x, Insert(y,s)) = Insert(y, Remove(x,s));
132

```

```

133  (*{} Union s          = s          ;
134  Insert(x,s) Union t  = Insert(x,s Union t) ;
135
136  {} Ints s            = {}          ;
137  x IsIn t =>
138  Insert(x,s) Ints t   = Insert(x,s Ints t) ;
139  x NotIn t =>
140  Insert(x,s) Ints t   = s Ints t      ;
141
142  s Minus {}          = s          ;
143  s Minus Insert(x, t) = Remove(x,s) Minus t ; *)
144
145  ofsort Bool
146
147  x IsIn {}          = false        ;
148  x eq y = true of FBool =>
149  x IsIn Insert(y,s) = true          ;
150  x ne y = true of FBool =>
151  x IsIn Insert(y,s) = x IsIn s      ;
152  x NotIn s          = not(x IsIn s) ;
153
154  s Includes {}      = true          ;
155  s Includes Insert(x,t) = (x IsIn s) and (s Includes t) ;
156
157  s eq t              = (s Includes t) and (t Includes s);
158
159  s ne t              = not(s eq t)   ;
160
161  (* s IsSubsetOf t   = t Includes s   ;
162
163  ofsort Nat
164
165  Card({})          = 0              ;
166  x NotIn s =>
167  Card(Insert(x,s)) = Succ(Card(s))   ; *)
168  endtype (* Set *)
169
170  (*=====*)
171  (*          TTS ADT definitions          *)
172  (*=====*)
173
174  (* The UserState is either busy or idle. *)
175  type UserState is Boolean
176  sorts UserState
177  opns
178  busy, idle : -> UserState
179  _ eq _, _ ne _ : UserState, UserState -> Bool
180  eqns
181  forall us1, us2 : UserState
182  ofsort Bool
183  busy eq busy = true;
184  busy eq idle = false;
185  idle eq busy = false;
186  idle eq idle = true;
187  us1 ne us2 = not(us1 eq us2);
188  endtype (* UserState *)
189
190  (******)
191

```



```

192 (* The Announcement type is used to refine/split gate sig *)
193 (* From UCMS: OCS plug-in, Terminating plug-in. *)
194 type Announcement is NaturalNumber
195 sorts Announcement
196 opns
197     (* Announcements included in the UCMS *)
198     callDenied, busySig, ringBack,
199     (* Additional messages between agents *)
200     request,
201     (* No announcement specified *)
202     noAnnouncement: -> Announcement
203     map : Announcement -> Nat
204     _ eq _, _ ne _ : Announcement, Announcement -> Bool
205 eqns
206     forall a1, a2 : Announcement
207     ofsort Nat
208         map(callDenied)      = 0;
209         map(busySig)         = succ(0);
210         map(ringBack)        = succ(succ(0));
211         (* Add new announcements/messages here when necessary *)
212         map(noAnnouncement) = succ(succ(succ(0)));
213         map(request)         = succ(succ(succ(succ(0))));
214     ofsort Bool
215         a1 eq a2 = map(a1) eq map(a2);
216         a1 ne a2 = not(a1 eq a2);
217 endtype (* Announcement *)
218
219 (*****)
220
221 (* The User type contains the User sort, *)
222 (* which is an enumeration of user identifiers *)
223 (* that can initiate or receive call requests. *)
224 type User is NaturalNumber
225 sorts User
226 opns
227     userA, userB, userC : -> User
228     map : User -> Nat
229     _ eq _, _ ne _ : User, User -> Bool
230 eqns
231     forall user1, user2 : User
232     ofsort Nat
233         map(userA)      = 0;
234         map(userB)      = succ(0);
235         map(userC)      = succ(succ(0));
236         (* Add users here as necessary *)
237     ofsort Bool
238         user1 eq user2 = map(user1) eq map(user2);
239         user1 ne user2 = not(user1 eq user2);
240 endtype (* User *)
241
242 (* List of users, implemented as a set (useful for the OCS list). *)
243 (* We avoid the problem with ISLA's renaming in actualization. *)
244 type UserList0 is Set
245 actualizedby User using
246 sortnames
247     User for Element
248     Bool for FBool
249 endtype (* UserList0 *)
250

```

```

251 type UserList is UserList0 renamedby
252 sortnames
253   UserList for Set
254 opnnames
255   EmptyUserList for {} (* Empty list of users *)
256 endtype (* UserList *)
257
258 (*****
259
260 (* The OCS check is either allowed or denied *)
261 (* This type extends UserList. *)
262 (* From UCMS: OCS plug-in *)
263 type OCScheck is UserList
264 opns
265   allowed, denied : User, UserList -> Bool
266 eqns
267   forall u : User,
268     ul : UserList
269     ofsort Bool
270     allowed (u, ul) = u NotIn ul;
271     denied (u, ul) = u IsIn ul;
272 endtype (* OCScheck *)
273
274 (*****
275
276 (* The Feature sort is an enumeration of the *)
277 (* features to which users can subscribe, *)
278 (* including the basic call BC. *)
279 type Feature is NaturalNumber
280 sorts Feature
281 opns
282   BC, (* Basic Call *)
283   CND, (* Call Name Delivery *)
284   OCS (* Originating Call Screening *) : -> Feature
285   map : Feature -> Nat
286   _ eq _, _ ne _ : Feature, Feature -> Bool
287 eqns
288   forall f1, f2 : Feature
289   ofsort Nat
290     map(BC) = 0;
291     map(CND) = succ(0);
292     map(OCS) = succ(succ(0));
293     (* Add new features here when necessary *)
294   ofsort Bool
295     f1 eq f2 = map(f1) eq map(f2);
296     f1 ne f2 = not(f1 eq f2);
297 endtype (* Feature *)
298
299 (* List of features, implemented as a set. *)
300 (* We avoid the problem with ISLA's renaming in actualization *)
301 type Flist0 is Set
302 actualizedby Feature using
303 sortnames
304   Feature for Element
305   Bool for FBool
306 endtype (* Logs0 *)
307

```

```

308 type Flist is Flist0 renamedby
309 sortnames
310   Flist for Set
311 opnnames
312   EmptyFList for {} (* Empty list of features *)
313 endtype (* Flist *)
314
315 (*****)
316
317 (* A record for the user information, to be handled by its agent. *)
318 (* Format: info(userID, Features, OCSList) *)
319 type UInfo is UserList, FList
320 sorts UInfo
321 opns
322   info: User,      (* User identifier *)
323         FList,     (* List of subscribed features *)
324         UserList  (* Screened list, for OCS *) -> UInfo
325   uid : UInfo     (* Extract the user identifier *) -> User
326   fl  : UInfo     (* Extract the list of features *) -> FList
327   ocsl: UInfo     (* Extract the OCS list *) -> UserList
328
329   _ eq _, _ ne _ : UInfo, UInfo -> Bool
330 eqns
331   forall u1, u2 : User,
332         f11, f12: FList,
333         u11, u12: UserList,
334         uinf : UInfo
335   ofsort Bool
336     (u1 eq u2) and (f11 eq f12) and (u11 eq u12) =>
337       info(u1, f11, u11) eq info(u2, f12, u12) = true;
338     not((u1 eq u2) and (f11 eq f12) and (u11 eq u12)) =>
339       info(u1, f11, u11) eq info(u2, f12, u12) = false;
340     info(u1, f11, u11) ne info(u2, f12, u12) =
341       not(info(u1, f11, u11) eq info(u2, f12, u12));
342   ofsort User
343     uid(info(u1, f11, u11)) = u1;
344   ofsort FList
345     fl(info(u1, f11, u11)) = f11;
346   ofsort UserList
347     ocsl(info(u1, f11, u11)) = u11;
348 endtype (* UInfo *)
349
350 (*****)
351
352 (* The Direction is either fromMedium or toMedium. *)
353 (* This type is isomorphic to UserState, so the *)
354 (* latter can be reused by renaming. *)
355 type Direction is UserState renamedby
356 sortnames Direction for UserState
357 opnnames
358   fromMedium for idle
359   toMedium for busy
360 endtype (* Direction *)
361
362 (*=====*)
363 (* Stub Path ADT definitions *)
364 (*=====*)
365
366 (* Entry and exit points of each stub in the maps *)
367 (* Also used to describe start/end points in plug-ins. *)

```

```

368 type StubPath is NaturalNumber
369 sorts StubPath
370 opns
371   in1, out1, out2, (* SO stub in Root map *)
372   in2, out3, out4, (* ST stub in Root map *)
373   in3, out5 (* SD stub in Terminating plug-in *) : -> StubPath
374   map : StubPath -> Nat
375   _ eq _, _ ne _ : StubPath, StubPath -> Bool
376 eqns
377   forall sp1, sp2 : StubPath
378   ofsort Nat
379     map(in1) (* From req *) = 0;
380     map(out1) (* To stub ST *) = succ(map(in1));
381     map(out2) (* To sig *) = succ(map(out1));
382     map(in2) (* From stub SO *) = succ(map(out2));
383     map(out3) (* To ring *) = succ(map(in2));
384     map(out4) (* To sig *) = succ(map(out3));
385     map(in3) (* From in2 *) = succ(map(out4));
386     map(out5) (* To out3 *) = succ(map(in3));
387     (* Add new identifiers here when necessary. *)
388   ofsort Bool
389     sp1 eq sp2 = map(sp1) eq map(sp2);
390     sp1 ne sp2 = not(sp1 eq sp2);
391 endtype (* StubPath *)
392
393 (* List of stub path identifiers, implemented as a set. *)
394 (* We avoid the problem with ISLA's renaming in actualization *)
395 type SPList0 is Set
396 actualizedby StubPath using
397 sortnames
398   StubPath for Element
399   Bool for FBool
400 endtype (* SPList0 *)
401
402 type SPList is SPList0 renamedby
403 sortnames
404   SPList for Set
405 opnames
406   EmptySPList for {} (* Empty list of stub path identifiers. *)
407 endtype (* SPList *)
408
409 (*=====*)
410 (* Behaviour Description *)
411 (* (structure, components, and stubs) *)
412 (*=====*)
413
414 behaviour
415
416 (* Gates not visible to the users are set to be internal. *)
417 hide
418   agent2agent (* Inter-agent communication channel *)
419 in
420 (
421   (* We create as many user/agent pairs as necessary. *)
422   UserAgentFactory [req, ring, sig, disp, init, agent2agent]
423   |[agent2agent]|
424   (* Agents communicate through some medium. *)
425   Medium[agent2agent]
426 )
427

```

```

428 where
429
430  (*****
431  (* Process UserAgentFactory: To create and initialize users and agents. *)
432  (*****
433
434  process UserAgentFactory [req, ring, sig, disp, init, agent2agent]:noexit :=
435    init ?userId:User ?userFeatures:FList ?OCSlist:UserList ?state:UserState;
436    (
437      (* Create the user and its associated agent *)
438      (
439        User [req, ring, sig, disp] (userId)
440        |[req, ring, sig, disp]|
441        Agent [req, ring, sig, disp, agent2agent]
442          (info(userId, userFeatures, OCSlist), state)
443      )
444      |||
445      (* Prepare to accept new creation requests *)
446      UserAgentFactory [req, ring, sig, disp, init, agent2agent]
447    )
448  endproc (* UserAgentFactory *)
449
450
451  (*****
452  (* Process Medium: For inter-agent communication. *)
453  (*****
454
455  process Medium[agent2agent]: noexit :=
456    (* Simulates a reliable FIFO channel of length 2. *)
457    (* Allows for 2 requests to be sent simultaneously, *)
458    (* as required by several tests. *)
459    agent2agent !toMedium ?from:User ?to:User ?msg:Announcement;
460    (
461      agent2agent !fromMedium !from !to !msg;
462      (*_PROBE_*) Medium[agent2agent]
463      []
464      agent2agent !toMedium ?from2:User ?to2:User ?msg2:Announcement;
465      agent2agent !fromMedium !from !to !msg;
466      agent2agent !fromMedium !from2 !to2 !msg2;
467      (*_PROBE_*) Medium[agent2agent]
468    )
469    (* Other media could be considered through the use of *)
470    (* process parameter (buffer length), full synchronization *)
471    (* (to ensure mutual exclusion), internal actions (unreliable *)
472    (* channels), etc. *)
473  endproc (* Medium *)
474
475
476  (*****
477  (* Process User: Simplistic user, who does not do or know much... *)
478  (*****
479

```

```

480 process User [req, ring, sig, disp] (userId: User): noexit :=
481   (* Initiate a call request and get an announcement/signal *)
482   req !userId ?callee:User;
483   sig !userId ?msg:Announcement;
484   (*_PROBE_*) User [req, ring, sig, disp] (userId)
485   []
486   (* Receive a ring *)
487   ring !userId;
488   (*_PROBE_*) User [req, ring, sig, disp] (userId)
489   []
490   (* Observe a displayed phone number *)
491   disp !userId ?caller:User;
492   (*_PROBE_*) User [req, ring, sig, disp] (userId)
493 endproc (* User *)
494
495
496 (*****)
497 (* Process Agent: where most of the intelligent and knowledge resides. *)
498 (*****)
499 process Agent [req, ring, sig, disp, agent2agent]
500   (ui:UInfo, state:UserState): noexit :=
501   (* ui is a record containing userId, userFeatures, OCSlist *)
502   hide
503     chk, (* Checks the OCS list *)
504     pds, (* Prepares a denied signal *)
505     vrfy, (* Verifies whether the callee is busy *)
506     pbs, (* Prepares a busy signal *)
507     prbs, (* Prepares a ringBack signal *)
508     upd (* Updates the busy state of the callee *)
509   in
510     (* Originating role, Agent:O, call request *)
511     (
512       req !uid(ui) ?userT:User;
513       (* Problem : state not set to busy *)
514       SO[chk, pds](Insert(in1, EmptySPLlist), ui, state, userT)
515       >>
516       accept ui:UInfo, state:UserState, msg:Announcement, userT:User,
517         outPaths:SPLlist
518     in
519       [out1 IsIn outPaths] ->
520         (* Forward the request to Agent:T *)
521         (
522           agent2agent !toMedium !uid(ui) !userT !request;
523           (*_PROBE_*) Agent[req, ring, sig, disp, agent2agent](ui, state)
524         )
525       (* [] is used as a composition because end points are all *)
526       (* mutually exclusive in the plug-ins *)
527       []
528       [out2 IsIn outPaths] ->
529         (* Signal an announcement to User:O *)
530         (
531           sig !uid(ui) !msg;
532           (*_PROBE_*) Agent [req, ring, sig, disp, agent2agent](ui, state)
533         )
534     )
535   []

```

```

536     (* Originating role, Agent:O, acknowledgement from Agent:T *)
537     (
538         agent2agent !fromMedium ?userT:User !uid(ui) ?msg:Announcement
539                                 [msg ne request];
540         sig !uid(ui) !msg;
541         (*_PROBE_*) Agent [req, ring, sig, disp, agent2agent] (ui, state)
542     )
543     []
544     (* Terminating role, Agent:T, handles a call request *)
545     (
546         agent2agent !fromMedium ?userO:User !uid(ui) !request;
547         (* Stub ST will call the Terminating plug-in directly *)
548         ST[vrfy, pbs, prbs, upd, disp] (Insert(in2, EmptySPLList), ui,
549                                         state, userO)
550     >>
551     accept ui:UInfo, state:UserState, msg:Announcement, userO:User,
552             outPaths:SPList
553     in
554     (
555         [out4 IsIn outPaths] ->
556         (* Send the acknowledgement signal/announcement to Agent:O *)
557         (
558             agent2agent !toMedium !uid(ui) !userO !msg;
559             (*_PROBE_*) exit
560         )
561         (* ||| is used as a composition because end points are *)
562         (* in parallel and can both occur. *)
563         |||
564         (
565             [out3 IsIn outPaths] ->
566             (* Ring the callee User:T *)
567             (
568                 ring !uid(ui);
569                 (*_PROBE_*) exit
570             )
571             []
572             [out3 NotIn outPaths] ->
573             (* No ring *)
574             (*_PROBE_*) exit
575         )
576     )
577     >> (*_PROBE_*) Agent [req, ring, sig, disp, agent2agent] (ui, state)
578 )
579
580 where
581
582     (*****
583     (* Stub Process SO: From Root Map *)
584     (*****
585     process SO[chk, pds](inPaths:SPList,
586                         ui:UInfo,
587                         state: UserState,
588                         userT: User)
589                         : exit (UInfo, UserState, Announcement, User, SPList) :=
590
591     (* Selection policy: OCS overrides BC *)
592     [OCS IsIn fl(ui)] ->
593     (
594         (* First in1 parameter is the plug-in start point *)
595         OCS[chk, pds](in1, ui, state, userT)

```

```

596     >>
597     (* Connect the resulting end points to the stub exit paths *)
598     accept ui:UInfo, state:UserState, msg:Announcement, userT:User,
599           piep:SPList
600           (* piep is the resulting list of plug-in end points *)
601     in
602       [out1 IsIn piep] ->
603         (*_PROBE_*) exit (ui, state, msg, userT, Insert(out1, EmptySPList))
604       []
605       [out2 IsIn piep] ->
606         (*_PROBE_*) exit (ui, state, msg, userT, Insert(out2, EmptySPList))
607     )
608   []
609   [OCS NotIn fl(ui)] ->
610   (
611     Default[pds](in1, ui, state, userT)
612     >>
613     (* Connect the resulting end points to the stub exit paths *)
614     accept ui:UInfo, state:UserState, userT:User, piep:SPList in
615       [piep eq Insert(out1, EmptySPList)] ->
616         (*_PROBE_*) exit (ui, state, noAnnouncement, userT,
617                           Insert(out1, EmptySPList))
618     )
619   endproc (* SO *)
620
621   (*****
622   (* Stub Process ST: From Root Map *)
623   (*****
624   process ST[vrfy, pbs, prbs, upd, disp] (inPaths:SPList,
625                                         ui: UInfo,
626                                         state: UserState,
627                                         userO: User)
628     : exit (UInfo, UserState, Announcement, User, SPList) :=
629
630   Terminating[vrfy, pbs, prbs, upd, disp] (in2, ui, state, userO)
631   >>
632   (* Connect the resulting end points to the stub exit paths *)
633   accept ui:UInfo, state:UserState, msg:Announcement, userT:User,
634         piep:SPList
635   in
636     [out3 NotIn piep] ->
637       (*_PROBE_*) exit (ui, state, msg, userT, Insert(out4, EmptySPList))
638     []
639     [out3 IsIn piep] ->
640       (*_PROBE_*) exit (ui, state, msg, userT,
641                         Insert(out3, Insert(out4, EmptySPList)))
642
643   endproc (* ST *)
644
645   endproc (* Agent *)
646
647   (*****
648   (* Plug-in processes *)
649   (*****
650
651

```



```

652  (*****
653  (* Process Default: does nothing but check the start point. *)
654  (*****
655  process Default[dummy] (pisp:StubPath, (* Plug-in start point *)
656  ui: UInfo,
657  state: UserState,
658  userOT: User)
659  : exit (UInfo, UserState, User, SPList) :=
660  [pisp eq in1] ->
661  (*_PROBE_*) exit (ui, state, userOT, Insert(out1, EmptySPList))
662  endproc (* Default *)
663
664
665  (*****
666  (* Process Terminating: checks whether the callee is busy, & features *)
667  (*****
668  process Terminating[vrfy, pbs, prbs, upd, disp] (pisp:StubPath,
669  ui: UInfo,
670  state: UserState,
671  userO: User)
672  : exit (UInfo, UserState, Announcement, User, SPList) :=
673  [pisp eq in2] ->
674  (
675  vrfy; (* Verifies the busy status *)
676  (
677  [state eq idle] ->
678  (
679  (
680  SD[disp](Insert(in3, EmptySPList), ui, state, userO)
681  >>
682  accept ui:UInfo, state:UserState, userO:User, piep:SPList
683  in
684  [out5 IsIn piep] ->
685  upd; (* Updates the busy status *)
686  (*_PROBE_*) exit (ui, busy, any Announcement, userO,
687  Insert(out3, Insert(out4, EmptySPList)))
688  )
689  |||
690  prbs; (* Prepares the ringBack signal *)
691  (*_PROBE_*) exit (ui, any UserState, ringBack, userO,
692  Insert(out3, Insert(out4, EmptySPList)))
693  )
694  []
695  [state eq busy] ->
696  (
697  pbs; (* Prepares the busy signal *)
698  (*_PROBE_*) exit(ui, state, busySig, userO, Insert(out4, EmptySPList))
699  )
700  )
701  )
702
703  where
704

```

```

705      (*****)
706      (* Stub Process SD: From Terminating plug-in UCM *)
707      (*****)
708      process SD[disp](inPaths:SPList,
709                      ui:UInfo,
710                      state: UserState,
711                      userO: User)
712          : exit (UInfo, UserState, User, SPList) :=
713
714      (* Selection policy: CND overrides BC *)
715      [CND IsIn fl(ui)] ->
716      (
717          CND[disp](in3, ui, state, userO)
718          >>
719          accept ui:UInfo, state:UserState, userO:User, piep:SPList in
720              [out5 IsIn piep] ->
721                  (*_PROBE_*) exit (ui, state, userO, Insert(out5, EmptySPList))
722      )
723      []
724      [CND NotIn fl(ui)] ->
725      (
726          Default[disp](in1, ui, state, userO)
727          >>
728          (* Connect the resulting end points to the stub exit paths *)
729          accept ui:UInfo, state:UserState, userO:User, piep:SPList in
730              [piep eq Insert(out1, EmptySPList)] ->
731                  (*_PROBE_*) exit (ui, state, userO, Insert(out5, EmptySPList))
732      )
733
734      endproc (* SD *)
735      endproc (* Terminating *)
736
737
738      (*****)
739      (* Process OCS: checks whether the callee is on the OCS list *)
740      (*****)
741      process OCS[chk, pds] (pisp:StubPath, (* Plug-in start point *)
742                          ui: UInfo,
743                          state: UserState,
744                          userT: User)
745          : exit (UInfo, UserState, Announcement, User, SPList) :=
746      [pisp eq in1] ->
747      (
748          chk; (* Checks the OCS list, implemented as a field in ui. *)
749          (
750              [allowed(userT, ocsl(ui))] ->
751                  (*_PROBE_*) exit(ui, state, noAnnouncement, userT, Insert(out1, EmptySPList))
752              []
753              [denied(userT, ocsl(ui))] ->
754                  (
755                      pds; (* Prepares the Denied announcement *)
756                      (*_PROBE_*) exit(ui, state, callDenied, userT, Insert(out2, EmptySPList))
757                  )
758          )
759      )
760      endproc (* OCS *)
761
762

```

```

763  (*****
764  (* Process CND: checks whether the callee is on the OCS list *)
765  (*****
766  process CND[disp] (pisp:StubPath, (* Plug-in start point *)
767      ui: UInfo,
768      state: UserState,
769      userO: User)
770      :exit (UInfo, UserState, User, SPList) :=
771  [pisp eq in3] ->
772  (
773      disp !uid(ui) !userO;
774      (*_PROBE_* exit (ui, state, userO, Insert(out5, EmptySPList))
775  )
776  endproc (* CND *)
777
778  (*****
779  (*
780  (* TEST PROCESSES
781  (*
782  (*****
783
784  (* BASIC CALL (BC): 3 TESTS *)
785
786  (* Info: A calls B idle (ringBack first). Reject on ring. *)
787  (* Plug-ins: SO=Default, ST=Terminating, SD=Default *)
788  (* Abstract sequence: <req, vrfy*, prbs*, upd*, sig, ring> *)
789  (* Constraints: [busy(A)], [idle(B)] *)
790  process t1 [req, ring, sig, disp, init, reject, success]: noexit :=
791  init !userA !Insert(BC, EmptyFList) ?dummy:UserList !busy;
792  init !userB !Insert(BC, EmptyFList) ?dummy:UserList !idle;
793  req !userA !userB;
794  sig !userA !ringBack;
795  (
796  ring !userB; success; stop
797  []
798  ring ?dummy:User [dummy ne userB]; reject; stop
799  )
800  endproc (* t1 *)
801
802  (* Info: A calls B idle (ring first). Reject on sig. *)
803  (* Plug-ins: SO=Default, ST=Terminating, SD=Default *)
804  (* Abstract sequence: <req, vrfy*, prbs*, upd*, ring, sig> *)
805  (* Constraints: [busy(A)], [idle(B)] *)
806  process t2 [req, ring, sig, disp, init, reject, success]: noexit :=
807  init !userA !Insert(BC, EmptyFList) ?dummy:UserList !busy;
808  init !userB !Insert(BC, EmptyFList) ?dummy:UserList !idle;
809  req !userA !userB;
810  ring !userB;
811  (
812  sig !userA !ringBack; success; stop
813  []
814  sig ?dummy1:User ?dummy2:Announcement
815  [(dummy1 ne userA) or (dummy2 ne ringBack)]; reject; stop
816  )
817  endproc (* t2 *)
818

```

```

819      (* Info: A calls B busy. Reject on sig. *)
820      (* Plug-ins: SO=Default, ST=Terminating, SD=- *)
821      (* Abstract sequence: <req, vrfy*, pbs*, sig> *)
822      (* Constraints: [busy(A)], [busy(B)] *)
823      process t3 [req, ring, sig, disp, init, reject, success]: noexit :=
824          init !userA !Insert(BC, EmptyFList) ?dummy:UserList !busy;
825          init !userB !Insert(BC, EmptyFList) ?dummy:UserList !busy;
826          req !userA !userB;
827          (
828              sig !userA !busySig; success; stop
829              []
830              sig ?dummy1:User ?dummy2:Announcement
831                  [(dummy1 ne userA) or (dummy2 ne busySig)]; reject; stop
832          )
833      endproc (* t3 *)
834
835      (* CALL NAME DELIVERY (CND): 2 TESTS *)
836
837      (* Info: A calls B idle, displays, ring first. Reject on disp. *)
838      (* Plug-ins: SO=Default, ST=Terminating, SD=CND *)
839      (* Abstract sequence: <req, vrfy*, prbs*, disp, upd*, ring, sig> *)
840      (* Constraints: [busy(A)], [idle(B)], B has CND *)
841      process t4 [req, ring, sig, disp, init, reject, success]: noexit :=
842          init !userA !Insert(BC, EmptyFList) ?dummy:UserList !busy;
843          init !userB !Insert(CND, Insert(BC, EmptyFList)) ?dummy:UserList !idle;
844          req !userA !userB;
845          (
846              disp !userB !userA;
847              ring !userB;
848              sig !userA !ringBack;
849              success; stop
850              []
851              disp ?dummy1:User ?dummy2:Announcement
852                  [(dummy1 ne userA) or (dummy2 ne ringBack)]; reject; stop
853          )
854      endproc (* t4 *)
855
856      (* Info: A calls B idle, displays, ringBack first. Reject on disp. *)
857      (* Plug-ins: SO=Default, ST=Terminating, SD=CND *)
858      (* Abstract sequence: <req, vrfy*, prbs*, disp, upd*, sig, ring> *)
859      (* Constraints: [busy(A)], [idle(B)], B has CND *)
860      process t5 [req, ring, sig, disp, init, reject, success]: noexit :=
861          init !userA !Insert(BC, EmptyFList) ?dummy:UserList !busy;
862          init !userB !Insert(CND, Insert(BC, EmptyFList)) ?dummy:UserList !idle;
863          req !userA !userB;
864          (
865              disp !userB !userA;
866              sig !userA !ringBack;
867              ring !userB;
868              success; stop
869              []
870              disp ?dummy1:User ?dummy2:Announcement
871                  [(dummy1 ne userA) or (dummy2 ne ringBack)]; reject; stop
872          )
873      endproc (* t5 *)
874

```

```

875  (* ORIGINATING CALL SCREENING (OCS): 3 TESTS *)
876
877  (* Info: A calls B, allowed. Reject on sig. *)
878  (* Plug-ins: SO=OCS, ST=Terminating, SD=Default *)
879  (* Abstract sequence: <req, chk*, vrfy*, prbs*, upd*, ring, sig> *)
880  (* Constraints: [busy(A)], [allowed(B)], [idle(B)] *)
881  process t6 [req, ring, sig, disp, init, reject, success]: noexit :=
882    init !userA !Insert(OCS, Insert(BC, EmptyFList))
883      !Insert(userC, EmptyUserList) !busy;
884    init !userB !Insert(BC, EmptyFList) ?dummy:UserList !idle;
885    req !userA !userB;
886    ring !userB;
887    (
888      sig !userA !ringBack; success; stop
889      []
890      sig ?dummy1:User ?dummy2:Announcement
891        [(dummy1 ne userA) or (dummy2 ne ringBack)]; reject; stop
892    )
893  endproc (* t6 *)
894
895  (* Info: A calls B, allowed but busy. Reject on sig. *)
896  (* Plug-ins: SO=OCS, ST=Terminating, SD=- *)
897  (* Abstract sequence: <req, chk*, vrfy*, pbs*, sig> *)
898  (* Constraints: [busy(A)], [busy(B)], [allowed(B)] *)
899  process t7 [req, ring, sig, disp, init, reject, success]: noexit :=
900    init !userA !Insert(OCS, Insert(BC, EmptyFList)) !EmptyUserList !busy;
901    init !userB !Insert(BC, EmptyFList) ?dummy:UserList !busy;
902    req !userA !userB;
903    (
904      sig !userA !busySig; success; stop
905      []
906      sig ?dummy1:User ?dummy2:Announcement
907        [(dummy1 ne userA) or (dummy2 ne busySig)]; reject; stop
908    )
909  endproc (* t7 *)
910
911  (* Info: A calls B, denied. Reject on sig. *)
912  (* Plug-ins: SO=OCS, ST=-, SD=- *)
913  (* Abstract sequence: <req, chk*, pds*, sig> *)
914  (* Constraints: [busy(A)], [denied(B)] *)
915  process t8 [req, ring, sig, disp, init, reject, success]: noexit :=
916    init !userA !Insert(OCS, Insert(BC, EmptyFList))
917      !Insert(userB, EmptyUserList) !busy;
918    req !userA !userB;
919    (
920      sig !userA !callDenied; success; stop
921      []
922      sig ?dummy1:User ?dummy2:Announcement
923        [(dummy1 ne userA) or (dummy2 ne callDenied)]; reject; stop
924    )
925  endproc (* t8 *)
926

```

```

927 (* OCS and CND: 1 TEST *)
928
929 (* Info: A calls B, allowed, displays. Reject on sig. *)
930 (* Plug-ins: SO=OCS, ST=Terminating, SD=CND *)
931 (* Abstract sequence: <req, chk*,vrfy*,prbs*, disp, upd*,ring, sig> *)
932 (* Constraints: [busy(A)], [allowed(B)], [idle(B)] *)
933 process t9 [req, ring, sig, disp, init, reject, success]: noexit :=
934   init !userA !Insert(OCS, Insert(BC, EmptyFList))
935     !Insert(userC, EmptyUserList) !busy;
936   init !userB !Insert(CND, Insert(BC, EmptyFList)) ?dummy:UserList !idle;
937   req !userA !userB;
938   disp !userB !userA;
939   ring !userB;
940   (
941     sig !userA !ringBack; success; stop
942     []
943     sig ?dummy1:User ?dummy2:Announcement
944       [(dummy1 ne userA) or (dummy2 ne ringBack)]; reject; stop
945   )
946 endproc (* t9 *)
947
948 (* ROBUSTNESS : 5 TESTS *)
949
950 (* Info: A calls A, busy. Reject on sig. *)
951 (* Plug-ins: SO=Default, ST=Terminating, SD=- *)
952 (* Abstract sequence: <req, vrfy*, pbs*, sig> *)
953 (* Constraints: [busy(A)] *)
954 process t10 [req, ring, sig, disp, init, reject, success]: noexit :=
955   init !userA !Insert(BC, EmptyFList) ?dummy:UserList !busy;
956   req !userA !userA;
957   (
958     sig !userA !busySig; success; stop
959     []
960     sig ?dummy1:User ?dummy2:Announcement
961       [(dummy1 ne userA) or (dummy2 ne busySig)]; reject; stop
962   )
963 endproc (* t10 *)
964
965 (* Info: A calls B idle, then C calls B busy. Reject on sig. *)
966 (* Plug-ins: SO=Default, ST=Terminating, SD=Default *)
967 (* Abstract sequence: <req, vrfy*,upd*, ring, sig, req, vrfy*,pbs*, sig> *)
968 (* Constraints: [busy(A)], [idle(B)], [busy(C)] *)
969 process t11 [req, ring, sig, disp, init, reject, success]: noexit :=
970   init !userA !Insert(BC, EmptyFList) ?dummy:UserList !busy;
971   init !userB !Insert(BC, EmptyFList) ?dummy:UserList !idle;
972   init !userC !Insert(BC, EmptyFList) ?dummy:UserList !busy;
973   req !userA !userB;
974   ring !userB;
975   sig !userA !ringBack;
976   req !userC !userB;
977   (
978     sig !userC !busySig; success; stop
979     []
980     sig ?dummy1:User ?dummy2:Announcement
981       [(dummy1 ne userC) or (dummy2 ne busySig)]; reject; stop
982   )
983 endproc (* t11 *)
984

```

```

985  (* Info: A calls B busy while B calls A busy. Reject on sig.          *)
986  (* Plug-ins: SO=Default, ST=Terminating, SD=-                          *)
987  (* Abstract sequence: <req, req, vrfy*, vrfy*, pbs*, pbs*, sig, sig>* *)
988  (* Constraints: [busy(A)], [busy(B)]                                   *)
989  process t12 [req, ring, sig, disp, init, reject, success]: noexit :=
990    init !userA !Insert(BC, EmptyFList) ?dummy:UserList !busy;
991    init !userB !Insert(BC, EmptyFList) ?dummy:UserList !busy;
992    (
993      req !userA !userB;
994      (
995        sig !userA !busySig; exit
996        []
997        sig !userA ?dummy:Announcement [(dummy ne busySig)]; reject; stop
998      )
999      |||
1000     req !userB !userA;
1001     (
1002       sig !userB !busySig; exit
1003       []
1004       sig !userB ?dummy:Announcement [(dummy ne busySig)]; reject; stop
1005     )
1006   )
1007   >> success; stop
1008 endproc (* t12 *)
1009
1010 (* Info: A calls A, denied. Reject on sig.                              *)
1011 (* Plug-ins: SO=OCS, ST=-, SD=-                                        *)
1012 (* Abstract sequence: <req, chk*, pds*, sig>                            *)
1013 (* Constraints: [busy(A)], [denied(A)]                                  *)
1014 process t13 [req, ring, sig, disp, init, reject, success]: noexit :=
1015   init !userA !Insert(OCS, Insert(BC, EmptyFList))
1016     !Insert(userA, EmptyUserList) !busy;
1017   req !userA !userA;
1018   (
1019     sig !userA !callDenied; success; stop
1020     []
1021     sig ?dummy1:User ?dummy2:Announcement
1022       [(dummy1 ne userA) or (dummy2 ne callDenied)]; reject; stop
1023   )
1024 endproc (* t13 *)
1025
1026 (* Info: A calls B, denied, then A calls C, denied. Reject on sig.    *)
1027 (* Plug-ins: SO=OCS, ST=-, SD=-                                        *)
1028 (* Abstract sequence: <req, chk*, pds*, sig, req, chk*, pds*, sig>    *)
1029 (* Constraints: [busy(A)], [denied(B)], [denied(C)]                    *)
1030 process t14 [req, ring, sig, disp, init, reject, success]: noexit :=
1031   init !userA !Insert(OCS, Insert(BC, EmptyFList))
1032     !Insert(userB, Insert(userC, EmptyUserList)) !busy;
1033   req !userA !userB;
1034   sig !userA !callDenied;
1035   req !userA !userC;
1036   (
1037     sig !userA !callDenied; success; stop
1038     []
1039     sig ?dummy1:User ?dummy2:Announcement
1040       [(dummy1 ne userA) or (dummy2 ne callDenied)]; reject; stop
1041   )
1042 endproc (* t14 *)
1043
1044 endspec (* TTS *)

```


Appendix C: Comparing Val And Conf

This appendix develops and illustrates the comparison between our validity relation (val) and the LOTOS conformance relation (conf) discussed in Section 6.2.4 and summarized by Figure 39 on page 183.

In a “perfect world”, requirements are already formalized (by *Req*) and a test suite with the same discriminatory power as the canonical tester of the requirements can be generated (represented by $TS \cong CT(Req)$). In the “real world” however, requirements are usually informal and such canonical tester cannot be defined. This represents the first comparison criteria. The second one is concerned with the presence or absence of a non-empty set of rejection test cases ($REJECT(TS)$).

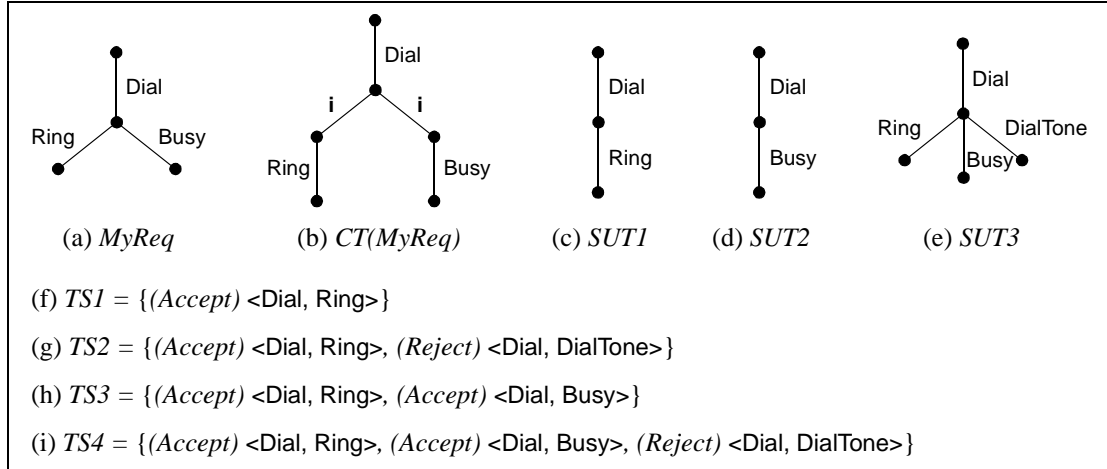
Figure 60 classifies test suites according to these two criteria. As a result, there are four possible combinations of criteria, which lead to various propositions to be explained:

FIGURE 60. val vs. conf: Classification of Criteria and Associated Propositions

Validation Test Suite <i>TS</i>		
	Real world: $\neg(ACCEPT(TS) \cong CT(Req))$	Perfect world: $ACCEPT(TS) \cong CT(Req)$
$REJECT(TS) = \emptyset$	Proposition 5, Proposition 6	Proposition 8
$REJECT(TS) \neq \emptyset$	Proposition 7	Proposition 9, Proposition 10

In order to illustrate the different comparisons, the requirements, canonical tester, SUTs and validation test suites shown in Figure 61 will be used. *MyReq* represents the requirements of a simple telephone connection in the form of an LTS, and $CT(MyReq)$ is the corresponding canonical tester. In this figure, the test type (*Accept* or *Reject*, see Table 15) of each test case is specified.

FIGURE 61. Example Requirements, Canonical Tester, SUTs, and Test Suites



This simple connection example contains three possible SUTs as well as four test suites, one for each of the four categories presented in Figure 60. Five facts need to be noted at this point:

- All the acceptance test cases are reductions of *CT(MyReq)*.
- The acceptance test cases in *TS3* and *TS4* have the same discriminatory power as the canonical tester (i.e. $TS3 \cong CT(MyReq)$ and $TS4 \cong CT(MyReq)$).
- $\neg(SUT1 \text{ \underline{conf} } MyReq)$ because *SUT1* fails the test $\langle Dial, Busy \rangle$.
- $\neg(SUT2 \text{ \underline{conf} } MyReq)$ because *SUT2* fails the test $\langle Dial, Ring \rangle$.
- *SUT3* conf *MyReq*, although *SUT3* supports the trace $\langle Dial, DialTone \rangle$. In fact, this SUT is an extension of the requirements (*SUT3* ext *MyReq*).

Real World, No Rejection Test Cases

In the real world, the set of acceptance test cases is likely to have a lower discriminatory power than *CT(Req)* because the latter (just like *Req*) is usually too complex or even infinite. When no rejection test cases are used, conf and val are related according to Proposition 5:

$$\text{If } \neg(ACCEPT(TS) \cong CT(Req)) \wedge REJECT(TS) = \emptyset, \text{ then } SUT \text{ \underline{conf} } Req \Rightarrow SUT \text{ \underline{val} } TS \quad (\text{PROP. 5})$$

In this case, if the conformance of *SUT* to *Req* is established, then *SUT* is also valid with respect to *TS*. Unfortunately, establishing the conformance first is not particularly useful to the testing framework of SPEC-VALUE. Manipulating the implication leads to a more useful form:

If $\neg(\text{ACCEPT}(TS) \cong \text{CT}(Req)) \wedge \text{REJECT}(TS) = \emptyset$, then $\neg(\text{SUT} \underline{\text{val}} TS) \Rightarrow \neg(\text{SUT} \underline{\text{conf}} Req)$ (**PROP. 6**)

Proposition 6 states that if validity cannot be established, then neither can conformance. This can also be linked to Proposition 2, which describes how testing can invalidate a specification under test. By transitivity, a well-know conclusion of the conformance theory is found again: if a SUT fails to pass a sound test suite derived from a model, then the SUT does not conform to that model.

As an example, *TS1* (Figure 61(f)) satisfies the precondition of Proposition 6:

- $\neg(\text{SUT2} \underline{\text{val}} TS1)$, and indeed $\neg(\text{SUT2} \underline{\text{conf}} MyReq)$.
- $\text{SUT1} \underline{\text{val}} TS1$, yet $\neg(\text{SUT1} \underline{\text{conf}} MyReq)$. This shows that, when non-exhaustive, a sound test suite cannot be used to guarantee conformance.

Real World, With Rejection Test Cases

This is the most realistic situation where rejection test cases are added to a sound but non-exhaustive collection of acceptance test cases. A rejection test such as <Dial, DialTone> can be used to ensure that a requirement that cannot be captured by the LTS (e.g. a dial should not result in a dial tone) is still taken into consideration during the validation. In this situation, Proposition 7 holds:

If $\neg(\text{ACCEPT}(TS) \cong \text{CT}(Req)) \wedge \text{REJECT}(TS) \neq \emptyset$, then $\underline{\text{conf}}$ and $\underline{\text{val}}$ are incomparable (**PROP. 7**)

For example, using *TS2* from Figure 61, counter-examples can be forged for all possible hypotheses (*Req* is itself used as a possible SUT):

- Does $\underline{\text{val}} \Rightarrow \underline{\text{conf}}$? No because: $\text{SUT1} \underline{\text{val}} TS2$, yet $\neg(\text{SUT1} \underline{\text{conf}} MyReq)$
- Does $\underline{\text{val}} \Rightarrow \neg \underline{\text{conf}}$? No because: $MyReq \underline{\text{val}} TS2$, yet $MyReq \underline{\text{conf}} MyReq$
- Does $\neg \underline{\text{val}} \Rightarrow \underline{\text{conf}}$? No because: $\neg(\text{SUT2} \underline{\text{val}} TS2)$, yet $\neg(\text{SUT2} \underline{\text{conf}} MyReq)$
- Does $\neg \underline{\text{val}} \Rightarrow \neg \underline{\text{conf}}$? No because: $\neg(\text{SUT3} \underline{\text{val}} TS2)$, yet $\text{SUT3} \underline{\text{conf}} MyReq$

It is the last case that is of interest to SPEC-VALUE: even with an incomplete test suite, rejection test cases can show that a conforming SUT (e.g. *SUT3*) can still be invalid. This particularity is seldom used in LOTOS-based testing [196].

Perfect World, No Rejection Test Cases

In a perfect world where $CT(Req)$ (or an equivalent test suite) can be computed from requirements and in the absence of rejection test cases, the val and conf relations become indistinguishable, as suggested by Proposition 8:

If $ACCEPT(TS) \equiv CT(Req) \wedge REJECT(TS) = \emptyset$, then $SUT \underline{val} TS \Leftrightarrow SUT \underline{conf} Req$ **(PROP. 8)**

This proposition is trivial to prove: when $REJECT(TS) = \emptyset$, Definition 6.2 reduces to the expression $SUT \underline{val} TS \Leftrightarrow SUT \underline{passes} ACCEPT(TS)$. When $ACCEPT(TS) \equiv CT(Req)$, this expression becomes equivalent to Proposition 3.

Using *TS3* from Figure 61, where $TS3 \equiv CT(MyReq)$, several examples can be illustrated:

- $\neg(SUT1 \underline{val} TS3)$, and indeed $\neg(SUT1 \underline{conf} MyReq)$.
- $\neg(SUT2 \underline{val} TS3)$, and indeed $\neg(SUT2 \underline{conf} MyReq)$.
- $SUT3 \underline{val} TS3$, and indeed $SUT3 \underline{conf} MyReq$.
- $MyReq \underline{val} TS3$, and indeed $MyReq \underline{conf} MyReq$.

Perfect World, With Rejection Test Cases

In this situation, val offers several benefits as it can detect all invalid SUTs that conf can, and it can also detect conforming but invalid SUTs (Proposition 9):

If $ACCEPT(TS) \equiv CT(Req) \wedge REJECT(TS) \neq \emptyset$, then $SUT \underline{val} TS \Rightarrow SUT \underline{conf} Req$ **(PROP. 9)**

For example, this time using *TS4*:

- $\neg(SUT1 \underline{conf} MyReq)$, and indeed $\neg(SUT1 \underline{val} TS4)$.
- $\neg(SUT2 \underline{conf} MyReq)$, and indeed $\neg(SUT2 \underline{val} TS4)$.

-
- $SUT3 \underline{\text{conf}} MyReq$, yet $\neg(SUT3 \underline{\text{val}} TS4)$.
 - $MyReq \underline{\text{val}} TS4$, and indeed $MyReq \underline{\text{conf}} MyReq$.

Using the LOTOS conformance testing theory only, $SUT3$ would never have been found to be invalid according to the requirements, even with a complete test suite with the same discriminatory power as the canonical tester.

What is missing from the current LOTOS conformance testing theory is the use of rejection test cases. When they are added, then the new relation becomes equivalent to validity, as shown in Proposition 10.

$$\begin{array}{l} \text{If } ACCEPT(TS) \cong CT(Req) \wedge REJECT(TS) \neq \emptyset, \\ \text{then } SUT \underline{\text{val}} TS \Leftrightarrow (SUT \underline{\text{conf}} Req \wedge SUT \underline{\text{failsall}} REJECT(TS)) \end{array} \quad \textbf{(PROP. 10)}$$

The classification of Figure 60, together with Propositions 5 to 9, leads to the summary illustrated using relation sets in Figure 39.

Index

A

A posteriori 101, 238
A priori 101, 102, 238
Abstract Data Type (ADT) 31, 154
Abstract sequence 171, 176, 231
ACCEPT 177, 180
Anchored component 122
Appropriate mutant 300
Architecture 20

B

Basic Behaviour Expression (BBE) 242
Behaviour expression 30, 242
Behaviour tree 35
Black box testing 175

C

Call Number Display (CND) 117
Canonical tester 38, 97, 182, 379
Capability Maturity Model (CMM) 5
Causal relationship 3
Causality 45, 50
Chisel 70, 84, 271, 284, 323

COMET 325

Common behaviour 277
Communication Entity Block (CEB) 286
Compositional coverage 253
Conformance testing 93, 173
Conformance Testing Methodology and Framework (CTMF) 94, 173, 218
Construction approach 74, 259, 264, 275, 288, 307
 Analytic 75
 Partial automation 155
 SPEC-VALUE 125
 Synthetic 75
Construction guidelines 129, 308, 316
Coupling effect 295
Coverage 101, 187
 Structural, *see Structural coverage*

D

Design 19
Design errors
 Semantic 88
 Syntactical 88
Design pattern 104, 106, 188

Detectability 92
Device Entity Block (DEB) 286
Discriminatory power 182
Dynamic Causal Tree (DCT) 50

E

Effectiveness 300, 303
Equivalence checking 41
Equivalent mutants 298
Event coverage 243

F

FDT 5
Feature 2, 19, 271
 Interaction problem (FI) 2
Feature interaction 213, 269, 274, 279
Force (pattern) 104, 186
Formal Description Technique, *see* FDT
Formal methods 19, 51
Formal Specifications Maturity model (FSM) 5, 6,
 111, 155, 318, 327
Functional coverage 238

G

General Packet Radio Service (GPRS) 262
Global UCM 273
Grey box testing 175
Group Communication Server (GCS) 257

I

Implementation test suite 179
Incremental feature addition 281
Inspection 173
Interleaving 47
Interoperability 94

L

Labelled Transition Systems, *see* LTS
Language of Temporal Ordering Specification, *see*
 LOTOS

Logical Entity Block (LEB) 286
LOLA 43, 248, 252, 260, 268
LOT2PROBE 249, 250, 311
LOTOS 127
 Abstract Data Type 31
 Action 30
 Behaviour expression (BE) 30
 Environment 30
 Gate 30, 132
 Gate splitting 144
 Operators 30
 Process 30
 Testing 95
 Tools 42
LOTOSphere 320
LTS 9, 33, 100, 219
 Behaviour tree 35

M

May pass verdict 98
May test 99
Message Sequence Chart, *see* MSC
Mobile Application Part (MAP) 292
Model 20
Model checking 41
MSC 2, 56, 68, 150, 321
Must pass verdict 98
Must test 99
Mutant 102, 295
Mutation analysis 295, 302, 304, 311
Mutation operator 296
Mutation score (MS) 300
Mutation testing 102, 295

O

Object Constraint Language (OCL) 58, 82
Originating Call Screening (OCS) 117

P

Pattern 103
 Design, *see Design pattern*
 Testing, *see Testing pattern*
Pattern catalog 105
Pattern language 105, 189
Pattern template 104, 187
Petri net 57
Point-To-Multipoint-Group Call (PTM-G) 262
Pomset 49
Preamble 94
Probe 103, 239
Probe insertion 239, 241, 245
Process 18
 Maturity 18
 Models 112
Property 87
 Liveness 88
 Responsiveness 88
 Safety 88
Protocol 17
Protocol engineering 17, 75
Prototype 20, 112, 259, 264, 275, 288, 292, 307
Public Land Mobile Network (PLMN) 262

R

REJECT 177, 180, 379
Reject test 99
Reject verdict 98
Rejection test 89, 224, 260

Relation 95

 Bisimulation 36, 48
 Conformance (conf) 38, 41, 89, 182, 379
 Congruence 36
 Equality 36
 Equivalence 35, 48, 87
 Extension (ext) 38
 I/O conformance (ioco) 100
 Reduction (red) 38, 89
 Testing equivalence (te) 36
 Trace equivalence (tr) 36
 Validity (val) 180, 182, 379
Requirement 16, 93, 226, 379
 Capture with UCMs 114, 258, 262, 271, 286
 Functional 16
 Non-functional 16
Requirements engineering 1, 16
Research hypothesis 6, 313
Ripple set 216, 219
Robustness 204, 231
ROOM 78, 82

S

Scenario 63
Scenario integration 120, 273
SDL 55, 83, 84, 321
Selective mutation 296
Self-coverage 292
Sequence 242, 312
Sequence coverage 245
Service 2, 18, 19
Simplified Basic Call (SBC) 285
Single BBE 242, 246, 312
Software engineering 18
Specification 19, 93
Specification and Description Language, *see SDL*
Specification Under Test 176

SPEC-VALUE 7, 111, 125, 314, 318, 324
 Construction guidelines 129
 Contributions 10, 313
 Experiments 257, 306, 316
 Mutant generation 296
 Structural coverage 238
 Testing approach 170
Stakeholder 3, 16
Standard 19, 325
Statement probe 240
Step-by-step execution 41
Strategy
 Alternative 191
 Causally linked stubs 210
 Concurrent Paths 194
 Loop 197
 Multiple start points 200
 Single stub 207
Strong mutation 296
Structural coverage 237, 260, 266, 281, 290, 293,
 311, 316
Structure adequacy 282
Style and content guidelines 115
Systems engineering 17
Systems Engineering Capability Model (SECM) 17

T

Test 21
Test body 176
Test case 94, 96
Test case generation 221
Test case management 254
Test goal 91, 176, 184, 218
Test preamble 176
Test purpose 94, 176, 218
Test selection 90, 265, 276, 289, 309
Test suite 90, 94
Test suite validation 294
Test type 176, 188
Test verification 176

Testability 92
 Limit 92, 182
Testing 42, 90
Testing equivalence (te) 36
Testing framework 169, 316
Testing pattern 103, 105, 184, 217, 303, 310
Tests group 177
Timethread Map Description Language (TMDL) 128
Timethreads 320
Traceability 21
TTCN 55, 70, 95, 174, 177, 179, 219, 327, 329
TTS
 Basic Call UCM 118
 CND UCM 119
 Construction guidelines application 157
 Integrated view 120
 Mutants 298
 Mutation analysis 301
 OCS UCM 119
 Structural coverage 249
 Test goal 226
 Testing 226

U

UCM 22, 72, 357
 Abort 27, 140
 Anchored component 122
 AND-fork 25, 134, 194
 AND-join 25, 194
 Binding 26
 Bound 24
 Cardinalities 25
 Causal relationship 22
 Component 22, 143, 146
 Component notation 25
 Dynamic component 26
 Dynamic responsibility 26, 141
 End point 24
 Failure point 27, 140
 Interaction 27, 135
 Interaction point 132
 Navigator (UCMNAV) 28
 OR-fork 24, 134, 192
 OR-join 24, 134, 192
 Path 22, 146
 Plug-in 26, 136, 207
 Pool 26
 Responsibility 5, 22, 132
 Role 143
 Route 24
 Scenario definition 28, 220, 329
 Selection policy 26, 120, 137
 Shared responsibility 27, 151
 Start point 24
 Stub 26, 136, 207, 210
 Style and content guidelines 115
 Timer 27, 139
 Unbound 23
 Unconstrained style 122
UCMNAV 273
UML 58, 321, 324
 Activity diagram 58, 72
Unified Modeling Language, *see UML*
Use case 63, 69, 217
Use Case Maps, *see UCM*

V

V&V, *see Validation and Verification*
Validation 20, 87, 172
Validation and verification 41, 53, 60, 87, 101, 170
Validation test suite 176, 309, 379
Validation testing 172, 173
Validity relation (val), *See Relation Validity* 180
Value selection 222
Verdict 98
Verification 20, 87

W

White box testing 175

