

A Systematic and Traceable Progression from Scenario Models to Communicating Hierarchical State Machines

by

**Francis Bordeleau,
B.Sc., B.Sc.A., M.C.S.**

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of

Doctor of Philosophy

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6
August 12, 1999

© Copyright

1999 Francis Bordeleau

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis,

**A Systematic and Traceable Progression from Scenario Models to
Communicating Hierarchical State Machines**

submitted by

Francis Bordeleau, B.Sc, B.Sc.A, M.C.S.

in partial fulfilment of the requirements
for the degree of Doctor of Philosophy

Chair, Department of Systems
and Computer Engineering

Thesis Supervisor

External Examiner

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
November 16, 1999

Abstract

Real-time systems represent very complex engineering artifacts that are used at all levels of activity in today's society: telephony, banking, health care, traffic control, manufacturing, avionics, aerospace, management systems, etc. In the last decade, the object-oriented paradigm has emerged as a leading development technology in the real-time system industry. Existing object-oriented methodologies allow smoothing the transition between requirements and implementation by combining the use of different types of models to express different views of the systems. In spite of the fact that the object-oriented technology has allowed important progress in the development of complex real-time systems, significant problems remain. Two of these problems are the lack of traceability between models, and the lack of model transition techniques.

This thesis addresses the difficult problem of defining a systematic and traceable progression between requirement level scenario descriptions and communicating hierarchical state machine models in the context of complex real-time system design.

This thesis contributes to solving this problem in the following manner. First, we define a modeling process called RT-TROOP (Real-Time TRaceable Object-Oriented Process). The RT-TROOP modeling process is defined in terms of a set of models and a set of modeling phases. The models include STD (Scenario Textual Description), UCM (Use Case Map), MSC (Message Sequence Chart), and ROOM (Real-Time Object-Oriented Modeling). The modeling phases decompose the overall set of design issues that must be addressed in the design of real-time systems into subsets that are addressed in different modeling phases. Second, we define a set of design patterns to help making the transition between scenario models and communicating hierarchical state machines. Third, we define a set of traceability relations between the RT-TROOP models.

Acknowledgements

After working on this research project for over six years, there are many people that I want to thank.

I want thank Ray Buhr, my supervisor and friend, for the freedom he gave me in conducting this research project. One thing that I learn from Ray is the passion for complex system design. I wish him a great retirement. It is well deserved.

I want to express a special thanks to Bran Selic, who supported this research from the beginning. Without his constant support and constructive comments this Ph.D thesis would not be what it is today. Bran also accepted to sit on the thesis review committee in spite of an extremely busy schedule. Thanks a million!

I want to thank Jean-Pierre Corriveau, friend and colleague, who accepted to read drafts of this thesis and make judicious comments under tight deadlines. Jean-Pierre played the role of supervisor during the last stage of this thesis, while Ray was away on sabbatical and then retired, without receiving official recognition. It is deeply appreciated.

I want to thank Carmine Ciancibello from Nortel Networks who supported this research project in its initial phase. The core of this thesis has been written during the nine months I spend in the Spectrum Tool Support group at Nortel (between November 1995 and September 1996). Being at Nortel to work on this research project kept me in constant contact with the reality of complex system development. There should more people like Carmine in the industry that believe in giving true support to students.

I want to thank Michel Barbeau, from the department of Computer Science in University of Sherbrooke, for accepting to be the thesis external examiner.

I want to thank Murray Woodside and Luigi Logrippo for accepting to be on my thesis review committee and for providing constructive comments.

I want to thank Daniel Biagé, Marc-André Cloutier, André Hamilton, Paul Leduc, and the team of software designers at CML Versatel for giving me the opportunity to implement the research results of this thesis in an industrial context. This was tremendously useful.

I want to sincerely thank several colleagues, friends, and students who contributed to this thesis in different ways (in alphabetic order): Daniel Amyot, Olivier Basset, Don Cameron, Ron Casselman, who in my mind should receive more credits for the development of the Use Case Map modeling technique, Marc Frappier, Michel Locas, Trevor Pearce, Marco Perisic, Dorin Petriu, Sebastien Picard, Vojislav Radonjic, and Fahim Sheik. I have without doubt left out some people from this list, for which I apologize.

I want to thank the School of Computer Science at Carleton who gave me the opportunity to complete the work of this research project while being a full-time faculty member.

I want to thank TRIO (now CITO) and the Real-Time and Distributed Systems Group of the Department of Systems and Computer Engineering at Carleton for their financial support during the first three years (1993 to 1996) of this project.

I want to thank ObjecTime Limited for providing licenses of the ObjecTime Developer toolset and support.

I also want to thank my parents Normand and Louise, and sisters Marie-Claude, Isabelle, and Catherine, for their constant support.

Finally, and foremost, I want to reserve my warmest thanks to my wife Louise-Marie and my three children Xavier, Raphaëlle, and Delphine for their love and support. You sincerely deserve a large part of this thesis. I love you.

Table of Contents

CHAPTER 1 Introduction.....	1
1.1 Context and Motivations.....	1
1.2 Problem Description	4
1.2.1 Lack of traceability	4
1.2.2 Lack of model transition techniques.....	7
1.3 Objectives	11
1.4 Thesis Contributions	11
1.4.1 Definition of the RT-TROOP Modeling Process	12
1.4.2 Integration of the UCM Modeling Technique in a Concrete Modeling Process.....	15
1.4.3 Definition of a Set of Hierarchical State Machine Design Patterns.....	16
1.4.4 Definition of Traceability Relations	18
1.4.5 Development of Case Studies	19
1.5 Thesis Outline	20
CHAPTER 2 Modeling Techniques and Notation.....	22
2.1 Requirements	23
2.1.1 Requirements Table	23
2.1.2 Scenario Textual Description (STD).....	25
2.2 UCM	29
2.2.1 Basic UCM Notation	30
2.2.2 Introducing Components in UCM	33
2.2.3 Path Segment Connectors	34
2.2.4 Other Path Notations.....	38
2.2.5 Related Path Set	40
2.2.6 Path Interaction Notation	41
2.2.7 Composite Use Case Maps	43

2.2.8	UCM Model.....	44
2.2.9	UCM Model Analysis.....	46
2.3	MSC.....	47
2.3.1	Basic MSC.....	48
2.3.2	HMSC.....	57
2.3.3	Additional MSC Concepts and Notations.....	60
2.3.4	MSC Model.....	63
2.3.5	MSC Skeleton.....	64
2.4	ROOM.....	65
2.4.1	Motivation for Using ROOM.....	65
2.4.2	ROOM Structure Notation.....	66
2.4.3	ROOMChart Notation.....	71
2.4.4	ROOM Model.....	77
2.5	Summary.....	77
CHAPTER 3 RT-TROOP Modeling Process.....		78
3.1	Modeling Process.....	79
3.2	Overview of RT-TROOP Modeling Process.....	84
3.2.1	Characteristics.....	85
3.2.2	Composition of RT-TROOP.....	86
3.2.3	Role of the Different Models.....	88
3.2.4	Modeling Phases.....	91
3.3	From STD to UCM.....	96
3.3.1	Relationship between UCM Models and STD Models.....	96
3.3.2	Generation of the UCM Model.....	97
3.4	UCM Modeling Phase.....	100
3.4.1	Modeling Activities.....	101
3.4.2	Structure.....	102
3.4.3	Path.....	104
3.5	Transition from UCM to MSC.....	105

3.5.1	Relationship between MSC Models and UCM Models.....	107
3.5.2	Generation of MSC Skeletons	110
3.5.3	Definition of Message Sequences.....	117
3.6	Specification MSC Modeling	119
3.6.1	Modeling Activities	120
3.7	From MSC to ROOM Structure.....	125
3.7.1	Relationship between ROOM Model and MSC Models	126
3.7.2	Generation of a ROOM Structure Model from a MSC Model	128
3.7.3	Definition of ROOM Role Structures from MSCs	131
3.8	ROOM Structure Modeling	138
3.8.1	Definition of Final Actors.....	139
3.8.2	Definition of System Protocol Classes	140
3.8.3	Definition of System Contracts.....	141
3.9	Addition of ROOM Structure Information to the MSC model.....	143
3.10	Component Behavior Modeling in MSC	146
3.11	From MSC to ROOMChart	149
3.11.1	Generation of ROOMChart Models from Customized MSC model	150
3.11.2	Definition of Role Behaviors.....	152
3.12	ROOMChart Modeling	156
3.13	Adapting the RT-TROOP Modeling Phases to Concrete Cases	158
3.13.1	General Process Issues.....	159
3.13.2	Reusing Existing ROOM Components.....	166
3.13.3	Using Requirement Model That Contains MSCs	168
3.13.4	Using Existing Design Patterns (Mediator Pattern).....	170
3.14	Chapter Summary	174
CHAPTER 4 Behavior Integration Patterns		176
4.1	Introduction.....	177
4.2	Description of the ATM system	184
4.3	Scenario Partitioning Pattern	188

4.4	State Machine Integration Pattern.....	194
4.5	Mode-Oriented Behavior Pattern.....	205
4.6	Mutually Exclusive Scenario Pattern.....	212
4.7	Scenario Interaction Patterns	220
4.7.1	Scenario Composition.....	222
4.7.2	Waiting Place	224
4.7.3	Timed Waiting Place.....	226
4.7.4	Scenario Aborting	228
4.8	More Patterns.....	230
4.9	Chapter Summary	231

CHAPTER 5 Application of RT-TROOP Modeling Process: A Simple Printer

	System Case Study	232
5.1	Requirements	233
5.1.1	STDs for Iteration 1	235
5.2	From STD to UCM (Iteration 1).....	238
5.3	UCM Modeling Phase (Iteration 1)	241
5.4	Transition from UCM to MSC (Iteration 1).....	247
5.4.1	Generation of MSC Skeletons	247
5.4.2	Definition of Message Sequences (Iteration 1).....	254
5.5	Specification MSC Modeling (Iteration 1)	259
5.6	From MSC to ROOM Structure (Iteration 1)	260
5.7	ROOM Structure Modeling (Iteration 1).....	266
5.8	Addition of ROOM Structure Information to the MSC model (Iteration 1).....	269
5.9	Component Behavior Modeling in MSC (Iteration_1).....	273
5.10	From MSC to ROOMChart (Iteration 1)	279
5.10.1	PrintFile	280
5.10.2	StopPrinting	286
5.11	ROOMChart Modeling (Iteration 1).....	289
5.12	Testing of Iteration 1	291

5.13	Iteration 2	292
5.13.1	STDs for Iteration 2	293
5.13.2	UCM Model (Iteration 2).....	294
5.13.3	Specification MSC Model (Iteration 2)	296
5.13.4	ROOM Structure Model (Iteration 2)	297
5.13.5	Customized MSC Model (Iteration 2)	300
5.13.6	ROOMChart Model (Iteration 2).....	302
5.13.7	Testing of Iteration 2.....	312
5.13.8	Error Fixing (Iteration 2)	313
5.14	Chapter Summary	322

CHAPTER 6 Definition of Traceability Relations Between Requirements, Scenario

Models, and Communicating Hierarchical State Machines323

6.1	Traceability Relations	324
6.1.1	Characteristics.....	325
6.1.2	Notation	326
6.1.3	Inter-Model Traceability	327
6.1.4	Inter-Version Traceability	329
6.1.5	Design Decision Traceability.....	330
6.2	Traceability Relation Between STD and System Requirements	333
6.3	Traceability Relations in UCM Models.....	334
6.3.1	Inter-Model Traceability between UCM Models and STDs.....	335
6.3.2	Inter-Version Traceability in UCM Models	337
6.3.3	Design Decision Traceability in UCM Models.....	340
6.4	Traceability Relations in MSC Models.....	341
6.4.1	Inter-Model Traceability Between MSC Models and UCM Models....	341
6.4.2	Intra-Model Traceability in MSC Models	345
6.4.3	Design Decision Traceability in MSC Models	349
6.5	Traceability Relations in ROOM Models.....	349
6.5.1	Inter-Model Traceability Between ROOM Models and MSC Models.	349
6.5.2	Inter-Version Traceability in ROOM Models	354

6.5.3	Design Decision Traceability in ROOM Models.....	356
6.6	Summary.....	357
CHAPTER 7 Conclusion		359
7.1	Summary of Thesis Contributions	359
7.1.1	Definition of the RT-TROOP Modeling Process	359
7.1.2	Integration of the UCM Modeling Technique	360
7.1.3	Definition of Behavior Integration Patterns.....	361
7.1.4	Traceability Relations	362
7.1.5	Development of Case Studies	362
7.2	Implementation Results	363
7.2.1	Implementation of the Modeling Process	363
7.2.2	Implementation of the Behavior Integration Patterns.....	366
7.2.3	Implementation of the Traceability Relations.....	367
7.3	Impact of Contributions	367
7.4	Future Work	368
References.....		371

List of Figures

FIGURE 1.	A simple traceable format for requirements	24
FIGURE 2.	ATM withdraw transaction STD.....	28
FIGURE 3.	A simple UCM path.....	30
FIGURE 4.	Unbound and bound maps	34
FIGURE 5.	Path segment connectors.....	35
FIGURE 6.	Generalized path segment synchronization connector.....	37
FIGURE 7.	Combination of path segment connectors.....	37
FIGURE 8.	Waiting places.....	38
FIGURE 9.	Stub	39
FIGURE 10.	A UCM related path set	40
FIGURE 11.	Path interaction notation	41
FIGURE 12.	Composite use case maps	44
FIGURE 13.	Related path sets and composite maps.....	45
FIGURE 14.	Composition of a UCM model.....	45
FIGURE 15.	Basic MSC notation	49
FIGURE 16.	Alternative and parallel inline expressions.....	52
FIGURE 17.	Iteration inline expression.....	53
FIGURE 18.	Coregion.....	54
FIGURE 19.	Using timer in basic MSC.....	55
FIGURE 20.	MSC reference	56
FIGURE 21.	Instance decomposition.....	57
FIGURE 22.	High level MSC (HMSC)	59
FIGURE 23.	HMSC parallel composition defined in standard MSC'96.....	60
FIGURE 24.	HMSC parallel composition used in this thesis	60
FIGURE 25.	MSC with contract identifier and message data box	62

FIGURE 26.	Synchronous communication symbol.....	63
FIGURE 27.	An example of MSC skeleton.....	64
FIGURE 28.	ROOM structure	67
FIGURE 29.	ROOM layering	70
FIGURE 30.	Structural replication.....	71
FIGURE 31.	Basic ROOMCharts notation	73
FIGURE 32.	More complex ROOMCharts notation	73
FIGURE 33.	Iterative process	84
FIGURE 34.	RT-TROOP modeling process	87
FIGURE 35.	Transition from STD to UCM.....	96
FIGURE 36.	Generation of UCM from STD.....	99
FIGURE 37.	Unbound UCM map resulting of the transition from STD.....	100
FIGURE 38.	UCM modeling phase	101
FIGURE 39.	Generation of UCM from UCM requirements	103
FIGURE 40.	Responsibility decomposition.....	105
FIGURE 41.	Path factoring and merging.....	105
FIGURE 42.	Transition from UCM to MSC.....	106
FIGURE 43.	Relationship between UCM and MSC.....	109
FIGURE 44.	Generation of HMSC and basic MSC skeleton from UCM	110
FIGURE 45.	Generation of an HMSC from UCMs.....	112
FIGURE 46.	Generation of an MSC skeleton from a UCM path	114
FIGURE 47.	Expressing a UCM stub in a basic MSC skeleton	115
FIGURE 48.	Expressing a UCM timer in a basic MSC skeleton	117
FIGURE 49.	Description of message sequence in MSC S1	118
FIGURE 50.	Specification MSC modeling phase.....	120
FIGURE 51.	Component decomposition in MSC model.....	122
FIGURE 52.	Message decomposition	123
FIGURE 53.	MSC restructuring.....	125

FIGURE 54.	Transition from MSC to ROOM structure.....	126
FIGURE 55.	Relationship between MSC and ROOM.....	128
FIGURE 56.	Relationship between MSC and ROOM structure.....	131
FIGURE 57.	Definition of role actors for MSC S1.....	133
FIGURE 58.	Communication diagram for MSC S1	134
FIGURE 59.	Definition of the role protocol classes	136
FIGURE 60.	Definition of ROOM role structure for MSC S1	137
FIGURE 61.	ROOM structure modeling	138
FIGURE 62.	ROOM role structures RS1 and RS2	139
FIGURE 63.	Definition of system actors	140
FIGURE 64.	Definition of system protocol classes	141
FIGURE 65.	Definition of system contracts	142
FIGURE 66.	Customization of MSC model	144
FIGURE 67.	Description of message sequence in MSC S1	146
FIGURE 68.	Component behavior modeling in MSC	147
FIGURE 69.	Description of message sequence in MSC S1	149
FIGURE 70.	Transition from customized MSC model to ROOMChart model.....	150
FIGURE 71.	Relationship between MSC and ROOM behavior.....	152
FIGURE 72.	ROOMChart state machine for component A	154
FIGURE 73.	ROOMChart modeling	156
FIGURE 74.	Different ways of addressing sets of scenarios	160
FIGURE 75.	Component behavior modeling in specification MSC.....	161
FIGURE 76.	Using a single MSC model	163
FIGURE 77.	Bypassing role model definition in the RT-TROOP modeling process ...	166
FIGURE 78.	RT-TROOP modeling using existing ROOM components.....	167
FIGURE 79.	RT-TROOP modeling using MSC requirements	169
FIGURE 80.	Structure of the mediator pattern	172
FIGURE 81.	RT-TROOP modeling using the mediator pattern.....	173

FIGURE 82.	From a set of scenarios to a set of component behavior	182
FIGURE 83.	Set of behavior integration patterns	184
FIGURE 84.	Main components of the ATM system	185
FIGURE 85.	ATM transaction UCM	187
FIGURE 86.	UCM for a withdraw transaction	187
FIGURE 87.	UCM for a deposit transaction	188
FIGURE 88.	Scenario partitioning representation	192
FIGURE 89.	Partitioning of the ATM scenarios	193
FIGURE 90.	Integration of control scenarios	201
FIGURE 91.	Operational state	202
FIGURE 92.	Transaction state	203
FIGURE 93.	ATM controller component behavior	203
FIGURE 94.	Adding the error handling scenarios to the ATM control component behavior	204
FIGURE 95.	Mode-oriented behavior with sequential mode toggling	209
FIGURE 96.	Mode-oriented behavior with explicit mode toggling	210
FIGURE 97.	TV/VCR remote control with sequential toggling	211
FIGURE 98.	Mode-oriented behavior with explicit mode toggling	211
FIGURE 99.	Structure of the scenario option state machine	216
FIGURE 100.	Structure of the scenario option state machine	218
FIGURE 101.	Operational state of the ATM control component behavior	219
FIGURE 102.	ExecuteOper state of the ATM control component behavior	219
FIGURE 103.	Scenario composition	224
FIGURE 104.	Waiting place	226
FIGURE 105.	Timed waiting place	228
FIGURE 106.	Scenario aborting	230
FIGURE 107.	PrintFile STD	236
FIGURE 108.	StopPrinting STD	237

FIGURE 109. PrintFile UCM	239
FIGURE 110. StopPrinting UCM	240
FIGURE 111. Related path sets of the PrinterSystem UCM model	242
FIGURE 112. PrinterSystem stubs.....	242
FIGURE 113. The PrinterSystem and its environment.....	243
FIGURE 114. PrintFile UCM	244
FIGURE 115. getResources UCM (S1)	244
FIGURE 116. releaseResources UCM (S2).....	245
FIGURE 117. StopPrinting UCM	245
FIGURE 118. Composite UCM Map of the PrinterSystem.....	246
FIGURE 119. Related path sets of the PrinterSystem UCM model	248
FIGURE 120. Generation of the PrintFile HMSC	249
FIGURE 121. Renamed PrintFile HMSC	249
FIGURE 122. setupPrinting basic MSC skeleton	251
FIGURE 123. getResources basic MSC skeleton	252
FIGURE 124. getNextChar basic MSC skeleton	252
FIGURE 125. printChar basic MSC skeleton	252
FIGURE 126. terminatePrinting basic MSC skeleton	253
FIGURE 127. releaseResources Basic MSC skeleton	253
FIGURE 128. StopPrinting basic MSC skeleton	254
FIGURE 129. setupPrinting basic MSC	255
FIGURE 130. getResources basic MSC	256
FIGURE 131. getNextChar basic MSC	257
FIGURE 132. printChar basic MSC	257
FIGURE 133. terminatePrinting basic MSC	258
FIGURE 134. releaseResources basic MSC	258
FIGURE 135. stopPrinting basic MSC	259
FIGURE 136. printFile role actors.....	261

FIGURE 137. printFile communication diagram.....	262
FIGURE 138. Definition of printFile protocol classes	263
FIGURE 139. ROOM role structure required for the printFile UCM	264
FIGURE 140. stopPrinting communication diagram.....	265
FIGURE 141. Definition of stopPrinting protocol classes.....	265
FIGURE 142. ROOM role structure required for the stopPrinting UCM.....	266
FIGURE 143. PrinterSystem actors	267
FIGURE 144. PrinterSystem protocol classes	268
FIGURE 145. PrinterSystem ROOM structure.....	269
FIGURE 146. setupPrinting customized MSC	270
FIGURE 147. getResources customized MSC	271
FIGURE 148. getNextChar customized MSC	271
FIGURE 149. printChar customized MSC	271
FIGURE 150. terminatePrinting customized MSC.....	272
FIGURE 151. releaseResources customized MSC	272
FIGURE 152. StopPrinting customized MSC.....	273
FIGURE 153. PrintFile HMSC	274
FIGURE 154. setupPrinting customized MSC	275
FIGURE 155. getResources customized MSC	276
FIGURE 156. getNextChar customized MSC	276
FIGURE 157. printChar customized MSC	277
FIGURE 158. terminatePrinting customized MSC.....	277
FIGURE 159. releaseResources customized MSC	278
FIGURE 160. StopPrinting customized MSC.....	279
FIGURE 161. printerDriver role behavior for the PrintFile MSC	280
FIGURE 162. Printer role behavior for the PrintFile MSC	284
FIGURE 163. printerDriver role behavior for the StopPrinting MSC.....	286
FIGURE 164. Printer role behavior for the StopPrinting MSC	288

FIGURE 165. printerDriver complete behavior model.....	289
FIGURE 166. Printer complete behavior model.....	291
FIGURE 167. StartUp STD.....	293
FIGURE 168. Shutdown STD.....	294
FIGURE 169. StartUp UCM.....	295
FIGURE 170. Shutdown UCM.....	295
FIGURE 171. StartUp specification MSC.....	296
FIGURE 172. Shutdown specification MSC.....	297
FIGURE 173. Control scenarios protocol classes.....	298
FIGURE 174. Control scenarios ROOM structure.....	298
FIGURE 175. PrinterSystem actors.....	299
FIGURE 176. PrinterSystem protocol classes.....	299
FIGURE 177. PrinterSystem ROOM structure.....	300
FIGURE 178. StartUp customized MSC.....	301
FIGURE 179. Shutdown customized MSC.....	301
FIGURE 180. Role behavior state machines for the StartUp scenario.....	303
FIGURE 181. Role behavior state machines for the Shutdown scenario.....	306
FIGURE 182. Control-level state machine for the PrinterDriver actor.....	309
FIGURE 183. Control-level state machine for the Printer actor.....	310
FIGURE 184. Resulting PrinterDriver ROOMChart for iteration 2.....	311
FIGURE 185. Resulting Printer ROOMChart for iteration 2.....	312
FIGURE 186. Modified version of the PrinterDriver ROOMChart for iteration 2.....	316
FIGURE 187. Modified shutdown customized MSC.....	319
FIGURE 188. Modified shutdown specification MSC.....	320
FIGURE 189. Modified Shutdown UCM.....	320
FIGURE 190. Modified Shutdown STD.....	321
FIGURE 191. Relationship between UCM and MSC.....	342

List of Tables

TABLE 1.	RT-TROOP modeling phases.....	92
TABLE 2.	Modified RT-TROOP modeling phases.....	165

CHAPTER 1 Introduction

1.1 Context and Motivations

Real-time systems represent very complex engineering artifacts that are used at all levels of activity in today's society: telephony, banking, health care, traffic control, manufacturing, avionics, aerospace, management systems, etc. The size and complexity of these systems is constantly increasing. Also, their reliability is of the first importance for the good working order of society. Failures can result in human death and important financial losses.

These systems are characterized by the following aspects:

- **Distribution.** A real-time system is composed of a set of components that are, in general, geographically distributed over different computing sites. These sites can be located arbitrarily far from each other. In order to provide the overall system behavior, system components need to communicate together.
- **Concurrency.** In real-time systems, concurrency exist both at the system level, where several scenarios¹ can execute concurrently, and at the component level, where system components can all be active simultaneously.

1. In this thesis, a scenario is a sequence of responsibilities that must be executed by the system and the user of the system in order to achieve a system functionality.

- **Event-Driven.** Real-time systems implement a variety of functionalities that can be invoked in an unpredictable way. The execution of these functionalities is triggered by external entities sending events to the system.
- **Real-time.** Real-time systems are time constrained. This means that in response to a given incoming event, the system must execute the appropriate functionality in an acceptable (finite) period of time.
- **Dynamics.** The structure of real-time systems dynamically changes over time, which means that components can be created, moved, and destroyed while the system is running. Moreover, the collaboration patterns between components also change dynamically as the system executes.

In real-time systems, the overall system behavior emerges from the collaboration of all individual components. Because of the different characteristics mentioned above, the overall behavior of real-time systems is usually very complex. The complexity of real-time systems comes from the combination of those factors. A major problem we, humans, have in the engineering of such systems is understanding and defining how the required system behavior is to be achieved by its components.

This difficult engineering problem has been studied by many researchers in the last few decades. Those research efforts have led to the definition of different techniques and methods in the area of modeling and design ([10], [11], [13], [17], [18], [36], [40], [44], [46], [47], [51], [74], [83], [87], [93]), simulation ([10], [23], [68]), verification ([10], [40], [41]), and testing ([10], [14]), [77]).

In the last decade, the object-oriented paradigm has emerged as a leading development technology in the real-time system industry. This paradigm allows modeling systems in a way that very much corresponds to the way humans see the world, i.e. in terms of a set of collaborating (or communicating) components, or objects. There currently exists several object-oriented methodologies ([11], [13], [47], [83], [93]). An important aspect of those methodologies is that they allow smoothing the transition between requirements and

implementation by combining the use of different types of models to express different views of the systems. Each of these models focuses on specific system aspects such as scenarios, inter-component communication, structure, component behavior, and inheritance. This enables designers to concentrate on different issues at different times. Models used in current methodologies include CRC cards ([8]), use cases ([13], [47]), UCM (Use Case Maps) ([18]), interaction diagrams ([13], [47]), MSC (Message Sequence Charts) ([43], [93]), different types of structure models ([13], [93]), hierarchical state machines ([13], [36], [83], [93]), activity diagrams ([13]), and class hierarchy models ([13], [83]).

Another important aspect of object-oriented methodologies is that they are well adapted to iterative development. The objective of iterative development consists in decomposing the overall set of requirements into subsets that can each be addressed in a different development cycle, called an iteration. Each iteration involves performing the five main activities of object-oriented development, i.e. requirement capturing, analysis, design, implementation and testing. Thus, all the different models must be revisited and modified at each iteration to reflect the new set of requirements. At the object level, the addition of new requirements to the system may result in the introduction of new objects in the system, or in the modification of the behavior of existing objects. The clear separation between object behavior (or internal logic) and system structure in object-oriented systems facilitates those two types of system modifications.

Overall, we can say that object-oriented technology provides a foundation to build complex real-time systems. The use of object-oriented technology in large industrial systems has demonstrated that it can help reducing the cost associated with maintenance and evolution ([32], [26]).

1.2 Problem Description

In spite of the fact that the object-oriented technology has allowed important progress in the development of complex real-time systems, significant problems remain. In this thesis, we focus on two of these problems: the lack of traceability between models, and the lack of model transition techniques. The two problems are closely related.

1.2.1 Lack of traceability

Because object-oriented methodologies combine the use of several different models, and because those models are constantly modified as new requirements are added, and as bugs are fixed, maintaining consistency between all the models is a key issue. It is, in general, difficult and expensive to do so. Yet, it is crucial. The lack of consistency makes models obsolete, and thus unusable. In the context of large industrial system development where teams of designers are involved, maintaining consistency between models constitutes a major problem.

One way to facilitate the maintenance of consistency between models consists in defining traceability relationships. *Traceability* is the property that defines how different models relate to each others in the context of a development process. It allows the linking of elements contained in different models. We identify two main reasons for why this type of traceability is particularly important: model consistency, and testing.

The existence of traceability relationships allows evaluating the impact of modifications on the different models, and making the changes to affected models in a consistent manner. Thus, if a modification is made, for example, to one scenario, designers can evaluate

where changes should be made in the different models. Reciprocally, if an error is found at the code level, we can trace it back to the different models, and ultimately to requirements, and see where the error has been introduced. Thus, overall consistency between models can be maintained.

The definition of traceability relationships is also very important for the purpose of testing. To check that a given model, say M1, is correct with respect to another model, say M2, we must first establish semantics relationships between elements of the two models. For example, if we want to verify the correctness of a set of communicating state machines with respect to a high-level scenario (such as a use case [47]), we must first establish a relationship between the high-level responsibilities (in terms of which the scenario is defined), and the actions and messages defined in the state machines. This is precisely what traceability relationships do.

The idea of traceability is not new. Its importance is widely recognized. In his book on object-oriented software engineering, Jacobson says:

"Traceability is a tremendously important property in system development. Each major system will be altered during its lifetime. Whether the changes emanate from changed requirements or responses to trouble shooting, we will always need to know where the changes need be made in the source code."

In spite of this, current object-oriented methodologies are weak on traceability. Most methodologies allow maintaining traceability between objects defined in the different models², but very few define traceability between other types of model elements, such as high-level responsibilities, messages, component interfaces, states, and transitions. It is our belief that the existence of fine-grained traceability relations between model elements increases system maintainability and extensibility [27].

2. This traceability relationship is based essentially on a naming convention that links objects defined in different models if they have the same name.

Literature on Traceability

Paradoxically, very few technical papers address this very important problem. In [47], Jacobson et al. define traceability relationships at two different levels: 1- traceability between use cases and interaction diagrams, more specifically a one-to-many traceability relationship between responsibilities in use cases and messages in interaction diagrams, and 2- traceability between objects contained in different models, e.g. objects contained in the analysis model and objects contained in the design model. In a similar context, Andersson and Bergstrand [5] define a traceability relationship between use cases and MSCs [43], and they suggest using SDL for system modeling. However, they do not discuss further the relationship between MSC and SDL.

Corriveau [27], and McGregor and Korson [66] define object-oriented development processes based on traceability. They highlight the importance of traceability relationships for integrated testing, and discuss important issues that must be considered in the definition of such relationships. However, because they do not define development processes in terms of specific models, they do not define concrete traceability relationships between models.

Others, like Haugen [39] and Koskimies et al. [51], discuss the definition of a concrete traceability relationship between interaction diagrams and flat state machines. They are mainly concerned with the definition of links between transitions and states in state machines, and messages in interaction diagrams. Similar traceability relationships are defined between MSC'96 and ROOM by Leue et al. [58].

Also, a special issue of Communications of the ACM journal [49] has recently been dedicated to requirements traceability. In that issue, different aspects of requirements traceability are discussed. Requirements traceability is defined in [29] as the ability to describe and follow the life of a requirement, in both a forward and backward direction. In his introductory paper, Jarke [49] identifies four different kinds of requirements traceability links: *forward from requirements*, *backward to requirements*, *forward to requirements*, and *backward from requirements*. Based on what Jarke considers to be most comprehensive

survey of traceability practice to date, Ramesh [79] identifies two categories of traceability user organizations: *low-end users*, who see traceability mostly as a costly defense against criticism and liability lawsuits, and *high-end users*, who see traceability as an investment in corporate knowledge asset management within and beyond information system engineering. The lack of traceability results in a decrease of overall system quality, and thus increases project costs and time [29].

To the best of our knowledge, there is currently no research project that addresses the traceability issue in the global context of going from high-level scenario models, in which concurrency and interactions between scenarios can be expressed, to communicating hierarchical state machines. This problem is a complex one. It requires considering issues like concurrency and interactions between scenarios, refinement of high-level responsibilities into actions and messages, allocation of responsibilities to components (or objects), definition of system structure, definition of interface components, structuring of hierarchical state machine to facilitate maintenance and extensibility, etc.

Our objective is to define traceability between concepts and notations used in existing modeling techniques, instead of in terms of an underlying formal language.

1.2.2 Lack of model transition techniques

An important issue that arises when combining several models into a single methodology lies in the transitions that are required to go from one model to another. Because object-oriented development requires using several different models when going from requirements to implementation, transitions between models constitute an important part of the development activities.

One way to facilitate the transitions between models consists in defining a set of model transition techniques. The role of those model transition techniques is to specify how the

information contained in one model can be used for the definition of another model, and what is the additional information that needs to be input by the designer. Such transition techniques can be particularly useful for inexperienced designers that are often faced with the difficult problem of how to start when several decisions need to be taken.

In current object-oriented processes, the transitions between models are conducted in an informal and adhoc manner. Some general heuristics have been proposed [11], [47], [93] to help taking certain decisions, but overall model transitions are weakly defined. This makes moving from one model to another error prone. It also complicates the uniform application of a development process in the context of large system development as different designers relates model elements in different manners.

Literature on Model Transition Techniques

There are several papers in the literature that discuss a transition between a pair of models. Jacobson et al. [47], and Andersson and Bergstrand [5] discuss the transition between use cases and some types of interaction diagrams. They establish a relationship between use case responsibilities and messages in the interaction diagrams. However, they do not define a more global approach that would allow making the transition between the two models in a systematic manner.

Other papers discuss automatic transitions between some types of interaction diagrams and state machines. The work of Haugen [39] and Koskimies et al. [51] focuses on the definition of a transition between interaction diagrams and flat state machines. [39] defines a general approach for making the transition between an MSC model [43] and an SDL model [45]. [51] defines a synthesis algorithm that allows for the automatic generation of state machines from a set of scenario diagrams. Leue et al. [58] defines a similar approach for MSC'96 and ROOM hierarchical state machines. Synthesis algorithm can be useful for the generation of test components (test actors in ROOM) in which the structuring of the state machine is not important. In this case, only the black box behavior matter.

However, in our opinion, such algorithms are not suitable for the design of complex real-time system components. We see three main problems with these automatic synthesis methods: 1- they require all design decisions to be taken at the interaction diagram (or MSC) level, 2- they do not allow considering nonfunctional requirements, and 3- they give no flexibility for structuring hierarchical state machines³. Behavior structuring is a main issue in complex component design since it constitutes an important factor of maintainability and extensibility.

In the context of requirement engineering, Somé [95] defines a method that incrementally constructs timed automata from scenarios with timing constraints. This method is based on a semi-formal language where scenario are described as natural language sentences shaped by a formal syntax. The generated automata are used to analyze requirements.

Methods for the automatic synthesis of state machines have also been defined in the context of artificial intelligence ([50], [70]) and control theory ([6], [78]). In the context of artificial intelligence, [50] defines an algorithm that generates reactive plans, defined by means of finite state machines, from goals defined using modal temporal logic formulas [63]. This paper also reviews important papers that address the problem of generating reactive plans for discrete-event reactive systems. Also in the context of artificial intelligence, [70] defines two algorithms that generate state machines with minimal number of states from a specification described by means of sequences of input/output strings. This paper also reviews several important papers that address the same problem. In the context of control theory, Ramage and Wonham [78] define an approach that aims at synthesizing the behavior of controllers for discrete-event systems, defined by means of state machines, from formal specifications. In [6], Barbeau et al. define a synthesis method that generates timed transition graphs from temporal logic formulas. This method addresses safety, liveness, and real-time constraints. Interested readers can find in [6] a description of the basic controller synthesis problem as well as a review of important papers on the subject.

3. Automatic generation algorithms will be further discuss in section 1.4.3 and in Chapter 4

In a subfield of formal methods, several researchers worldwide, like Schot [87] and Pires [74] at University of Twente, and Rumpe [84] at Munich University of Technology, have defined development methods based on the concept of correctness preserving transformations. The *correctness preserving transformation (CPT)* approach consists in moving from a high-level system specification to an implementation using exclusively a set of predefined transformations that have been proved to be correctness preserving. Because each transformation establishes an explicit relationship between elements of the input model and elements of the transformed model (or more precisely between the model's elements input into the transformation and the model's elements that result from the application of the transformation), the use of the CPT approach ensures a complete traceability between requirements and implementation. However, one of the main drawbacks of this approach is that designers can only use transformations that have been proved to be correctness preserving, i.e. transformations contained in the CPT catalogue. This is considered an important limitation of the approach.

In the context of formal verification, [3], [4], [7], [16], [41], and [59] define transition methods that allow generating formal models for the purpose of model verification. The objective of these transition methods is to automatically generate formal representations that can be verified using formal verification tools. They map elements of the input model into related elements in the output model. We believe that this approach can be useful in complex system development to verify critical aspects of systems.

In this thesis, our objective is to define a set of model transition techniques that allow for a systematic transition between models, not to automate the transition between models. We believe that in the design of complex real-time systems there are too many decisions to be taken, and those decisions are too complex to be completely automated. Complex real-time system design is a creative process. Designers can adapt to new problems, and find creative solutions to those problems. Automatic methods cannot adapt to new problems.

1.3 Objectives

This thesis addresses the difficult problem of defining a systematic and traceable progression between requirement level scenario descriptions and communicating hierarchical state machine models in the context of complex real-time system design.

The objectives of the thesis are:

- To define a modeling process and a set of design patterns that allow making the transition between requirements and communicating hierarchical state machines in a systematic and traceable manner
- To integrate UCM in a concrete modeling process
- To establish a set of traceability relations between the different models used in the modeling process
- To develop case studies to highlight important issues, and to illustrate the different concepts and methods developed in the thesis

1.4 Thesis Contributions

This thesis makes five main contributions to software engineering and real-time system development: 1- definition of the RT-TROOP modeling process, 2- integration of the UCM modeling technique in a concrete modeling process, 3- definition of a set of hierarchical state machine design patterns, 4- definition of traceability relations between the

models of RT-TROOP, and 5- development of case studies. These contributions are described in the five next sections (section 1.4.1 to section 1.4.5).

1.4.1 Definition of the RT-TROOP Modeling Process

Most of the design methodologies used in the real-time system industry today, e.g. ROOM [93], SDL [46], StateChart [36], UML [12], aims at producing communicating state machine models from requirements. One of the difficulties faced by designers when using such methodologies lies in the large gap that exists between requirements and communicating state machines. For this reason, scenario based approaches ([18], [47]) have become very popular in the object-oriented community. Those approaches allow smoothing the transition between requirements and communicating state machines.

In this thesis, we define a systematic and traceable modeling process, called RT-TROOP (Real-Time TRaceable Object Oriented Process) that allows making the transition between scenario textual descriptions, defined at the requirement level, and communicating hierarchical state machines in the context of complex real-time system design. This process is defined in terms of a set of models and a set of modeling phases.

The RT-TROOP modeling process has already been implemented in an industrial development process at CML Technologies⁴, and is being implemented in a second development process at CRC (Communication Research Center). These two industrial implementations of the RT-TROOP modeling process will be briefly discussed in section 7.2.

4. CML Technologies is a 150 employees company specialized in the development of telecommunication systems such as mobile radio consoles, air traffic control communications, Enhanced 9-1-1 emergency calling systems, and other specialized switching systems for customized computer telephony applications.

Models

The RT-TROOP modeling process combines the use of four different modeling techniques: STD, UCM, MSC, and ROOM. The first three are used to model scenarios at different level of details, while ROOM is used to model communicating hierarchical state machines.

- STD (Scenario Textual Description) is our own template for describing scenarios at the requirement level. This template is similar to the one used in [86] to describe use cases.
- UCM [19] is a modeling technique to graphically describe scenarios in terms of sequences of high level responsibilities in the context of component structure diagrams. It also allows for the explicit description of concurrency and interactions between scenarios. The use of UCM in industrial projects indicates that it is a powerful technique for modeling complex systems. The UCM modeling technique is supported by the UCM navigator tool [65].
- The ITU standard MSC [44] is a modeling technique that allows the description of system scenarios in terms of sequences of messages exchanged between system components, and between the system and its environment. This modeling technique is widely used in the industry, and is implemented in many CASE tools.
- ROOM [93], which is now part of UML-RT (Unified Modeling Language for Real-Time), is a simple and powerful modeling technique that allows describing real-time systems both at a schematic level, using structure diagrams and hierarchical state machines, and at a detailed level, using programming languages like C++ and Smalltalk. This methodology is supported by the ObjecTime toolset, which provides modeling, simulation, and code generation. It is used in many high-tech companies worldwide.

We believe that the different concepts and methods defined in this thesis could also be adapted to other modeling techniques, such as UML [13] and SDL [46].

Modeling Phases

The objective of the RT-TROOP modeling process is to decompose the overall set of design issues that must be addressed in the design of real-time systems into subsets that can be addressed in different modeling phases.

The RT-TROOP modeling phases include both *model transition phases*, which define the steps that should be carried out when making the transition between models, and *in-model modeling phases*, which define the modeling activities that may take place in the different models. This thesis does not aim at making contributions to the in-model modeling phases per se. Each of these modeling techniques is the subject of numerous papers that describe how they can be used.

What the RT-TROOP does though is integrate such techniques into a single modeling process. For this purpose, we define a set of model transition phases. In this thesis, we consider the transition between:

- STD and UCM
- UCM and MSC
- MSC and ROOM structure models
- MSC and ROOM behavior models, called ROOMCharts

Each of these transitions is decomposed into a set of steps that each addresses a different set of modeling issues, such as definition of messages, definition of communication protocols, definition of hierarchical state machines, etc. While most of these transitions can be partly automated, they all require adding new design information. Thus, the process as a whole remains a creative one.

With respect to the definition of the RT-TROOP process, the thesis contribution is a first cut at a comprehensive process going from requirements, or more specifically from a set of scenario textual descriptions, to an executable model from which implementation can

be automatically generated. Some parts of this process are described at a more detailed level than others, and constitute the technical contributions of this work.

1.4.2 Integration of the UCM Modeling Technique in a Concrete Modeling Process

The UCM modeling technique has been successfully used in several industrial projects in the last decade. In [18], the authors discuss the relationships between UCM and other modeling techniques such as collaboration graphs, class hierarchy diagrams, interaction diagram, and visibility graphs. However, the UCM modeling technique has not yet been integrated in a concrete modeling process. In the context of the unification of modeling techniques, where UML becomes the standard, such integration is crucial for the future of UCM.

The RT-TROOP modeling process defined in this thesis provides such integration. It defines a concrete method for making the transition between UCM and MSC'96 [44]. This method defines at a detailed level how elements of UCM models relate to elements of MSC models. The RT-TROOP modeling process also defines how the scenario interaction information contained in UCM models can be used to build hierarchical state machine models from scenario models (see section 1.4.3).

Because of the close semantic relationships that exists between ROOM and MSC models, and UML models, we believe that the research results of this thesis provide the basis for the integration of the UCM modeling technique in UML.

The integration of the UCM modeling technique in a concrete modeling process that allows moving from requirements to implementation is an important contribution of this thesis.

1.4.3 Definition of a Set of Hierarchical State Machine Design Patterns

Scenario models and communicating hierarchical state machine models provide two orthogonal views of real-time systems. The former describes system behavior as sequences of responsibilities that need to be executed by components in order to achieve overall system objectives, while the later describes complete component behavior in terms of states and transitions.

One of the most crucial and complex phases of real-time system design lies in the transition that is required to go from system behavior (defined by means of scenario models) to component behavior (described by means of communicating hierarchical state machine models). Among the factors that contribute to this complexity are⁵:

- Large number of scenarios
- Concurrency and interactions between scenarios
- Scenarios of different types
- Dynamic modification of scenarios
- Unpredictability of external events
- Incompleteness of scenario models
- Maintainability and extensibility of component behavior

When defining the behavior of a component from a set of scenarios, all these factors must be considered. Moreover, designers must also consider other nonfunctional requirements, such as performance and robustness. Therefore, in order to meet design objectives, designers must synthesize all the information contained in the scenario models, consider the non-

5. These factors are discussed in more details in section 4.1.

functional requirements, and produce a set of hierarchical state machines that altogether satisfy the overall requirements of the system. Without a rigorous approach, this transition is error prone.

In the current literature, some papers, like [51], [58], and [69], define methods, based on synthesis algorithms, that perform automatic transition between message sequence diagram and state machines. Such method allows completely automating the transition between message sequence diagrams and state machines. Their main advantage, beside the fact that they perform automatic generation of state machines, is that they allow maintaining a complete traceability between scenario models and state machine models. They also ensure the correctness of the state machines with respect to the scenarios described in the message sequence diagrams. However, as previously discussed, they do not consider several important issues related to nonfunctional requirements, and concurrency and interactions between scenarios.

In this thesis, we define a different approach, based on *patterns*, that allows considering those issues. The use of standard design patterns ([24], [35], [67], [96]) has rapidly increased in the industry in the last few years. This approach consists in defining a set of solutions that can be applied by designers when facing specific design problems. However, to our knowledge, there exist no patterns to help designers in the definition of communicating hierarchical state machines from scenario models.

This thesis proposes a set of behavior integration patterns that deals with the following issues: scenario partitioning, state machine composition (integration), mode-oriented behavior, mutually exclusive scenarios, and different types of scenario interactions that include scenario composition, scenario aborting, scenario interaction through waiting places, and scenario interaction through timed waiting places.

This part of the project is conducted in four steps:

- Definition of the patterns in terms of a problem, a context, and a solution.

- Documentation of the patterns, using modeling techniques such as UCM, MSC, and ROOM.
- Illustration of the patterns using concrete examples.
- Evaluation of the consequences of the patterns.

The definition of design patterns to help designers making the transition between scenario models and hierarchical state machines is an important contribution of this thesis. There are, to our knowledge, no other research projects that address this problem in the context of concurrent and interacting scenarios. It is our belief that such design patterns will benefit both experienced and inexperienced designers.

1.4.4 Definition of Traceability Relations

Traceability relations provide the glue that allows combining a set of different models in a single modeling process. These traceability relations allow maintaining consistency between the different models. The definition of fine-grained traceability relations between scenario textual descriptions, UCM, MSC, and communicating hierarchical state machines, defined using ROOM, constitutes one of the main contributions of this thesis.

In order to define traceability relations between different models, it is necessary to understand clearly the different concepts and notations that are used in each of those models. The concepts used in existing scenario models include the following: triggering and resulting event, pre and post conditions, system states, responsibilities, messages, internal actions, scenario interactions, scenario concurrency, component, allocation of responsibilities to components, dynamic creation and destruction of components, and dynamic modification of scenarios. These concepts, or subset of them, are expressed using different notations in different modeling techniques. On the other hand, communicating hierarchical state machine modeling techniques use the following concepts: component, interface

component, communication links, state, composite state, (transition, state entry, and state exit) actions, messages, etc. Again, these concepts are expressed using different notations in different modeling techniques. In the definition of traceability relations between scenario models and hierarchical state machines, all those aspects must be considered.

This thesis defines three different traceability relations: *inter-model traceability*, which establishes traceability links between elements of different models, *inter-version traceability*, which establishes traceability links between elements of different versions of a single model, and *design decision traceability*, which establishes traceability links between the rationale of a design decision and specific requirements that led to (or that justify) the decision.

These relations could be implemented in a tool to facilitate the maintenance of consistency between models. This would allow evaluating the impact of modifications on the different models.

1.4.5 Development of Case Studies

In this thesis, printer system case study is developed to illustrate the different phases of the RT-TROOP modeling process. The development of this simple system allows us to illustrate the systematic and traceable aspects of RT-TROOP modeling. This case study is conducted in two iterations to demonstrate the iterative nature of our modeling process. This case study also gives examples of how the behavior integration patterns can be used, and how the fine-grained traceability information maintain throughout the process can be used to resolve design problems resulting from undesired scenario interactions.

In the context of this research project, several other systems have also been developed both in an industrial (Nexus project at CML Technologies) and academic (PBX, fax

machine, ATM system, Train Traffic Control System, House Security System [73]) context.

The development of these systems also constitutes an important contribution of the thesis research work.

1.5 Thesis Outline

The thesis is structured as follows.

In Chapter 2, we describe the different modeling techniques used in this thesis. It includes scenario textual descriptions (STD), UCM, MSC and ROOM. For each of these modeling techniques, we first give a brief overview of their main characteristics, and then describe the set of concepts and notations that are used in this thesis.

In Chapter 3, we define the RT-TROOP modeling process. This process combines the use of the four models described in Chapter 2. It allows moving from a set of scenario textual descriptions to a ROOM model in a systematic and traceable manner. This chapter defines the role of the different models used in RT-TROOP, and describes the modeling phases that compose it. It also discusses the use of the RT-TROOP in different context.

In Chapter 4, we define a set of patterns that focus specifically on the design of hierarchical state machines from large sets of scenarios.

In Chapter 5, we use a simple printer system to illustrate how the RT-TROOP modeling process can be applied in practice. The different modeling phases of RT-TROOP are illustrated and discussed in relation with the different modeling issues associated with them.

In Chapter 6, we define a set of traceability relations between STD, UCM, MSC, and ROOM.

Finally, in Chapter 7, we summarize the thesis contributions, discuss their implementation in industrial contexts, discuss their impact on software engineering and tool development, and give a list of topics for future research.

CHAPTER 2 Modeling Techniques and Notation

In this chapter, we describe the notations and modeling techniques used in this thesis. This includes: the format for requirements, which is composed of a requirements table and a set of scenario textual descriptions (STD), the Use Case Maps (UCM) modeling technique, the Message Sequence Charts (MSC) modeling technique, and the ROOM modeling technique. For each of these, we first give a brief overview of their main characteristics, and then describe the set of concepts and notations that are used in this thesis.

It should be noted that only a subset of the concepts and notation of the modeling techniques is used in this thesis. Our primary objective is to show how a set of different modeling techniques can be combined in a single traceable development process, not to define a complete mapping between modeling techniques.

One of the aspects that have been left out of the thesis, for sake of conciseness, is the one of structure dynamics. However, the three modeling techniques we use allow modeling structure dynamics. Also, the current implementations of the RT-TROOP modeling process include structure dynamics. The integration of structure dynamics in the RT-TROOP modeling process is described in [72]. Readers interested in the structure dynamic notation are referred to [18], [44], and [93].

2.1 Requirements

Requirements describe the needs or desires for a product, i.e. what the system should do. Requirements may be divided into two different categories: functional requirements, that describes system services or functions, and nonfunctional requirements, that describe the constraints under which the system must operate and the constraints that apply to the development process. The process of defining requirements is called requirement engineering. It constitutes the first phase of any development process. Requirements engineering is, by itself, a complex process, which we do not cover in this thesis.

The final product of requirement engineering is a *requirement document*. The format and composition of this document vary from one methodology to another. The requirement document may contain different types of descriptions such as: a general system description, list of functional and nonfunctional requirements, textual description of scenarios (like Jacobson use cases [47]), definition of external actors, a list of prescribed system components, etc. These descriptions may be more or less formal depending on the methodology.

The requirement document used in this thesis is composed of a *requirements table* and a set of *scenario textual descriptions (STDs)*. The format we use for these is described in the next two sections.

2.1.1 Requirements Table

In order to enable requirement traceability, requirements must be written in a traceable format. A simple way to achieve this consists in writing the requirements in a structured

tabular format, and in attaching to each requirement a unique identifier that can be referenced in other documents or models. In order to maintain backward traceability to the different stakeholders and system description documents from which requirements are extracted, individual requirements can also be linked to sentences or paragraphs in the document from which they originate.

The definition and maintenance of traceability links between requirements and system description documents are part of requirements engineering responsibilities, and therefore are not explicitly addressed in this thesis.

Format of Requirements Table

In this thesis, we use the table format of Figure 1 as a template for requirement description. It associates a unique identifier of the form FR*i* (Functional Requirement *i*) or NFR*i* (Non-Functional Requirement *i*) with each requirement.

Functional and nonfunctional requirements can also be placed in different tables. However, because this thesis is concerned with the traceability aspect of the requirement table, and not with its structuring, the structuring issues are irrelevant. Therefore, our only requirement, with respect to the requirement table, is that it associates a unique identifier with each traceable requirement.

FIGURE 1. A simple traceable format for requirements

ID	Description
FR1	
FR2	
FR3	
FR4	
NFR1	
NFR2	
NFR3	

A third traceability column could also be added to the requirements table to allow for backward traceability to sentences or paragraphs in the system description documents (elements of previous documents).

2.1.2 Scenario Textual Description (STD)

Scenario textual descriptions aim at describing system scenarios in a structured textual format. This type of scenario description, which has been popularized by Jacobson under the name of *use cases* [47], has received a wide level of acceptance in the object-oriented community. STDs are used to organize requirements on a scenario basis. Because of their textual and high level nature, scenario textual descriptions facilitate communication between clients (or stakeholders) and system designers.

In this thesis, we postulate that system scenarios are described in the requirement document in the form of STDs. Defining the system scenarios is a task that is part of requirement engineering, and therefore is outside the scope of this thesis. Readers interested in high level scenario (or use cases) description are referred to [47] and [86]. In this thesis, STDs constitute the starting point of our modeling process.

Because we want scenario descriptions to contain specific elements, we define our own format of scenario textual description, simply called STD. Each STD is defined in terms of the following elements:

- A unique *identifier*
- A brief textual *description* of the overall objective of the scenario
- A set of *external actors* that participate in the scenario
- A set of possible *triggering events*
- A *Precondition* that must be satisfied in order to enable the execution of the scenario
- A sequence of *responsibilities* (or steps) that defines the scenario
- A *Postcondition* that must evaluate to true after the execution of the scenario
- A set of possible *resulting events*
- A set of *alternative scenarios*
- A set of *nonfunctional requirements* that apply to the scenario
- A *comment* section that may be used by designers as a free format text window to specify different issues related to the scenario

An important aspect of STDs is that they group together a main scenario with a set of alternatives. This allows defining in a single logical entity a set of closely related scenarios. We call this a *scenario cluster*. Because of its highly cohesive nature, a scenario cluster offers high potential of reuse as a building block in different scenario models. For example, when a software architect builds a telephony application that requires a “make call” scenario, it is natural to think that this scenario will come with a set of alternatives, like invalid dialed number and busy call, that may occur while executing the “make call” scenario. An analogy can be made with component-based architecture in which components encapsulate behavior that can deal with both normal operation and error cases.

In order to allow for traceability, each STD element is labelled with a unique identifier that can be used for reference in other models. Also, each STD element may be explicitly linked to related requirements in the requirement list. Linking STD elements to require-

ments allows maintaining traceability between scenario descriptions and the overall set of system requirements.

We call the *STD model* the set of overall STDs that specifies the system. This set is incrementally built through the iterations.

STD Format

We use the table format of Figure 2 as a template for textual scenario description (STD). In this figure, a description of an ATM (Automatic Bank Teller Machine) withdraw transaction is given. The last column of the table is used to maintain requirement traceability information (see Section 6.2).

FIGURE 2. ATM withdraw transaction STD

STD Identifier: ATM withdraw transaction	
Description: Describes the steps of a normal withdraw transaction	
External Actors: User, Central Bank System (CBS)	
Precondition: ATM is idle	
Triggering event: A user insert a valid bank card	
<ol style="list-style-type: none"> 1. User enter a valid bank card. 2. ATM swallow the bank card and reads card information. 3. ATM initiates the transaction. 4. ATM asks the user to enter PIN. User enters PIN. 5. CBS validates PIN. 6. ATM asks user to choose a transaction option. User chooses the withdraw option. 7. ATM asks for amount to withdraw. User enters amount. 8. ATM sends a withdraw transaction request to CBS. 9. CBS verifies that the user account balance is sufficient to cover the requested amount. 10. CBS registers the withdraw transaction. 11. ATM dispenses cash. User picks up cash. 12. ATM prints a transaction receipt. User picks up the receipt. 13. ATM returns the bank card. User picks up the card. 	
Postcondition: ATM returns to its idle state	
Resulting event: ATM returns the bank card	
Alternatives: <ul style="list-style-type: none"> - If the user enters three successive invalid PINs, then the transaction is refused and the card is not returned to the user. - If the user account balance is insufficient, then the transaction is refused. - If the ATM does not have enough cash, then the transaction is refused. 	
Nonfunctional requirements: <ul style="list-style-type: none"> - A transaction must be completed in less than two minutes - ATM can only handle one transaction at the time. 	
Comments: <ul style="list-style-type: none"> - A transaction can be cancelled at any time before the transaction is sent to the CBS. 	

2.2 UCM

UCM (Use Case Map) [18] is a high level scenario modeling technique defined for real-time object-oriented system design. It is based on a simple and expressive visual notation that allows describing scenarios at an abstract level in terms of sequences of responsibilities¹ over a set of components.

The primary objective of the UCM modeling technique is to capture and analyze system behavior at an abstract level; UCM describes scenarios at an abstraction level that is above both inter-component communication and detailed level component behavior. It allows focusing on individual scenario description, scenario interaction, and responsibility allocation, before introducing inter-component communication. As is, UCM models can be used as a specification for the modeling of inter-component communication.

UCM also provides two very important features:

- It allows superimposing scenarios on system structure

This enables designers to visualize scenarios in the context of a system structure. It also provides a mechanism by which responsibilities can be allocated to system components.

- It allows combining sets of scenarios in a single diagram

This enables designers to express scenario clusters and scenario interactions in a graphical manner. It also provides a mechanism that can be used by designers to analyze the overall system behavior that emerges from scenario combinations.

1. In UCM, responsibilities are described using informal textual descriptions.

The UCM modeling technique also provides notations to describe the structure dynamics aspect of real-time systems. However, this aspect of UCM is not explicitly addressed in this thesis.

In this section, we describe the UCM terminology and notations used in this thesis. Additional notations required in the different case studies will be described as they are used. We also briefly describe the development and analysis of UCM models. Readers interested in more details are referred to [18].

2.2.1 Basic UCM Notation

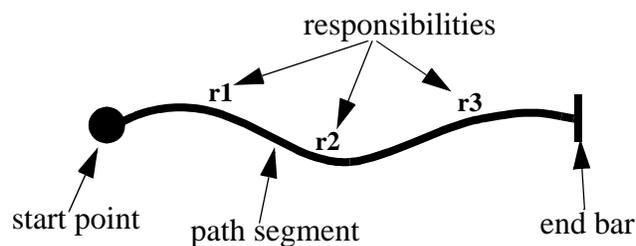
A *use case path* represents a path along which scenarios flow in the system. They express the sequences of responsibilities that need to be executed by system components in order to achieve the overall objective of the system in response to a given triggering event.

In this section, we first describe the basic notations used to describe paths, and then we describe how system components can be introduced in UCM maps.

Basic Path Notation

In Figure 3, the basic elements that compose a use case path are illustrated.

FIGURE 3. A simple UCM path



Start Point

The execution of a use case path begins at a start point. A start point is illustrated in UCM by means of a filled circle placed at the origin of a path segment (see Figure 3). A start point is defined by means of a set of possible triggering events and an optional precondition. A triggering event is defined by means of a unique identifier and an optional list of input parameters that can be associated with the event. If a precondition is specified, this precondition must evaluate to true to enable the execution of the path. Otherwise, if no precondition is specified, then by default it is set to true, and therefore the start point will be triggered each time that the triggering event occurs.

Formally, a start point is specified in terms of:

- A set of possible *triggering events*
- A *precondition* that need to be satisfied to enable the triggering of the path

Each triggering event is defined by a unique identifier and an optional list of input parameters associated with the event.

Responsibilities

As previously mentioned, a use case path describes a sequence of responsibilities that need to be executed by system components in response to a given triggering event. At the UCM modeling level, these responsibilities are high level ones. A responsibility is usually defined by means of a responsibility identifier and short textual description that describes in prose the nature of the responsibility. Thus, at this stage, responsibilities remain informal elements of a system model that need to be more precisely defined in later stages of the development process.

Responsibilities are visually illustrated in UCM by means of responsibility identifiers placed along path segments.

Formally, a responsibility is defined in terms of:

- A *responsibility identifier*, which allows uniquely identifying a responsibility in a UCM model, and
- A *responsibility description*, which gives a textual description of the responsibility

To avoid the creation of cumbersome UCMs, the responsibility identifiers that are placed along the path segments are usually short identifiers, i.e. two or three characters (letters and digits). These identifiers are usually inexpressive, e.g. r12. For this reason, a good practice consists in defining a responsibility index in each UCM map. This index provides the name of the responsibility associated with the identifier together with a short textual description.

Path segment

A path segment expresses an ordered sequence of path segment elements that need to be executed by system components. It is visually illustrated in by means of a curve joining together the sequence of path segment elements. A path segment may be composed of zero or more path segment elements.

A path segment element is either:

- A *responsibility*, or
- A *waiting place* (see Section 2.2.4), or
- A *stub* (see Section 2.2.4)

End bar

The execution of a path terminates at an end bar. An end bar is visually illustrated in UCM by means of a thick perpendicular line placed at the end of a path segment. An end bar is defined by means of an optional resulting event and a postcondition. A resulting event is defined by means of a unique identifier and an optional list of output parameters associated with the event.

Formally, an end bar is specified in terms of:

- A *Postcondition* that must hold after the execution of the path, and
- A set of possible *resulting events*

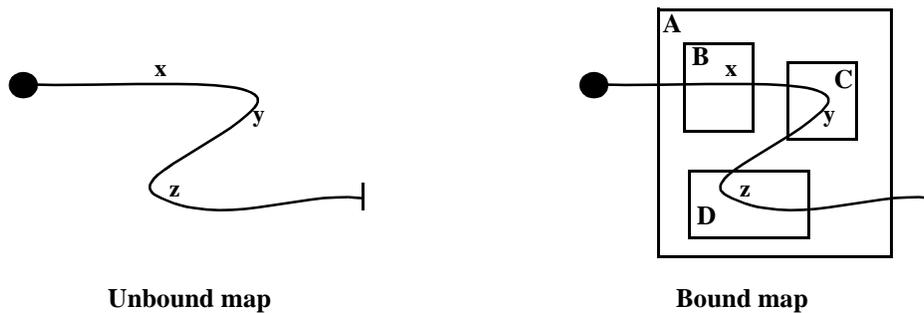
Each resulting event is defined by a unique identifier and an optional list of output parameters associated with the event.

Path

A complete UCM path may be composed of one or more path segments connected together by means of segment connectors (see definition of segment connectors in Section 2.2.3). The first path segment of a path must start with a start point, and the last segment of a path must terminate with an end bar.

2.2.2 Introducing Components in UCM

A designer can also use the UCM modeling technique to describe paths in the context of the system structure. This is done by superimposing paths on a system structure as illustrated in the right diagram of Figure 4. In UCM, components are visually illustrated using labelled rectangles. At this level, the system structure is only defined as a set of components; inter-component communication is not yet defined. We call such a diagram a *bound use case map*, or simply *bound map*.

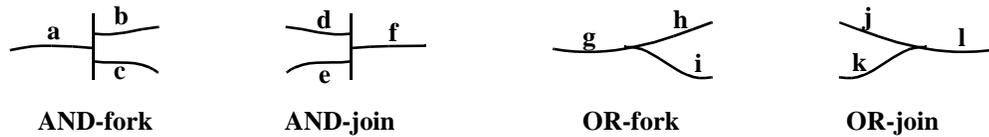
FIGURE 4. Unbound and bound maps**Responsibility allocation**

In bound maps, responsibilities are allocated to components. For example in Figure 4, responsibility *x* is allocated to component **B**, responsibility *y* is allocated to component **C**, and responsibility *z* is allocated to component **D**.

2.2.3 Path Segment Connectors

In addition to purely sequential paths, as the one illustrated in the previous section, the UCM notation can also be used to describe more complex cases that involve concurrent path segments or alternative ones.

To describe such cases, the UCM modeling technique uses path segment connectors. A path segment connector is either an AND-fork, an AND-joint, an OR-fork, or an OR-joint. These connectors are illustrated in Figure 5. In this figure, each path segment is labelled with a different identifier. The execution of the four path diagrams given in this figure goes from left to right.

FIGURE 5. Path segment connectors**AND-fork**

An AND-fork is used to illustrate a point along a path where the execution of a single path segment forks into the execution a set of two or more concurrent path segments. Thus, the AND-fork connector is syntactically and semantically of the form one-to-N. Syntactically, it is one-to-N because it connects together one path segment on its incoming side to N path segments on its outgoing side. Semantically, it is one-to-N, because the termination of the execution of the one path segment on its incoming side results in the concurrent execution of N path segments on its outgoing side.

The semantics of the AND-fork given in Figure 5 is as follow. Once the execution of path segment **a** is completed, then the concurrent execution of path segments **b** and **c** may start.

AND-join

An AND-join is used to illustrate a point along a path where several concurrent path segments synchronize together and result in the execution of a single path segment. The AND-join connector is syntactically and semantically N-to-one. Syntactically, it is N-to-one because it connects together N path segments on its incoming side to one path segment on its outgoing side. Semantically, it is N-to-one because the completion of the execution of the N path segments on its incoming side results in the execution of the one path segment on its outgoing side.

The semantics of the AND-join given in Figure 5 is as follow. Once the execution of path segment **d** and **e** is completed, then the execution of path segment **f** may start.

OR-fork

An OR-fork is used to show a point along a path where alternative branches may be followed. Each branch is associated with a distinct path segment. Thus, the OR-fork connector is syntactically one-to-N, because it connects together one path segment on its incoming side to N path segments on its outgoing side. However, unlike the AND-fork, it is semantically one-to-one, because the execution of the one path segment on its incoming side results in the execution of only one segment on its outgoing side. The OR is an exclusive OR. The path segment that will be executed on the outgoing side depends on the context. The path segments on the outgoing side are usually guarded by expressions that are defined in terms of system states.

The semantics of the OR-fork given in Figure 5 is as follows. Once the execution of path segment *g* is completed, then the execution of path segment *h* or *i* will be triggered. Thus, the OR-join diagram illustrates two possible paths: one formed by path segments *g-h*, and one formed by path segments *g-i*.

OR-join

An OR-join is used to illustrate a point along a path where two or more incoming path segments merge into a single one without requiring any synchronization or interaction between the incoming path segments. The OR-join connector is syntactically N-to-one; it connects two or more path segments on its incoming side to only one path segment on its outgoing side. However, it is semantically one-to-one; the execution of any of the path segments on its incoming side results in the execution of the path segment on its outgoing side.

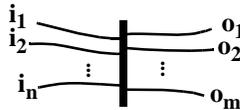
The semantics of the OR-join given in Figure 5 is as follows. The execution of either path segment *j* or *k* will result in the execution of path segment *l*. Thus, the OR-join diagram illustrates two possible paths: one formed by path segments *j-l*, and one formed by path segments *k-l*.

Generalized Path Segment Synchronization Connector

In this thesis, we also define a generalized path synchronization connector which can be used to illustrate cases where an arbitrary number, say N , of incoming path segments need to synchronize together to trigger the execution of an arbitrary number, say M , of outgoing path segments. This connector is somehow a generalization of the AND-fork and AND-join segment connectors previously described. This connector is illustrated in Figure 6. This connector is syntactically and semantically N -to- M .

The semantics of the diagram of Figure 6 is as follow. Once the execution of the N incoming path segments (i_1, i_2, \dots, i_n) is completed, then the executed of the M outgoing paths (O_1, O_2, \dots, O_m) starts

FIGURE 6. Generalized path segment synchronization connector



Combination of Path Segment Connectors

This set of connectors can be combined together to describe more complex paths. Some examples of the type of path constructions that can be described by combining these connectors are illustrated in Figure 7.

FIGURE 7. Combination of path segment connectors



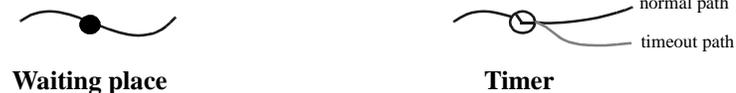
2.2.4 Other Path Notations

Two other types of UCM notations are used in this thesis: waiting place and stub.

Waiting Places

Waiting places are used to indicate a point along a path where the progression of the path is blocked until a predefined unblocking event occurs. We identify two different types of waiting places: a regular waiting place, simply called waiting place, and a timed waiting place, called timer. These are illustrated in Figure 8 and described below.

FIGURE 8. Waiting places



Waiting Place

A regular waiting place identifies a point along the path at which the progression of a path is blocked until a predefined unblocking (or triggering) event occurs. After the unblocking event is received, the progression (execution) of the path may continue. Since regular waiting places have no notion of time, their use can lead to path deadlock, i.e. the path may wait for the unblocking event forever. Visually, waiting places are illustrated using filled circles placed along a path.

Waiting places are used to illustrate points along paths where interactions with other paths or with the environment of the system occurs. Waiting places can be associated with both synchronous and asynchronous interactions (see Section 2.2.6). A starting point constitutes a special use of waiting place.

Timer (or timed waiting place)

A timer is a special type of waiting places that will only wait for a certain period of time before continuing on. Thus, the use of timer prevents the occurrence of deadlock. Timers are visually represented by clock-like icon placed along a path.

Timers are usually followed by an or-fork connector that illustrates the two alternate paths that could be taken after the timer; one that illustrates the case where the expected unblocking event occurred before the timeout occurred (normal path), and the other that illustrates the case where the timeout event occurred first (timeout path).

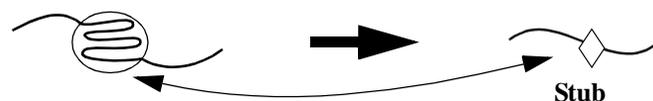
A timer can also be used along a path without being followed by an OR-fork to introduce time delay on the execution of the path.

Stub

The UCM modeling technique also provides a mechanism for path abstraction, called stub. A stub illustrates part of a path that is abstracted in the context of the map in which it is used. In a UCM model, the expansion of the stub is either described in separate maps, or remains to be defined later when details will be added to the UCM model. Stubbing constitutes an important mechanism for iterative development. It also reduces the cluttering of models by hiding details that are less important in the context of a given map.

An example of path stubbing is given in Figure 9. On the left side of the figure, a detailed path is shown. On the right side of the figure, the stubbed version of the same path is given. We observe that the path segment enclosed in the circle, in the left diagram, has been collapsed into a stub. A UCM tool could for example allow to see the stub expansion by double-clicking on the stub.

FIGURE 9. Stub



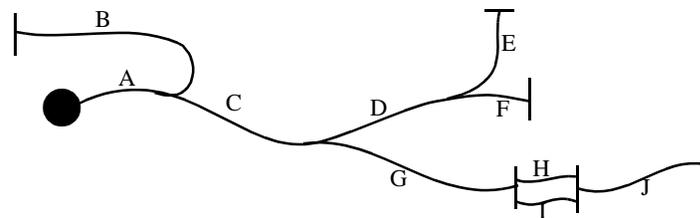
2.2.5 Related Path Set

In this thesis, we call *related path set* a set of paths that can be triggered from a single starting point. Formally, a related path set is composed of:

- A starting point,
- A set of path segments,
- A set of segment connectors, and
- A set of end bars (each end bar is associated with a distinct terminating path segment).

An abstract example of a related path set is given in Figure 10. This related path set expresses four different possible paths: A-B, A-C-D-E, A-C-D-F, and A-C-G-(H ||| I)-J².

FIGURE 10. A UCM related path set



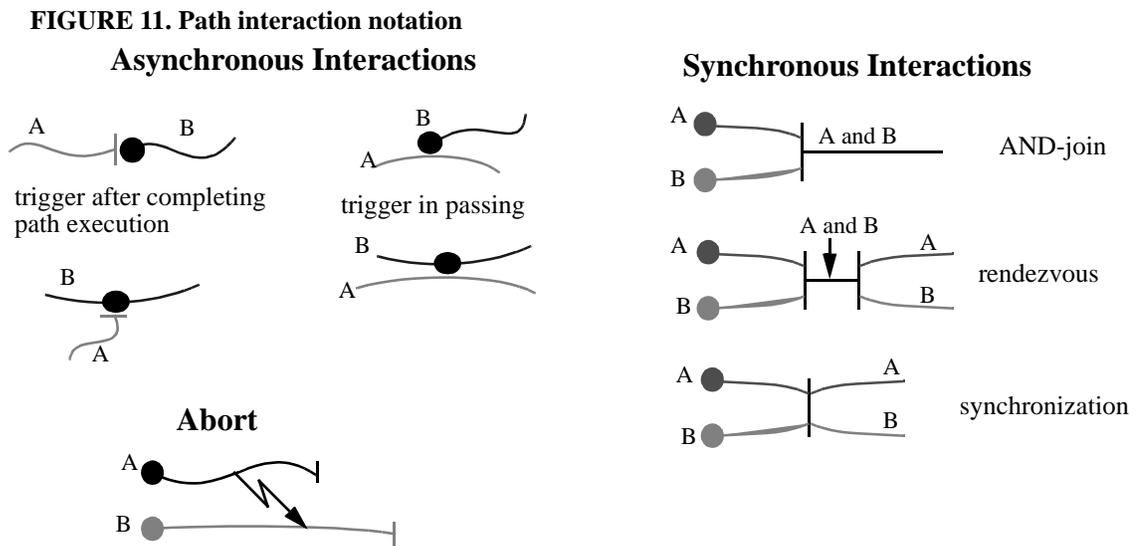
Because a related path set expresses a set of paths that can be triggered from a single triggering event, it constitutes a cohesive logical entity. A good use of related path sets consists in grouping a path that describes a main functionality of the system with the set of possible alternative paths. Alternative paths may include paths such as exception paths and error paths. Once defined this type of related path set can be reused in the development of other systems that intend to provide the same type of functionality.

The concept of related path set plays a central role in this thesis. It is particularly important in the transition between UCM and MSC.

2. (H ||| I) means that path segments H and I are executed in parallel.

2.2.6 Path Interaction Notation

An important aspect of the UCM modeling technique is that allows describing of path interactions. The different UCM notations used to describe path interactions are illustrated in Figure 11.



Asynchronous interactions

Trigger after completing path execution

This type of interaction is used to illustrate cases where the completion of the execution of a path triggers another path that is waiting on a waiting place. The waiting can either be a start point or a waiting place along a path. Both cases are illustrated in Figure 11.

To couple two paths, say path A and path B, together in a trigger-after-completing-path-execution manner, the following rule must be respected: the resulting event of path A must correspond to the triggering event of path B, and the postcondition of path A must satisfy the precondition of path B.

Trigger in passing

This type of interactions is used to illustrate cases where a waiting place, positioned either at the beginning (start point) or along a path, is triggered by another path in an asynchronous manner.

To couple two paths, for example A and B in Figure 11, together in a trigger-in-passing manner, there is an implicit send-triggering-event responsibility defined along path A at the point where it passes beside the waiting place. The triggering event sent by path A corresponds to the one defined in the waiting place at which path B is waiting.

Synchronous interactions**AND-join**

This type of interaction is used to illustrate cases where the synchronization of two, or more, paths results in the execution of a single one. As described previously (see Section 2.2.3) this type of interaction can be generalized for N incoming paths resulting in M outgoing ones.

Rendezvous

This type of interactions is used to illustrate cases where two, or more, paths synchronize together to execute a certain path segment (sequence of responsibilities) before returning to the execution of their own respective path.

Synchronization

This type of interactions is used to illustrate cases where two, or more, paths synchronize together and then return to the execution of their own respective path.

Abort

The abort notation is used to illustrate cases where the execution of a path, for example path A in Figure 11, interrupts the execution of another, path B in Figure 11.

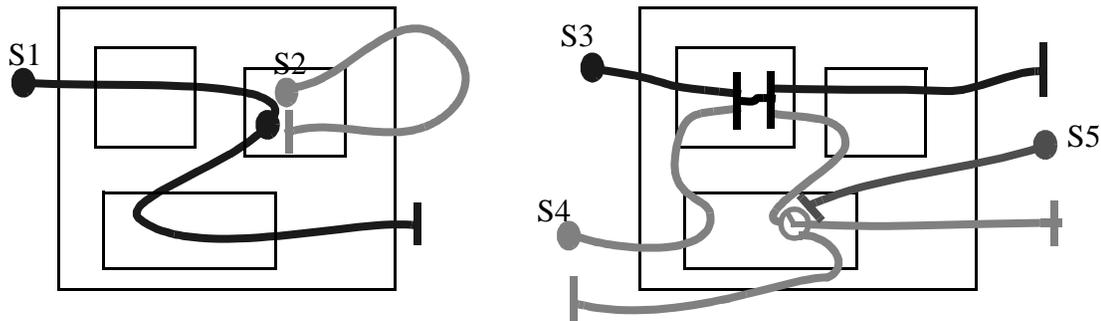
2.2.7 Composite Use Case Maps

An important aspect of the UCM modeling technique is that a set of paths can be coupled together in more complex diagrams, called *composite use case maps* (or simply *composite maps*). This allows expressing interactions and concurrency between sets of paths that are contained in different related path sets.

In composite maps, the interaction between paths is visually expressed using path interaction notation (see Section 2.2.6). This capability of grouping sets of paths into complex composite UCMs constitutes an important aspects of UCM modeling.

Two abstract examples of composite maps are given in Figure 12. In the first example (left side of Figure 12), scenario S1 triggers scenario S2 in passing, and then waits for the completion of scenario S2 before continuing its execution. In the second example (right side of Figure 12), a composite map illustrating more complex inter-scenario relationships between scenarios S3, S4, and S5 is given. The second example illustrates the use of a synchronous interaction and a timer (timed waiting place).

FIGURE 12. Composite use case maps



2.2.8 UCM Model

A *UCM model* is composed of a set of *UCM maps* (simply labelled UCM_i in figures throughout the thesis) and a set of *UCM related path sets*.

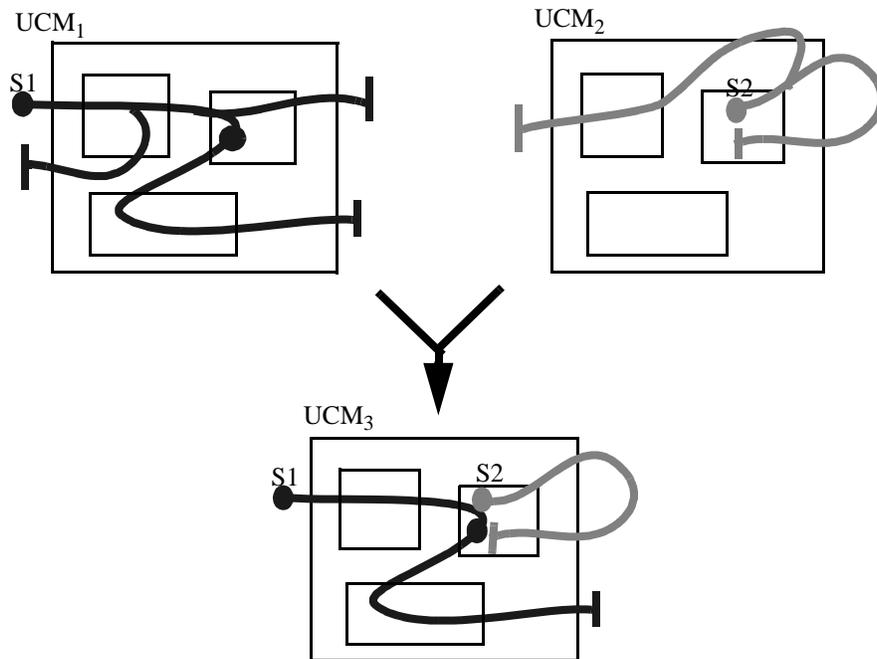
A UCM map is either a simple map, which describes a single related path set, or a composite map, which describes system relationships among a set of paths. Formally speaking, a UCM map is composed of:

- A set of paths
- A set of path interactions (possibly empty)
- A set of components (possibly empty)
- A responsibility allocation relation that links responsibilities to components (if there are components)

Related path sets constitute the building blocks of UCM maps. They are the basic elements from which composite maps are built. It should be noted that a related path set can be involved in several UCM maps to illustrate different path relationships. But, each path in a UCM model is uniquely contained in a related path set, i.e. each path belongs to exactly one related path set.

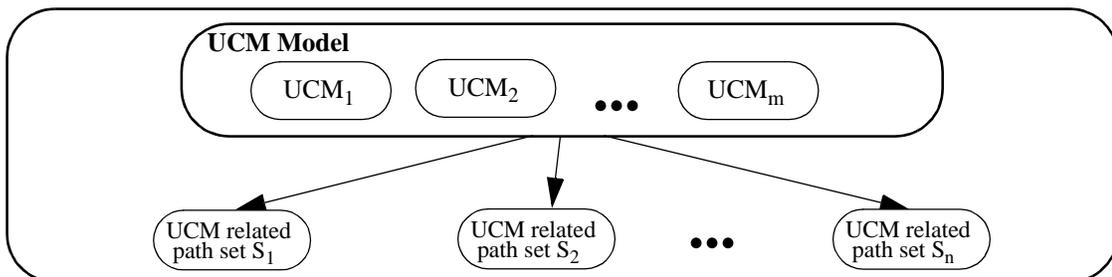
In Figure 13, we illustrate the definition of a composite map from two related path sets. In this figure three UCMs are defined: UCM_1 , UCM_2 , and UCM_3 . UCM_1 and UCM_2 are two simple maps that each contains a related path set, which are respectively labelled S_1 and S_2 . UCM_3 is a composite map that defines a relationship between a path of S_1 and a path of S_2 .

FIGURE 13. Related path sets and composite maps



The relationship between UCM model, UCM maps (simply labelled UCM_i), and UCM related path set is illustrated in Figure 14.

FIGURE 14. Composition of a UCM model



UCM Model Versions

Many different versions of a UCM model are usually defined in a development process. For example, a different version may be defined for each iteration in an iterative process. When needed, we refer to a specific version, say version n , of a UCM model using `UCM_modeln`.

2.2.9 UCM Model Analysis

Once defined, a UCM model can be used for requirement analysis. It allows analyzing the set of system scenarios at a global level before stepping into more detailed modeling phases.

In the analysis of the UCM model, designers must first ensure that:

- All the scenarios described in the requirements have been addressed at a sufficient level of detail.
- The model is correct with respect to the intention of the user (or client).
- There are no inconsistencies in the model and/or requirements³.

For the purpose of validation, the UCM model can be used to communicate with the clients and among development teams to make sure that the requirements have been well understood.

Also at this stage, techniques such as symbolic execution can be used to verify the correctness of the high level design model and to identify issues that will need to be solved in later phases of the development process. For this purpose, techniques such as the one pre-

3. Consistency is difficult to prove in practice, in the case of real systems, without the use of some formal models. However, a certain level of consistency check can be provided using UCM by analyzing the different maps. A more complete consistency check is discussed in [3] and [4].

sented in [3] can be used. This technique consists in generating a LOTOS model from the UCM model, and then to use different LOTOS tools to:

- Execute the model to discover the high-level behavior of the system.
- Verify that the UCM model is correct with respect to different system properties such as ambiguities, unspecified issues, race conditions, and inconsistencies.
- Check if the composition of several scenarios result in the emergence of new undesired behavior paths.

Formal verification at this stage of the development process can allow discovering feature interaction problem that would otherwise only be discovered at the detailed level modeling stage or at the implementation testing stage. The discovery of errors in early development stages significantly reduces development time and cost.

Research to provide UCM with an executable semantics is under way ([3], [4]). However, since the focus of this thesis is on the integration of the UCM modeling technique in a global modeling process, and not on requirement analysis, this specific topic will not be further discuss in this thesis.

2.3 MSC

The ITU Z.120 standard MSC (Message Sequence Chart) language constitutes a well known and widely used description techniques in the real-time system development industry. MSCs, or variations of them that appeared under different names like *Interaction Diagrams*, *Time Sequence Diagrams*, *Temporal Message Flow Diagrams*, have been used in the industry for many years in a wide range of applications including telephony, traffic control, industrial process control, and aerospace.

The role of MSC is to describe sequences of messages that are exchanged between system components in order to achieve system functionalities. MSCs can be used in early stages of real-time system design to express system behavior requirements. They can also be used to illustrate the result of system execution (or simulation).

The main objective of the MSC modeling technique is to describe end-to-end message sequences that need to be implemented by the system.

Since its standardization in 1993 [43], MSC has evolved from a simple message sequence description language that had the power of expressing purely sequential behavior, to a more complete description language [44] that has the power of expressing more complex system behavior using constructors like alternatives, parallel composition and loop. In other words, in its first version, MSC had the power to express individual scenarios, while in its current version, it has the power of expressing sets of related scenarios in clusters.

The current version of MSC [44], sometimes called MSC'96, is composed of different types of MSC diagrams: basic MSCs and HMSCs⁴.

In this section, we first describe the MSC notation used in this thesis, for both basic MSC and HMSC, and then discussed the particularities of MSC in the context of this thesis. When the notation used in the thesis differs from the one of the standard MSC [44], we give both notations and justify our choice.

2.3.1 Basic MSC

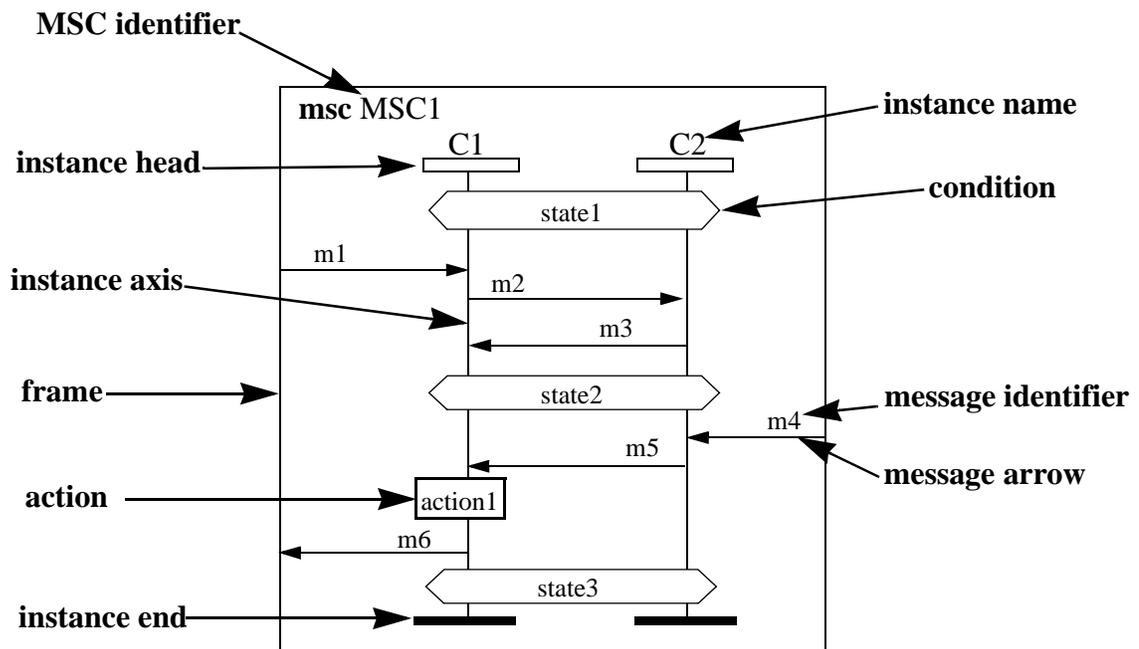
A *basic MSC* describes a sequence of messages exchanged between system components or between the components and the environment of the system. An example of a basic

4. In this thesis, the acronym HMSC is used to refer to high-level MSC.

MSC describing a standard message sequence is given in Figure 15. In this figure the different elements of a basic MSC are identified.

In basic MSC, time increases from top to bottom.

FIGURE 15. Basic MSC notation



Frame

A basic MSC is enclosed within a *frame* which delimits the boundary of the system in the context of the current MSC specification.

Identifier

The MSC *identifier* is placed at the top left corner of the frame following the reserved word **msc**.

Instance

In a basic MSC, each system component that participates in the execution of the message sequence is represented by means of a component *instance*, which is graphically composed of an *instance head*, an *instance axis* and an *instance end*.

Message

A Message is represented by a directed arrow, called *message arrow*, that goes from the sender instance to the receiver instance. The sender side of the message arrow is referred to as *message_out* and the receiver side is referred to as *message_in*. The *message identifier* is placed on top of the arrow. A message exchanged between a system component and the environment of the system is connected to the system component on one side and to the system boundary on the other side. In this case, the frame represent the environment of the system.

Action

In addition to messages, MSC also allows specifying *actions*. An *action* corresponds to an internal activity executed by a component. In MSCs, actions are only described by means of an *action identifier* associated with a free format textual description. The textual description is informal, but it should be given in structured and descriptive manner.

Condition

In MSC, a state is represented by means of a *condition*. A condition may apply to the complete set of component instances, in which case we refer to it as a system state, or only to a (non-empty) subset of them. Ultimately, it may apply to only one instance, in which case we refer to it as a component state.

The semantics of a condition is as follows. If a message arrow is preceded by a condition, then the condition must evaluate to true before the message can be triggered. If no condi-

tion is specified before a message arrow, then the condition is implicitly considered to be true, and the message may be triggered at any time by the arrival of the triggering signal. In MSC modeling, conditions can be used to restrict how basic MSC can be composed in HMSC (Section 2.3.2).

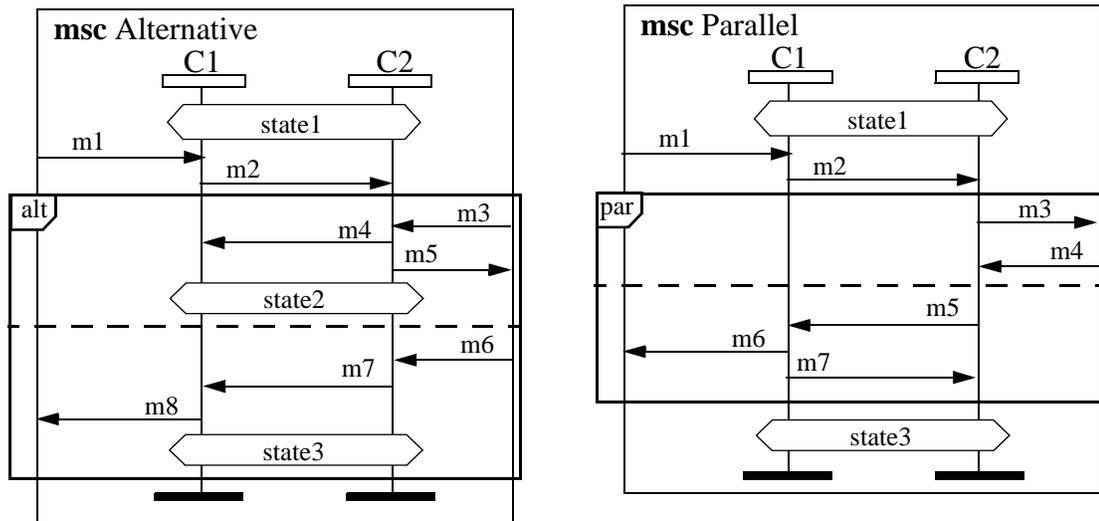
Inline Expressions

Basic MSC also provides notations for inline specification of alternative composition, parallel composition, iteration, optional region and exception. Examples of inline alternative composition and parallel composition are given in Figure 16. Both of these inline expressions are graphically represented by an inline expression box with their respective keywords, **alt** or **par**, placed in the top left corner section of the box. Also, inside the inline expression box, they both use dashed lines to separate operands. The semantics of the expressions is as follows. In the alternative expression only one of the operand will be executed, while in the parallel expression all the operands will be executed in parallel.

The Alternative MSC of Figure 16 should be interpreted as follows. The system is in state **state1** until component **C1** receives message **m1** from the environment, which will cause message **m2** being sent by component **C1** to component **C2**. Then, the MSC reaches the alternative section entered. The semantics of the alternative inline expression is such that after receiving message **m2** from component **C1**, component **C2** will either receive message **M3** or **M6** from the environment. Then depending on which message is received first, component **C2** will either send messages **M4** and **M5**, and go to state **state2**, or will execute messages **M7** and **M8** and go to state **state3**.

The interpretation of the Parallel MSC is similar to the previous one until it reaches the inline expression box. As previously mentioned, in the case of the parallel expression, all operands will be executed in parallel. In the example, it means that the two sequences of messages contained in the inline expression, i.e. sequence **M3**, **M4** and sequence **M5**, **M6**, **M7**, will be executed in parallel. Once the execution of both sequences is completed, the system enters state **state3**.

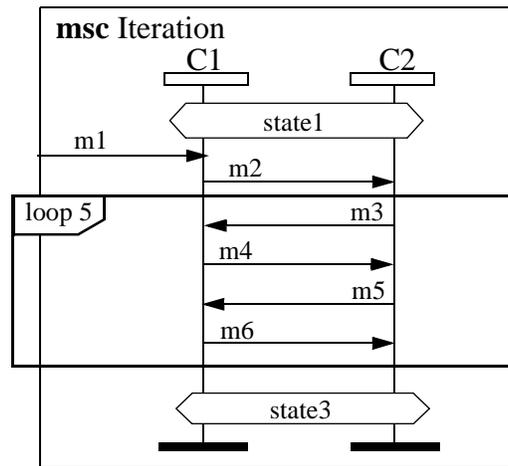
FIGURE 16. Alternative and parallel inline expressions



An example of the iteration inline expression is illustrated in Figure 17. The iterative section, which in this case is composed of messages m3, m4, m5 and m6, is placed in an inline expression box with the key word **loop** followed by the number of iterations, which in this case is 5. In the general form of the iteration inline expression, the number of iteration is specified as a pair of operands $\langle n, m \rangle$ which implies that the iterative section will be executed at least n times and at the most m times. The keyword **inf** can be used to specify unbounded iterations, like **loop** $\langle n, \text{inf} \rangle$. If only one operand is specified, as in the current example, then the number of repetition is fixed. If no operand is specified, then it is interpreted as **loop** $\langle 1, \text{inf} \rangle$.

The basic MSC of Figure 17 should be interpreted as follows. The MSC starts in state **state1** and exits this state when component **C1** receives message **m1** from the environment. Component **C1** responds by sending message **m2** to component **C2**, which will bring the system to the iterative section. Then the iterative section, which is composed of messages **m3**, **m4**, **m5** and **m6**, is repeated 5 times after which the system enters in state **state3**.

FIGURE 17. Iteration inline expression



Optional regions and exceptions can also be represented using a similar inline box notation with the keyword **opt** and **exc** in the top left corner section of the inline expression box. The **opt** operator describes a section of a basic MSC that is only executed under certain conditions. The **exc** operator is the same as an alternative where the second operand is the entire rest of the MSC.

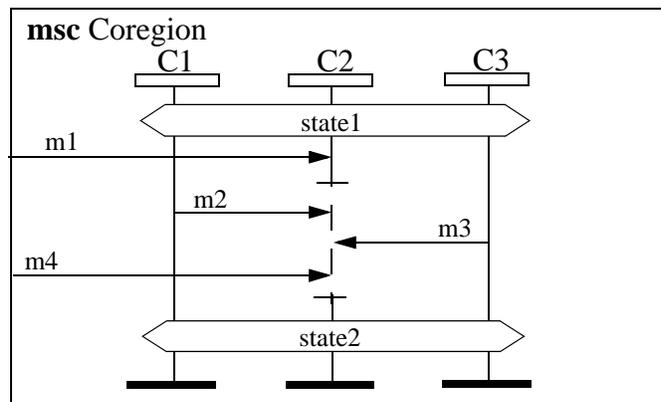
Coregion

The total ordering of events along an instance axis is not always appropriate. Sometimes, in MSCs, it is convenient to specify an unordered set of events. This allows to specify a “don’t care” with respect to message ordering. In MSC the concept of coregion allows the specification of such unordered set of messages. A coregion is graphically represented in a basic MSC by placing a dashed line section along an instance’s axis. An example of a coregion specification is given in Figure 18.

The interpretation of Figure 18 should be as follows. The system is in state **state1** until component **C2** receives a message **m1** from the environment. Then, component **C2** waits for the reception of message **m2** from component **C1**, message **m3** from component **C3** and message **m4** from environment. These three messages can be received in any order.

Once the three messages are received, then the system goes to state **state2**.

FIGURE 18. Coregion

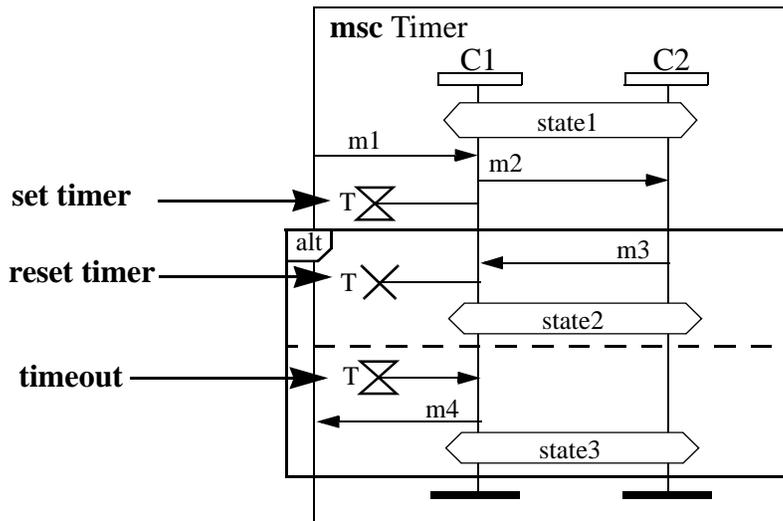


Timer

Since MSC was developed for the specification of real-time systems, basic MSC provides for the specification of timer and timer operations. An example of use of timer specification is given in Figure 19. In this figure, we identify the three types of timer operations (or messages): set timer, reset timer and timeout.

The MSC of Figure 19 should be interpreted as follows. The system stays in state **state1** until component **C1** receives a message **m1** from the environment. Then, component **C1** sends a message **m2** to component **C1**, set timer **T**, and wait for the arrival of either a message **m3** from component **C2**, or a timeout message from timer **T**. The alternative inline expression is used to specify the alternative. If message **m3** arrives first, then the timer is reset and the system is placed in state **state2**. Otherwise, if the timeout message arrives first, then a message **m4** is sent to the environment and the system is placed in state **state3**.

FIGURE 19. Using timer in basic MSC



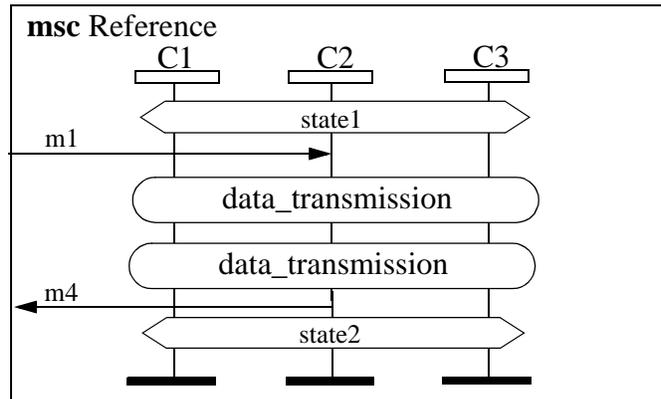
MSC Reference

An MSC reference is used to refer to other MSCs of the MSC model. It may refer to both basic MSCs and HMSCs. An MSC reference provides a mechanism that allows for abstraction and reuse. It allows use of existing MSCs in the definition of new ones, thus avoiding duplication.

The composition of MSCs within other MSCs by means of MSC references is guided by a set of composition rules defined in [44]. The description of these rules is beyond the scope of this thesis.

An example of MSC reference is given Figure 20. In this MSC, after the reception of message m1 by component C2, components C1, C2, and C3 enter two successive data transmission phases after which C2 returns message m4. In this MSC, the details of data transmission are abstracted. Also, both `data_transmission` MSC references refer to the same MSC, which only has to be written once.

FIGURE 20. MSC reference



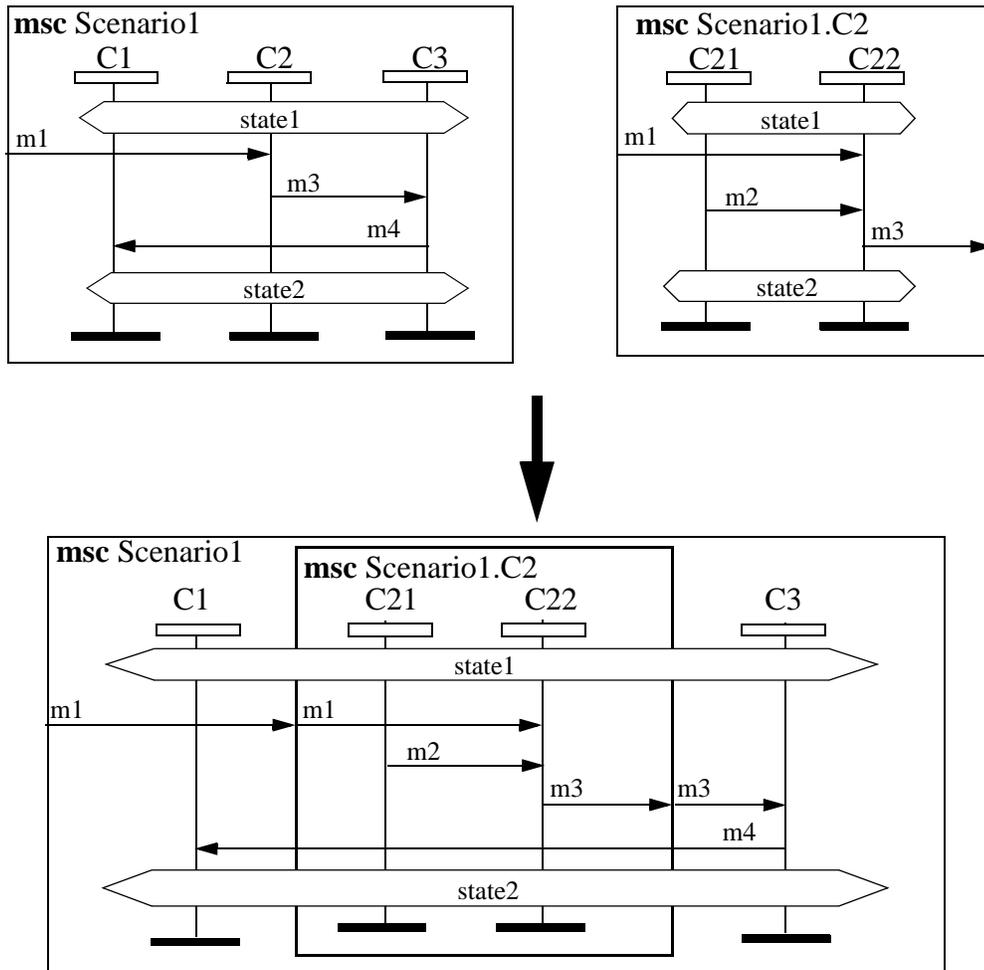
Component Decomposition

Finally, since the modeling of any complex system requires component decomposition, the basic MSC notation also provides support for instance decomposition. This means that any instance in a basic MSC can be decomposed and expressed using a separate basic MSC. In this thesis, we go a step further by allowing the graphical composition of basic MSCs together with instance decomposition MSCs in single MSC diagrams.

An example of instance decomposition is given in Figure 21. In this example, the decomposition of instance C2 is given by the basic MSC Scenario1.C2. This basic MSC is composed of two instances: C21 and C22. When decomposing an instance, one must ensure that all the messages that are sent and received by the instance in the former MSC can also be handled by internal instances in the decomposed MSC. In the present case, we observe that message m1 that is received by C2 in MSC Scenario1 is internally received by C22 in the decomposed MSC Scenario1.C2, and message m3 that is sent by C2 in Scenario1 is internally sent by C22 in the decomposed MSC.

The MSC resulting of the composition of MSC Scenario1 and Scenario1.C2 is given at the bottom of Figure 21.

FIGURE 21. Instance decomposition



2.3.2 HMSC

One of the main difference between traditional MSC (as described in the [43]), and other similar trace diagrams, and the new MSC standard described in [44] is the addition of the HMSC (High level MSC).

The primary goal of HMSC is to compose sets of basic MSCs into more complex message sequences. HMSC provides a means to graphically define how a set of MSCs can be com-

bined. A HMSC can be seen as road map, or flowchart, in as much as that they express the possible sequences of basic MSCs that can be executed as the result of an input message (event). In [44], an HMSC is defined as a graph in which each node is either: a start symbol, an end symbol, an MSC reference, a condition, a connection point, or a parallel frame.

While traditional MSC are limited to the description of one scenario at the time, MSC'96, with the basic MSC inline expressions and HMSCs, has the ability to express scenario clusters (i.e sets of related scenarios).

In Figure 22, the main elements that compose a HMSC are identified.

Identifier and Condition

MSC identifiers and *conditions* used in HMSC have the same semantics as the one used in basic MSC.

Start

Start symbol indicates the starting point of a HMSC.

MSC Reference

MSC references are as discussed in Section 2.3.1. They may refer to either a basic MSC or to a HMSC.

HMSC Reference Connectors

The elements of HMSC can be connected together using either sequential composition (Figure 22), alternative composition (Figure 22), and parallel composition (Figure 24).

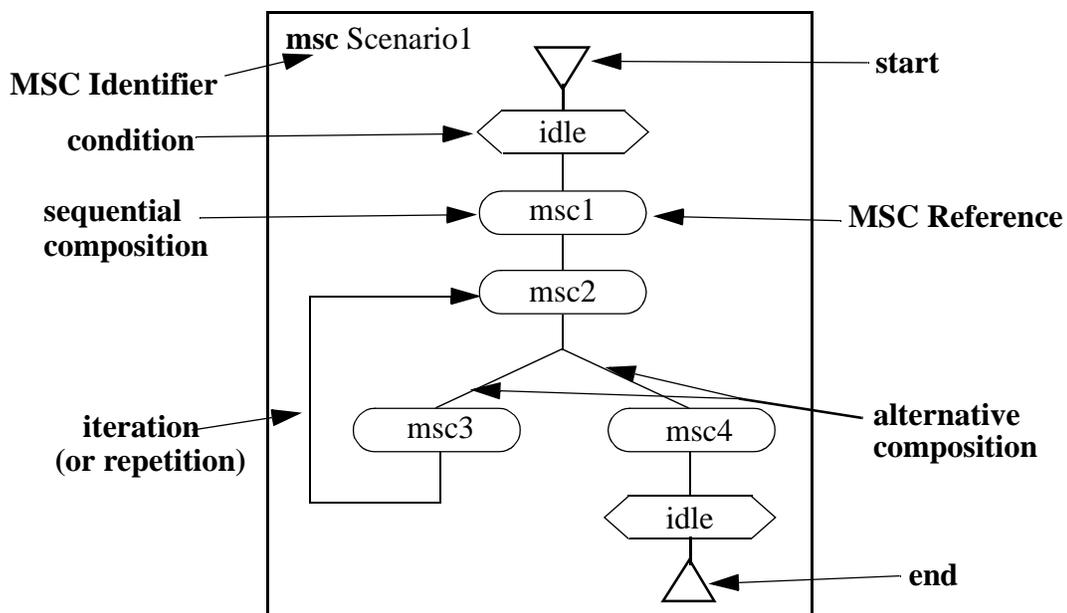
Stop

A *stop symbol* is placed at the end of each possible path in an HMSC.

The scenario expressed in Figure 22 can be described as follows. First, the system, which starts in the state *idle*, successively executes *msc1* and *msc2* before getting at the alternative connection point. Then, it either executes *msc3* and loops back to *msc2*, or it executes *msc4* and gets back to state *idle* before terminating. The alternative taken after the execution of *msc2* may be either guarded by the terminating condition of *msc2* and the initial conditions of *msc3* and *msc4*, or, if no such condition exists, it is determined by the first incoming message after the execution of *msc2*.

Thus, conditions may be used to guard the composition of MSCs in HMSCs.

FIGURE 22. High level MSC (HMSC)



Parallel Composition

In this thesis, we do not use the standard MSC'96 notation for HMSC parallel composition. Standard HMSC uses a nested box expression for parallel composition (Figure 23). The problem with this notation is that it becomes cumbersome when used to describe complex cases. For this reason, we chose to replace it by the ones given in Figure 24. This new MSC parallel composition notation has been inspired by the end-fork and end-join

notation of UCM. The semantics of the end-fork (left hand-side diagram of Figure 23 and Figure 24) is that after executing *msc1*, both *msc2* and *msc3* will be executed in parallel. The semantics of the end-join (right hand-side diagram of Figure 23 and Figure 24) is that the execution of *msc6* will only start after both *msc4* and *msc5* have completed their execution.

FIGURE 23. HMSC parallel composition defined in standard MSC'96

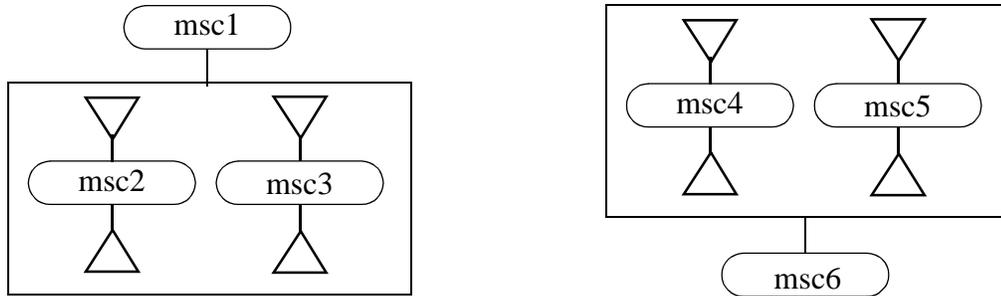
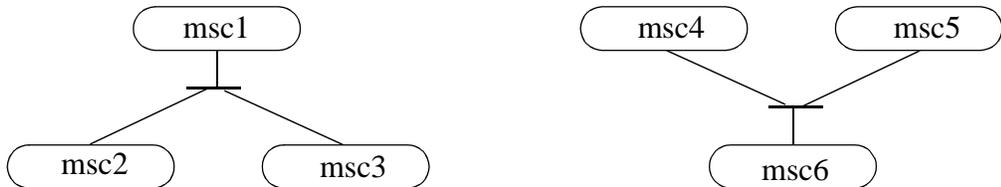


FIGURE 24. HMSC parallel composition used in this thesis



2.3.3 Additional MSC Concepts and Notations

In addition to the standard MSC notation previously described in this section, we introduce three new elements for basic MSC: *contract identifier*, *data type box* and *synchronous message arrow*.

Contract Identifier

In ROOM, the sending and the reception of messages requires the specification of both a message identifier and a port identifier that indicates the port by which the message is sent. Thus, in order to maintain a strong traceability between MSC message arrows and ROOM transitions, we need to add information to the message arrows that allows the identification of the ports by which the messages are exchanged. However, since the sender and the receiver use ports that have, in general, different identifiers, the use of port identifiers would require the addition of two port identifiers per message arrow. We find this solution too clumsy.

In the RT-TROOP modeling process, we propose to use ROOM *contract identifiers* to identify ports on message arrows. As defined in [87], a ROOM *contract* is composed of a pair of ports and the binding that links them. Since contract identifiers constitutes unique identifiers, they can be used to specify in an unambiguous manner the pair of ports associated to a given message

Thus, in RT-TROOP, both the message and the contract identifiers are attached to MSC message arrows. Graphically, the message identifier is placed above the arrow while the contract identifier is placed below.

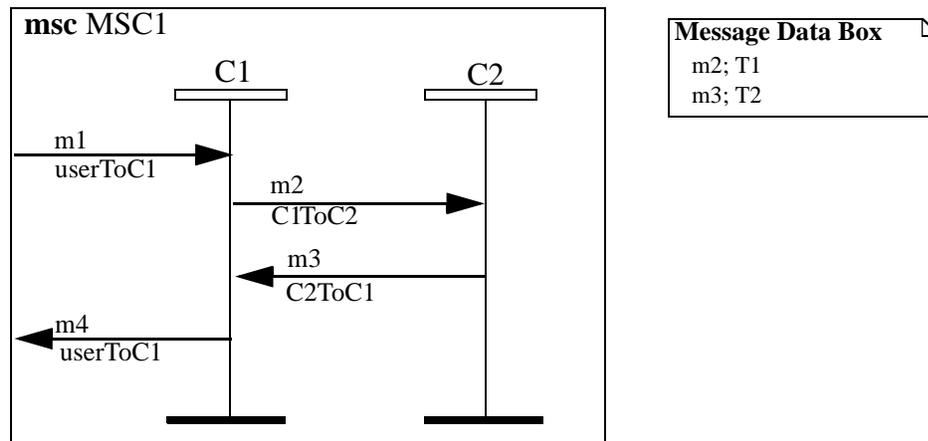
An example of contract specification is given in Figure 25. In this figure, message m1 is sent through contract UserToC1, message m2 is sent through contract C1ToC2, message m3 is sent through contract C2ToC1 and message m4 is sent through contract C1TOUser.

Message Data Box

Also, in real-time systems, message exchanges often imply data transfers. Data transfer constitutes an essential aspect of real-time system modeling. For this reason, in the RT-TROOP modeling process, we allow the specification of data types associated with mes-

sage exchanged at the MSC level. For this purpose, we introduce *message data boxes* that can be placed beside or below the MSC frame in basic MSC⁵. An example of a message data box is given in Figure 25. In this example, data of type T1 is associated with message m2, and data of type T2 is associated with message m3.

FIGURE 25. MSC with contract identifier and message data box



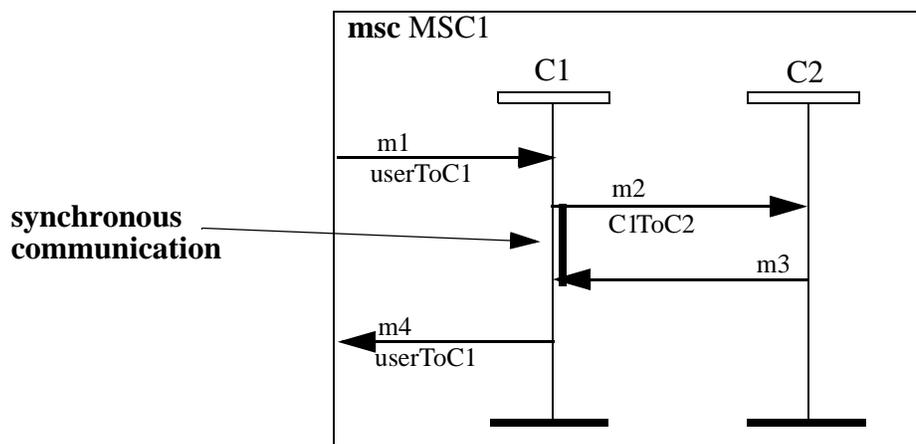
Synchronous Message Arrow

In standard MSC, there is only one type of communication; that is asynchronous communication. However, the modeling of real-time systems often requires both asynchronous and synchronous communication. The choice of a particular type of communication, which constitutes a design issue, is guided by a variety of considerations that are outside the scope of this thesis. In the RT-TROOP modeling process, we believe that it is important to support both types of communication. Therefore, we introduce a new symbol to the standard basic MSC notation to specify *synchronous communication*. This new symbol is illustrated in Figure 26.

5. Another option would have been to add message data types beside the message name on the message arrows. We rejected this option because it tends to clutter MSC diagrams.

In this figure, message *m2* and message *m3* are linked together by a perpendicular line to form a synchronous communication. The implied semantics is that between the sending of message *m2* and the reception of message *m3*, the execution of *C1* is suspended, i.e. *C1* would allow the reception of any other messages. Syntactically, because in synchronous communication both messages are sent through the same contract, there is no need to specify the contract on the reply message.

FIGURE 26. Synchronous communication symbol



2.3.4 MSC Model

An *MSC model* is composed of a set of *MSCs*. An *MSC* is either a *HMSC* or a basic *MSC*. *HMSCs* describe possible sequences of *MSC* execution, while basic *MSCs* describe the actual sequences of messages exchanged between components.

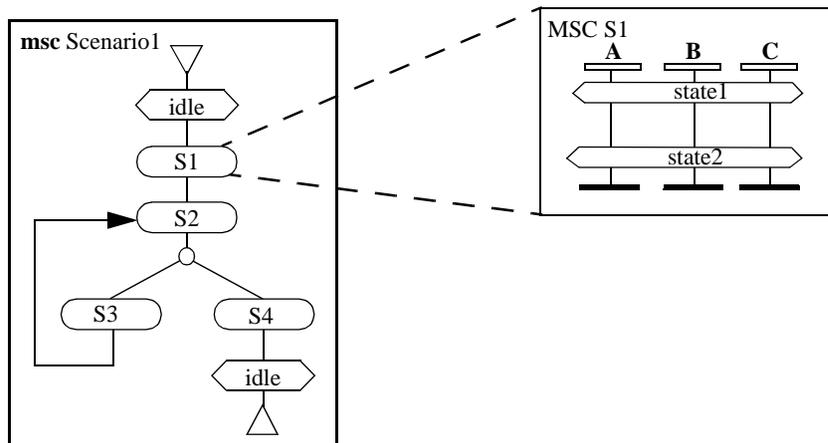
It should be noted that any given *MSC* can be referenced to in more than one *MSC*. It should also be noted that *MSC* references can be used in both *HMSCs* and basic *MSCs*. Thus a basic *MSC* can refer to other *MSCs*. This provides a mechanism for abstraction and reuse.

Also, many different versions of an MSC model are usually defined in a development process. For example, a different version may be defined for each iteration in an iterative process. When needed, we refer to a specific version, say version n , of a MSC model using MSC_model_n .

2.3.5 MSC Skeleton

Another important concept used in this thesis is the one of MSC skeleton. An *MSC skeleton* is simply an MSC that contains no message arrows and no instance actions. The composition of HMSCs is the same in MSC skeletons as it is in regular MSCs. However, the composition of basic MSCs is different. In MSC skeletons, basic MSCs are only composed of instance timelines and optional initial and terminal conditions. We call basic MSC skeletons the basic MSCs contained in MSC skeletons. An example of an MSC skeleton is given in Figure 27.

FIGURE 27. An example of MSC skeleton



In the context of this thesis, and more particularly in the transition between UCM and MSC (Section 3.5), MSC skeleton are augmented with UCM responsibility identifier placed on the component timeline (an example is given in Figure 46).

2.4 ROOM

The ROOM methodology was developed by Selic et al. [93] to be used at a detailed level in real-time system design. ROOM, which is implemented in the ObjecTime toolset, provides an expressive modeling language supported by a powerful development methodology. The popularity of ROOM and ObjecTime has rapidly increased in the software industry worldwide with a consequence that the ROOM notation is now part of the UML-RT notation [62].

In ROOM, systems are modeled by means of communicating hierarchical state machines. The ROOM modeling language is composed of two distinct parts: a structure part, simply called ROOM Structure, and a behavior part, called ROOMCharts (inspired by statecharts [36]).

In this section, after discussing the main motivations for using ROOM in our methodology, we briefly describe the basic ROOM Structure and ROOMCharts concepts and notations that are used in this thesis. Other more complex concepts and notations will only be described as required when used in examples, or case studies. A complete description of the ROOM modeling language and methodology can be found in [87]. The brief descriptions given in this section does not express all the richness and the power of the ROOM modeling language.

2.4.1 Motivation for Using ROOM

The main motivation for choosing ROOM in this project, instead of the ITU standard formal description technique SDL, is that we believe that it provides a more powerful detail-

level modeling technique than SDL. SDL is, in our opinion, not well adapted to the modeling of complex real-time systems. For example, SDL does not explicitly support structure dynamics. Also, SDL uses a flat state machine model that becomes rapidly difficult to use as the size and the complexity of the system increases⁶. The main strength of ROOM comes from the fact that it implements powerful object-oriented and real-time system modeling capabilities, while remaining simple and intuitive to use.

Also, unlike UML, ROOM and SDL-based methodologies [39], [46] provide an executable semantics. This means that the models produced using ROOM can be executed at the modeling stage. Thus, we do not have to wait until implementation is completed to conduct system testing, which entails that errors can be discovered earlier in the development process. In our opinion, executability is an essential property of a modeling technique.

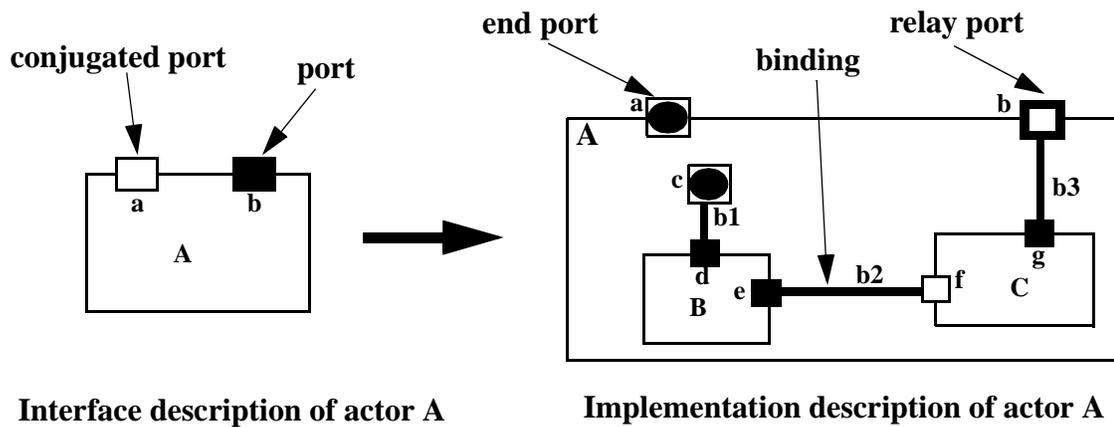
Also, ROOM is supported by a powerful tool named ObjecTime. An evaluation of the ObjecTime tool is given in [9] and [34].

2.4.2 ROOM Structure Notation

A ROOM structure is mainly composed of a set of actors, a set of ports, and a set of contracts. In this section, we briefly describe the basic concepts and notations of ROOM Structure modeling. To illustrate the concepts and notations, the diagrams of Figure 28 will be used.

6. A comparison of ROOM and SDL is provided in [87].

FIGURE 28. ROOM structure



Actors

Actors constitute the primary structural elements of ROOM modeling. In ROOM modeling, all significant system components are modeled by means of actors. They constitute independent, concurrently active logical machines. Actors provide a formally defined interface, which is specified in terms of a set of interface components, called ports. This strategy ensures a complete encapsulation of the implementation of the actors. Visually, actors are represented by labelled rectangles.

There exist two different descriptions of an actor: the *interface description* and the *implementation description*. The *interface description* gives the external view of the actor. It is defined as a set of ports. For example in Figure 28, actor A is externally defined by the set of interface components {a, b}.

The *implementation description* gives the internal view of the actor. It is composed of an implicit behavior, which is separately described by means of a ROOMCharts model (see Section 2.4.3), and a structure. The structure is mainly defined in terms of a set of behavior interface components, called end-ports (see below), a set of internal actors, and a set of contracts (see below). In Figure 28, the internal structure of actor A is defined a set of two end ports, {a, c}, a set two internal actors, {B, C}, and a set of three contracts, {b1, b2, b3}.

Technically, each actor is an instance of an actor class, whose sole purpose is to localize the definition of these instances.

Ports

ROOM interface components are called *ports*. Ports constitute the means by which an actor implementation, contained within an encapsulation shell, can communicate with its environment. The implementation of an actor only interacts directly with ports; it is decoupled from the environment so that the same actor can be used in a variety of contexts (or systems).

Technically, a port is defined as an instance of a protocol class. A ROOM *protocol class* is defined as a set of incoming message types and a set of outgoing message types. An optional specification of valid message exchange sequences (in the form of MSCs), and an optional specification of the expected quality of service (e.g. service time) can also be specified. However, in the context of this thesis, we limit the definition of protocol classes to incoming and outgoing messages.

We say that a port p_1 is a *conjugated version* of a protocol class C_1 if the messages defined in port p_1 are the “inverse” of the ones defined in C_1 , i.e. the incoming messages of p_1 correspond to the outgoing messages of C_1 , and the outgoing messages of p_1 corresponds to the incoming messages of C_1 . Conjugated ports are graphically represented by white filled squares, while unconjugated ports are black filled.

We distinguish two different types of port: *end ports* and *relay ports*. *End port* is the mechanism by which structure and behavior are linked together in ROOM models. End ports are implicitly connected to the behavior part (ROOMChart model) of an actor. *Relay ports* is the mechanism by which internal actors can communicate with the environment of their containing actor.

Since actors are completely encapsulated structural entities, it is not possible to distinguish, from an external (user) point of view, the types of ports that are defined at the interface of an actor. Externally, all ports look similar; they are graphically represented as black or white squares. Thus, the users of an actor can only know its set of ports and the communication protocols associated with them. They do not have to know if a port is an end port or a relay port.

However, from an internal point of view, ports have a different graphical representation. End ports are illustrated using a square with an internal circle, while relay ports are illustrated with a square inside a larger square. Black and white are used to indicate if a port is conjugated or not.

Bindings and Contracts

Two other important structural concepts in ROOM are the ones of binding and contract. A *binding* is the symbol that is used to connect two compatible ports. It is graphically represented by an undirected line that connects the two ports. We say that two ports are *compatible* if and only if the set of outgoing message types of one port is included in the set of incoming message types of the other port. This means that all the messages sent at one port can be received at the other port.

A *contract* is defined as 3-tuple composed of a binding and the pair of actor ports that are linked by the binding. Formally, a contract is defined in terms of:

- A *binding-name*
- Two end-ports: *end-point*₁, and *end-point*₂

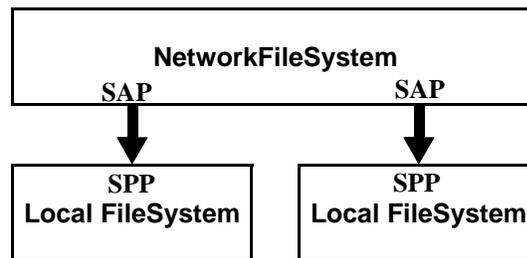
where *end-point*_{*i*} (*i* = 1, 2) is the unique identifier that identifies end-point_{*i*}. This identifier is composed of the name of the port and the name of the actor to which it belongs. For convenience, in this thesis, we use the binding identifiers as contract identifiers. For example in Figure 28, we identify three contracts named **b1**, which is formally defined as <b1,

$c/A, d/B$ >, b_2 , which is formally defined as $\langle b_2, e/B, f/C \rangle$, and b_3 , which is formally defined as $\langle b_3, b/A, g/C \rangle$.

Layer Connections

In contrast to bindings, *layer connections* are directed relationships and model situations in which there is an asymmetric dependency between two actors. This asymmetry is typically in the form of a client-server relationship: the client cannot function unless a server is present, whereas the server can exist and function independently of any particular client. As illustrated in [29], the graphical representation of a layer connection is a directed arc between two actors.

FIGURE 29. ROOM layering



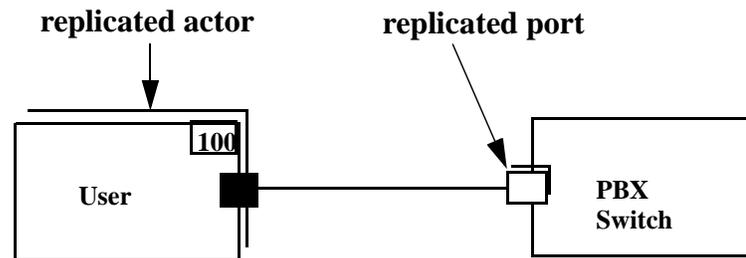
A layer connection connects one or more *service access points* (SAP) on the client to a *service provision point* (SPP) on the server. SAPs and SPPs are also instances of protocol classes like ports. Hence, all the rules of port connections apply. However, since the number of such connections tends to be very large, to reduce visual clutter, each layer connection typically represents multiple individual point-to-point connections. For the same reason, SAPs and SPPs are not rendered explicitly. They are defined at the actor behavior level (i.e. in ROOMChart models).

The layer connections and bindings that interconnect actors defines the complete set of possible communication relationships between actors.

Replication

In real-time system modeling, we often need to model a set of instances of the same type that need to be handled in some uniform fashion. For example, a PBX telephone system can be modeled as a set of users all connected to the same PBX switch. In ROOM, both actor instances and ports can be replicated. ROOM's graphical notation for replication is illustrated in Figure 30.

FIGURE 30. Structural replication



Structure Dynamics

The ROOM language also provides for the description of important structure dynamics concepts such as dynamic creation and destruction of actors (optional actors), multiple containment and actor importation (imported actors). However, since the structure dynamics aspect of real-time system modeling is not covered in this thesis, the description of these concepts has been left out. Readers interested in ROOM structure dynamics are referred to [93].

2.4.3 ROOMChart Notation

In ROOM, actor behavior is defined by means of ROOMCharts which are composed of a Schematic Level and a Detail Level. At the schematic level, ROOMCharts use a hierarchical state machine model similar to StateCharts [36]. At the Detail Level, ROOMCharts use

languages like C⁺⁺ or RPL (a specific ROOMCharts prototyping language based on Smalltalk) to describe the functions that are encapsulated in transitions in a state transition diagrams. In this section, we describe the ROOM notation and concepts that are used in this thesis for behavior modeling.

State Machine

A ROOMChart state machine description consists of the following attributes:

- *Behavior interface*, defined by the following triple:
$$\text{behavior-interface} = \langle \text{end-ports}, \text{saps}, \text{spps} \rangle$$
- *Functions*, a set of functions or procedures that can be invoked in transition code
- *Variables*, a set of extended variables that can be used within the state machine at a detail level
- *States*, the set of states that compose the state machine
- *Transitions*, the set of transitions

ROOMCharts uses a run-to-completion model of event processing. This means that event are processed one at the time. Thus, within an actor behavior, once the handling of a message is started, no other messages can be processed until the handling of that message is completed⁷.

In Figure 31 and Figure 32, the different elements of the ROOMCharts notation are identified. In Figure 31, a flat state machine is given to illustrate the basic ROOMCharts notations, while in Figure 32 a hierarchical state machine (obtained by decomposing state S1 of Figure 31) is given to illustrate the more complex ROOMCharts notations. These elements are described in the rest of this section.

7. The run-to-completion model of processing used in ROOM implies queueing of messages at the interfaces.

FIGURE 31. Basic ROOMCharts notation

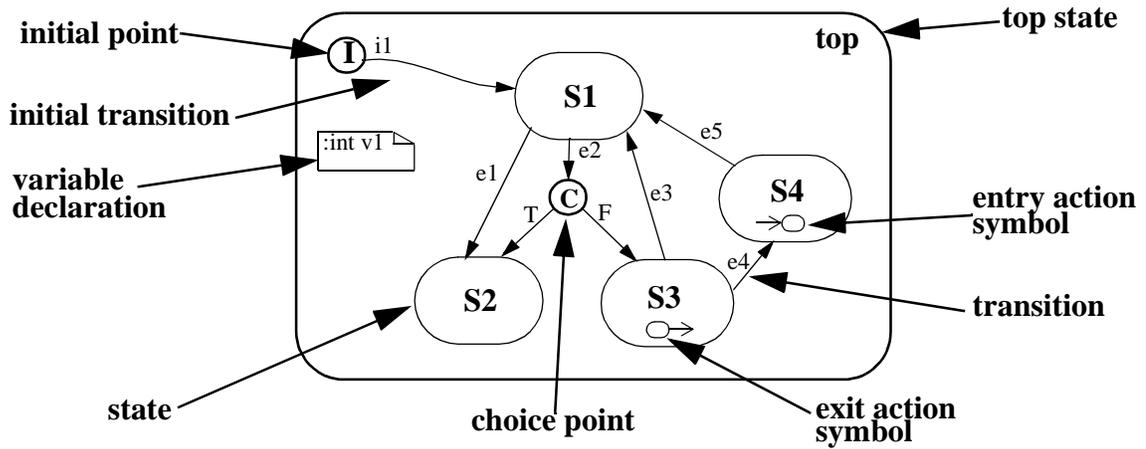
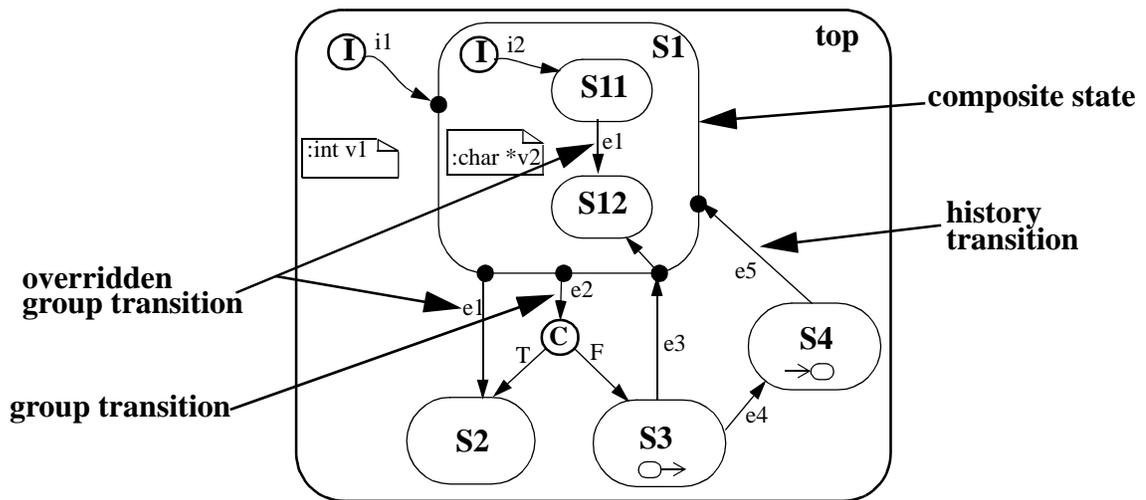


FIGURE 32. More complex ROOMCharts notation



State

In ROOMCharts, states are graphically represented by labelled rounded corner rectangles. When an actor is in a state, it is ready to process external events. ROOMCharts does not provide any explicit “receive” primitive. It is automatically assumed whenever the behavior is in a state.

One of the particularities of ROOMCharts is that it uses a hierarchical state machine model, which allows states to be decomposed into lower level state machines. This constitutes a powerful mechanism that reduces the complexity of large state machines. In this context, a state may be composed of the following optional attributes:

- *Variables*, the set of extended variables that can be used within the state at the detail level
- *Substates*, the set of states that compose the internal state machine
- *Transitions*, the set of transitions between substates
- *Entry action*, an optional code segment that is automatically executed whenever the state is entered regardless of which transition is taken into the state
- *Exit action*, an optional code segment that is automatically executed whenever a transition is taken out of the state

A state that is decomposed into a lower level state machine is called a *composite state*. In ROOMCharts, the word **top** is a reserved identifier that is used to label the uppermost state of an actor behavior.

Also, in ROOMCharts, a composite state has a notion of history in that, at execution time, it always remember which of its substates was last visited. When a history transition is taken into the state, a composite state returns to its last visited substate. As a notation convention, a transition that terminates on the outside border of a composite state automatically goes to history. For example, in Figure 31, after the execution of the transition associated with the event **e5** (from state **S4**) the behavior may either enter state **S11** or **S12** depending on which is the last visited substates of state **S1**.

Transitions

A transition is the means by which actor behavior goes from one state to another. Graphically, a transition is represented by a directed arrow that goes from the source state to the

destination state. A transition is formally composed of a *trigger specification* and a *transition action code* description. The *trigger specification* of a transition is defined by the following triple:

$$\text{trigger-specification} = \langle \text{signal, interface-component, condition} \rangle,$$

where *signal* is the signal that triggers the event, *interface-component* specifies the end-port or saps by which the signal is received, and *condition* represent the guard condition of the transition. This condition must evaluate to true to allow the transition to be taken. An event triggers a transition only if it satisfies the trigger specification of one of the outgoing transitions of the current state.

Transition action code is defined in terms of a sequence of primitive actions described at the programming language (RPL or C++) level. Primitive actions include sending messages, computing values, assigning new values to variable, etc.

A transition may be composed of several transition segments, each of which is defined in a different composite state context. This occurs when a transition cuts across composite states boundaries. This type of transition is called *segmented transition*. Transition e3 in Figure 32 is an example of such transition that is composed of two transition segments: one segment between states S3 and S1, and another segment between the border of state S1 to state S12. In such a transition, the occurrence of the transition triggering event results in the successive execution of the different segments that form the overall transition. If there is an entry action associated with a composite state visited as part of a segmented transition, its entry code will be executed between the execution of the external and internal segments attached to this state. In the case of a segmented transition, the fact that a transition is executed in a run-to-completion manner means that no new event can be processed until the execution of all the transition segments is completed.

Transitions can also be split into multiple segments at a *choicepoint*. An example of a transition point is given in Figure 31. In such case, only one of the outgoing transition segment linked to the choicepoint will be taken after the guard of the choicepoint is evaluated.

Group transitions are transitions that origin at the border of a composite state. These transitions apply equally to all substates unless explicitly overridden. ROOM scoping rules are such that triggers on the innermost current state take precedence over equivalent triggers in higher scope. For example, in Figure 31, if the behavior is in state S11, then occurrence of event e1 will bring the behavior in state S12 because the transition between S11 and S12 will take precedence over the group transition that goes from S1 to S2. This is called an *overridden group transition*.

The use of choicepoints and group transitions can significantly simplify the graphical representation of complex behavior, and consequently, can lead to more understandable models.

Initial Point and Initial Transition

In ROOMCharts, the *initial point* is the (non-state) point in which a state is placed when it is created. If there exists an *initial transition* between the initial point and an initial state, this transition is triggered spontaneously when the component is created. Thus, the transition code will be executed, and the behavior component will be placed in the initial state. However, if there exists no transition from the initial point, the containing state will remain in a non-state until a transition brings it in a defined substate.

Variables

Variables can be defined in ROOMCharts as instances of detail level data classes.

2.4.4 ROOM Model

A *ROOM model* is composed of a set of actor classes, a set of protocol classes, and a set of data classes.

Also, many different versions of a ROOM model are usually defined in a development process. For example, a different version may be defined for each iteration in an iterative process. When needed, we refer to a specific version, say version n , of a ROOM model using ROOM_model _{n} .

2.5 Summary

In this chapter, the main notations and concepts that are used in the RT-TROOP modeling process have been described. These notations and concepts address the modeling of the important aspects of real-time systems at different levels of abstraction. The grouping of these concepts and notations in a single modeling process, called RT-TROOP, is described in the next chapter.

CHAPTER 3 RT-TROOP Modeling Process

In Chapter 2, we described four models (STD, UCM, MSC, and ROOM) that are used at different stages of real-time system modeling. In this chapter, we define a concrete modeling process, called RT-TROOP (Real-Time TRaceable Object Oriented Process), that combines the use of these four models, and that allows moving from a set of requirements to an implementation in a systematic and traceable manner. By concrete modeling process, we mean a process that is defined at a sufficient level of details to be directly implementable.

As previously mentioned, structure dynamics have been left out of the thesis, for sake of conciseness. However, since UCM, MSC, and ROOM all provide notation for modeling structure dynamics, this issue can be added to RT-TROOP modeling process in a straightforward manner. Indeed, a report describing how structure dynamics can be modeled at different level of abstraction using UCM, MSC, and ROOM in a traceable manner has already been written [72]. And the current industrial applications of the RT-TROOP process has proven that modeling structure dynamics does work.

In this chapter, we:

- Give our definition of a modeling process (section 3.1).
- Give an overview of the RT-TROOP modeling process (section 3.2).
- Describe each of the modeling phases that compose the RT-TROOP modeling process (section 3.3 to section 3.12). Transition techniques and design phases are illustrated with abstract examples. They will be illustrated with a concrete case study in Chapter 5.

- Discuss how the RT-TROOP modeling process adapts to specific design contexts (section 3.13).
-

3.1 Modeling Process

Modeling Process versus Development Process

In the context of this thesis, the term *modeling process* means a set of rules that define how a system implementation should be built from a set of system requirements. Such a process includes a description of the models that are produced and in what order. The proposed process also gives a description of the different modeling phases, and a set of rules and guidelines to help system designers meet the overall system requirements.

We divide modeling phases in two distinct categories: *model transition phases*, which define the steps that should be carried out when making the transition between models, and *in-model modeling phases*, which define the modeling activities that may take place in the different models.

In comparison to a modeling process, a *development process* has a larger scope as it should include, in addition to modeling, other development aspects such as: requirement capturing and classification, risk analysis, testing, deployment, maintenance, and so on.

Even though some issues related to the integration of RT-TROOP modeling in an industrial development process are discussed in Chapter 7, the integration of the RT-TROOP modeling process into a global development process is outside the scope of this thesis. The RT-TROOP modeling process is defined independently from any particular development process. We envision that RT-TROOP modeling could be integrated in existing develop-

ment processes such as the Rational Unified Process (RUP) [55] or the Unified Software Development Process [48].

Model Transition Phases

Because object-oriented modeling processes combine the use of several models, the transitions between models constitute key steps. When conducted in an ad hoc manner, these transitions are error prone. They often introduce inconsistencies between models that make these models obsolete and unusable. Model transitions are particularly problematic for inexperienced designers, who usually do not understand very well the role of the different models in the overall process and the relationships that exists between them.

One way to facilitate model transitions consists in defining a set of model transition techniques. The role of those model transition techniques is to define how the information contained in one model can be used for the definition of another model, and what is the information that needs to be input by the designer. The goal of a model transition is to produce a model that is consistent with previous models.

In this chapter, we define a set of model transition techniques that allow moving between STD, UCM, MSC'96, and ROOM in a systematic and traceable manner.

The model transitions techniques defined in this thesis establish semantic links between elements of the different models. These links are defined based on our understanding of the different notations and on the role played by the different models in the overall modeling process, not on a formal semantics of the models. There are two main reasons for this: 1- our main motivation for the definition of the RT-TROOP modeling process is to provide a concrete solution for a problem often raised by industrial software engineers that concerns the difficulty to build communicating state machine models from scenario models, not establish formal transformations between models, and 2- among the models used in RT-TROOP, only MSC has a formal semantics [44], both UCM and ROOM have a formal syntax, defined respectively in [2] and [93], but no formal semantics. Moreover, the defi-

nition of formal semantics relations between RT-TROOP models would require the definition of (or the use of an existing) meta-model that could be used to specify the semantics of each of the models. The definition of formal semantic relations between the models is outside the scope of this thesis. It constitutes by itself the subject of another research project. Therefore, one may see the inter-model links defined in this thesis as arbitrary ones.

While some aspects of the transition techniques defined in the RT-TROOP modeling process can be automated, they all require inputs (design decisions) from the designer. Our intent is not to completely automate the transitions between models, but rather to facilitate a designer's task by listing the set of decisions that must be taken in performing the transitions, and by identifying the information that could be used to take the decisions. We strongly believe that the definition of these transition techniques can significantly reduce the complexity of a designer task, in particular the task of inexperienced designers.

The definition of the model transition techniques constitutes an important contribution of this thesis.

In-Model Modeling Phases

In-model modeling phases define the modeling activities that may take place in a specific model. This includes design transformations such as: definition of new elements, refinement of existing elements, composition of existing elements into a single new one, and deletion of existing elements.

The major objective of this thesis is to define a modeling process that integrates existing modeling techniques. We make no attempt to contribute to the development of any of the modeling techniques per se. Each of the modeling techniques we use has been the subject of numerous publications that discuss and illustrate with case studies how they can be used in the development of real-time systems. These papers describe different design con-

cepts and design transformations that can be used in the models. In this thesis, we focus instead on how to use the modeling techniques together.

In the in-model modeling phase sections of this chapter, we list and briefly describe design transformations that can be used in the different models in the context of the RT-TROOP modeling process. The set of in-model design transformations that we describe does not pretend to be complete in any way. Therefore, system designers using RT-TROOP should not limit themselves to the set design transformations defined here, and should use others as needed. In this chapter, we describe the ones that we believe are the most important. The in-model modeling phases and the design transformations that can be applied in these phases are discussed for the sake of process description completeness. We want to give an overview of the different phases that compose the RT-TROOP modeling process in order to highlight where design decisions can be taken along the modeling process: in the model transition phases or in the in-model modeling phases.

Iterative and Incremental Process

The objective of iterative development consists in decomposing the overall set of requirements into subsets that can be each addressed in a different development cycle, called iteration. Each of the system models must be revisited and possibly modified at each iteration to reflect the new set of requirements. At the model level, the addition of new requirements may result in the introduction of new model elements or in the modification or deletion of existing ones.

An important advantage of the iterative approach is that it should produce a completely integrated and tested system (that has a reduced set of functionality) at the end of each iteration. Because a system implementation is typically built in each iteration, and because the duration of iteration is relatively short, the iterative process allows quick feedback on system design. Thus, the late integration disasters of the waterfall model, and other similar processes, can be avoided.

In an iterative process, the implemented system functionality (i.e. the coverage of the overall set of requirements) gradually increases over iterations. The philosophy behind the approach is to start from something small that works and add to it. The process ends when all system requirements have been addressed, and the complete system functionality has been implemented.

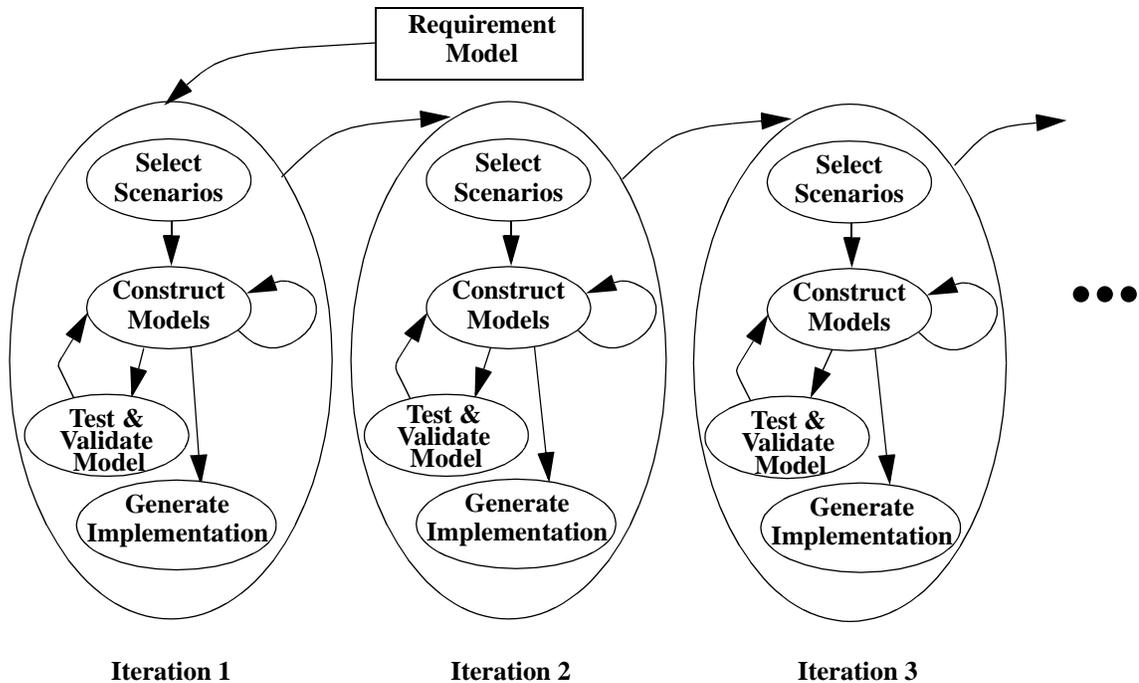
The starting point of an iteration is the definition of a new version of the requirement model, which we take to include a set of scenarios and a set of general requirements. The requirement model is built in an incremental manner, which means that new scenarios and general requirements are added at each iteration. Incremental (upward compatible) modification of scenarios and general requirements is also allowed. Examples of incremental scenario modifications include addition of new responsibilities in existing scenarios, refinement of existing responsibilities, and addition of new alternative scenarios to a scenario. In this chapter, we only deal with incremental modification of requirements.

Because of the potential impact they have on system models, non-incremental modifications of requirements need to be considered separately. Examples of non-incremental modifications of scenarios include the removal of an existing scenario, the modification of the ordering of responsibilities in a scenario, and the deletion of a responsibility in a scenario. In order to deal with non-incremental modifications, a modeling (and a development) process must allow for the precise evaluation of the impact of modifications, and for the maintenance of model consistency through modifications. For example, if an error is discovered in a scenario at the implementation level, the scenario need to be modified, and all models must be revisited and modified in a consistent manner to reflect the modification. In this thesis, we use traceability relations, defined in Chapter 6, to deal with the issue of non-incremental modifications.

The result of each iteration is a new version of the different models of the system, and a new version of the system implementation. Each model must be tested and validated to ensure correctness.

A diagram illustrating our proposed iterative approach is given in Figure 33¹.

FIGURE 33. Iterative process



3.2 Overview of RT-TROOP Modeling Process

In this section, we give an overview of the RT-TROOP modeling process; we discuss its main characteristics, describe the role of the models that compose it, and give a table that summarizes the role of the different RT-TROOP modeling phases.

1. This diagram has been taken from [93].

3.2.1 Characteristics

RT-TROOP is an iterative object-oriented modeling process for real-time systems that has two main characteristics: scenario-driven and strong traceability.

Scenario-driven

Each RT-TROOP iteration is defined in terms of a new set of scenarios. During an iteration, the focus is on the design of the newly introduced scenarios, and on the scenario interactions (possible conflicts) that results from their introduction in the system. Scenarios are first captured at an abstract in a textual form, and progressively refined until they reach a sufficient level of details to be integrated in the detailed-level component behavior model.

Strong traceability

In RT-TROOP, strong traceability is maintained between model elements throughout the entire process. To achieve this strong traceability, we define three types of traceability relations: *inter-model traceability*, which allows linking elements of different models, *inter-version traceability*, which allows linking elements of different versions of a model, and *design decision traceability*, which allows linking each design decisions to a specific set of requirements and model elements to the design decisions from which they resulted. This allows evaluating the impact of modifications on the different models, and therefore allows maintaining consistency between the different models. The different types of traceability relations are discusses in more details in Chapter 6.

Although traceability is not addressed explicitly in this chapter, the modeling phases that are defined and the notation that is used allow maintaining a fine-grained traceability between requirements and implementation.

3.2.2 Composition of RT-TROOP

RT-TROOP is defined in terms of a set of models and a set of modeling phases. The models currently used in RT-TROOP are STD, UCM, MSC, and ROOM. The modeling phases include both model transition phases and in-model modeling phases. The role of the different models and modeling phases are described in the rest of this section.

An overview of the RT-TROOP process is given in Figure 34². It illustrates the different models and modeling phases that compose it. In this diagram, models are illustrated using boxes and the modeling phases are illustrated using numbered circles. The arrows drawn between model boxes illustrate inter-model relationships. These arrows are drawn as unidirectional arrows in this diagram. The direction of the arrows correspond to forward engineering direction. However, since the RT-TROOP modeling process maintains backward traceability relations between models³, the arrows can also be taken in the opposite direction.

The focus of this diagram is on models and modeling phases that take place in a single iteration. In order to get the complete picture of the modeling process, readers should remember that each model is incrementally built through iterations. Thus each iteration, except the very first one, starts with the set of models produced in the previous iteration. During an iteration, modifications are made to the different models to reflect the new set of requirements. The relationship between successive versions of a model (i.e. model versions built in successive iterations) is not illustrated in this diagram.

Each RT-TROOP iteration is triggered by the definition of a new version of the STD model. The iteration may also contain a set of new general requirements⁴. The new STD model may contain new scenarios and/or modified ones. The issue of decomposing the

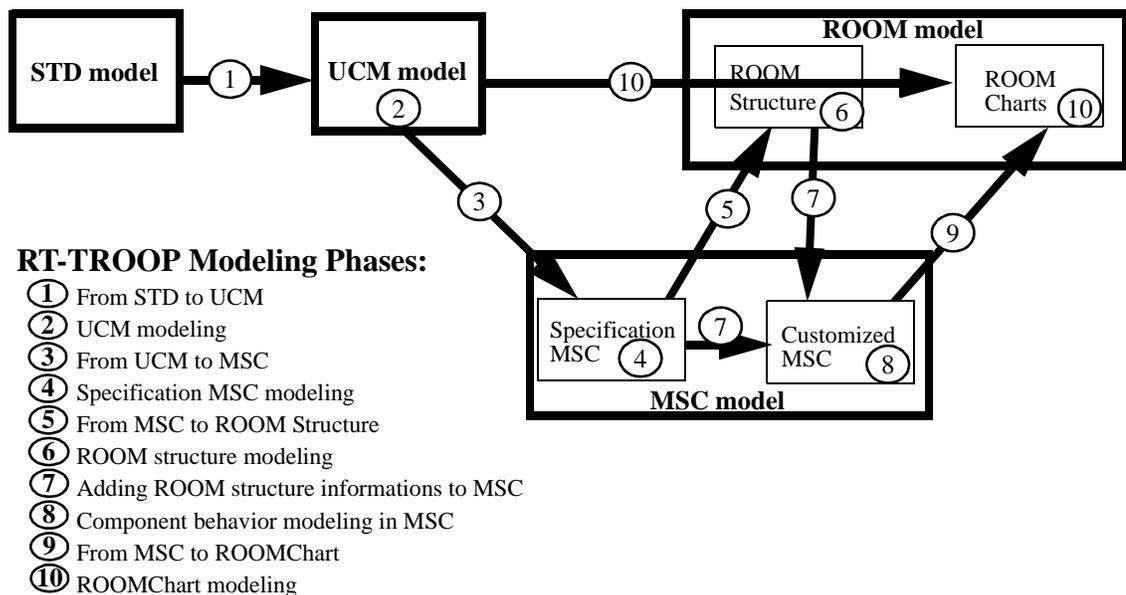
2. The ROOMChart modeling phase (10) illustrated in Figure 34 is located in the ROOMChart model, but because it requires input from the UCM model, an arrow is drawn between the UCM model and the ROOMChart model.

3. Backward traceability relations between RT-TROOP models are defined in Chapter 6.

overall set of STDs into subsets, each of which being addresses in different iteration, is a development process issue that is outside the scope of this thesis. Each iteration ends with a new version of the different models.

The numbers used to identify the modeling phases in Figure 34 should not be interpreted as a prescribed sequence. They correspond to one possible sequence. This sequence is the general sequence executed when building a system from scratch. Other sequences are also possible. Some of them will be discussed in section 3.13.

FIGURE 34. RT-TROOP modeling process



In terms of models, RT-TROOP could be extended with other models such as UML structure diagram to model class hierarchies, formal verification methods, and performance analysis models. RT-TROOP is intended to be an open process that allows for the integration of different models. In order to integrate a new model, say M, in the RT-TROOP modeling process, we need define to define:

4. The set of general system requirements is not shown in the RT-TROOP modeling process diagram, but it is implicitly part of it.

1. The role of M in the modeling process
2. A set of transition techniques between M and the other models (STD, UCM, MSC, and ROOM)
3. A set of traceability relationships between M and the other models

3.2.3 Role of the Different Models

Each RT-TROOP modeling iteration starts with a new version of the STD model and it produces the following models: a UCM model, a MSC model, and a ROOM model. The production of the ROOM model of the system constitutes the end goal of the modeling process⁵.

STD model

The STD model is the starting point of the RT-TROOP modeling process. It contains the set of scenarios that must be implemented in the system during an iteration. The textual and high-level nature of STDs allows capturing scenarios at the requirement level. This facilitates communication with stakeholders.

The process of defining the different scenarios contained in the STD model is a task that is part of requirement engineering, and therefore is outside the scope of this thesis.

UCM model

The UCM model is the high-level design model of the system. It describes the different scenarios of the system, the interactions among them, and the set of components that are responsible for their execution.

5. Implementation can then be automatically generated from the ROOM model.

Because of the high-level behavior view it provides, and because of its expressive and intuitive graphical nature, the UCM model is particularly useful to analyze and discuss system issues, and to communicate with stakeholders.

The UCM model produced during the UCM modeling phase can be used to validate requirements, or more specifically STDs, and high-level system design. This can help to discover system issues that occur because of the combination of scenarios in composite maps. Examples of issues that can be discovered by analyzing UCM models include: race conditions, scenario conflicts, direct and indirect coupling between scenarios, and scenario concurrency. It can also be used to identify possible bottlenecks in the system.

MSC model

The MSC model focuses on inter-component communication. It describes the set of scenarios (defined in the UCM model) in terms of message sequences.

The MSC model plays a central role in the RT-TROOP modeling process as it provides the bridge between UCM high-level model and ROOM detailed model. The level of details in the MSC model gradually increases as we progress in an iteration. To capture the increasing level of details of the MSC model, RT-TROOP defines two different MSC models: a *specification MSC*, which is defined independently of any ROOM model, and a *detailed MSC*, which is defined in relationship with a specific ROOM model.

The specification MSC model is defined as a refinement of the UCM scenarios in terms of components, inter-component messages, and actions. At this level, the MSC model is considered abstract in as much as it is not linked to any particular ROOM model; the messages contain no information about the ports by which they are sent, the only states defined in the model are system states that have no direct equivalence in the ROOM model, and the components (instances) are not yet bound to any particular ROOM actors.

The information contained in the specification MSC model is used as the basis for the definition of the ROOM structure model, which is defined in terms of a set of actors (components), ports and bindings. In fact, the specification MSC model contains all the necessary information for the definition of the ROOM structure of the system. It defines system components, it identifies the pairs of communicating components, and it explicitly describes the list of messages exchanged between components.

Once the ROOM structure of the system is defined, the MSC model is updated to reflect the specific structure of the system. At this point, a new MSC model, called the *customized MSC model*, is defined. This model is obtained by adding ROOM structure information to the specification MSC model.

Finally, the customized MSC model is used to smooth the transition between system behavior modeling and component behavior modeling. In order to facilitate this transition, component behavior information (i.e. component states, transition code, entry and exit code) is introduced in the customized MSC model. The resulting customized MSC model is then used as a main input in the definition of the ROOMChart component behavior models.

The customized MSC model produced during an iteration can be used to test the ROOM model at the end of the iteration.

ROOM model

The ROOM model is composed of two distinct parts: the ROOM structure model and the ROOMChart model. The production of the complete ROOM model of the system is the end goal of the RT-TROOP modeling process. The ROOM model is the blueprint from which code is automatically generated.

The ROOM structure model describes the structure of the system in terms of a topology of communicating components (actors). It is defined mainly in terms of a set of actors that communicate through ports and bindings.

The ROOMChart model describes the detail-level behavior of each of the actors contained in the ROOM structure model by means of a hierarchical state machine. The ROOMChart model contains a set of component behavior hierarchical state machines (one per system actor).

3.2.4 Modeling Phases

As illustrated in Figure 34, RT-TROOP modeling proceeds through a sequence of different modeling phases. The activities performed in each of these phases, as well as their inputs and outputs, are summarized in Table 1. Each of these modeling phases is described in more detail in the following sections of this chapter.

TABLE 1. RT-TROOP modeling phases

	Input	Objective and Activities	Output
1- From STD to UCM	- a new version of the STD model	<p>Objective: Express the scenarios described in the STD model using the UCM notation</p> <p>Activities:</p> <ul style="list-style-type: none"> - create a UCM related path set for each STD contained in the STD model - map each STD scenario onto a specific UCM path 	- a set of new unbound UCM related path sets
2- UCM Modeling	<ul style="list-style-type: none"> - a set of new unbound UCM related path sets - the existing UCM model 	<p>Objective: Integrate the new UCM paths in the existing UCM model</p> <p>Activities:</p> <ul style="list-style-type: none"> - define system components - allocate responsibilities to system components - define interactions between scenarios - restructure UCM maps 	- a new version of the UCM model
3- From UCM to MSC	- a new version of the UCM model	<p>Objective: Express each UCM path in terms of a sequence of inter-component messages and component actions</p> <p>Activities:</p> <ul style="list-style-type: none"> - generate a HMSC for each UCM related path set - define a basic MSC for each UCM path segment - define data objects associated with messages - describe UCM responsibilities in terms of a sequence of MSC messages and actions 	- a set of new specification MSCs

	Input	Objective and Activities	Output
4- Specification MSC modeling	<ul style="list-style-type: none"> - a set of new specification MSCs - the existing specification MSC model 	<p>Objective:</p> <p>Integrate the set of new specification MSCs in the specification MSC model of the system</p> <p>Activities:</p> <ul style="list-style-type: none"> - introduce new components - refine high-level messages in more detailed ones - define new system states - restructure MSCs 	<ul style="list-style-type: none"> - a new version of the specification MSC model
5- From MSC to ROOM structure	<ul style="list-style-type: none"> - a new version of the specification MSC model 	<p>Objective:</p> <p>Define ROOM role structures on a per scenario (or scenario related path set) basis</p> <p>Activities:</p> <ul style="list-style-type: none"> - define role actors - define role protocol classes - define inter-actor bindings - define ROOM data classes for data associated with messages. 	<ul style="list-style-type: none"> - a set of new ROOM role structures
6- ROOM structure modeling	<ul style="list-style-type: none"> - a set of new ROOM role structures - the existing global ROOM structure model of the system 	<p>Objective:</p> <p>Integrate the ROOM role structures in the global ROOM structure of the system</p> <p>Activities:</p> <ul style="list-style-type: none"> - define system actors - define system protocol classes - define actor bindings - define the type of inter-actor communication (sap-spp or ports) - restructure system 	<ul style="list-style-type: none"> - a new version of the global ROOM structure model of the system

	Input	Objective and Activities	Output
7- Adding ROOM structure information to MSC	<ul style="list-style-type: none"> - a new version of the global ROOM structure model - the existing version of the specification MSC model - the existing customized MSC model 	<p>Objective:</p> <p>Add ROOM structure information to the MSC model</p> <p>Activities:</p> <ul style="list-style-type: none"> - add the binding information to the messages in the MSC model - link each MSC component instance to a specific ROOM actor 	<ul style="list-style-type: none"> - a new version of the customized MSC model containing ROOM structure information
8- Component behavior modeling in MSC	<ul style="list-style-type: none"> - a new customized MSC model 	<p>Objective:</p> <p>Introduce component behavior information in the customized MSC model</p> <p>Activities:</p> <ul style="list-style-type: none"> - define component states in the MSC model - identify transition actions - identify state entry and exit actions - identify synchronous messages (i.e. invoke in ROOMCharts) 	<ul style="list-style-type: none"> - a customized MSC model containing component behavior information

	Input	Objective and Activities	Output
9- From MSC to ROOMChart	<ul style="list-style-type: none"> - a new version of the customized MSC model containing both structure and component behavior information 	<p>Objective:</p> <p>Define component behavior (ROOMChart model) on a per component/scenario basis</p> <p>Activities:</p> <ul style="list-style-type: none"> - define flat state machines on a per component/scenario (or scenario related path set) basis - define ROOMChart states and transitions - introduce code-level information in the ROOMChart models - define required ROOM data classes 	<ul style="list-style-type: none"> - a set of ROOMChart role behaviors
10- ROOM-Chart modeling	<ul style="list-style-type: none"> - a set of new ROOMChart role behaviors - the UCM model - the existing ROOMChart model 	<p>Objective:</p> <p>Integrate the role behaviors in the component behaviors (one for each ROOM actor)</p> <p>Activities:</p> <ul style="list-style-type: none"> - use hierarchical state machines to structure component behavior - define entry and exit code for composite states - restructure component behavior 	<ul style="list-style-type: none"> - a new version of the ROOMChart model - a new version of the complete ROOM model (end objective of the overall modeling process)

The main focus of the RT-TROOP modeling process is on model integration, not on object-oriented modeling patterns or heuristics. For this reason, the RT-TROOP modeling phases focus mainly on establishing semantic relationships between elements of the different models. The only design patterns we define are the ones for hierarchical state machine design. We do not, for example, define criteria that could be used to group messages into protocols, to group responsibilities into objects, to define meaningful objects and classes, etc. Readers interested in such heuristics and patterns are referred to the existing literature, e.g. [28], [35], [97].

Relationship between UCM Paths and STDs

STD and UCM both describe scenario paths at the same level of abstraction. In fact, UCM maps are usually associated with a textual description that briefly describes its overall objective and its different elements: responsibilities, pre and postconditions, triggering and resulting events, alternatives, and nonfunctional requirements that apply to the scenario.

Also, STD and UCM both provide for the grouping of related scenarios into scenario related path sets. STD groups together main scenarios with alternatives, while UCM uses the concept of UCM related path set to express alternatives to main scenarios.

Relationship between UCM Structure and STDs

Concerning structure, STDs contain very little information. In fact, STDs are only concerned with the set of external actors that participate in the execution of the scenario. Such external actors may either represent users (humans) of the system or other systems with which the system communicates during the execution of the scenario.

3.3.2 Generation of the UCM Model

In the transition between STD and UCM, our objective is to generate a UCM model that associates a UCM related path set with each STD contained in the STD model. At a detailed level, each of the elements described in a STD is mapped onto a UCM element. This ensures a strong traceability between the two models.

The details of the generation of a UCM model from an STD model are described hereafter.

Path

In the context of RT-TROOP, we establish a one-to-one relationship between UCM related path sets and STDs. More specifically, we generate a UCM map, containing a single related path set, for each STD contained in the STD model. Each path within a UCM related path set is associated with a specific scenario in an STD. An STD scenario can be either the main scenario of a STD or one of its alternatives.

At a detailed level, we establish a one-to-one relationship between the following elements of UCM and STD models:

- UCM triggering event and STD triggering event
- UCM precondition and STD precondition
- UCM responsibility and STD responsibility
- UCM resulting event and STD resulting event
- UCM postcondition and STD postcondition

In an iteration, designers may have to define new UCM paths or modify existing ones to reflect incremental modifications that have been made to existing STDs. For the purpose of modifications, designers may need to:

- Introduce new responsibilities on an existing UCM path.
- Introduce new UCM paths in an existing related path set.

Structure

At the structure level, a UCM component is generated for each external actor defined in the STD.

Because system components are not yet defined at this point of the modeling process, the resulting UCM maps are unbound maps (see section 2.2.2 for definition of unbound maps).

Example

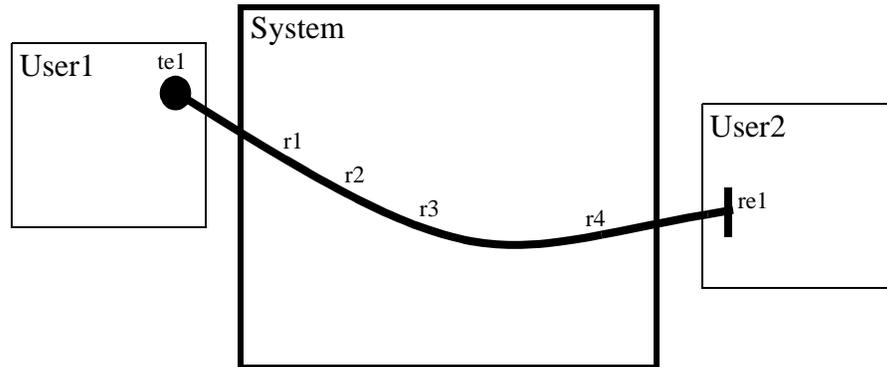
In Figure 36 and Figure 37, an example illustrating the transition between STD and UCM is given.

FIGURE 36. Generation of UCM from STD

STD1: S1	
Description:	
External Actors: User1, User2	
Precondition: The system must be in state S1	
Triggering event: The scenario is triggered by the reception of message te1 from User1	
<ol style="list-style-type: none"> 1. r1 <description of the responsibility> 2. r2 <description of the responsibility> 3. r3 <description of the responsibility> 4. r4 <description of the responsibility> 	
Postcondition: The system ends in state S2	
Resulting event: As a result of the execution of scenario S1, message re1 is sent to User2	
Alternatives: none	
Nonfunctional requirements: <ul style="list-style-type: none"> - the scenario S1 must be executed in no more that 5 seconds - the communication between the users and the system must be implemented using a secure communication protocol 	
Comments:	

FIGURE 37. Unbound UCM map resulting of the transition from STD

ucm S1



te1: triggering event for path P1
 (precondition(te1): System is in the state S1)
 re1: resulting event for path P1
 (postcondition(re1): System is in the state S2)

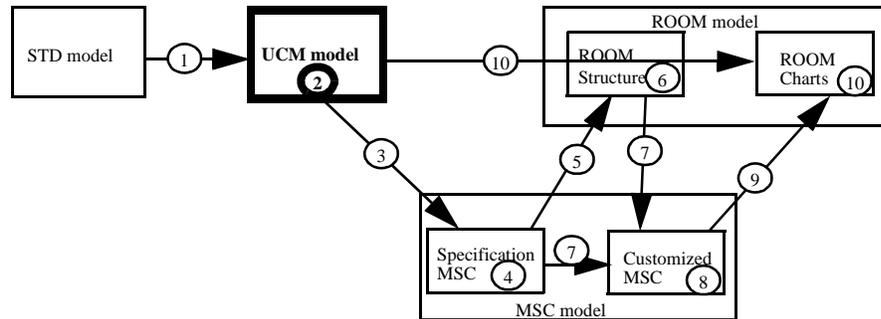
3.4 UCM Modeling Phase

UCM is more than just a graphical representation for scenario textual descriptions. It is a modeling technique that aims at producing a high-level design model of the system. As described in section 2.2, the production of a UCM model requires the definition of system paths, the definition of system components, the allocation of path responsibilities to components, and the description of inter-scenario relationships.

In the transition between STD and UCM, a set of unbound UCM related path sets have been produced. These related path sets are described independently from each other. The objective of the UCM modeling phase is to integrate the related path sets produced in the transition between STD and UCM in the existing UCM model (i.e. the one produced as a result of the previous iteration). The inputs of the UCM modeling phase are the set of new

unbound paths and the existing UCM model. The output is a new version of the UCM model. The UCM modeling phase is highlighted in Figure 38.

FIGURE 38. UCM modeling phase



3.4.1 Modeling Activities

In this modeling phase, different modeling activities may need to be carried out by designers:

- Definition of system components.
- Allocation of responsibilities to components.
- Specification of scenario interactions.
- Addition of new paths to existing composite maps to describe relationships between existing paths and new ones.
- Restructuring of maps.

Map restructuring may be required for several reasons such as: to satisfy new requirements, to increase the level of details in the UCM model, to decompose maps that have become too complex through the iterations, and to increase reusability. UCM maps may be restructured both at the structure and path level.

In the UCM modeling phase, the UCM modeling technique is used as described in [18]. The definition of the elements that composes a UCM model can be done in different orders. The order in which the elements are defined in the UCM model depends on the information contained in the requirements and on the information contained in the existing UCM model. For example, if system components are specified in the requirement document, we generally start by describing the component context diagram⁶, and then express the different scenarios in the context of this diagram. On the other hand, if no components are specified in the requirement document, then we may try to map the scenarios to different sets of components until we find a satisfying solution. Thus, the UCM model constitutes a good model for studying design alternatives.

In RT-TROOP, we build two types of UCM maps: the ones that individually describe related path sets, and the ones that describe relationships between paths contained in different related path sets. The first ones constitute the building blocks for second ones (composite maps).

In this section, we separately discuss modeling activities related to structure and path.

3.4.2 Structure

Component Definition and Responsibility Allocation

As a result of the transition between STD and UCM, a set of unbound maps containing individual related path set has been produced. However, from a system design viewpoint, these maps are incomplete because they do not define the system components that are

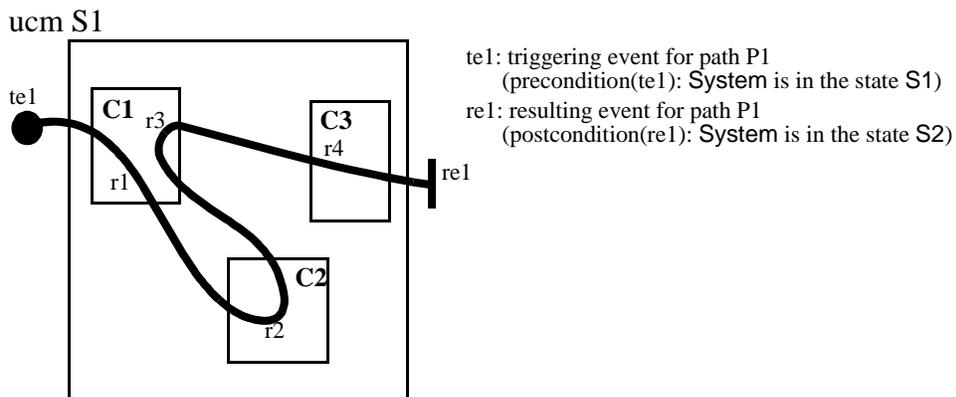
6. A *component context diagram* is a diagram in which only the topology of the components that compose the system and its environment are given. The communication links between components are not shown in a component context diagram.

responsible for executing scenario responsibilities. In the UCM modeling phase, we define system components, and allocate to them scenario responsibilities.

In UCM modeling, responsibility allocation provides the mechanism by which system behavior paths and system components are linked together. When allocating responsibilities to components, designers must ensure that the responsibilities that are allocated to a component are consistent with the role the component plays in the system. The role description of the components constitutes the main information used by designers for the purpose of responsibility allocation.

An example of component definition and responsibility allocation is given in Figure 39. In this example, three component, C1, C2, and C3, have been defined to execute scenario S1. In terms of responsibility allocation, responsibility r1 and r3 are allocated to component C1, responsibility r2 is allocated to component C2, and responsibility r4 is allocated to component C3.

FIGURE 39. Generation of UCM from UCM requirements



Component Decomposition

In the UCM modeling phase, designers may also decompose existing components into a set of subcomponents.

3.4.3 Path

Specification of Interactions and Concurrency

One of the important features of the UCM modeling technique is the ability to describe interactions and concurrency between scenarios. In the UCM modeling phase, one of the main responsibilities of the designers is to specify these inter-scenario relationships. This is achieved by combining sets of related paths in composite maps (section 2.2.7), each of which specifying a set of important relationships between system scenarios. Inter-scenario interactions are specified using the UCM path interaction notation (section 2.2.6).

Examples of scenario interactions are illustrated in section 2.2.6 (Figure 11).

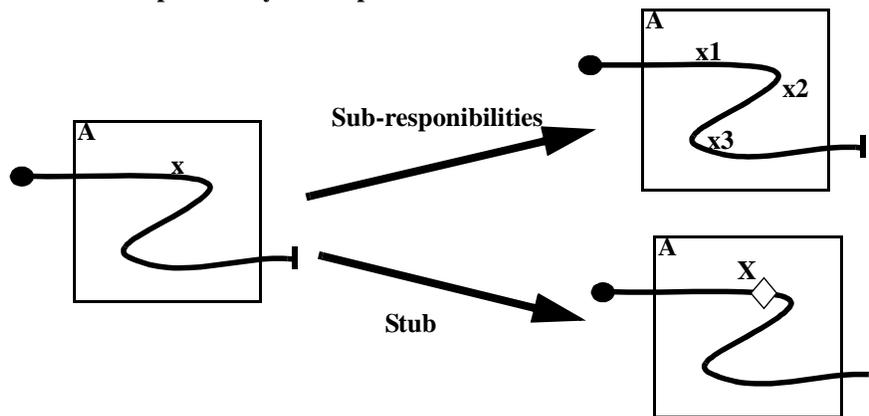
Path Restructuring

At this stage, designers may also need to restructure UCM maps at the path level. We distinguish two types of path restructuring:

- Responsibility decomposition.

Responsibility decomposition allows refining existing responsibilities into sub-responsibilities or stubs. An example of responsibility decomposition is given in Figure 40. In this figure, the two types of responsibility refinement are illustrated. In the first case (top right side of the figure), responsibility x is refined into a sequence of three sub-responsibilities $x1$, $x2$ and $x3$. In the second case (bottom right side of the figure), responsibility x is replaced by a stub X .

FIGURE 40. Responsibility decomposition

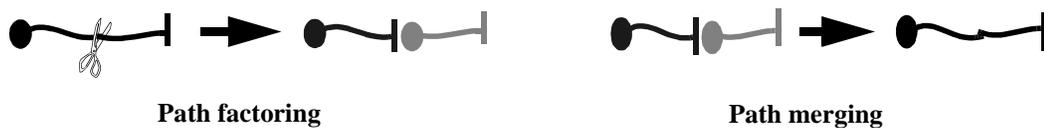


- Path restructuring.

Path restructuring may include *path factoring*, i.e. cutting a path into a set of sub-paths, and *path merging* (the opposite process), i.e. merging a set of individual paths into a more complex path or into a related path set.

The two types of path restructuring are illustrated in Figure 41.

FIGURE 41. Path factoring and merging

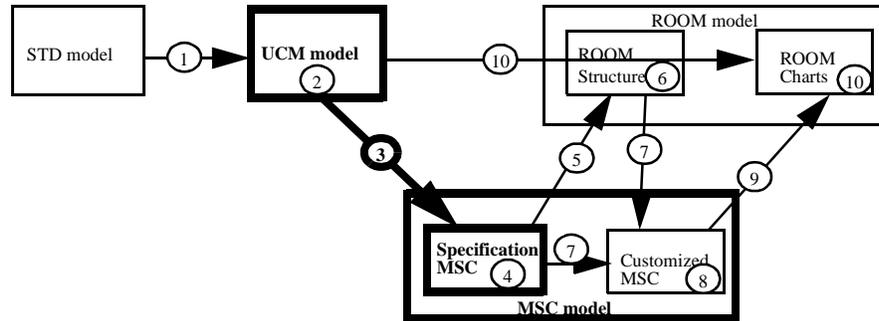


3.5 Transition from UCM to MSC

The objective of the transition between UCM and MSC is to express each of the paths contained in the UCM model as a sequence of inter-component messages and component actions. The input of the transition is a new version of the UCM model, and the output is a

set of new specification MSCs. The transition between UCM and MSC is highlighted in Figure 42.

FIGURE 42. Transition from UCM to MSC



The generation of a specification MSC model from a UCM model is conducted in two steps:

1. Generation of MSC Skeletons

This step consists in generating a MSC skeleton (see definition of MSC skeleton in section 2.3.5) for each UCM related path set contained in the UCM model. This step can be completely automated, and therefore would not have to be performed by designers.

2. Definition of Message Sequences

This step consists in expressing each of the UCM responsibilities in terms of a message sequence, or more generally in terms of a sequence of MSC activities⁷.

In the following sections, we separately describe these two steps. But before going on with the description of these two steps, we analyze the semantic relationship that exists between UCM and MSC models.

7. MSC activities include both messages and actions. See section 2.3.1 for more details.

3.5.1 Relationship between MSC Models and UCM Models

UCM and MSC are two modeling techniques that focus on scenario description. In order to establish a transition between models produced by these two modeling techniques, we need to analyze how their respective concepts relate to each other. Here is a comparison of their main concepts:

- **Scenario Description.** One of the main conceptual differences between UCM and MSC lies in the abstraction level at which they describe scenarios. UCM describes scenarios in terms of sequences of responsibilities. UCM responsibilities are described at a high level of abstraction by a label and a brief textual description. Such a description abstract away inter-component communication. On the other hand, MSC describes scenarios in terms of sequences of inter-component messages and internal actions.
- **Components.** UCM and MSC both allow for component definition. On one hand, UCM defines components in terms of the role they play in a scenario (by means of a short textual description), and the set of responsibilities they must provide in the context of the scenario. UCM does not provide notation to describe the internal logic of components. UCM allows describing, at a high level of abstraction, what a component does in the context of a scenario, but it does not describe how the component does it. On the other hand, MSC allows for component description at a lower level of abstraction. MSC describes component behavior mainly in terms of a set of conditions, that may be viewed as states, and a set of messages exchanged between the component and its environment. (A set of internal actions may also be included.)
- **Related path set.** Both description techniques provide for related path set description. On one hand, UCM describes scenarios by means of UCM related path sets, which allows composing several alternate paths into a single UCM diagram using segment connectors. On the other hand, MSC uses the HMSC notation to describe related path sets. Each HMSC describes a set of related message sequences.

The main difference between the two modeling techniques lies in the fact that UCM uses a single type of notation to describe both individual scenarios and related path sets, while MSC uses the basic MSC notation to describe individual scenarios, but normally uses the HMSC notation to describe related path sets⁸.

- **Scenario interactions.** The capability to explicitly describe scenario interactions is a feature that exists in UCM, but does not exist in MSC. Scenario interactions are captured in UCM models using path interaction notation (see section 2.2.6).

In conclusion, UCM and MSC are two scenario modeling techniques that express scenario in the context of system components. However, they describe scenarios and components at different levels of abstraction.

Relationship between elements of UCM and MSC Models

Prior to defining the details of the transition between UCM and MSC, it is important to understand the relationship that exists between elements of the UCM and MSC models. A schematic view of this relationship is illustrated in Figure 43.

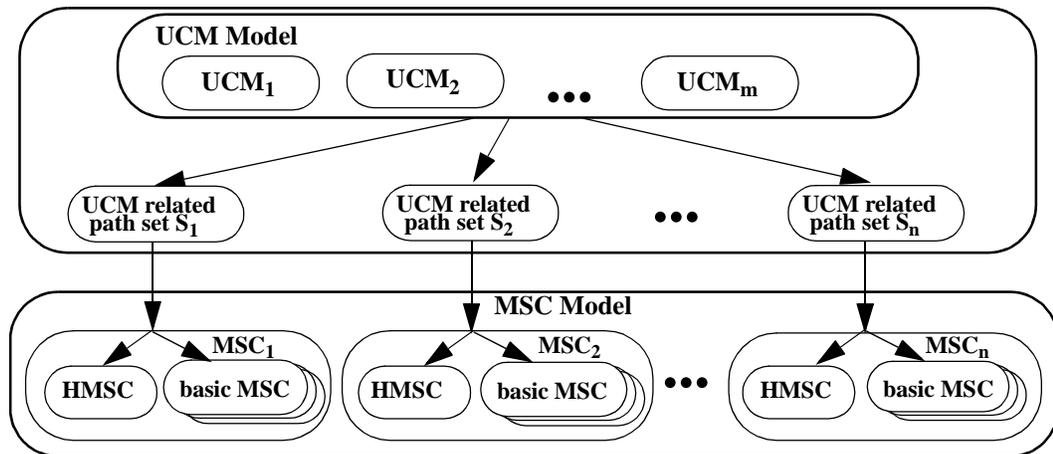
A UCM model is composed of a set of UCM maps, labelled UCM_1 , UCM_2 , and UCM_m in Figure 43, and a set of UCM related path sets, labelled S_1 , S_2 , and S_n (see section 2.2.8 for more details). Similarly, a MSC model is composed of a set of individual MSCs, labelled MSC_1 , MSC_2 , and MSC_n in Figure 43 (see section 2.3.4 for more details).

In the RT-TROOP modeling process, we establish a one-to-one relationship between UCM related path sets and MSCs. We define a basic MSC for each path segment contained in the UCM related path set, and define a HMSC that captures the relationship

8. It should be noted here that related path sets can be completely described in basic MSCs using the inline expressions. The two notations are semantically equivalent. A HMSC can be transformed into an equivalent basic MSC, and vice-versa. However, the cumbersomeness of basic MSCs rapidly increases with the number of scenarios contained in a related path set. The resulting basic MSCs may become very difficult to read. For this reason, we strongly recommend using the HMSC notation to describe related path sets.

between the path segments. The HMSC describes the possible sequences of basic MSC execution.

FIGURE 43. Relationship between UCM and MSC



At a detail level, we establish relationships between the following elements of the two models:

- UCM components and MSC instances
- UCM triggering and resulting events and MSC messages
- UCM preconditions and postconditions and MSC conditions (expressing system states)
- UCM responsibilities and MSC sequences of messages and actions
- UCM path segments and basic MSCs
- UCM path connectors and HMSC alternative and parallel composition constructs (MSC reference connectors)
- UCM stubs and MSCs (which could be either HMSCs or basic MSCs)

It is important to note that when going from UCM to MSC, the explicit interactions between paths expressed in UCM maps are lost⁹. MSC does not allow to explicitly express interaction between scenarios. The interactions between them are performed through con-

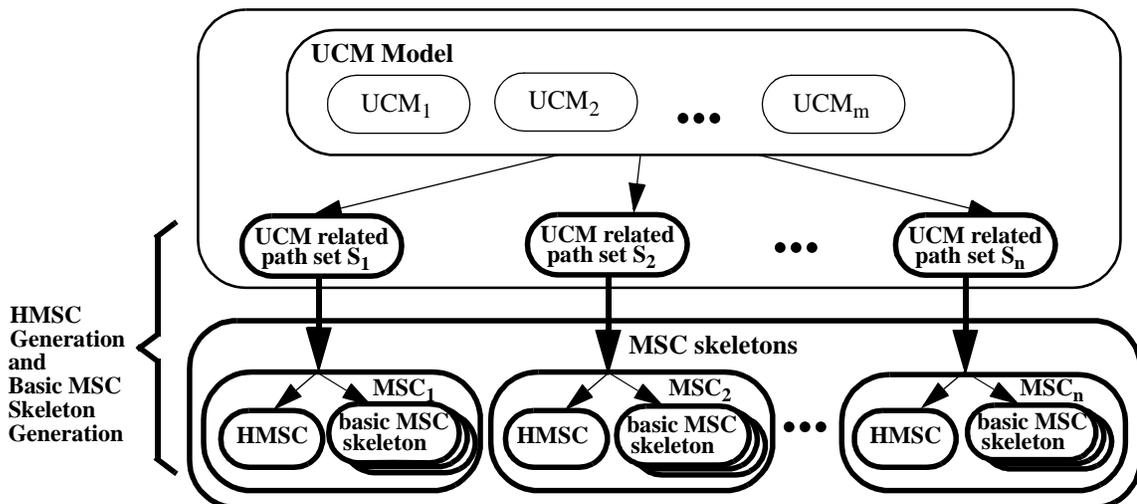
dition sharing. If we think of MSC conditions as system states, it means that when the system is in a given state S , any MSC starting in state S may be triggered (but only one will be triggered).

3.5.2 Generation of MSC Skeletons

The objective of this step is to generate an MSC skeleton for each related path set contained in the UCM model (see definition of UCM related path set in section 2.2.5).

As illustrated in Figure 44, we associate a MSC skeleton with each UCM related path set contained in the UCM model. An MSC skeleton is composed of an optional HMSC (there is no need to produce a HMSC if the UCM related path set only contains one path segment) and a set of basic MSC skeletons (one per MSC references contained in the HMSC). MSC skeletons are simply labelled $MSC_1, MSC_2 \dots MSC_n$ in the figure. MSC_1 is associated with related path set S_1 ; MSC_2 is associated with related path set S_2 , and so on.

FIGURE 44. Generation of HMSC and basic MSC skeleton from UCM



9. An example of this is given in section 5.4, where the abort interaction between the stopPrinting path and the printFile path is lost in the transition between UCM and MSC.

The generation of MSC skeletons from UCM related path sets is conducted as follows:

- One HMSC is created for each related path set (the HMSC expresses the path segment structure of a related path set).
- One basic MSC skeleton is created for each path segment contained in related path set.

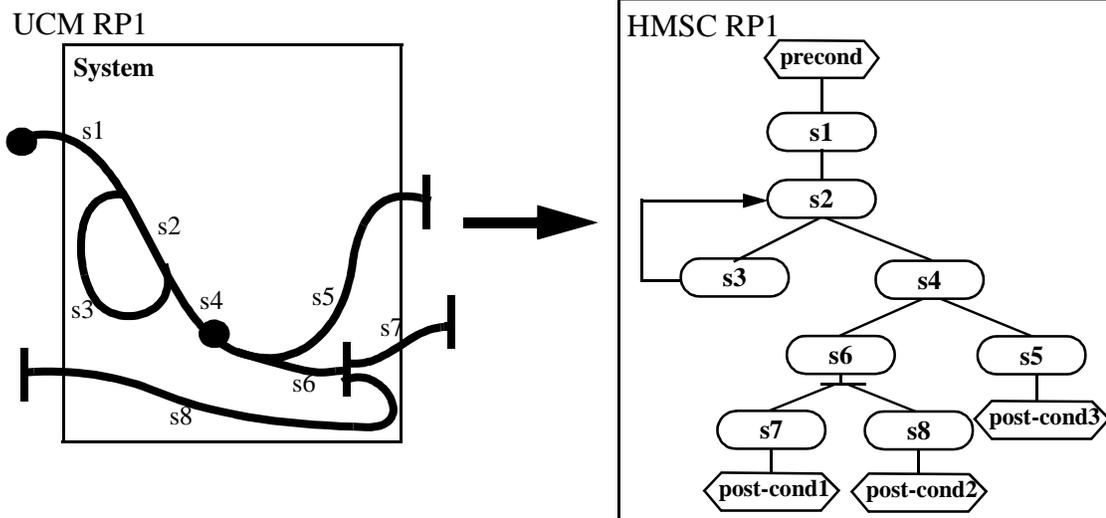
In the rest of this section, we separately describe the generation of HMSCs, and the generation of basic MSC skeletons. These steps can be considered as two distinct steps. They can be completely automated.

3.5.2.1 Generation of HMSCs

In this step, we are concerned with the generation of a HMSC that reflects the path segment structure of the UCM related path set. The generation of a HMSC from a related path set is conducted as follows:

- One MSC reference is created in the HMSC for each path segment contained in the UCM related path set.
- MSC references are connected together using HMSC reference connectors (see section 2.3.2) to reflect the path segment structure of the UCM related path set.
- The precondition and postcondition associated with the UCM related path set are explicitly introduced in the HMSC as MSC conditions.

An example of the generation of an HMSC from a UCM is given in Figure 45. Because system components are not expressed in HMSCs, we have chosen to not illustrate them in the UCM of Figure 45. In this figure, we observe that one MSC reference box is generated for each path segment contained in the related path set, and that the connection between MSC reference reflects the path segment connections of the UCM related path set.

FIGURE 45. Generation of an HMSC from UCMs

If the related path set is composed of a single path segment, then it may be described directly by means of a basic MSC.

In this step, explicit traceability may be maintained by labelling each MSC reference with the identifier of the UCM path segment to which it corresponds.

This step can be completely automated in a tool.

3.5.2.2 Generation of Basic MSC Skeletons

The generation of a basic MSC skeleton from a UCM path segment is conducted as follows.

- One MSC instance (component) is created for each component contained in the UCM related path set.
- If the path segment contains a start point, then a triggering event message arrow is created and placed at the top of the MSC, just after the system initial state (condition)¹⁰. This arrow is connected on its sender side to the component that generates the triggering event. This component may be either an internal component of the system, if the

execution of the path is triggered by an internal component, or to the MSC frame, if the path is triggered by an external component (user) not explicitly represented in the UCM. However, since the receiver of the message is not yet defined at this stage, the message arrow remains unconnected on its receiver side.

Since the execution of a path always requires a triggering event to start, a triggering event message arrow always needs to be defined in a MSC. In cases where a MSC is defined at the highest level by means of a HMSC, the triggering event message arrow is placed in the basic MSC skeleton that corresponds to the first path segment executed in the UCM related path set to which the MSC corresponds.

- If the path segment contains a UCM end bar for which a resulting event is specified, then a resulting event message arrow is created and placed just above the final condition (postcondition) in the basic MSC. It is connected on its receiver side to the component to which the resulting event is destined. This component may be either an internal component or the MSC frame in cases where the resulting event goes back to the environment¹¹. Since the sender of the resulting event is not yet defined at this stage, the resulting event message arrow remains unconnected on its sender.

If the path segment contains a UCM end bar, but no resulting event is specified for the end bar, then no resulting event message arrow is created. While the execution of a path always needs a triggering event to start, it may terminate without sending back a resulting event. Thus, resulting event message arrows are not defined in all cases.

- A responsibility is shown using a thick black line segment placed on the instance axis to which it has been allocated. The causal ordering of the responsibility is maintained through the transition. Thus, responsibilities are placed sequentially on the component timelines.

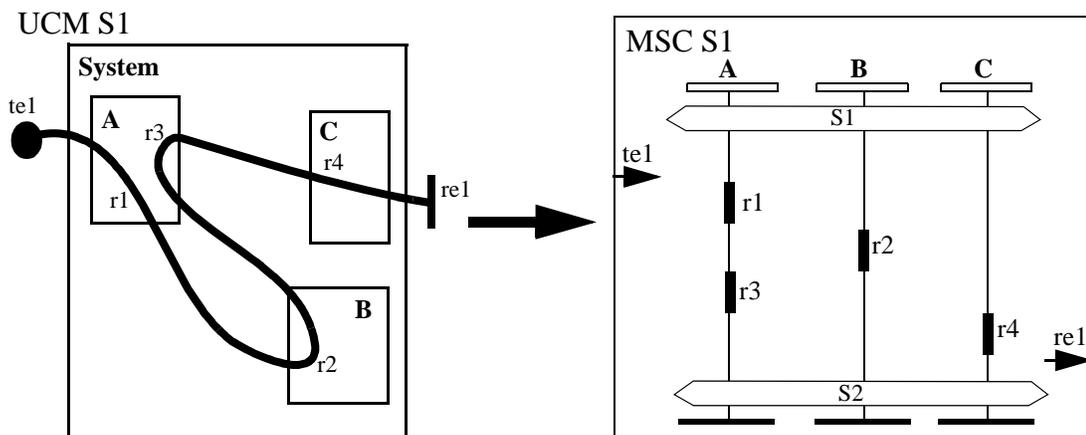
10. As defined in section 2.3.5, MSC skeletons do not contain message arrows. However, because triggering and resulting events are defined in the UCM models, we introduce triggering and resulting event messages in MSCs in this phase.

11. The environment is composed of the set of components that are external to the system and that participate in the execution of the scenario.

In this step, we maintain explicit traceability by labelling the MSC with the same identifier as the UCM, and by labelling components, state conditions and responsibilities with the same identifiers as the ones used in the UCM.

In Figure 46, an example of the generation of a basic MSC skeleton from a path segment is illustrated. We observe that one MSC instance is created for each component identified in the UCM, and that responsibilities have been placed on the different timelines. The sequential ordering of the responsibilities is preserved in the transition from UCM to MSC. We also observe that the triggering event and resulting event arrows have been placed on the frame of the UCM, but have not been connected to any component timeline. The connection of these arrows to particular component timelines will be done in the message sequence definition step (section 3.5.3). Also, the pre and postconditions that apply to the UCM path are expressed as initial and final conditions, labelled as S1 and S2 in Figure 46, in the MSC.

FIGURE 46. Generation of an MSC skeleton from a UCM path



te1: triggering event for UCM S1
 (precondition(te1): System is in the state S1)
 re1: :resulting event for UCM S1
 (postcondition(re1): System is in the state S2)

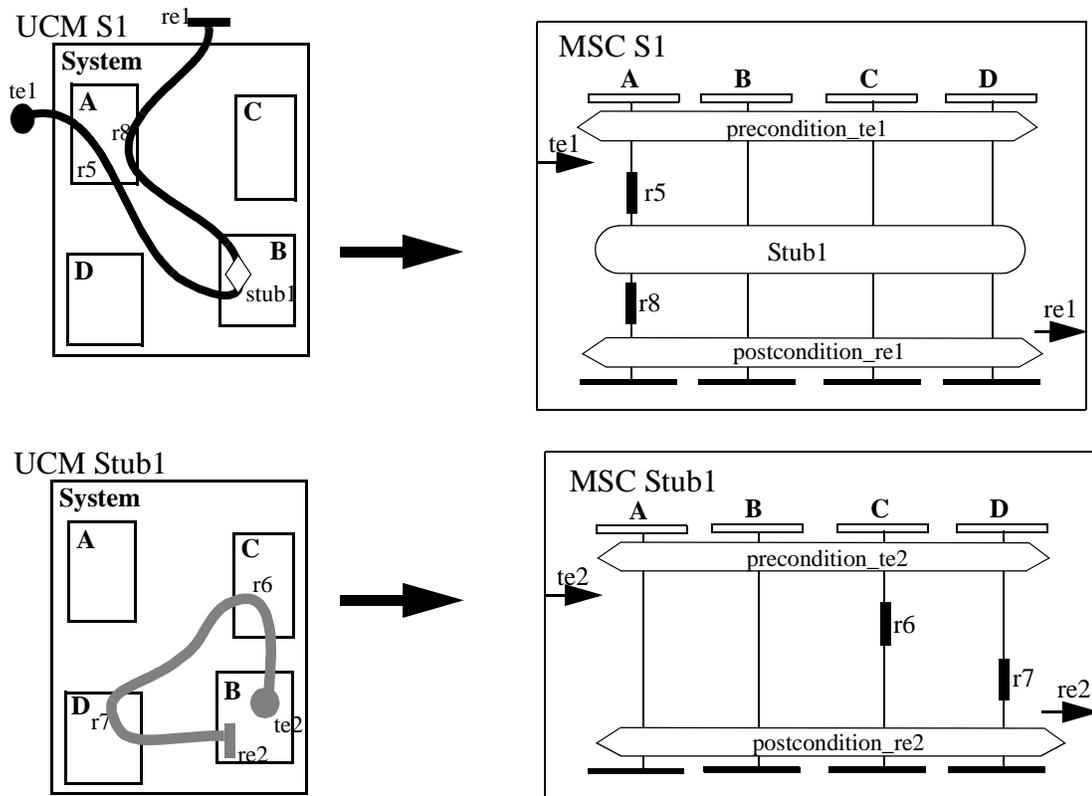
UCM Stubs in Basic MSC Skeletons

Stubs¹² are represented in basic MSCs using MSC references.

In Figure 47, an example of expressing a UCM stub in a basic MSC skeleton is illustrated. On the top left corner of the figure, the UCM S1 is given. This UCM is mainly composed of two responsibilities, r5 and r8, and a stub, stub1. In the bottom left corner of the figure, the UCM corresponding to Stub1 is given. This stub is composed of two sequential responsibilities, r6 and r7.

On the top right corner of the figure, the basic MSC skeleton, MSC S1, that corresponds to UCM S1 is given. We observe in this basic MSC skeleton that the UCM stub is expressed as a MSC reference. The basic MSC skeleton that corresponds to stub1 is illustrated in the bottom right corner of the figure.

FIGURE 47. Expressing a UCM stub in a basic MSC skeleton



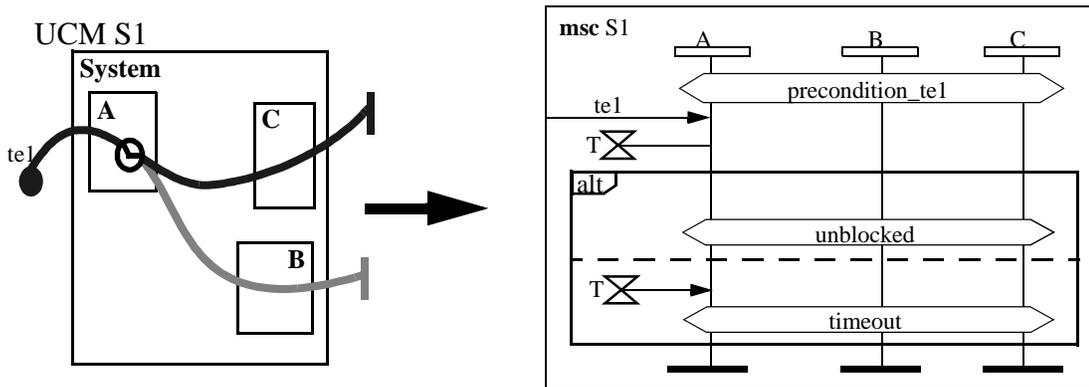
12. See definition of stubs in section 2.2.4.

UCM Timers in Basic MSC Skeletons

Timers are represented in basic MSCs using MSC timer notation. If a UCM path segment contains a timer (or timed waiting place), then an MSC timer is introduced in the basic MSC skeleton. Timer message (`setTimer`, `timeout`, and `clearTimer`) arrows are also inserted in the MSC skeletons.

In Figure 48, an example illustrating the generation of a basic MSC skeleton from a UCM containing a timer is given. On the left side of the figure, UCM S1 is given. This UCM is composed of a first path segment that contains a timer and two alternate paths: one triggered by the reception of a timeout event and one triggered by the reception of an unblocking event. On the right of the figure, the basic MSC skeleton that corresponds to the first path segment of the UCM, i.e. the one containing the timer, is given¹³. We observe in this basic MSC skeleton that the UCM timer is expressed by means of an MSC timer. Also, an alternative inline expression is introduced immediately after the MSC timer. This inline expression contains two operands: the first one corresponds to the case where an unblocking message is received (this unblocking message is undefined in the MSC skeleton), and the second one corresponds to the case where a `timeout` message is received from the timer.

¹³In this example we only illustrate the basic MSC skeleton that corresponds to the first segment of the UCM. The generation of an MSC skeleton from the complete UCM related path set requires the use of one HMSC and three basic MSC (one of each path segment).

FIGURE 48. Expressing a UCM timer in a basic MSC skeleton

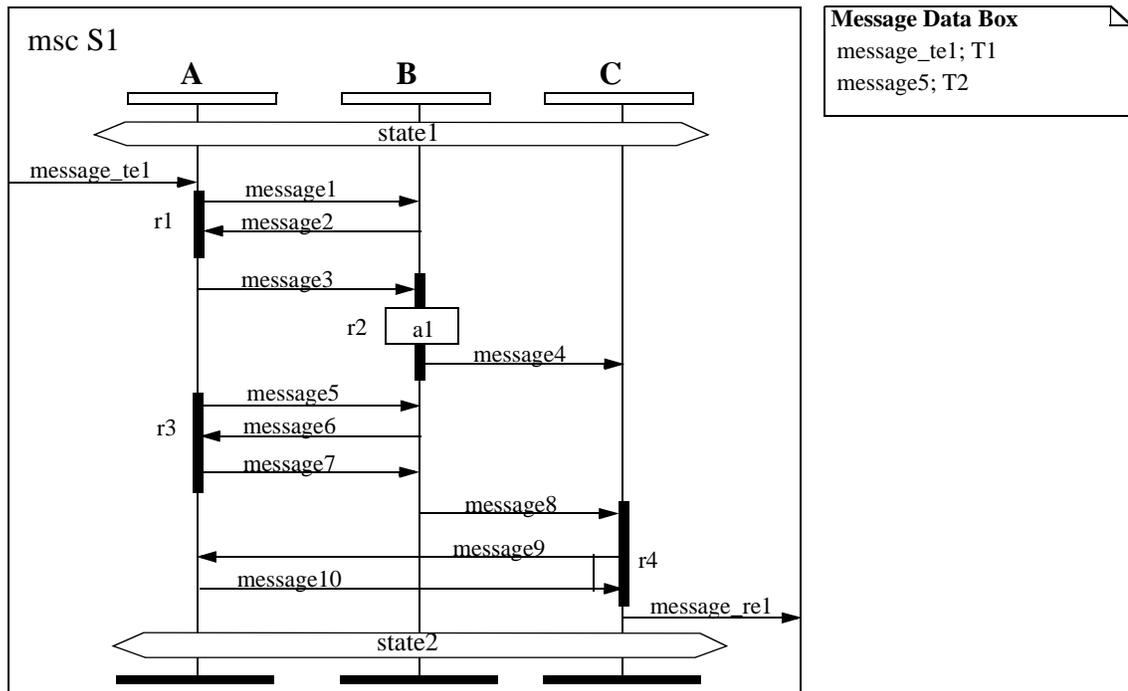
We envision that this step could be completely automated in a tool.

3.5.3 Definition of Message Sequences

The objective of this step is to express each of the UCM responsibilities placed on the instance axis of the different basic MSC skeletons in terms of a sequence of messages and actions. We also define the data associated with messages in message data boxes.

In Figure 49, an example of message sequence definition in an MSC is given. This MSC constitutes a refinement of the basic MSC skeleton given in Figure 46. We observe, in this figure, that the triggering event (`message_te1`) and resulting event (`message_re1`) have both been connected to specific instances (components) in the MSC, which are respectively A and C. Also, we observe that every responsibility illustrated in Figure 46 has been expressed as a sequence messages and actions.

FIGURE 49. Description of message sequence in MSC S1



In this figure, different types of responsibility refinement are illustrated.

- Responsibility **r1** is expressed as a sequence of two messages (**message1** and **message2**): one going from A to B, and one from B to A.
- Responsibility **r2** is expressed as a message (**message3**) send from A to B, followed by the execution of action **a1** by component B, and completed by a message (**message4**) send by B to C.
- Responsibility **r3** is expressed as a sequence of three messages (**message5**, **message6** and **message7**) send between A and B.
- Responsibility **r4** is expressed as a message (**message8**) sent from B to C, followed by a call-return message exchange from C to A (**message9** and **message10**).

These only illustrate some of the possible types of responsibility refinement. Other types of refinement are also possible. In particular, MSC parallel composition and alternative inline expression can be used.

Standard communication patterns could also be plugged in the basic MSC skeleton to provide the detailed level communication mechanisms that are required to enable inter-component communication. An example of such design patterns is given in [22].

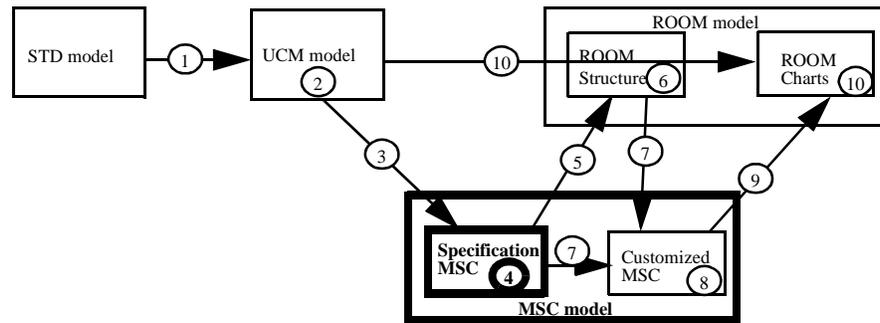
Data Definition

In RT-TROOP, the data objects associated with inter-component messages are specified in the specification MSC modeling phase. The data information defined here will later be used in the ROOM structure modeling phase (section 3.7.3) to define the different protocol classes. In the RT-TROOP, the data objects associated with the different messages are described in message data boxes associated with MSCs (Message Data Box are defined in section 2.3.3).

An example of data definition in a specification MSC is given in Figure 49. In this MSC, two messages involving data are defined: `message_te1` that involves a data object of type T1, and `message5` that involves a data object of type T2.

3.6 Specification MSC Modeling

The objectives of the specification MSC modeling phase is to integrate the set of new specification MSCs defined in the transition between UCM and MSC in the specification MSC model of the system, and to add new elements to the MSC model. There are two main inputs to this modeling phase: the set of new specification MSCs produced in the transition between UCM and MSC, and the existing specification MSC model (i.e. the one produced in the previous iteration). The output is a new version of the specification MSC model. The specification MSC modeling phase is highlighted in Figure 50.

FIGURE 50. Specification MSC modeling phase

3.6.1 Modeling Activities

The abstraction mechanism plays a very important role in the modeling of complex real-time system. This mechanism consists in considering only a subset of the system elements at each level of abstraction. Thus, details are progressively added to the system models as we move from requirements, to high level models, to detailed level models, to implementation. In the specification MSC modeling phase, details with respect to components (instances), messages, and system states (conditions) are added to the system model. The level of details is increased by adding new elements or by refining existing ones.

The modeling activities that may be carried out in the specification MSC modeling phase includes the following:

- Introduction of new components
- Decomposition (or refinement) of existing messages into a set of more detailed ones
- Definition of new system states (MSC conditions)
- Restructuring of the MSC model

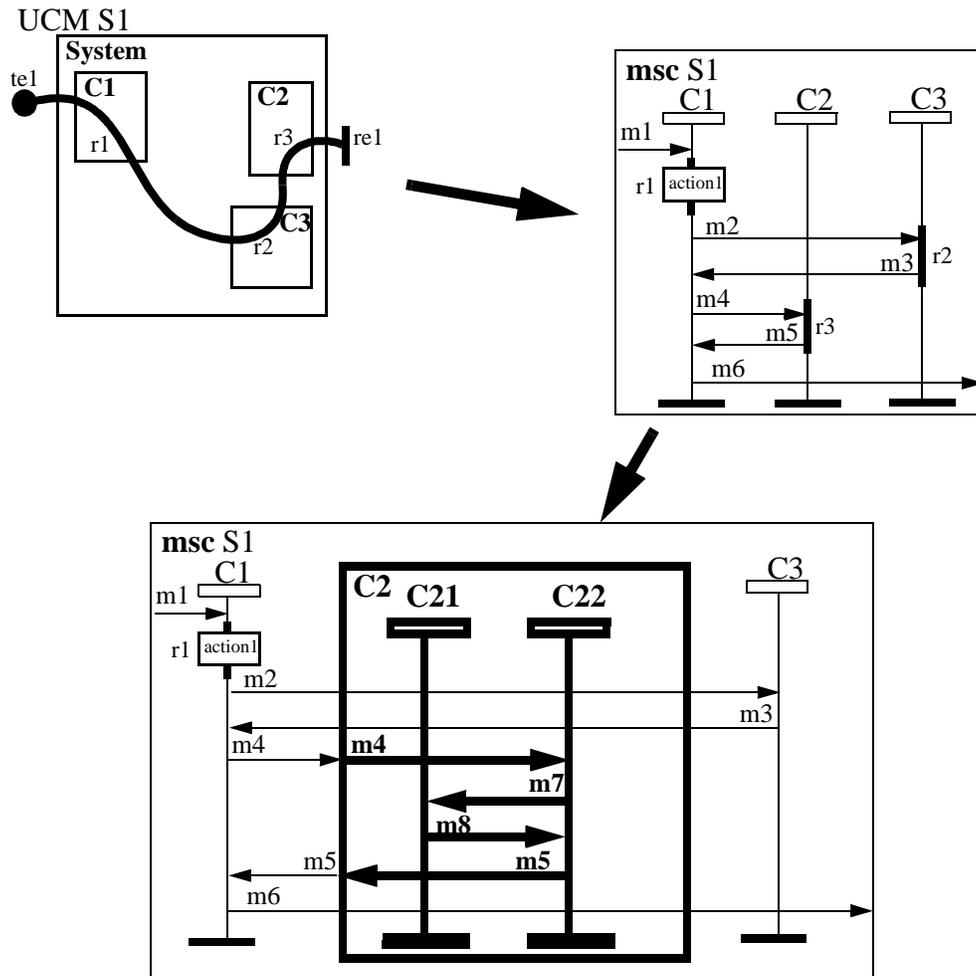
Definition of new components

When designing real-time systems, new system components often need to be introduced in the system as we move towards implementation. In the context of RT-TROOP modeling, the UCM model constitutes the high level model of the system. This model only contains the main components of the system. These components correspond to entities, or objects, that play a significant role in the context of the high level scenarios described in the UCMs. However, as we move towards a ROOM detailed level model, new components usually need to be introduced. These new components may result from the decomposition of already existing components. They may also correspond to entities, or objects, that were not important in the context of the UCM high level modeling, but that need to be introduced in detailed level modeling.

In RT-TROOP modeling, detailed level components are introduced at the MSC level, and more precisely in the specification MSC modeling phase.

An example of MSC component decomposition is given in Figure 51. This figure illustrates the generation of a MSC from a UCM map, followed by the decomposition of component **C2** in the MSC. The MSC model elements introduced by the component decomposition are highlighted in the figure. For the sake of clarity, we did not represent the initial and terminal state conditions in the MSC.

FIGURE 51. Component decomposition in MSC model



Message Decomposition

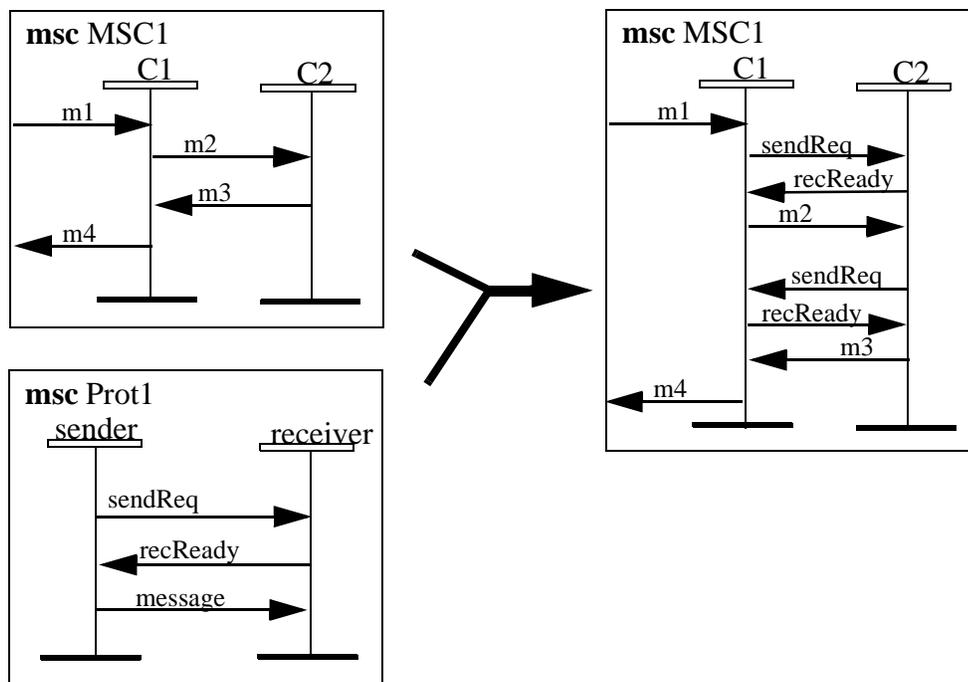
Similarly to system components, inter-component messages often need to be decomposed into sequences of more detailed messages as we move down the abstraction levels. RT-TROOP modeling provides for message decomposition at the MSC level.

There are different reasons for carrying out message decomposition. In some cases, it is purely a question of abstraction, where a message defined in a previous version of the model needs to be refined to reflect the increasing level of details of the system models.

For example, what is considered to be a single message sent from one component to another early in the design process may later need to be replaced (or implemented) by a sequence of messages to satisfy security or robustness communication requirements. In other cases, messages need to be decomposed to satisfy new requirements.

An example of MSC message decomposition is illustrated in Figure 52. In this figure, each of the two messages exchanged between components C1 and C2 are replaced by a sequence of three messages corresponding to the communication protocol Prot1 used between the two components.

FIGURE 52. Message decomposition



Definition of System States

System states are represented in MSCs by means of conditions (see definition in section 2.3.1). In MSC models, conditions provide the mechanism by which MSCs can be combined into larger MSCs. The definition of every HMSC and basic MSC must contain

the specification of both a precondition (also called initial condition) and a postcondition (also called end condition). In cases where the specification of a condition is missing, this condition is implicitly considered to be TRUE.

In the transition between UCM and MSC, only the preconditions and postconditions associated with related path sets have been defined. At the MSC level, this means that the precondition and postcondition of the HMSCs have been completely specified, but only the precondition of basic MSCs containing triggering events, and the postcondition of basic MSCs containing resulting events have been defined.

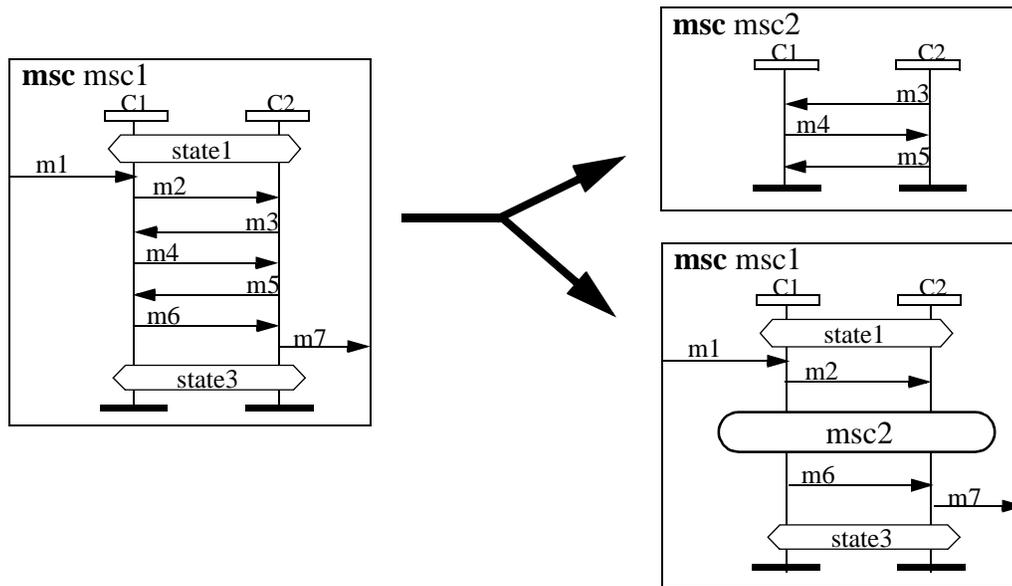
In this modeling phase, the precondition and postcondition of basic MSCs may be specified in terms of system states. They may also be left to be defined in the Component Behavior Modeling in MSC modeling phase (section 3.10). Also, other intermediate system states may be specified in the basic MSCs as required.

MSC Restructuring

In this modeling phase, the specification MSC model may also be restructured. MSC model restructuring may be required for several reasons such as to increase reusability, or to decompose MSCs that have become too large over the iterations in simpler ones.

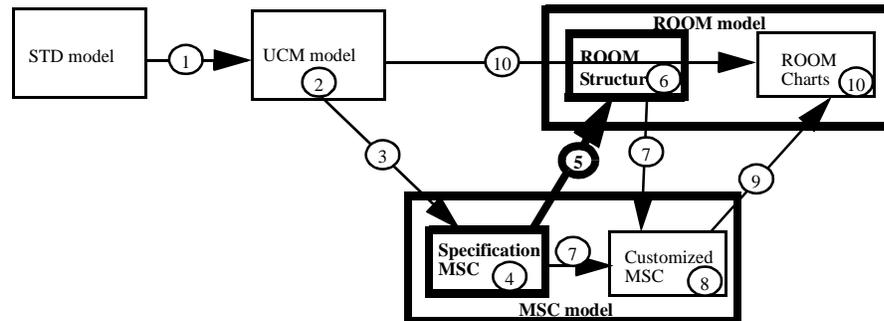
An example of MSC restructuring is given in Figure 53. In this example, MSC `msc1` given in the left side of the figure is restructured. The restructuring is done in two phases. First, a new MSC, labeled `msc2`, containing messages `m3`, `m4`, `m5` is defined. Second, messages `m3`, `m4`, `m5` of MSC `msc1` are replaced by an MSC reference to the newly defined MSC `msc2`. The main advantage of this type of restructuring is that it allows for the reuse of the newly created MSC.

FIGURE 53. MSC restructuring



3.7 From MSC to ROOM Structure

The ROOM structure modeling phase makes the transition between the specification MSC model and the ROOM structure model. The objective of this phase is to define a set of ROOM role structures on a per scenario (or related path set) basis. The input of the transition is a new version of the specification MSC model, and the output is a set of ROOM role structures. The transition from MSC to ROOM structure is highlighted in Figure 54 in the context of the overall RT-TROOP modeling diagram.

FIGURE 54. Transition from MSC to ROOM structure

In this section, we:

- Analyze the overall relationship that exists between ROOM models and MSC models.
- Describe the generation of a ROOM structure model from a MSC model.
- Describe the definition of ROOM role structures from MSCs.

3.7.1 Relationship between ROOM Model and MSC Models

MSC and ROOM are two modeling techniques that primarily rely on inter-component communication. They describe systems as communicating state machines. The model they produce are mainly composed of components, states and messages exchanged between components. However, these two modeling techniques significantly differ with respect to the two following aspects:

- **Modeling focus.** MSC is a scenario-centric modeling technique that describes scenarios at the level of inter-component communication. It describes systems on a per scenario basis. A MSC model is composed of a set of MSCs, each of which describes a possible sequence of messages (and internal actions) exchanged between system components and between the system and its environment. MSC models can be used as a specification for more detailed models.

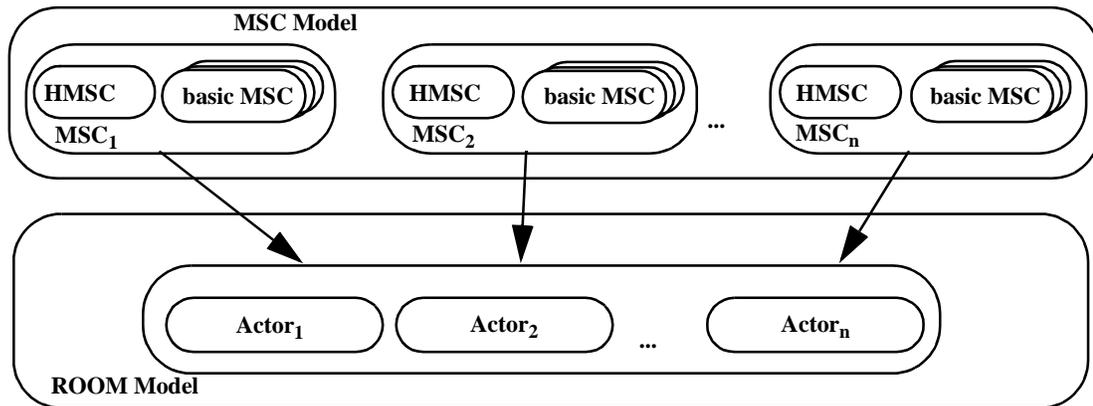
On the other hand, ROOM is a component-centric detail level modeling technique. It aims at describing system structure and actor behavior in the context of the overall system, i.e. in the context of a set of different scenarios. Thus each actor in a ROOM model is responsible for the execution of a set of system responsibilities that may be part of many different scenarios.

The modeling focus difference between MSC and ROOM can be summarized by saying that saying that MSC is scenario-driven, while ROOM is component-driven. These constitute two orthogonal views of real-time systems.

- **Level of details.** MSC models describe systems in terms of components, states, messages, and internal actions. These model elements are atomic in MSC; they are simply described by means of an identifier. The fact that MSC does not require a detailed description of the different model elements allows a designer producing system models that are independent of implementation.

On the other hand, ROOM uses two different levels of modeling: a schematic level at which system structure and actor behavior internal logic are defined, and a detail level at which model elements such as communication protocol, (transition, entry, and exit) action code, and functions are defined at the level of programming languages. The fact that ROOM uses a programming language at the detail level facilitate the transition between ROOM models and implementations. It also maintains a complete traceability between ROOM models and implementations.

A schematic view of the relationship between MSC and ROOM is given in Figure 55. This figure illustrates the fact that when moving from MSC to ROOM, modelers have to consider all the different MSCs, and that any given ROOM actor can be involved in the execution of several different MSCs. When making the transition from MSC to ROOM, modelers must ensure that the resulting ROOM model satisfies the whole set of MSCs.

FIGURE 55. Relationship between MSC and ROOM

In order to analyze the conceptual relationships between MSC and ROOM elements, we separately consider structure (section 3.7.2) and behavior (section 3.11.1). In these sections, the diagram of Figure 55 will be refined to reflect the structure and behavior relationship between UCM and ROOM.

3.7.2 Generation of a ROOM Structure Model from a MSC Model

In real-time system development, structure modeling requires the definition of:

- Components
- Component interfaces
- Inter-component communication links

System structures are described in MSC by means of a set of components and a set of messages exchanged between components. At the MSC level, component interfaces and inter-component communication links (or channel) are not explicitly defined. They are left out to be defined at a more detailed modeling level.

In ROOM, a system structure is described in terms of a set of actors and a set of contracts¹⁴. Also, each ROOM actor is defined, at the structure level, by means of a set of for-

mally defined interface components, called ports, that specify the set of messages that can be send and received by the actor on a given port.

In the RT-TROOP modeling process, a ROOM structure model is produced from a set of MSCs. In order to reduce the complexity associated with the definition of system structures involving a large number of scenarios (MSCs) and components, we proceed in two steps:

1. Definition of role structures

This step consists in defining structures on a per scenario (or related path set) basis. We call such structures *role structures* because they only involve parts of the elements that compose the overall system structure. This way, only subsets of the system scenarios and components are handled at once.

The definition of the role structures is the objective of the current modeling phase (transition from MSC to ROOM structure).

2. Definition of a global system structure

This step consists in defining a global system structure that integrates all the role structures.

The definition of the global system structure is carried out in the ROOM structure modeling phase (section 3.8)

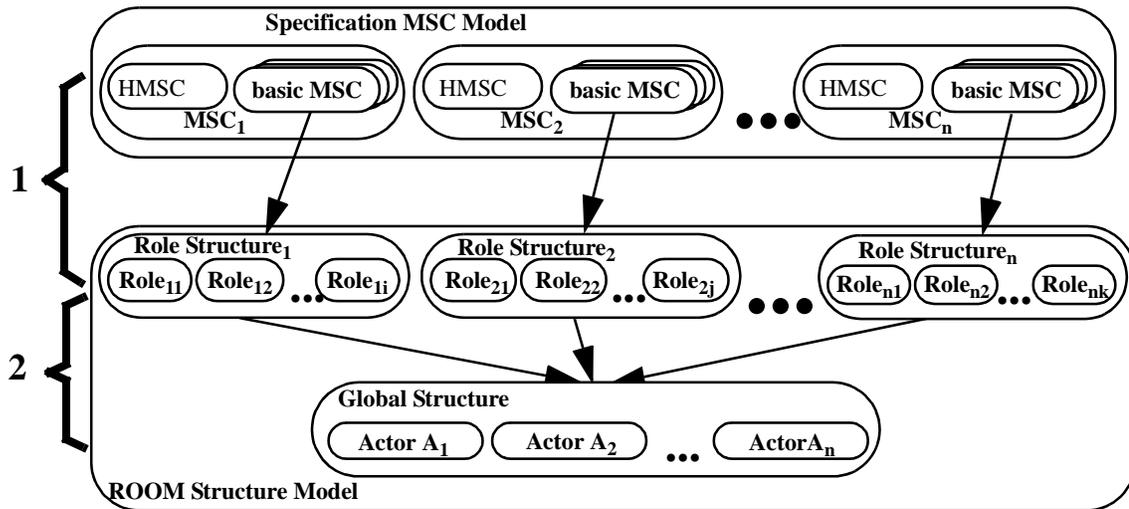
These two steps are illustrated in Figure 56. This figure illustrates the relationships that exist between MSCs and ROOM role structures, and between a set of ROOM role structures and a ROOM global structure. Components that compose role structures are called *roles*. That is because from a system point of view, the components defined in the MSC model represent a set of roles that need to be filled in the system implementation. In ROOM, roles are modeled by means of actors.

14.A contract consists of a binding (inter-component communication link) and a pair of ports (interface components) that it connects

In the first step, i.e. the one between MSCs and ROOM role structures, one role structure is defined for each MSC contained in the specification MSC model. A ROOM actor is created for each MSC components, and MSC messages are packaged into a set of protocol classes that are afterwards used to define actor ports. ROOM bindings are also defined to connect pairs of ports. In Figure 56, we observe that the information contained in the HMSCs is not used in this phase. That is because HMSCs contain no information concerning system structure. All the information required to produce a ROOM role structure from an MSC model is contained in the sets of basic MSCs.

In the second step, i.e. the one between ROOM role structures and a ROOM global structure, a set of communicating ROOM actors is defined to filled the roles defined in the role structures. When making the transition between a set of role structures and a global system structure, modelers must ensure that all the roles defined in the role structures are filled by actors in the system structure. One actor in the global structure may fill several roles, and the set of roles filled by an actor do not need to be in distinct role structures. A single actor may play two or more roles in a single role structure. For example, actor A1 in Figure 56 may fill roles $Role_{11}$, $Role_{14}$, $Role_{22}$, $Role_{41}$, $Role_{42}$, $Role_{45}$, and $Role_{n2}$, and actor A2 may filled a single role $Role_{n1}$.

FIGURE 56. Relationship between MSC and ROOM structure



In practice, the definition of a ROOM role structure is not required. The modeler may decide to skip this step, address the whole set of scenarios and components all at once, and directly produce the global ROOM structure of the system without any loss of rigor in the overall modeling process. The reasons for using or not using role structure modeling depends on the number and complexity of the scenarios and components that compose the system, and on the experience of the modeler. However, if defined role structures can be reused in other systems.

3.7.3 Definition of ROOM Role Structures from MSCs

The objective of this step is to define a ROOM role structure that enables the execution of the message sequences defined in the specification MSC model. At this stage, one role structure is defined for each MSC contained in the specification MSC model.

A ROOM structure is composed of a set of actors and a set of contracts that enables communication between actors. In this step, we establish a one-to-one relationship between MSC components (instances) and ROOM actors. Thus, the set of ROOM actors can be

automatically generated from MSC. Consequently, the only model elements that remain to be defined by designers at this stage are contracts. The definition of a contract requires the definition of a pair of ports and a binding.

ROOM role structure definition is conducted in four steps:

1. Definition of actors
2. Identification of inter-component communication
3. Definition of protocol classes
4. Definition of contracts.

In the following sections, we discuss each of these three steps.

For a given MSC, several different structures can be defined. The choice of a particular structure among possible ones constitutes a design decisions that is based on different criterion, such as performance and reuse. At this stage, different design alternatives may be investigated. Also, this step provides an excellent opportunity for reusing existing structures, contracts and standard high-level structural patterns [35].

3.7.3.1 Definition of Role Actors

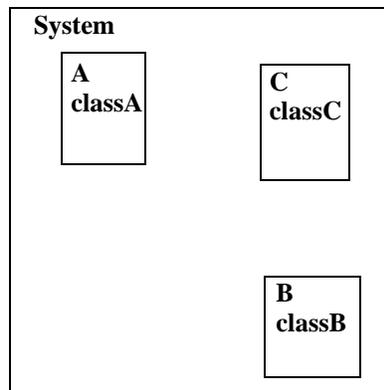
The first thing that must be done when defining a ROOM structure is to define the set of actors that compose the system. In the definition of the role structures, we define the set of role actors.

In ROOM, the definition of an actor requires the specification of the class of the actor and the specification of the properties of the actor. The actor properties that must be specified are the replication factor, if it is different that one, and the structure dynamics properties¹⁵.

¹⁵The structure dynamics aspect has been left out of the thesis. In terms of structure dynamics properties, a ROOM actor is either fixed, optional, or imported. It can also be substitutable. See [93] for more information on this topic.

In the transition between MSC and ROOM structure, we define a ROOM role actor for each instance contained in the MSC. In Figure 57, the set of ROOM role actors that corresponds to the MSC model of Figure 49 are defined: role actor A of class `classA`, role actor B of class `classB`, and role actor C of class `classC`. The replication factor of these actors is implicitly one.

FIGURE 57. Definition of role actors for MSC S1



3.7.3.2 Identification of Inter-Component Communication

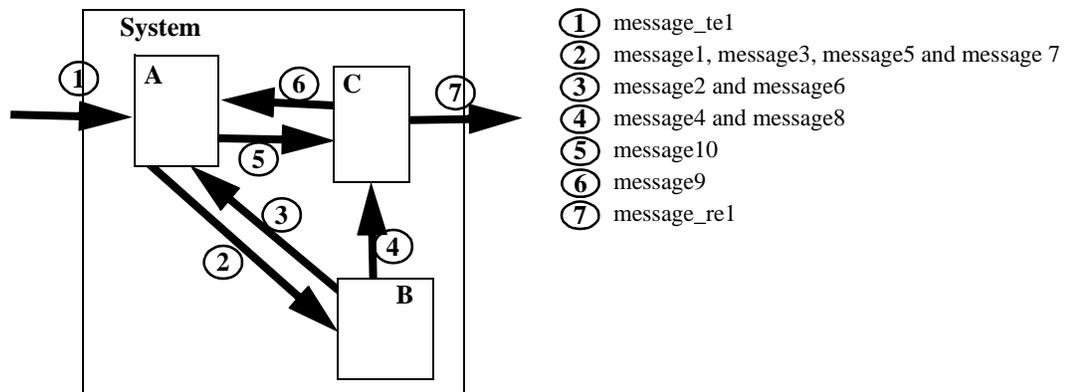
In this step, we identify the pairs of components that communicate together in MSCs, and, for each pair, we identify the set of messages involved in the communication. To help doing that, we use *communication diagrams*. A communication diagram is composed of: a set of system components, a set of unidirectional communication arrows between these components, and a set of messages associated with each communication arrow. In RT-TROOP modeling, these diagrams are used to graphically display the required communication between system components in the context of the component context diagram. The information contained in communication diagrams can be automatically generated from the specification MSC model.

In the RT-TROOP modeling process, this step is intended to be completely automated. Therefore, communication diagrams do not constitute yet another type of models that need to be produced by designers. Such diagrams are automatically generated to help

designer synthesize the information that is required to define the communication protocols.

In Figure 58, an example of communication diagram is given. This diagram illustrates the different messages involved in the MSC model of Figure 49. In this figure, we identify seven sets of inter-component communication. These are illustrated in Figure 58 using black arrows and are labelled from 1 to 7. The messages involved in each of the communication arrows shown in this figure are described at the top right corner of the figure. The direction of the arrows illustrates the direction of the message sending. For example, arrow 2 describes the messages send by component A to component B, while arrow 3 describes the messages sent by component B to component A. The fact that there exists two arrows between a pair of components does not mean that we need to define two different contracts. We may define a single duplex contract.

FIGURE 58. Communication diagram for MSC S1



3.7.3.3 Define Role Protocol Classes

As described in section 2.4.2, ROOM protocol classes are mainly defined in terms of a set of incoming messages and a set of outgoing messages. The objective of the current step is to define a set of protocol classes that will be used to define a ROOM role structure for a given MSC.

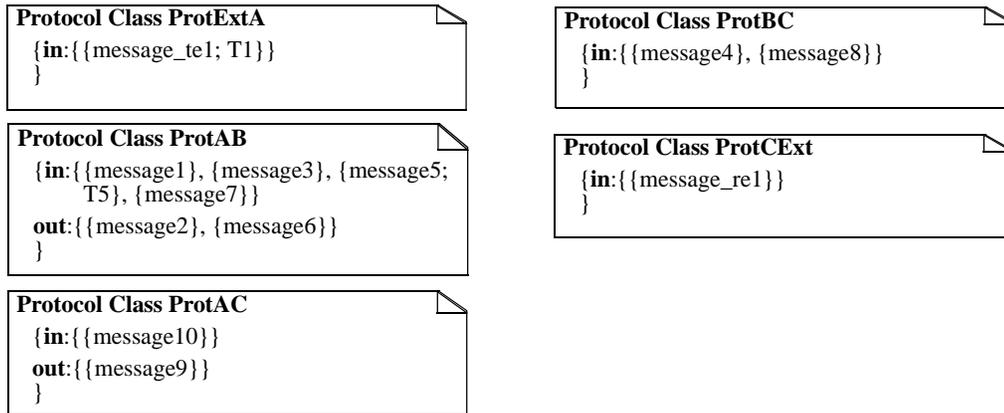
In order to define these protocol classes, the information contained in the communication diagram is used. Here, the work of the designer consists in grouping sets of related messages into protocol classes. The number of protocol classes defined and the grouping of messages into protocol classes constitute design decisions.

The protocol classes defined in this step are called *role protocol classes* because they are defined in the context of a ROOM role structure, i.e. on a per MSC basis. Later, when the global ROOM structure of the system is defined, the protocol classes defined here will be used to define the final protocol classes. These final protocol classes will be obtained by merging several role protocol classes.

In the current example, five new protocol classes are defined. These classes are given in Figure 59. In this case, we took the decision to group the messages in protocol classes in such a way that only one contract will be defined between each pair of communicating components. Also, a single protocol class will be used to define the two ports involved in a same contract, i.e. one port will be the conjugated version of the other one.

At this stage, existing protocol classes can also be used instead of defining new ones.

Also, if there are data associated with messages, the ROOM data classes are defined in this step. Once defined, the data class information is added to the messages in the protocol class. In the role protocol classes of Figure 59, the data classes T1 and T2 defined in the MSC of Figure 49 have been respectively added to messages `message_te1` and `message5`.

FIGURE 59. Definition of the role protocol classes

3.7.3.4 Definition of the Role Contracts

Once a set of role protocol classes is defined, we use these classes to define the set of contracts that are required to enable the execution of the message sequences defined in the specification MSC model. This set of contracts together with the set of components form the ROOM role structure associated with a particular MSC.

In Figure 60, a ROOM role structure is defined for the MSC of Figure 49. In this figure, the ports are labelled with identifiers of the form:

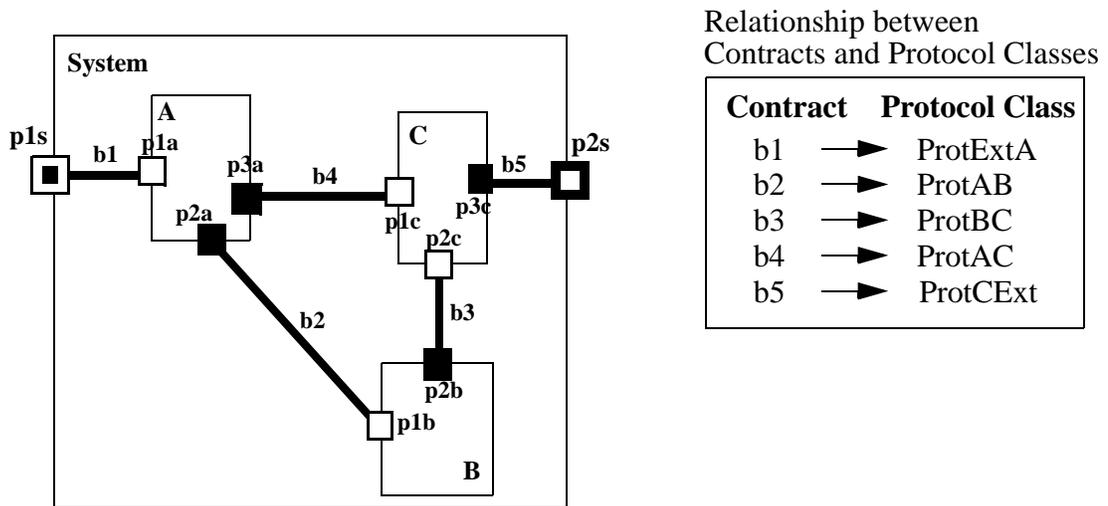
p<port_identification_number><port_owner>.

For example, port p1a is the port 1 of component A. The bindings are labelled b1, b2, b3, b4 and b5. Also, we identify five different contracts: (b1, {p1s, p1a}), (b2, {p2a, p1b}), (b3, {p2b, p2c}), (b4, {p3a, p1c}) and (b5, {p3c, p2s}). In practice, more expressive names should be used.

Since the binding identifier is unique for each structure level, we use it as contract identifier. Thus, we say that component A and component B communicate together using contract b2.

On the right side of Figure 60, a relationship between contracts and protocol classes is established. This relationship should be read as follows: the ports that are part of the contract identified at the tail of the arrow are instances of the protocol class identified at the head. For example, the relationship between b2 and ProtAB means that the two ports that compose contract b2, i.e. p2a and p1b, are both instances of the protocol class ProtAB described in Figure 59.

FIGURE 60. Definition of ROOM role structure for MSC S1



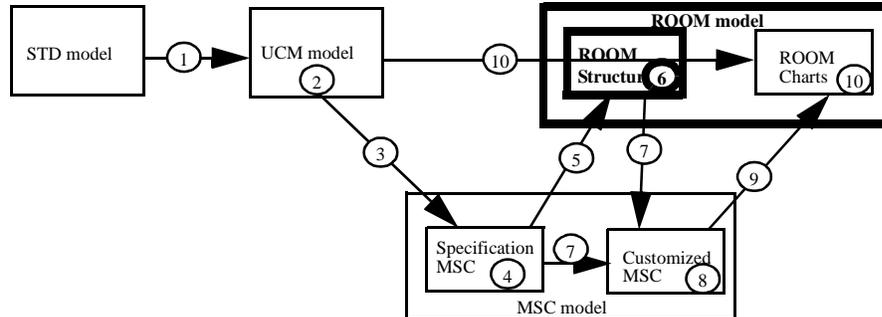
The pair of ports that are linked together in a contract, e.g. p2a and p1b in Figure 60, must be compatible, i.e. the set of outgoing messages defined in one port must be contained in the set of incoming messages of the other port. In the current example, we make sure that this rule is respected by using the same protocol class to define both ports, and by making one a conjugated port¹⁶. For example, p2a and p1b are both instances of the protocol class ProtAB and port p2a is defined as a conjugated port.

¹⁶We recall that a port p1 is a *conjugated version* of a protocol class C1 if the messages defined in port p1 are the “inverse” of the ones defined in C1, i.e. the incoming messages of p1 corresponds to the outgoing messages of C1, and the outgoing messages of p1 corresponds to the incoming messages of C1. Conjugated ports are graphically represented by white filled squares, while unconjugated ports are black filled.

3.8 ROOM Structure Modeling

In the transition from MSC to ROOM structure, a set of role structures have been defined. The objective of the current step is to define the overall ROOM structure of the system by integrating all the role structures previously defined. We call this structure the *global ROOM structure* of the system. The inputs of this modeling phase are a set of new ROOM role structures, and the existing global ROOM structure model of the system. The output is a new version of the global ROOM structure model of the system. The ROOM structure modeling phase is highlighted in Figure 61.

FIGURE 61. ROOM structure modeling



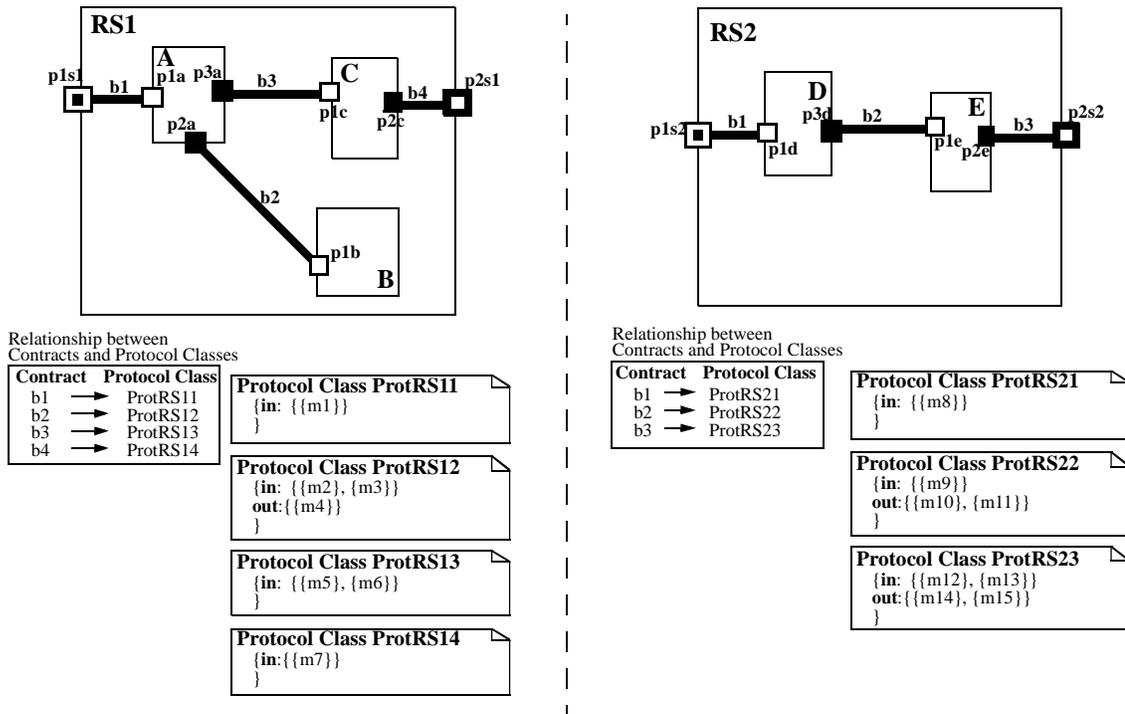
To produce the global ROOM structure, we need to define:

1. The set of system actors (the actors are defined in terms of the set of roles they play in the overall system)
2. A set of system protocol classes
3. A set of system contracts

In the following sections, we discuss each of these three steps.

To illustrate the structure integration steps, we use, as an abstract example, the two role structures given in Figure 62. In the following sections, we show how role structures RS1 and RS2 can be integrated in a global structure. A concrete example will be discussed in Chapter 5.

FIGURE 62. ROOM role structures RS1 and RS2



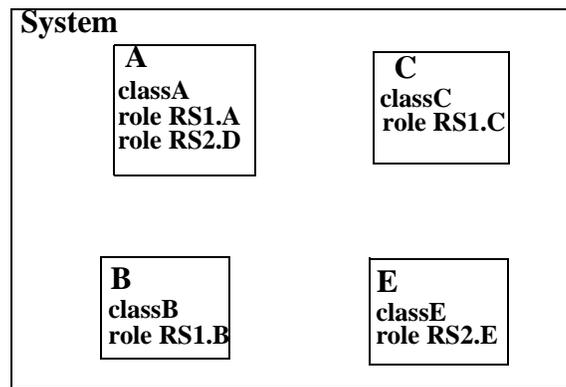
3.8.1 Definition of Final Actors

As previously discussed, the different actors defined in the role structures can be viewed, from a system point of view, as containing a set of roles that need to be filled by components in the final model of the system. In this step, we define the set of system actors and assign a set of roles to each of them. At this stage, existing actor classes may be used to fill roles, or new ones may be defined. The resulting actors constitute the main run-time components of the system.

In Figure 63, we define a set of four system actors that integrates the two role structures RS1 and RS2: actor A that plays the role of actor A in RS1 and the role of actor D in RS2, actor B that plays the role of actor B in RS1, actor C that plays the role of actor C in RS1, and actor E that plays the role of actor E in RS2.

At this stage, the class and properties (optional, structure dynamics properties) of the actors must also be specified. In Figure 63, the class of actors A, B, C, and E are respectively classA, classB, classC, and classE.

FIGURE 63. Definition of system actors



3.8.2 Definition of System Protocol Classes

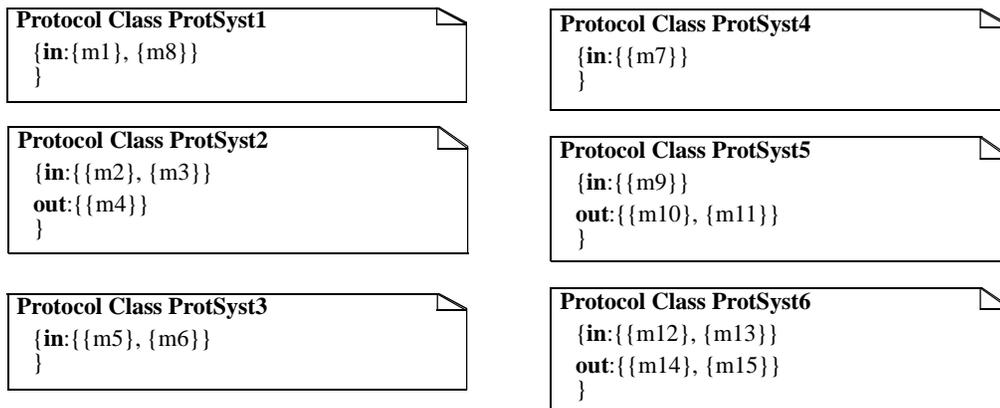
The objective of this step is to define the set of system protocol classes¹⁷. In this step, the role protocol classes defined in the previous step must be grouped into more general protocol classes. The set of role protocol classes that are relevant to a particular actor is the set of role protocol classes that are used in the different roles assigned to the actor.

¹⁷These protocol classes are called *system protocol classes* because they are the ones that appear in the global ROOM Structure model of the system.

At this stage, designers may either decide to reuse existing protocol classes, or they may define new ones. The choice of defining a new protocol class or of using an existing one constitutes a design decision.

In the context of our example, we define six new protocol classes. These protocol classes are described in Figure 64. In this example, we decided to group protocol classes ProtRS11 and ProtRS21 into a single protocol class ProtSyst1. In the case of the other system protocol classes, there exists a one-to-one relationship with the role protocol classes in the following way: ProtSyst2 corresponds to ProtRS12, ProtSyst3 corresponds to ProtRS13, ProtSyst4 corresponds to ProtRS14, ProtSyst5 corresponds to ProtRS22, and ProtSyst6 corresponds to ProtRS23.

FIGURE 64. Definition of system protocol classes



3.8.3 Definition of System Contracts

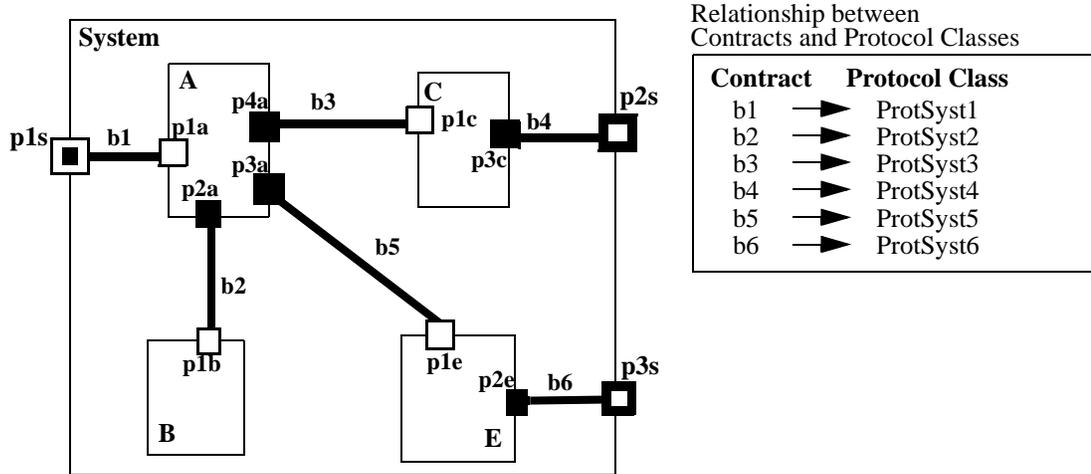
Once the sets of system actors and protocol classes are defined, the last step consists in defining the set of system contracts. To achieve this, we need to define the set of actor ports¹⁸ as instances of the system protocol classes, and link them in a way that is consis-

18. SAP-SPP inter-actor communication can also be defined as an alternative to port-binding communication (see Layer Connections in section 2.4.2 for definition of SAP-SPP communication).

tent with the linkage of the role structures. At the end of this step, the definition of the global ROOM structure of the system is completed.

In Figure 65, we define the set of contracts for our example system. As in the previous ROOM structure models, the relationship between contracts and protocol classes is described in the box beside the structure model. We observe that the resulting global ROOM structure of the system is consistent with the role structures RS1 and RS2. The messages exchanged between role actors in the role structures can also be exchanged in the global structure between the actors that play these roles. For example, in role structure RS2 messages m9, m10 and m11 can be exchanged between role actor D and E. In the global structure, these messages can also be exchanged between actor A, that plays the role of actor D in RS2 (RS2.D), and actor E, that plays the role of actor E in RS2 (RS2.E).

FIGURE 65. Definition of system contracts

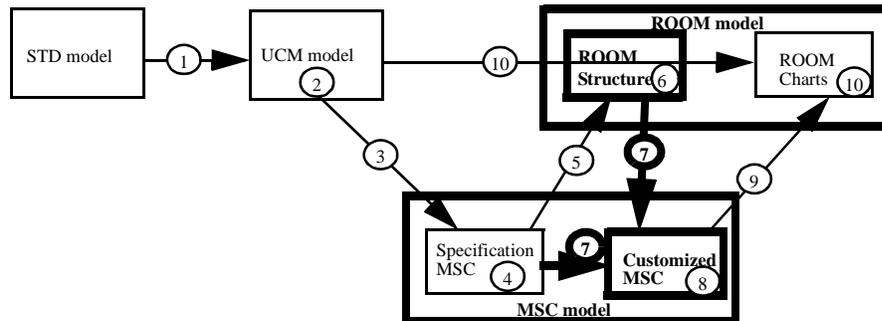


3.9 Addition of ROOM Structure Information to the MSC model

In RT-TROOP modeling, the role of the MSC modeling technique is to smooth the transition between a high-level scenario model and a detail-level ROOM model. In order to achieve this, the specification MSC model produced in the specification MSC modeling phase needs to be customized for a specific ROOM model. This customizing is done by adding ROOM structure information, and by introducing component state information to the different MSC components.

The objective of the current step is to add ROOM structure information, defined in the ROOM structure modeling phase, to the MSC model. This results in the creation of a new MSC model called the *customized MSC model*. While the specification MSC model is defined independently of any ROOM model, the customized MSC model is a customizing of the specification MSC model that is defined for a specific ROOM model. The inputs of this modeling phase are the ROOM structure model for which designer wants to customize the MSC, the specification MSC model, and the existing customized MSC model. The output is a new version of the customized MSC model that is complete with respect to ROOM structure information¹⁹. This phase is highlighted in Figure 66.

¹⁹The customized MSC model as a whole is not complete until component behavior information has been added (see section 3.10).

FIGURE 66. Customization of MSC model

The definition of the customized MSC model is conducted in two modeling steps:

1. Addition of ROOM Structure Information to the MSC Model

This modeling step consists in adding the ROOM structure information (defined in section 3.7.3) to the specification MSC model.

2. Component Behavior Modeling in MSC

This modeling step consists in introducing component behavior elements, such as component states and state transitions, in the MSC model of the system. This modeling phase is described in section 3.10.

Addition of ROOM Structure Information to the MSC Model

Linking MSC Instances to ROOM Actors

In the specification MSC model, components (or instances) are simply defined by means of an identifier. There is no relationship with concrete system components or classes. In order to customize the MSC for a specific ROOM structure, links between MSC components and ROOM actors must be defined.

Adding Port Information to Messages

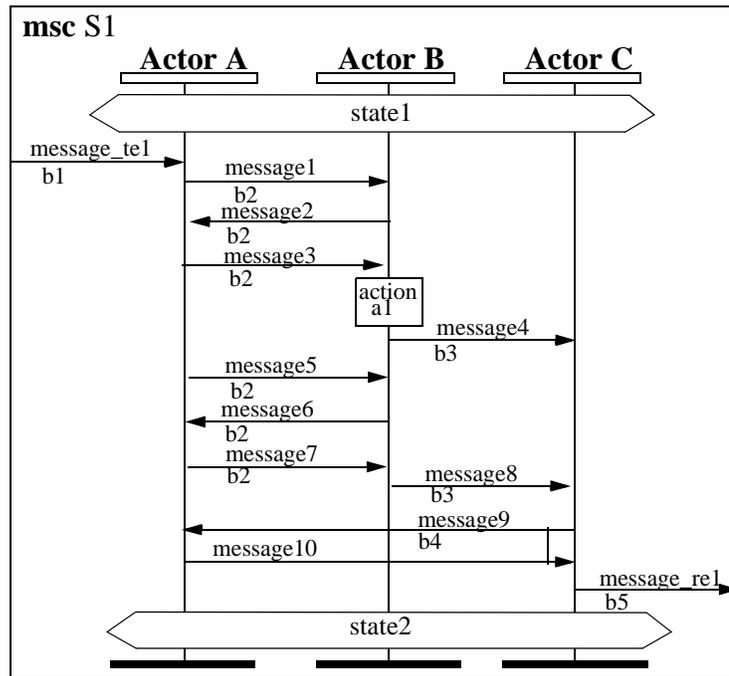
In the specification MSC model, messages are defined in terms of a message identifier and an optional message data object type. However, in ROOM, a message also needs to be specified in terms of a port (or contract) identifier by which the message is sent. Thus, we need to add contract information on each message arrow in the MSC model²⁰.

Example

In Figure 67, an example of the result of adding ROOM structure information to a MSC model is illustrated. The MSC S1 illustrated in this figure is obtained by adding ROOM structure information of Figure 60 to the specification MSC of Figure 49. In this figure, each MSC component has been linked to a specific actor in the ROOM structure: MSC component A is linked to actor A, MSC component B is linked to actor B, and MSC component C is linked to actor C. We also observe that, on each message arrow, the contract used to exchange the message is specified. In this case, the contracts correspond to the ones defined in the role structure of Figure 60. Also, we observe that the contract is not specified on the reply message arrow (`message10`) of the invoke (`message9`) call made by component C. That is because the semantics of an invoke message is such that the reply message is implicitly sent back using the same contract as the invoke message.

²⁰.As discussed in Chapter 2.4.2, contract identifiers are used to specify ports on message arrows.

FIGURE 67. Description of message sequence in MSC S1

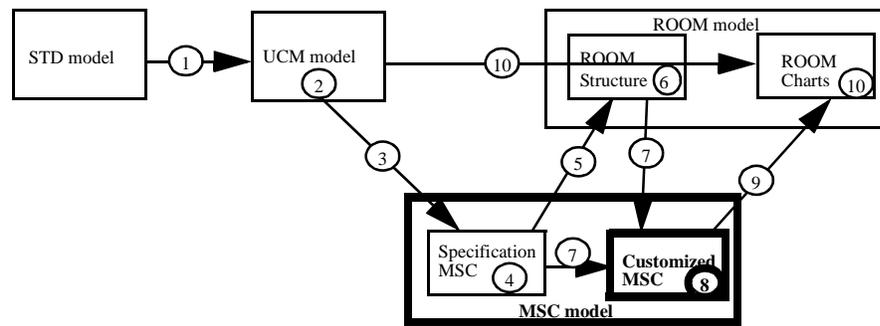


3.10 Component Behavior Modeling in MSC

The objective of this modeling phase is to introduce component behavior elements such as component states, transition actions, and entry and exit actions in the customized MSC model of the system. This way, designers can focus on the definition of component behavior on a per scenario basis before starting ROOM actor behavior modeling, in which the behavior of each actor needs to be completely defined at the detailed level. In this step, only the behavior related to the execution of a single MSC is defined. The input of this transition is the new customized MSC model (defined in section 3.9), and the output is a new version of the customized MSC model. The MSC behavior modeling phase is highlighted in section 68.

The MSC model produced at the end of this step can be used for the verification of the final ROOM model.

FIGURE 68. Component behavior modeling in MSC



Modeling Activities

In order to facilitate the definition of component behavior, the following component behavior information may be introduced in the customized MSC model.

Decomposition of System States

In the specification MSC model produced in the specification modeling phase (section 3.6), each MSC contains a set of system states. In order to make the transition to component behavior, these states need to be decomposed and expressed as a set of component states.

Definition of Component States

In order to introduce detail-level elements in the MSC model it is important to emphasize the fact that ROOM does not provide any receive statement that can be used in transition actions. An incoming message can only be handled while being in an explicit state. For this reason, a state needs to be specified before every incoming message arrow in MSCs, unless the incoming arrow is part of a call-return (invoke) message.

Transition Triggering Messages

The incoming arrow located below a state on a component timeline corresponds to the triggering message of the transition.

Transition, State Entry and State Exit Actions

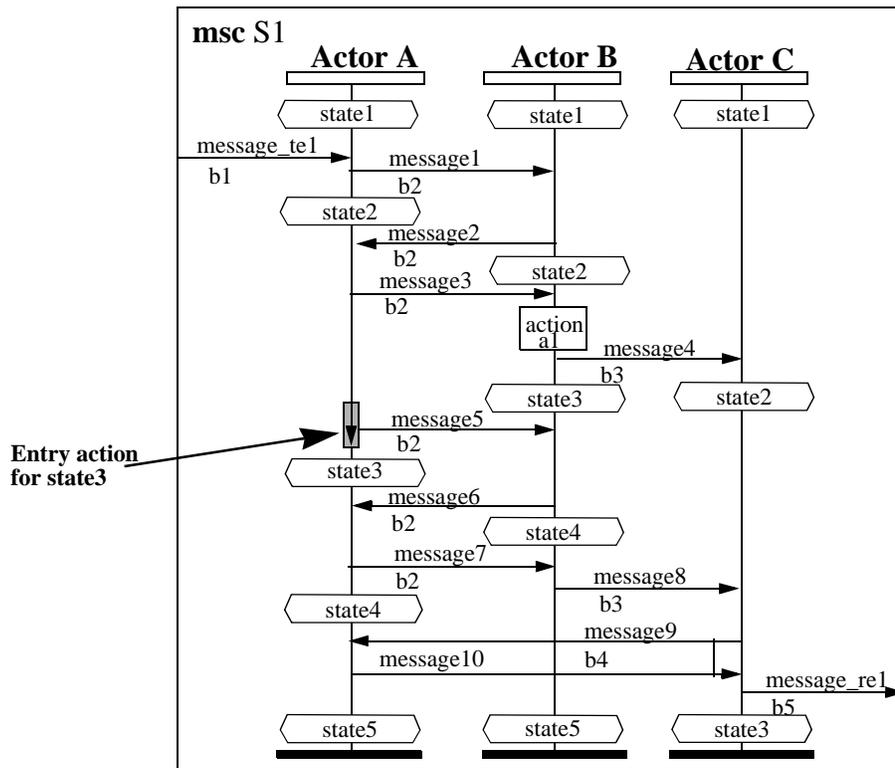
The set of outgoing messages placed between the triggering message arrow and the next state are by default considered to be part of the transition actions. If the designer wants to group some of these message arrows in state entry or exit actions, s/he has to make this decision explicit in the customized MSC by using proper notation. We identify messages that are part of a state entry actions by linking them to an entry action icon²¹ () placed on the instance axis of the message sender immediately before a state, and messages that are part of a state exit actions by linking them to an exit action icon () placed on the instance axis of the message sender immediately after a state. An example of entry action specification is given in Figure 69.

Example

An example of component behavior modeling in MSC is illustrated in Figure 69. We observe that in this customized MSC model, the information related to UCM responsibilities has been completely removed. That is because at this stage this information is no longer relevant.

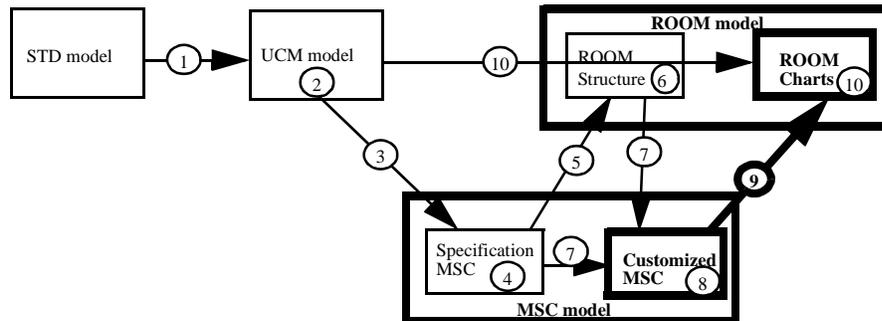
21. The MSC entry action icon and exit action icon are not part of the standard MSC notation. They are used in the RT-TROOP modeling process to specify state entry and exit actions in customized MSCs.

FIGURE 69. Description of message sequence in MSC S1



3.11 From MSC to ROOMChart

The objective of the transition from MSC to the ROOMChart is to define ROOMChart role behavior state machines on a per component/scenario basis, i.e. for each MSC, a ROOMChart role behavior is defined for each component involved in the MSC. The input to this transition is a customized MSC model that contains both structure and component behavior information, and the output is a set of ROOMChart role behaviors. The transition between customized MSC and ROOMChart is illustrated in Figure 70.

FIGURE 70. Transition from customized MSC model to ROOMChart model

In the current section, we:

- Analyze the relationship that exists between ROOMChart models and MSC models, and describe the steps used in RT-TROOP to define a ROOMChart model
- Described the steps used to define a ROOMChart role behavior model from a customized MSC model.

3.11.1 Generation of ROOMChart Models from Customized MSC model

MSC and ROOM are two modeling techniques that describe component behavior by means of state machines, i.e. in terms of component states and transitions. However, these two modeling techniques significantly differ with respect to two aspects:

- **Flat state machines vs. hierarchical state machine.** In MSC models, the description of component behavior is limited to flat state machines. Also, MSC does not explicitly define transitions; all messages and actions located between two states are implicitly considered to be part of the transition actions.

In ROOMChart models, component behavior may be structured and modeled using hierarchical state machines. This allows simplifying the design and maintenance of complex component behavior.

- **Level of detail.** In MSC, component behavior is described in terms component states, messages send, messages received, and internal actions. These model elements, which are considered atomic in MSC, are simply described by means of an identifier.

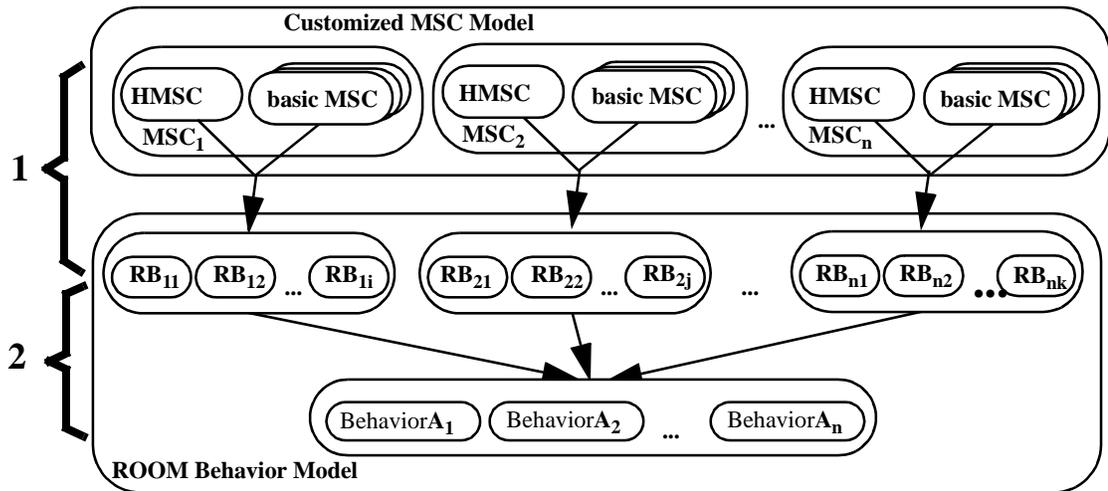
In ROOM, actor behavior modeling involves two different level of description: a high level at which actor behavior is defined in terms of states and transitions, and a detailed level at which transition actions, entry actions, exit actions, and functions are described at the level of programming languages. This allows separating the internal logics of actor behavior from the programming details.

In RT-TROOP, the design of ROOM behavior models proceeds in two steps:

1. **Definition of role behaviors.** As in the case of ROOM structure definition, when making the transition between MSC and ROOM behavior, we may start by defining behavior on a per MSC basis before working at the global level. We call the resulting behaviors *role behaviors*, because they correspond to the behavior of role actors defined in the ROOM role structures. Role behaviors are defined by means of flat state machines. This step is described in the next section (section 3.11.2).
2. **Integration of the role behaviors into component behaviors.** In order to obtain the overall ROOM model of the system, the set of role behaviors defined in the previous step must be integrated into a set of component behaviors. Component behaviors are defined by means of hierarchical state machines. This step is carried out in the ROOM-Chart modeling phase (section 3.12).

These steps are illustrated in Figure 71.

FIGURE 71. Relationship between MSC and ROOM behavior



3.11.2 Definition of Role Behaviors

In the context of RT-TROOP modeling, the objective of the role behavior modeling step is to define a ROOMChart model for each component defined in the customized MSC model.

This phase involves two different levels of modeling: the schematic (state machine) level and the code (programming language) level.

Schematic Level

At the schematic level, the definition of the ROOMChart state machine is conducted as follows:

- Each MSC condition is mapped onto a ROOMChart state.
- Each incoming MSC message is mapped onto a ROOMChart transition triggering message, unless the incoming message is part of a call-return (invoke) message in which case it corresponds to the reply message associated with the call-return.

- Each outgoing MSC message is mapped onto a ROOMChart message sending action.
- Each MSC component action is mapped onto a ROOMChart function call.
- Each message identified as an entry or exit action in a MSC is placed into a ROOMChart state entry or exit code.
- The set of outgoing messages, not already identified as state entry or exit action, and component actions located between two MSC states (conditions) are grouped into ROOMChart transitions.
- States and transitions in the ROOMChart model must be connected in conformance with the MSC model.

Code Level

In customized MSC models, transitions, entry actions, and exit actions are only specified in terms of message identifiers and action labels²². However in ROOMCharts, behavior models need to be formally described at the detail-level using programming languages²³. At this stage, designers must:

- Describe transitions, entry actions, and exit actions in code.
- Write the code in the transitions for data manipulation and implement the functions that correspond to MSC component actions.
- Define a ROOMChart function for each instance action defined in the MSC model; the function constitutes an implementation of the MSC action.
- Implement the data classes used in the definition of extended state variables or in data objects associated with messages.

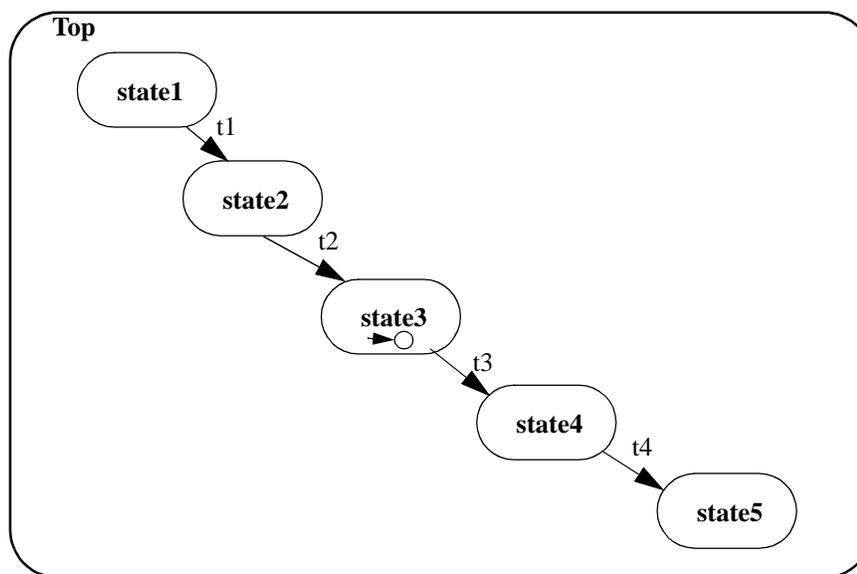
22.A textual description of actions can also be provided in MSC models.

23.ROOM allows using any programming language at the detail-level. The ObjecTime Developer toolset [68], which implements the ROOM notation, allows using either C⁺⁺ or RPL, a ROOM specific language based on a subset of Smalltalk.

Example

An example of the generation of a ROOMChart state machine from the customized MSC of Figure 69 is illustrated in Figure 72. In this figure the schematic part of the ROOMChart state machine is given. The state machine given in this figure is the one corresponding to component A in the MSC of Figure 69.

FIGURE 72. ROOMChart state machine for component A



The code of the different transitions of the state machine is defined below. The programming language used in this example is RPL.

```

ACTOR CLASS A
  BEHAVIOR{
    LANGUAGE 'RPL'
    FSM
      TRANSITIONS{
        DEFINE t1
          FROM STATE S1
          TO STATE S2
          TRIGGERS
            {DEFINE SIGNALS {message_te1} ON {protExtA};}
          ACTION
  
```

```

        {} |
        SEND protAB SIGNAL %message1
        ENDSEND};
DEFINE t2
FROM STATE S2
TO STATE S3
TRIGGERS
    {DEFINE SIGNALS {message2} ON {protAB};}
ACTION
    {} |
    SEND protAB SIGNAL %message3
    ENDSEND};
DEFINE t3
FROM STATE S3
TO STATE S4
TRIGGERS
    {DEFINE SIGNALS {message6} ON {protAB};}
ACTION
    {} |
    SEND protAB SIGNAL %message7
    ENDSEND};
DEFINE t4
FROM STATE S4
TO STATE S5
TRIGGERS
    {DEFINE SIGNALS {message9} ON {protAC};}
ACTION
    {} |
    REPLY %message10 ENDREPLY};
} /* end of transitions in top */

```

The entry code of state S3 is defined as follows.

```

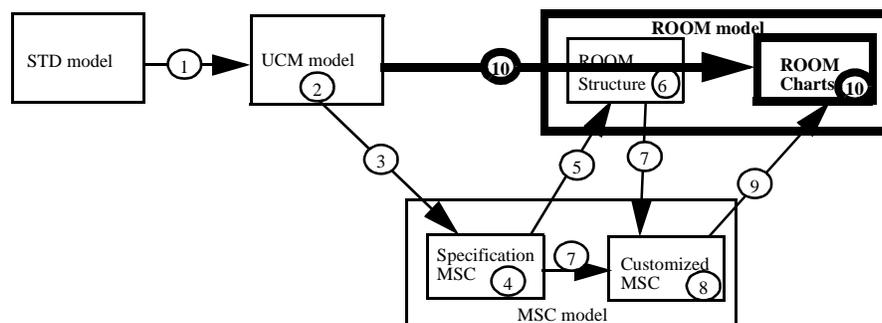
DEFINE S3
ENTRY ACTION
    {} |
    SEND protAB SIGNAL %message5
    ENDSEND}

```

3.12 ROOMChart Modeling

The objective of the ROOMChart Modeling phase is to produce a complete ROOMChart component behavior for each actor contained in the global ROOM system structure. In this phase, the set of ROOMChart role behaviors defined in the previous phase are integrated into the existing ROOMChart component behaviors. For this purpose, the scenario relationship information contained in the UCM model is used as a main input. Therefore the inputs of this phase are the existing ROOMChart component behaviors, the set of new ROOMChart role behaviors, and the UCM model. The output is a new version of the ROOMChart component behaviors that implement the set of STDs introduced at the beginning of the iteration. The ROOMChart modeling phase is highlighted in Figure 73 in the context of the overall RT-TROOP modeling diagram.

FIGURE 73. ROOMChart modeling



The definition of the ROOMChart component behaviors constitutes the end goal of the RT-TROOP modeling process.

Behavior Integration

Designing the behavior of a component that plays roles in several different scenarios constitutes a complex design task. Several important aspects of real-time systems such as scenario concurrency and interactions, performance, robustness, reuse, maintainability, and extensibility need to be considered. Moreover, when we add the set of scenarios that are required to handle aspects such as control, configuration, maintenance and robustness, the complexity of this task increases significantly. Then, even the design of a simple system becomes no longer trivial when we add all these requirements. For this reason, it becomes important to define design techniques, or patterns, that facilitate the definition of component behavior from a set of scenarios. This problem is addressed in the next chapter (Chapter 4).

In order to properly define component behaviors, it is not sufficient to understand the details of each scenario in isolation, but it is also required (crucial) to understand the relationships between scenarios. It is as important to understand scenario interactions in scenario-centric models as it is to understand component interactions in component-centric models. In both cases, the overall system behavior emerges from the interaction (or collaboration) of entities.

There currently exists, to our knowledge, no technique or pattern that allow defining component behavior from a set of concurrent and interacting scenarios in a systematic and traceable manner. In the previous sections, we defined a set of modeling phases that allow defining component behavior state machines on a per scenario basis. In this modeling phase, we need to integrate a set of scenarios (or roles) in a single component behavior.

Approach used in RT-TROOP

In RT-TROOP, we take advantage of the modeling power of ROOMCharts hierarchical state machines to structure component behavior in a hierarchical manner. This allows the

grouping of related states and transitions in higher level ones to facilitate understandability, testing, maintenance, and modification of complex behavior.

In RT-TROOP modeling, behavior integration is considered to be creative task performed by the designer. In this thesis, we propose a set of design patterns that can be used by designers.

Design patterns for behavior integration are defined in Chapter 4. A concrete example of behavior integration will be provided in Chapter 5.

3.13 Adapting the RT-TROOP Modeling Phases to Concrete Cases

In the previous sections of this chapter, the different phases of RT-TROOP modeling have been defined and illustrated in a sequential manner. However, the development of real-time systems is a complex process that can not always been carried out in such a linear and sequential manner. As mentioned in section 3.2, the ordering and the content of the RT-TROOP modeling phases may differ depending of the nature of the project (design context). The modeling phase ordering given in Figure 34 is only one possible ordering sequence, which corresponds to modeling from scratch.

The objective of this section is to show that, although presented in a sequential manner in the previous sections of this chapter, the RT-TROOP modeling process is a flexible one that can adapt to different design contexts. For this purpose, we describe three concrete cases that are often encountered in industrial system development: reuse of existing components, use of a requirement model that contains MSCs, and reuse of existing design patterns. To illustrate the reuse of an existing pattern, the mediator pattern from [35] is used.

In this section, we first discuss on general modeling phase ordering issues that apply to any system development project (section 3.13.1), and then discuss the three specific cases previously mentioned (section 3.13.2, section 3.13.3, and section 3.13.4). In each of these sections, we first describe the specific design context, and then discuss its impact on the different RT-TROOP modeling phases.

3.13.1 General Process Issues

In this section, we discuss general issues that relate to the ordering and content of the different RT-TROOP modeling phases.

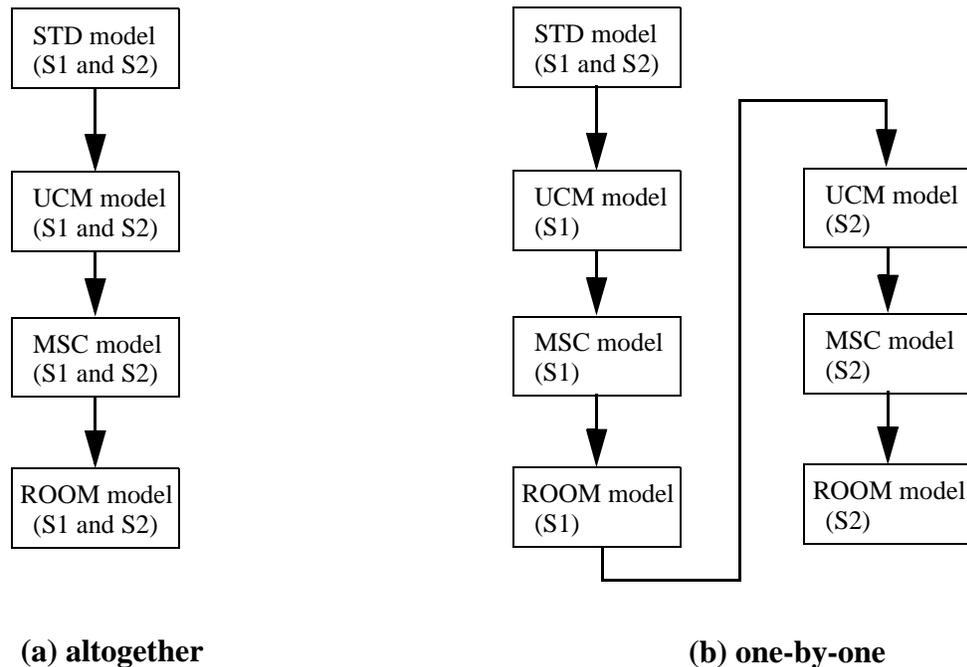
Different Ways of Addressing Sets of Scenarios

Even within an iteration, the RT-TROOP modeling process can be applied in an iterative manner. The set of scenarios that must be implemented by a designer may be addressed altogether or the scenarios may be addressed one-by-one. The two approaches are illustrated in Figure 74 with a simple example in which two scenarios, S1 and S2, must be implemented. In the first case (Figure 74 a), the designer addresses the scenarios as a set. S/he traverses the whole sequence of modeling phases considering the scenarios altogether, i.e. in each modeling phase, the designer addresses the whole set of scenarios. In this case, each of the modeling phases is executed exactly only once during an iteration.

In the second case (Figure 74 b), a designer picks one scenario from the STD model and traverses the whole modeling process before picking the next scenario. In this case, an iteration that contains n scenarios is somehow transformed into n iterations each containing a single scenario. Therefore, each of the modeling phases is executed once for each scenario.

In both cases, the resulting set of models should be the same. However, the process used to integrate them in the models is different. In the first case, scenarios are addressed simultaneously during the iteration, while in the second case, they are addressed successively.

FIGURE 74. Different ways of addressing sets of scenarios



Component Behavior Modeling in the Specification MSC Model

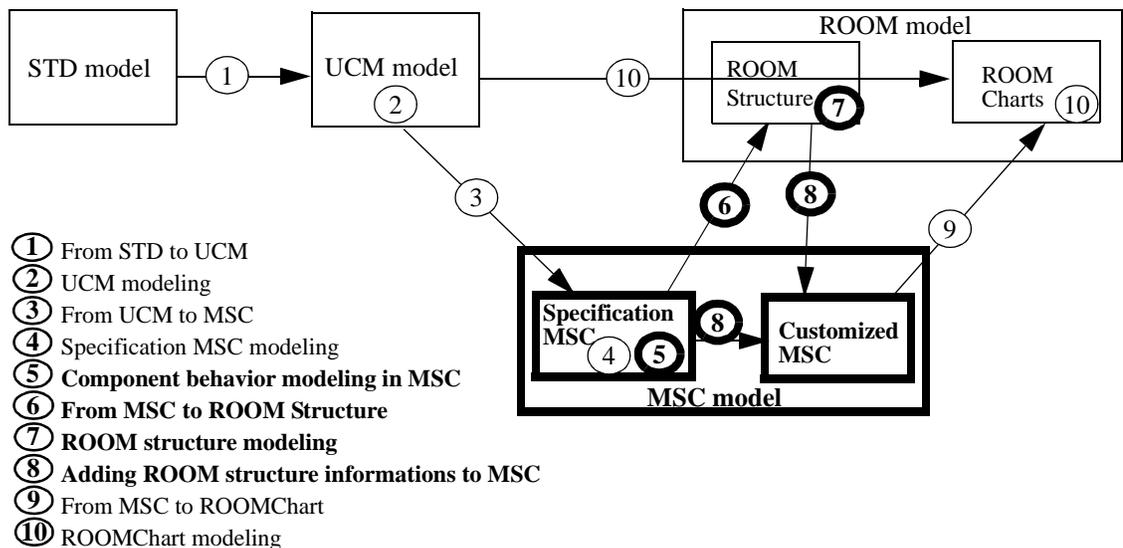
In the description of the RT-TROOP process, we placed the MSC component behavior modeling phase in the customized MSC model. The main reason for doing it this way is that component behavior information is not required until the transition from MSC to ROOMChart. So there is no need to define it before.

However, in practice, MSC component behavior modeling is very often conducted in the specification MSC model instead of in the customized MSC model. Component states are introduced in the specification MSC to describe role behavior independently of the system model in which the roles will be integrated.

The RT-TROOP modeling process makes no distinction with respect to where (and when) MSC component behavior modeling is conducted. It can be conducted either in the specification MSC model after the specification MSC modeling phase, or in the customized MSC model after the addition of ROOM structure information to the MSC model.

A modified version of the RT-TROOP modeling process in which MSC component behavior modeling is conducted in the specification MSC model is illustrated in Figure 75. The set of modeling phases illustrated in this diagram is the same as the one previously described, but the numbering (ordering) of the modeling phases is different. The MSC component behavior modeling phase is, in this case, conducted immediately after the specification MSC modeling phase. Also, the MSC behavior modeling phase is now located in the specification MSC model instead of in the customized MSC model.

FIGURE 75. Component behavior modeling in specification MSC



Using a Single MSC Model

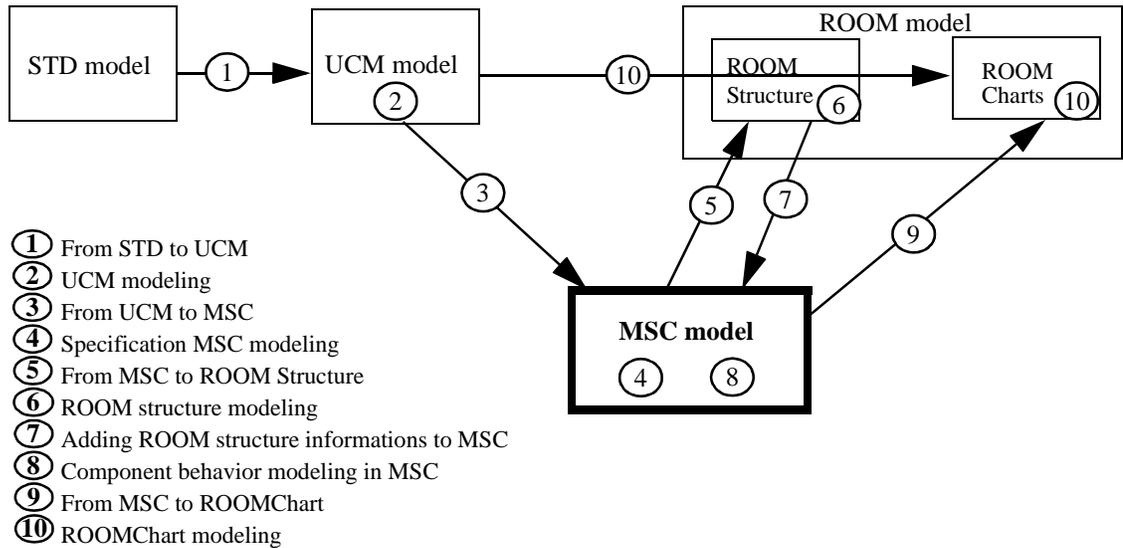
In the description of the RT-TROOP modeling process, we explicitly defined two different MSC models: the specification MSC model and the customized MSC model. This allows

to separately describe the MSC elements that can be defined independently of ROOM models and the elements that are bound to a specific ROOM model. The main motivation for separately defining specification MSCs is reuse. If separately defined, the specification MSC model can be reused with different ROOM models.

However, most of the time in practice, a single MSC model is defined. This model, which is first defined with specification MSC information (i.e. instances, messages, actions, and system states), is progressively refined during the iteration to include component behavior information and ROOM structure information.

When using the RT-TROOP modeling process, users may decide to use a single MSC model or to use both a specification MSC model and a customized MSC model. Users may even decide to separately define specification MSCs only for a subset of scenarios that they wish to reuse, and not define it for other scenarios.

A modified version of the RT-TROOP modeling process is illustrated in Figure 76. The modeling phases described in this diagram, their ordering, and their content are exactly the same as the one previously used (Figure 34). The only difference between this diagram and the one of Figure 34 is that in this case a single MSC model is defined instead of two.

FIGURE 76. Using a single MSC model

Bypassing Role Model Definition

In the previous sections, the definition of ROOM structure models and ROOMChart models are both carried out in two modeling phases. In the definition of a ROOM structure model, we first define a set of role structures in the transition between MSC and ROOM structure, and then we integrate the set of role structures in a single system structure model in the ROOM structure modeling phase. In a similar manner, the transition between MSC and ROOMChart models produces a set of role behaviors that are integrated in component behaviors in the ROOMChart modeling phase.

The main reason why we defined two distinct modeling phases (i.e. the model transition from MSC to ROOM and the in-model modeling in ROOM) to produce ROOM models is to separately discuss the issues related to the definition of role models and the ones related to role model integration. It is important that designers have a good understanding of these issues not only for producing good design models, but also for the purpose of maintaining the models.

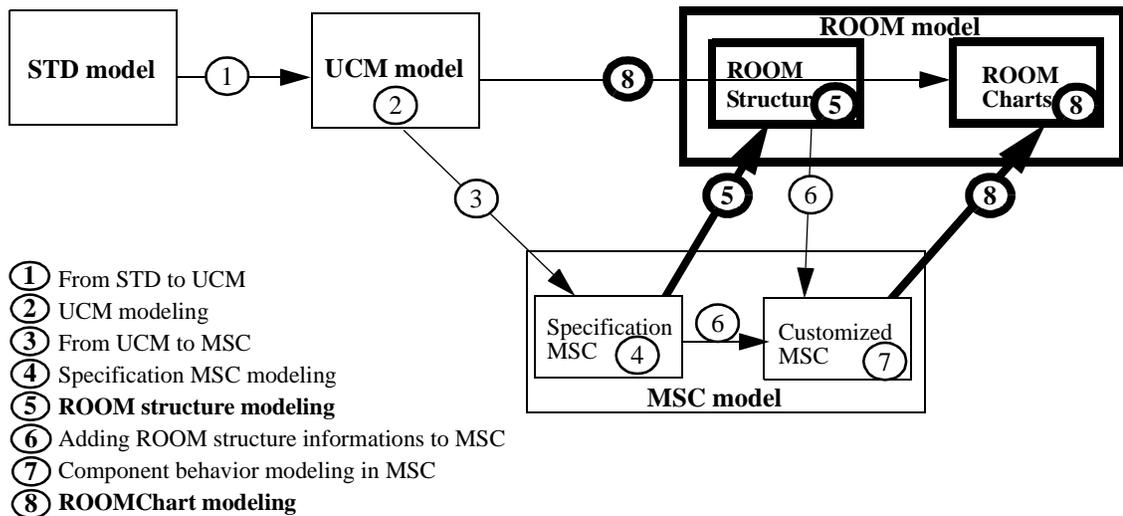
However, in practice, role structure and role behavior models are usually not explicitly defined. Designers go directly from a set of MSCs to a ROOM system model. In such a case the model transition phase from MSC to ROOM and the ROOM in-model modeling phase are combined in a single modeling phase. The fact that the two modeling phases are combined in a single one and that the role models are not explicitly defined does not affect the systematic and traceable nature of the RT-TROOP modeling process.

In some cases, it is well justified to explicitly define and maintain role models. When separately defined, role models can be reused in other systems. Role structure and role behavior models can also be maintained in association with UCM maps and MSCs to document design patterns at different levels of abstraction.

A modified version of the RT-TROOP modeling process in which role models are not defined is illustrated in Figure 77. In this diagram, a single modeling phase is used to produce a ROOM structure model and ROOMChart component behavior models. The description of the resulting modified ROOM structure modeling phase and ROOMChart modeling phase is given in Table 2. If we compare the description of the two modeling phases of Table 2 with the corresponding ones in Table 1, we observe that the inputs of the modeling phases have been modified to adapt to the new context, and that the content of the modeling phases have been modified to contain modeling activities that were previously distributed over two distinct modeling phases. The ROOM structure modeling phase of Table 2 regroups modeling activities of the transition from MSC to ROOM structure and modeling activities of the ROOM structure modeling phase of Table 1. In a similar manner, the ROOMChart modeling phase of Table 2 regroups modeling activities of the transition from MSC to ROOMChart and modeling activities of the ROOMChart modeling phase of Table 1.

TABLE 2. Modified RT-TROOP modeling phases

	Input	Objective and Activities	Output
ROOM structure modeling	<ul style="list-style-type: none"> - a new version of the specification MSC model - the existing global ROOM structure model of the system 	<p>Objective: Integrate the ROOM role structures in the global ROOM structure of the system</p> <p>Activities:</p> <ul style="list-style-type: none"> - define system actors - define system protocol classes - define actor bindings - define the type of inter-actor communication (sap-spp or ports) - define ROOM data classes for data associated with messages. - restructure system 	<ul style="list-style-type: none"> - a new version of the global ROOM structure model of the system
ROOMChart modeling	<ul style="list-style-type: none"> - a new version of the customized MSC model containing both structure and component behavior information - the UCM model - the existing ROOMChart model 	<p>Objective: Integrate MSC role behaviors in the component behaviors (one for each ROOM actor)</p> <p>Activities:</p> <ul style="list-style-type: none"> - define ROOMChart states and transitions - introduce code-level information in the ROOMChart models - define required ROOM data classes - use hierarchical state machines to structure component behavior - define entry and exit code for composite states - restructure component behavior 	<ul style="list-style-type: none"> - a new version of the ROOMChart model - a new version of the complete ROOM model (end objective of the overall modeling process)

FIGURE 77. Bypassing role model definition in the RT-TROOP modeling process

3.13.2 Reusing Existing ROOM Components

The reuse of existing components is an important aspect of system development. Some types of components like hardware component drivers, database proxies (defined for specific databases), specialized external communication components²⁴, and log manager components can be used in many different systems.

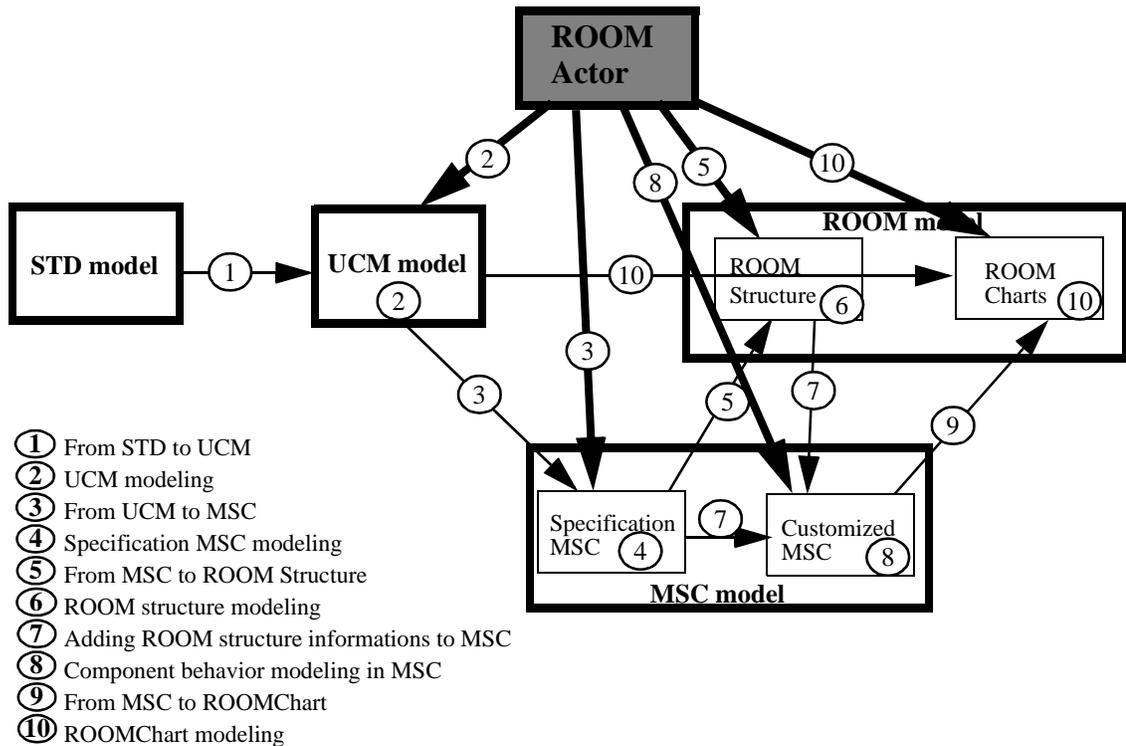
When reusing a component, the component comes with its structure, and behavior. The roles that may be played by the component, and the set of responsibilities that may be allocated to it are (implicitly or explicitly) defined in the component description.

In the context of object-oriented systems, component reuse is done at the class level. In the context of ROOM modeling, actor classes are the reused entities. The reused ROOM actor classes are used to create components (instances, or actor references) in the new system.

24. External communication components can be specialized based on the protocol they use for communication, e.g. a TCP/IP external communication component may be defined.

The decision of reusing an existing ROOM actor class has an impact in all of the different models used in the RT-TROOP modeling process. This is illustrated in Figure 78 and described thereafter.

FIGURE 78. RT-TROOP modeling using existing ROOM components



Modifications to the RT-TROOP Modeling Phases

The reuse of an existing ROOM component (actor class) in the modeling of a new system has the following impact on the modeling phases:

1. Same as before.
2. Introduce the reused component in the UCM model. This in effect constrains the set of design choices. The roles played by a reused component must be consistent with its description, and the responsibilities that are allocated to it must be part of the responsibilities it can execute.

3. Use the messages defined in the interface components (ports) of the reused component (actor class) in the description of the message sequences
4. Same as before.
5. Introduce the reused actor class, protocol classes, and data classes in the ROOM model of the new system. Use the reused ROOM component in the definition of the role structures of the system.
6. Use the reused component in the definition of the global structure of the system.
7. Same as before.
8. Introduce the state of the reused component in the different MSCs in which the component is used.
9. Same as before. However, there is no need to define the role behaviors of the reused component.
10. Use the component behavior of the reused component.

3.13.3 Using Requirement Model That Contains MSCs

In the telecommunication industry, requirements are often described by means of a set of MSCs. These MSCs are defined at an abstraction level that corresponds to the specification MSCs defined in RT-TROOP modeling. They may also contain component behavior information.

These MSCs specify:

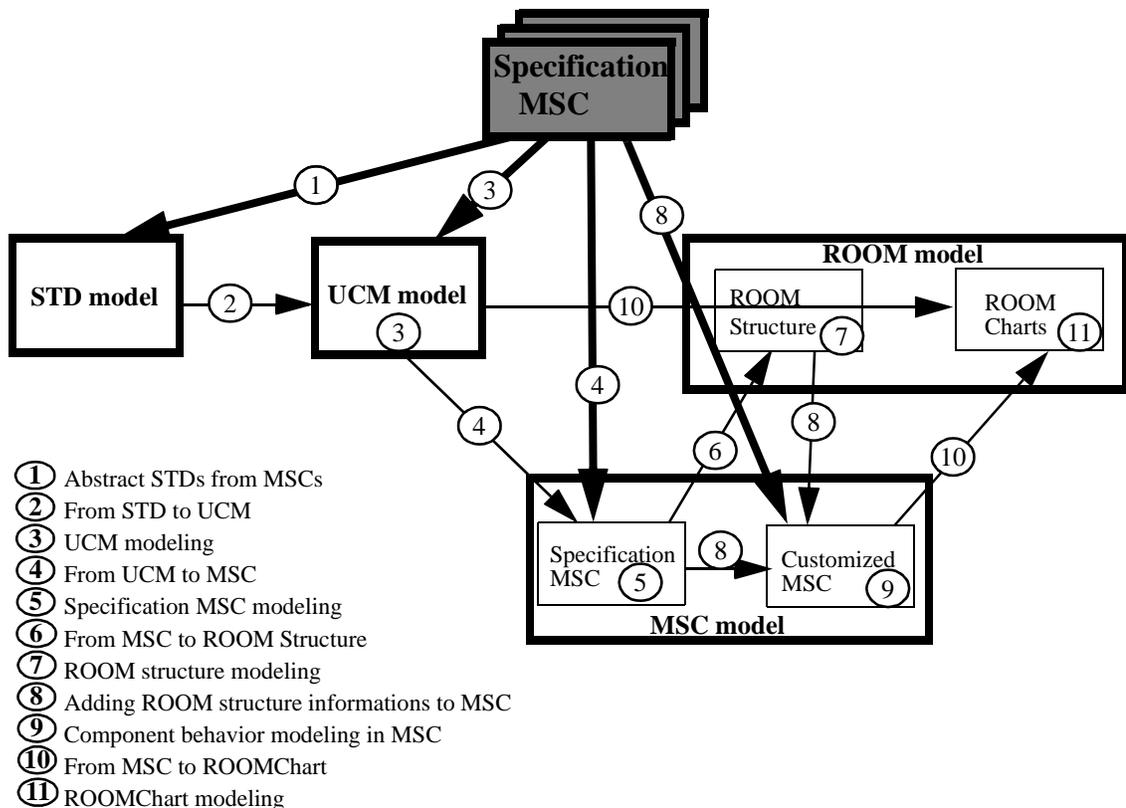
- System scenarios in terms of sequences of messages exchanged between components.
- A set of roles that must be filled in the system.

In such case, the first step does not consist in producing a UCM model from a set of scenario requirements. It may instead consist in abstracting a STD description or a UCM

model from the requirement MSCs. The designer may also decide to define the ROOM role structures associated with the requirement MSCs.

The impact of using a requirement model that contains MSCs is illustrated in the diagram of Figure 79 and described thereafter. In this diagram, we chose to define the STDs from MSCs first, and then use the same ordering of modeling phases as before.

FIGURE 79. RT-TROOP modeling using MSC requirements



Modifications to the RT-TROOP Modeling Phases

The use of a requirement model that contains MSCs has the following impact on the modeling phases:

1. Abstract STDs from the requirement MSCs.
2. Same as before. The mapping of STDs onto UCM paths does not change.

3. Define components and allocate responsibilities in conformance with the requirement MSCs.

Also, the implicit interactions between scenarios in the MSCs must be explicitly described in the UCM model.

4. Introduce the requirement MSCs in the specification model of the system, and link each of those MSCs with its associated UCM related path set (This is almost what we do in phase 1).
5. Same as before.
6. Same as before.
7. Same as before.
8. Use the component behavior information contained in the requirement MSC if any.
9. Same as before.
10. Same as before.
11. Same as before.

3.13.4 Using Existing Design Patterns (Mediator Pattern)

When applying a design pattern, part of the model elements are predefined. In this section, we discuss how the RT-TROOP modeling process adapts to the use of existing design patterns. We use the Mediator pattern defined in [35] to show the impact of the use of a design pattern on the RT-TROOP modeling phases. Different types of pattern may have a different impact on the RT-TROOP modeling process. The modification that must be made to the modeling phases depends of the nature of the design process.

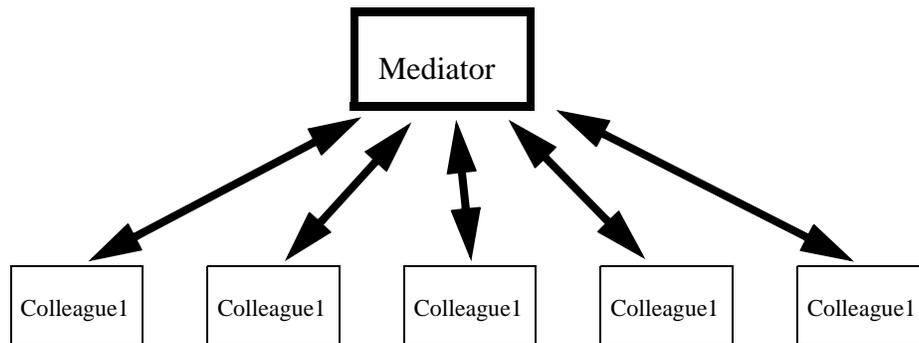
Description of the Mediator Pattern.

The mediator pattern defines two different roles for components: a *mediator* that is responsible for controlling the execution of scenarios, and a set of *colleagues* that are responsible for executing scenario responsibilities. The goal of the mediator pattern is:

- To decouple colleagues from the scenarios in which they are used.
- To centralize the control of scenario execution in the mediator.

In this context, colleagues are defined as autonomous components independently of any particular scenarios. This has the advantage of producing highly reusable components. The integration of a set of colleagues in a specific system context (i.e. to satisfy a specific set of scenarios) is done through the definition of a mediator component that is responsible for controlling (coordinating) the execution of the scenarios, and for delegating the execution of scenario responsibilities to the colleagues. All scenario responsibilities are executed by colleagues.

Structurally speaking, there is no direct relationship between the colleagues. They are completely decoupled from each other. The relationship between the mediator and the colleagues is one-to-many. The colleagues communicate with the mediator to inform it of specific events, and the mediator communicate with the colleagues to request the execution of scenario responsibilities. The structure of the mediator pattern is illustrated in Figure 80.

FIGURE 80. Structure of the mediator pattern

The mediator pattern also describes typical sequences of interactions between the mediator and the colleagues using interaction diagrams.

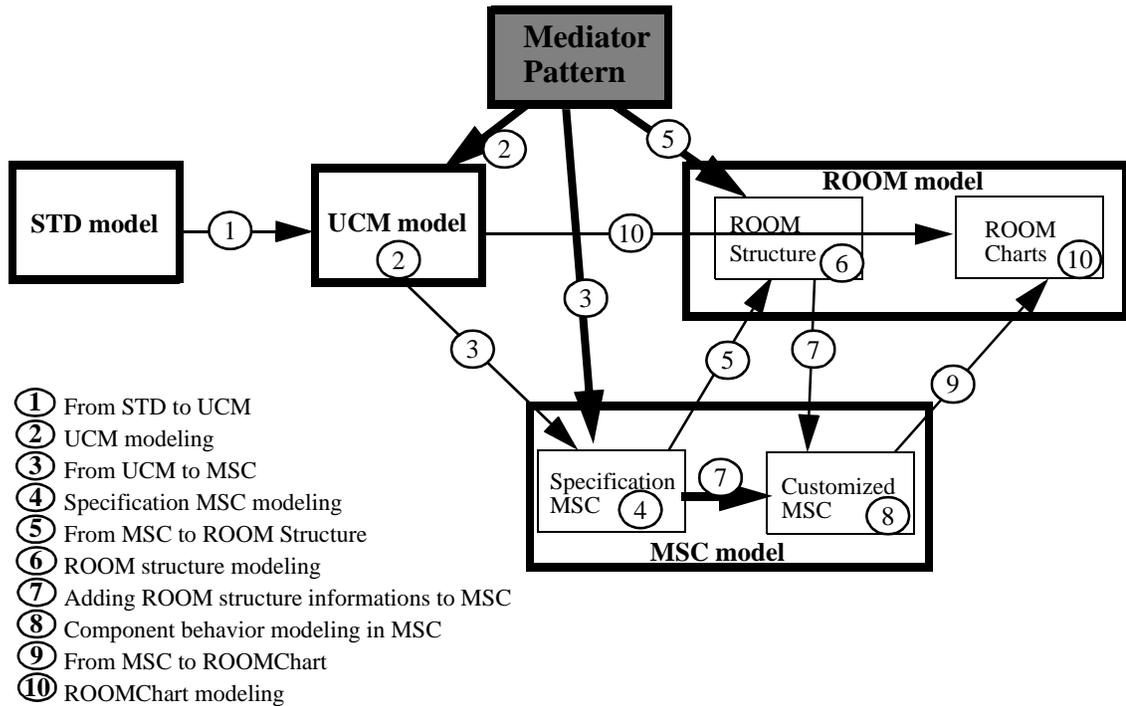
Impact of the Mediator Pattern on RT-TROOP Modeling Phases

The decision to use the mediator pattern has an important impact on the different modeling phases of the RT-TROOP process. In this case, the ordering of the modeling phases remain the same, but the inputs to and the content of the modeling phases are different.

When using the mediator pattern, the role of the colleagues may be played by reused components. However, because the behavior of a mediator component is defined in function of a specific set of scenarios and a specific set of components that are responsible for executing scenario responsibilities, the mediator component itself usually needs to be defined from scratch.

The impact of using the mediator pattern in RT-TROOP is illustrated in the diagram of Figure 81 and described thereafter. In cases where the colleague roles are played by reused components, the diagrams of Figure 78 and Figure 81 must be combined.

FIGURE 81. RT-TROOP modeling using the mediator pattern



Modifications to the RT-TROOP Modeling Phases

The use of the mediator pattern has the following impact on the RT-TROOP modeling phases:

1. Same as before.
2. Define a mediator and a set of colleague components, and allocate responsibilities to these components in conformance with their role. The mediator is responsible for controlling the execution of the different system scenarios, and to relay information to the appropriate components. The colleagues are the ones responsible for executing scenario responsibilities.
3. Define the specification MSCs in conformance with the message sequences defined in the mediator pattern.
4. Same as before.

5. Define the role structures in conformance with the mediator pattern structure.
 6. The integration of the set of roles structures in a global system structure remains the same as before.
 7. Same as before.
 8. Same as before.
 9. Same as before.
 10. Same as before.
-

3.14 Chapter Summary

In this chapter, we defined the RT-TROOP modeling process. This process allows for a systematic progression from a set of scenario textual descriptions, defined at the requirement level, to a ROOM model, from which implementation can be automatically generated. It establishes the basis for the definition of a fine-grained traceability between requirements and implementation.

The RT-TROOP modeling process makes use of four different models: STD, UCM, MSC, and ROOM. It defines the roles of the models, and the content of the modeling phases that compose it.

We defined four model transition techniques:

- STD to UCM
- UCM to MSC
- MSC to ROOM structure

- MSC to ROOMChart

While most of these transitions can be partly automated, they all require adding new design information. Thus, the process as a whole remains a creative one; it cannot be completely automated. For example, in the transition between UCM and MSC, we can automatically generate the HMSC model and the skeleton of the different basic MSC diagrams, but the sequences of actions and messages have to be defined by the designer as a refinement of the UCM high-level responsibilities. In the transition between MSC and ROOM structure, the set of messages relevant to an actor can be automatically generated from MSC, but the grouping of those messages into ROOM protocol classes has to be done by the designer. Finally, in the transition between scenario models (UCM and MSC) and ROOM hierarchical state machines, model elements such as primitive states, transition triggering events, and transition actions can be obtained directly from MSC diagrams, and some information concerning the structuring of the hierarchical state machines can be obtained from the UCM diagrams, but the definition and the overall structuring of the hierarchical state machines remains a designer task.

In Chapter 5, a simple case study is developed to illustrate the RT-TROOP modeling process with a concrete example.

The traceability relations of RT-TROOP are formalized in Chapter 6.

CHAPTER 4 Behavior Integration Patterns

In this chapter, we define a set of patterns that aims at designing the hierarchical state machines of complex components from large and complex scenario models.

The design patterns we define in this chapter are not specific to the RT-TROOP modeling process and the models it uses. They can be applied in different modeling process with different models. For this reason, we use the term *interaction diagram* in a generic manner to refer to MSC-like diagrams. We also use the term *hierarchical state machine* in a generic manner without implying any specific relationship to the ROOMChart model. The roles of interaction diagram and hierarchical state machine can be played as well by UML interaction diagram and UML hierarchical state machine.

This chapter is structured as follows.

- Section 4.1 discusses the motivations for the definition of the design patterns, and describes the characteristics of the design patterns defined in this chapter.
- Section 4.2 describes the ATM (Automatic Teller Machine) system that is used throughout the chapter to illustrate the design patterns.
- Sections 4.3 through 4.7 define a set of design patterns.

In this chapter, we have deliberately adopted the presentation format of [35] in all its idiosyncrasies and repetitions.

4.1 Introduction

Motivations

Scenario models and communicating hierarchical state machine models provide two orthogonal views of real-time systems. The former describes system behavior as sequences of responsibilities that need to be executed by components in order to achieve overall system objectives, while the later describes complete component behavior in terms of states and transitions.

One of the most crucial and complex phases of real-time system design lies in the transition that is required to go from system behavior (defined by means of scenario models) to component behavior (described by means of communicating hierarchical state machine models). Among the factors that contribute to this complexity are:

- **Large number of scenarios.** Typical real-time systems are composed of very large sets of scenarios, and each component is usually involved in the execution of many different scenarios. A component behavior needs to satisfy each of the scenarios in which it is involved.
- **Concurrency and interactions between scenarios.** Concurrency and interactions between scenarios is an aspect of real-time systems that makes such systems particularly complex and difficult to design. It is not sufficient to define component behaviors so that they can execute all the different scenarios individually; it is also necessary to define them so that they can execute scenarios in the overall context of concurrency and interactions.

- **Scenarios of different types.** Real-time systems implement scenarios of different types. Some scenarios describe normal situations, while others describe alternatives and error paths. They may also describe scenarios of different logical levels such as control, configuration, service interruption, failure, error recovery, maintenance, and main functionality. All these different types of scenarios must be considered in the definition of component behaviors.
- **Dynamic modification of scenarios.** The set of scenarios that can be executed (valid scenarios) by a system or a component can change over time; the set of scenarios that can be executed at time T1 can differ from the set of scenarios that can be executed at time T2. For example, there may exist several scenarios that aim at providing the same type of functionality in a given system, but depending on the types of components involved in the execution of the functionality, the set of scenarios that can be used may vary.
- **Unpredictability of external events.** Because real-time systems are event-driven, the order in which the scenarios can be triggered is unpredictable. Thus, when defining component behaviors, designers cannot presuppose any particular sequence of events. Component behaviors must be robust enough to allow all possible sequences.
- **Incompleteness of scenario models.** Scenario models describe typical system behavior paths at an abstract level. As such, we can say that scenario models are intentionally incomplete. Firstly, they are incomplete with respect to the overall set of system scenarios. Very often, scenario models do not explicitly describe all possible scenarios. They only describe the most important ones. Secondly, they are incomplete with respect to the overall set of requirements. Some types of requirements are not explicitly described in scenario models. For example, requirements related to reliability and robustness are usually not described in scenario models. Finally, they are also incomplete with respect to level of details that they give. Scenario models do not give the same level of details

than state machines, or implementation. Thus, when making the transition between scenarios models and hierarchical state machines, designers must add information not described in scenario models.

- **Maintainability and extensibility of components.** Since most industrial systems have a long lifecycle, it is very important to build system components so that they can be easily maintained and extended. Thus, it is not sufficient to define component behaviors that satisfy the current requirements, it is also very important to define them to facilitate future modifications. The structuring of component behavior is one of the most important factors of component maintainability and extensibility. This is an important nonfunctional requirement that must be considered by designers.

When defining the behavior of a component from a set of scenarios, all these factors must be considered. Moreover, designers must also consider other nonfunctional requirements, such as performance and robustness. Therefore, in order to achieve good design, designers must synthesize all the information contained in the scenario model, consider the nonfunctional requirements, and produce a set of hierarchical state machines that altogether satisfy the overall requirements of the system. Without a rigorous approach, this transition is error prone and very unlikely to succeed.

In the current literature, some papers, e.g. [51], [53], [58], and [69], define methods, based on synthesis algorithms, that perform automatic generation of state machines from a set of interaction diagrams. Such methods allow completely automating the transition between interaction diagrams and state machines. Their main advantage, beside the fact that they perform automatic generation of state machines, is that they allow maintaining a complete traceability between scenario models and state machine models. They also ensure the correctness of the state machines with respect to the scenarios described in the message sequence diagrams. However, they do not consider several important design issues like interactions between scenarios, and structuring of state machines to facilitate modifications and reuse. Moreover, none of the existing methods have yet solved the problem of automatically integrating concurrent and interacting scenarios in the general case.

RT-TROOP Proposed Solution

In this thesis, in order to address such issues, we take a different approach. This approach is based on the definition of design patterns. The use of design patterns ([24], [35], [67], and [96]) has rapidly increased in the industry in the last few years. The patterns approach consists in defining a set of solutions that can be applied by designers when facing specific design problems. Patterns can be classified in terms of their application domain, the aspect of system development that they address, and the level of abstraction at which they can be applied.

In terms of application domain, patterns have been defined for both general problems of any type of systems, and for problems that are specific to some application domains like real-time systems (including concurrent and distributed systems) [56], [85], [96], Corba [67], avionics [56], and so on. In terms of system development aspects, patterns have been defined to address aspects like: enterprise design [67], process and organization [96], system design [35], and software design [24], [96]. In terms of abstraction levels, patterns have been defined at the architecture and structure level [24], [35], [96], behavior level [24], [35], [96], and programming level [24], [35], [96].

However, there exist to our knowledge no patterns that address the difficult problem of integrating a set of possibly concurrent and interacting scenarios into a set of component behaviors. In this thesis, we define a set of patterns to help designers defining communicating hierarchical state machines from scenario models.

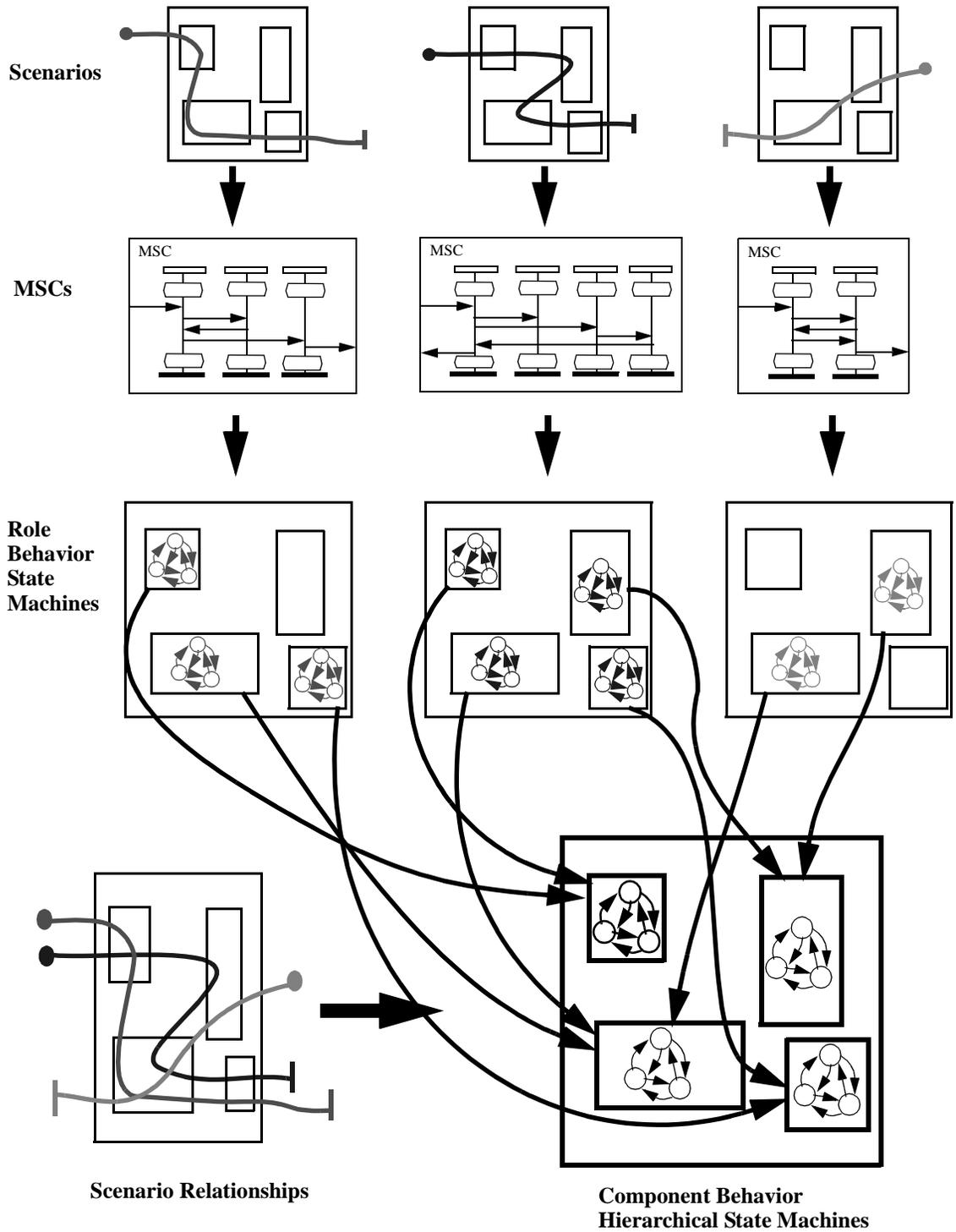
The definition of patterns to help designers making the transition between scenario models and hierarchical state machines is one of the main contributions of this thesis. Such patterns will be benefit to both experienced and inexperienced designers.

Approach

The patterns we define use both the detail-level scenario information contained in the interaction diagrams and the inter-scenario relationship information contained in the UCM models to design the hierarchical state machine of complex components.

These patterns define the behavior of components using a two-step approach. In the first step, we use the detail-level scenario descriptions provided by interaction diagrams to define state machines on a per scenario basis. In the context of the RT-TROOP modeling process, these state machines correspond to the role behavior state machines that are defined in the transition from MSC to ROOMChart (section 3.11). In the second step, we use the scenario relationship information to compose the state machines obtained in the first step into more complex hierarchical state machines. This approach is illustrated in Figure 82.

FIGURE 82. From a set of scenarios to a set of component behavior



Forces

The following forces apply to the overall set of design patterns.

- Design for locality of change. Locality of change is a property that aims at minimizing the number of places in a system where the modification of a given requirement would impact. It also aims at facilitating the precise evaluation of “where” in the system a requirement modification would impact. Locality of change is an important factor of maintainability and extensibility.
- Decouple individual scenario descriptions and scenario interaction descriptions.
- Increase traceability between hierarchical state machine models and scenario models.

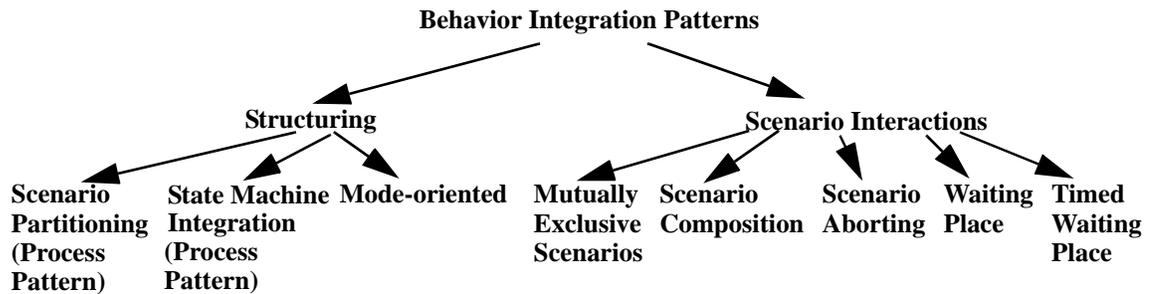
Objective

The objective of this chapter is to define a set of patterns that allows designing hierarchical state machines from scenario models. The set of patterns proposed here does not pretend to be complete in any way. It includes a set of general patterns that can be used as a starting point to define a more complete catalogue of such patterns in specific development contexts.

The set of patterns presented in this chapter provide solutions for a set of important problems that are encountered in the design of real-time system components. We hope that the definition of this set of patterns will trigger the definition of other behavior integration patterns.

This chapter proposes a set of behavior integration patterns that deals with the following issues: scenario partitioning, state machine integration, mode-oriented behavior, mutually exclusive scenarios, and different types of scenario interaction. Figure 83 gives an overview of the patterns. The two first patterns, i.e. scenario partitioning and state machine integration, are process patterns, while the others are design patterns.

FIGURE 83. Set of behavior integration patterns



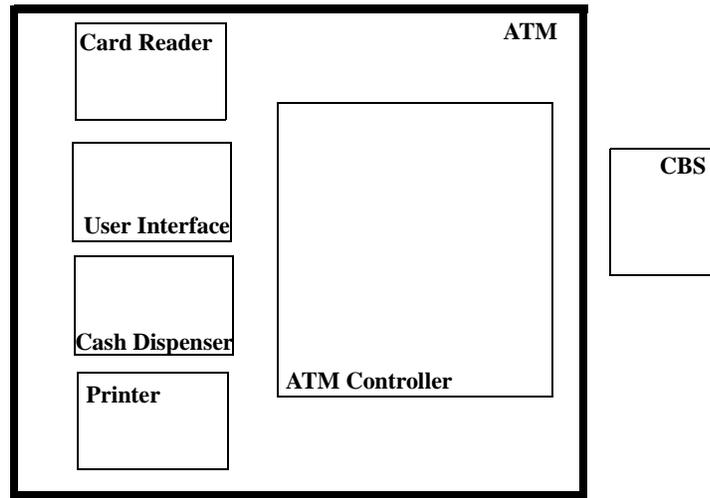
4.2 Description of the ATM system

In this chapter, an Automatic Teller Machine (ATM) system will be used to illustrate patterns. This ATM system is a conventional one that allows for withdraw, deposit, bill payment, and account update.

ATM Components

The ATM system is composed of a set of geographically distributed ATMs and a Central Bank System (CBS), which is responsible for maintaining client information and accounts; and for authorizing and registering all transactions. Each ATM is composed of a ATM controller, a card reader, a user interface (composed of a display window and a keypad), a cash dispenser, an envelop input slot (used for deposit and bill payments), and a receipt printer. The ATM controller is responsible for controlling the execution of all ATM scenarios, and for communicating with the CBS. The different components of the ATM system are illustrated in Figure 84.

FIGURE 84. Main components of the ATM system



ATM Scenario

With respect to scenarios, the ATM system contains the following scenarios:

- A start-up scenario that describes the steps required to bring the system to its operational state. These steps include the configuration of each component of the ATM system, and the establishment of the communication with the CBS.
- An abstract *Transaction* scenario, which describes the sequence of responsibilities common to all transactions. It includes reading the card number, verifying the PIN (Personal identification Number), getting the user transaction selection, executing the required transaction, printing a receipt, and returning the card.
- One scenario for each of the different types of transaction offered by the ATM system, i.e. withdraw, deposit, bill payment, and account update. Each of these scenarios describes the details of a transaction, as well as the set of alternative scenarios associated with them.
- A set of maintenance scenarios that describes different aspects of system maintenance such as: verify the states of the different ATM components, write messages to log files, trigger system alarms, and so on.

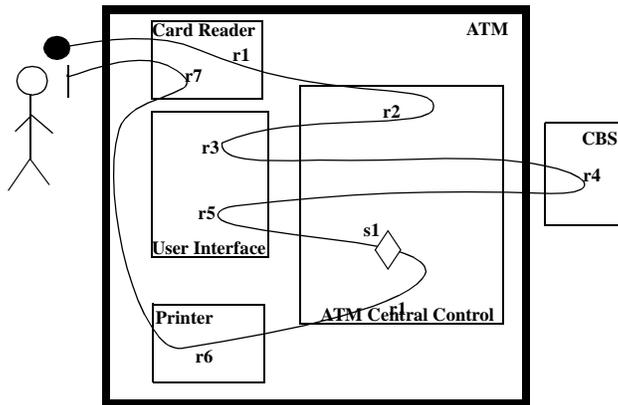
- A shutdown scenario that describes the steps to be carried out when shutting down the ATM. The shut down steps includes turning off the different ATM components, and releasing (or closing) communication with the CBS.
- A set of error handling scenarios that takes care of different types of errors that can occur in the system.
- A set of reconfiguration scenarios that allows reconfiguring different parts of the system.
- A set of secure communication scenarios that allows communicating with the Central Bank System.

UCM Model

Each of these ATM scenarios can be expressed using a separate UCM. For the purpose of the current example, we only illustrate three of them: the general transaction scenario, the withdraw scenario, and the deposit scenario.

In Figure 85, a UCM describing the general ATM transaction is given. This UCM describes the set of actions that are common to all ATM transactions. It also contains a stub, labeled S1, which is to be filled at run-time by a concrete ATM transaction UCM that corresponds to the transaction option chosen by the user. UCMs describing the concrete transactions withdraw and deposit are given respectively in Figure 86 and Figure 87.

FIGURE 85. ATM transaction UCM



Transaction scenario

Description: This scenario describes the steps that are common to all ATM transactions

Precondition: ATM is idle.

Triggering event: User inserts bank card

- steps:**
- r1. ATM swallows bank card and reads card informations.
 - r2. ATM initiates transaction.
 - r3. ATM asks user to enter PIN. Users enters PIN.
 - r4. CBS validates PIN.
 - r5. ATM asks user to choose a transaction option. User enters option.
 - s1. ATM executes the choosen transaction.
 - r6. ATM prints a transaction receipts. User picks up the receipt.
 - r7. ATM returns bank card. User picks up the card.

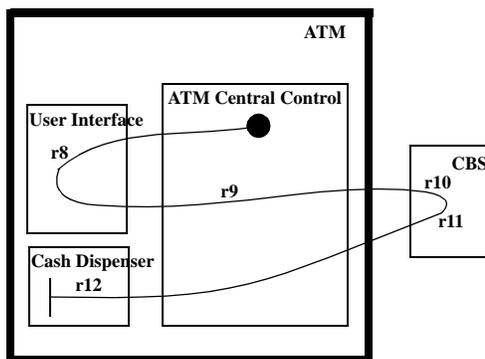
Resulting event: ATM returns the bank card.

Postcondition: ATM returns to idle.

Alternatives:

Nonfunctional Requirement: There could not be more than one active transaction scenario at the time.

FIGURE 86. UCM for a withdraw transaction



Withdraw transaction

Description: Describes the steps of a normal withdraw transaction

Precondition: ATM is waiting for an option to be entered

Triggering event: User chooses the withdraw option

- steps:**
- r8. ATM asks for amount to withdraw. User enters amount
 - r9. ATM sends a withdraw transaction request to CBS.
 - r10. CBS verifies that the user account balance is sufficient to cover the requested amount.
 - r11. CBS registers the withdraw transaction.
 - r12. ATM dispenses cash. User picks up cash.

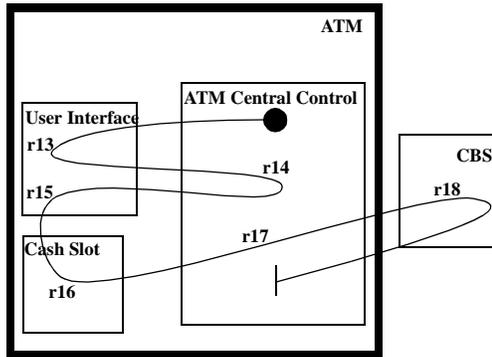
Resulting event: ATM dispenses cash

Postcondition: withdraw completed

Alternatives: Insufficient funds

Nonfunctional Requirements:

FIGURE 87. UCM for a deposit transaction

**Deposit transaction****Description:****Precondition:** ATM is waiting for option to be entered**Triggering event:** User chooses the deposit option**steps:**

r13.ATM asks for amount to deposit. User enters amount

r14.ATM initiates a deposit transaction with CBS.

r15.ATM asks for deposit envelop.

r16.ATM swallows the deposit envelop. User enters deposit envelop.

r17.ATM sends a deposit transaction request to CBS.

r18.CBS registers the deposit transaction.

Resulting event: ATM dispenses cash**Postcondition:** deposit completed**Alternatives:****Nonfunctional Requirements:**

4.3 Scenario Partitioning Pattern

The Scenario Partitioning Pattern is a process pattern that defines a general approach to hierarchical state machine structuring. It aims at partitioning the overall set of scenarios that must be implemented in a component behavior into subsets, called *scenario clusters*, that relate to different aspects of the system. Each scenario cluster can then be addressed separately at the state machine design level. A role behavior state machine can be produced for each scenario using the RT-TROOP modeling process. The global component behavior is then obtained by integrating the different role behavior state machines in a hierarchical state machine.

Motivation

Real-time system components are complex design artifact that may be involved in the execution of large sets of scenarios. For example, industrial system components must usually implement a set of scenarios that relate to different system aspects such as control, configuration, shutdown, maintenance, and normal operation. Also, each of these groups of scenarios may be further decomposed into smaller sets of related scenarios. For example, a subset of the control scenarios may be related to system restart (with different types of restart), while another subset may be relate to error handling¹, and yet another may be related to preparing the system for reconfiguration. Also, the set of configuration scenarios may be composed of several distinct subsets of scenarios each related to a different type of system configuration, and the set of normal operation scenarios may be composed of different types of system functionality. Moreover, system components may require the use of specific communication scenarios to communicate with other components using specific communication protocols.

As a result, even components that are apparently simple, become rather complex to design when placed in an industrial context that requires for robustness, high maintainability and extensibility.

The ATM system described in section 4.2 is a concrete example of such a system.

Applicability

Use the Scenario Partitioning Pattern to partition scenarios when a component behavior must implement a large set scenarios. This pattern can be used for any type of systems; it is not specific to real-time systems (other proposed patterns are more specific to real-time systems).

1. In error handling, we include both error detection and error recovery.

Problem

Real-time system components are difficult to design, maintain, and extend. In general, the complexity of performing those tasks increases exponentially with the number of scenarios. Designing the behavior of a component in the context of a single scenario is an easy task, however designing the behavior of a component that plays roles in many different scenarios is a much more complex task.

When designing such systems, designers must establish a strategy that allows dealing with a large number of scenarios, and with the complex relationships that exist between these scenarios. Experienced designers have learned to deal with these issues, but inexperienced designers have a lot of problems dealing with them.

Also, if the approach taken by designers is not somehow standardized, different designers will deal with it in a different manner, which will result in a lack of consistency in the structuring of the state machines. This can significantly increase the cost of maintenance and extensions.

Large monolithic sets of scenarios also constitute a problem at the requirement level where they may be very difficult to use and maintain. The search for a specific scenario in a requirement model that contains hundreds of scenarios may be laborious. The partitioning of those scenarios into well-defined subsets, or clusters, can significantly facilitate their use and maintenance.

Forces

- Partition the overall set of system scenario in clusters of logically related scenarios.
- Reduce the number of scenarios that have to be addressed at once at the design level.
- Facilitate the use and maintenance of large sets of scenarios.

- Increase system maintainability and extensibility by grouping scenarios that are related to the same system aspect in a single scenario cluster.

Solution

The scenario partitioning pattern uses the divide-and-conquer approach to reduce the complexity of designing the behavior of components involved in large sets of scenarios. It aims at partitioning the overall set of scenarios into scenario clusters that each relate to a different aspect of the system. These scenario clusters can then be addressed separately in the design process.

This pattern exploits the hierarchical nature of hierarchical state machines to facilitate the design, maintenance, and modification of component behavior. It allows structuring the hierarchical state machine based on the different clusters of scenarios that the component must satisfy, and on the relationships that exists between the scenarios.

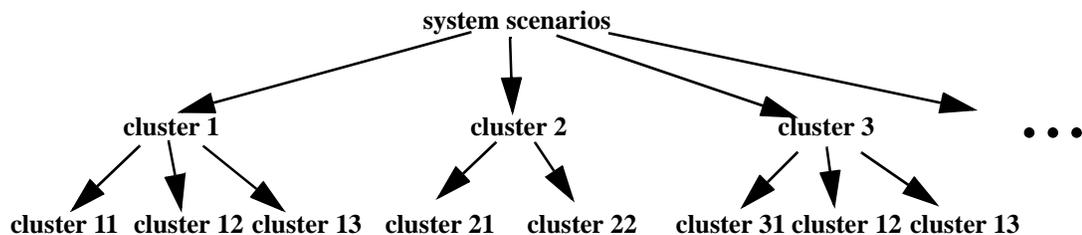
The application of this pattern is conducted in four steps.

1. Analyze the overall set of scenarios. At this stage, it is very important not only to understand each scenario in isolation, but also to understand the relationships between the scenarios.
2. Partition the overall set of scenarios into clusters of logically related scenarios. The process of partitioning scenarios into clusters is hierarchical in nature. Scenario partitioning should be recursively applied until the desired level of granularity is reached.
3. Implement each of the individual scenarios by means of a separate set of role behavior state machines. These role behavior state machines can be defined as described in the RT-TROOP modeling process.
4. Integrate the state machines defined in step 3 in a single hierarchical state machine. The integration of state machines can be done using the state machine integration pattern defined in the next section (section 4.4).

The key for success in applying this pattern lies in the understanding of the relationships that exist between the different scenarios. A lack of understanding of these relationships results in weak partitioning of scenarios. This eventually results in systems that become increasingly difficult to design as scenarios are added. On the other hand, a good understanding of scenario relationships allows for efficient planning of the system design.

In order to emphasize the hierarchical nature of the scenario partitioning process, we use a tree representation to illustrate the relationship between the different clusters. An example of such representation is given in Figure 88. The relationship expressed between a node and its children in this tree representation is of the type “composed of”. We say that the set of system scenario is composed of cluster1, cluster2, cluster3, and so on. Also, cluster1 is recursively composed of cluster11, cluster12, and cluster13. The partitioning process is a completely recursive process.

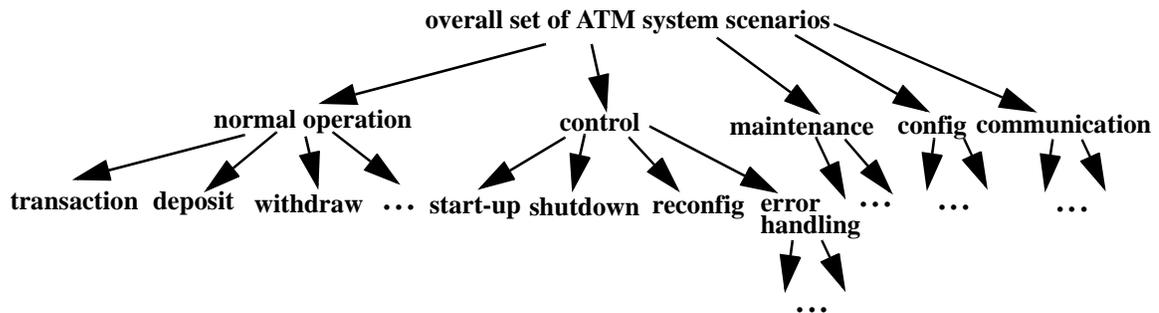
FIGURE 88. Scenario partitioning representation



Example

The set of ATM scenarios enumerated in section 4.2 (ATM Scenario page 185) contains scenarios related to different aspects of the system. After analyzing the overall set of ATM scenarios, scenarios have been partitioned into four clusters: normal operation, control, maintenance, configuration, and secure communication. The result of the partitioning is given in Figure 89. Because of a lack of space, only part of the scenarios are shown in this figure.

FIGURE 89. Partitioning of the ATM scenarios



The scenarios located in the leaves of the tree can be implemented in role behavior state machines using the described in the RT-TROOP modeling process.

Consequences

- The scenario partitioning pattern facilitates complex component behavior design, maintenance, and extensibility, by partitioning the overall set of scenarios into clusters and by addressing each cluster separately at the state machine design level.
- The scenario partitioning pattern allows decoupling the different logical levels of the component behavior. For example, the control level can be decoupled from the operational level, and the different system functionalities can be decoupled from each other.
- When used at the system level, the result of scenario partitioning may be used as an input for the planning of the different iterations. Each iteration can be defined in terms of scenario clusters resulting from the application of the scenario partitioning pattern. An iteration may contain part of cluster, or one or more clusters depending of the size and complexity of the clusters, and on the resource available in the development team
- It allows establishing a strong traceability between the structure of a hierarchical state machine and a scenario model.

4.4 State Machine Integration Pattern

The State Machine Integration Pattern is a process pattern that defines a compositional approach to hierarchical state machine design. It allows designing a component behavior hierarchical state machine from a set of simpler state machines. The main issue that need to be resolved in this pattern is the one of integrating a set of state machines that may define different aspects of a component behavior in a single hierarchical state machines. Understanding scenario relationships is a key issue in applying the State Machine Integration Pattern.

Motivation

New scenarios constantly need to be integrated in existing systems to satisfy new requirements. The integration of a new scenario in a system results in the integration of a new set of role behavior state machines in the system, i.e. the component behavior of several component will need to be modified to allow new sequences of actions that are defined by means of role behavior state machines. For this reason, state machine integration constitutes a main issue in real-time system design.

The Scenario Partitioning Pattern aims at producing a set of hierarchical state machines that each encapsulates a cluster of system scenarios. These hierarchical state machine are completely decoupled (independent) from each other. This means that under certain assumptions we could replace the internal state machine of a composite state by another state machine without affecting (modifying) the rest of the overall state machine. What the Scenario Partitioning Pattern did not discuss is how to compose a set of hierarchical state machines in a single component behavior. The main issue that needs to be resolved at this

level lies in the relationship that exists between the different scenarios associated with the state machines.

Problem

Integrating a set of new role behavior state machines in an existing component behavior is a difficult task. Often, the structure of component behavior hierarchical state machines is not strong enough to adapt to the integration of new scenarios. If the component behavior is not properly structured, the integration of a new scenario may require a major restructuring of the component hierarchical state machine. The cost of major restructuring is very expensive in terms of both time and errors introduced. When restructuring a state machine, designers must ensure that the resulting component behavior can still correctly execute all the scenarios already integrated in the component. Unless a systematic approach is used, major restructuring usually involves major debugging.

Problems in hierarchical state machine structures often result from a lack of understanding of scenario relationships, or a lack of a systematic approach in expressing those scenario relationships in terms of state machine constructs (structure). The integration of a new state machine by a designer that does not have a good understanding of the overall system behavior results in a component behavior state machine that is difficult to maintain and extend.

If properly structured, the impact of integrating a new scenario is limited to a well-defined subset of the overall component behavior hierarchical state machine.

Applicability

The State Machine Integration Pattern can be used for the design of hierarchical state machines in any type of components that is composed of a set of existing state machines.

Forces

- Allow for the design of component behavior from existing state machines.
- Structure hierarchical state machines (component behavior) so that it allows for the integration of new role behavior state machines at minimal cost.
- The integration must be done in scaleable manner, i.e. the cost of maintaining the component and the cost of adding new scenarios (role behavior state machines) should not increase exponentially as the component grows.
- High cohesion of scenarios contained in a composite state. Thus the locality of change property is satisfied.
- Allow for reuse of existing state machines.
- Maintain traceability between state machine structure and scenario model.
- Increase component behavior maintainability and extensibility by structuring state hierarchical state machines in a way that is consistent with the scenario partitioning.

Solution

The State Machine Integration Pattern defines a compositional approach to component behavior hierarchical state machine design. It defines a component behavior as a set of integrated simpler state machines each of which being associated with a set of scenarios.

The approach taken in this pattern is similar to the approach used in system structure design where a system is defined as a set of communicating components. In the case of system structure, the overall behavior of the system is the result of component integration. In this pattern, a hierarchical state machine is defined as a set of simpler state machines, where each state machine implements a set of scenarios. In this case, the overall behavior of a component is the result of state machine integration.

The starting point of the application of this pattern is a set of role behavior state machines, each of which being associated with a specific scenario², that must be integrated in an existing component behavior hierarchical state machine. The top level of the component behavior hierarchical state machine is normally occupied by the control state machine. Then, the other state machines, like the normal operation state machine or the configuration state machine, are integrated in the appropriate composite state of the control state machine. The state machine integration pattern is completely recursive.

The guiding principle behind this pattern consists in separating the two important aspects of scenario models: individual scenario description, and inter-scenario relationships. We distinguish three important types of inter-scenario relationships that must be considered when integrating scenarios in a single component:

- *Scenario interaction* relationship. This type of relationship is the strongest of the three from a semantic viewpoint. It exists between scenarios that interact in a specific manner. Different types of interactions are used in real-time systems, e.g. one scenario may *excludes*, *waits for*, *aborts*, *rendezvous* or *joins* another. These specific interactions can be captured in UCMs using the path interaction notation (section 2.2.6). We suggest that the exact interaction relationship between two scenarios determines how these scenarios are to be integrated into a hierarchical state machine.
- *Scenario dependency* relationship. A scenario dependency relationship exists between a scenario S1 and a scenario S2, if scenario S2 is used in the description of S1. Examples of this type of relationship include stubs in UCM, and the “uses” and “refines” relationships defined by Jacobson in [47] and described in [86]. In the ATM system,

2. Each role behavior state machine is associated with (is an implementation of) a specific scenario in the high-level scenario model (which is the UCM model in the RT-TROOP modeling process). Therefore, there exists a one-to-one relationship between a role behavior state machine and a scenario. In the description of the current pattern, we may use the terms *role behavior state machine* or *scenario* depending on the aspect we want to emphasize.

such a relationship exists between the abstract *Transaction* scenario and each of the scenarios that correspond a specific transaction (i.e. withdraw, deposit, bill payment, and account update).

- *Scenario clustering* relationship. This type of relationship is used to capture the coexistence of two or more scenarios inside a same conceptual regrouping called a *cluster*. This regrouping corresponds to a specific aspect of the system. At this point in time, aspects that we have observed to lead to such regrouping include control, configuration, communication, error recovery, normal operation, etc. For example, the **startUp** and **shutdown** scenarios are both part of the control cluster of the ATM system, and the **deposit** and **withdraw** scenarios are both part of the operational cluster (see Figure 89).

In the State Machine Integration Pattern, the structure of the component behavior hierarchical state machine is defined so that it reflects the relationships between scenarios.

The state machine integration pattern can be described as a three step process:

1. Analyze the relationships between the scenarios (role behavior state machines) that are to be integrated and the ones already implemented in the system.

The use of the UCM model is particularly useful at this stage to understand scenario interactions.

2. Determine where in the component behavior hierarchical state machine, each of the new scenarios (role behavior state machines) must be integrated.
3. Integrate each of the role behavior state machines (associated with new scenarios) in the appropriate state of the existing component behavior state machine. This step is repeated until all role behavior state machines are integrated. Some restructuring may be done after each integration. Restructuring may include grouping a set of states into a composite state, defining state entry or exit actions, distributing the set of actions executed on a transition over a set of transition segments, etc.

Different types of scenario relationships lead to different types of state machine integration. For example, if two scenarios are contained within the same scenario cluster, and the scenario cluster constitute a logical level of the component behavior, then the two role behavior state machines associated with the scenarios will be integrated in the same composite state. The way the integration is done depends on the specific type of interaction that exists between the two scenarios. A set of patterns dealing with different types of scenario interaction is defined in section 4.6 (Mutually Exclusive Scenario Pattern) and section 4.7 (Scenario Interaction Patterns).

Example

We now apply the State Machine Integration Pattern to the design of the ATM controller component. This component is responsible for controlling all aspects of the ATM. It coordinates the work of the different ATM components, and communicates with the CBS to carry out transactions.

For the purpose of illustrating the state machine integration pattern, we first consider the two main control scenarios, startup and shutdown, and the set of transaction scenarios. Then, we discuss the integration of error handling scenarios in the resulting component behavior state machine of the ATM controller. The integration of the other scenarios can be carried out in a similar manner.

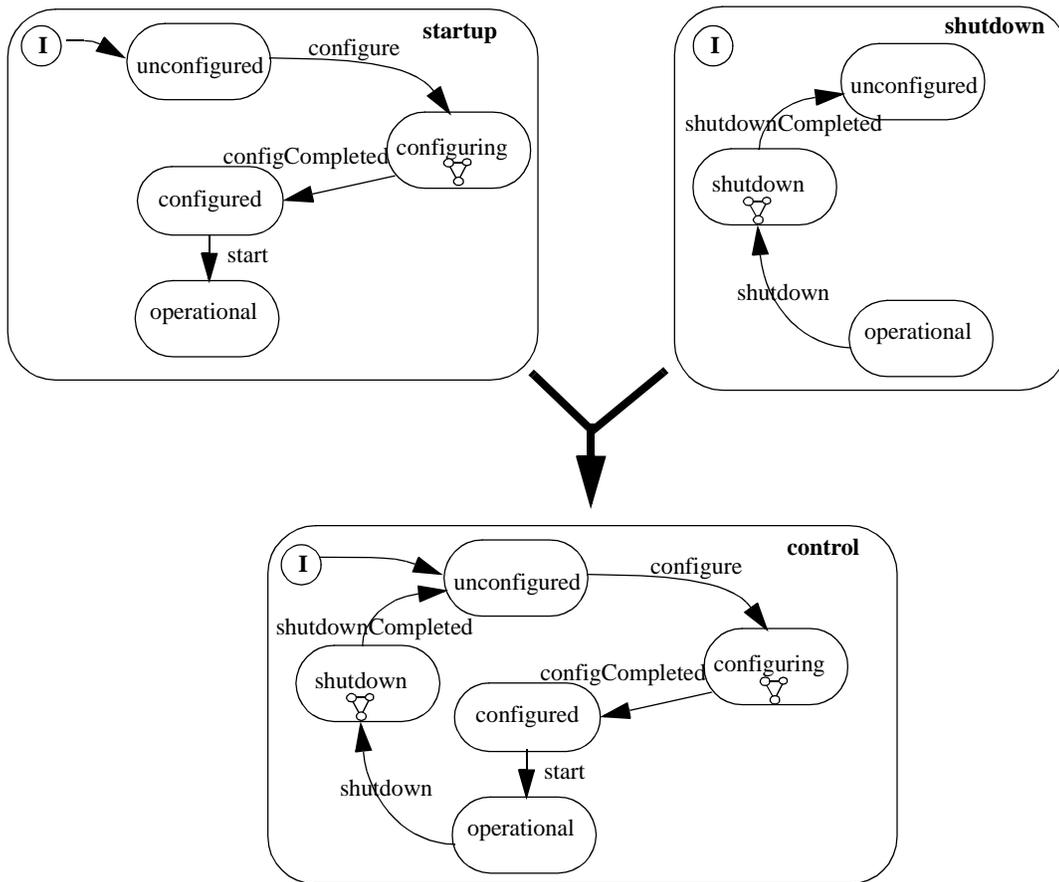
The design of the component behavior of the ATM controller is conducted in three phases:

1. Definition of the control state machine
2. Definition of the operational state machine
3. Integration of the two state machines in a single one

Integration of Control Scenarios

First, we integrate the two role behavior state machines associated with the control scenarios: the **startup** state machine and the **shutdown** state machine. These state machines are illustrated in the top part of Figure 90. Both of these state machines are hierarchical state machines. In the startup state machine, the details of the startup configuration are encapsulated in the configuring composite state. In the shutdown scenario, the different steps of the scenario are encapsulated in the composite shutdown state.

In this case, the integration of the state machines is rather simple since the end state of one scenario is the initial state of the other scenario. The result of the integration of the two state machines is illustrated in the bottom part of Figure 90. The resulting state machine is a very general one that could be used for different types of systems.

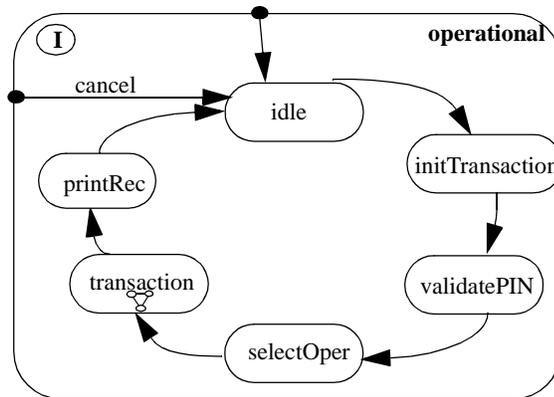
FIGURE 90. Integration of control scenarios

ATM Transactions

Second, we build a state machine for the operational aspect of the ATM controller. This state machine encapsulates all the ATM transaction scenarios. In this case, the general transaction scenario is a high-level scenario that uses the other transaction scenarios for the purpose of specific transactions. For this reason, we establish a “uses” relationship between the general transaction scenario and the other transaction scenarios. This is reflected at the hierarchical state machine level by the definition of a transaction composite state in the general transaction state machine. This composite state encapsulates the whole set of transaction scenarios.

The general transaction scenario could be described by the operational state machine given in Figure 91. This state machine describes the steps that are common to all transactions. We observe that this state machine is defined independently of any particular transactions. This would allow reusing this state machine in different versions of the ATM that offer different types of transactions.

FIGURE 91. Operational state

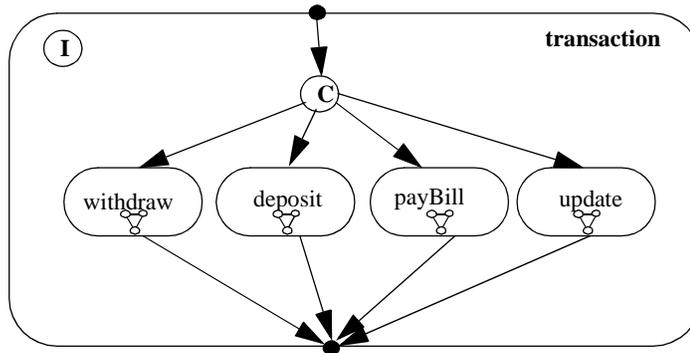


At the level of the transactions, the state machine of Figure 92 could be defined. This state machine, called *transaction*, contains a set of composite states that correspond to the different types of transactions offered by the system. Each of these composite states encapsulates a state machine that describes the steps required for a transaction.

This transaction state machine is designed using the Mutually Exclusive Scenario Pattern (Section 4.6). This pattern ensures that the scenarios can only be executed one at a time. Once defined, this state is placed in the *transaction* composite state of the *operational* state.

Then, the transaction state machine of Figure 92 is plug-in the transaction state of the operational state machine of Figure 91. The result is a concrete operational state machine that encapsulates all the transaction related behavior of an ATM. This state machine is completely independent of the control state machine previously defined. Therefore, it could be used with different control level state machines.

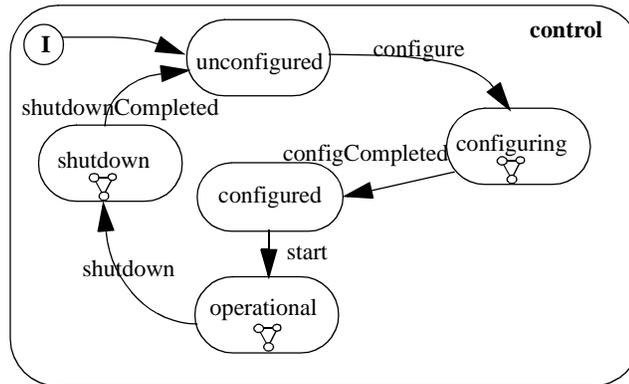
FIGURE 92. Transaction state



Integration of ATM Transactions in the Top Level Control State Machine

Finally, we plug-in the operational state machine, defined in Figure 91, in the operational state of the top level control state machine. As a result, the operational state of the control state machine is modified to become a composite state. The resulting control state machine is illustrated in Figure 93.

FIGURE 93. ATM controller component behavior



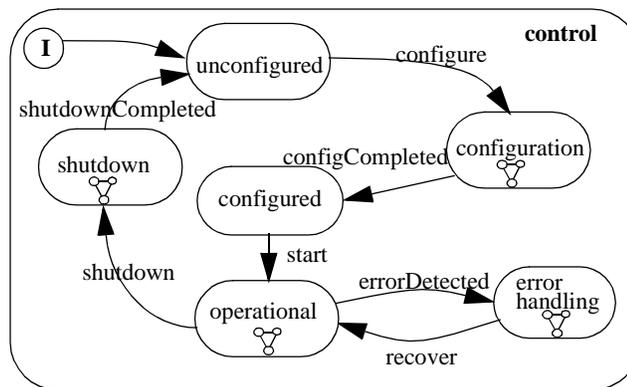
Integration of Error Handling

Now assume that at this point we want to introduce error handling in the ATM controller. The first thing we would need to do would be to define a role behavior state machine for each of the error handling scenarios we want to introduce in the ATM controller. This can be done using the RT-TROOP modeling phases. Then, we integrate all the resulting sce-

narios in a single *error handling* state machine. This can be done in a similar manner as in the previously cases.

Finally, the last step consists in integrating the resulting state machine in the control state machine of the ATM controller. The result of this step is shown in section 94.

FIGURE 94. Adding the error handling scenarios to the ATM control component behavior



Consequences

- This form of component behavior design allows for the reuse of existing state machine. It allows designing complex component behavior from a set of simpler (specialized) ones. Ultimately, it could allow building the whole behavior of a component from existing state machines.
- This approach provides for the development of libraries of specialized (and well-documented) state machines that can be used for the definition of complex component behavior. The main advantage of this approach is that designers can benefit from the existence of state machines that have already been designed and tested by others in different contexts. Thus, the designers can benefit from the experience of other designers.
- This pattern allows for packaging of existing state machines for the development (definition, or design) of customized systems, or components. For example in the context of the ATM system, different types of ATM machines could be defined by packaging dif-

ferent sets of specialized state machines. If the request was made to develop a “fast cash” ATM machine, i.e. a machine that only allows for withdraw and account update, then the different scenarios and associated state machines that have been already developed for the standard ATM machine could be reused with minor modifications.

- This pattern also allows reusing existing state machines in the design of new component behavior. For example, a generic control state machine can be used in the design of components that significantly differ at the functional level. Similarly, a state machine that defines the steps required to establish a telephone connection could be reused in many systems that need to establish such a connection. This could be the case in the ATM system if, in the start-up configuration scenario, the system need to establish a telephone connection with the central bank system (CBS).
- This pattern allows establishing a strong traceability relation between the structure of a hierarchical state machine and the scenario models that it must satisfy. Thus, it facilitates the maintenance of consistency between hierarchical state machines and the scenario models as the system is modified and extended.
- The proposed structuring of hierarchical state machines allows increasing both maintainability and extensibility. For example in the ATM system, if an error is found in the execution of the withdraw scenario, the designer knows that he/she should look for the error in the withdraw state and nowhere else. Similarly, if the requirements change for a scenario, only the state containing this functionality is subject to modifications.

4.5 Mode-Oriented Behavior Pattern

The Mode-Oriented Behavior Pattern uses a hierarchical state machine model to define the behavior of a component that must implement a set of scenarios that apply to different

behavior modes. This pattern allows dividing the overall set of scenarios into subsets that are specific to each behavior mode. It allows reducing the number of scenarios that need to be addressed at once.

This pattern allows decoupling scenarios that apply to different behavior mode. Thus, the impact of a scenario modification is limited to one encapsulated state machine (internal of a composite state machine).

Motivation

Real-time system components are often described at an abstract level in terms of set of behavior modes. In such a case, the overall set of scenarios that apply to the component is partitioned into mutually exclusive subsets. Each subset is associated with a specific behavior mode. The component changes from one mode to another upon reception of specific messages, or events.

There exists many examples of system components that are defined using behavior modes. A typical example is a TV/VCR remote control that controls both a TV and a VCR. This type of remote control has two modes: a TV mode, and a VCR mode. The user can toggle between the modes by pressing a TV/VCR button. The user inputs are interpreted differently depending of the mode in which the component is.

Another example is a telephone system that offers the “call-forward” feature. In this case, the telephone of a user that subscribes to the “call-forward” feature is either in the “normal” mode or in the “call-forward” mode. If it is in the “normal” mode, any call made to the corresponding phone number will result in the telephone ringing. However, if the telephone is in the “call-forward” mode, any call made to the corresponding phone number will result in the forwarding of the call to another predetermined phone number. In this case, the telephone of the subscriber that invoked the “call-forward” feature will not be ringing. The telephone cannot be in both modes at the same time. The telephone remains

in a given mode (normal or call-forward) until the user decides to change it by entering the appropriate command on the telephone keypad.

Other examples include:

- Electronic watch, which at a first level could be in time display mode, stop-watch mode, timer mode, or alarm mode. At a second level each of these modes breaks down into a normal mode or setup mode
- House security system, which normally have two main modes: unarmed and armed.
- VCR which have a normal and a programming mode
- Pacemaker case study used in [76] as four different modes: an idle mode, an AVI mode, a self inhibited mode, and self triggered mode

One of the characteristics of a mode-oriented component is that the component stays in its current mode until a specific mode toggling event, or message, is sent by a user. Mode toggling scenarios and functional scenarios are completely decoupled from each other.

Sometimes the different modes are explicitly described in the requirements, and sometimes they come from a design decision.

Applicability

Use the Mode-Oriented Behavior Pattern when a component must implement a set of scenarios that apply to different behavior modes.

Problem

Implementing (integrating) a large set of scenarios in a single component behavior constitutes a difficult design problem. Very often the overall set of scenarios can be divided into subsets of scenarios that apply to different behavior modes. There is usually no direct relationship between scenarios contained in the different modes.

Forces

- Reduce the number of scenarios to address at once.
- Decouple scenarios that apply to different behavior modes.
- Make the set of modes explicit in the hierarchical state machine.
- Decouple the two main aspects of scenario models: scenario interactions and individual scenario description.
- Encapsulate the set of scenarios that apply to a specific mode in a composite state.

Solution

The mode-oriented behavior state machine uses a two-level hierarchical state machine structure to structure the component behavior: a mode toggling level and a functional scenario level. In this pattern, the operational scenarios of the system are encapsulated in the second level of the hierarchical state machine. The first level can be viewed as the control level of the mode-oriented state machine. Depending of the actual mode of the component, a different set of scenarios can be executed. The behavior that results from the reception of a message depends of the mode in which the component is.

The transitions between the different modes, called mode toggling, differ from one system to another. In this pattern, we distinguish two different types of mode toggling: *sequential mode toggling* and *explicit mode toggling*.

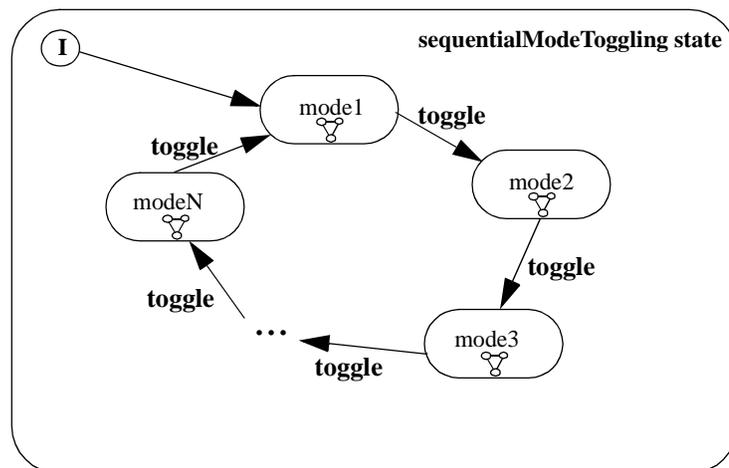
Two Types of Mode-Oriented State Machines

We define two different state machine structures (to be used at the mode toggling level of the hierarchical state machine) for mode-oriented components: a sequential mode toggling state machine and an explicit mode toggling state machine.

In the case of sequential mode toggling, a single toggling event is sufficient. In this case, the reception of the toggling event will make the component switch to the next mode. The structure of a sequential toggling state machine is illustrated in Figure 95. In this state machine structure, the mode composite states are linked together by toggling transitions in a circular manner.

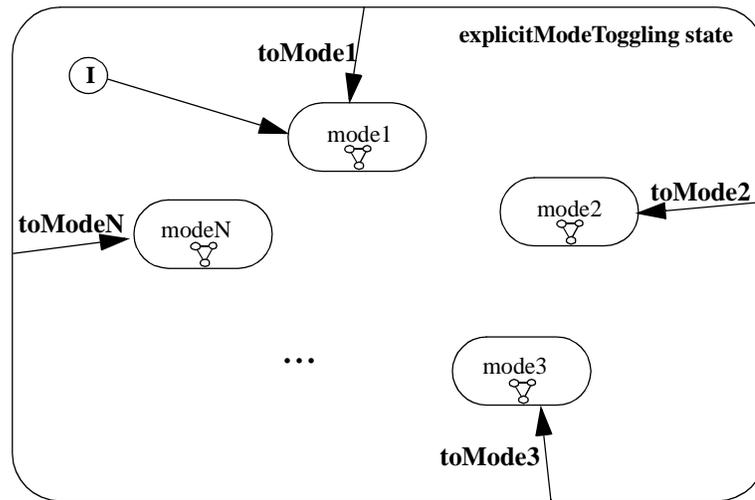
In this state machine, `mode1` is defined as the default state, and thus is the one entered by the component when it is created.

FIGURE 95. Mode-oriented behavior with sequential mode toggling



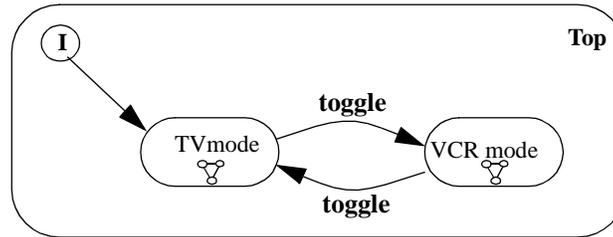
In the case of explicit mode toggling, a different toggling event, or message, must be defined for each behavior mode. In this case, the reception of a toggling event explicitly determines the mode to which the component must switch. The structure of an explicit mode toggling state machine is illustrated in Figure 96. In this state machine structure, there is no direct links between mode composite states. All toggle transitions are defined as group transitions coming from the boundary of the encapsulating state (called **explicit mode toggling state** in Figure 96) to the specific mode composite state.

In this state machine, `mode1` is defined as the default state, and thus is the one entered by the component when it is created.

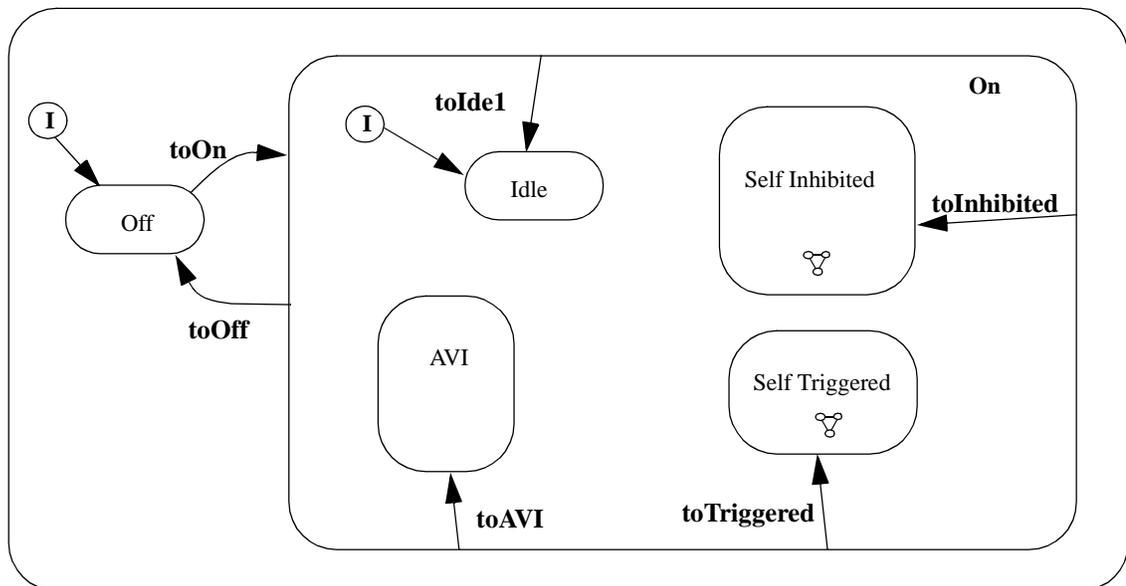
FIGURE 96. Mode-oriented behavior with explicit mode toggling

Example

A simple example of the application of the Mode-Oriented Behavior Pattern is given in Figure 97. In this figure, the behavior of a TV/VCR remote control is illustrated. This component behavior is defined using the mode-oriented behavior pattern with sequential toggling. At the first level, the hierarchical state machine is composed of two composite states: a TVmode state (default state) and a VCRmode state. Each of these composite states is internally composed of a state machine that implements the set of scenarios that can be executed while being in the mode. The internal state machines of the two modes are not illustrated here.

FIGURE 97. TV/VCR remote control with sequential toggling

The pacemaker component developed in [76] is an example of mode-oriented behavior with explicit mode toggling. This example is illustrated in Figure 98

FIGURE 98. Mode-oriented behavior with explicit mode toggling

Consequences

- Reduce the number of scenarios to address at once by partitioning the overall set of scenarios into subsets that can be addressed in different behavior modes (composite states).
- Increase maintainability and extensibility by making the set of modes explicit by associating them with a composite state.

- Decouple the two main aspects of scenario models: scenario interactions and individual scenario description (as a sequence of responsibilities). In this case, a composite state is defined for each system mode, and the set of scenarios that apply to a mode are encapsulated in the mode composite state.

The first level of the state machine, called the mode level, is composed of the set of mode composite states. At this level, the possible transition between modes is defined. At the second level, each mode separately implements its own set of scenarios.

- Make the set of modes explicit by associating them with a composite state.
- Encapsulate the set of scenarios that apply to a specific mode in a composite state.

4.6 Mutually Exclusive Scenario Pattern

The Mutually Exclusive Scenario Pattern uses a hierarchical state machine model to define a component behavior that ensures the mutual exclusion of a set of option scenarios. This pattern provides loose coupling between individual option scenarios by encapsulating the details of each scenario in a single composite state. Thus, it allows modifying any option scenario without affecting the others. Also, adding new option scenarios can be achieved at a minimal cost.

Motivation

In their operational state, real-time systems often offer a set of mutually exclusive functional options. As an example, consider an ATM system that offers a set of different transaction options that include deposit, withdraw, bill payment, and account update. When using the system, the user may choose among the set of available transaction options by

pressing the key associated with the desired transaction. ATM machines only allow for the execution of one of these options at a time. If a user wants to do more than one transaction, he/she has to do it in a sequential manner, and wait for the completion of one transaction before triggering another one.

In terms of scenarios, we describe the ATM system using the following transaction scenarios:

- An abstract *Transaction* scenario, which describes the sequence of responsibilities common to all transactions. It includes reading the card number, verifying the PIN, getting the user transaction selection, executing the required transaction, printing a receipt, and returning the card.
- One concrete transaction scenario for each of the different types of transactions offered by the ATM system, i.e. withdraw, deposit, bill payment, and account update. Each of these scenarios describes the details of a transaction.

Also, at the UCM modeling level, the mutual exclusion of the scenarios is explicitly specified by means of proper scenario documentation. This can be achieved either by specifying scenario execution constraints or by specifying appropriate preconditions. In our ATM example, the mutual exclusiveness of the transaction scenarios is specified in the transaction UCM (Figure 85) by means of the nonfunctional requirement given at the bottom of the textual description associated with the UCM. This constraint specifies that: “There could not be more than one active transaction scenario per ATM”.

The problem faced by designers in such a case consists in defining the behavior of the ATM central controller in such a way that the mutual exclusion of the transaction scenarios is ensured.

Another example of mutually exclusive scenarios can be found in fax machines. Fax machines have two main options: sender and receiver. These two options are mutually exclusive. A fax can send or receive messages, but it cannot do both at the same time. The

first event received by a fax while being in its *idle* state determines the mode of the fax machine. The fax machine returns to its idle state after completing reception or sending.

Applicability

Use the Mutually Exclusive Scenario Pattern when a system must implement (or when a component must coordinate the execution of) a set of mutually exclusive scenarios.

Problem

The general problem faced by designers when they have to integrate a set of mutually exclusive scenarios consists in defining the behavior of system components in such a way that the mutual exclusion of the scenarios is ensured. The Mutually Exclusive Scenario Pattern provides a solution for this problem.

Forces

- Ensure the mutual exclusion of scenario execution.
- Make the set of options explicit in the state machine structure.
- Decouple the two main aspects of scenario models: scenario interactions and individual scenario description (as a sequence of responsibilities).
- Locality of change.
- Allow for addition of new option scenarios at a minimal cost.

Solution

The Mutually Exclusive Scenario Pattern solves the mutually exclusive scenario problem in the following manner:

1. It allocates the control of the whole set of mutually exclusive scenarios to a single component, called the scenario coordinator (or controller).
2. It defines the behavior of the scenario coordinator to only allow the execution of one scenario at the time.

The scenario control allocation is usually done at the interaction diagram modeling level, as it constitutes one of the main criteria for the definition of message sequences.

At the component behavior level, the Mutually Exclusive Scenario Pattern defines the behavior of the scenario coordinator by means of a two-level hierarchical state machine. The first level, called the *scenario option* level, describes the relationship between individual scenarios, while the second, called the *scenario description* level, describes the specifics of each scenario.

The scenario option level focuses on establishing the mutual exclusion relationship that must hold between the scenarios. At the scenario option level each individual scenario is encapsulated in a composite state that contains a state machine describing the steps of the scenario. At this level, the choice among the different options is explicitly given by the structure of the state machine.

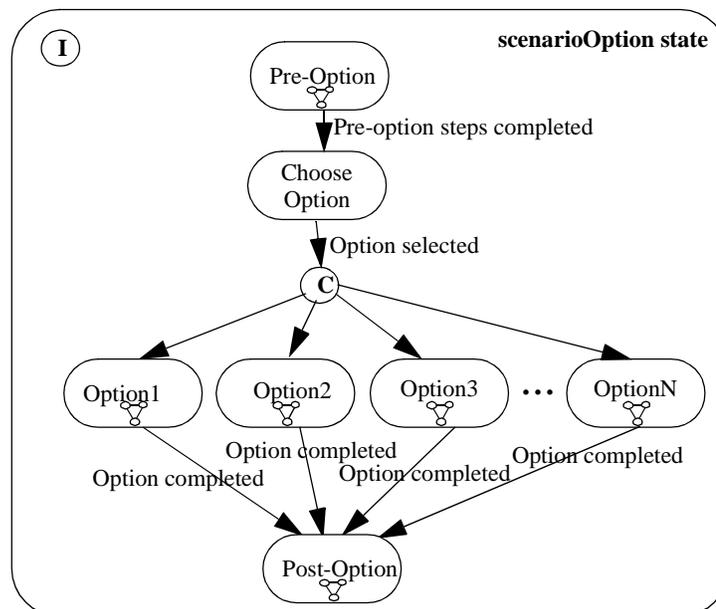
The structure of the scenario option state machine is given in Figure 99. This state machine is composed of the following parts:

- **A Pre-Option composite state:** This composite state contains the sequence of steps that need to be executed before choosing a specific option. This sequence may contain some initialization steps as well as steps that are common to all options. Pre-option steps are represented here in a composite state for the sake of clarity. However in practice, they are often not all grouped together in a single composite state, but are rather expressed as a sequence of states and transitions that lead to the ChooseOption state.

When the execution of the pre-option steps is completed, the state machine progresses to the ChooseOption state.

- **A ChooseOption state:** This state represents the point where the component behavior waits for the selection of an option. After receiving the appropriate message the component moves to the option choice point.
- **An Option choice point:** This choice point is defined in terms of the different options offered by the component. Depending on the message received by the component in the ChooseOption state, a transition to the requested option state will be taken.
- **A set of Option composite states:** This set of states contains one composite state per option offered by the component. Each of these composite state machines contains a lower level state machine that describes the sequence of steps required by an option.
- **A Post-Option composite state:** This composite state contains the sequence of steps that need to be executed after the execution of a specific option. This sequence may contain some termination steps as well as steps that are common to all options. Like the pre-option steps, post-option steps are represented here in a composite for the sake of clarity. However in practice, they are often not all grouped together in a single composite state, but are rather expressed as a sequence of states and transitions.

FIGURE 99. Structure of the scenario option state machine

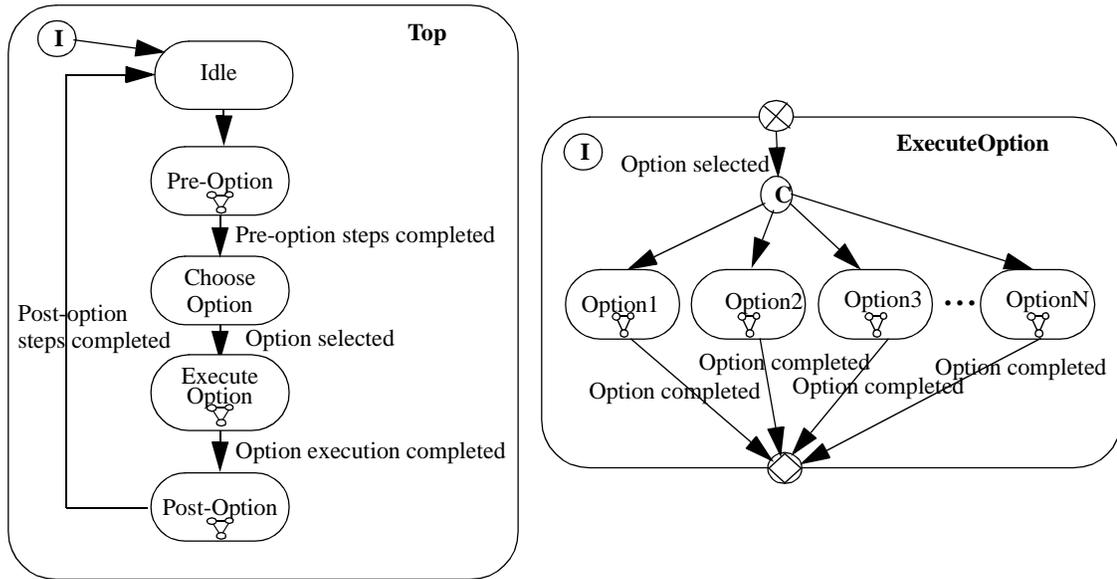


The scenario description level focuses on the description of the individual scenarios. At this level, each scenario is defined by means of a separate state machine that describes the sequence of actions that must be executed for the purpose of the scenario. Thus, each individual scenario is encapsulated in a composite state.

The structure of the hierarchical state machine used in this pattern allows decoupling the two main aspects of scenario models: scenario interactions and individual scenario description (as a sequence of responsibilities). This pattern allows for locality of change. It also allows maintaining strong traceability links between scenario models and component behavior. Thus, if requirements are modified, only a well-defined (and encapsulated) part of the hierarchical state machine will need to be modified. For example, if the requirements of one scenario are modified, modifications will be limited to the composite state that contained the modified scenarios. The rest of the hierarchical state machine will remain intact. This way maintainability and extensibility are increased.

An alternate representation (in terms of hierarchical state machine structure) of the Mutually Exclusive Scenario Pattern is given in Figure 100. The two representations are equivalent in terms of behavior.

FIGURE 100. Structure of the scenario option state machine



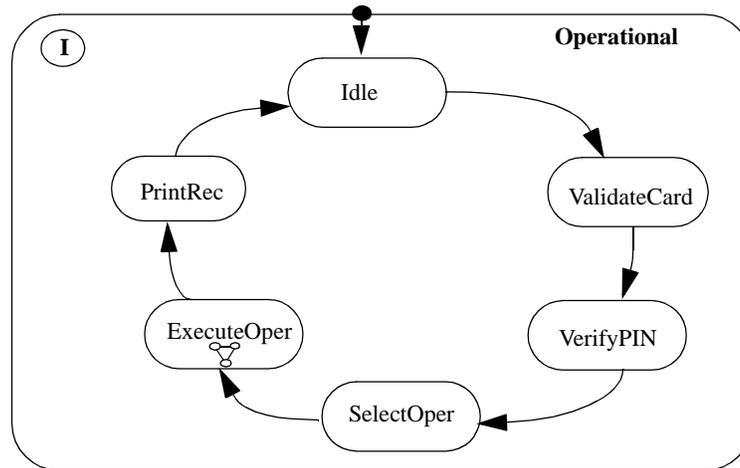
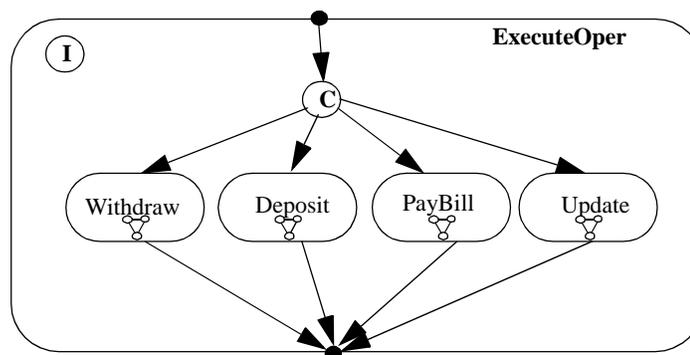
Example

Going back to our ATM machine design example, the application of the Mutually Exclusive Scenario Pattern results in the application of the following steps.

First, we allocate the control of the all transaction scenarios, (which include abstract transaction, withdraw, deposit, bill payment, and account update) to the ATM controller component.

Second, we define the first level state machine of the ATM controller. The resulting state machine is given in Figure 101 and Figure 102.

Third, we integrate the transaction scenario state machines in the ATM coordinator hierarchical state machine. These scenarios are not showed here.

FIGURE 101. Operational state of the ATM control component behavior**FIGURE 102. ExecuteOper state of the ATM control component behavior**

Consequences

- Allows for locality of change by using a two level hierarchical state machine structure. The first level describes the set of options, while the second level describes the details of the different option scenarios. The details of each option scenario are encapsulated in a composite state. Thus, modifications to the set of options only impacts at the first level, while modifications to an existing option scenario only impact in the composite state that encapsulate the modified option scenario. This allows increasing maintainability and extensibility.

- The two level hierarchical state machine used in the Mutually Exclusive Scenario Pattern allows modifying the set of system options at a minimal cost. The addition of a new option or the removal of an existing one only requires the addition or the deletion of an option composite state at the first level of the hierarchical state machine. This provides for high maintainability and extensibility of the set of option scenarios.
 - Increase component understandability by making explicit the set of options. The option level state machine makes explicit the set of options by defining a composite state for each option (the option composite state is given the name of the system option defined in the requirements), and by using a choicepoint in the state machine structure to emphasize its optional nature.
 - Facilitate individual scenario modifications by encapsulating each scenario in a composite state. The details of each option scenario are encapsulated in a composite state. This allows modifying the behavior of a scenario without affecting the behavior of the other scenarios.
 - Decouples option scenarios. The fact that different option scenarios are completely decoupled from each other ensures that there will be no interaction between them.
 - Uses a centralized approach, which increases the dependence on the controller component.
-

4.7 Scenario Interaction Patterns

Motivation

In real-time systems, scenarios often interact with each other. Different types of scenario interaction are possible. In this section, we define a set of design patterns that deals with

different types of basic scenario interactions. This set of patterns includes scenario composition, waiting place, timed waiting place, and scenario aborting.

Applicability

Real-time systems and any other types of systems that allow for scenario interactions.

Problem

Many of the important problems that occur in real-time systems are the result of scenario interactions.

One of the difficult problems faced by real-time system designers is to ensure that the overall set of component behavior hierarchical state machines they produce correctly handles the different scenario interactions. This problem becomes particularly difficult to solve when the number of scenarios and the number of scenario interactions become large.

Forces

- Define hierarchical state machine structure so that individual scenario description and scenario interactions are decoupled as much as possible.
- Maintain traceability between hierarchical state machine structure and scenario interactions in the scenario model.
- The patterns defined in this section allow establishing traceability between scenario interactions and hierarchical state machine structures. Thus, we can determine where in a hierarchical state machine a scenario interaction is handled.

Solution

The objective of the design patterns proposed in this section is to define fragments of state machine structure to deal with different types of scenario interactions. These state machine fragments be used at all levels of hierarchical state machines.

Solutions to the specific types of scenario interaction are defined in the following four sections (section 4.7.1 to section 4.7.4).

It should be noted that all scenario interactions take place in a component. In the description of the scenario interaction patterns, we only describe the structure of the component behavior in which the interaction occurs. This component is the one responsible for handling the interaction in a correct manner. The behavior of all other components involved in the interacting scenarios is not affected by the interaction. Therefore, the description of the behavior of these components is of no interest in the context of the scenario interaction patterns.

In the description of the different patterns, we use composite state to abstract from scenario details that are not important in the context of the proposed solution. These composite states can be decomposed when applying the pattern without altering the nature of the pattern solution.

4.7.1 Scenario Composition

Scenario composition is a scenario interaction in which the termination of the execution of a first scenario triggers the start of the execution of a second scenario. In order to define a composition interaction between two scenarios, say S1 and S2, two conditions must be verified:

1. The postcondition of S1 satisfies the precondition of S2.

2. The resulting event of S1 is the triggering event of S2.

Solution

When defining the behavior of the component responsible for the scenario composition interaction, the designer must ensure that the two scenarios will execute sequentially in the resulting system, i.e. the termination of the first scenario will trigger the execution of the second one.

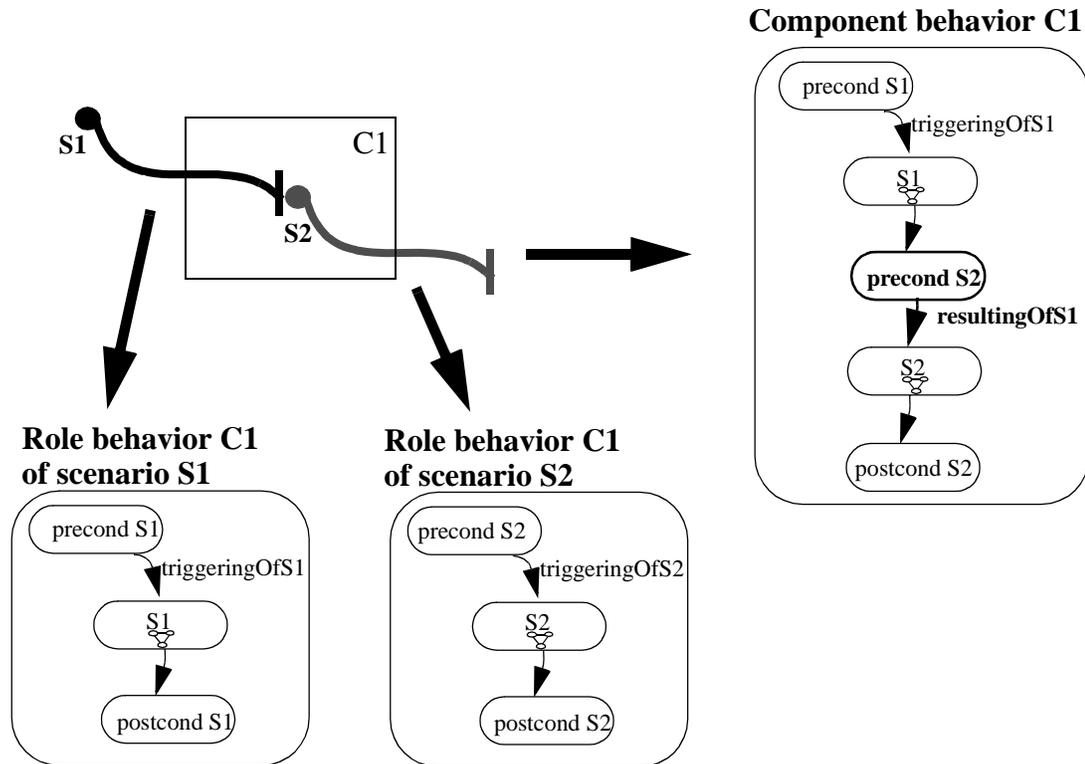
In Figure 103, the structure of the hierarchical state machine of the component responsible for a scenario composition interaction is illustrated. This figure illustrates the scenario interaction by means of a UCM map. This map contains two scenarios, S1 and S2, where the completion of S1 triggers the execution of S2. Below the UCM map, the role component behaviors of component C1 in the context of the two scenarios are given. These two role component behaviors are composed of three states: a precondition state, a postcondition state, and a composite state that encapsulate the steps of the scenarios.

On the right hand side of the figure, the structure of the hierarchical state machine of component C1 is given. We observe in this figure that the two independent sequences of states, described in the role component behaviors, have been merged into a single sequence of states that contains the steps of the two scenarios. We also observe that the postcondition of S1 have disappeared in the merging. This is explained by the fact that when composing two scenarios, the postcondition of the first scenario must satisfy the precondition of the second one (i.e. the postcondition of the first scenario must be a substate of the precondition of the second one). Thus, when merging the two role behavior state machines we only keep the precondition of the second scenario.

After getting to the precondS2 state, component C1 waits for the reception of the resulting event of scenario S1. Once this event is received, component C1 continues on with the execution of scenario S2.

This resulting event must be sent to C1 by another component involved in the scenario.

FIGURE 103. Scenario composition



4.7.2 Waiting Place

A waiting place illustrates a point at which a scenario is blocked until an unblocking event arrives and allows the scenario to continue its execution. The unblocking event is normally generated by another scenario. It could also be introduced by an external user of the system. Waiting places are used in scenario description to specify some type of synchronization between two scenarios, or between a scenario and an external user.

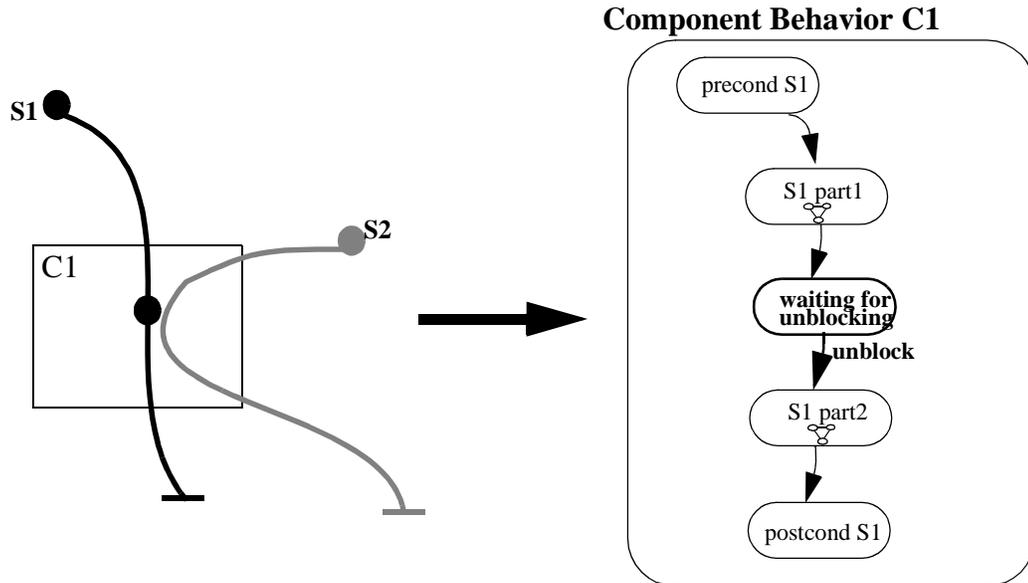
When defining the behavior of the component responsible for the implementation of a scenario waiting place, the designer must ensure that the execution of the scenario will be stopped at the waiting place until the unblocking event is received. S/he must also ensure that the generation of the unblocking event by the unblocking scenario will correctly unblock the scenario waiting at the waiting place.

In Figure 104, a scenario interaction using a waiting place is illustrated. This map contains two scenarios: S1 that contains a waiting place, and S2 that is responsible for unblocking S1. In this case, the waiting place is located in component C1, which means that C1 is the component responsible for implementing the waiting place.

Solution

This type of scenario interaction can be handled by the simple hierarchical state machine structure given in the right hand side part of Figure 104. We observe in this figure that the responsibilities of scenario S1 are grouped in two parts: part1 that contains the responsibilities that take place before the waiting place, and part2 that contains the responsibilities that take place after the waiting place. These two parts of scenario S1 are grouped in two composite states in the state machine of component C1 that is responsible for implementing the waiting place. Also, we introduce a `waiting_for_unblocking` state between the two composite states, `S1part1` and `S1part2`, that encapsulate the responsibilities of scenario S1. The transition between the `waiting_for_unblocking` state and the composite state `S1part2` is taken upon the arrival of the `unblock` message. Once the `unblock` message is received, the execution of scenario S1 resumes in state `S1part2`.

FIGURE 104. Waiting place



4.7.3 Timed Waiting Place

Scenarios can also interact by means of a timed waiting place. This case is similar to the previous one, except that while being in the waiting place, two different events can occur: the normal unblocking event, and a timeout event. Depending on the received event, two different paths can be taken when leaving the waiting place: a normal execution path that is taken if the unblocking event arrives before the timeout event, and a timeout path that is taken if the timeout event arrives first.

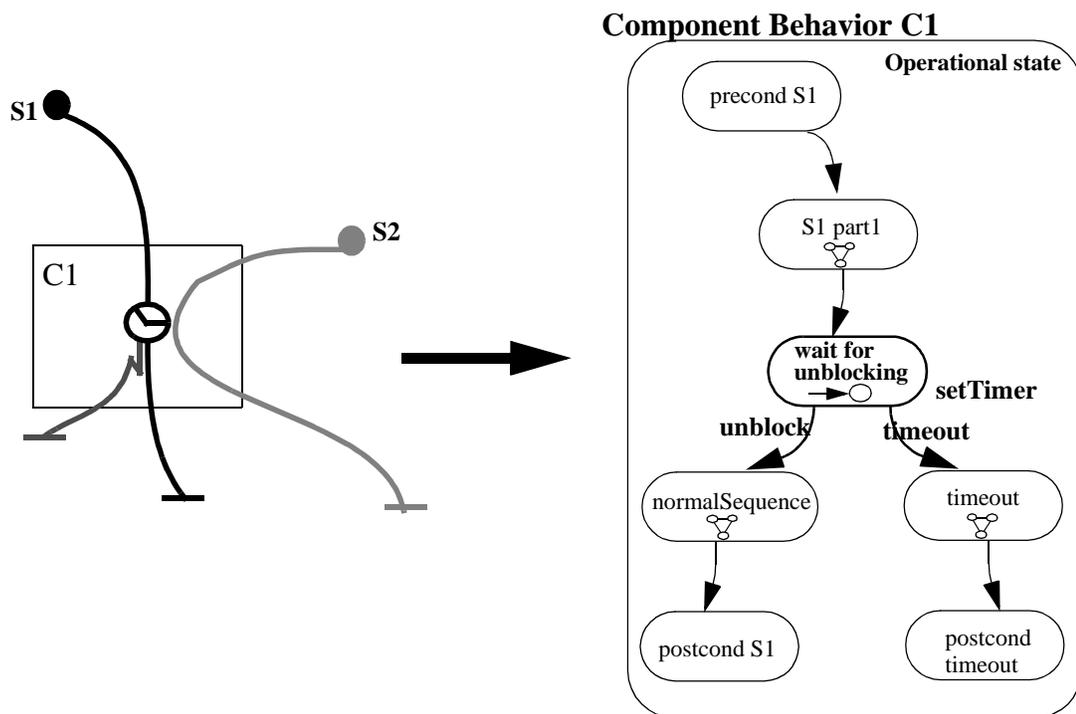
A scenario interaction using a timed waiting place is illustrated in Figure 105. In this figure, scenario S1 contains a timed waiting place that can be unblocked by scenario S2. The timed waiting place of scenario S2 is located in component C1, which means that C1 is responsible for implementing the timed waiting place.

Solution

A solution for implementing timed waiting place interactions is given in the right hand side part of Figure 105. The first part of this solution is similar to the one defined in section 4.7.2. It splits the responsibilities of scenario S1 in two parts: part1 that contains the responsibilities that take place before the waiting place, and part2 that contains the responsibilities that take place after the waiting place. These two parts of scenario S1 are grouped in two composite states in the state machine of component C1: S1part1 and S1part2. Also, as in section 4.7.2, a `waiting_for_unblocking` state is defined and connected by a transition to the state S1part1.

Then, the second part of this solution is different than the one defined in section 4.7.2. In this pattern, we need to set a timer upon entering the `waiting_for_unblocking` state. This is achieved by defining state entry action in the `waiting_for_unblocking` state that sends a `setTimer` message to a timer. Then, depending on the message that is received one of two transitions is taken. If the `unblock` message is received, then the execution of scenario S1 resumes as the transition to state S1part2 is taken. On the other hand, if a `timeout` message is received, then the timeout path is executed and the scenario terminates.

FIGURE 105. Timed waiting place



4.7.4 Scenario Aborting

A significant portion of real-time system scenarios is related to aspects such as error detection and recovery, interruption, and exception handling. The nature of the interaction between those scenarios and normal operation scenarios is different than the type of interactions that take place between two normal operation scenarios.

The main difference is that the triggering of a scenario of this type interrupts (at least temporarily) the execution of ongoing scenarios. For example, in an ATM system, if the user presses the cancel button at any time before the system starts to process a transaction, the system will interrupt the ongoing scenario and terminate the session. Another example is telephone systems where the occurrence of an onhook event (from any of the users) in a

two-way telephone call will result in the interruption of any ongoing scenarios, and will terminate the call.

When designing component behavior, designers must ensure that the triggering of any scenarios of this type will interrupt the execution of ongoing scenarios before executing the sequence of responsibilities (activities or actions) required for the interrupting scenario. The scenario interruption pattern provides a solution for this problem.

A scenario aborting interaction is illustrated in Figure 106. In this figure, the triggering of scenario S2 aborts the execution of scenario S1. The abort arrow is located in component C1, which means that C1 is responsible for implementing the aborting of scenario S1.

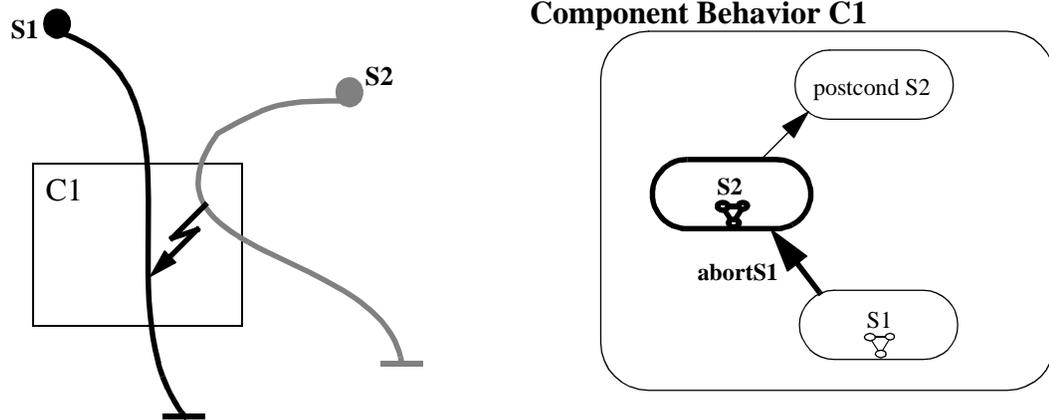
Solution

A state machine structure that allows handling scenario aborting interactions is given in the right hand side part of Figure 106.

In this case, the solution consists in encapsulating the interruptible scenario in a composite state, and in defining a group transition from the border of the composite state that lead to the execution of the aborting scenario. Therefore, no matter where component C1 is in the execution of scenario S1, the arrival of the scenario aborting message `abortS1` will interrupt the execution of scenario S1 and start the execution of scenario S2 (which is also encapsulated in a composite state, simply labelled S2, in this case)

This pattern can also be used in cases where a single scenario can abort the execution of a set of scenarios. In this case, the whole set of scenarios that can be aborted are encapsulated in a single composite state.

FIGURE 106. Scenario aborting



4.8 More Patterns

We view the set of patterns defined in this chapter as a starting point for the definition of a catalogue of patterns specialized in the design of complex hierarchical state machines. Many other patterns can be defined:

- Concurrent scenarios

In this case, a solution consists in allocating the control of each concurrent scenario to a different concurrent component.

- Scenario synchronization
- Uninterruptable scenario (or sequence of actions).
- Error recovery
- and so on

4.9 Chapter Summary

In this chapter, we defined a set of design patterns that allow designing the hierarchical state machine of complex components in a systematic manner using scenario models. The hierarchical state machines are built in two steps: definition of the state machines associated with individual scenarios, and integration of existing state machines into more complex hierarchical state machines.

From our experience the patterns defined in this chapter are highly reusable. Their usage allows:

- Reducing the time required to design complex component behavior.
- Increasing the quality of the design of complex component behavior.
- Reducing the time required to test complex component behavior.
- Reducing undesired scenario interactions.
- Increasing system maintainability and extensibility by providing for consistent structuring of hierarchical state machines across a whole system.

The set of patterns proposed in this chapter is not specific to the three modeling techniques used in this thesis, i.e. UCM, MSC and ROOM. They can be adapted for other models such as the ones used in UML.

CHAPTER 5 Application of RT-TROOP Modeling Process: A Simple Printer System Case Study

One of the objectives of this thesis is to show how the proposed RT-TROOP modeling process can be used in practice. In this chapter, we illustrate the different RT-TROOP modeling phases using a simple printer system. The development of this simple system allows us to illustrate the systematic and traceable aspects of RT-TROOP modeling. We use two iterations to demonstrate the iterative nature of our modeling process. This case study also gives examples of how the behavior integration patterns described in Chapter 4 can be used, and how the fine-grained traceability information maintain throughout the process can be used to resolve design problems resulting from undesired scenario interactions.

However, the simple case study developed in this chapter does not allow us to illustrate all the different aspects of RT-TROOP modeling. For example, the printer system contains a single level of components, which means that component decomposition is not illustrated.

This chapter is structured as follows. In section 5.1, we describe the requirements of the printer system. From section 5.2 to section 5.11, we successively apply the RT-TROOP modeling phases to produce a ROOM model of the system. The modeling phases are applied here in the same sequence as they are described in Chapter 3. Each of the modeling phases is illustrated and discussed in relation with the different issues associated with it. In the first iteration, we explicitly illustrate and discuss each step of the RT-TROOP modeling process. In section section 5.12, we discuss the testing of the ROOM model pro-

duced in iteration 1. Then, section 5.13 describes the second iteration of the printer system development. From section 5.13.1 to section 5.13.6, we illustrate and discuss the different models produced during the second iteration. For the sake of conciseness, in the second iteration, we only give the resulting models and discuss their relationship with the models of the previous iteration. In section 5.13.7, we discuss the testing of the ROOM model produced in iteration 2. In section 5.13.8, we show how the traceable nature of the RT-TROOP modeling process can be used to fix errors discovered in the testing stage. Finally, we summarize in section 5.14.

5.1 Requirements

The current case study consists in developing a simple printer system, called `PrinterSystem`. `PrinterSystem` is composed of two components: a printer driver and a printer.

The printer driver component is responsible for:

- Controlling the execution of the different scenarios.
- Managing printer requests.
- Feeding the printer with file characters obtained from the file system via its standard interface.
- Ensuring that at the termination of each printing request (no matter if the printing terminates normally or is interrupted), the requested file is closed and the printer is released.

The printer component is responsible for printing the characters received from the printer driver.

The environment of `PrinterSystem` is composed of:

- A user who enters printer commands
- A file system by which files are accessed
- A piece of paper on which the characters are printed

The file system provides the usual set of file handling functions which include:

- `openRead fileName`, which opens the file *fileName* in the reading mode. If the file is successfully placed in the reading mode, then the `fileOpened` message is returned. Otherwise, an `error` message is returned (there exist different types of errors message).
- `close fileName`, which closes the file *fileName*. If the file is successfully closed, then the `fileClosed` message is returned. Otherwise, an `error` message is returned (there exist different types of errors message).
- `readNextChar fileName`, which reads the next character of the file *fileName*. If the next character is successfully read, then this character is returned. If end-of-file is reached in *fileName*, then an `eof` message is returned. Otherwise, an `error` message is returned.

Other functions like `openwrite`, `write`, and `concatenate` are also defined.

For the purpose of this case study, we abstract from the queueing mechanism that is required in `PrinterSystem` to handle multiple printing requests. We only consider one request at the time. If `PrinterSystem` is busy printing a file, it will simply reject other printing requests. The addition of a printer queue could be addressed in a future iteration.

PrinterSystem Scenarios

The current case study is concerned with the implementation of four main scenarios:

- `PrintFile`, which describes the steps required to print a requested file
- `StopPrinting`, which describes the steps required to stop the printing of a file
- `StartUp`, which describes the steps required to bring the system to its operational state

- **Shutdown**, which describes the steps required to shutdown the system

Following the approach proposed by the scenario partitioning process pattern (section 4.3), we split the four scenarios into two scenario clusters: one containing the operational scenarios and one containing the control scenarios. In this case study, each scenario cluster is addressed in a separate iteration.

As previously mentioned, the development of this case study is conducted in two iterations. In terms of scenarios, the content of the iterations is defined as follows:

- Iteration 1 is concerned with the implementation of the operational scenarios, namely the **PrintFile** and **StopPrinting** scenarios. Alternatives to these two scenarios are not considered in this iteration.
- Iteration 2 is concerned with integration of the control scenarios: the **StartUp** and **Shutdown** scenarios. Again, alternative scenarios are not considered in that iteration.

The scenario content of the iterations has been chosen to illustrate different aspects of the RT-TROOP modeling process. The first iteration illustrates the definition of RT-TROOP models from scratch, while the second iteration illustrates the integration of scenarios that relate to different aspects of the system (operational and control). Also, in the second iteration, we show how the traceability maintained by the RT-TROOP modeling process can be used to modify system models in a consistent manner.

5.1.1 STDs for Iteration 1

PrintFile Scenario

The **PrintFile** scenario may be triggered at any time when **PrinterSystem** is idle. To start the printing of a file, the user enters the **print** command together with the name of the file

to be printed, `fileName`. As a result of the `print` command, the system first performs the initialization responsibilities required for the printing of the file, and then enters the printing loop until end-of-file is reached. The initialization of the printing consists in requesting the printer and opening the file to be printed. The printing loop consists of the following sequence of responsibilities: get next character from the file system and print the character. Once end-of-file is reached, the printer is released and the file `fileName` is closed. The STD format of the `PrintFile` scenario is given in Figure 107.

FIGURE 107. PrintFile STD

STD Identifier: PrintFile	
Description: Scenario describing the steps required to print a file.	
External Actors: User, File System	
Precondition: PrinterSystem is idle	
Triggering event: A user enters the print command together with the name of the file to be printed	
<ol style="list-style-type: none"> 1. Receive the print command together with the name of the file to be printed from the user 2. Request the printer 3. Open the file 4. Execute the printing loop until the EOF event is received from the file system. This loop consists of the following steps: <ul style="list-style-type: none"> - get the next character - if not EOF, then print the character on paper 5. Release the printer 6. Close the file 	
Postcondition: PrinterSystem returns to its idle state	
Resulting event: The file is printed	
Alternatives: <ul style="list-style-type: none"> - If PrinterSystem is busy printing, then the print request is rejected. (Not included) - If the printer is unavailable for printing, then the print request is rejected. (Not included) - If the file cannot be opened, then the print request is rejected. In this case, the printer previously requested must be released. (Not included) 	
Nonfunctional requirements:	
Comments: <ul style="list-style-type: none"> - Steps 5 and 6 can be inverted. 	

StopPrinting Scenario

The StopPrinting scenario may be triggered at any time when PrinterSystem is in the state of printing a file. The StopPrinting scenario is triggered by the entry of the stop command, together with the name of the file, fileName, that is being printed. On the reception of the stop command, the printer is released and the file fileName is closed.

FIGURE 108. StopPrinting STD

STD Identifier: StopPrinting	
Description: Scenario describing the steps required to stop the printing of a file.	
External Actors: User, File System	
Precondition: PrinterSystem is printing	
Triggering event: A user enters the stop command together with the name of the file that is being printed	
<ol style="list-style-type: none"> 1. Receive the stop command together with the name of the file that is being printed from the user 2. Release the printer 3. Close the file 	
Postcondition: PrinterSystem returns to its idle state	
Resulting event: The printing of the file is cancelled	
Alternatives: - If the name of the file entered by the user does not correspond to the name of the file that is being printed, then the stop request is rejected. (Not included)	
Nonfunctional requirements:	
Comments: - The reception of the stop command stops the printing of the file that is being printed - Steps 2 and 3 can be inverted.	

5.2 From STD to UCM (Iteration 1)

The objective of this model transition is to produce a UCM related path set for each STD contained in the STD model. In `PrinterSystem`, we have two STDs: `PrintFile` and `Stop-Printing`.

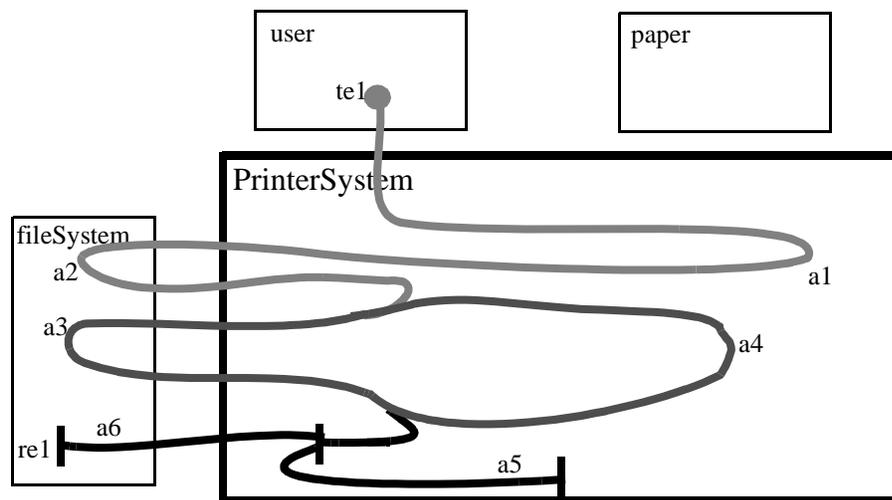
PrintFile Related Path Set

In Figure 109, the UCM related path set corresponding to the `PrintFile` STD is illustrated. We observe that the UCM path described in this figure is composed of three distinct parts:

- An initialization part, which is composed of the start point `te1` and two responsibilities.
 - `setPrinter (a1)`: This responsibility essentially consists in initializing the printer for printing.
 - `openFile (a2)`: This responsibility consists in opening the file to be printed in the reading mode.
- A print character loop part, which is composed two responsibilities.
 - `getNextCharacter (a3)`: This responsibility consists in reading the next character from the file that the system is currently printing,
 - `printCharacter (a4)`: This responsibility consist in printing the character received on paper (paper component in the diagram).
- A termination part, which is composed of the end bar associated with `re1` and two responsibilities. These two responsibilities can be executed in any order.
 - `releasePrinter (a5)`: This responsibility consists in a set of printer activities that are required to complete the printing job and return the printer in the idle state.
 - `closeFile (a6)`: This responsibility consists in closing the printed file.

The PrintFile scenario is triggered by the reception of the print command with the name of the file to be printed. The precondition associated with the start point **te1** states that the path will only start if the state of PrinterSystem is idle, and the postcondition associated with the end bar **re1** states that PrinterSystem will be placed in the idle state after the completion of the PrintFile path. The termination part will be executed once end-of-file is reached in the file to be printed.

FIGURE 109. PrintFile UCM



te1: print message with the name of the file to be printed (pre-condition: the system is in the idle state)
 re1: printing completed (post-condition: the system returns in the idle state)
 a1: request printer
 a2: open file
 a3: get next character
 a4: print character
 a5: release printer
 a6: close file

StopPrinting Related Path Set

In Figure 110, the UCM related path set corresponding to the StopPrinting STD is illustrated. This related path set is composed of a single path, which contains the start point **te2**, end bar **re2**, and two responsibilities:

releasePrinter (a5): This responsibility consists in a set of printer activities that are required to complete the printing job and return the printer in the idle state.

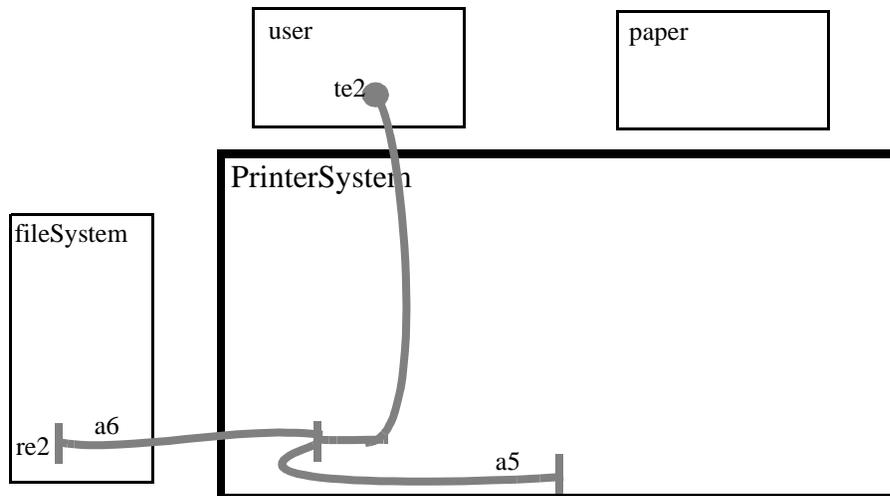
closeFile (a6): This responsibility consists in closing the printed file.

As in the case of the PrintFile related path set, these two responsibilities can be executed in any order.

This path is triggered by the entry of the stop command with the name of the file that is being printed. This scenario can only be triggered if PrinterSystem is in the printing state (precondition). As a result of executing the StopPrinting path, PrinterSystem will be returned in the idle state (postcondition).

Also, the execution of this path must interrupt the execution of the PrintFile UCM (related path set).

FIGURE 110. StopPrinting UCM



te2: stop printing (pre-condition: printing state)
 re2: printing stopped (post-condition: the system is returned to the idle state)
 a5: release printer
 a6: close file

5.3 UCM Modeling Phase (Iteration 1)

The objective of the UCM modeling phase is to integrate the unbound UCM related path sets defined in the previous phase in the UCM model of the system. In this case, we need to integrate two related path sets: `PrintFile` (Figure 109) and `StopPrinting` (Figure 110).

In this modeling phase, we:

- Restructure the UCM related path set to introduce stubs.
- Define system components.
- Define interaction between paths.
- Allocate responsibilities to the different components.

As mentioned in section 3.4, the order in which these different elements are introduced in the UCM model varies. In the present case, the modeling activities are carried out in the order they are described above.

Introducing Stubs

The first thing we do in this phase is simplify the UCM related path sets produced in section 5.2 by introducing two stubs. The first one, called `getResources`, describes the sequence of responsibilities that must be executed before entering the printing loop, i.e. the `openFile` and `requestPrinter` responsibilities of the `PrintFile` scenario.

The second one, called `releaseResources`, describes the sequence of responsibilities that must be executed when exiting the printing loop, i.e. the `closeFile` and `releasePrinter` responsibilities. This stub is used in both related path sets.

The two stubs are illustrated in Figure 112. The two related path sets that result from the introduction of the stubs are illustrated in Figure 111.

FIGURE 111. Related path sets of the PrinterSystem UCM model

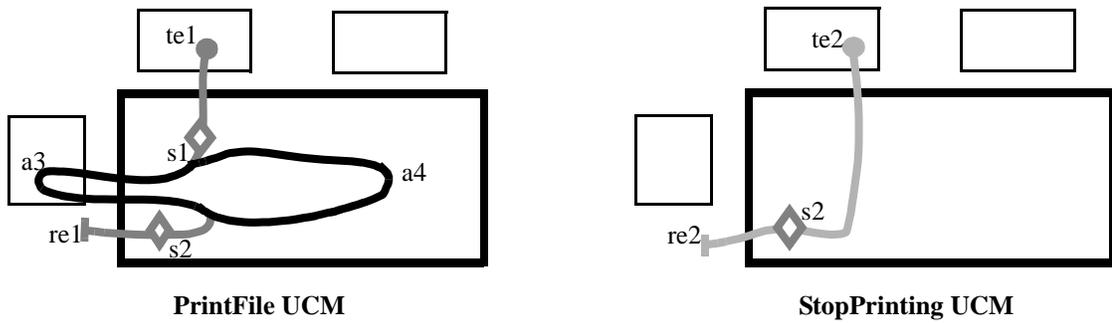
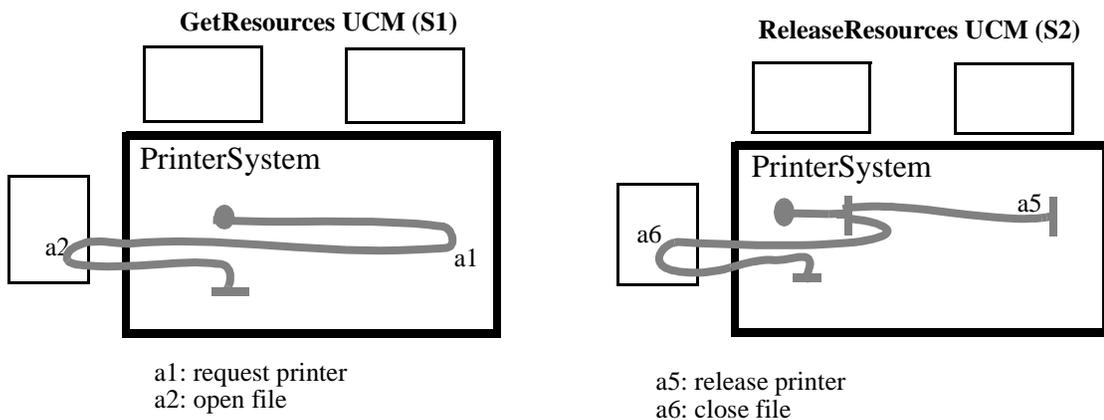


FIGURE 112. PrinterSystem stubs



Component Definition

In the requirements of PrinterSystem, the set of components that compose both the system and its environment are specified. PrinterSystem is composed of two components:

- printerDriver, which is responsible for controlling the printing process
- printer, which offers the standard printer functionality

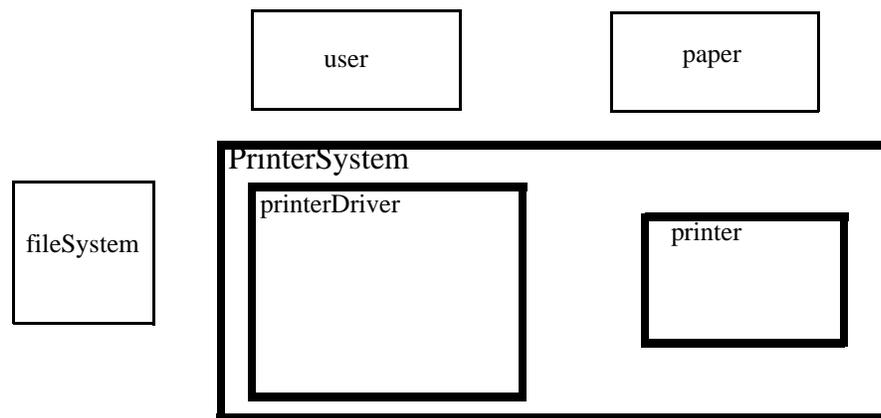
These two components collaborate together to provide the overall functionality of the system.

The environment of `PrinterSystem` is mainly composed of three components:

- `user` that can generate two different events: `print` and `stop`
- `fileSystem` with which `PrinterSystem` communicates to get the characters to be printed
- `paper` on which characters are printed

The resulting component context diagram is given in Figure 113.

FIGURE 113. The `PrinterSystem` and its environment



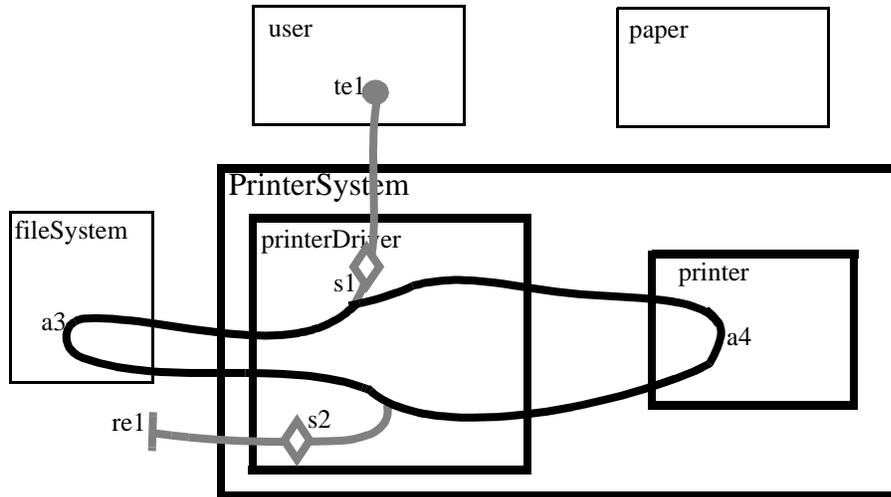
Responsibility Allocation

We can now describe the different scenarios of `PrinterSystem` in the context of the component context diagram using the UCM notation. Figure 114 through Figure 117 illustrate the bound version of the four UCM maps previously defined in Figure 111 and Figure 112.

In the textual description of the scenario given in the system requirements, no explicit responsibility allocation is provided. However, it is mentioned in the system requirements that the `printerDriver` component is responsible for controlling the printing process, and that the `printer` component must be modeled as a generic commercial printer, and therefore provides the usual printing functionality. It is also mentioned that files are accessed through the existing `fileSystem` for which a set of existing functions is given. As men-

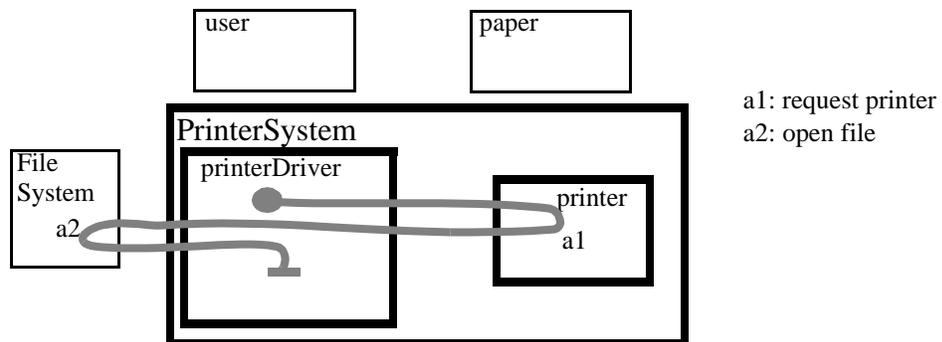
tioned in section 3.4.2, the role description of the system components constitutes the main information used for the purpose of responsibility allocation.

FIGURE 114. PrintFile UCM



- te1: start printing (pre-condition: the system is in the idle state)
- re1: printing completed (post-condition: the system returns in the idle state)
- s1: get resources stub (request printer, open file)
- s2: release resources stub (release printer, close file)
- a3: get next character
- a4: print character

FIGURE 115. getResources UCM (S1)



- a1: request printer
- a2: open file

FIGURE 116. releaseResources UCM (S2)

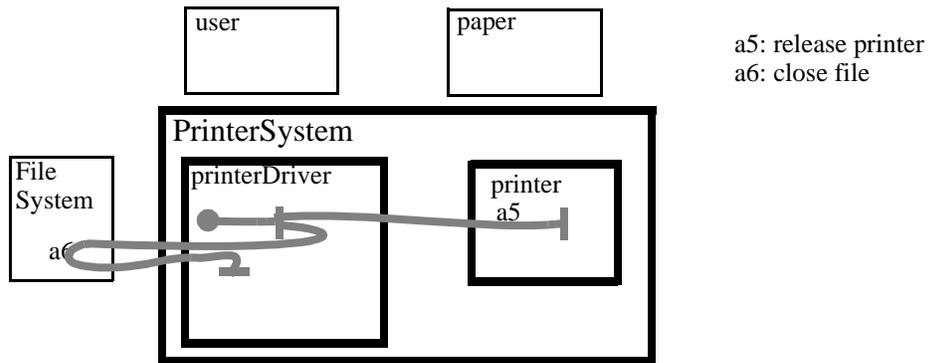
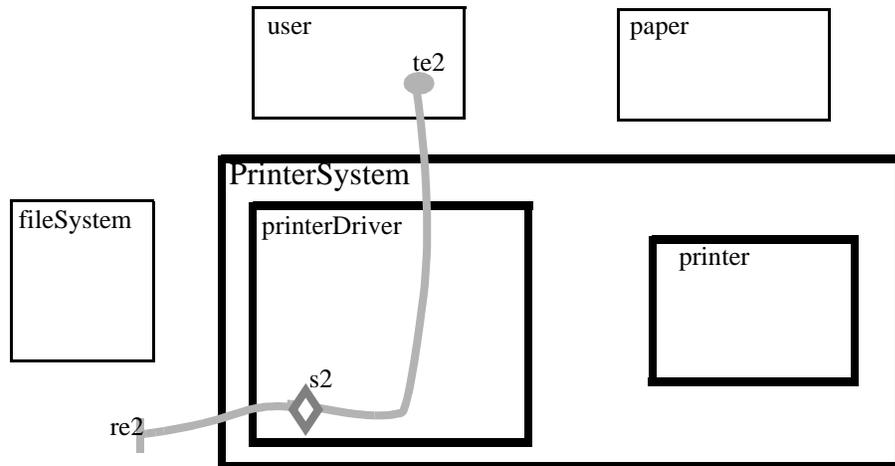


FIGURE 117. StopPrinting UCM



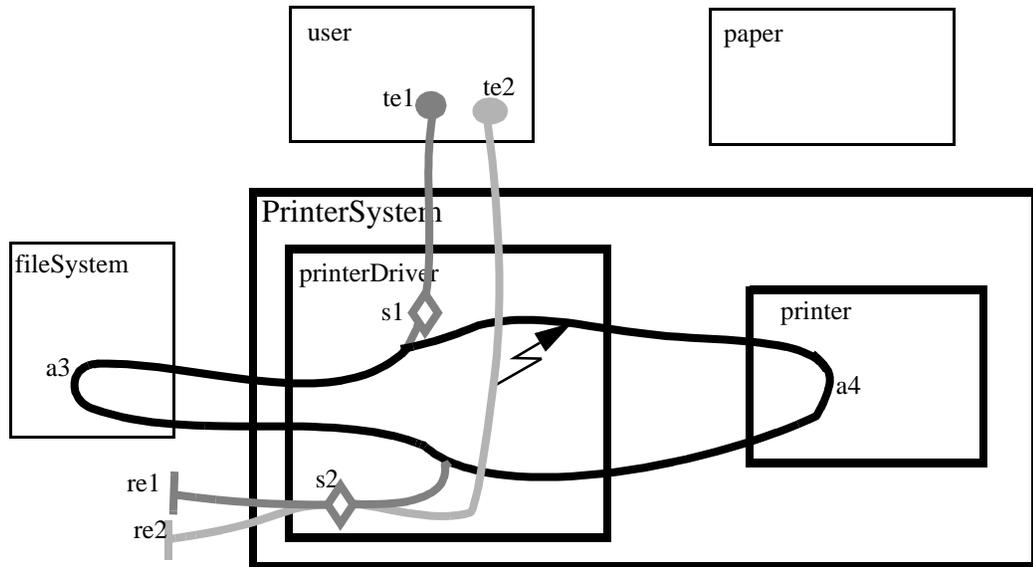
te2: stop printing (pre-condition: printing state)
 re2: printing stopped (post-condition: the system is returned to the idle state)
 s2: release resources

Define Scenario Interaction

Figure 118 illustrates the global compound UCM map of PrinterSystem. In this UCM, the two scenarios that compose PrinterSystem are composed together in a single diagram. The interrupt printing loop responsibility previously described in prose is now visually expressed using the abort symbol (⚡). The abort, in this case, expresses the

interaction between the two paths. Also, since both paths contain the release resources stub, the two paths have been coupled at the `releaseResources` stub location.

FIGURE 118. Composite UCM Map of the PrinterSystem



- te1: start printing (pre-condition: the system is in the idle state)
- te2: stop printing (pre-condition: printing state)
- re1: printing completed (post-condition: the system returns in the idle state)
- re2: printing stopped (post-condition: the system is returned to the idle state)
- s1: get resources stub (request printer, open file)
- s2: release resources stub (release printer, close file)
- a3: get next character
- a4: print character

Final UCM Model of PrinterSystem

The UCM model is at this point completed. The composite UCM of Figure 118, together with the two UCM stubs described in Figure 115 and Figure 116, form the complete UCM model of `PrinterSystem`. Once validated, this model is used to define the MSC model.

5.4 Transition from UCM to MSC (Iteration 1)

We now define the specification MSC model. This modeling phase proceeds in two steps:

1. Generation of MSC Skeletons
2. Definition of Message Sequences

In the following three sections we apply these steps.

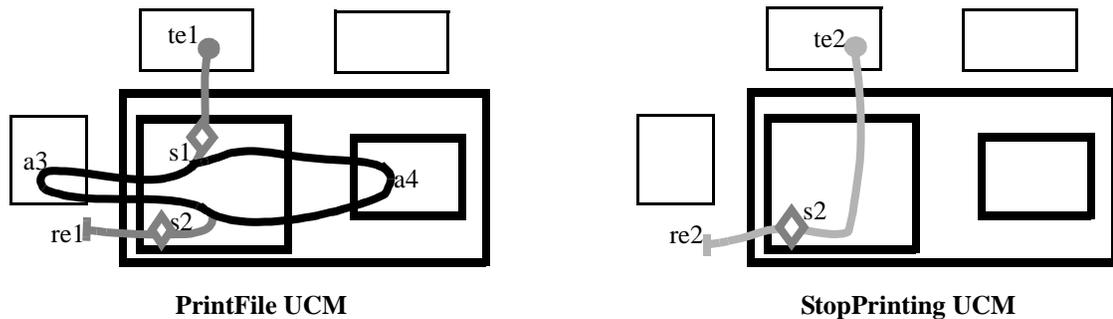
5.4.1 Generation of MSC Skeletons

This step is composed of two steps:

- Generation of HMSCs
- Generation of Basic MSC Skeletons

The input for the MSC phase is the set of UCM related path sets described in section 5.3. PrinterSystem is composed of two UCM related path sets: the PrintFile UCM related path set and StopPrinting related path set. The two related path sets are illustrated in Figure 119.

FIGURE 119. Related path sets of the PrinterSystem UCM model



5.4.1.1 Generation of HMSCs

The goal of this step is to generate one HMSC for each related path set contained in the UCM model. Hereafter, we discuss the generation of a HMSC for each of them.

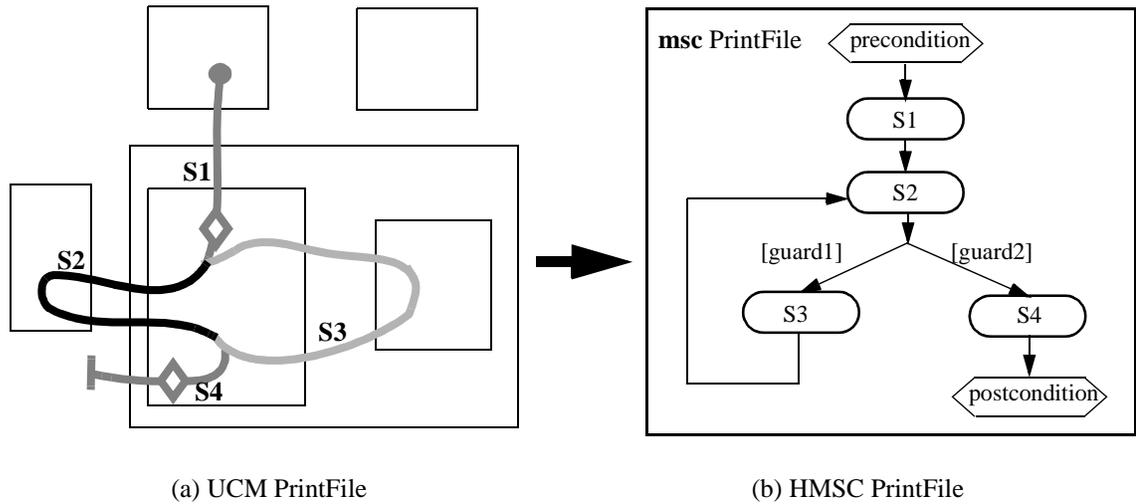
PrintFile HMSC

The generation of the PrintFile HMSC from the UCM given in Figure 114 is illustrated in Figure 120. We observe in this diagram that:

- A MSC reference (rounded rectangles) is created for each path segment contained in the use case map.
- The structure of the UCM path segments is expressed in the resulting HMSC.
- The pre and post conditions associated with the UCM path (or more generally with the UCM related path set) are explicitly introduced in the system model at this stage.

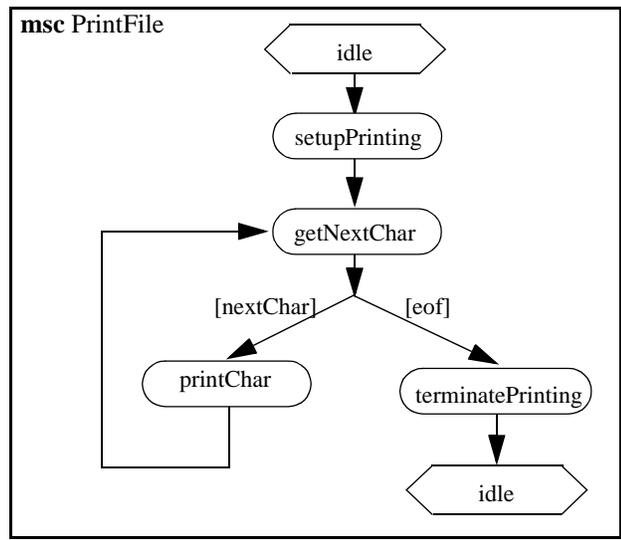
Also, we observe that the component information (i.e the component context diagram), contained in the UCM is not present in the HMSC.

FIGURE 120. Generation of the PrintFile HMSC



In order to obtain a more expressive MSC model, we rename the different UCM path segments (and consequently the different MSC references). Figure 121 gives the HMSC obtained from Figure 120 by renaming the different path segment according to the role they play in the overall scenario expressed by the UCM. The new identifier given in this figure will be used from now on, in this section, to refer to the different path segments and corresponding basic MSC.

FIGURE 121. Renamed PrintFile HMSC



StopPrinting HMSC

Since the StopPrinting UCM is composed of a single path segment, the generation of a HMSC is not required. In such a case, we directly generate a basic MSC from the UCM.

5.4.1.2 Generation of Basic MSC Skeletons

In the previous step, a HMSC has been produced for the PrintFile UCM. This HMSC contains four MSC references. The goal of this step is to produce basic MSC skeletons. In the context of PrinterSystem, we need to create seven basic MSCs: four for the PrintFile UCM, i.e. one per path segment (or one per MSC reference contained in the HMSC), one for the StopPrinting UCM, and one for each stub.

The resulting basic MSC skeletons are given in Figure 122, Figure 123, Figure 124, Figure 125, Figure 126, Figure 127, and Figure 128. The basic MSC skeletons given in these figures are composed of the five components that compose PrinterSystem and its environment. Also, PrinterSystem, which is composed of printerDriver and printer components, is delimited from its environment, which is composed of user, fileSystem, and paper components.

In these figures, we observe that:

- One component instance is created in the MSC for each component defined in the UCM related path set to which the MSC relates.
- Each UCM responsibility is placed on the axis of the component to which it has been allocated in the UCM model.
- The identifiers (labels) used in the different basic MSC skeletons are the same as the ones used in the UCMs. This ensures traceability between MSC and UCM.

In the following of this section, we separately describe the basic MSC skeletons associated with the PrintFile scenario and the one associated with the StopPrinting scenario.

PrintFile Basic MSC Skeleton

In Figure 122, Figure 124, Figure 125, and Figure 126, a basic MSC skeleton is provided for each MSC reference contained in the HMSC of Figure 121. Also, the basic MSC skeletons generated from the two UCM stubs are illustrated in Figure 123 and Figure 127.

In Figure 122, the `setupPrinting` basic MSC skeleton is given. In this basic MSC, an idle initial state is defined according to the precondition of the `setupPrinting` segment of the `PrintFile` UCM. Also, a triggering event message (`print`) arrow is defined. Since the `PrintFile` scenario is initiated by the user, this arrow is connected to `user` component on its sender side. However, since the receiver component within the `PrinterSystem` has not been yet defined, the receiver side of the arrow is connected to the `PrinterSystem` frame. The data type `FileName` associated with the `print` message is defined in the message data box located below the basic MSC skeleton.

We also observe that the UCM stub `GetResource` is represented in the MSC skeleton by means of a MSC reference. The MSC skeleton that corresponds to the `GetResource` stub is given in Figure 123. The stub `GetResource` is composed of two responsibilities, `a1` and `a2`, executed sequentially.

FIGURE 122. `setupPrinting` basic MSC skeleton

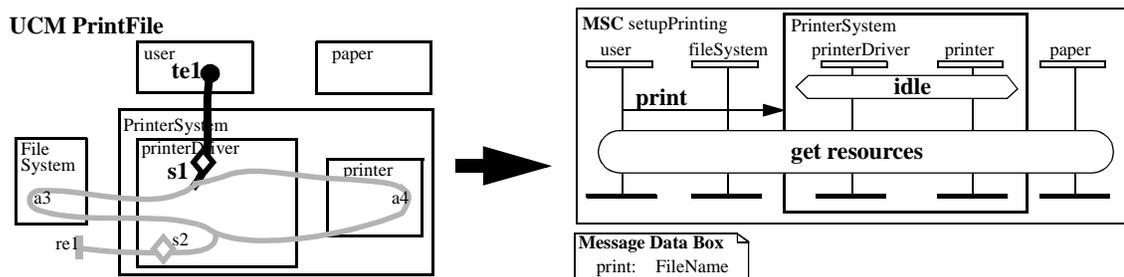
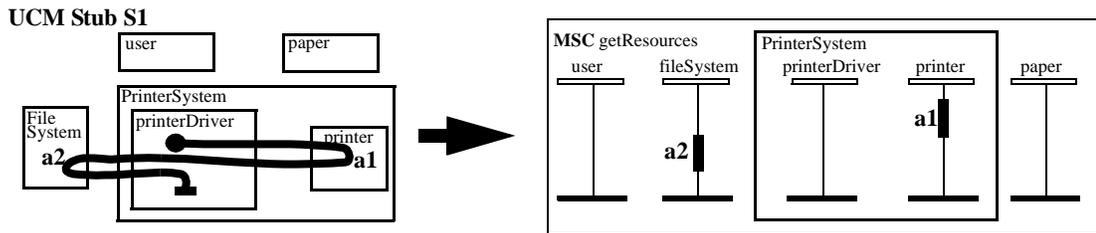


FIGURE 123. getResources basic MSC skeleton



In Figure 124 and Figure 125, the `getNextChar` and `printChar` basic MSC skeletons are given. These two basic MSC skeletons only contain a single responsibility, which are respectively `a3` and `a4`.

FIGURE 124. getNextChar basic MSC skeleton

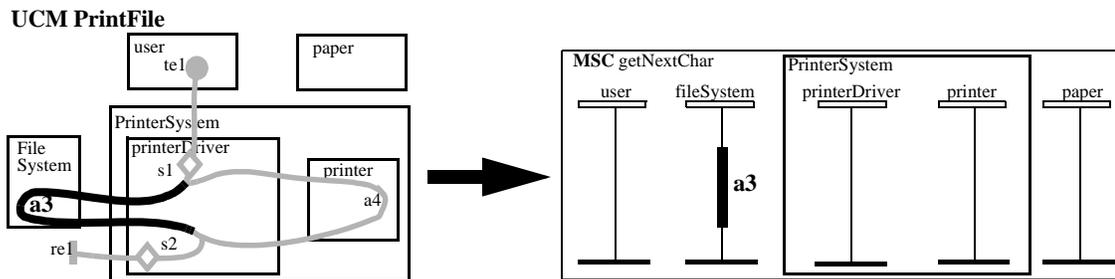
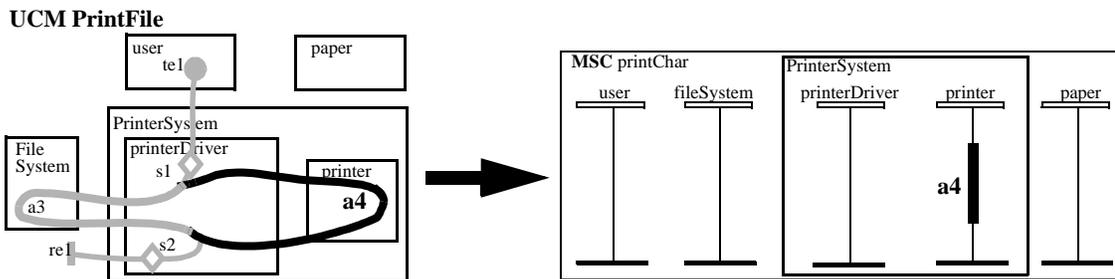


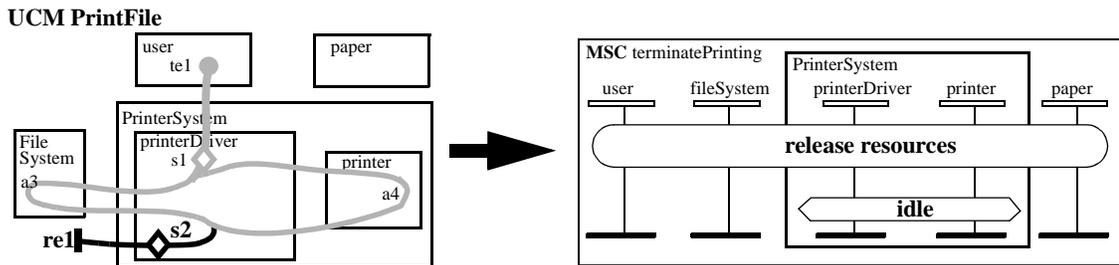
FIGURE 125. printChar basic MSC skeleton



In Figure 126, the `terminatePrinting` basic MSC skeleton is given. In this basic MSC skeleton, we observe that the UCM stub is mapped onto a MSC reference. We also observe that even if this path segment corresponds to a UCM path segment that contains an end bar, no resulting event message arrow is defined. That is because the execution of

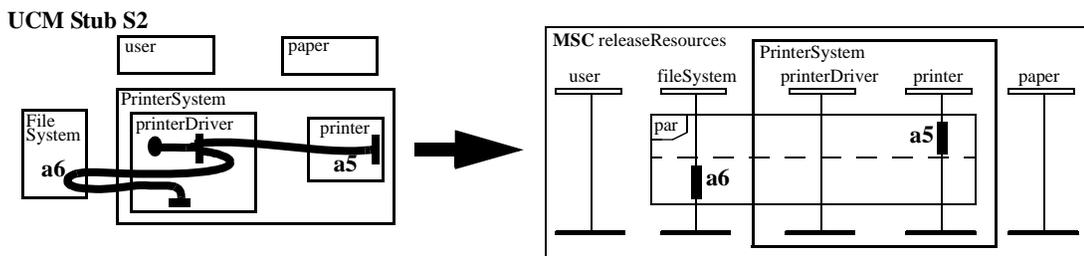
the PrintFile scenario does not terminate by sending back a resulting event. Finally, an idle terminating state is defined in the basic MSC skeleton according to the postcondition of the terminatePrinting segment of the PrintFile UCM.

FIGURE 126. terminatePrinting basic MSC skeleton



The `releaseResources` stub is described in a separate basic MSC skeleton (Figure 127). In this basic MSC skeleton, the two parallel path segment of the `releaseResources` UCM map (Figure 116) are expressed using the MSC parallel inline expression (box with “par” indicated in the top left corner).

FIGURE 127. releaseResources Basic MSC skeleton



StopPrinting Basic MSC Skeleton

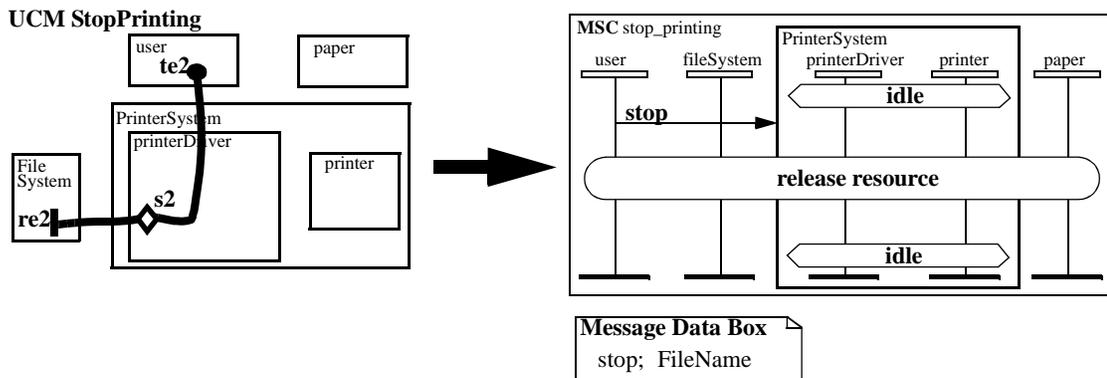
In Figure 128, the basic MSC skeleton generated from the `StopPrinting` UCM (Figure 117) is given. The `StopPrinting` MSC is initiated by the `user` sending of a `stop` message to `PrinterSystem`. Then, the `printerDriver` initiate the execution of the

releaseResources stub. This UCM stub is the same as the one executed in the terminatePrinting MSC (see the description of the terminatePrinting MSC for details).

As in the case of the setupPrinting basic MSC skeleton, the triggering event message arrow (stop) is connected on its sender side to user, and is connected on its receiver side to the frame of PrinterSystem.

We observe that, in this case, the system states (initial and final states), which correspond to the UCM pre and postconditions, are defined in the basic MSC skeleton, while in the previous case (PrintFile MSC) they were defined in the HMSC. Because in this case there is no HMSC, we define the system states in the basic MSC.

FIGURE 128. StopPrinting basic MSC skeleton



5.4.2 Definition of Message Sequences (Iteration 1)

The goal of the message sequence definition step is to express each of the responsibilities contained in the basic MSC skeletons in terms of message sequences. The result of this step is the production of the specification MSC model of the system. This MSC model only contains components, system states, and message arrows.

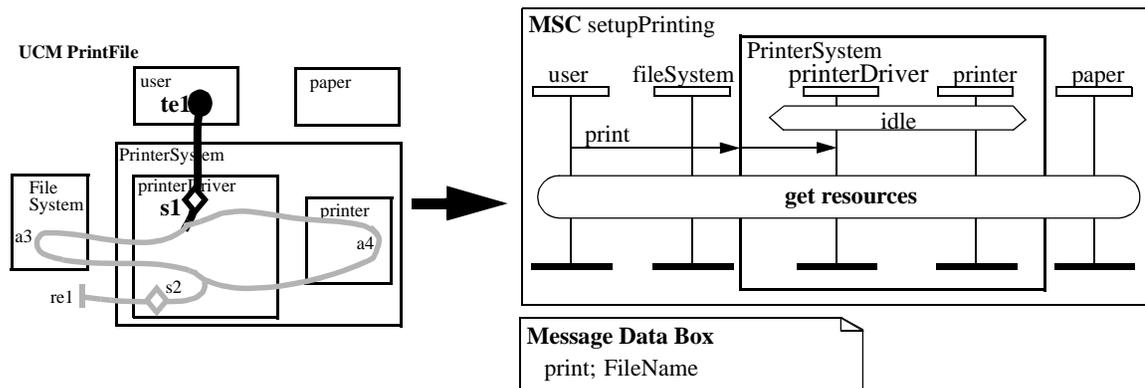
In this section, we illustrate the definition message sequence for the two PrinterSystem scenarios: PrintFile and StopPrinting.

PrintFile MSC

In Figure 129, Figure 130, Figure 131, Figure 132, and Figure 133 the different UCM responsibilities expressed in the basic MSC skeleton of Figure 122, Figure 123, Figure 124, Figure 125, Figure 127, and Figure 126 are refined and described by means of message sequences.

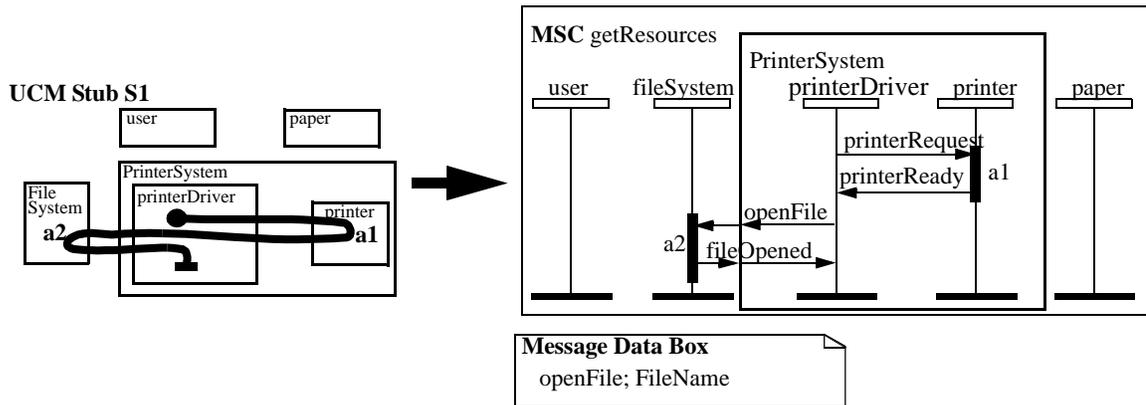
In the `setupPrinting` basic MSC given in Figure 129, the triggering event message arrow, on which the `print` message is sent, has been connected to the `printerDriver`.

FIGURE 129. `setupPrinting` basic MSC



In Figure 130, the `getResources` basic MSC is illustrated. In this figure, responsibility `a1` (`requestPrinter`) is refined as two successive messages: a `printerRequest` message sent from `printerDriver` to `Printer`, and a `printerReady` message sent from the `printer` to the `printerDriver`. Responsibility `a2` (`openFile`) is also refined as two successive messages: an `openFile` message sent from `printerDriver` to `fileSystem`, and a `fileOpened` message sent from the `fileSystem` to the `printerDriver`.

FIGURE 130. getResources basic MSC



In the getNextChar basic MSC given in Figure 131, responsibility a3 (get next char) is refined as a readNextChar sent from the printerDriver to the fileSystem, followed by either a nextChar message or a eof message sent by the fileSystem to the printerDriver. The alternative between the two possible messages is expressed using the MSC inline alternative expression (box with “alt” indicated in the top left corner). The inline alternative expression contains, in this case, two alternatives: one for the nextChar message, and one for the eof message.

As expressed in the HMSC of Figure 121, the basic MSC that will be executed after the getNextChar MSC depends on the message sent back by the fileSystem. If a nextChar message is sent back then the printChar MSC will be executed, otherwise if an eof message is sent back the terminatePrinting MSC will be executed.

FIGURE 131. getNextChar basic MSC

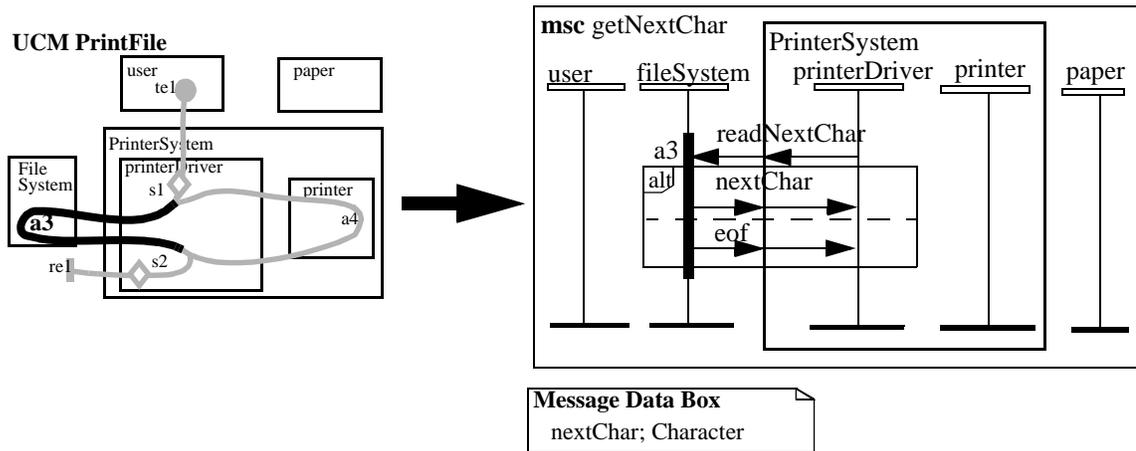


Figure 132 gives the refinement of responsibility a4 (print character). This responsibility is refined by two successive messages: a print message sent by printerDriver to printer followed by a char message sent from printer to paper.

FIGURE 132. printChar basic MSC

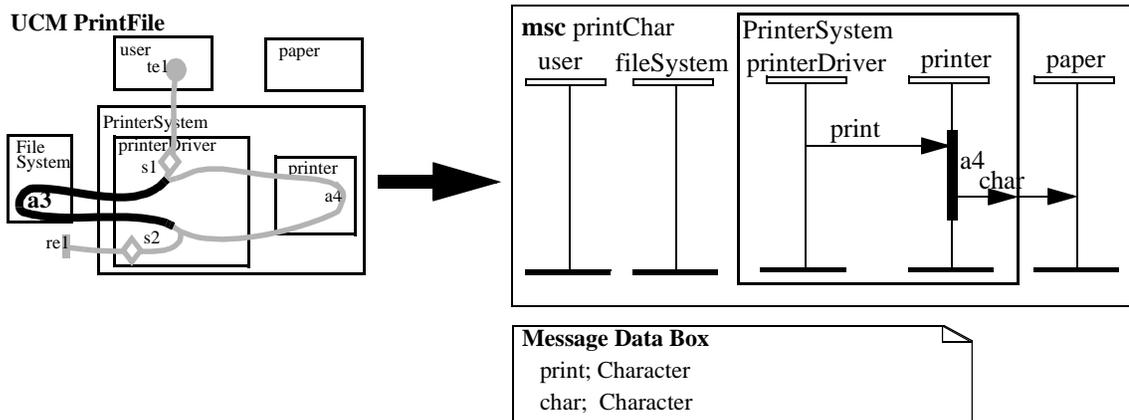
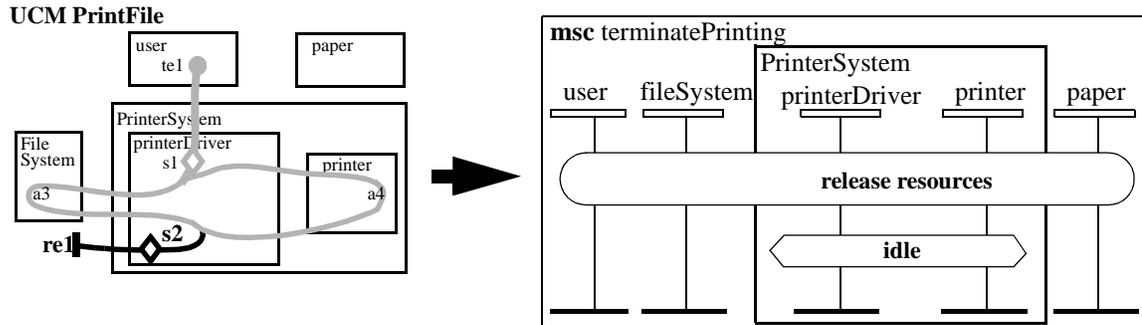


Figure 133 gives the terminatePrinting basic MSC. Since there were no responsibilities in the terminatePrinting basic MSC skeleton, this basic MSC is identical to the basic MSC skeleton (Figure 126).

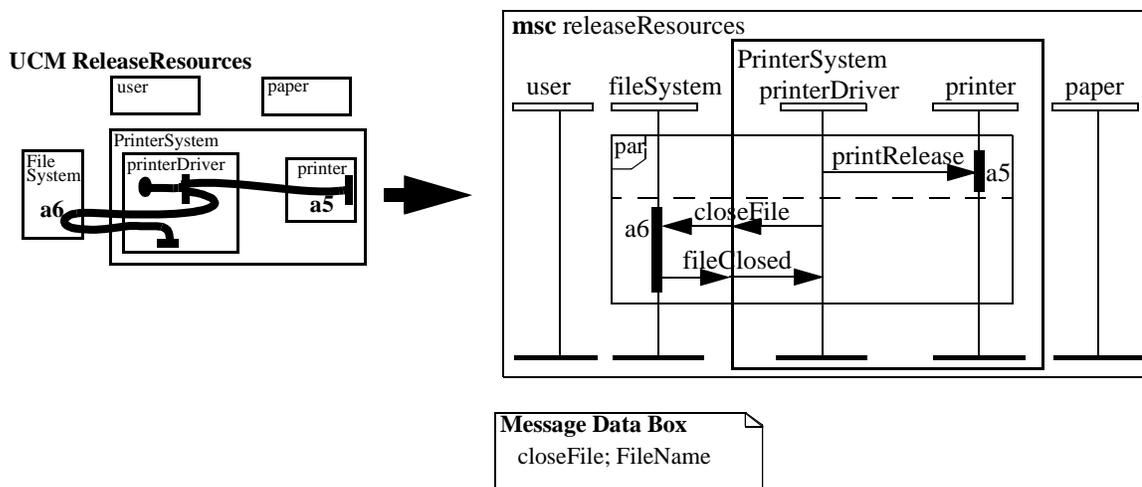
FIGURE 133. terminatePrinting basic MSC



The HMSC of Figure 121 together with the set of basic MSCs described in Figure 129, Figure 131, Figure 132 and Figure 133 constitute the first version of the PrintFile MSC.

In Figure 134, the releaseResources basic MSC is illustrated. In this figure, responsibility a5 is refined as a single printerRelease message sent from printerDriver to Printer, while responsibility a6 is refined by a closeFile message sent from printerDriver to fileSystem, followed by a fileClosed message sent back from fileSystem to printerDriver. We recall that these two responsibilities are executed in parallel.

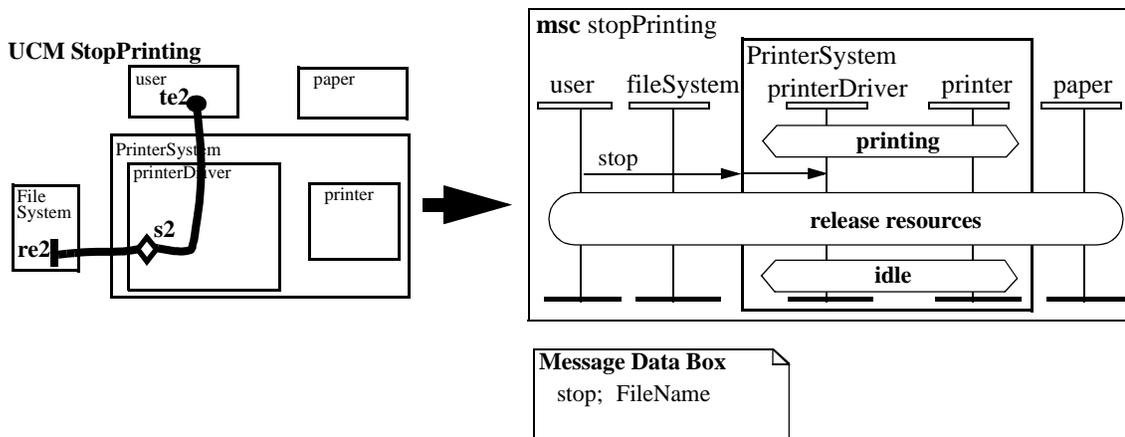
FIGURE 134. releaseResources basic MSC



StopPrinting MSC

In Figure 135, the triggering event message arrow of the StopPrinting basic MSC skeleton is connected to printerDriver. The resulting MSC constitutes the first version of the StopPrinting MSC. The releaseResources basic MSC is given in Figure 134.

FIGURE 135. stopPrinting basic MSC



5.5 Specification MSC Modeling (Iteration 1)

In this case study, because of the simplicity of PrinterSystem, we do not need to introduce any new component at this stage. There is no need to restructure the MSC model neither.

5.6 From MSC to ROOM Structure (Iteration 1)

At this stage, we need to define the ROOM structure of `PrinterSystem`. The goal of this step is to produce a ROOM role structure for each related path set¹ defined in the specification MSC model.

The definition of ROOM role structures is conducted in four steps:

1. Definition of role actors.
2. Identification of communication.
3. Definition of protocol classes.
4. Definition of the contracts.

In this section, we use these three steps to produce the ROOM role structures required for the execution of the `PrintFile` MSC (or scenario) and the `StopPrinting` MSC (scenario).

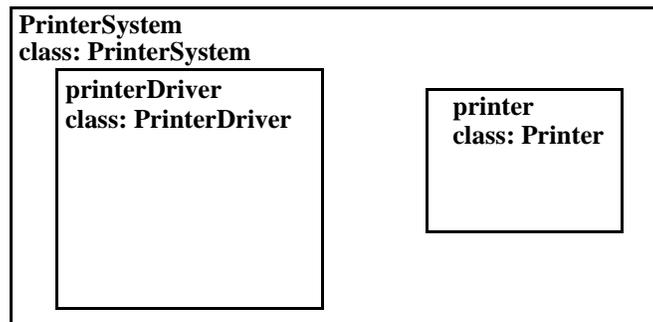
PrintFile

Definition of Role Actors

In this step, we define the set of role actors that compose the system under development, which in the present case is `PrinterSystem`. The external entities are left out.

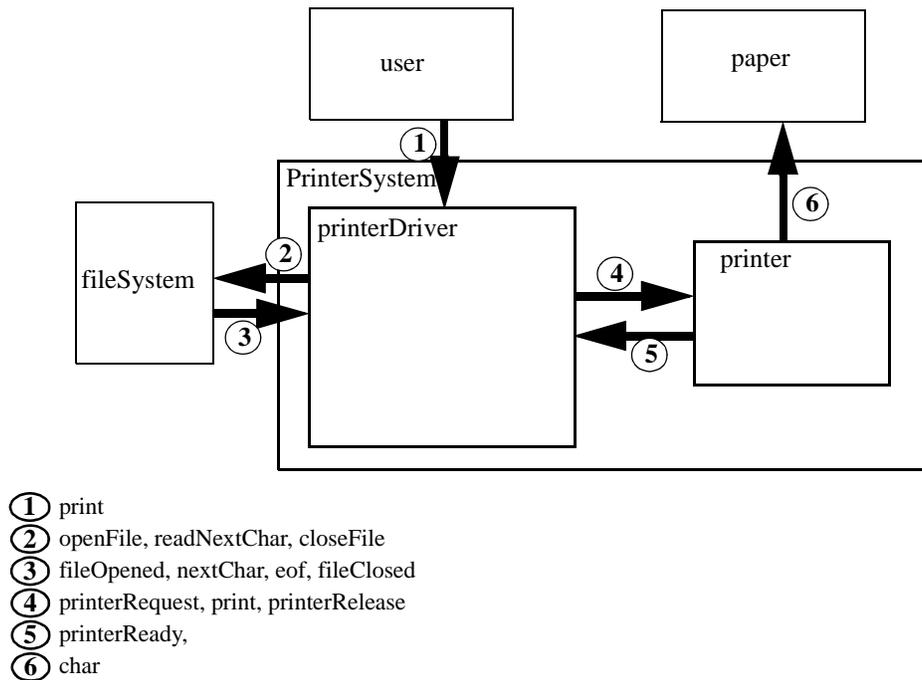
In Figure 136, the role actors of the `PrintFile` scenario are given.

1. By related path set we mean the MSC that is associated with a UCM related path set. This MSC is a HMSC associated with a set of basic MSCs if the UCM related path set it represents contains more than one paths segment, or a single basic MSC otherwise.

FIGURE 136. printFile role actors

Communication Identification

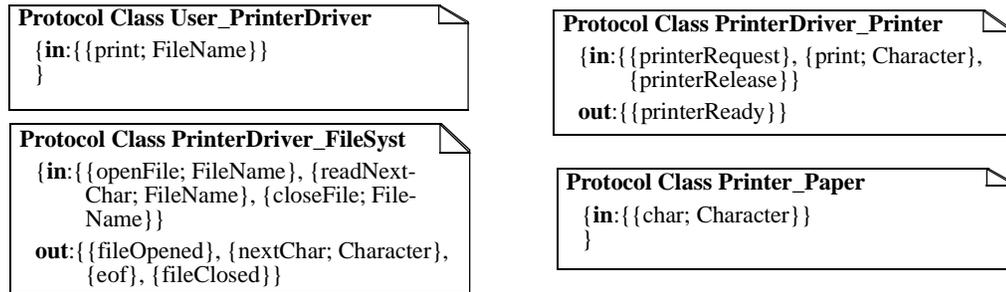
In this step, we capture the inter-component communication required for the execution of the PrintFile MSC (figures 129 to 133) in a communication diagram. The resulting communication diagram is given in Figure 137. In this case, the execution of the MSC requires communication between: user and printerDriver (one-way communication), printerDriver and fileSystem (two-way communication), printerDriver and printer (two-way communication), and printer and paper (one-way communication). The messages involved in the different communications are given at the bottom of Figure 137.

FIGURE 137. printFile communication diagram

Protocol Class Definition

We now define the set of protocol classes that will be used to define the different contracts required in the PrintFile ROOM role structure. The main decisions that need to be taken at this stage concern the logical grouping of the different messages identified in the communication diagram of Figure 137 into protocol classes.

In the case of the PrintFile scenario, we decide to define only one protocol class per pair of communicating components. Therefore, we group all the messages exchanged between the two components in one protocol class. Thus, four protocol classes are created. The resulting protocol classes are given in Figure 138.

FIGURE 138. Definition of printFile protocol classes

Contract Definition

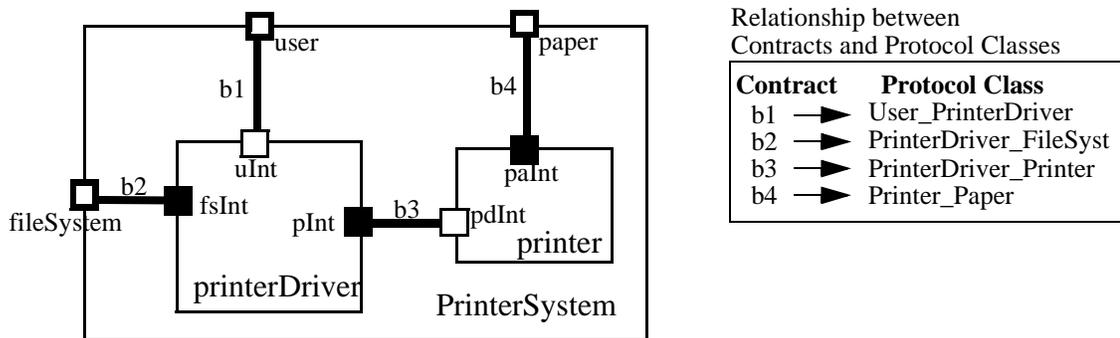
A ROOM structure is composed of a set of components (actors) and a set of contracts that bind components together in a communication topology. At this stage, the set of system components is already defined. Therefore, contracts are the only missing model elements to the ROOM structure. Moreover since the required protocol classes have been defined in the previous step, we have all the information needed to completely define the ROOM structure associated with the PrintFile MSC.

The goal of this step is to use the different protocol classes defined in Figure 138 to define the required contracts. In this case, we define four contracts: one between `user` and `printerDriver`, one between `printerDriver` and `fileSystem`, one between `printerDriver` and `Printer`, and one between `printer` and `paper`.

The resulting ROOM structure is given in Figure 139 together with the relationship that exists between the different contracts and protocol classes defined in the previous step (Figure 138). In this ROOM structure, the relay ports of `PrinterSystem` are labelled using the name of the external (environment) components to which they correspond, and the different contracts are labelled as `b1`, `b2`, `b3` and `b4`. Also, the ports of the internal components, i.e. `printerDriver` and `Printer`, are labelled according to the component to which they are connected. Thus, the ports of the `printerDriver` are labelled `ulnt` (**u**ser **I**nterface), `fsInt` (**f**ile**S**ystem **I**nterface) and `plnt` (**p**rinter **I**nterface), and the ports of the `printer` have been labelled `palnt` (**p**aper **I**nterface) and `pdInt` (**p**rinter**D**river **I**nterface).

We observe that the environment components (`user`, `fileSystem` and `paper`) that were illustrated in the previous models of the system are no longer illustrated in the ROOM structure of `PrinterSystem`. That is because from this step on, we shift the focus to `PrinterSystem` itself. The environment components were illustrated in the previous models of the system mainly to help to understand the relationships (communication) that exist between `PrinterSystem` and its environment. As a result, the different protocol classes that are required to enable the communication between `PrinterSystem` and its environment have been defined. We now concentrate on the development of `PrinterSystem` itself and its internal components.

FIGURE 139. ROOM role structure required for the `printFile` UCM



StopPrinting

Definition of Role Actors

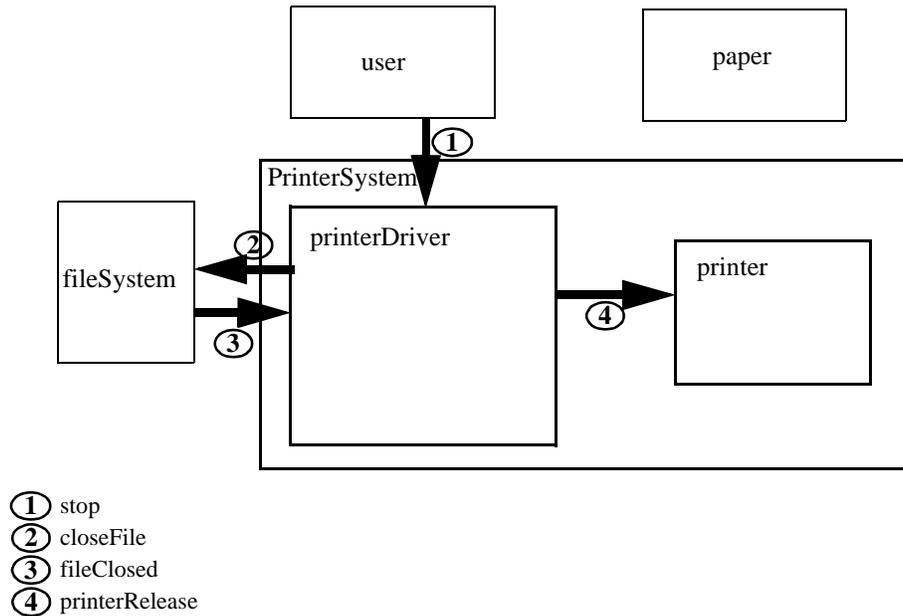
The ROOM actors that play roles in the `StopPrinting` MSC are the same that the ones defined for the `PrintFile` MSC (Figure 136).

Communication Identification

The communication diagram of Figure 140 captures the inter-component communication required for the execution of the `StopPrinting` MSC given in Figure 135. In this case, the execution of the MSC requires communication between `user` and `printerDriver` (one-way

communication), printerDriver and fileSystem (two-way communication), and printerDriver and printer (one-way communication). The messages involved in the different communications are given at the bottom of Figure 140.

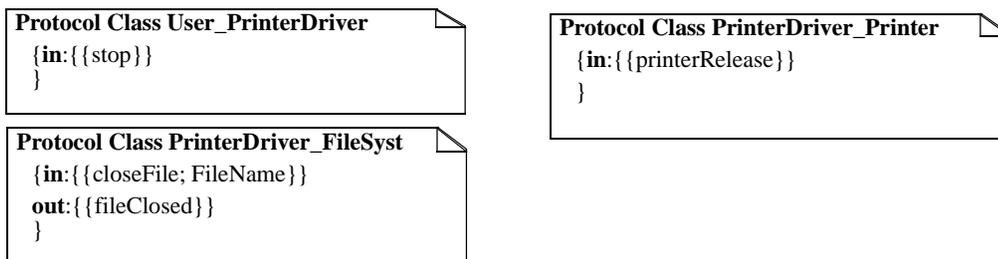
FIGURE 140. stopPrinting communication diagram



Protocol Class Definition

We now define the set of protocol class that will be used to define the different contracts required in the StopPrinting ROOM role structure. As in the case of the PrintFile MSC we define only one protocol class between each pair of communicating components. The resulting protocol classes are given in Figure 141.

FIGURE 141. Definition of stopPrinting protocol classes

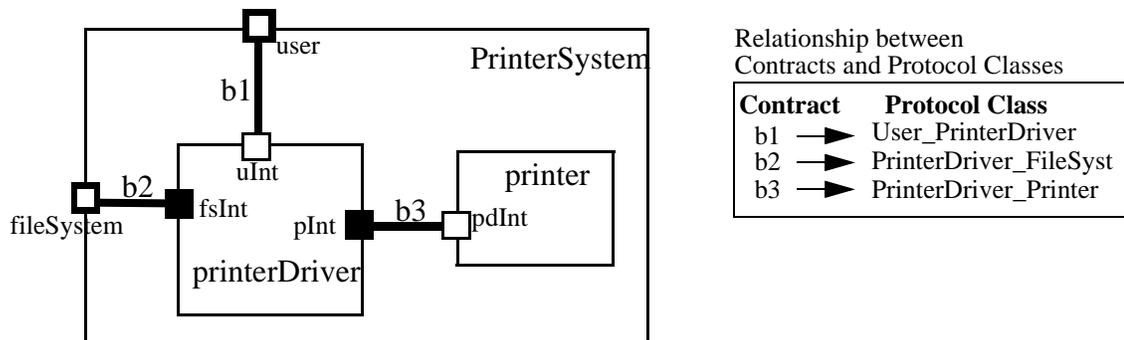


Contract Definition

In this step, we use the protocol classes given in Figure 141, to define the different contracts required to define the ROOM role structure associated with the StopPrinting MSC (Figure 128).

The resulting ROOM structure and the relationship between contracts and the protocol classes defined in Figure 141 are given in Figure 142. The labelling of the different elements that compose the ROOM structure is done in same the way as it was done in the ROOM role structure associated to the PrintFile MSC (Figure 139).

FIGURE 142. ROOM role structure required for the stopPrinting UCM



5.7 ROOM Structure Modeling (Iteration 1)

The objective of this step is to integrate the role structures defined in the previous modeling phase in a global ROOM structure model of the system.

To produce the global ROOM structure, we need to:

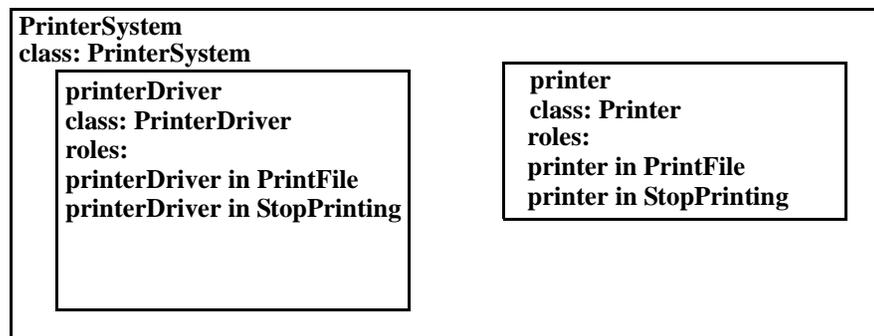
1. Define the set of system actors (the actors are defined in terms of the set of roles they play in the overall system).

2. Define the set of system protocol classes.
3. Define the set of system contracts.

Definition of System Actors

The actors that compose the final ROOM structure are obtained by integrating the different role structures. In this step, an actor is defined in terms of its actor class and the set of roles it plays in the different role structures. The two actors that compose PrinterSystem are given in Figure 143.

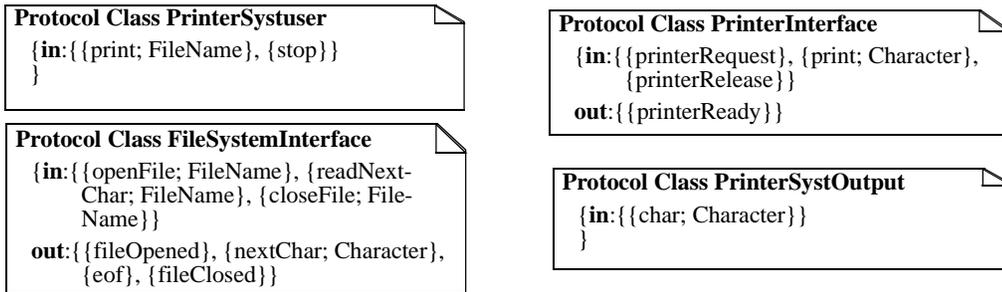
FIGURE 143. PrinterSystem actors



Definition of System Protocol Classes

Using the information contained in the different role protocol classes defined in Figure 5.6, we now define four final protocol classes that will be used to define the global ROOM structure of PrinterSystem.

The resulting set of final protocols are given in Figure 144.

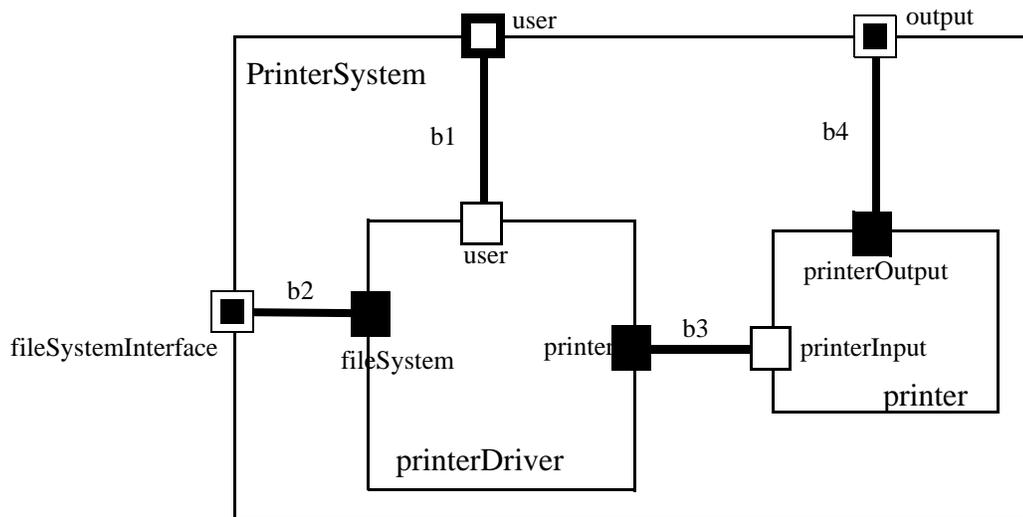
FIGURE 144. PrinterSystem protocol classes

Definition of the Global ROOM Structure

The generation of the global ROOM structure of a system is done by merging together the different role structures defined in the previous modeling phase. In the current case study, two role structures have been defined (Figure 139 and Figure 142). The merging of the two role structures results in the global ROOM structure given in Figure 145. This ROOM structure model constitutes the complete structure of **PrinterSystem**.

The contracts defined in this ROOM structure are based on the set of final protocol classes defined in the Figure 144.

FIGURE 145. PrinterSystem ROOM structure



Relationship between
Contracts and Protocol Classes

Contract	Protocol Class
b1	PrinterSystuser
b2	FileSystemInterface
b3	PrinterInterface
b4	PrinterSystOutput

5.8 Addition of ROOM Structure Information to the MSC model (Iteration 1)

The goal of this step consists in linking the MSC model with the ROOM structure of the system defined in section 5.7. This results in the definition of the customized MSC model of the system. In this step, we update the different basic MSCs defined in the specification MSC model (defined in section 5.4.2) with the contracts defined in the ROOM global structure of the system.

Since the ROOM structure defined in section 5.7 only contains `PrinterSystem` components, i.e. `printerDriver` and `Printer`, we, from this step on, only consider these components in the customized MSC model.

Also, since there is a one-to-one mapping between the roles defined in the specification MSC model and system components defined in the ROOM model, the linkage between MSC components and ROOM system actors is trivial. The `printerDriver` actor of the system plays the roles of the `printerDriver` in both scenarios, and the `printer` actor of the system plays the roles of the `printer` in both scenarios.

In the following of this section, we give the result of updating of the `PrintFile` MSC and `StopPrinting` MSC with the contract information.

PrintFile MSC

In Figure 146, Figure 147, Figure 148, Figure 149, Figure 150, and Figure 151, we give the MSCs obtained by adding the contract information to the MSCs given in Figure 129, Figure 130, Figure 131, Figure 132, Figure 133, and Figure 134. The `terminatePrinting` basic MSC (Figure 150) does not get to be modified because it contains no message arrow.

FIGURE 146. `setupPrinting` customized MSC

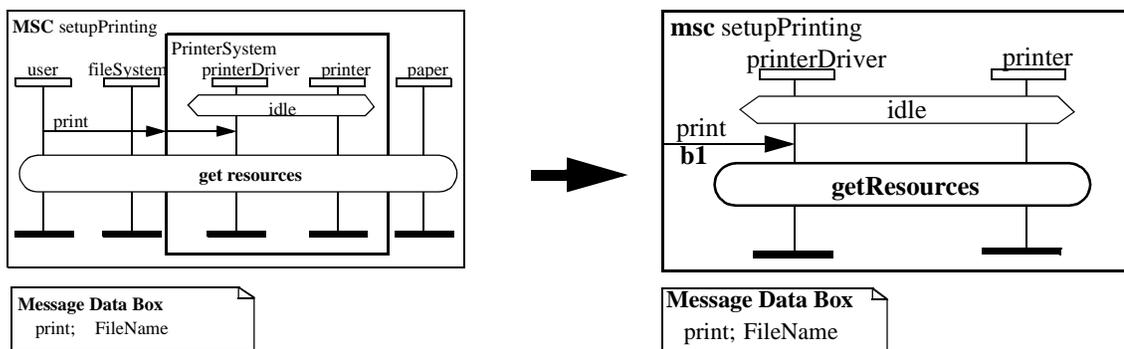


FIGURE 147. getResources customized MSC

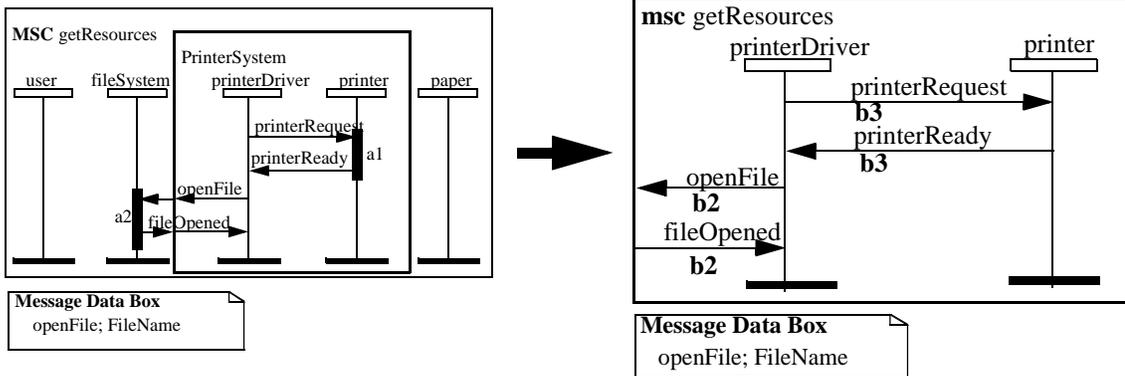


FIGURE 148. getNextChar customized MSC

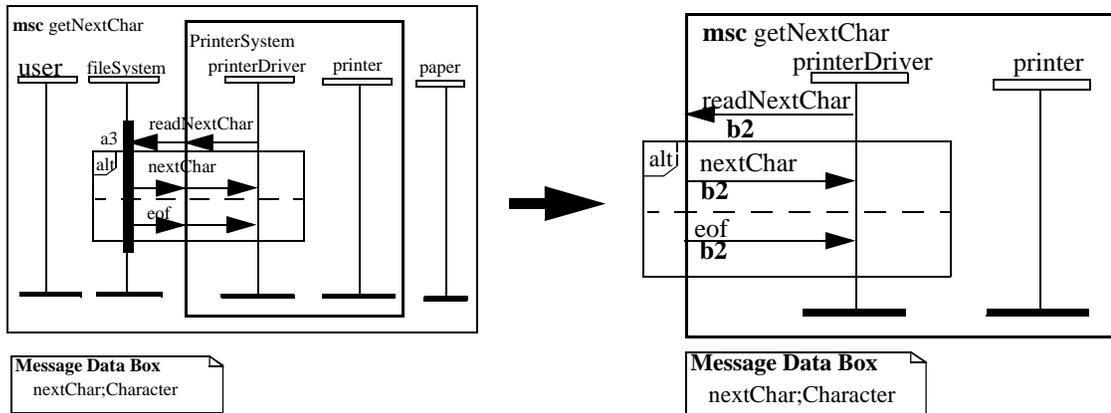


FIGURE 149. printChar customized MSC

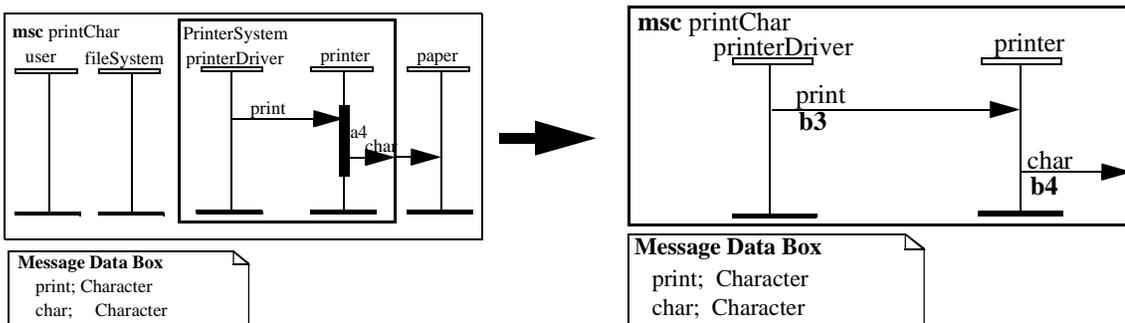


FIGURE 150. terminatePrinting customized MSC

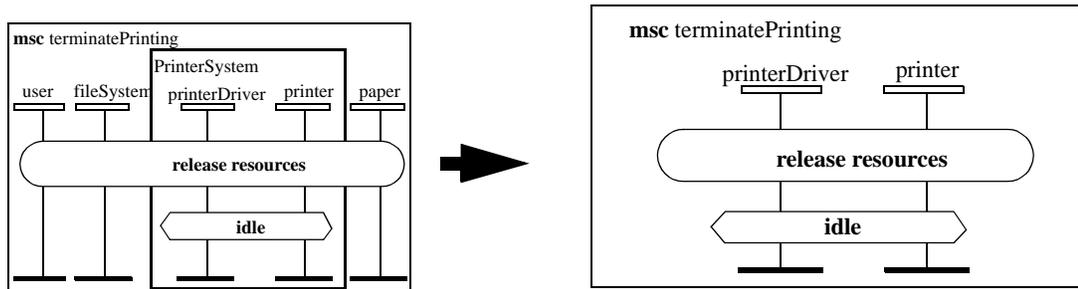
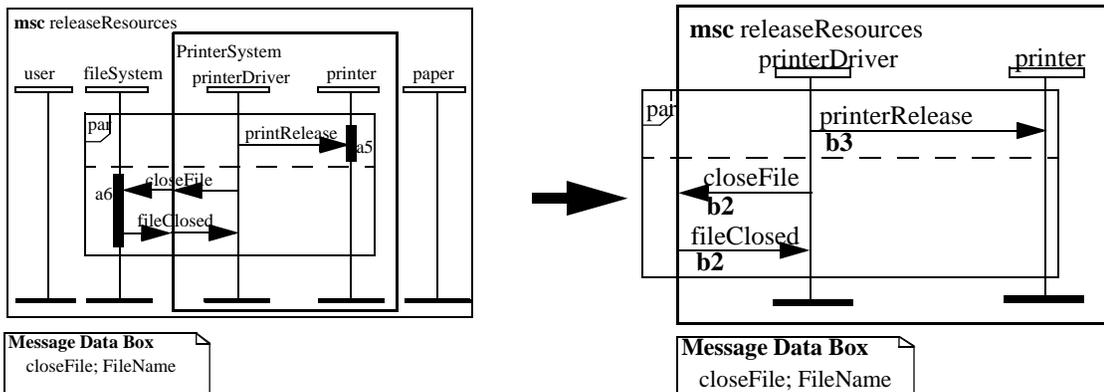


FIGURE 151. releaseResources customized MSC

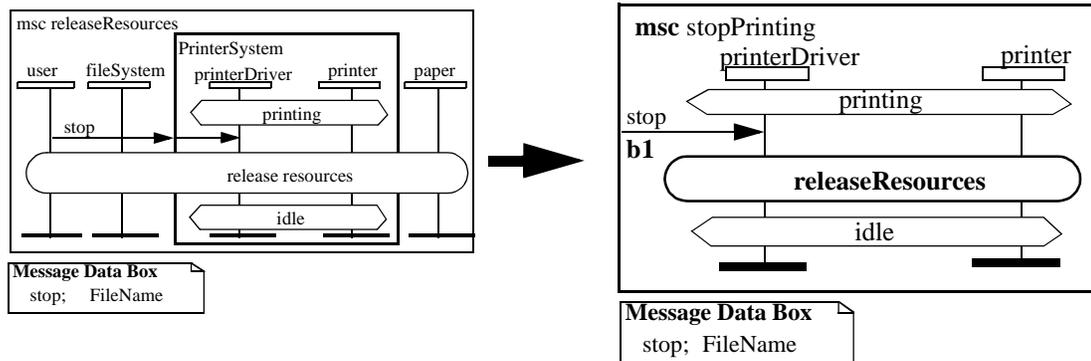


The HMSC of Figure 121 together with the set of basic MSCs described in Figure 146, Figure 147, Figure 148, Figure 149, Figure 150 and Figure 151 constitute the second version of the PrintFile MSC. This version is the one that would be used for the purpose of testing.

StopPrinting MSC

In Figure 152, the customized StopPrinting basic MSC is given.

FIGURE 152. StopPrinting customized MSC



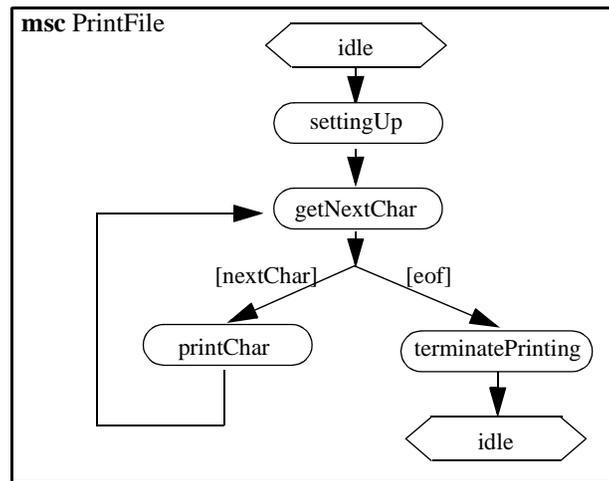
5.9 Component Behavior Modeling in MSC

(Iteration 1)

In this step, we introduce component behavior information in the basic MSCs of the customized MSC model.

For convenience, we reproduce the PrintFile HMSC in Figure 153. (Note that this HMSC is the same as the one given in Figure 121.)

FIGURE 153. PrintFile HMSC



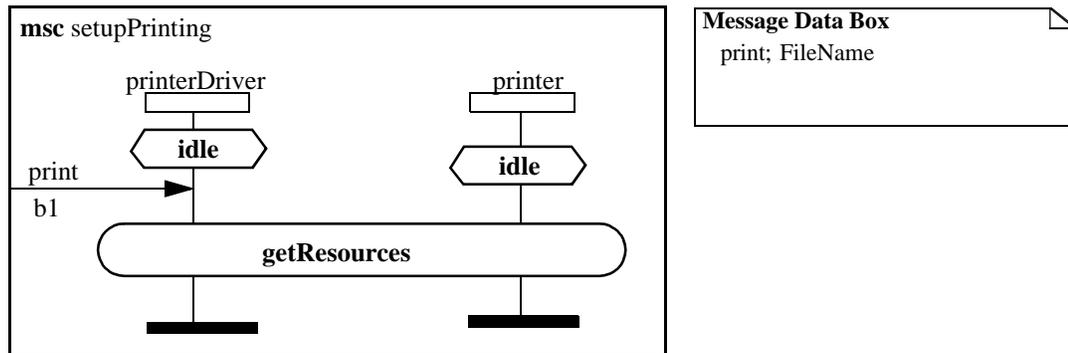
In the following of this section, we individually refine the PrintFile MSC and the Stop-Printing MSC.

PrintFile MSC

In Figure 154, Figure 155, Figure 156, Figure 157, and Figure 159, we give the customized versions of the basic MSCs given in Figure 146, Figure 147, Figure 148, Figure 149, and Figure 151.

In Figure 154, the new version of the `setupPrinting` basic MSC is given. In this basic MSC, the `idle` system state that existed in previous version of the basic MSC is decomposed into two `idle` component states.

FIGURE 154. setupPrinting customized MSC



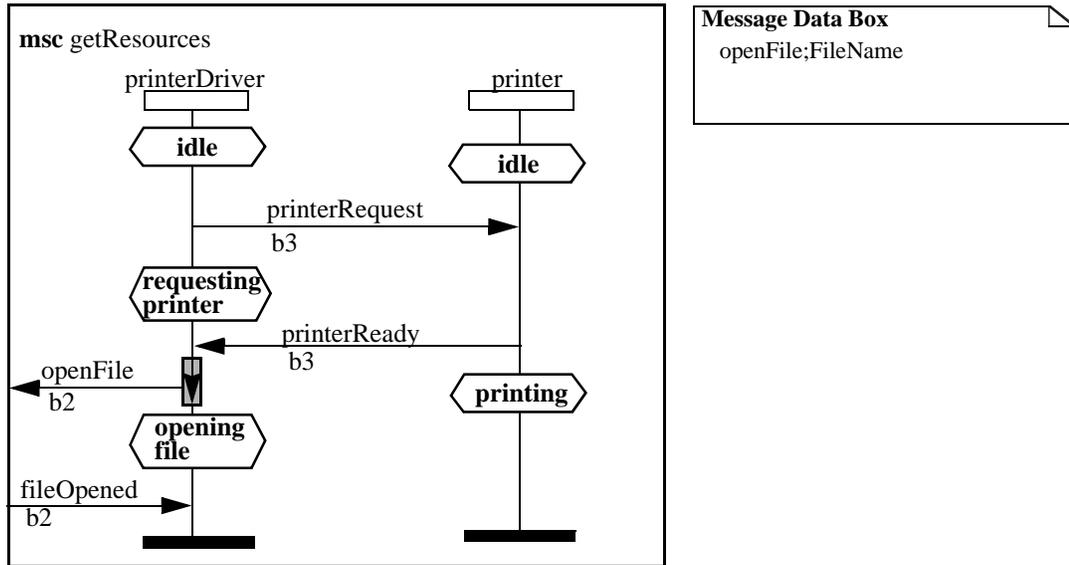
In Figure 155, the new version of the `getResources` basic MSC is given. In this basic MSC, two component behavior elements have been introduced: component states and state entry code.

With respect to states, we first observe that the `idle` global state of `PrinterSystem` that was defined in the previous version of the basic MSC, is decomposed into two `idle` components states, one in `printerDriver` and one in `Printer`. Also, a component state is introduced in components before every incoming message arrow. For example, a `requestingPrinter` state is defined in `printerDriver` just before the `printerReady` message is received.

Also, we observe that, in `printerDriver`, the `openFile` outgoing message has been identified as part of the entry-code () of the `openingFile` state.

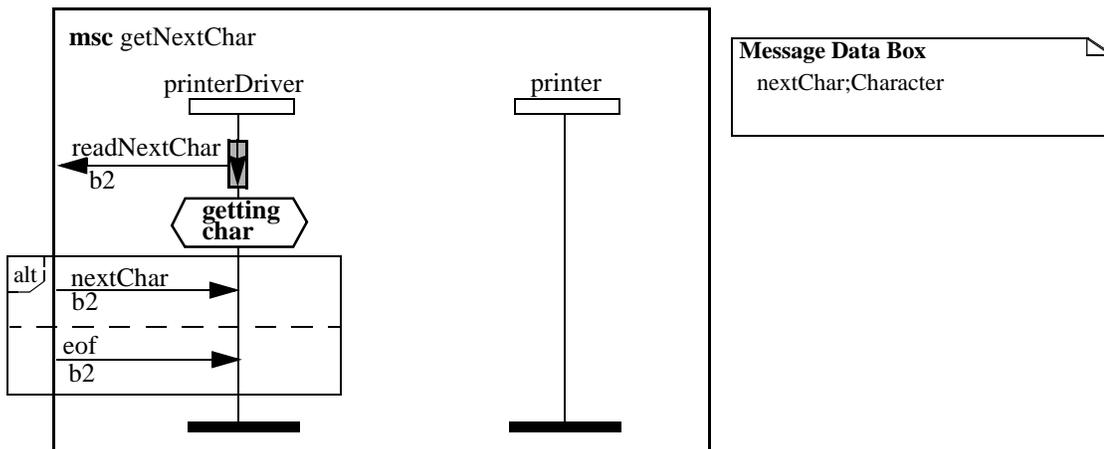
All the other outgoing messages, i.e. the ones that are not explicitly identified neither as entry-code nor as exit-code in the basic MSC, are considered by default to be part of the transition code. In this case, the `printerRequest` message constitutes such a message for `printerDriver`, and the `printerReady` message constitutes such a message for `Printer`.

FIGURE 155. getResources customized MSC



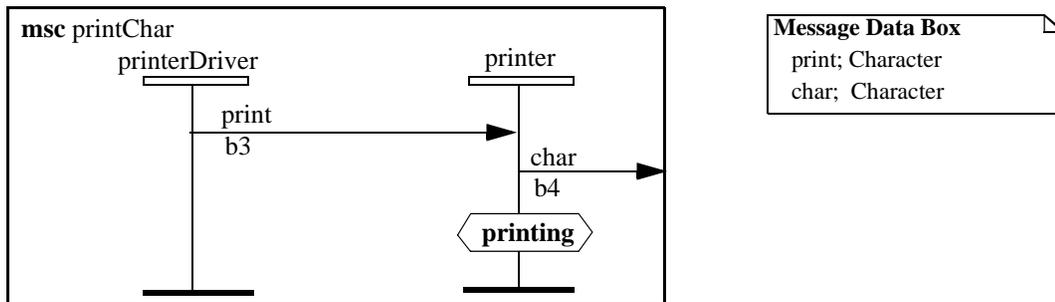
In Figure 156, the new version of the getNextChar basic MSC is given. Since getNextChar only involves interactions between printerDriver and the environment of Printer-System (i.e. the fileSystem component), the customized version of the basic MSC given here only adds detail level elements to printerDriver. In this detail-level basic MSC, a gettingChar state is introduced in printerDriver and the sending of the readNextChar message is identified as entry-code for this state. An alternative fragment (alt) is shown for the printerDriver lifeline, containing messages nextChar, eof, and eof, all associated with the b2 data box.

FIGURE 156. getNextChar customized MSC



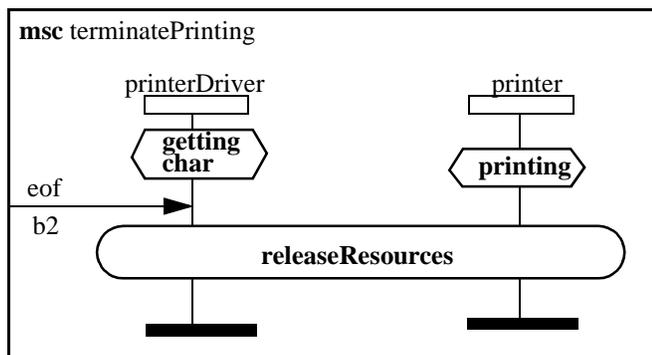
In Figure 157, the new version of the `printChar` basic MSC is given. In this basic MSC, only a printing state is defined in `printer` after the sending of the `char` message. This indicates that, after sending the `char` message, `printer` goes back to the printing state previously defined in Figure 154.

FIGURE 157. `printChar` customized MSC



In Figure 158, the new version of the `terminatePrinting` basic MSC is given. In this basic MSC, the idle system state that existed in previous version of the basic MSC is decomposed into two idle component states.

FIGURE 158. `terminatePrinting` customized MSC

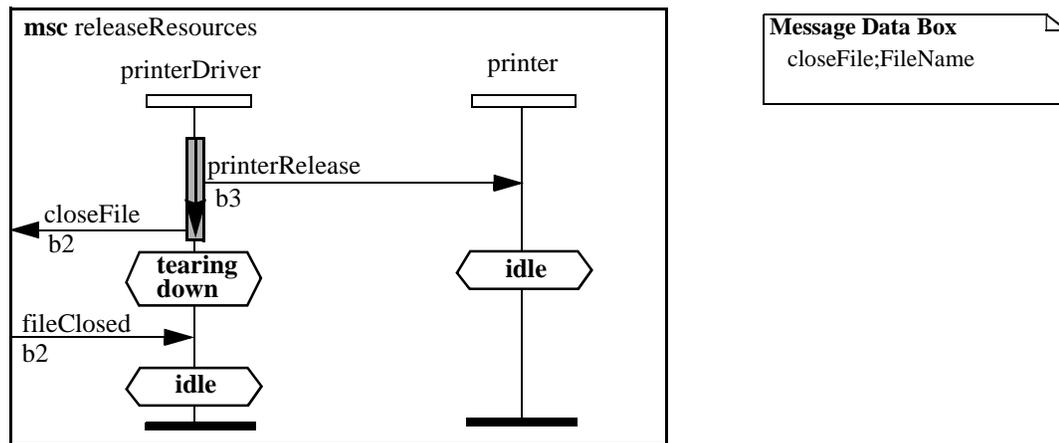


In Figure 159, the new version of the `releaseResources` basic MSC is given. In this basic MSC, component states and entry-code are introduced. In `printerDriver`, a tearing-Down state is defined after the `closeFile` message, and two messages, i.e. `printerRelease` and `closeFile`, are identified as entry-code in the `tearingDown` state. Also, it is

specified that both `printerDriver` and `printer` return to their respective `idle` state after the execution of `terminatePrinting`.

Also, we observe in this figure that the ordering of the `printerRelease` and `closeFile` messages that was non-deterministically defined in the previous basic MSCs using the parallel composition inline expression is now deterministic. The `printerRelease` message is first sent, and then the `closeFile` is sent.

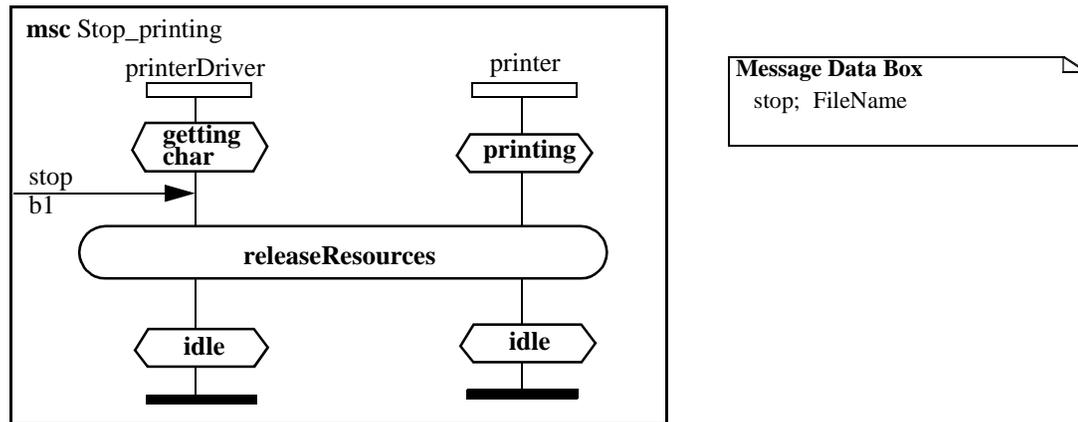
FIGURE 159. releaseResources customized MSC



The HMSC of Figure 153 together with the set of basic MSCs described in Figure 146, Figure 148, Figure 149, and Figure 151 constitute the customized version of the `PrintFile` MSC.

StopPrinting MSC

In Figure 157, the new version of the `StopPrinting` MSC is given. The refinement of this MSC is similar to the one of Figure 151. The only difference concerns the refinement of the printing system state defined in Figure 152 into component states: a `gettingChar` state in `printerDriver` and a `printing` state in `Printer`.

FIGURE 160. StopPrinting customized MSC

This MSC constitutes the customized version of the StopPrinting MSC.

Final MSC Model of the PrinterSystem

The PrintFile MSC and the StopPrinting MSC defined in this step form together the final customized MSC model of PrinterSystem. This model can be used to test the final ROOM model at the end of the iteration.

5.10 From MSC to ROOMChart (Iteration 1)

This step consists in producing actor behavior on a per MSC basis. Thus, for each MSC contained in the final MSC model of PrinterSystem (i.e. the PrintFile MSC and the StopPrinting MSC), we produce a ROOMCharts model of printerDriver and Printer.

In the following of this section, we produce the role behavior models for the PrintFile and StopPrinting MSC. In this step, only the information concerning the grouping (structur-

ing) of states into composite ones is required from the modeler. All the rest of the information that is required to produce the ROOMCharts model is contained in the MSC. Thus, transitions and entry-code defined in the different basic MSCs are mapped directly to ROOM constructs in ROOMCharts.

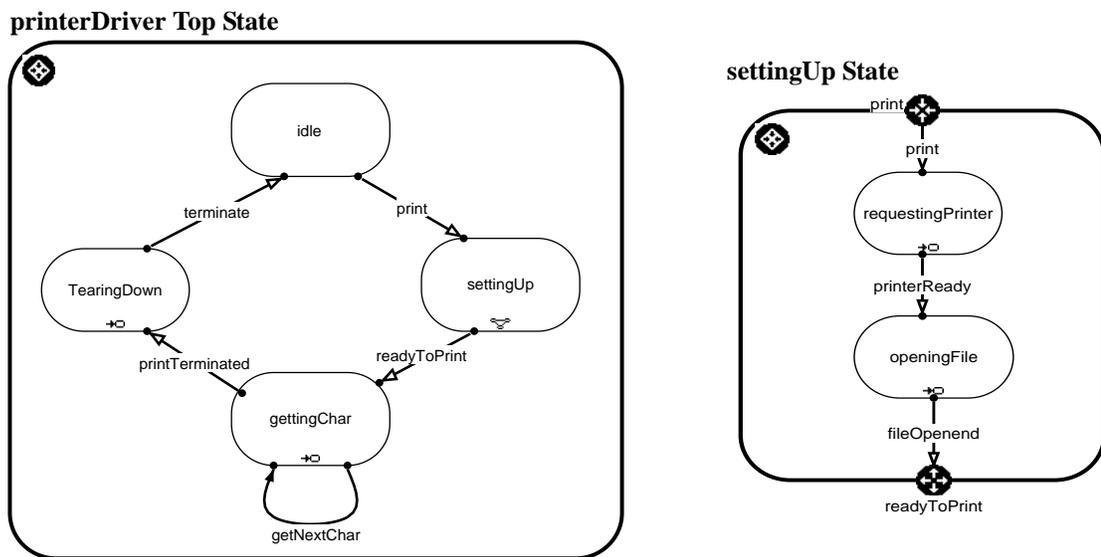
5.10.1 PrintFile

Figure 161 and Figure 162 give the role behavior (ROOMCharts model) of printerDriver and printer in the context of the PrintFile MSC.

printerDriver State Machine

In the printerDriver ROOMCharts model (Figure 161), the requestingPrinter and openingFile states defined in the PrintFile MSC are grouped together in the settingUp composite state. The idle state constitutes both the initial state (precondition) and the final state (postcondition) for the execution of the PrintFile scenario.

FIGURE 161. printerDriver role behavior for the PrintFile MSC



Entry Code

Below, we give the entry-code of the requestingPrinter, openingFile, gettingChar and tearingDown states.

```

state requestingPrinter:
{
  entry action:
  { | |
    SEND printer SIGNAL %printerRequest
    ENDSEND
  }
} /* end of state requestingPrinter */;
state openingFile:
{
  entry action:
  { | |
    SEND fileSystem SIGNAL %openFile
    ENDSEND
  }
} /* end of state openingFile */;
state gettingChar:
{
  entry action:
  { | |
    SEND fileSystem SIGNAL %nextChar
    ENDSEND
  }
} /* end of state gettingChar */;
state TearingDown:
{
  entry action:
  { | |
    SEND printer SIGNAL %printerRelease
    ENDSEND.
    SEND fileSystem SIGNAL %closeFile
    ENDSEND
  }
} /* end of state TearingDown */;

```

printerDriver Transitions

The ROOM description of the transitions defined in the printerDriver top state is given below.

```

transitions: /* of state top */
{
  transition terminate/TearingDown:
  {
    source: state TearingDown
    destination: state idle
    triggered by:
    {
      event:
      {signals: {%fileClosed}
      on: {fileSystem}
      }
    }
  } /* end of transition terminate */;
  transition printTerminated/gettingChar:
  {
    source: state gettingChar
    destination: state TearingDown
    triggered by:
    {
      event:
      {signals: {%eof}
      on: {fileSystem}
      }
    }
  } /* end of transition printTerminated */;
  transition print/idle:
  {
    source: state idle
    destination: state settingUp
    triggered by:
    {
      event:
      {signals: {%print}
      on: {user}
      }
    }
  } /* end of transition print */;
  transition getNextChar/gettingChar:
  {
    source: state gettingChar

```

```

destination: state gettingChar
triggered by:
  {
    event:
      {signals: {%nextChar}
        on: {fileSystem}
      }
  }
code:
  { | |
    SEND printer
      SIGNAL %char
      DATA msg data
    ENDSEND
  }
  } /* end of transition getNextChar */;
transition readyToPrint/settingUp:
  {
    source: state settingUp
    destination: state gettingChar
  } /* end of transition readyToPrint */;
} /* end of transitions in: top */

```

The ROOM description of the transitions defined within the settingUp state follows:

```

transitions: /* of state settingUp */
  {
    transition printerReady/requestingPrinter:
      {
        source: state requestingPrinter
        destination: state openingFile
        triggered by:
          {
            event:
              {signals: {%printerReady}
                on: {printer}
              }
          }
      }
    } /* end of transition printerReady */;
    transition fileOpenend/openingFile:
      {
        source: state openingFile
        destination: state border to transition readyToPrint/settingUp
        triggered by:
          {
            event:
              {signals: {%fileOpened}

```

```

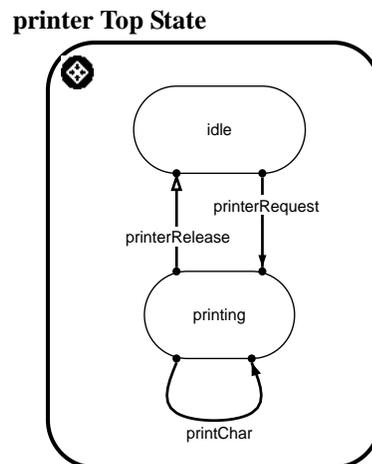
        on: {fileSystem}
        }
    }
} /* end of transition fileOpenend */;
transition print/settingUp:
{
source: state border from transition print/idle
destination: state requestingPrinter
} /* end of transition print */;
} /* end of transitions in: settingUp */

```

Printer State Machine

In Figure 162, we give the ROOMCharts model of the `printer` actor in the context of the `PrintFile` scenario. The `idle` state constitutes both the initial state (precondition) and the final state (postcondition) for the execution of the scenario.

FIGURE 162. Printer role behavior for the `PrintFile` MSC



Printer Transitions

The ROOM description of the transitions defined in the `printer` top state is given below.

```

transitions: /* of state top */
{
transition printChar/printing:

```

```

{
  source: state printing
  destination: state printing
  triggered by:
  {
    event:
    {signals: {%char}
     on: {printerInput}
    }
  }
  code:
  { | |
    SEND printerOutput
    SIGNAL %char
    DATA msg data
    ENDSEND
  }
} /* end of transition printChar */;
transition printerRelease/printing:
{
  source: state printing
  destination: state idle
  triggered by:
  {
    event:
    {signals: {%printerRelease}
     on: {printerInput}
    }
  }
} /* end of transition printerRelease */;
transition printerRequest/idle:
{
  source: state idle
  destination: state printing
  triggered by:
  {
    event:
    {signals: {%printerRequest}
     on: {printerInput}
    }
  }
  code:
  { | |
    REPLY %printerReady ENDREPLY
  }
} /* end of transition printerRequest */;
} /* end of transitions in: top */

```

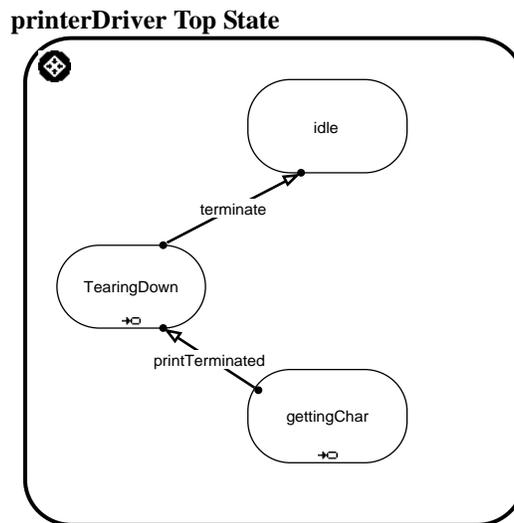
5.10.2 StopPrinting

Figure 163 and Figure 164 give the role behavior (ROOMCharts model) of printerDriver and printer in the context of the StopPrinting MSC.

printerDriver State Machine

The printerDriver ROOMCharts model (Figure 163) corresponding to the StopPrinting scenario is obtained directly from the StopPrinting MSC. The state `gettingChar` constitutes the initial state (precondition) of this scenario, and the state `idle` constitutes the final state (postcondition).

FIGURE 163. printerDriver role behavior for the StopPrinting MSC



Entry Code

Below, we give the entry-code of the `tearingDown` state.

```

state TearingDown:
{
  entry action:
  { | |
    SEND printer SIGNAL %printerRelease
    ENDSSEND.
    SEND fileSystem SIGNAL %closeFile
    ENDSSEND
  }
} /* end of state TearingDown */;

```

printerDriver Transitions

The ROOM description of the transitions defined in the printerDriver top state is given below.

```

transition printTerminated/gettingChar:
{
  source: state gettingChar
  destination: state TearingDown
  triggered by:
  {
    event:
    {signals: {%stop}
     on: {user}
    }
    or
    event:
    {signals: {%eof}
     on: {fileSystem}
    }
  }
} /* end of transition printTerminated */;
transition terminate/TearingDown:
{
  source: state TearingDown
  destination: state idle
  triggered by:
  {
    event:
    {signals: {%fileClosed}
     on: {fileSystem}
    }
  }
}

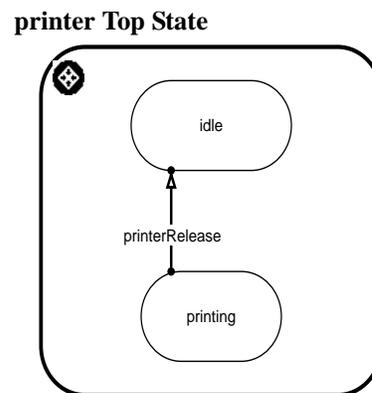
```

```
 } /* end of transition terminate */;
```

Printer State Machine

In Figure 164, we give the ROOMCharts model of the printer actor in the context of the StopPrinting scenario. The printing state is the initial state (precondition) of the scenario, and the idle state is the final state (postcondition).

FIGURE 164. Printer role behavior for the StopPrinting MSC



Printer Transitions

The ROOM description of the transition defined in the printer top state is given below.

```

transition printerRelease/printing:
{
  source: state printing
  destination: state idle
  triggered by:
  {
    event:
    {signals: {%printerRelease}
      on: {printerInput}
    }
  }
}
} /* end of transition printerRelease */;
```

5.11 ROOMChart Modeling (Iteration 1)

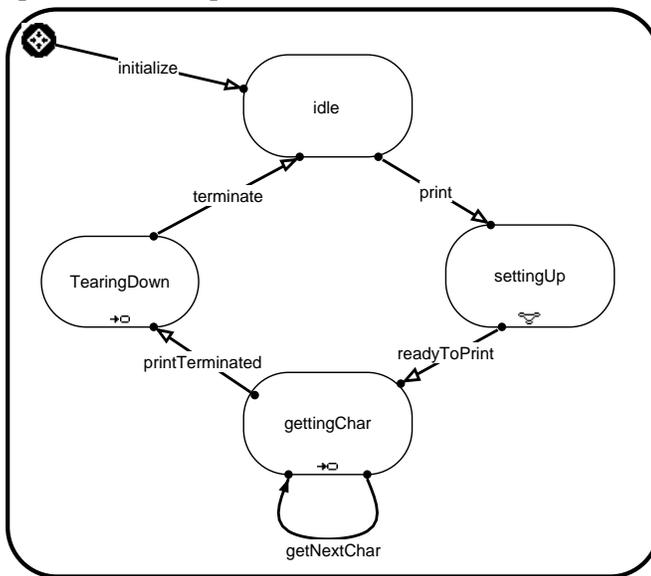
In this step, we define a ROOMCharts component behavior model for each actor contained in the global ROOM structure defined in Figure 145. The component behavior model of an actor is defined by integrating the role behaviors of all the roles played by the actor.

printerDriver

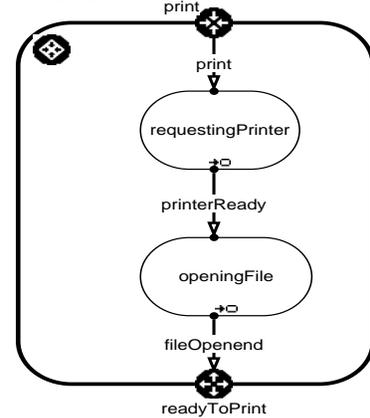
The ROOMCharts component behavior of the printerDriver actor is given in Figure 165. In this case, we use the scenario aborting pattern (section 4.7.4) to integrate the two role behavior state machines. At the schematic level, this ROOMCharts model is exactly the same as the one given in Figure 161.

FIGURE 165. printerDriver complete behavior model

printerDriver Top State



settingUp State



The entry-code and transitions are also the same as the one given for the ROOMChart role model of the printerDriver actor given in Figure 161 with the exception of the printTerminated transition. The ROOMCharts description of this transition is given below. The only difference between this transition and the printTerminated transition of the printerDriver actor given in Figure 161 is that in this case the transition can be triggered by two different events: one associated with the stop signal and one associated with the eof signal. In the printerDriver ROOMChart of Figure 161, the transition could only be triggered by the eof signal.

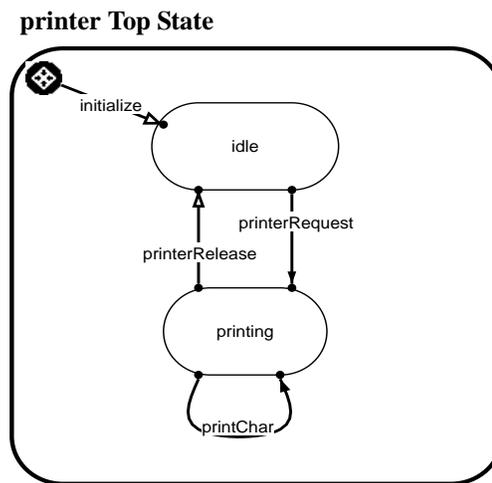
```

transition printTerminated/gettingChar:
  {
    source: state gettingChar
    destination: state TearingDown
    triggered by:
      {
        event:
          {signals: {%stop}
           on: {user}
          }
        or
        event:
          {signals: {%eof}
           on: {fileSystem}
          }
      }
  }
  } /* end of transition printTerminated */;

```

Printer

The ROOMCharts component behavior of the printer actor is given in Figure 166. This ROOMCharts model is exactly the same as the one given in Figure 162. Because the transitions are also the same, we do not rewrite them here.

FIGURE 166. Printer complete behavior model

5.12 Testing of Iteration 1

At the end of each iteration, the resulting complete ROOM model must be tested against the scenario models of the system. The objective of this step is to ensure that the resulting ROOM model can correctly execute the set of system scenarios. This includes both testing each scenario in isolation and testing interactions between scenarios.

At this point, we can test the complete ROOM model of `PrinterSystem` against the `PrintFile` and `StopPrinting` scenarios using the customized MSCs produced during the iteration. In this case study, the testing of `PrinterSystem` is conducted in two steps:

1. Generate the execution MSCs of the system using the ObjecTime Developer toolset
2. Manually compare the execution MSCs with the customized MSCs

The comparison of the execution MSCs and customized MSCs showed that the ROOM model produced in this first iteration is correct with respect to both the `PrintFile` and `StopPrinting` scenarios. It also showed that the triggering of the `StopPrinting` scenario aborts the execution of the `PrintFile` scenario.

We also tested `PrinterSystem` to ensure that at the termination of each printing request (no matter if the printing terminates normally or is interrupted), the requested file is closed and the printer is released.

In this iteration, we did not consider robustness issues at all. For example, if for some reasons the printer does not respond to the `printerRequest` message or if the file cannot be opened, the `printerDriver` would end up in a deadlock situation.

Also, as mentioned at the beginning of the iteration, alternatives to the main scenarios have not been considered. In this iteration, we assume that the requested file exists and can be printed. We also assume that the name of the file of a `StopPrinting` request corresponds to the name of the file that is being printed. The elimination of these assumptions would typically come in future iterations.

Therefore, we conclude that the current ROOM model satisfies the objective of iteration 1 and thus we can move on to iteration 2.

5.13 Iteration 2

In this section, we give the different models produced during iteration 2 and discuss the main issues encountered during this second iteration. Because the goal of this iteration is to illustrate the integration of scenarios that relate to different aspects of the system (i.e. operational and control), we concentrate more specifically on the two modeling phases in which scenario integration is done: the ROOM Structure Modeling phase and ROOM-Chart Modeling phase.

5.13.1 STDs for Iteration 2

In the second iteration of this case study, we focus on the definition and integration of the two control scenarios: the **StartUp** scenario and **Shutdown** scenario.

The **StartUp** and **Shutdown** STDs are respectively given in Figure 167 and Figure 168.

FIGURE 167. StartUp STD

STD Identifier: StartUp	
Description: Scenario describing the steps required to start up and configure PrinterSystem.	
External Actors: System Controller, File System	
Precondition: PrinterSystem is unconfigured and running.	
Triggering event: The SystemController enters the startConfig command	
<ol style="list-style-type: none"> 1. PrinterSystem receives the startConfig command from the system controller 2. Get PrinterSystem config data from the file system 3. Set internal data 4. Configure Printer 5. Notify the system controller of the successful termination of system configuration and wait for the start message 6. Receive the start command from the system controller 7. Start PrinterSystem (i.e. bring the system to its operational state) 	
Postcondition: PrinterSystem is configured and operational	
Resulting event: none	
Alternatives: - If the config data cannot be obtained from the file system, then the StartUp request is rejected. (Not included) - If the printer cannot be configured, then the StartUp request is rejected. (Not included)	
Nonfunctional requirements:	
Comments: - In the context of this case study, we only deal with the main scenario of this STD.	

FIGURE 168. Shutdown STD

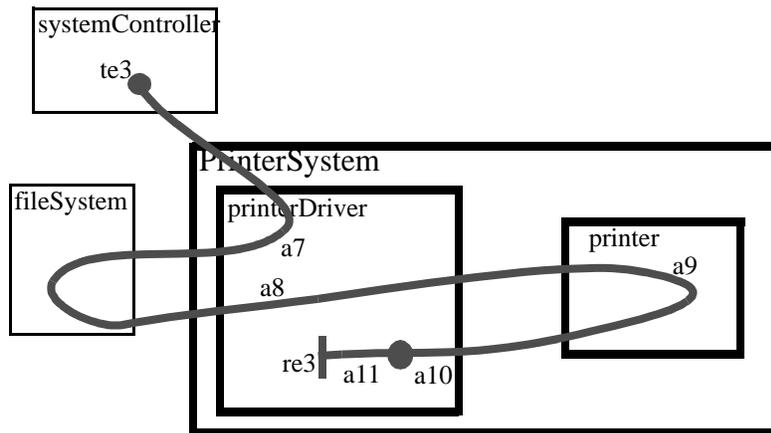
STD Identifier: Shutdown	
Description: Scenario describing the steps required to shutdown PrinterSystem.	
External Actors: System Controller	
Precondition: PrinterSystem is in the operational state	
Triggering event: The SystemController enters the shutdown command	
<ol style="list-style-type: none"> 1. Receive the shutdown command from the SystemController 2. Shutdown the printer 3. Clear current configuration 	
Postcondition: PrinterSystem returns to the unconfigured state	
Resulting event: none	
Alternatives: none	
Nonfunctional requirements: none	
Comments:	

5.13.2 UCM Model (Iteration 2)

In this second iteration, we add two UCMs to the UCM model of iteration 1: the StartUp UCM and the Shutdown UCM. These two UCMs are respectively given in Figure 169 and Figure 170. Because there are no direct interactions between these two UCM paths, there is no need to combine them in a single composite UCM. In fact, the execution of the StartUp UCM path is a precondition to the execution of the Shutdown UCM path.

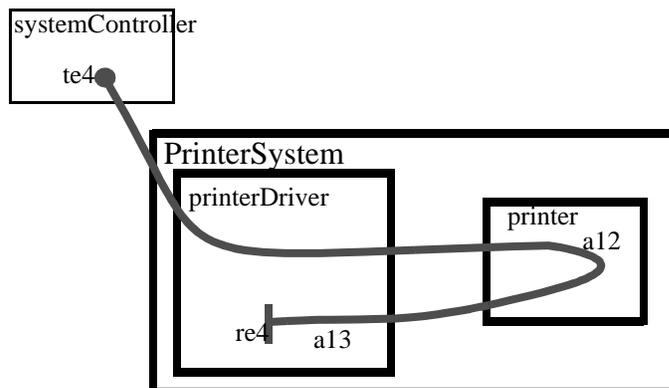
Also, because they correspond to scenarios that relate to a different system aspect than the operational scenarios of iteration 1, there is no direct interactions between these new UCM paths and the ones of iteration 1. For this reason there is no need to combine them with the paths defined in iteration 1 into a composite UCM.

FIGURE 169. StartUp UCM



te3: startUp command (pre-condition: the system is in the unconfigured state)
 re3: system start up is completed (post-condition: the system is in the operational state)
 a7: get PrinterSystem config data from the file system
 a8: set internal data
 a9: configure Printer
 a10: notify System Controller of the termination of system configuration and wait for the system controller to enter the start command
 a11: start PrinterSystem

FIGURE 170. Shutdown UCM



te4: shutdown command (pre-condition: the system is in the operational state)
 re4: the system shutdown is completed (post-condition: the system is in the unconfigured state)
 a12: shutdown printer
 a13: clear current configuration

5.13.3 Specification MSC Model (Iteration 2)

The specification MSCs corresponding to the StartUp and Shutdown UCMs are respectively given in Figure 171 and Figure 172.

In the StartUp specification MSC, we define two high-level messages (`getConfigData` and `configData`) that are exchanged between the `printerDriver` component and the `fileSystem` external component. These messages are not part of standard file system interfaces. They are used in this case study for purpose of conciseness. When implementing `PrinterSystem`, these two messages would need to be decomposed into messages that are part of the standard file system interface. Message decomposition can be done using the technique described in section 3.6.1.

Also, in this case study, we do not define the `PrinterSystConfigData` and `PrinterConfigData` data classes. The data classes would need to be defined in order to implement the system.

FIGURE 171. StartUp specification MSC

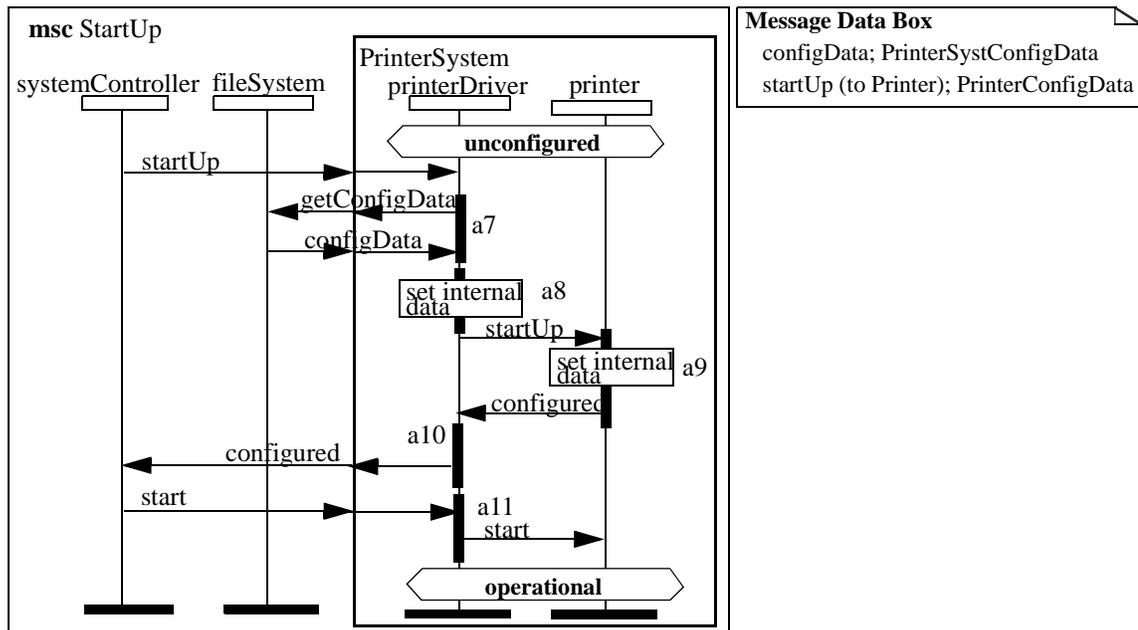
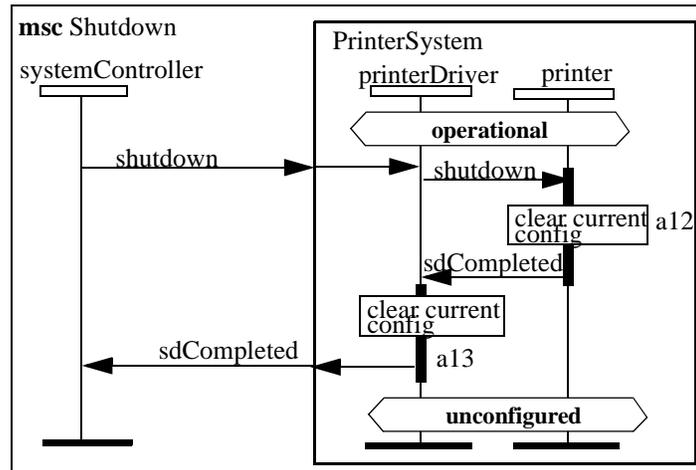


FIGURE 172. Shutdown specification MSC



5.13.4 ROOM Structure Model (Iteration 2)

In this section, we illustrate how a global ROOM structure is built from a set of role structures. We first give the role structure model that corresponds to the two control scenarios, and then, we integrate the resulting structure in the ROOM structure produced in the first iteration (Figure 145).

5.13.4.1 Definition of the Role Structure for Control Scenarios

The set of protocol classes and the ROOM structure associated with the control scenarios of PrinterSystem addressed in this iteration (i.e. the StartUp and Shutdown scenarios) are respectively given in Figure 173² and Figure 174.

2. The name of the protocol class used to communicate with the fileSystem in the context of the control scenarios is called `I2FileSystemInterface` as it refers to the protocol required to communicate with the fileSystem in the context of the scenarios of iteration 2.

FIGURE 173. Control scenarios protocol classes

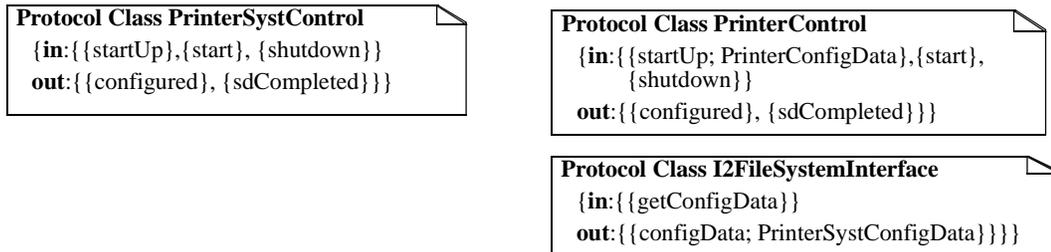
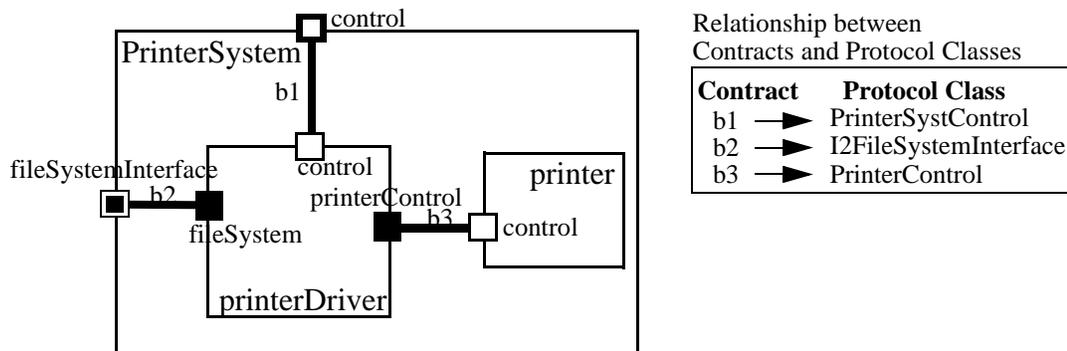


FIGURE 174. Control scenarios ROOM structure



5.13.4.2 Definition of the Global ROOM Structure

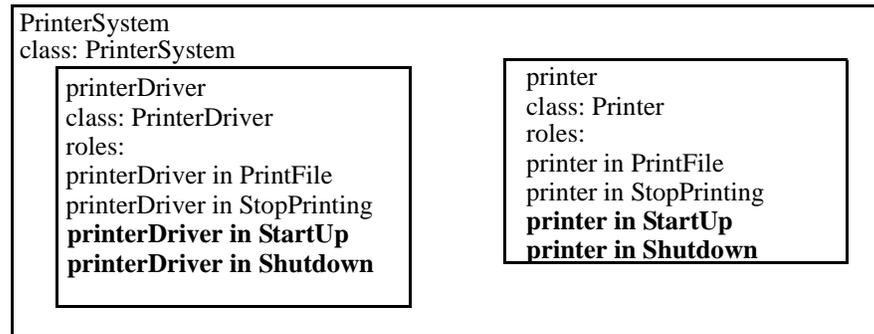
The definition of the global ROOM structure of a system is done by integrating the different role structures defined in an iteration with the ROOM structure model produced in the previous iteration. In the current iteration, we need to integrate the role structure associated with the control scenario (Figure 174) with the ROOM structure obtained in the first iteration (Figure 145). The result is given in Figure 177. This ROOM structure model constitutes the complete structure of PrinterSystem after iteration 2.

In order to obtain the new ROOM structure of PrinterSystem, we need to update the list of roles associated with the system actors (Figure 143), the set of protocol classes (Figure 144), and the ROOM structure (Figure 145). The result of those updates is respectively given in Figure 175, Figure 176, and Figure 177. The information added in this iteration is showed in bold in these figures.

Roles

In the current iteration, we add the roles associated with the **StartUp** and **Shutdown** scenarios to **PrinterSystem** components. The result is given in Figure 175.

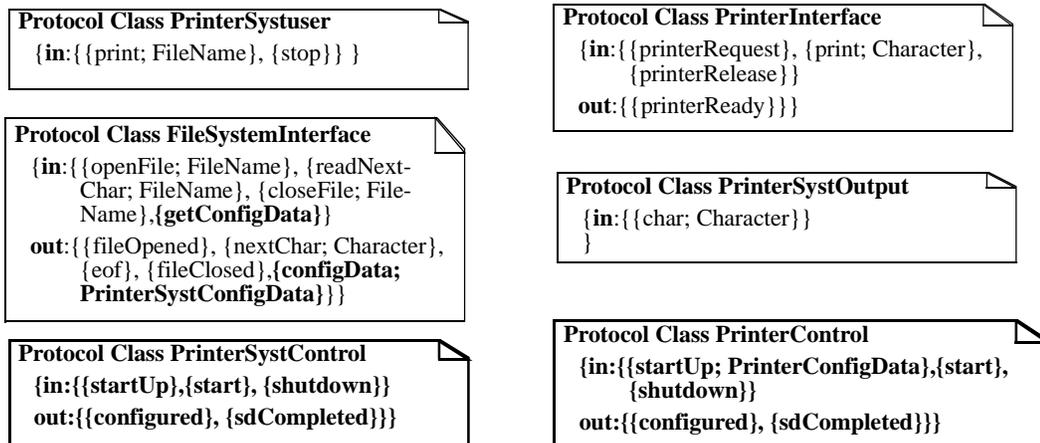
FIGURE 175. PrinterSystem actors



Protocol Classes

Also in this iteration, we defined two new protocol classes, the **PrinterSystControl** and **PrinterControl** protocol classes, and added two new messages (**getConfigData** and **configData**) to the existing **FileSystemInterface** protocol class. The resulting set of protocol classes is given in Figure 176.

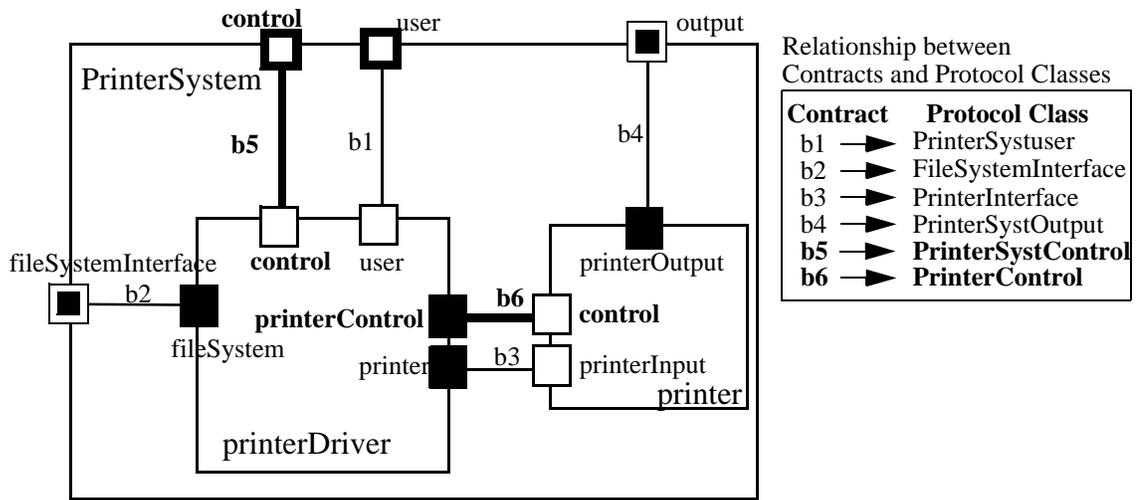
FIGURE 176. PrinterSystem protocol classes



System Structure

Finally, we add the two new contracts, labelled b5 and b6 to the ROOM structure produced in the previous iteration. The new ROOM structure is given in Figure 177. The contracts defined in this ROOM structure are based on the set of protocol classes defined in the Figure 176.

FIGURE 177. PrinterSystem ROOM structure



5.13.5 Customized MSC Model (Iteration 2)

In Figure 178 and Figure 179, the customized MSCs corresponding to the StartUp and Shutdown specification MSCs (described in Figure 171 and Figure 172) are given. We recall that customized MSCs are obtained from specification MSCs by adding ROOM structure information and component behavior information (see section 5.8 and section 5.9 for more details).

FIGURE 178. StartUp customized MSC

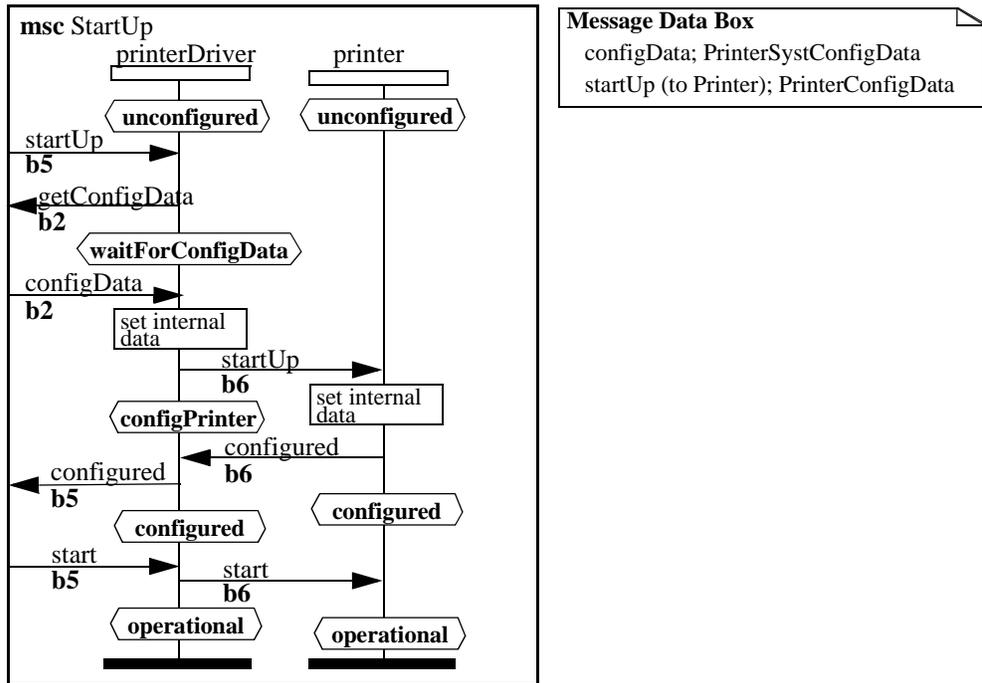
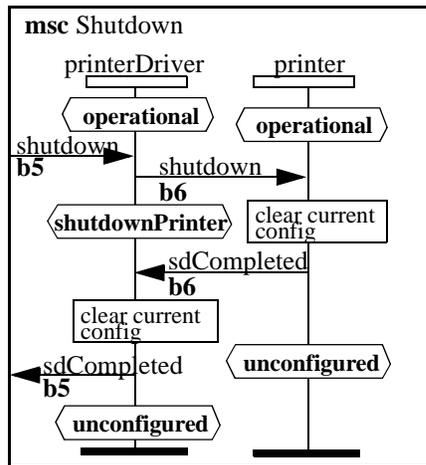


FIGURE 179. Shutdown customized MSC



5.13.6 ROOMChart Model (Iteration 2)

In this section, we illustrate how complete component behavior ROOMChart models are built from a set of role behavior ROOMChart models. We first give the role behavior ROOMChart models associated with the two control scenarios, and then, we integrate the resulting ROOMChart models with the ROOMChart models produced in the first iteration (Figure 165 and Figure 166).

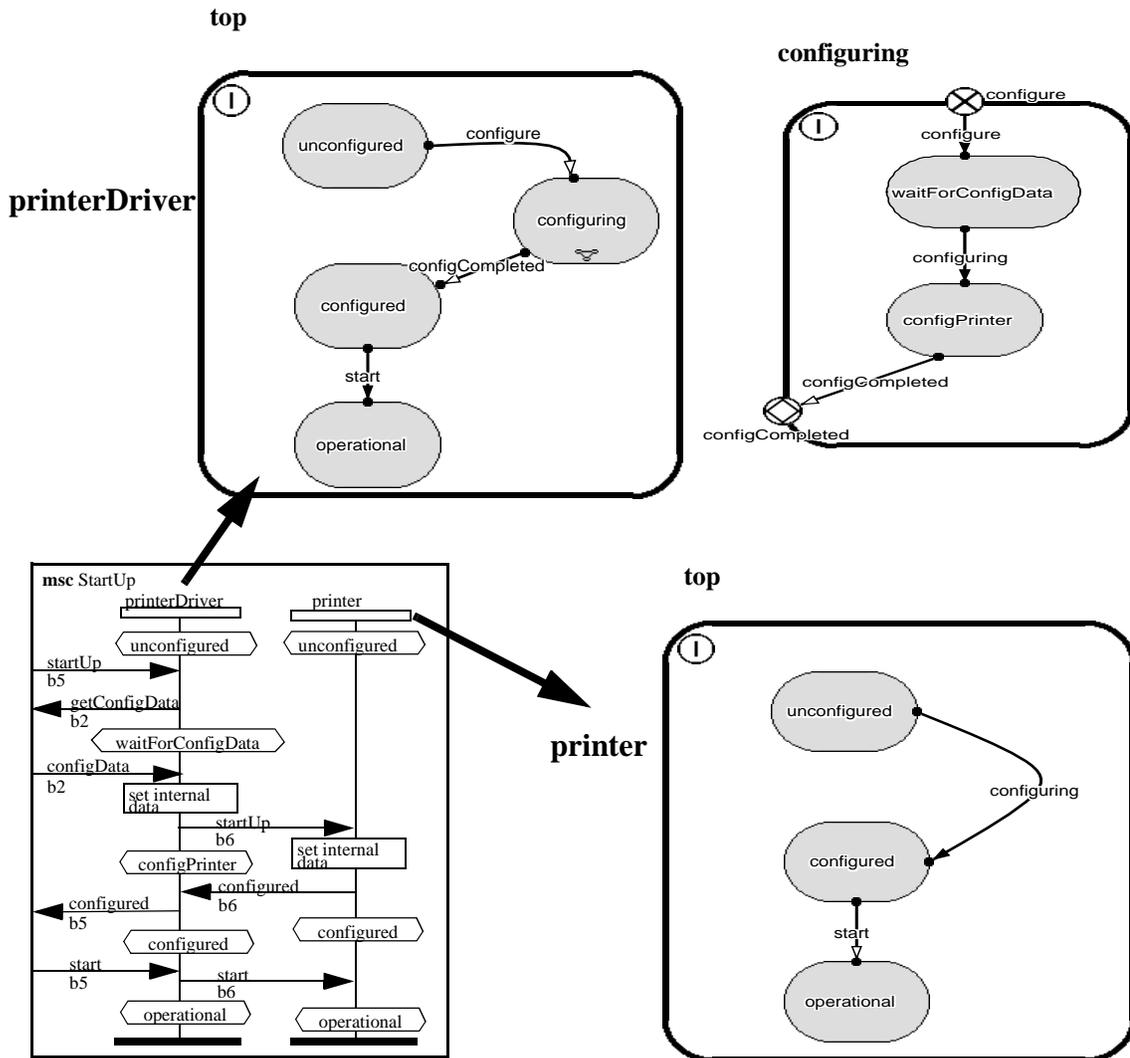
5.13.6.1 Definition of the Role Behavior State Machines for the Control Scenarios³

In Figure 180 and Figure 181, the role behavior state machines associated with the StartUp and Shutdown scenario are given. The primitive states defined in the different state machines correspond to the states (conditions) defined in the customized MSCs. In PrinterDriver, some states have been grouped into composite states to match the general control-level state machine structure described in Figure 90 of the State Machine Integration Pattern section (section 4.4). The role behavior state machine associated with Printer uses a variation of the control-level state machine structure in which the configuration and shutdown phases are described by a single transition instead of a composite state.

The content of the different ROOMChart transitions is given below the figures. Readers can easily verify that the triggering events and code defined on the different transitions correspond to the messages and actions defined in the customized MSCs.

3. The ROOMCharts (state machines) diagrams given in this section look slightly different than the one used in the previous sections of this chapter (i.e. the sections corresponding to the first iteration). For example, the rounded rectangle representing states are colored in grey, instead of being white. This is due to the fact that we used two different versions of ObjecTime Developer to produce the documentation; the documentation of the first iteration was produced using ObjecTime Developer 5.1, while the documentation of the second iterations was produced using ObjecTime Developer 5.2. There is no semantic difference between the notation of the two versions. So, regardless of whether the states are white or grey, they have the same semantics. Also, the description format of the transitions is slightly different, but contains the same basic information.

FIGURE 180. Role behavior state machines for the StartUp scenario



Transitions of the PrinterDriver State Machine in the StartUp Scenario

```

transitions /* of state top */
{
  define configure
    from state unconfigured
    to state configuring transition configure
    triggers
    {
      define signals {startUp} on {control};
    };
  define configCompleted
    from state configuring transition configCompleted state configPrinter

```

```

to state configured;
action
    { | |
      SEND control SIGNAL %configured ENDSEND
    };
define start
from state configured
to state operational
triggers
    {
      define signals {start} on {control};
    }
action
    { | |
      SEND printerControl SIGNAL %start ENDSEND
    };
} /* end of transitions in top */

```

The internal transitions of the configuring composite state are defined as follows.

```

transitions /* of state startUp */
{
define configure
from border transition configure state unconfigured
to state waitForConfigData
action
    { | |
      SEND fileSystem SIGNAL %getConfigData ENDSEND
    };
define configuring
from state waitForConfigData
to state configPrinter
triggers
    {
      define signals {configData} on {fileSystem};
    }
action
    { | aPrinterConfigData |
      SELF setInternalData: msg data.
      SEND printerControl SIGNAL %startUp
        DATA aPrinterConfigData ENDSEND
    };
define configCompleted
from state configPrinter
to border transition configCompleted
triggers

```

```

    {
        define signals {configured} on {printerControl};
    };
} /* end of transitions in configuring */

```

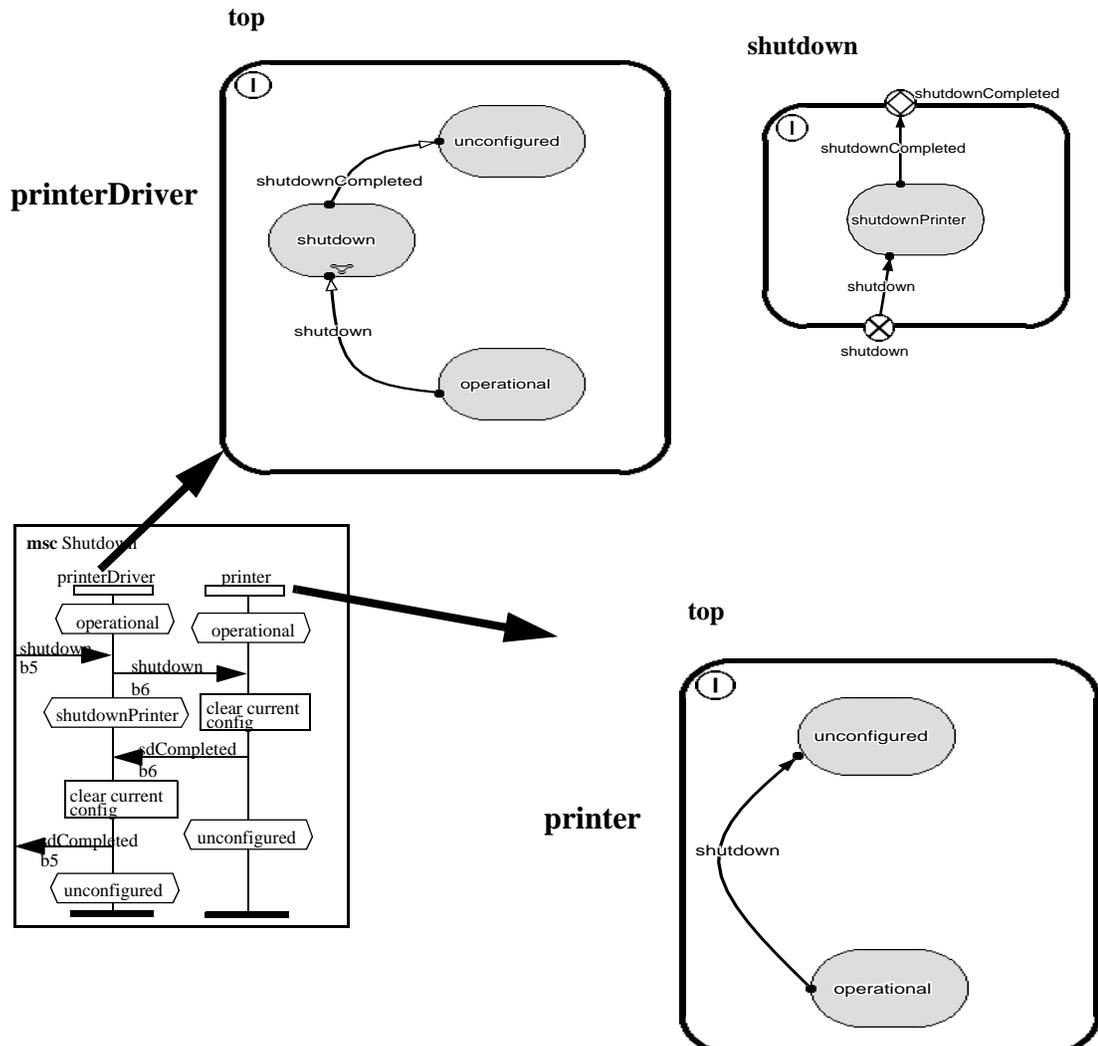
Transitions of the Printer State Machine in the StartUp Scenario

```

transitions /* of state top */
{
define configuring
    from state unconfigured
    to state configured
    triggers
    {
        define signals {startUp} on {printerControl};
    }
    action
    { | |
        SELF setInternalData: msg data.
        SEND printerControl SIGNAL %configured ENDSEND
    };
define start
    from state configured
    to state operational
    triggers
    {
        define signals {start} on {printerControl};
    };
} /* end of transitions in top */

```

FIGURE 181. Role behavior state machines for the Shutdown scenario



Transitions of the PrinterDriver State Machine in the Shutdown Scenario

```

transitions /* of state top */
{
  define shutdown
    from state operational
    to state shutdown transition shutdown
    triggers
    {
      define signals {shutdown} on {control};
    };
  define shutdownCompleted
    from state shutdown transition shutdownCompleted state shutdownPrinter
    to state unconfigured;
}
    
```

```
 } /* end of transitions in top */
```

The internal transitions of the **shutdown** composite state are defined as follows.

```
transitions /* of state shutdown */
{
define shutdown
  from border transition shutdown state operational
  to state shutdownPrinter
  action
    { | |
      SEND printerControl SIGNAL %shutdown ENDSSEND
    };
define shutdownCompleted
  from state shutdownPrinter
  to border transition shutdownCompleted
  triggers
    {
      define signals {sdCompleted} on {printerControl};
    }
  action
    { | |
      SELF clearCurrentConfig.
      SEND control SIGNAL %sdCompleted ENDSSEND
    };
} /* end of transitions in shutdown */
```

Transitions of the Printer State Machine in the Shutdown Scenario

```
transitions /* of state top */
{
define shutdown
  from state operational
  to state unconfigured
  triggers
    {
      define signals {shutdown} on {printerControl};
    }
  action
    { | |
      SEND printerControl SIGNAL %sdCompleted ENDSSEND
    };
} /* end of transitions in top */
```

Definition of Control-Level State Machines

We can now integrate the role behavior state machines of Figure 165 and Figure 166 into control-level state machines that include both the **StartUp** and **Shutdown** scenarios. We produce a state machine for each component (**printerDriver** and **printer**). The integration is conducted following the approach proposed by the State Machine Integration process pattern defined in section 4.4. This pattern suggests structuring the hierarchical state machine so that it reflects the relationships that exist between the scenarios. In the current case, the two scenarios to be integrated are part of the same scenario subset, namely the control scenario subset.

Moreover, the relationship between them is such that the execution of the **StartUp** scenario is a precondition for the execution of the **Shutdown** scenario, and the execution of the **Shutdown** scenario brings **PrinterSystem** back in the state that constitutes the precondition of the **StartUp** scenario. However, there is no direct interaction between the two scenarios. The resulting event of a scenario does not trigger the execution of the other one. The two scenarios are executed sequentially, but their triggering requires the reception of specific external messages. We integrate the **StartUp** scenario and **Shutdown** scenario using the process illustrated in Figure 90 of section 4.4.

The two resulting state machines are given in Figure 182 and Figure 183. The transitions of these hierarchical state machines are the same as the ones defined above. The state machines contained in the **startUp** and **shutdown** composite state are also the same as the ones given in Figure 165 and Figure 166.

Readers should note that even if the two resulting control-level state machines look slightly different, they use the same structuring approach. The difference between the two state machines lies in the content of the configuring and shutdown phases. In the case of the **printerDriver**, the description of both phases requires the use of composite states. In the case of the **printer**, the content of both phases is simple enough so that they can be

described by means of a single transition. The two resulting state machines can be viewed as two different variations of the same state machine structuring pattern.

FIGURE 182. Control-level state machine for the PrinterDriver actor

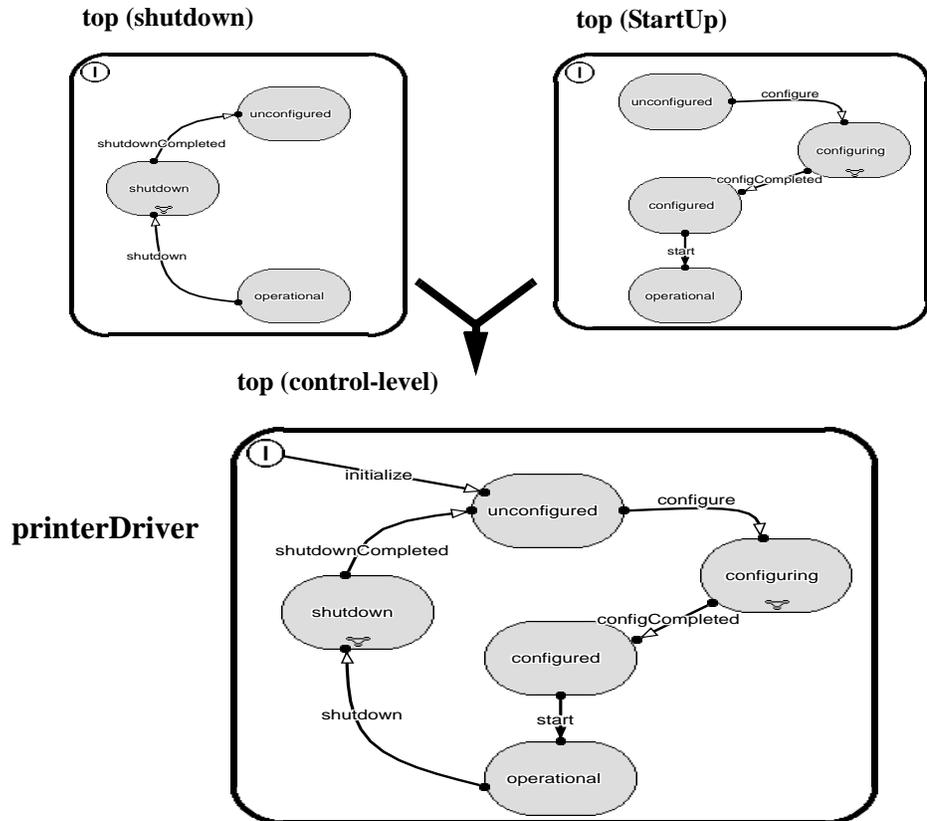
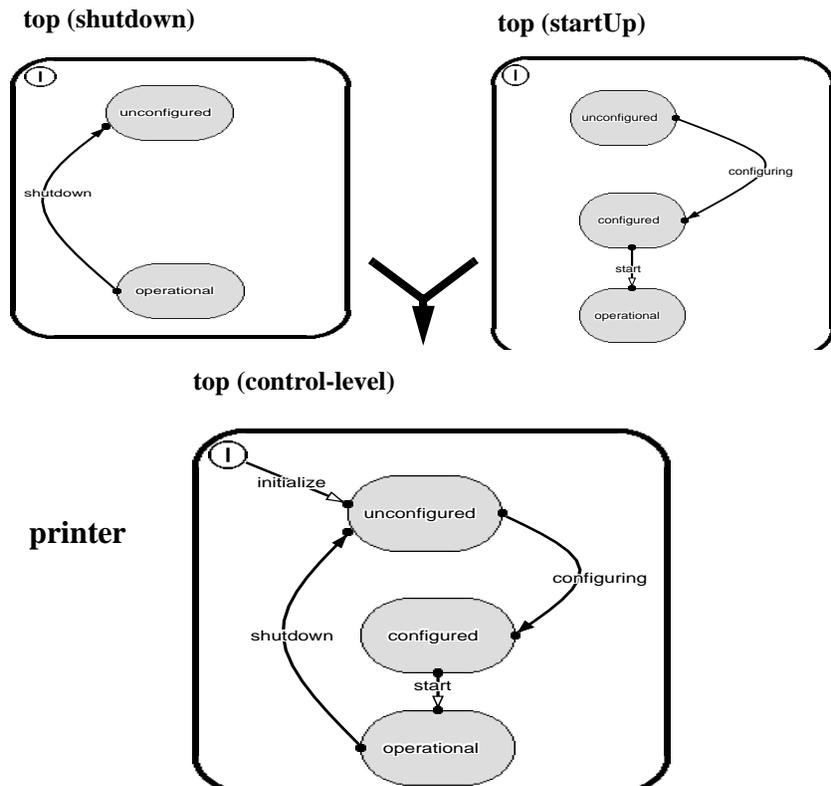


FIGURE 183. Control-level state machine for the Printer actor



5.13.6.2 Integration of the Role Behavior State Machines into Component Behavior State Machines

We can now integrate the control-level state machines produced in the current iteration to the ROOM model produced in the previous iteration. Once again, we conduct the integration following the approach proposed by the State Machine Integration process pattern (section 4.4). This pattern suggests structuring the hierarchical state machine so that it reflects the relationships that exist between the scenarios.

In the current case, we must integrate a control-level state machine and an operational-level state machine. For this purpose, we use the integration process described in the example section of section 4.4 (page 199 to page 204). As a result, the control-level state machine is placed at the top level of the component behavior, and the operational-level state machine is placed inside the operational state of the control-level state machine.

Because it now contains a state machine, the operational state of the control-level state machine becomes a composite state. All transitions remain as previously defined.

The same integration process is used for both the printerDriver component and the printer component. The result of the integration is given in Figure 184 and Figure 185.

FIGURE 184. Resulting PrinterDriver ROOMChart for iteration 2

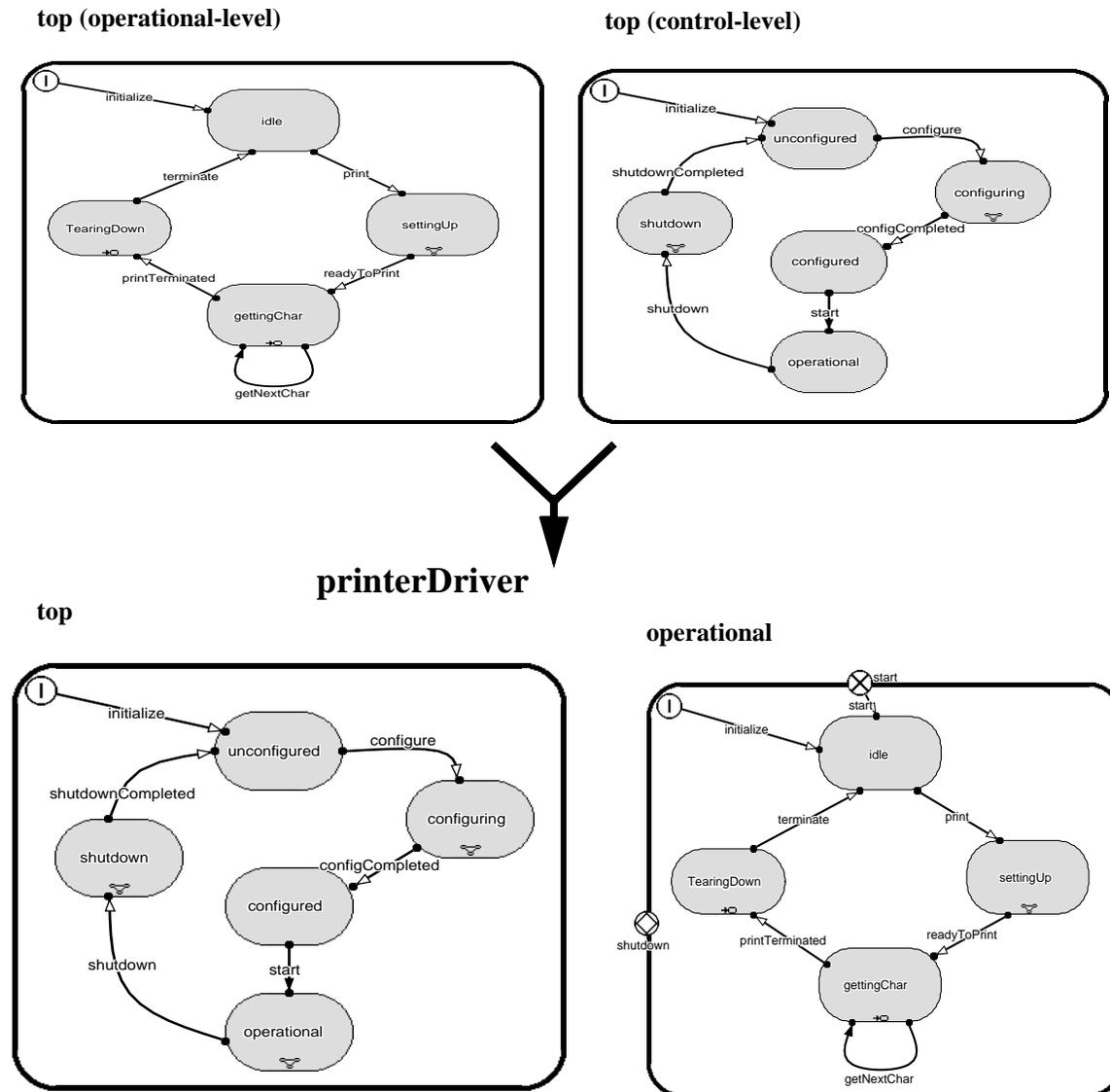
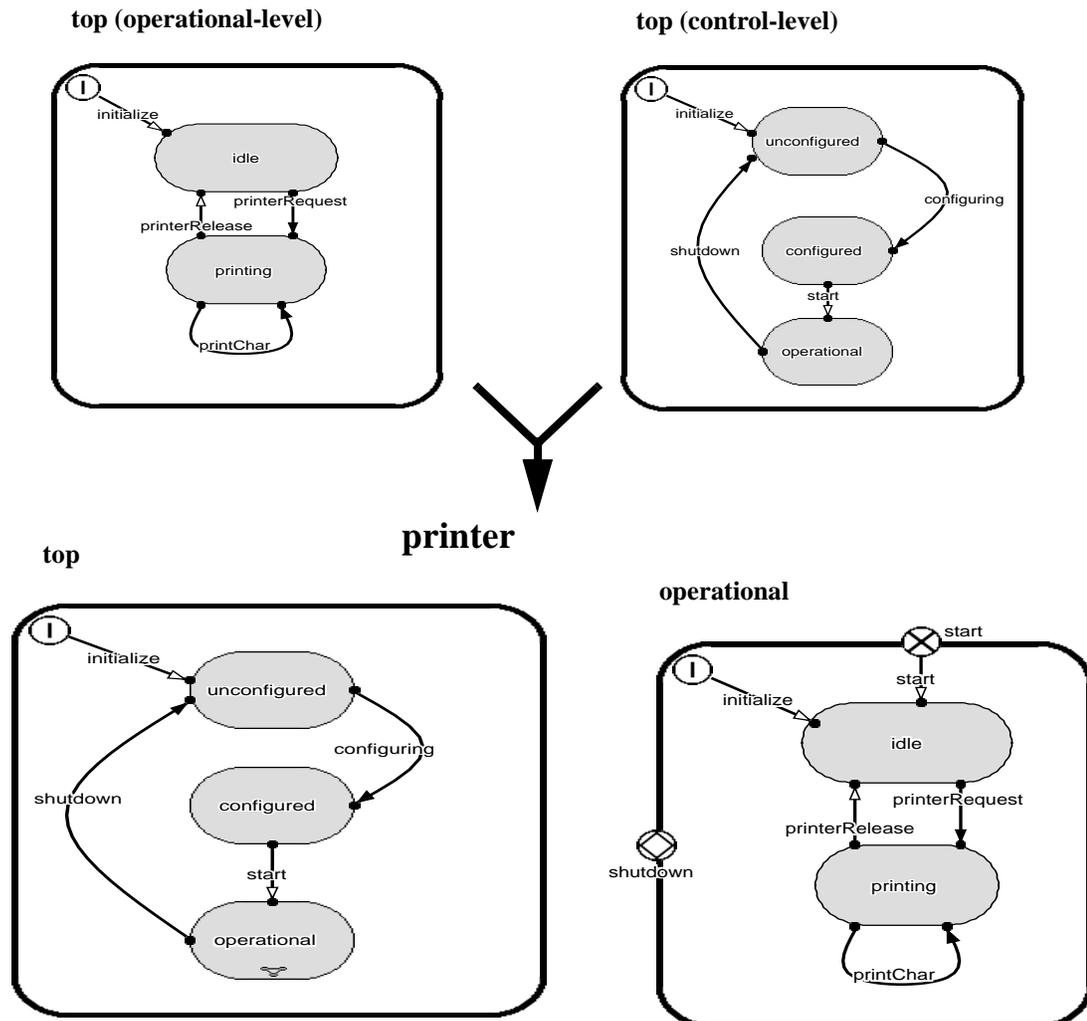


FIGURE 185. Resulting Printer ROOMChart for iteration 2



5.13.7 Testing of Iteration 2

When testing the ROOM model resulting from Iteration 2, we can show that the resulting model can correctly execute each of the four implemented scenarios (PrintFile, Stop-Printing, StartUp, and Shutdown) individually. The testing results are obtained by normally comparing the execution MSCs produced by the ObjecTime Developer toolset (as a result of model execution) with the customized MSCs produced during the iteration.

If we now test the resulting ROOM model more thoroughly, and more specifically if we test `PrinterSystem` for possible undesired scenario interactions, we discover that the resulting ROOM model does not satisfy all requirements. In particular, if a `shutdown` message is received by `PrinterSystem` while it is printing a file, the `Shutdown` scenario will be executed without closing the file and releasing the printer. This violates one of the important requirements listed in `PrinterSystem` requirements (section 5.1). This problem emerges as a result of scenario integration, namely the integration of the `PrintFile` scenario and the `Shutdown` scenario. In the next section, we discuss how the problem can be fixed.

5.13.8 Error Fixing (Iteration 2)

The issue now consists in fixing the problem without affecting the overall consistency of the different models. The strong traceable nature of the RT-TROOP modeling process is used in this phase to modify the models in a consistent manner. In this section, we discuss and illustrate how modifications must be made to the different models in order to solve the problem.

While in the previous phases of this case study we used the RT-TROOP process in a forward manner, i.e. going from requirements to a ROOM model, in this phase we go backward. We start by fixing the problem at the ROOMChart level and then use the traceability information maintained by the RT-TROOP process to propagate the modifications backward to the different models. This way, overall consistency can be maintained between the models.

In order to fix the problem, we first need to determine the cause of the problem. In the current case, we inferred that the problem occurred when the `Shutdown` scenario is triggered while `PrinterSystem` is executing the `PrintFile` scenario. The second step of our error fixing consists in determining where the problem must be fixed. By analyzing the set of

requirements and the set of responsibilities that have been allocated to the different components (in the UCM modeling phase of the two iterations), we observe that the `printerDriver` is the component that is responsible for ensuring that after each printing job the file is closed and the printer is released. Therefore, modification must be done in the `printerDriver`.

The only `printerDriver` operational state that can guarantee that there is no opened file and no requested printer (that has not been released) is the `idle` state. Therefore, one immediate solution to the problem consists in updating the precondition of the `Shutdown` scenario to become `operational.idle` (i.e. the `idle` state located in the `operational` composite state). This modification of the precondition ensures that the execution of the `Shutdown` scenario would satisfy the requirement.

Following the backward ordering of the RT-TROOP process, the correction of the problem will successively involve the evaluation of modifications in:

- The `ROOMChart` model
- The customized MSC model
- The `ROOM` structure model
- The specification MSC model
- The UCM model
- The `STD` model

It is important to note that the ordering given here is not a strict one. For example, the modification of the specification MSC model could be done before the modification of the `ROOM` structure model.

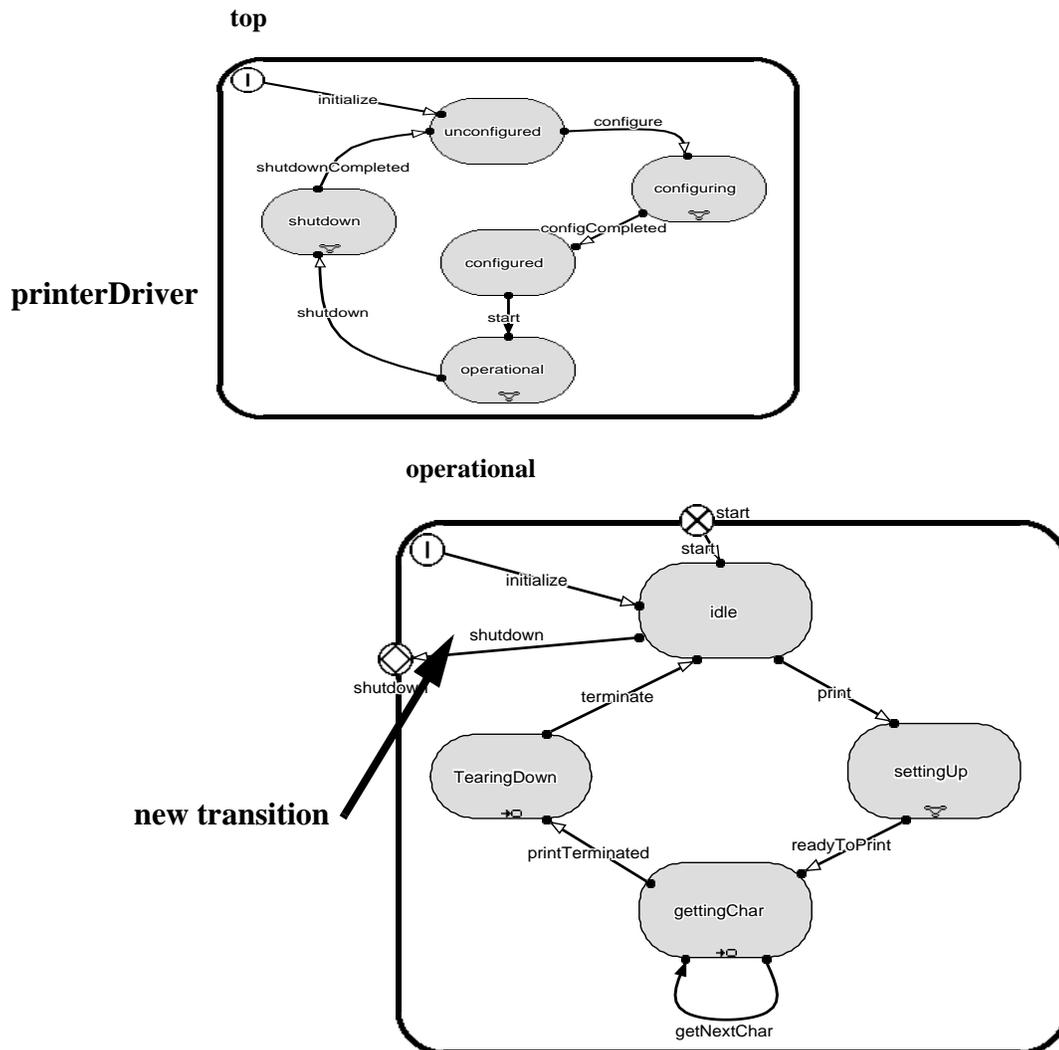
In the remainder of this section, we successively analyze the impact of the modifications on the different RT-TROOP models.

5.13.8.1 ROOMChart Model Modification

As previously mentioned, we first need to modify the ROOMChart model of the `printerDriver`. Because the problem is related to scenario integration and not to individual scenarios, we do not have to modify the role behavior models associated with the different scenarios. Only the component behavior model is modified.

One way to ensure that the `Shutdown` scenario will only be executed if the `printerDriver` is in the `idle` state consists in replacing the `shutdown` group transition that is located at the border of the operational state (and thus is taken upon arrival of the `shutdown` message no matter which operational state the system is in) by a `shutdown` transition linked specifically to the `idle` state. This way, the `Shutdown` scenario can only be triggered if the `printerDriver` is in the `idle` state (or more specifically in the `operational.idle` state). The modified version of the `printerDriver` hierarchical state machine is given in Figure 186 (this figure is the modified version of Figure 184).

FIGURE 186. Modified version of the PrinterDriver ROOMChart for iteration 2



The description of the new **shutdown** transition (the one located inside the **operational** state between the **idle** state and the **operational** state border) and the modified description of the **shutdown** transition (the one located in the **top** state between the **operational** state and the **shutdown** state) are given below. In this case, because the **shutdown** transition located in the **top** state is a continuation of the **shutdown** transition located in the **operational** state, the **shutdown** transition of the **top** state has no triggering event.

```

define shutdown /* of state operational */
from state operational
to state shutdown transition shutdown
  
```

```
triggers
{
  define signals {shutdown} on {control};
};

define shutdown /* of state top */
  from state operational transition shutdown state idle
  to state shutdown;
```

At the design documentation level, it is important to note that the new transition is different in nature than the other transitions already contained in the ROOMCharts models of PrinterSystem. While all other transitions are related to some specific scenario responsibilities, this new transition is introduced as a means of solving a scenario integration problem.

Problem with the Current Solution

One problem introduced by the current solution is that in cases where the shutdown message is received by the printerDriver while it is not in the idle state, the message will be discarded and the system controller will not be explicitly informed that the Shutdown scenario has not be executed. The system controller will only know that the Shutdown scenario has not been executed because s/he will not receive a shutdownCompleted message.

Alternative

A better solution would require an explicit handling of the shutdown message in all cases. In cases where the system is not idle, a shutdownRejected message could be send back to the system controller, so that s/he knows that the Shutdown scenario cannot be currently executed. We consider this as being an alternative to the Shutdown scenario. The implementation of this alternative is not included in this case study. It would need to be implemented in a later iteration.

Emergency Shutdown

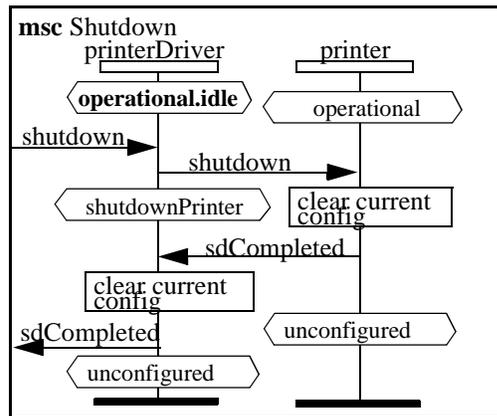
Also, one thing that the scenarios previously described do not allow is an emergency shutdown. We consider an emergency shutdown as being a separate scenario that would require the definition of a new triggering event (e.g. an `emergencyShutdown` message). This case is not covered in this case study. With the current set of scenarios, if the system controller wants to shutdown the system immediately, s/he could stop the current printing job by sending a `stop` message first, which would bring the system back to the idle state, and then send the shutdown message.

5.13.8.2 Customized MSC Model Modification

We now need to modify the customized MSC model to make it consistent with the ROOMChart model. From the modifications we made to the ROOMChart model, we have that the `Shutdown` scenario can now only be triggered when the `printerDriver` is in the `operational.idle` state. Therefore, `operational.idle` constitutes the new precondition of the `Shutdown` scenario. The other scenarios are not directly affected by the modification. This means that the parts of the models that are related to the other scenarios are not affected by the modification. For this reason, we, from this point on, focus on the parts of the models that are related to the `Shutdown` scenario. Moreover, we know that only model elements related to the precondition of the `Shutdown` scenario must be modified.

In the customized MSC model, modifications are restricted to the `Shutdown` customized MSC. The only required modification consists in changing the initial state of the `printerDriver` to `operational.idle`. Figure 187 gives the modified version of the `Shutdown` customized MSC given in Figure 179.

FIGURE 187. Modified shutdown customized MSC



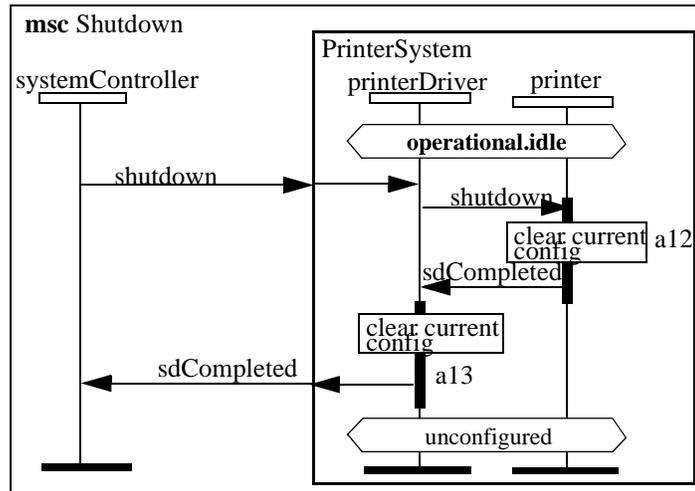
5.13.8.3 ROOM Structure Model Modification

In this case, because the structure of the system is not affected by the modification, the ROOM structure remains unchanged. If fixing the problem required adding (modifying, or deleting) new components, new ports, or new messages to existing protocol classes, then the ROOM structure would need to be modified accordingly.

5.13.8.4 Specification MSC Model Modification

In the specification MSC model, we simply need to modify the initial condition of the Shutdown MSC to reflect the fact that the Shutdown scenario can now only be executed if PrinterSystem is both operational and idle. We call the new system state *operational.idle* (even if only the *printerDriver* explicitly needs to be in the *operational.idle*). Figure 188 gives the modified version of the specification MSC given in Figure 172.

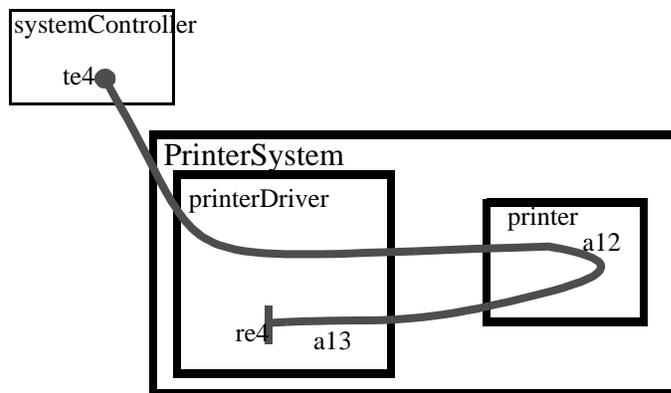
FIGURE 188. Modified shutdown specification MSC



5.13.8.5 UCM Model Modification

In the UCM model, the only required modification consists in changing the precondition of the Shutdown UCM to operational.idle. The modified Shutdown UCM is given in Figure 189. This new UCM corresponds to the modified version of the Shutdown UCM given in Figure 170.

FIGURE 189. Modified Shutdown UCM



- te4: shutdown command (pre-condition: the system is in the operational.idle state)**
- re4: the system shutdown is completed (post-condition: the system is in the unconfigured state)**
- a12: shutdown Printer**
- a13: clear current configuration**

5.13.8.6 STD Model Modification

Finally, we make two modifications to the Shutdown STD. First, the precondition of the Shutdown STD is changed to `operational.idle`. This makes the STD consistent with the UCM description of the Shutdown scenario. Second, we add the alternative discussed in section 5.13.8.1 to the alternative section of the STD. This alternative is identified as “not included” as it is not implemented in the current version of `PrinterSystem`. Figure 190 gives the modified version of Shutdown STD described in Figure 170.

FIGURE 190. Modified Shutdown STD

STD Identifier: Shutdown	
Description: Scenario describing the steps required to shutdown <code>PrinterSystem</code> .	
External Actors: System Controller	
Precondition: <code>PrinterSystem</code> is in the operational.idle state (i.e. the system is operational and idle)	
Triggering event: The system controller enters the shutdown command	
<ol style="list-style-type: none"> 1. Receive the shutdown command from the system controller 2. Shutdown the printer 3. Clear current configuration 	
Postcondition: <code>PrinterSystem</code> returns to the unconfigured state	
Resulting event: none	
Alternatives: - If the system is in the operational state but not in the idle state the shutdown request is rejected. (Not included)	
Nonfunctional requirements: none	
Comments:	

5.14 Chapter Summary

In this chapter, we illustrated the application of the RT-TROOP modeling phases using the development of a simple `PrinterSystem`. This example illustrates the different models and modeling phases that are used in RT-TROOP modeling to go from requirements to a complete detail-level ROOM model. It illustrates how RT-TROOP modeling moves in a systematic manner through abstraction levels, and how it maintains traceability between elements of the different models.

Moreover, we demonstrated how the RT-TROOP modeling process can be used in the context of iterative development. We also showed how the strong traceability maintained by the RT-TROOP process can be used to modify the different system models in a consistent manner.

CHAPTER 6 Definition of Traceability Relations Between Requirements, Scenario Models, and Communicating Hierarchical State Machines

The overall objective of RT-TROOP modeling process is to define a systematic and traceable progression from requirements, and more particularly scenario textual descriptions, to communicating hierarchical state machines. In Chapter 3, we defined the RT-TROOP modeling process in terms of a set of models and modeling phases. This modeling process allows for a systematic progression from scenario models to communicating hierarchical state machines. The systematic aspect of the process and the notation we used in the different model transition phases pave the way for the definition of traceability relations. In this chapter, we define a set of traceability relations that links elements contained in the different models of the RT-TROOP modeling process.

Our goal here is not to define a formal static semantics and syntax for the different models, but rather to establish traceability relation between specific elements of different models, or different versions of a model. For example, we are not interested in formally defining the composition of a UCM model, but we are interested in defining traceability relations between elements of a UCM model and elements of a MSC model, or between elements of one version of a UCM model and elements of another version. Formal definition of MSC and ROOM models are respectively described in [44], and [93]. There is no formal definition of UCM yet.

In this chapter, we:

- Define three different types of traceability relations.
- Define a set of concrete traceability relations between requirements, STD, UCM, MSC, and ROOM.

The set of traceability relations defined in this chapter allows capturing the relationships established, in Chapter 3, between model elements. This set of traceability relations does not pretend to be complete. Our objective is not to define a complete traceability between the different models, but rather to define a set of traceability relations that allows supporting the modeling phases of the RT-TROOP modeling process defined in Chapter 3. The set of traceability relations is intended to be open so that new relations can be added as required.

6.1 Traceability Relations

One of the main objectives of this thesis is to define a set of concrete fine-grained traceability relationships between the models that are used to go from requirements to implementation.

We define three different types of traceability relations: *inter-model traceability*, which establishes traceability links between elements of different models, *inter-version traceability*, which establishes traceability links between elements of different versions of a single model, and *design decision traceability*, which establishes traceability links between the rationale of a design decision and specific requirements that led to (or that justify) the decision.

In this section, we first describe the characteristics of the traceability relations (section 6.1.1) and the notation we use to describe them (section 6.1.2), and then define the three types of traceability relations (section 6.1.3, section 6.1.4, and section 6.1.5).

6.1.1 Characteristics

Backward Traceability

The traceability relations we define are *backward traceability relations*. This means that when making the transition from one model to another (inter-model traceability), or one version of a model and newer version of the same model (inter-version traceability), traceability links are directionally defined from the target model to the source model. When needed, *forward traceability* can be automatically calculated from the backward traceability relations.

Element Level versus Model Level Traceability

We distinguish between traceability relationships at the element level and at the model level. At the element level, each element of a model can be associated with zero or more elements in another model, or in another version of a model.

At the model level, we may also have many-to-one and many-to-many relationships. This reflects the fact that more than one elements of a model can be associated with a set of elements in another model, or in another version of a model.

In this thesis, because the model level traceability relationships can be automatically obtained from the element level ones, we only explicitly define traceability relations at the element level.

Transitivity

The transitive nature of the traceability relations we define allows obtaining a complete backward traceability to requirements. Thus, any model element, in any model (including implementation), that relates to some requirements can be transitively trace back to those requirements.

6.1.2 Notation

The traceability relations that we define establish cardinality relationships between model elements. To express these relations, we use a simple and compact notation that emphasizes the cardinality aspect of the relations. Two examples of traceability relations expressed using our notation are given below.

$$E1 \text{ tr-}>_{1..*} E2$$
$$E1 \text{ tr-}>_{0,1} E2$$

In the first case, the relation expresses the fact that an element of type E1 can be associated with one or more elements of type E2. The second one expresses the fact that an element of type E1 can be associated with zero or one element of type E2. This basic notation can be easily translated into other formalisms, like UML associations or BNF.

In our notation, we express alternatives using a vertical bar (|) between elements. An example is given below. In this example, an element of type E1 can be associated with either an element of type E2 or an element of type E3.

$$E1 \text{ tr-}> (E2 | E3)$$

6.1.3 Inter-Model Traceability

Inter-model traceability relations define traceability links between elements of different models. They allow keeping track of the transformations that take place in the transitions between different models. Such traceability provide the semantic glue that allows combining different models (or modeling techniques) in a single development process. The definition of such traceability relations requires a profound understanding of both the semantics of the notation used in the models and the relationship that exists between the models in the context of the development process.

As previously discussed, a concrete development (or modeling) process defines a partial ordering among the set of models used in the process. In this context, we use the term *source model* to refer to the input model to a model transition, and the term *target model* to refer to the model that is produced as result of the model transition. The backward inter-model traceability relations directionally defined from elements of the target model to elements of the source model.

The transition between two models may give rise to different types of traceability relationships between elements of the two models:

- One-to-one relationship - Two different cases give rise to one-to-one relationships. The first one occurs when a model element in the source model is mapped to a semantically equivalent element in the target model. A typical example of this occurs when a system component is represented in two different models. In this case, we define a traceability relation between the model elements representing the component in the two models.

The second case occurs when a model element in the source model is mapped into a set of elements in the target model. For example, in the transition between UCM and MSC, a UCM responsibility r_1 may be expressed as a set of messages (m_1, m_2, \dots, m_n)

exchanged between components (instances) in a MSC model. In this case, we define a one-to-one traceability relation between each of the messages m_i defined in the MSC model and the UCM responsibility r_1 .

- One-to-many relationship - One-to-many relationships occur when a set of model elements in the source model are mapped into a single element in the target model. For example, a transition t_1 in a ROOMChart model can group together a set of message sending (m_1, m_2, \dots, m_n) defined in a MSC model. In this case, we define a traceability relation between the ROOMChart transition t_1 and each of the messages m_i in the MSC model.
- One-to-zero (or no relationship) - One-to-zero relationships occur when some elements in the target model do not relate to any element in the source model. This happens whenever a new component is first introduced in the system in a specific model. For example, because system components are first introduced in the UCM model, UCM components, except for the ones that correspond to external actors, cannot be traced back to any STD elements.

In this chapter, we define inter-model traceability relations between the following models:

- STD and requirements
- UCM and STD
- MSC and UCM
- ROOM and MSC

We do not formally define inter-model traceability relations between ROOM and UCM. However, in Chapter 4, we defined a set of design patterns that allow establishing traceability between UCM models and ROOM behavior (ROOMCharts) models, or more precisely between scenario relationships and the structure of hierarchical state machines.

6.1.4 Inter-Version Traceability

In order to maintain complete traceability between requirements and implementation, it is not sufficient to maintain traceability between elements of the different models (i.e. models associated with different modeling techniques), but it is also necessary to maintain traceability between elements of different versions of a model. *Inter-version traceability* relations define such traceability links. They allow keeping track of the model transformations that take place in the different versions of a model.

The backward traceability relation between two successive versions of a model M , say M_n and M_{n-1} , are directionally defined from elements of M_n , which result from the application of transformations, to the elements of M_{n-1} , to which the transformations have been applied.

The modifications that are made to a model may be the result of different types of transformations. Examples of transformations include:

- Introduction of a new element - a new element is introduced in the current version of a model. In such case, because the new element does not result from the application of a model transformation to elements of a previous version of the model, we cannot establish traceability links (relationship) between the new element and other model elements. The new element is the result of a design decision, and therefore should have design decision traceability information (see next section), but does not have neither inter-model nor inter-version traceability information.
- Decomposition of an existing element - An element in one version of a model is decomposed into a set of refined elements in a later version of the same model. For example, an element $E1$ of M_{n-1} is decomposed into three elements $E11$, $E12$, and $E13$ in M_n . In such case, we define a one-to-one inter-version traceability relation between each of the elements $E11$, $E12$, and $E13$ on one side and $E1$ on the other side.

- Composition of existing elements into a new element - A set of elements in one version of a model are merged into a single element in a later version of the same model. For example, a set of elements $E1$, $E2$, and $E3$ of M_{n-1} are merged into a single element E in M_n . In such case, we define a one-to-many inter-version traceability relation between E on one side and $E1$, $E2$, and $E3$ on the other side.
- Replacement of an existing element - An element in one version of a model is replaced by a another elements in a later version of the same model. For example, an element $E1$ of M_{n-1} is replaced by an element $E2$ in M_n . In such case, we define a one-to-one inter-version traceability relation between $E1$ on one side and $E2$ on the other side.
- Also, an element E defined in M_{n-1} may be unaffected by model transformations executed in M_n . In this case, we define a one-t-one inter-version traceability relation between the element E of M_n and element E of M_{n-1} .

It is important to note that the purpose of defining inter-version traceability relations is not to define a semantics for model transformation, or a set of transformation rules that specify why and how transformation should be applied. Our goal here is rather to identify the information that need to be recorded in order to maintain traceability between requirements and implementation.

6.1.5 Design Decision Traceability

Design decisions lead to the application of a model transformations which may transform existing sets of model elements into new sets of model elements, or introduce new elements in a model. In a multi-model development process, design decisions are taken both in the models (in-model modeling phases) and in the transition between models (model transition phases). In the previous two sections, we defined inter-model and inter-version traceability relations which allows maintaining traceability relationships between model

elements resulting from design decisions. However, we have not discussed yet how the relationship between design decisions and requirements can be maintained.

In practice, the justification of design decisions often comes from requirements. For example, requirements may explicitly require (impose) the use of specific components, services, or communication protocols. Requirements may also impose a certain level of robustness which lead to the definition of specific scenarios or to the choice of a particular system components.

Maintaining proper documentation of design decisions is very important for several reasons.

- The modification of requirements may affect design decisions taken during the development process, which in turn affect model elements resulting from the design decisions. If traceability relationships between design decisions and requirements are not explicitly maintained, it becomes impossible to evaluate the impact of requirement modification at the design decision level.
- Stakeholders are often interested in knowing how their requirements have been addressed in the system. They are usually not interested so much in the detailed elements that are related to their requirements, but rather in the major design decisions that resulted from their requirements. Therefore, we must be able to know how each requirements is addressed in the system in terms of design decisions.
- At different point in time during their lifecycle, systems need to be restructured. The motivations for conducting system restructuring may be varied. They may include: increase of maintainability and extensibility, use of a new technology, increase of system performance, adapt to a new standard protocol, and so on. When conducting system restructuring, system designers (or architects) must ensure that all existing requirements remain satisfied. Therefore all model elements and design decisions that are related to existing requirements should be carefully analyzed before being modified.

In order to avoid incorrect modifications of design decisions, design decisions must be properly documented. Without proper documentation, it is in general very difficult to remember why certain design decisions have been taken.

Design decision traceability becomes even more important in the context of large development projects that involve groups of designers. In such projects, tracking design decisions is a difficult task. Also, some designers often leave before the end of the project, and the rationale behind their design decisions is lost. In an ideal process each design decision would be justified by a reference to requirements (if such a link exists) and/or a brief textual description that explain the rationale of the decision. This way, if requirements change, the impact of the changes on the models can also be evaluated at the design decision level.

To maintain information concerning the justification of design decisions, we define *design decision traceability* relations. Design decision traceability relations establish traceability links between design decisions and requirements whenever appropriate (i.e. when a design decision is justified by some specific requirements). This allows documenting the rationale of design decisions. Design decision traceability constitutes an extra level of traceability on top of inter-model and inter-version traceability. While inter-model and inter-version traceability establish linkages between model elements, design decision traceability establishes linkage between the design decisions that give birth to these elements and requirements.

The objectives of establishing design decision traceability are threefold:

- It allows tracing back design decision to specific requirements.
- It allows linking model elements to design decisions.
- It allows distinguishing design decisions that are justified by requirements from others.

In order to maintain proper traceability information, design decision traceability information should be added at two different levels in the models:

- At the model level, where the set of design decisions taken in a model are summarized. At this level, each design decision is linked to the set of requirements that justifies it.
- At the model element level, where the set of elements that resulted from a design decision are linked to the design decision at the model level. Thus, model elements that have been defined because of some requirements are transitively linked back to those requirements (via the design decision documentation at the model level).

6.2 Traceability Relation Between STD and System Requirements

As discussed in section 2.1.2, the goal of STD is to organize system requirements on a per scenario basis. Thus, elements of STD can be linked to individual requirements in the requirements list. This allows maintaining a strong traceability between scenarios and general system requirements.

Formally, the traceability relations between STD elements and the list of system requirements are defined as follows. First, since each scenario is defined to satisfy specific functional requirements, we define a traceability link between each STD and the set of functional requirements that justifies its existence. We record this information by linking the STD identifier with the set of related requirements.

(TR1) STD_identifier $tr \rightarrow 1..*$ requirement

Also, each STD element is linked to the set of requirements that applies to it.

(TR2) STD_description $tr \rightarrow 1..*$ requirement

(TR3) STD_externalActorst $tr \rightarrow 1..*$ requirement

(TR4) STD_triggerEvent $tr \rightarrow 1..*$ requirement

(TR5) STD_precondition $tr \rightarrow 1..*$ requirement

(TR6) STD_responsibility $tr \rightarrow 1..*$ requirement

(TR7) STD_resultingEvent $tr \rightarrow 1..*$ requirement

(TR8) STD_postcondition $tr \rightarrow 1..*$ requirement

(TR9) STD_alternatives $tr \rightarrow 1..*$ requirement

(TR10) STD_nonFunctionalReq $tr \rightarrow 1..*$ requirement

The traceability relation between STD elements and requirements is many-to-many, i.e. each scenario element may be linked to a set of requirements, and a single requirement may be linked to more than one scenario elements.

6.3 Traceability Relations in UCM Models

In this section, we define a set of traceability relations for UCM model elements.

6.3.1 Inter-Model Traceability between UCM Models and STDs

UCM models are composed of two parts, path and structure. In this section, we separately analyze the traceability relationships that exist between these two parts of UCM models and STDs.

Traceability between UCM Paths and STDs

STD and UCM both describe scenario paths at the same level of abstraction. In fact, UCM maps are usually associated with a textual description that briefly describes its overall objective and its different elements: responsibilities, pre and post-conditions, and triggering and resulting events, alternatives, and non-functional requirements that apply to the scenario.

Also, STD and UCM both allow for the grouping of related scenarios into scenario clusters; STD groups together main scenarios with alternatives, while UCM uses the concept of related path set to express alternatives to main scenarios.

In the context of RT-TROOP, we establish a direct relationship between a STD and a UCM related path set; we associate an STD with each UCM related path set. This ensures a strong traceability between UCM models and requirements. Formally, the traceability relation between UCM path and STD is defined as follows.

(TR11) UCM_relatedPathSet $tr \rightarrow$ STD_identifier

Then each path within a UCM related path set is associated with a specific scenario in a STD. In the following relation, the term STD_scenario can refer to either the main scenario described in an STD, or one of its alternatives.

(TR12) UCM_path $tr \rightarrow$ STD_scenario

At a more detailed level, we define a one-to-one relationship between elements of a UCM path and elements of a STD as follows:

(TR13) UCM_triggerEvent $tr \rightarrow$ STD_triggeringEvent

(TR14) UCM_precondition $tr \rightarrow$ STD_precondition

(TR15) UCM_responsibility $tr \rightarrow$ STD_responsibility

(TR16) UCM_resultingEvent $tr \rightarrow$ STD_resultingEvent

(TR17) UCM_postcondition $tr \rightarrow$ STD_postcondition

Traceability between UCM Structure and STDs

In terms of structure, STDs contain very little information. In fact, STDs are only concerned with the set of external actors that participate in the execution of the scenario. Such external actors may either represent users (humans) of the system or other systems with which the system communicate during the execution of the scenario.

In relation with the UCM model, each of the STD external actors is mapped into a UCM component. However, system components described in a UCM model can not be linked back to any STD element. Therefore, the traceability relation between UCM components and STD external actors can be expressed as follows.

(TR18) UCM_component $tr \rightarrow_{0,1}$ STD_externalActor

6.3.2 Inter-Version Traceability in UCM Models

Model Transformations in UCM models

UCM is more than just a graphical representation for scenario textual descriptions. It is a high level modeling technique that allows performing design activities such as:

- Definition of a new map in a UCM model.
- Path composition (in composite maps), which allows composing a set of paths in a single map. This allows describing relationships, like interaction and concurrency, between a set of paths.
- Responsibility decomposition, which allows refining existing responsibilities into sub-responsibilities or stubs.
- Definition of new paths in a related path set.
- Definition of new responsibilities on a path.
- Definition of new components in a map.
- Component (or structure) decomposition, which allows decomposing a system components into a set of sub-components.
- Responsibility allocation (or re-allocation), which allows allocating sets of responsibilities to specific components.
- Path restructuring, which may include path factoring, that allows cutting a path into a set of sub-paths, and path merging (the opposite process), that allows merging a set of individual paths into a more complex path or into a related path set.

To keep track of the modifications that are made to a UCM model in the different iterations, we define a set of inter-version traceability relations between elements of a UCM model. This allows linking elements of different versions of a UCM model. As previously

discussed, these relations are backward ones, i.e. they are directionally defined from elements of UCM_model_n to elements of UCM_model_{n-1} .

Inter-Version Traceability in Relations

UCM Models and Maps

First, because UCM models are composed of a set of UCM maps, and because maps are incrementally added to UCM models through iterations, we define an inter-version traceability relation between UCM maps. A map UCM_i contained in UCM_model_n may either:

- come from a previous version of the model (UCM_model_{n-1}), in which case we need to define a traceability link between UCM_i of UCM_model_n and UCM_i of UCM_model_{n-1} , or
- be defined in the current version of the UCM model (UCM_model_n), in which case there is no traceability link to elements of previous versions of the UCM model.

This relation is captured as follows.

(TR19) $UCM_map \xrightarrow{tr}{}_{0,1} UCM_map$

Paths

We establish a $0, \dots, *$ inter-version traceability relation between UCM paths of two different versions of a UCM model. This reflects the fact that:

- a new UCM path p may be introduced in UCM_model_n . In this case, p is not linked to any path in UCM_model_{n-1} .

- a UCM path p in UCM_model_n may come directly from UCM_model_{n-1} (i.e. p existed in UCM_model_{n-1} and is not modified in UCM_model_n), or p may be a sub-path of a path in UCM_model_{n-1} (i.e. p is the result of a path factoring). In this case, p is linked to exactly one path in UCM_model_{n-1} .
- a UCM path p in UCM_n may be the result of merging a set of paths contained in UCM_{n-1} . In this case, p is linked to each of the path of UCM_{n-1} involved in the merging.

This relation is expressed as follows.

(TR20) $UCM_path \text{ tr} \rightarrow_{0,..,*} UCM_path$

Responsibilities, and Components

A similar reasoning applies to UCM responsibilities and UCM components as we define the two following relations.

(TR21) $UCM_responsibility \text{ tr} \rightarrow_{0,1} UCM_responsibility$

(TR22) $UCM_component \text{ tr} \rightarrow_{0,1} UCM_component$

Stubs

A UCM responsibility may also be replaced by a stub. For this purpose, we define a one-to-one traceability relation between UCM stub and UCM responsibility.

(TR23) $UCM_stub \text{ tr} \rightarrow UCM_responsibility$

In a UCM model, each stub must be associated with one or more maps that might be plugged-in the stub at run-time. To capture this relationship, we define a $1,..,*$ traceability relation between between UCM stub and UCM map.

(TR24) $UCM_stub \text{ tr} \rightarrow_{1,..,*} UCM_map$

It should be noted that a UCM map can be associated with more than one stub.

Path Interactions

In the UCM modeling phase, sets of scenarios may be composed in composite maps. This allows specifying important system interactions between scenarios. Therefore, new scenario interactions may be defined in UCM models using the scenario interaction notation. For this purpose, we define the following traceability relation between scenario interaction symbols.

(TR25) UCM_scenarioInteraction $tr \rightarrow_{0,1}$ UCM_scenarioInteraction

Responsibility Allocation

To capture the relationship that is established between responsibilities and component during responsibility allocation, we define a one-to-many (0,...,*) traceability relation between UCM components and UCM responsibilities. This reflects the fact that each component in a UCM model may be responsible for implementing zero or more responsibilities.

(TR26) UCM_component $tr \rightarrow_{0,...,*}$ UCM_responsibility

6.3.3 Design Decision Traceability in UCM Models

To keep track of the relationships that may exist between the design decisions discussed in section 6.3.2 and requirements, we define traceability relations between design decisions and requirements. The cardinality is zero when the design decision is not linked to any requirements, i.e. it results of a pure design decision, and it is one or more when the design decision is justified by some requirements.

(TR27) UCM_designDecision $tr \rightarrow_{0,...,*}$ requirement

Also, in order to maintain traceability between elements resulting from UCM model transformations and design decisions, we define a traceability relation between UCM elements and the design decisions that resulted in their creation. The 0,1 cardinality reflects the fact that a UCM element may be the result of a design decision (cardinality 1), or it may come directly from another model (cardinality 0).

(TR28) UCM_element $tr \rightarrow_{0,1}$ UCM_designDecision

6.4 Traceability Relations in MSC Models

In this section, we define a set of traceability relations for UCM model elements.

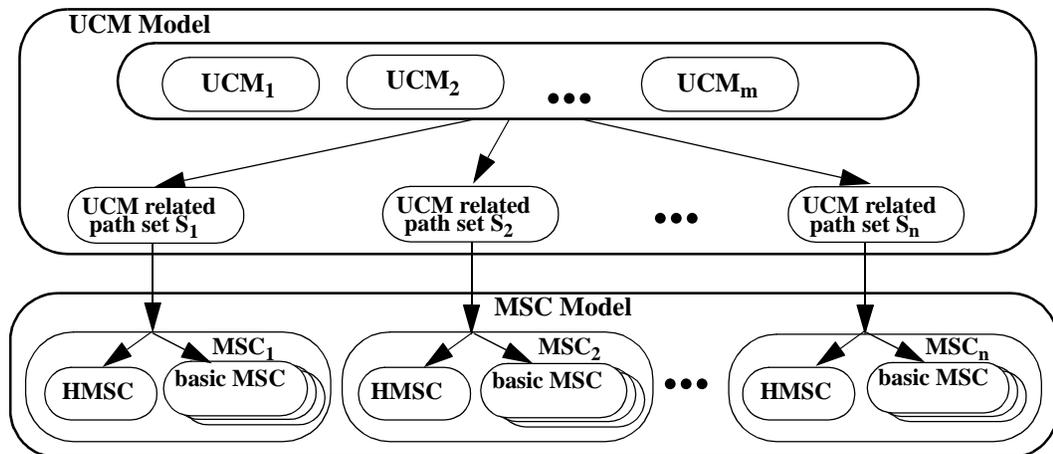
6.4.1 Inter-Model Traceability Between MSC Models and UCM Models

High-Level Relationship between UCM and MSC Models

Prior to defining a set of detailed-level traceability relations between elements of UCM and MSC, we briefly review the high-level relationship that exists between UCM models and MSC models. A schematic view of this relationship is illustrated in Figure 191. A UCM model is composed of a set of UCM maps, labelled $UCM_1, UCM_2, \dots, UCM_m$ in Figure 191, and a set of UCM related path sets, labelled S_1, S_2, \dots, S_n (see section 2.2.8 for more details). Similarly, a MSC model is composed of a set of individual MSCs, labelled $MSC_1, MSC_2, \dots, MSC_n$ in Figure 191 (see section 2.3.4 for more details).

We establish a one-to-one traceability relationship between UCM related path sets and MSCs. Moreover, if a related path set is composed of more than one path segment, then the generated MSC is composed of a HMSC and a set of basic MSC, where for each path segment in the related path set we define an MSC reference in the HMSC and a corresponding basic MSC.

FIGURE 191. Relationship between UCM and MSC



At a detail level, we establish traceability relationships between the following elements of the two models:

- UCM components and MSC instances
- UCM triggering and resulting events and MSC messages
- UCM responsibilities and MSC sequences of messages and actions
- UCM path segments and MSC references in HMSC
- UCM path segments and basic MSCs
- UCM path segment connectors and MSC reference connectors in HMSCs
- UCM stubs and HMSCs or basic basic MSCs

Traceability Relations between HMSCs and UCM Models

At the HMSC level, we define the following inter-model traceability relations between MSC and UCM. These relations come directly from the previous discussion on the high-level relationship between UCM and MSC models.

(TR29) $MSC_hmsc \text{ } tr \rightarrow UCM_relatedPathSet$

(TR30) $MSC_reference \text{ } tr \rightarrow UCM_pathSegment$

This traceability relation (TR30) is generalized to include UCM stubs in TR40.

(TR31) $MSC_referenceConnector \text{ } tr \rightarrow UCM_segmentConnector$

HMSCs also contains MSC conditions. At the HMSC level, conditions correspond to system states, i.e. they apply to the overall set of instances (components) contained in the MSC. When making the transition between UCM and MSC, UCM preconditions and postconditions are mapped into MSC conditions. However, not all MSC conditions are associated with UCM preconditions and postconditions. Some of them are introduced as result of design decisions. Thus, the traceability relation between MSC conditions and UCM preconditions and postconditions is optional (0,1).

(TR32) $MSC_condition \text{ } tr \rightarrow_{0,1} (UCM_precondition \mid UCM_postcondition)$

This traceability (TR32) relation is generalized to include UCM waiting places in TR38.

Traceability Relations between Basic MSCs and UCM Models

At the basic MSC level, we define the following inter-model traceability relations between MSC and UCM.

First, we associate a basic MSC with each UCM path segment.

(TR33) $MSC_basicMsc \text{ } tr \rightarrow UCM_pathSegment$

Then, we express the high-level scenario description given by the UCM path segments in terms of basic MSCs. This results in the following traceability relations.

Structure Level

At the structure level, each UCM component is associated with a MSC instance. However, the inverse is false. Some MSC instance may have no equivalence in the UCM model. Therefore, the relation between MSC instance and MSC component is optional (0,1).

(TR34) $MSC_instance \text{ } tr \rightarrow_{0,1} UCM_component$

Also, we introduce a MSC timer for each UCM timer. The relation is optional (0,1) because a MSC timer can be introduced in a MSC model as a result of a design decision with no equivalence in a UCM model.

(TR35) $MSC_timer \text{ } tr \rightarrow_{0,1} UCM_timer$

Path Level

At the path level, UCM start point (defined in terms of a precondition and a set of possible triggering events), end bar (defined in terms of a postcondition and a set of possible resulting events), responsibilities, waiting places, stubs must be expressed in terms of basic MSC elements.

When moving from UCM to MSC, all UCM triggering and resulting events are mapped into MSC messages, and every UCM responsibility is expressed as a sequence of MSC messages and actions. Therefore, a MSC message may be linked to either a UCM responsibility, a UCM triggering event, or a UCM resulting event.

(TR36) $MSC_message \text{ } tr \rightarrow (UCM_responsibility \mid UCM_triggeringEvent \mid UCM_resultingEvent)$

A MSC action on the other hand can only be linked to a UCM responsibility.

(TR37) $MSC_action \ tr \rightarrow UCM_responsibility$

Semantically, a UCM waiting place is a system state in which a scenario is blocked waiting for an unblocking event. Thus, when moving from UCM to MSC, we associate a MSC condition with each UCM waiting place. This leads to the generalization of TR32 that can be rewritten as follows.

(TR38) $MSC_condition \ tr \rightarrow_{0,1} (UCM_precondition \ | \ UCM_postcondition \ | \ UCM_waitingPlace)$

Also, if the waiting place is a timed waiting place (timer), we define a MSC message from the component that contains the UCM timer to a MSC timer instance. This leads to the generalization of TR36 that can be rewritten as follows.

(TR39) $MSC_message \ tr \rightarrow (UCM_responsibility \ | \ UCM_triggeringEvent \ | \ UCM_resultingEvent \ | \ UCM_timer)$

Finally, we associate a MSC reference (in basic MSCs) with each UCM stub contained in a path segment. This leads to the generalization of TR30 that can be rewritten as follows.

(TR40) $MSC_reference \ tr \rightarrow (UCM_pathSegment \ | \ UCM_stub)$

6.4.2 Intra-Model Traceability in MSC Models

Model Transformations in MSC models

A MSC model is composed of a set of HMSCs and basic MSCs. It is incrementally built through iterations as new scenarios are added to the system, or existing ones are modified.

The addition of a new scenario to the system may result either:

- In the creation of a new HMSC, in the case where the new scenario is part of a new UCM related path set.
- In the addition of new MSC references (and corresponding basic MSCs) to an existing HMSC, in the case where the new scenario is added to an existing UCM related path set.

In both cases, it requires the introduction of new MSC references in HMSCs and the definition of new basic MSCs.

The modification of existing scenarios may require the creation of new elements in basic MSCs, or the modification of existing ones.

Model transformations that may be done in MSC models during an iteration include:

- Definition of new HMSCs or basic MSCs.
- Introduction of new MSC references.
- Definition of new instances (components), messages, actions, conditions, and references.
- Decomposition (or refinement) of existing instances (components), messages, and conditions.

These model transformations may be carried out as a result of MSC restructuring, introduction of new scenarios, or refinement of existing scenarios.

To keep track of the modifications that are made to a MSC model in the different iterations, we define a set of inter-version traceability relations between elements of MSC models. This allows linking elements of different versions of a MSC model. As previously discussed, these relations are backward ones, i.e. they are directionally defined from elements of MSC_model_n to elements of MSC_model_{n-1} .

Intra-Model Traceability Relations

HMSC

A HMSCs contained in a MSC_model_n may either:

- Come from a previous version of the model (MSC_model_{n-1}), in which case we need to establish a one-to-one traceability relation between the HMSC of the two versions of the MSC model.
- Be defined in the current version of the MSC model (MSC_model_n), in which case there is no traceability link to elements of previous version of the MSC model.

This is captured by the following optional (0,1) traceability relation.

(TR41) $MSC_hmsc \text{ } tr \rightarrow_{0,1} MSC_hmsc$

Reference and Basic MSC

A similar reasoning also apply to MSC references and basic MSCs. This lead to the following traceability relations.

(TR42) $MSC_reference \text{ } tr \rightarrow_{0,1} MSC_reference$

(TR43) $MSC_basicMsc \text{ } tr \rightarrow_{0,1} MSC_basicMsc$

Basic MSCs are composed of instances, messages, actions, conditions, and references. The following inter-version traceability relations are defined for these elements.

Instance, Message, and Condition

During an iteration, the set of instances, messages, and conditions that exist in MSC_model_{n-1} may remain intact, or be decomposed into a set of refined ones. Also, new

instances, messages, and conditions may be introduced. For this purpose, the following inter-version traceability relations.

(TR44) $MSC_instance \text{ } tr \rightarrow_{0,1} MSC_instance$

(TR45) $MSC_message \text{ } tr \rightarrow_{0,1} MSC_message$

(TR46) $MSC_condition \text{ } tr \rightarrow_{0,1} MSC_condition$

Action

New MSC actions may also be introduced in a basic MSC as a result of scenario refinement. This lead to the following relation.

(TR47) $MSC_action \text{ } tr \rightarrow_{0,1} MSC_action$

Reference

Also, as a result of replacing a responsibility by a stub in a UCM model (TR23), the set of messages and actions that are associated with the replaced responsibility need to be replaced by a MSC reference in the MSC model¹. In this case, a tracability relation is established between the new MSC reference and the set of messages, actions, and conditions it replaces. Since each UCM responsibility is associated with at least one element in the MSC model, the resulting relation has cardinality 1,...,*.

(TR48) $MSC_reference \text{ } tr \rightarrow_{1,..,*} (MSC_message \mid MSC_action)$

1. As described by TR40, UCM stubs are mapped onto MSC references in the transition between UCM and MSC.

6.4.3 Design Decision Traceability in MSC Models

As in UCM models, we define traceability relations between design decisions and requirements. The cardinality is zero when the design decision is not linked to any requirements, i.e. it results of a pure design decision. It is one or more when the design decision is justified by some requirements.

$$\text{MSC_designDecision } tr \rightarrow_{0,\dots,*} \text{ requirement}$$

Also, in order to maintain traceability between elements resulting from UCM model transformations and design decisions, we define a traceability relation between UCM element and the design decision that resulted in its creation. The 0,1 cardinality reflects the fact that a UCM element may be the result of a design decision (cardinality 1), or it may come directly from another model (cardinality 0).

$$\text{MSC_element } tr \rightarrow \text{ MSC_designDecision}$$

6.5 Traceability Relations in ROOM Models

In this section, we define a set of traceability relations for ROOM model elements.

6.5.1 Inter-Model Traceability Between ROOM Models and MSC Models

When we say that we establish a relation between a ROOMChart element, like a transition action, and an MSC element, like a message, we actually refer to a specific message arrow in a specific basic MSC. Thus, from this point of view, every MSC message is unique,

even if two messages share the same name (identifier). The same principle applies to MSC instances. ROOM components are linked to specific instances in the MSC model, which are defined on a per MSC basis.

Traceability Between ROOM Structure Models and MSC Models

A ROOM structure model is defined in terms of a set of actors, a set of ports, and a set of bindings. In MSC models, system structure is defined in terms of a set of instances and a set of messages exchanged between the instances. The concepts of interface components and bindings (communication channel) do not explicitly exist in MSC.

In the definition of a ROOM structure model from a MSC model, we:

- Define a set of ROOM actors to play the roles defined by instances in the MSC models.
- Define a set of ROOM actor ports by which actor can exchange messages.
- Define bindings that actors together.

In this section, we establish traceability relations between structure elements of the two models.

Actor

A ROOM actor can play the role of one or more (1,..,*) MSC instances.

(TR49) ROOM_actor $tr \rightarrow_{1,..,*}$ MSC_instances

Port

A ROOM port can be associated with one or more MSC messages.

(TR50) ROOM_actorPort $tr \rightarrow_{1,..,*}$ MSC_messages

Binding

A ROOM binding can be linked to the set of MSC messages that can flow through it.

(TR51) ROOM_binding $tr \rightarrow_{1,..,*}$ MSC_messages

Inter-Model Traceability Between ROOMCharts and MSC Models

State machine

Because a state machine can encapsulate a set of scenarios (defined by means of MSCs), we establish a traceability relation between ROOMChart state machines and the set of HMSCs and basic MSCs that describe the scenarios that can be executed the state machine (the set of scenarios for which the state machine has been defined). The cardinality of the relation is zero if the state does not encapsulate any scenario. This is the case in particular for primitive states.

(TR52) ROOMChart_stateMachine $tr \rightarrow_{0,..,*}$ (MSC_basicMsc | MSC_hmssc)

State

A ROOMChart state can be associated with a MSC condition. All ROOMChart primitive states should be related to a MSC condition. A composite state on the other hand may have been created as a result of a design decision. Thus, ROOMChart states have an optional (0,1) relation with MSC conditions.

(TR53) ROOMChart_state $tr \rightarrow_{0,1}$ MSC_condition

Transition

A ROOMChart transition is composed of a sequence of transition segments. Each segment is associated with a (possibly empty²) sequence of MSC messages and actions. This is captured by the following relation.

(TR54) ROOMChart_transitionSegment $tr \rightarrow_{0,..,*} (\text{MSC_message} \mid \text{MSC_action})$

The first message defined in a transition must be an incoming message. It corresponds to the triggering event of the transition. Then, all other messages and actions take place on the transition itself. All of the messages, other than the incoming triggering event, must be outgoing messages, except the ones that correspond to the reply of synchronous communication.

A ROOMChart transition can be triggered by one or more triggering messages. Only the first segment of a transition can contain a triggering event.

(TR55) ROOMChart_transitionTriggering $tr \rightarrow_{1,..,*} \text{MSC_message}$

The transition action code executed on a transition contains zero or more transition actions. Each ROOMChart transition action can be related to a MSC message or a MSC action. However, part of the transition actions are defined for the purpose of data manipulation, i.e. computing data, assigning new values to variables, defining new data objects, packaging data in objects that are to be send with messages, and so on. The transition actions required for such data manipulation have usually no relationship with MSC elements. For this reason, the traceability between a ROOMChart transition action and MSC elements is optional (0,1).

(TR56) ROOMChart_transitionAction $tr \rightarrow_{0,1} (\text{MSC_message} \mid \text{MSC_action})$

2. The transition as a whole must be associated with a least one message, the one that corresponds to the transition triggering event, but transition segment can be empty.

Function

In the RT-TROOP modeling process, we associate a ROOMChart function with each MSC action. The function defines an implementation for the action defined in the MSC model. However, not all functions are associated with MSC actions. Some functions are defined in ROOMChart models for different purposes, e.g. data manipulation. Therefore, we define an optional (0,1) traceability relation between ROOMChart functions and MSC actions.

(TR57) ROOMChart_function $tr \rightarrow_{0,1}$ MSC_action

State Entry and Exit Action

Part of the actions executed on a transition can also be placed in state entry and exit actions. These actions that are executed at the very beginning or at the very end of the transition can contain the same type of actions as the ones contained in the transition themselves. The traceability relations between ROOMChart state entry and exit actions and MSC elements are described as follows.

(TR58) ROOMChart_stateEntryAction $tr \rightarrow_{0,1}$ (MSC_message | MSC_action)

(TR59) ROOMChart_stateExitAction $tr \rightarrow_{0,1}$ (MSC_message | MSC_action)

Timer

ROOMChart also allows for the use of timing services. In order to use timing services, users must define a Timing sap in the ROOMChart model of the actor that requires the services. In RT-TROOP, we define a timing sap in the ROOMChart of an actor every time the MSC instance that corresponds to the actor uses a timer in the MSC model. To capture this, we define a traceability relation between ROOMChart timing saps and MSC timers. This relation is optional because in some cases timing services may be defined in ROOMChart models even if no timer is defined in the MSC model.

(TR60) ROOMChart_timerSap $tr \rightarrow_{0,1}$ MSC_timer

6.5.2 Inter-Version Traceability in ROOM Models

During an iteration, any element of a ROOM model that existed in ROOM_model_{n-1} may remain intact, or be decomposed in a set of sub-elements. New elements may also be introduced.

To keep track of the modifications that are made to a ROOM model in the different iterations, we define a set of inter-version traceability relations between elements of ROOM models. This allows linking elements of different versions of a ROOM model. As previously discussed, these relations are backward ones, i.e. they are directionally defined from elements of ROOM_model_n to elements of ROOM_model_{n-1}.

In this section, we separately discuss inter-version traceability in ROOM structure models and in ROOMChart models.

Inter-Version Traceability in ROOMChart Models

ROOM structure models are composed of three types of elements: actors, ports, and bindings. During an iteration, the following operations may be done on structure elements.

- A new element may be created, in which case there is no traceability relation with elements of ROOM_model_{n-1}.
- An element of ROOM_model_{n-1} may be used as is in ROOM_model_n, in which case there is a one-to-one traceability relation between the element in the versions.

- An element of ROOM_model_{n-1} is decomposed in a set of sub-elements in ROOM_model_n , in which case there is a one-to-one traceability relation between each of the resulting elements in ROOM_model_n and the element of ROOM_model_{n-1} to which the decomposition has been applied.

This result in the following three traceability relations.

(TR61) $\text{ROOM_actor } tr \rightarrow_{0,1} \text{ROOM_actor}$

(TR62) $\text{ROOM_actorPort } tr \rightarrow_{0,1} \text{ROOM_actorPort}$

(TR63) $\text{ROOM_binding } tr \rightarrow_{0,1} \text{ROOM_binding}$

Inter-Version Traceability in ROOM Structure Models

State machines

During an iteration, a new state machine may be defined to satisfy new scenarios and plugged-in a state. It may also be defined as a result of the restructuring of a ROOMChart model.

(TR64) $\text{ROOMChart_stateMachine } tr \rightarrow \text{ROOMChart_stateMachine}$

States

New states may be introduced.

(TR65) $\text{ROOMChart_state } tr \rightarrow_{0,1} \text{ROOMChart_state}$

Transitions

During an iteration, a transition segment may be decomposed into a sequence of smaller segments. This occurs when restructuring ROOMChart models.

(TR66) ROOMChart_transitionSegment $tr \rightarrow_{0,1}$ ROOMChart_transitionSegment

New transition triggering events and transition actions may be introduced on a transition.

(TR67) ROOMChart_transitionTriggering $tr \rightarrow_{0,1}$ ROOMChart_transitionTriggering

(TR68) ROOMChart_transitionAction $tr \rightarrow_{0,1}$ ROOMChart_transitionAction

Functions

New functions may be introduced.

(TR69) ROOMChart_function $tr \rightarrow_{0,1}$ ROOMChart_function

State Entry and Exit Actions

New state entry and exit actions may be introduced in a state.

(TR70) ROOMChart_stateEntryAction $tr \rightarrow_{0,1}$ ROOMChart_stateEntryAction

(TR71) ROOMChart_stateExitAction $tr \rightarrow_{0,1}$ ROOMChart_stateExitAction

Timer

New timer saps may be introduced. Timer saps may be introduced in association with a MSC timer, or it may be introduced as a result of a design decision at the ROOM level.

(TR72) ROOMChart_timerSap $tr \rightarrow_{0,1}$ ROOMChart_timerSap

6.5.3 Design Decision Traceability in ROOM Models

Similarly as in the other models, we define traceability relations between design decisions and requirements. The cardinality is zero when the design decision is not linked to any

requirements, i.e. it results of a pure design decision. It is one or more when the design decision is justified by some requirements.

ROOM_designDecision $tr \rightarrow_{0,..,*}$ requirement

In ROOM modeling (both at the structure and at the behavior level), a design decision may consist in applying a design pattern.

Also, in order to maintain traceability between elements resulting from ROOM model transformations and design decisions, we define a traceability relation between ROOM element and the design decision that resulted in its creation. The 0,1 cardinality reflects the fact that a ROOM element may be the result of a design decision taken in the ROOM model (cardinality 1), or it may come directly from another model (cardinality 0).

ROOM_element $tr \rightarrow$ ROOM_designDecision

6.6 Summary

In this chapter, we defined three different types of traceability relations: inter-model traceability, which establishes traceability links between elements of different models, inter-version traceability, which establishes traceability links between elements of different versions of a single model, and design decision traceability, which establishes traceability links between the rationale of a design decision and specific requirements that led to the decision.

Then, we used these three types of traceability relations to define a set of fine-grained traceability relations for each of the models used in the RT-TROOP modeling process, i.e. STD, UCM, MSC, ROOM.

The resulting set of traceability relations allows maintaining consistency between the different models used in the RT-TROOP modeling process. It also allows evaluating the impact of requirement modification on the different model elements.

CHAPTER 7 Conclusion

As described in section 1.3, this thesis addresses the difficult problem of defining a systematic and traceable progression between scenario textual descriptions and communicating hierarchical state machine models in the context of complex real-time system design. In this chapter, we summarize the main contributions of the thesis, discuss our experience implementing them, discuss their impact on the discipline of software engineering, and give a list of topics for future research.

7.1 Summary of Thesis Contributions

In this section, we summarize the five main contributions of the thesis.

7.1.1 Definition of the RT-TROOP Modeling Process

In this thesis, we defined the RT-TROOP modeling process. This process allows for a systematic and traceable progression between scenario textual descriptions to a communicating hierarchical state machine model. It also allows establishing and maintaining traceability between requirements and implementation.

The RT-TROOP modeling process is defined in terms of a set of models and a set of modeling phases. The models include STD, UCM, MSC, and ROOM. The set of modeling phases contains both in-model modeling phases and model transition phases. The in-model modeling phases defines the types of model transformations that may be carried out in the different models. The model transition phases define how the transition between models can be made in a consistent manner. They define semantic relationships between elements of the different models, and define a set of steps, each addressing a specific aspect of real-time system design, that allows making the transitions in a systematic manner.

The main contribution of the RT-TROOP process is a first cut at a comprehensive process going from requirements, or more specifically from a set of scenario textual descriptions, to an executable model from which implementation can be automatically generated. Some parts of this process are described at a more detailed level than others, and constitute the technical contributions of this work.

7.1.2 Integration of the UCM Modeling Technique

The RT-TROOP modeling process defined in this thesis integrates the UCM modeling technique in a concrete modeling process. The set of modeling phases defined in RT-TROOP establishes semantics relationships between UCM on one side and STD, MSC, and ROOM on the other side. This by itself constitutes a main contribution of the thesis as it corresponds to a need often raised by industrial software engineers interested in using UCM.

In particular, we defined a concrete transition method between UCM and MSC. This method allows automatically generating a HMSC (High-level MSC) from a UCM related path set. It also allows defining a basic MSC from a UCM path segment in a systematic and traceable manner. In this step, each MSC element is linked back to a UCM element.

The main task of a designer here consists in refining each UCM responsibility as a sequence of MSC messages and actions.

In the context of the unification of modeling techniques, where UML becomes the standard, the integration of UCM in a concrete modeling process is crucial for the future of UCM. Because of the close semantic relationships that exists between ROOM and MSC models, and UML models, we believe that the research results of this thesis could provide the basis for the integration of the UCM modeling technique in UML.

7.1.3 Definition of Behavior Integration Patterns

This thesis proposes a set of process and design patterns to help designers making the transition between scenario models and hierarchical state machines. These patterns allow establishing a strong relationship between scenario models and hierarchical state machine structures. The set of patterns defined in this thesis includes patterns that deal with different aspects of hierarchical state machine design such as scenario partitioning, state machine integration, state machine structuring, and different types of scenario interactions.

The patterns we defined use a two-step approach. In the first step, the detail-level scenario descriptions provided by interaction diagrams is used to define state machines on a per scenario basis. In the context of the RT-TROOP modeling process, these state machines correspond to the role behavior state machines that are defined in the transition from MSC to ROOMChart (section 3.11). In the second step, inter-scenario relationship information, such as the scenario interaction information contained in the UCM model, is used to compose the state machines obtained in the first step into more complex hierarchical state machines.

The definition of these design patterns is an important contribution of this thesis. There are, to our knowledge, no other research projects that address this problem in the context of concurrent and interacting scenarios. We believe that the use of such patterns increases component behavior maintainability and extensibility.

7.1.4 Traceability Relations

In this thesis, we define three different types of traceability relations: *inter-model traceability*, which establishes traceability links between elements of different models, *inter-version traceability*, which establishes traceability links between elements of different versions of a single model, and *design decision traceability*, which establishes traceability links between the rationale of a design decision and specific requirements that led to (or that justify) the decision.

We then used these three types of traceability relations to define fine-grained traceability relations for STD, UCM, MSC, and ROOM in the context of the RT-TROOP modeling process.

The resulting relations could be implemented in a tool to facilitate the maintenance of consistency between models. This would allow evaluating the impact of modifications on the different models.

7.1.5 Development of Case Studies

In this thesis, a simple printer system case study has been developed to illustrate the different phases of the RT-TROOP modeling process. The development of this simple system allowed us to illustrate the systematic and traceable aspects of RT-TROOP modeling. In

the context this research project, several other systems have also been developed in both an industrial and academic context. Some of them have been the subject of reports ([72], and [73]), while others are still under development. The development of these different systems is by itself an important contribution of the thesis.

7.2 Implementation Results

In this section, we briefly relate our experience in implementing the RT-TROOP modeling process, the behavior integration patterns, and the traceability relations.

7.2.1 Implementation of the Modeling Process

The RT-TROOP modeling process is a concrete modeling process that has already been implemented in an industrial development process at CML Technologie¹s. Another implementation of the RT-TROOP modeling process is also underway at CRC (Communication Research Center) in a satellite communication project. In both of these industrial projects, RT-TROOP is used as the core of the development process. It has also been used by several undergraduate and graduate students in the context of different projects.

1. CML Technologies is a 150 employees company specialized in the development of telecommunication systems such as mobile radio consoles, air traffic control communications, Enhanced 9-1-1 emergency calling systems, and other specialized switching systems for customized computer telephony applications.

CML Technologies Project

The project at CML Technologies, which started in September 1998, aims at developing a new generation of programmable telephone switches for enhanced telephony services. This project, which is evaluated to between three and four million dollars over a period of two years, involves a team of twelve software engineers. It involves both software and hardware development. CML Technologies and the team of software engineers involved in the project had no previous background in object-oriented software development.

In this project, the RT-TROOP modeling process has been implemented using the Rational RequisitePro requirement management tool [81] and the ObjecTime Developer Toolset [68]. Because of the non-availability of a UCM tool on a Windows platform, the CML Technologies implementation of RT-TROOP does not formally include the UCM modeling technique. However, UCM are used informally in architecture meetings to capture scenario interactions. The scenario interaction information is maintained in a textual format. In this project, the ObjecTime Testscope testing tool [69] is also used for the purpose testing.

The feedback we have so far from CML Technologies senior managers and software engineers involved in the project is very positive. Senior managers are particularly impressed by the speed of development and by the quality of the documentation. The software engineers on the other hand believe that the systematic nature of the modeling process significantly simplifies their task.

CRC Satellite Communication Project

The project at CRC, started May 1999, aims at developing a satellite communication control system.

In this project, the full RT-TROOP modeling process will be implemented, including UCM. This implementation of RT-TROOP will be done using a set of tools that includes

the Rational RequisitePro [81] requirements management tool, the UCM Navigator tool [65], the new Rational Rose for Real-Time toolset [80], and the ObjecTime Testscope [69] testing tool.

Implementation Results

Experience with the integration of the RT-TROOP modeling process in an industrial context showed that the process can be quickly understood and applied. The systematic approach helps inexperienced designers understand the different issues. It also helps them designing system in a more rigorous manner. Once they understand the relationships between the different models, they can better explain and justify their design decisions in the context of the overall set of system models.

Overhead

Implementation of the process showed that the overhead of the process is very low if supported by tools. It also showed that the overhead of producing different models is overcome by the increase in system understandability provided by the different levels of abstraction of the different models.

Maintainability

The different levels of abstraction provided by the different models significantly increase system understandability and maintainability. For example, when a designer wants to understand the behavior of a set of components in the context of a certain scenario, s/he can refer to the customized MSC model to have all the details about the behavior of each component for the specific scenario. The fact that the MSC are always up-to-date simplifies considerably this type of activity.

Testing

The customized MSC model produced during the design phase of the system can be used to test the final ROOM model of the system. Tools like the ObjecTime Testscope tool can be used for this purpose.

7.2.2 Implementation of the Behavior Integration Patterns

The use of the behavior integration patterns in both industrial and academic contexts has been very conclusive.

In an industrial context, it simplified the task of integrating new scenarios into an existing system. It also allowed obtaining a consistent structuring of the hierarchical state machines across out the members of the development team, which increases the understandability of complex component hierarchical state machines.

A direct impact of the use of the patterns is a significant reduction of the time required to conduct design reviews by the system architects. The latter can navigate more easily in component behavior state machines to find required information. We believe that if the use of the design patterns helps the architects to find required information faster in a relatively new system, the benefits will be even higher as the system ages and gets more and more complex.

In the academic context, it significantly facilitates the teaching of complex component behavior design. The feedback we have from students is very positive. Students say that the design patterns help them understand the rational behind complex component behavior structuring.

7.2.3 Implementation of the Traceability Relations

In the CML Technologies project, because of the lack of support for inter-model traceability relations in the ObjecTime Developer toolset, only part of the traceability relations have been implemented: backward traceability to requirements, and backward traceability to STDs. The implemented traceability relations have shown to be particularly useful for the change impact analysis. However, it is too early to evaluate the long-term impact of the traceability relations on maintainability and extensibility.

7.3 Impact of Contributions

In this section, we discuss the impact of the thesis contributions on the work of experienced designers, inexperienced designers, and tool development.

Experienced Designers

From the experience we have using the RT-TROOP modeling process in an industrial context, we believe that the use of a systematic and traceable process can significantly facilitate system modifications by maintaining explicit relationships between model elements.

We also believe that the definition of design patterns to perform the transitions between scenario models and communicating hierarchical state machines can facilitate the task of industrial designers (experienced and inexperienced) and increase the reuse of standard solutions. They can also facilitate the uniform application of a methodology in the context of large development projects.

Inexperienced Designers

Teaching design to inexperienced designers is a difficult task. We believe that research results of this thesis will facilitate this task in three ways. First, the definition of a set of model transition techniques will help designers understand the transition between models as a set of steps, each involving different design issues. Second, the definition of a systematic and traceable modeling process will help them understand the transition between requirements and communicating hierarchical state machines as a rational process. Third, the definition of the design patterns will give them access to hierarchical state machine models developed by design expert, and thus give them access to expert knowledge that otherwise takes years to obtain.

Tool Development

With respect to tool development, we believe that the set of traceability relations defined in this thesis between STD, UCM, MSC and ROOM could be used as a basis for the integration of the four modeling techniques in a CASE tool. It could also be used as a basis for the definition of a formal framework for the verification of communicating hierarchical state machine models against scenario models. Research for the development of such a framework is under way.

7.4 Future Work

This section gives a list of future research work triggered by the research of this thesis.

- Adapt the RT-TROOP modeling process to UML models
- Define more behavior integration patterns

Define new integration patterns and add them to the existing set. The objective here is to develop a catalogue of patterns that could be used in the context of hierarchical state machine design.

The existing catalogue could also be customized for specific type of applications.

- Define a set of test cases for each design patterns

This would allow standardizing the testing of each design patterns and thus reducing the chances of errors in systems.

This way, testing experience can be shared among designers. Thus, the set of test cases associated with a design pattern could grow incrementally with the use of a pattern. A designer that uses a design pattern for the first time would have access to the test cases that have been developed over time by other designers.

- AugmentRT-TROOP with a testing process that would allow for the testing of sets of concurrent and interacting scenarios.

The importance of model verification and testing is widely recognized. In the last decades many different testing methods have been defined. State machine testing in particular has been the subject of many papers. More recently, with the growing popularity of scenario-driven (or use-case driven) approaches, several researchers have proposed using scenarios as a basis for system testing.

However, one aspect of system testing that is not well covered in current testing methods (approaches) is the one of scenario concurrency and interactions. Yet, scenario concurrency and interaction are two important causes of errors in real-time systems. Moreover, errors resulting from the concurrent execution of sets of scenario are often very difficult to identify.

In this context, one of our long-term objective is to define a testing process that aims at generating test components from sets of scenarios in the overall context of interaction and concurrency.

- Implement the traceability relations in a tool to increase the level of system traceability

Currently the ObjecTime Developer only provides for requirement traceability. We cannot, for example, establish traceability relationships between a message arrow in an MSC model and a scenario responsibility, or between a ROOMChart transition and a message arrow in an MSC model.

The inter-model traceability relations defined in this thesis could be implemented in a tool, like the ObjecTime Developer toolset or the new Rational Rose for Real-Time, to enable inter-model traceability.

- More implementations of the RT-TROOP modeling process
- Implementation of the RT-TROOP modeling process in the context of agent systems

The agent system group at Mitel Corporation showed a strong interest in applying the RT-TROOP modeling process for agent system development. This would require adapting the RT-TROOP modeling process to the specific context of agent systems.

References

- [1] R. Alur, T. Henzinger. "Real-Time Logics: Complexity and Expressiveness", *Inform. and Computation*. Vol. 104, No. 1, 1993, pp. 35-77
- [2] D. Amyot, A. Miga. *A Use Case Map Linear Form in XML, version 0.12*. <http://www.UseCaseMaps.org/UseCaseMaps/xml>, April 1999.
- [3] D. Amyot, F. Bordeleau, R.J.A. Buhr, L. Logrippo. "Formal Support for Design Techniques: a Timethreads-Lotos Approach", *Proceedings of the 8th International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE'95)*. October 17-20 1995, Montreal, Quebec, Canada.
- [4] D. Amyot. *Formalization of Timethreads Using LOTOS*. Master Thesis, Department of Computer Science, University of Ottawa, Canada, 1994.
- [5] M. Andersson, J. Bergstrand. *Formalizing Use Cases with Message Sequence Charts*. Master Thesis, Department of Communication Systems, Lund Institute of Technology, Sweden, 1995.
- [6] M. Barbeau, F. Kabanza, R. St-Denis. "A Method for the Synthesis of Controllers to Handle Safety, Liveness, and Real-Time Constraints", *IEEE Transactions on Automatic Control*. Vol. 43, No. 11, November 1998, pp. 1543-1559.
- [7] O. Basset. *Implementation of a ROOM Interpretation Method for ROOM Models*. Projet de fin d'études, Département d'Informatique, Institut National des Sciences Appliquées de Lyon, Lyon, France, 1996.
- [8] K. Beck, W. Cunningham. "A Laboratory For Teaching Object-Oriented Thinking", *Special issue of SIGPLAN Notices*, Volume 10, October 1989.
- [9] M. Bienvenue. "System Design in ObjecTime", *Byte Magazine*, December 1995, pp. 189-190.

- [10] G.v. Bochmann. <http://www.site.uottawa.ca/~bochmann>.
- [11] G. Booch. *Object-Oriented Design*. Benjamin/Cummings, 1994.
- [12] Grady Booch, James Rumbaugh, *Unified Method for Object-Oriented Development*, Documentation Set, Version 0.8, Rational Software Corporation, 1995.
- [13] G. Booch, I. Jacobson, J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
- [14] E. Brinksma et al. "A Formal Approach to Conformance Testing", *Proceedings of 2nd Workshop on Protocol Test Systems*. Berlin, FRG, October 1989.
- [15] F. Bordeleau, R.J.A. Buhr. "UCM-ROOM Modeling: From Use-Case Maps to Communicating State Machines", *Proceedings of IEEE Conference and Workshop on Engineering of Computer-Based Systems (ECBS'97)*. March 24-28 1997, Monterey, California.
- [16] F. Bordeleau. *Visual Descriptions, Formalisms, and the Design Process*. Master Thesis, School of Computer Science, Carleton University, Ottawa, Canada, 1993.
- [17] R.J.A. Buhr. *System Design with Ada*. Prentice Hall, 1984.
- [18] R. J. A. Buhr, R. S. Casselman. *Use Case Maps for Object-Oriented Systems*. Prentice Hall, 1996
- [19] R.J.A. Buhr, A. Hubbard. "Use Case Maps for Engineering Real Time and Distributed Computer Systems: A Case Study of an ACE-Framework Application", *Hawaii International Conference on System Sciences (HICSS'97)*, Jan 7-10, 1997, Wailea, Hawaii, <http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/hicss-final-public.ps>
- [20] R.J.A. Buhr, R.S. Casselman, T.W. Pearce. *Design Patterns with Use Case Maps: A Case Study in Reengineering an Object-Oriented Framework*, SCE 95-17, <http://ftp.sce.carleton.ca/UseCaseMaps/dpwucm.ps>.
- [21] R.J.A. Buhr. "Use Case Maps for Attributing Behaviour to Architecture", *Fourth International Workshop on Parallel and Distributed Real Time Systems (WPDRTS)*,

- April 15-16, 1996, Honolulu, Hawaii, <http://www.sce.carleton.ca/ftp/pub/Use-CaseMaps/attributing.ps>.
- [22] R.J.A. Buhr. "Design Patterns at Different Scales", *PLoP96*, Allerton Park Illinois, Sep 96. <http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/plop.ps>.
- [23] R.J.A. Buhr, G.M. Karam, G.M. Woodside, R. Casselman, G. Franks, H. Scott, D. Bailey. *TimeBench: A CAD Tool for Real-Time System Design*. Department of Systems and Computer Engineering, Carleton University, Ottawa, 1990.
- [24] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *A System of Patterns*. Wiley, 1996.
- [25] W.J. Brown, R.C. Malveau, H.W. McCormick III, T.J. Mowbray. *Anti Patterns, Refactoring Software, Architectures, and Projectin Crisis*. Wiley, 1998.
- [26] N.P. Capper, R.J. Colgate, J.C. Hunter, M.F. James. "The Impact of Object-Oriented Technology on Software Quality: Three Case Histories", *IBM System Journal* (33). Vol. 33, No. 1, 1994, pp. 131-157.
- [27] J.P. Corriveau. "Traceability Process for Large OO Project"s, *IEEE Computer*. Vol. 29, No. 9, September 1996, pp. 63-68.
- [28] D. de Champeaux, D. Lea, P. Faure. *Object-Oriented System Development*. Addison-Wesley, 1993.
- [29] R. Domges, K. Pohl. "Adapting Traceability Environments to Project-Specific Needs", *Communications of the ACM*. Vol. 41, No. 2, December 1998.
- [30] D.F. D'Souza, A.C. Wills. *Objects, Components, and Frameworks with UML, The Catalysis Approach*. Addison-Wesley, Object Technology Series, 1999.
- [31] ELUDO (Environement Lotos de l'Universite d'Ottawa). <http://Lotos.site.uottawa.ca>.
- [32] M.E. Fayad, W.-T. Tsai (Guest Editors). "Object-Oriented Experiences", *Communications of the ACM*. Vol. 38, No. 10, December 1995, pp. 51-53.

- [33] D.H. Firesmith, B. Henderson-Sellers, I. Graham, M. Page-Jones. *Open Modeling Language (OML) Reference Manual*. Addison-Wesley, 1998.
- [34] P. Freedman, J.-M. Goutal, Y. Leborgne, D. Gaudreau. *Etude sur des techniques, des methodes, et outils de developpement de logiciel de systemes temps reel*. Collection scientifique et technique du CRIM, no95/12-40, December 1995.
- [35] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns-Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [36] D. Harel. "StateCharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, Vol. 8, 1987, pp. 231-274.
- [37] D. Harel, A. Pnueli, J.P. Schmidt, R. Sherman. "On the formal scemantics of state-charts", *Proceedings of the second IEEE Symposium on Logic in Computer Science*. New York, 1997.
- [38] D. Harel. "On Visual Formalisms", *Communication of the ACM*. Vol. 31, No. 5, 1988, pp. 514-530.
- [39] O. Haugen. *MSC Methodology*. SISU II Report L-1313-7, Oslo, Norway, 1994.
- [40] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1990.
- [41] G. Holzmann. *The Theory and Practice of a Formal Method: NewCore*. Internal Report, At&T Laboratories, Murray Hill, New Jersey 07974, USA.
- [42] ISO-IEC/JTC1/SC21/WG1/FDT/C. *LOTOS, A Formal Description Technique Based on Temporal Ordering of Observational Behavior*. ISO International Standard IS807, February 1989.
- [43] ITU (1993). *Message Sequence Charts (MSC'93)*. Recommendation Z.120. Geneva.
- [44] ITU (1996). *Message Sequence Charts (MSC'96)*. Recommendation Z.120. Geneva.
- [45] ITU (1988). *Specification and Description Language (SDL)*. Recommendation Z.100. Geneva.
- [46] ITU (1992). *Specification and Description Language (SDL'92)*. Recommendation

- Z.100. Geneva.
- [47] I. Jacobson et al. *Object-Oriented Software Engineering (A Use Case Driven Approach)*. ACM Press, Addison-Wesley, 1992.
- [48] I. Jacobson, G. Booch, J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
- [49] M. Jarke (Guest Editor). "Requirements Tracing", *Communications of the ACM*. Vol. 41, No. 2, December 1998.
- [50] F. Kabanza, M. Barbeau, R. St-Denis. "Planning Control Rules for Reactive Agents", *Artificial Intelligence*, Vol. 95, 1997, pp. 67-113.
- [51] K. Koskimies, T. Mannistö, T. Systä, J. Tuomi. *On the Role of Scenarios in Object-Oriented Software Design*, Technical Report A-1996-1, Department of Computer Science, University of Tampere, Tampere, Finland.
- [52] K. Koskimies, T. Mannistö, T. Systä, J. Tuomi. *Automated Support for Dynamic Modeling of Object-Oriented Software*. Department of Computer Science, University of Tampere, Tampere, Finland. <http://www.uta.fi/~cstasy/scedpage.html>.
- [53] K. Koskimies, E. Makinen. "Automatic Synthesis of State Machines from Trace Diagrams". *Software Practice and Experience*, vol.24, No. 7, July 1994, pp. 643-658.
- [54] K. Koskimies, E. Makinen. *Inferring State Machines from Trace Diagrams*. Technical Report A-1993-3, Department of Computer Science, University of Tampere, Tampere, Finland.
- [55] P. Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley, 1999.
- [56] D. Lea. Doug Lea's Home Page. <http://g.oswego.edu/dl/>.
- [57] D. Lea. *Concurrent Programming in Java, Design Principles and Patterns*. Addison-Wesley, 1996.
- [58] S. Leue, L. Mehrmann, M. Rezai. *Synthesizing ROOM Models From Message Sequence Charts Specifications*. TR98-06, Department of Electric and Computer Engi-

- neering, University of Waterloo, Waterloo, Canada, 1998.
- [59] S. Leue and P.B. Ladkin. "Implementing and Verifying MSC Specifications Using Promela/XSpin". In: *J.-C. Grégoire, G. Holzmann and D. Peled (eds.), Proceedings of the DIMACS Workshop SPIN96, the 2nd International Workshop on the SPIN Verification System*. DIMACS Series Volume 32, American Mathematical Society, Providence, R.I., 1997
- [60] S. Leue and P.B. Ladkin. "Implementing and Verifying Scenario-Based Specifications Using Promela/XSpin", *Proceedings of the Second SPIN Workshop*. Rutgers University, New Brunswick, New Jersey, August 1996.
- [61] S. Leue. *Methods and Semantics for Telecommunications Systems Engineering*. Ph. D. Thesis. University of Berne, Switzerland, 1995.
- [62] A. Lyons. *UML for Real-Time Overview*. <http://www.objectime.com>, 1999.
- [63] Z. Manna, A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, 1991.
- [64] M. Makungu, M. Barbeau, R. St-Denis. "Synthesis of Controllers of Process Modeled as Colored Petri Nets", *Discrete Events Dynamic Systems: Theory and Applications*, Vol. 9, 1999, pp. 147-169.
- [65] A. Miga. Use Case Map Navigator. <http://www.usecasemaps.org/>.
- [66] J.D. McGregor, T.D. Korson. "Integrated Object-Oriented Testing and Development Processes", *Communications of the ACM*, Vol. 37, No. 9, September 1994, pp. 59-77.
- [67] T.J. Mowbray, R.C. Malveau. *Corba Design Patterns*. Wiley, 1997.
- [68] ObjecTime Limited. ObjecTime Developer. <http://www.objectime.com>.
- [69] ObjecTime Limited. TestScope. <http://www.objectime.com>.
- [70] A. Oliveira, S. Edwards. *Inference of State Machines from Examples of Behavior*. Technical Report, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, February 1995.

- [71] Object Management Group (OMG). *OMG Unified Modeling Language Specification, Version 1.3R9 (Draft)*. January 1999.
- [72] M. Perisic. *Adding Structure Dynamics to UCM-ROOM*. Honours project, Department of Systems and Computer Engineering, Carleton University, Ottawa, 1997.
- [73] S. Picard. *Development of a House Security System Using the RT-TROOP Modeling Process*. Projet de fin d'études, Département d'Informatique, Institut National des Sciences Appliquées de Lyon, Lyon, France, 1996.
- [74] L.F. Pires. *Architectural Notes: a Framework for Distributed System Development*. Ph. D. Thesis, University of Twente, 1994.
- [75] R. Pooley, P. Stevens. *Using UML: Software Engineering with Objects and Components*. Addison Wesley, 1999.
- [76] B.R. Douglas. *Real-Time UML, Developing Efficient Objects for Embedded Systems*. Addison-Wesley, 1998.
- [77] R.L. Probert, H. Ural, M.W.A. Hornbeek. "An Integrated Software Environment for Developing and Validating Standardized Conformance Tests", *Proceedings of 8th International IFIP Symposium on Protocol Specification, Testing, and Validation*. pp. 87-98, Atlantic City, NJ, 1988.
- [78] P.J.G. Ramage, W.M. Wonham. "The Control of Discrete Event Systems", *Proceedings of IEEE*, vol. 77, pp. 81-98, 1989.
- [79] B. Ramesh. "Factors Influencing Requirements Traceability Practice", *Communications of the ACM*. Vol. 41, No. 2, December 1998.
- [80] Rational Corporation. Rational Rose Real-Time Toolset. <http://www.rational.com>
- [81] Rational Corporation. Rational RequisitePro. <http://www.rational.com>
- [82] B. Regnell, M. Andersson, J. Bergstrand, "A Hierarchical Use Case Model with Graphical Representation", *Proceedings of IEEE Second International Symposium and Workshop on Engineering of Computer Based Systems (ECBS96)*. March 1996.

- [83] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorenson. *Object-Oriented Modelling and Design*, Prentice Hall 1991.
- [84] B. Rumpe. *Formal Method for Design of Distributed Object-Oriented Systems*. Ph. D. Thesis, Munich University of Technology, 1997.
- [85] D. Schmidt. <http://siesta.cs.wustl.edu/~schmidt/patterns-info.html>.
- [86] G. Schneider, J.P. Winters. *Applying Use Cases: A Practical Guide*. Addison-Wesley, 1998
- [87] J. Schot. *The Role of Architectural Semantics in the Formal Approach of Distributed Systems Design*. Ph. D. Thesis, University of Twente, 1990.
- [88] B. Selic, J. Rambaugh. *Using UML for Modeling Complex Real-Time Systems*. ObjecTime Limited, <http://www.objecttime.com>, 1999.
- [89] B. Selic. *Modeling Real-Time Distributed Software Systems*. ObjecTime Limited, <http://www.objecttime.com>, 1998.
- [90] B. Selic. *A Framework for Location Transparency in Distributed Systems*. ObjecTime Limited, <http://www.objecttime.com>, 1997.
- [91] B. Selic. "Recursive Control", In R. Martin at al. (eds.), *Patterns Languages of Program Design 3*. Addison-Wesley, 1998, pp. 147-162.
- [92] B. Selic. *Automatic Generation of Test Drivers From MSC Specs*. ObjecTime Limited, <http://www.objecttime.com>, 1996.
- [93] B. Selic, G. Gullickson and P.T. Ward. *Real-time Object-Oriented Modeling*. Wiley, 1994.
- [94] M. Shaw, D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
- [95] S.S. Somé. *La dérivation de spécification à partir de Scénario d'interactions*. Ph.D. Thesis, Department of Computer Science and Operational Research, University of Montreal, Canada, 1997.

- [96] J.M. Vlissides, J.O. Coplien, N.L. Kerth. *Pattern Languages of Program Design*. Addison-Wesley, 1996.
- [97] B. Wilhelm. *Designing for Concurrency*. <http://www.objecttime.com>, 1999.