

Humberto Nicolás Castejón

# Collaborations in Service Engineering:

Modeling, Analysis and Execution

Thesis for the degree of Philosophiae Doctor

Trondheim, November 2008

Norwegian University of Science and Technology  
Faculty of Information Technology, Mathematics and  
Electrical Engineering  
Department of Telematics



Norwegian University of  
Science and Technology

**NTNU**

Norwegian University of Science and Technology

Thesis for the degree of Philosophiae Doctor

Faculty of Information Technology, Mathematics and  
Electrical Engineering  
Department of Telematics

© Humberto Nicolás Castejón

ISBN 978-82-471-1275-5 (printed ver.)  
ISBN 978-82-471-1276-2 (electronic ver.)  
ISSN 1503-8181

Doctoral theses at NTNU, 2008:287

Printed by NTNU-trykk

*Para Nicolás y Olimpia, mis padres, con todo mi amor*

*Los días que pasan,  
Las luces del alba,  
Mi alma, mi cuerpo, mi voz, no sirven de nada  
Qué no daría yo por tener tu mirada,  
Por ser como siempre los dos  
Mientras todo cambia  
Porque yo sin ti no soy nada  
Sin ti no soy nada  
Sin ti no soy nada*

Amaral



---

# Abstract

The development of *distributed reactive systems* is a complex and error-prone process. These systems consist of many separate *autonomous* components that operate *concurrently* and continuously interact with each other and with the environment in order to deliver *services* to the end-users. These systems are becoming more and more sophisticated, providing an ever expanding portfolio of services. In many cases new and customizable services are considered the key to increased revenues. Being able to rapidly develop and incrementally deploy such services, while avoiding undesired interactions with already existing services, therefore becomes strategically important.

In this thesis we present a *service-oriented* and *model-driven* approach to engineering distributed reactive systems. The approach integrates model creation and analysis tightly. Services are explicitly modeled as collaborations between roles, using UML 2 collaborations. These offer powerful means to structure and reuse crosscutting system behavior, and to provide a high-level overview of it. Complex collaborations can be composed from more elementary ones, whose behavior can be completely described using sequence diagrams. The full behavior of a composite collaboration can in turn be described using a *choreography graph*, which defines the global execution ordering of its sub-collaborations. For each service collaboration, the state machine behavior of its roles can be automatically synthesized. These roles are then assigned to the components that will play them. The subsequent composition and coordination of such roles to form complete component behavior may call for human assistance.

Having confidence in the correctness of the models being created is important. For that reason we identify some good practices in service modeling. We also propose early analysis techniques that help to uncover undesired or overlooked behaviors at two critical points during the modeling process. First, each service model is analyzed separately in search of *realizability* problems. That is, pathologies that may cause the joint behavior of the set of distributed roles synthesized from the service model to be different from the global behavior specified in the model. We provide algorithms to detect some realizability problems. Moreover, we discuss the actual nature of those problems and their underlying causes. This is important not only to adopt the most appropriate resolution when problems are detected, but also to avoid them in the first place. Second, the potential interactions between the roles that any component may play are analyzed, in order to adopt appropriate coordination measures.

Finally, we recognize that service discovery and adaptation, and service personalization based on the service context and end-user preferences are increasingly important mechanisms. In this area we present *policy-based mechanisms* to allow personalization of services and runtime adaptation.

---

## Acknowledgments

I would like to express my most sincere gratitude to all the people that in one or another way have contributed to make this thesis a reality. First of all I would like to thank Rolv Bræk, my supervisor. He is an outstanding researcher and has been a continuous source of inspiration for me. I admire his profound knowledge of systems and services engineering, as well as his capacity to analyze problems. I also admire him as a person: his kindness, his love for nature and his ability to see the positive side of things. He is such a wonderful person! He believed in me from the very beginning of this “trip”, and was always there to encourage and support me in the difficult moments. For all this and much more ... *tusen hjertelig takk Rolv!* Thanks also to Turid, his wife, for all the good moments we have shared on the conference trips.

Special thanks go to Gregor von Bochmann, whom I had the luck to collaborate with during the last part of my PhD, and to the members of my committee, Prof. Tarja Systä, Prof. Joachim Fischer and Prof. Peter Herrmann, for gracefully accepting to review this work.

I am thankful to Jacqueline Floch and Josip Zoric for his support. I am also grateful to all my colleagues at the Department of Telematics for creating such a wonderful working environment. In particular, I would like to thank Randi and Mona for making simple all the administrative work, as well as Pål and Asbjørn for being always there when my computer was refusing to run. I am also thankful to my fellow doctoral students Frank, Fritjof, Haldor, Judith, Mazen, Paramai, Richard, Shanshan and Sune for many interesting discussions. Paramai and Mazen have also been great friends with whom to talk. I enjoyed a lot sharing office with Richard Sanders, while he was finishing his own PhD. He has always been supportive. Cyril Carrez, my officemate during the last years, deserves a special mention (well, do not believe it much; you know, I am just trying to be polite). He has always been a great source of help, and has provided me with very useful feedback on initial drafts of this document. I believe that working is much better when combined with fun, and we probably had the most funny office in the whole department! *Merci Cyril!*

Working far from home in a foreign country can be difficult at times. Fortunately I have been lucky to have a wonderful group of friends that made me feel like at home. I thank Chin Yu, Tore, Dionne, Dani, Esther, Øyvind, Sara, Markus, and the new team members, Elías and Ivar, for all the great moments we have had together. A big *Gracias!* goes to Javi. He has been a true friend, always there, listening to me in the good and the bad moments. I am also extremely grateful to Sigmund, for

being like a father for me in Trondheim, and to Anne Lise.

My family deserves the biggest of the acknowledgments. Quiero dar las gracias a mi madre, por no parar un momento de preocuparse por mi bienestar y por cuidarme en la distancia. Y a mi padre, por recordarme que no debo rendirme; por acompañarme día a día desde allí arriba y darme fuerzas. Mis hermanos, cuñados y sobrinos saben que soy parco en palabras. Por eso no es de extrañar que ahora me falten para agradecerles todo el cariño, el apoyo y los ánimos que me han dado durante todos estos años. ¡Gracias a todos!

Mil gracias a mi mujer Ruth por su constante apoyo en los momentos difíciles y por demostrarme que el verdadero amor es desinteresado. Esta tesis es en gran parte suya, pues sin ella a mi lado no lo hubiese logrado. Gracias también a mi pequeña brujilla, Ruth Astrid. Aunque ella aún no lo sabe, me ha ayudado muchísimo durante los últimos meses, pues sus sonrisas al llegar a casa recompensaban todo el esfuerzo de los largos días de trabajo.

Finally, I thank God for placing all these people on my way.

---

# Contents

<b>Abstract</b>	<b>iii</b>
<b>Acknowledgments</b>	<b>v</b>
<b>List of Figures</b>	<b>xi</b>
<b>I Overview</b>	<b>1</b>
<b>1 Introduction</b>	<b>3</b>
1.1 The Notion of Service . . . . .	3
1.2 Service Models and Design Models . . . . .	6
1.3 Contributions . . . . .	9
1.4 Current Status of Tool Support . . . . .	11
1.5 Outline . . . . .	11
<b>2 A Service Engineering Approach</b>	<b>13</b>
2.1 The overall idea . . . . .	13
2.2 Service modeling . . . . .	16
2.2.1 Behavioral interpretation of the specification . . . . .	24
2.3 Realizability of a Service Model . . . . .	25
2.4 Service role synthesis . . . . .	26
2.5 System composition . . . . .	27
<b>3 Related Work</b>	<b>33</b>
3.1 Work on service modeling . . . . .	33
3.2 Work on synthesis . . . . .	35
3.3 Work on realizability and implied scenarios . . . . .	37
<b>4 Summary of the papers</b>	<b>39</b>
<b>5 Conclusions</b>	<b>43</b>
5.1 Summary and Discussion . . . . .	43
5.2 Limitations and Future work . . . . .	47

<b>References</b>	<b>49</b>
<b>II Research Papers</b>	<b>57</b>
<b>6 Paper 1</b>	<b>61</b>
6.1 Introduction . . . . .	63
6.2 Goal-Oriented Service Collaborations . . . . .	64
6.3 UCMs for Describing the Goal-based Progress of Collaborations and their Inter-Relationships . . . . .	66
6.3.1 Basic UCM Notation . . . . .	68
6.3.2 Dependency Patterns . . . . .	69
6.4 Towards Automatic Synthesis of State-Machines . . . . .	73
6.4.1 An Example . . . . .	74
6.5 Related Work and Discussion . . . . .	76
6.6 Conclusions . . . . .	78
References . . . . .	79
6.A Synthesizing Algorithm . . . . .	81
<b>7 Paper 2</b>	<b>85</b>
7.1 Introduction . . . . .	87
7.1.1 Structure of the Paper . . . . .	90
7.2 Collaborations, Goals and Semantic Interfaces . . . . .	90
7.2.1 Collaboration Structure . . . . .	90
7.2.2 Collaboration Goals . . . . .	91
7.2.3 Collaboration Behavior . . . . .	91
7.2.4 Semantic Interfaces and Compatibility . . . . .	93
7.3 Composition from Collaborations . . . . .	95
7.3.1 Composition of Two-party Services and Semantic Interfaces from Two-party Collaborations . . . . .	95
7.3.2 Composition of Multi-party Services . . . . .	97
7.3.3 Towards Class Design . . . . .	99
7.4 Discussion . . . . .	100
7.4.1 Related Work . . . . .	100
7.4.2 Further Work . . . . .	100
7.5 Conclusion . . . . .	101
References . . . . .	101
<b>8 Paper 3</b>	<b>105</b>
8.1 Introduction . . . . .	107
8.1.1 Contributions and outline . . . . .	108
8.2 Understanding Collaborations . . . . .	109
8.3 Service Specification with Collaborations . . . . .	110
8.3.1 Case Study: A Transportation Service . . . . .	110
8.3.2 The Specification Approach in Detail . . . . .	110
8.4 Service Specification Validation: Detection of Implied Scenarios . . . . .	114

8.5	Related Work . . . . .	118
8.6	Discussion And Conclusions . . . . .	118
8.7	Acknowledgements . . . . .	119
	References . . . . .	119
8.A	Remarks to the Paper . . . . .	120
8.A.1	TransportService Collaboration Diagram . . . . .	120
8.A.2	Undetected Implied Scenarios . . . . .	121
<b>9</b>	<b>Paper 4</b>	<b>123</b>
9.1	Introduction . . . . .	125
9.2	Collaboration Goal Sequences . . . . .	127
9.2.1	Syntax for Goal Sequences . . . . .	128
9.2.2	Semantics for Goal Sequences . . . . .	130
9.3	Detection of Implied Scenarios . . . . .	137
9.4	Related Work . . . . .	140
9.5	Discussion And Conclusions . . . . .	141
	References . . . . .	141
<b>10</b>	<b>Paper 5</b>	<b>143</b>
10.1	Introduction . . . . .	145
10.2	Using Collaborations to Model Services . . . . .	148
10.2.1	A case study: TeleConsultation . . . . .	148
10.2.2	Collaboration structure . . . . .	149
10.2.3	Collaboration behavior: Choreography . . . . .	151
10.2.4	The nature of collaborations . . . . .	151
10.2.5	Notation . . . . .	154
10.3	Ordering operators for choreography . . . . .	156
10.3.1	Realization problems . . . . .	156
10.3.2	Sequence . . . . .	158
10.3.3	Alternatives . . . . .	163
10.3.4	Merge . . . . .	169
10.3.5	Loop . . . . .	170
10.3.6	Concurrency . . . . .	170
10.3.7	Interruption . . . . .	171
10.3.8	Activity invocation . . . . .	172
10.3.9	Related work on realizability . . . . .	172
10.3.10	System composition . . . . .	173
10.4	Going from collaborations to component designs . . . . .	174
10.4.1	Protocol derivation from service specification . . . . .	174
10.4.2	Protocol derivation for Petri-nets . . . . .	175
10.4.3	Semi-automatic designs of collaborations . . . . .	176
10.5	Conclusions . . . . .	176
	References . . . . .	176
10.A	Remarks to the Paper . . . . .	182
<b>11</b>	<b>Paper 6</b>	<b>185</b>

11.1	Introduction . . . . .	187
11.1.1	Outline . . . . .	190
11.2	Service Specification Approach: An Example . . . . .	191
11.3	Syntax and Semantics of Choreographies . . . . .	193
11.3.1	Syntax and Semantics for Sequence Diagrams . . . . .	193
11.3.2	Syntax and Semantics for Choreography Graphs . . . . .	197
11.4	Realizability of Choreographies . . . . .	205
11.4.1	Sequential Composition . . . . .	205
11.4.2	Alternative Composition . . . . .	211
11.4.3	Interruption . . . . .	215
11.4.4	Parallel Composition . . . . .	216
11.4.5	Conflicts between Concurrent Collaboration Instances . . . . .	216
11.5	Algorithms . . . . .	217
11.5.1	Detection of Race Conditions . . . . .	218
11.5.2	Detection of Ambiguous and Race Propagation . . . . .	245
11.6	Conclusions . . . . .	263
	References . . . . .	263
11.A	Propositions and Proofs . . . . .	267
11.A.1	Race Conditions in Send-Causal Sequence Diagrams . . . . .	268
11.B	Automata Theory . . . . .	269
11.B.1	Converting an FSA into a regular expression . . . . .	270
11.B.2	Eliminating $\epsilon$ -transitions . . . . .	271
<b>12</b>	<b>Paper 7</b>	<b>273</b>
12.1	Introduction . . . . .	275
12.2	Agent and Role Based Service Architecture . . . . .	276
12.2.1	Agents as System Components . . . . .	277
12.2.2	UML Collaborations as Services and Roles as Service Components . . . . .	277
12.2.3	Dynamic Role Binding . . . . .	279
12.3	Governing Service Execution with Policies . . . . .	280
12.3.1	Role-binding and Collaboration Policies . . . . .	281
12.3.2	Feature Selection Policies . . . . .	282
12.4	Conclusion . . . . .	284
	References . . . . .	285
<b>III</b>	<b>Appendices</b>	<b>289</b>
<b>A</b>	<b>Synthesis of Role State Machines</b>	<b>291</b>
A.1	Synthesis from UML 2 Sequence Diagrams . . . . .	293
A.2	Synthesis from Choreography Graphs . . . . .	299

---

## List of Figures

1.1	Services related to system components . . . . .	5
1.2	Service-oriented development: how to model services and derive system components? . . . . .	6
1.3	Collaboration-oriented development . . . . .	8
2.1	Service Engineering Approach . . . . .	14
2.2	UML collaborations for (a) the <i>Invite</i> service feature and (b) the <i>Basic-Call</i> service . . . . .	17
2.3	UML Collaboration for the TeleConsultation service . . . . .	21
2.4	Choreography of TeleConsultation . . . . .	23
2.5	(a) Non-local choice; (b) Implied scenario . . . . .	26
2.6	System diagram for the telecommunication system . . . . .	28
2.7	(a) Generic binding of roles to a system component type. (b) Illustration of generic role instantiation at runtime . . . . .	28
2.8	System diagram for the teleconsultation system . . . . .	31
6.1	UML 2.0 Collaboration for UserLogon Service . . . . .	65
6.2	UCMs for the UserLogon Service . . . . .	67
6.3	Sequential Dependencies . . . . .	69
6.4	Partial Goal Dependency (I) . . . . .	71
6.5	Partial Goal Dependency (II) . . . . .	72
6.6	Projection of Interactions into Collaboration Roles . . . . .	75
6.7	Synthesized State-machine for TerminalClientSession . . . . .	77
7.1	Service engineering overview . . . . .	89
7.2	The <b>UserCall</b> service specified as a collaboration with a goal expression . . . . .	90
7.3	State machine diagram for collaboration <b>UserCall</b> . . . . .	92
7.4	State machine diagram for <b>UserCall</b> with role states and service goal expression (UML enhancement illustrating role states in collaboration states) . . . . .	92
7.5	Sequence diagram for collaboration <b>Invite</b> . . . . .	93
7.6	Binding roles to component classes in a collaboration use . . . . .	94
7.7	<b>UserCall</b> composed of elementary features (subordinate collaboration uses) . . . . .	95
7.8	Overview of the subordinate collaboration uses of <b>UserCall</b> . . . . .	96

7.9	The collaboration <code>UserCallWithTransfer</code> . . . . .	97
7.10	Goal sequence for <code>UserCallWithTransfer</code> with related component structure . . . . .	98
7.11	Service composed of elementary services . . . . .	99
8.1	Transport service specified as a UML 2.0 collaboration (Step 3 of the specification approach) . . . . .	112
8.2	Goal sequence for <i>TransportService</i> . . . . .	113
8.3	Interaction diagrams for the <code>TransportService</code> 's sub-collaborations . . . . .	114
8.4	Sub-role sequences for (a) <i>Vehicle</i> and (b) <i>Terminal</i> ; (c) Cross-product of <i>Terminal</i> 's sequences . . . . .	116
8.5	(a) Revised version of the <i>TransportService</i> collaboration diagram; (b) System diagram for the transportation system . . . . .	122
9.1	(a) Transport service as a UML 2.0 collaboration; (b) Sequence diagrams for <code>BuyTicket</code> and <code>VehArrival</code> sub-collaborations; (c) Service-goal tree for <code>BuyTicket</code> . . . . .	126
9.2	Goal sequence for the <i>TransportService</i> collaboration . . . . .	128
9.3	Mapping of goal sequence elements to HCPN elements . . . . .	132
9.4	Nets for the <i>TransportService</i> case study . . . . .	134
10.1	Collaboration oriented development . . . . .	147
10.2	Activity diagrams describing a <code>TeleConsultation</code> . . . . .	149
10.3	Roles and sub-collaborations in the hospital visit . . . . .	150
10.4	Choreography for the <code>TeleConsultation</code> collaboration. . . . .	152
10.5	Alternative notations for role binding . . . . .	155
10.6	Problematic weak sequential compositions . . . . .	159
10.7	A sequence diagram for the voice collaboration illustrating mixed initiatives with common goals . . . . .	164
10.8	(a) Example of a non-local choice; (b) Non-local choice where a mixed initiative conflict cannot be detected; (c) Non-local choice where a mixed initiative conflict can be detected; (d) Possible scenario resulting from (a) . . . . .	165
10.9	Choices with (a) ambiguous propagation and (b) race propagation; (c) Behavior implied by (b) . . . . .	168
10.10	(a) Choice with race propagation; (b) Unfolding of (a); (c) Behavior implied by (a) . . . . .	169
10.11	Actors with role bindings . . . . .	174
11.1	Collaboration oriented development . . . . .	189
11.2	UML collaboration for the <i>ShuttleService</i> . . . . .	191
11.3	Sequence diagrams describing some elementary collaborations of <i>ShuttleService</i> . . . . .	192
11.4	Choreography of <i>ShuttleService</i> . . . . .	192
11.5	Some basic sequence diagrams (with conflicts) . . . . .	194
11.6	Example of sequential and parallel composition of activities in a choreography graph. . . . .	201

11.7	Examples of interrupting composition of activities in a choreography graph.	202
11.8	Examples of invocation composition of activities in a choreography graph.	204
11.9	Weak Sequence Problems . . . . .	207
11.10	Competing-initiatives choice . . . . .	212
11.11	(a) Non-deterministic and (b) Race choice propagation; (c) Behavior implied by (b) . . . . .	214
11.12	Replacement of collaborations with behavior described by several posets	218
11.13	Races with send-causal and weakly-causal compositions . . . . .	221
11.14	Examples of combinations of merge and decision nodes in a choreography graph . . . . .	223
11.15	Properly nested fork and join nodes in a choreography graph . . . . .	223
11.16	Two examples of loops in a choreography graph . . . . .	236
11.17	(a) Choice (ch1) with race propagation; (b) Automaton describing the significant part of $R2$ 's behavior in (a) for detection of race propagation	249
11.18	Behavior of role $p$ in $v_0$ . . . . .	253
11.19	(a) FSA for the choreography of Fig. 11.17(a); (b) FSA resulting after eliminating all $\epsilon$ -transitions in (a); (c) FSA resulting after eliminating all non-initial and non-final states in (b) . . . . .	271
12.1	Service Oriented Architecture . . . . .	277
12.2	The UserCall Service as a UML 2.0 Collaboration . . . . .	278
12.3	Role Request Pattern . . . . .	280
12.4	Role Binding . . . . .	283
A.1	Mapping of an activity into an HFMSM . . . . .	294
A.2	Composition of state machines . . . . .	296
A.3	Illustration of mapping of fork-join pairs into an HFMSM . . . . .	300
A.4	(a) Mutual invocation; (b) Simple invocation; (c) State machines for each of the activities in (b); (d) State machine resulting from the invocation composition in (b) . . . . .	301
A.5	(a) Interruption composition; (b) Partial mapping of (a) into an HFMSM; (c) Complete mapping of (a) into an HFMSM . . . . .	302



**Part I**  
**Overview**



---

# Introduction

The development of *distributed reactive systems* is a complex and error-prone process. Reactive systems are characterized by being continuously running and responding to stimuli from their environment [HP85]. When they are distributed they consist of many logically and/or physically separated *autonomous* components that operate *concurrently* and interact with each other and the environment in order to deliver *services* to the end-users. Typical examples include telecommunication systems, as well as automotive, avionics, process control or home automation systems. These systems are becoming more and more sophisticated and omnipresent in everyday life, providing an ever expanding portfolio of services being increasingly interconnected with each other. This is specially true for systems serving human beings, such as telecommunication systems, where new and customizable services are considered the key to increased revenues. Being able to rapidly develop and incrementally deploy such services, while avoiding undesired interactions with already existing services, therefore becomes strategically important. As a contribution to achieving this we offer a modeling approach where individual services are specified separately and then composed into full system functionality. The approach enables the behavior models to be analyzed at different stages to ensure the system behavior is the intended one. We have also proposed policy-based mechanisms to allow personalization of services and runtime adaptation.

In this chapter we first explain the concept of service, which is central to our work. We then discuss our motivation for this work and identify a series of research questions. A summary of the main contributions is thereafter presented, followed by the current status of tool support and a general outline of the thesis.

## 1.1 The Notion of Service

The service concept has been widely used in the telecommunications domain since the early 80's. Nowadays, service-orientation is gaining popularity in the software community at large as a new paradigm to ease the development of distributed information systems and business applications. Unfortunately, despite its wide use, there is not a well-established and common notion of the term "service".

The following example may help to understand the notion of service used in this thesis. Consider a 3G mobile phone terminal that can be used to access services such as videoconference calls, multiplayer games or remote banking. An obvious observation is that this terminal behaves differently in each of these services. For

example, it adopts the role of gamer in the multiplayer game, while it behaves as bank-client in the remote banking service. Moreover, the terminal may not always behave the same within a given service: in a videoconference call, for example, it will play the role of caller, when its end-user initiates the call, or the role of callee, if its end-user is invited to join a call. A second observation is that none of these services are possible without the joint effort of several system components<sup>1</sup>, which collaborate in order to achieve the service's goal. For example, a videoconference call involves two or more mobile phones, as well as a number of specialized network nodes required to establish and terminate the call, and to transmit the audio and video signals between the terminals. All these system components collaborate in order to allow the end-users to see and talk to each other, which is the videoconference's goal.

From the above example one can see that, at a high level of abstraction, a service can be understood as an *identified functionality* provided by a system to its end-users, with the intention of delivering some *value* to them. The complete functionality or behavior of a system arises from the composition of all the services provided by that system. A service is therefore a *partial system functionality*. This view of service agrees with the common understanding of service in everyday life, that is, a non-material good that is offered by a provider to a consumer, and that generates some value to the latter. This notion of service has long been used in the telecommunications domain to structure the functionality of telecommunication systems and networks. A service is seen as a set of features (i.e. increments of functionality [BDC<sup>+</sup>89, Zav01]) that end-users can access and be charged for. A similar notion of service can also be found in the area of communication protocols, where the service concept is used as an abstraction mechanism to better manage the complexity of protocol design. Here, a service specification describes the observable end-to-end behavior provided by a system to its end-users [BS80, VL85]. Within the software engineering community the understanding of the service concept is slightly different. In middleware technologies, such as Jini or CORBA, a service is seen as a function or operation that can be invoked on a software component. In the service-oriented computing (SOC) and service-oriented architecture (SOA) domains a service is normally understood as a software application that can be accessed through a well-defined, public interface. These notions of service are rather implementation-oriented, and consider services as small pieces of functionality provided by software components and applications.

The notion of service discussed above considers a system as a black-box that provides services to its end-users, but nothing is said about how those services are actually provided. The 3G-phone example reveals, however, that service provisioning involves several system components, which *collaborate* in order to achieve the purpose or *goal* of the service. Even in the case of a client-server type of service, the service is not possible without the collaboration of a client and a server. This has been recognized by several authors (e.g. [Bræ99, BKM07]) and is sometimes

---

<sup>1</sup>In this thesis the term “system component” is used in an abstract sense to mean an identified active object, process, agent or logical entity that has an autonomous behavior and is able to interact with other system components by means of message passing.

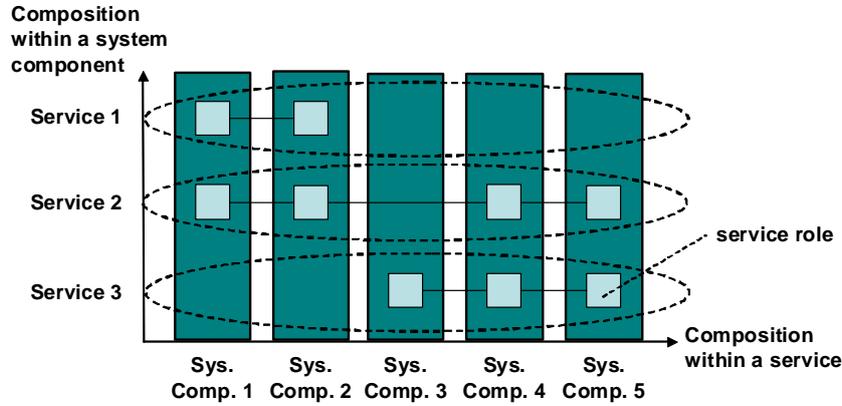


Figure 1.1: Services related to system components

referred to as the “crosscutting” nature of services [FK01, KM03]. We may therefore define a service as a partial system functionality with value for the system’s end-users and that arises from the collaboration among system components and environment entities. The problem with this definition is that it ties services to system components, so that specifying the behavior of services requires specifying the behavior of system components. This is unfortunate since there is not a one-to-one relationship between services and system components. As already revealed by our 3G example, system components may participate in the provision of several different services, either simultaneously or alternately. This means that, in the most general case, the behavior of services is composed from partial component behaviors, while the behavior of components is composed from partial service behaviors, as illustrated in Fig. 1.1. The ability to specify services independently of particular system designs or implementations is highly desirable. Separation between services and system components can be achieved with the notion of *service role*, understood as the part that a system component plays in a service.

Based on the above discussion we consider the following definition of service:

**Definition 1.1.** A service is an identified functionality with value for the service users, which is provided in a collaboration among service roles played by system components and/or service users.

**Definition 1.2.** A service role is the part a system component or service users plays in a service [Bræ99].

We note that this definition of service resembles the definition used in [San07]. In that work a service is defined as a collaboration between service roles that provides functionality to the end-users or environment. The main emphasis is therefore on the collaborating nature of services. In the definition presented here the emphasis is on functionality, considering collaboration among service roles as a necessary vehicle to provide such functionality.

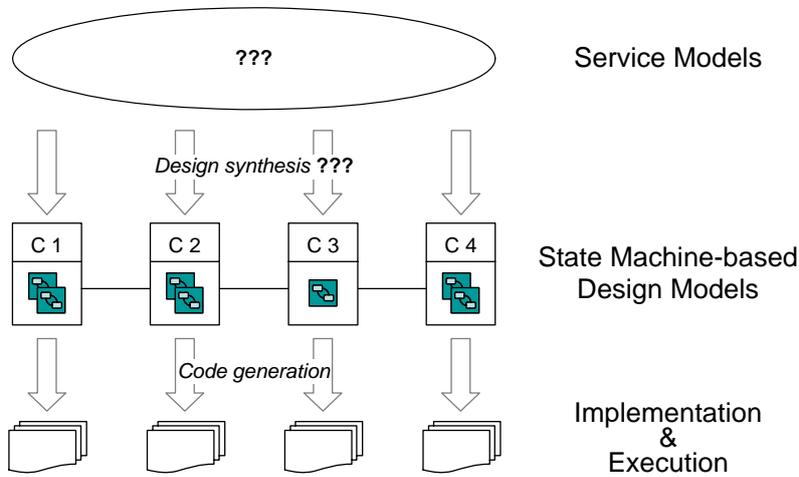


Figure 1.2: Service-oriented development: how to model services and derive system components?

## 1.2 Service Models and Design Models

For several decades now it has been a common practice to design distributed reactive systems in terms of loosely coupled components modeled as communicating state machines (see e.g. [Boc78, Bræ79]). The advantages of this approach are well-known: state machines allow a precise and complete description of the reactive behavior of components, they can be automatically analyzed, and can serve as input for the automatic generation of code. Modeling a reactive system in this way, from a set of end-user requirements, may however be challenging and error-prone. Requirements are normally given from a global point of view, and not from the point of view of individual system components. They describe how end-users access the system's services, and how the system components interact to provide those services. Describing the complete behavior of system components from such requirements is not straightforward. The reason is simple: as we mentioned in Section 1.1, there is not a one-to-one relationship between system components and services, thus modeling the behavior of a service does not amount to modeling the behavior of a component. Having a clear understanding of the collaborative behavior of components is essential to ensure the correctness of their behaviors and, thereby, of the services they help to provide. In order to support flexible and compositional service engineering there is a need to model services explicitly and independently of particular component designs, and to support the composition of components from service models. Figure 1.2 shows our vision of a service-oriented modeling approach where services models are first created, describing services explicitly, and thereafter used to synthesize the behavior of the individual system components. A set of questions arises concerning how services can best be modeled, how the behavior of system components can be synthesized and how to be sure that the behavior specified by the design model is exactly the behavior specified by the service models.

Interaction diagrams (e.g. MSCs [IT99] or UML sequence diagrams [OMG07]) are useful and popular means to capture collaborative behavior in the form of message exchanges between components. They have proven to be valuable during the early stages of the development process, and have found their way into numerous methodologies. While interaction diagrams are obvious candidates for the modeling of services, there are some drawbacks with their use. Because of the large number of possible execution traces in real systems, they are normally not used to define complete behaviors, but only exemplary scenarios. In many cases scenarios partially overlap, and the relationships among them are not clearly described. Assessing the completeness and consistency of a set of scenarios can therefore be difficult. We have asked ourselves if there are better ways to model services. *Is it possible to model service behavior more completely? Can it be done in a structured way without revealing more interaction detail than necessary? Can services be modeled as reusable building blocks, such that services can be composed from more elementary ones?*

A central hypothesis for the work presented in this thesis has been that UML 2 collaborations<sup>2</sup> [OMG07] provide useful mechanisms to give positive answers to the above questions. UML 2 collaborations fit nicely with the concept of service as it was defined in Section 1.1. They define a structure of partial object behaviors, called roles, and enable a precise definition of collaborative behavior using a variety of notations, such as interaction diagrams, state machines or activity diagrams. Collaboration roles can be bound to the roles of other enclosing collaborations and to system components by means of collaboration uses (see Fig. 2.2 on page 17). In this way, collaborations can be used as reusable building blocks for the compositional specification of services. This leads to the following research question:

**RQ1:** *How can UML 2 collaborations be utilized to model service behavior in a complete and modular way?*

We have found that decomposition of collaborations tend to result in elementary collaborations associated with interfaces. The behavior of such elementary collaborations can be completely described using, for example, interaction diagrams. The full behavior of a composite collaboration can then be specified with a *choreography* graph defining the global execution ordering of its sub-collaborations. Figure 1.3 illustrates key elements of a collaboration-oriented approach to service engineering:

- Service models are used to separately specify the global behavior of services. UML collaborations are used to define services and service features<sup>3</sup>. Sequence diagrams are used to describe the behavior of elementary collaboration and choreography graphs for specifying the global behavior of composite service collaborations.

---

<sup>2</sup>Collaborations are new modeling elements in the UML family. They were introduced in version 2.0 of the UML standard [OMG05]. UML 2 collaborations should not be mistaken with UML 1.x collaboration diagrams, which are now called communication diagrams.

<sup>3</sup>Service features are normally understood as increments of functionality that are added to a base service. In this thesis we do not make such distinction between base service and features. We understand service features as elementary services (i.e. providing less functionality) and model them in the same way as services.

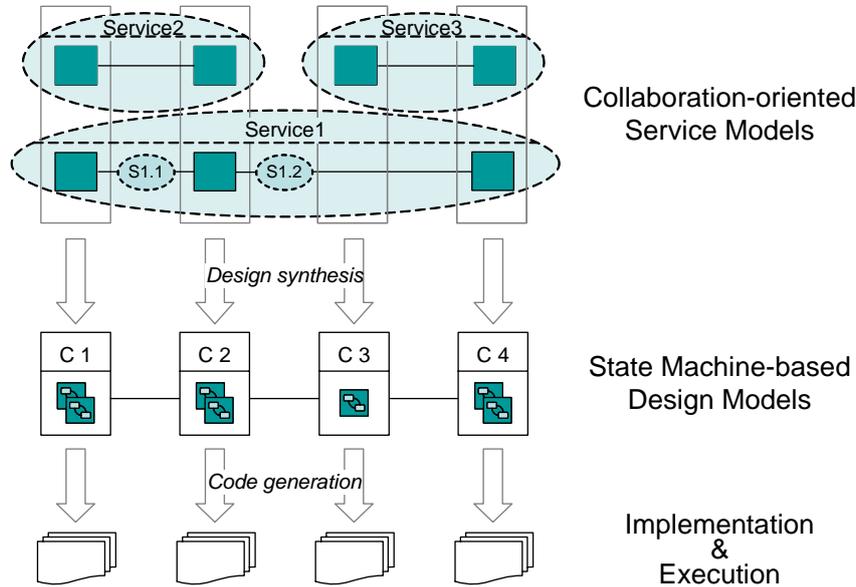


Figure 1.3: Collaboration-oriented development

- Design models describe the system structure, as well as the complete local behavior of each system component type. Asynchronously communicating state machines are used at this level.
- Implementations consist of executable code that is automatically generated from the design models.
- Execution platforms are the systems where software processes are executed to provide services. Such platforms may provide mechanisms for the dynamic selection and execution of roles based on, for example, the context situation and user preferences.

A crucial step in the approach outlined in Fig. 1.3 is the transition from a service model to a design model. In this step the collaboration roles are bound to system components. The behavior of these components is then synthesized as a composition of role behaviors. This step faces a fundamental problem. It may happen that the synthesized components behave correctly from their local points of view (i.e. as specified in the service model), but their joint behavior leads to scenarios that are not explicitly specified by the service model (i.e. *implied scenarios*). This is known as the *realizability* problem. A number of authors have studied the problems of realizability and implied scenarios (e.g. [AEY00, UKM04]). Still, we believe that a better understanding of the actual nature of these problems has been missing, as well as guidelines to avoid them in the first place, without compromising the expressive power of specification approaches. We have therefore asked ourselves: *What kind*

*of realizability problems may arise? What is the nature of these problems and what are their consequences? Is it possible to detect them by analyzing the service model? What are the possible solutions?*

Building on the hypothesis that UML 2 collaborations provide a useful basis for service modeling, we have formulated the following research question:

**RQ2:** *Can choreographies help to uncover realizability problems by studying the composition operators used to order elementary collaborations?*

The design models in Fig. 1.3 manages the associations that are possible between the system components, and the roles that these components may play. In many systems the structure of collaborating components is rather dynamic however. Links between components are created and deleted dynamically in order to provide service functionality. Only if such links can be created and if components can play appropriate roles, the service can actually be provided. This is what we call *dynamic role binding*. Consider, for example, a telephone call service. A call will only be successful if a link between two specific terminals (i.e. not whatever terminals) can be established. But not only that. The terminals must be able to execute the required roles for the call service, namely *caller* and *callee*. If the user being contacted is busy, his terminal may not be able to execute the *callee* role and the call will not succeed. Dynamic role binding encompasses increasingly important mechanisms such as service discovery and adaptation, and service personalization based on the service context and user preferences. Hardcoded mechanisms are often not flexible enough to control the process of dynamic role binding. We have asked ourselves how that flexibility can be achieved:

**RQ3:** *How can the dynamic role binding process be made as generic, flexible and scalable as possible?*

## 1.3 Contributions

This thesis has contributions in three areas: (1) system and service modeling; (2) analysis of service and design models; and (3) service execution. In the following we summarize the main contributions on each of these areas.

### System and Service Modeling

The following contributions address research question RQ1:

- Clarification of the nature of services (Section 1.1 and Paper 2).
- An approach to service-oriented and model-driven specification and design of distributed reactive systems (Chapter 2 and Papers 3 and 2). The approach models services explicitly using UML 2 collaborations, and enables the state machine-based behavior of individual system components to be automatically derived. The approach is compositional in nature and promotes separation of concerns.

- A notation to define the choreography of a composite collaboration, that is, the global execution ordering of its sub-collaborations, which effectively specifies the full behavior of the composite collaboration (Papers 3 and 6). Two formal semantics are given: one based on high-level coloured Petri-nets that represents the intended behavior (Paper 4); and one based on partial orders that represents the actual behavior (Paper 6).
- Guidelines to break-down the complexity of the modeling process and to minimize the amount of conflicts in the models (Chapter 2 and Paper 5).
- An algorithm for the synthesis of flat role state-machines from choreographies described with Use Case Maps (Paper 1), and an algorithm for the synthesis of hierarchical role state-machines from choreographies described with activity diagrams (Appendix A in Part III).

## Model Analysis

The following contributions address research question RQ2:

- Study of realizability problems for service models based on collaborations and choreographies. This includes:
  - A classification of the realizability problems in terms of choreography composition operators (Paper 5)
  - An analysis of the actual nature of the different realizability problems (Paper 5)
  - A discussion of possible solutions and their suitability (Paper 5)
  - Algorithms for the detection of some realizability problems (Paper 6)
- A technique for the detection of unexpected behaviors arising from unforeseen interactions between simultaneous occurrences<sup>4</sup> of service collaborations (Papers 3 and 4).

## Service Execution

The following contribution addresses research question RQ3:

- A study of dynamic role binding mechanisms, and a policy-based framework for managing it (Paper 7).

---

<sup>4</sup>Collaborations cannot be instantiated in UML, so we cannot talk about instances of a collaboration. In the rest of this thesis we will therefore use the term “collaboration occurrence” to denote that a set of system components are interacting as specified by a collaboration. We will also use the term “service occurrence” in a similar fashion, to denote that the service functionality is being delivered or provided to the end-users of the service.

## 1.4 Current Status of Tool Support

In the following we discuss the implementation status for the solutions presented in this thesis.

### System and Service Modeling

In principle, any UML 2 compliant tool may be used for modeling services as proposed in this thesis. Nevertheless, we have experimented with the TOPCASED CASE environment [TOP] in order to automatically generate a customized graphical editor with support for the adornments that we propose for UML 2 collaborations and activity diagrams. The results are satisfactory but still in a primitive stage.

Regarding the synthesis of state machines, there is currently not automated support.

### Model Analysis

Currently there is no tool support for the analysis techniques proposed in this thesis. We believe, however, that they could be implemented with reasonable effort and time, given the level of detail of the algorithms proposed in Paper 6, and the simple mapping to Petri Nets proposed in Paper 4.

### Service Execution

A prototype of the proposed policy-based and service-oriented architecture has been implemented as part of [Pha05].

## 1.5 Outline

This thesis is organized into three main parts. The main body of the thesis' work is presented in Part II as a collection of papers, and is complemented with some additional material in the appendix of Part III. Part I presents an overview of the work in Parts II and III and is further structured as follows:

- Chapter 1 (this chapter) presents the motivation for the service engineering approach investigated in this thesis. A series of research questions is identified and a summary of the main contributions is provided.
- Chapter 2 provides an overview of the proposed service engineering approach, placing the work presented in Parts II and III within a common framework.
- Chapter 3 provides an overview of some related work.
- Chapter 4 gives a short summary of each of the papers included in Part II.
- Chapter 5 summarizes and discusses the work, identifying some limitations and possible extensions.



---

# A Service Engineering Approach

In this chapter we present a service engineering approach for the development of reactive systems. First, in Section 2.1 we introduce the main lines of the approach and motivate the principles behind it with help of an example. In the following sections we explain in more detail each of the main activities integrating the approach.

## 2.1 The overall idea

Jackson's problem frames approach to the development of software systems [Jac05] advocates decomposing realistic problems into more elementary subproblems that fit into classes of problems for which a solution is known. Each sub-problem is then analyzed and described separately, without paying attention to their dependencies and eventual interactions. It is at the very end that the solutions to the subproblems are composed. At this stage, composition concerns may arise (i.e. interactions between the subproblems due to dependencies may be identified). Such concerns may uncover issues with requirements that were not initially visible, and may demand a revision of the subproblems. These underlying principles of the Jackson's approach are also at the heart of our approach. We believe that trying to handle all aspects of a system at once during the specification process is unrealistic, even for small systems. We therefore advocate breaking down the complexity of system specification and design by decomposing the system's functionality using services, roles and components, and treating their potential dependencies as a composition problem. We also advocate for an early analysis of the specification, so that potential flaws can be detected and resolved before the system is completely designed and implemented.

Figure 2.1 illustrates our approach to service engineering. It is an iterative approach consisting of five main activities:

1. **Service modeling:** A service model is created for each individual service to be provided by the system under development. Each service is modeled as a UML collaboration defining the structure of roles needed for the service, and its decomposition into elementary collaborations. The complete behavior of elementary collaborations is specified with sequence diagrams, while the global behavior of the service collaboration is described with a choreography graph describing the execution ordering of its sub-collaborations.

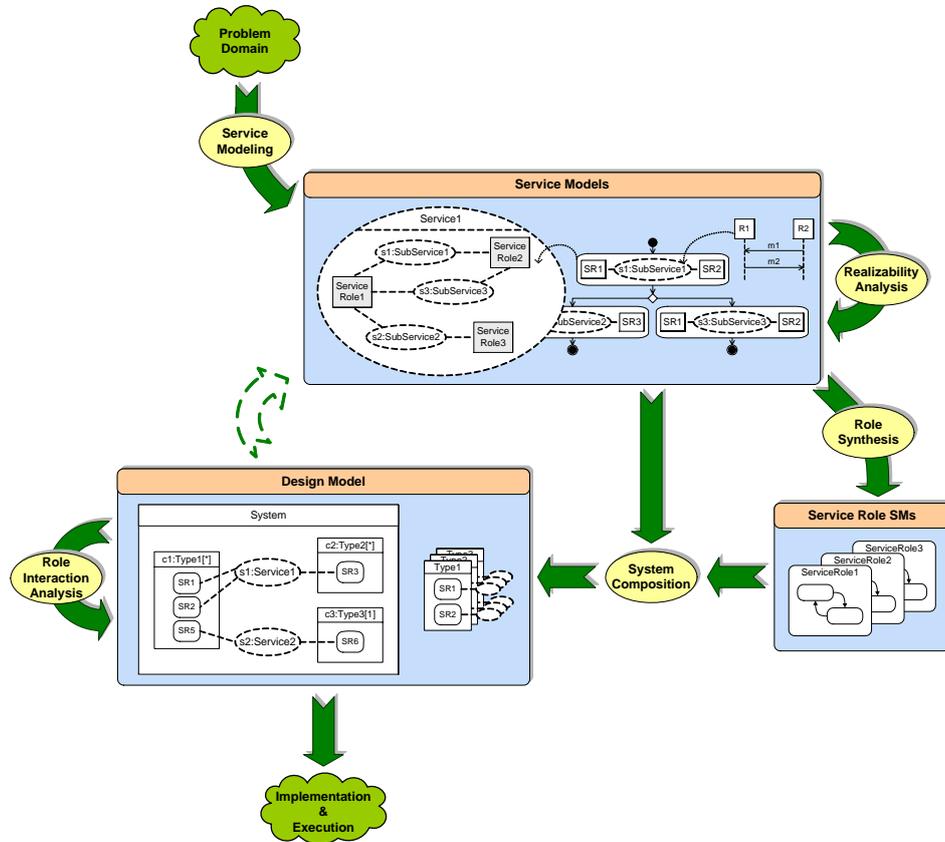


Figure 2.1: Service Engineering Approach

2. **Realizability analysis:** Each service model is analyzed in search of realizability problems. The aim is to ensure that the service model does not imply behaviors that are not explicitly specified, but that may arise in the design model.
3. **Service role synthesis:** For each service model, the local behaviors of its service roles are automatically synthesized in the form of state machines. The choreography graph and sequence diagrams are used as input for the synthesis.
4. **System composition:** The system structure is specified in terms of system components (with type and multiplicity) and their relationships. The complete behavior of each system component type is designed as a composition of the service roles that it may play. To determine the correct way of coordinating the role behaviors, an analysis of potential interactions is necessary (see next activity).

5. **Role interactions analysis:** A system component may simultaneously participate in different service collaborations, as well as in several occurrences of the same service collaboration. We analyze whether undesired interactions may arise between the roles that the system component may play in simultaneous service collaborations. The results of this analysis will dictate how role behaviors should be coordinated and composed into system component behaviors (see previous activity).

The final goal of this activity and the previous one is to design system components so that they can play appropriate roles in each service collaboration they participate in, without undesired interactions with other running roles. This can be seen as a problem of dynamic role binding, and policies may be defined to govern such role binding.

As a motivation for our approach we consider a simplified telecommunication system that offers a so-called *BasicCall* service, for peer-to-peer calls between users. This system consists of many peer components that may both initiate and accept calls, so multiple simultaneous calls are possible (i.e. multiple occurrences of the *BasicCall* service may coexist). Since this is a version of a well-known system (i.e. the PSTN and derivatives), some of the problems or “special” requirements that have to be considered during the specification and design process are already well-known. For example, it is known that a user may try to call another user who is already involved in a call (i.e. is busy). Two a-priori independent calls will then interact. Such particular interaction may seem obvious, but this is only because there already exists enough knowledge about this specific type of system. Foreseeing similar interactions in new systems might not be so easy, however. The service engineering approach proposed in this thesis is intended to help identify such interactions.

In order to make the specification and design process more manageable, we focus initially on the specification of one “isolated” occurrence of the *BasicCall* service. The possibility of having multiple simultaneous occurrences of *BasicCall*, and the potential interactions that may exist among them will be treated during system composition. The *BasicCall* service can be described as a collaboration between two service roles, namely *caller* and *callee*. The *caller* role defines the properties and behavior that a component must exhibit in order to initiate a call, talk and possibly disconnect. Similarly, the *callee* role defines the properties and behavior required from any component in order to accept an incoming call, talk and possibly disconnect.

Once the *BasicCall* collaboration has been completely specified, a realizability check may be performed to ensure that the service model does not imply behaviors that are not explicitly specified. If any realizability problem is detected, we may decide to modify the service model in order to solve it or we may opt for resolving the problem during the system composition process. Once the realizability analysis is finished, and eventual modifications to the service model are made, the design of the system components is undertaken. First, the behavior of the *caller* and *callee* roles is synthesized using the information provided by the *BasicCall*'s service model. Thereafter the roles need to be assigned to the system components that will execute them. For the sake of simplicity, we assume there is only one type of component

in the telecommunication system, namely *UserAgent*. Any instance of *UserAgent* should be able to initiate, as well as to accept calls, so both roles of *BasicCall* should be bound to the *UserAgent* type. The behavior of any *UserAgent* instance would therefore be a composition of the behaviors of the *caller* and *callee* roles. At this point we need to consider whether multiple occurrences of *BasicCall* may be running simultaneously, and whether they may interfere with each other. If so, they must be coordinated to avoid undesired behaviors. In the telecommunication system, simultaneous occurrences of *BasicCall* may interact in several ways. For example, given that there will be multiple instances of *UserAgent* in the system, a *UserAgent* that is already playing a role in a call may be requested to play a *callee* role in another call. Both roles will make use of a limited resource: the end-user; so their executions will certainly interfere with each other. Dealing with such interactions may require adopting appropriate design decisions or even modifying the specification of *BasicCall*. We consider this a natural step that helps us to better understand and document the intricacies of the system under development. In [Jac05], talking about the problem frames approach, Jackson mentions that “*if the deferred composition concerns then prove to demand substantial reworking of the subproblems to be composed, this is not a disadvantage of the separation: it is rather an indication that dealing simultaneously with the subproblems and their composition concerns would have been very difficult*”. We totally agree with this reasoning.

In the following sections we take a closer look at each of the activities of the proposed service engineering approach. Since the system composition and role interactions analysis are tightly integrated, we discuss them in the same section.

## 2.2 Service modeling

Assuming that a set of services have been identified from the requirements, the focus in this activity is on modeling each of those services using a combination of UML collaborations, activity diagrams and sequence diagrams. In the system to be, many occurrences of each service may, in general, happen either sequentially or simultaneously, and may sometimes need to be coordinated in order to prevent undesired interactions between them. We initially abstract away from these issues and just focus on modeling one single, “isolated” occurrence of each service.

Inspired by the work of Sanders [San07], we have chosen UML 2 collaborations [OMG07] as the modeling element to represent and structurally decompose services. UML collaborations are *structured classifiers*. As such they define a structure of roles that collaborate to accomplish a task, or achieve a goal. This includes the communication paths, specified by means of *connectors*, that should exist between any pair of objects playing the roles in order for the collaboration to happen. Figure 2.2(a) shows the diagram for an *Invite* collaboration. This diagram specifies that two roles, *Inviter* and *Invitee*, are needed to establish an *Invite* collaboration, and that a communication path must exist between the system components playing these roles. UML collaborations are also *behaviored classifiers*, and may therefore have associated behavior. An interesting feature of collaborations is that they can be defined in terms of other smaller collaborations, thus allowing a compositional specification

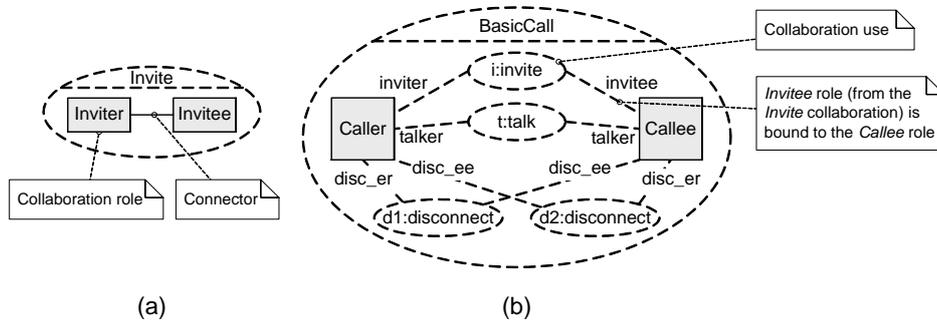


Figure 2.2: UML collaborations for (a) the *Invite* service feature and (b) the *BasicCall* service

of services. This is done by binding the roles of a (sub-)collaboration to the roles of a containing collaboration by means of *collaboration uses*. Figure 2.2(b) shows a UML 2 collaboration representing the *BasicCall* service. This collaboration consists of two roles, *Caller* and *Callee*, which may engage in a series of sub-collaborations. These sub-collaborations are represented as collaboration-uses, and their roles are bound to the *Caller* and *Callee* roles. For example, the *Invitee* role of the *Invite* collaboration has been bound to the *Callee* role. This means that any object playing the *Callee* role in *BasicCall* should have the properties and behavior specified for the *Invitee* role. For a more detailed discussion about the suitability of UML collaborations for service modeling we refer the reader to Paper 2 and to [San07].

The specification of a service collaboration can be divided into 5 steps:

1. Identification of the roles needed to provide the service.
2. Identification of the sub-collaborations in which the service roles may engage.
3. Structural composition of the sub-collaborations identified in the previous step.
4. Description of the global service behavior by specifying the order in which the sub-collaborations should be executed.
5. Description of the behavior of each sub-collaboration.

We detail each of these steps in the following.

**Identification of service roles.** As a first step we have to identify the service roles needed to provide the service. This follows from the problem domain and the logical architecture of the service execution environment. Each role specifies the properties and behavior that a component should have in order to participate in one single occurrence of the service under specification. Focusing on roles rather than on components has some advantages:

1. it allows to describe the service behavior independently of particular system components or final implementations, so that role behaviors can be reused as part of many component behaviors;
2. it also allows to consider synthesis of component behavior as a problem of role binding and composition, where assumptions about coordination of roles are made explicit.

As we already discussed, for the *BasicCall* service we can identify two roles, namely *caller* and *callee*. We note that in some cases there will be a one-to-one relation between service roles and system components, but this might not always be the case (e.g. in our telecommunication system, a system component may play both the *caller* and *callee* roles of the *BasicCall* service). It is therefore important to abstract away from particular components in the system and rather focus on the properties and behavior that they should have to participate in the service. Note however that taking a minimum of logical (but not physical) architectural aspects into account might still be necessary and beneficial for the identification of roles [Zav03].

**Identification of sub-collaborations.** While one may try to describe the complete behavior of a service collaboration with sequence diagrams, this may rapidly become a difficult task. The underlying reason is that the number of possible event orderings is normally very high, even for moderately complex services. One can address this problem by decomposing a service collaboration into smaller sub-collaborations. A useful decomposition results in more manageable collaborations whose behavior can be completely described with sequence diagrams. A side effect will be a larger set of smaller collaborations and a reduced cross-cutting overview, which may be considered as a drawback. We believe, however, that the benefits of such decomposition outweigh its disadvantages. Decomposition helps to focus separately on, and better understand, the different aspects of the service. Moreover, the need for later (re-)composition of the sub-collaborations' behaviors brings an opportunity to improve the service model, since the relationships and dependencies between the different sub-collaborations must be explicitly specified. This helps to increase the understanding of the global service behavior, and can be exploited in the analysis of the service model. Finally, smaller self-contained collaborations tend to be more reusable.

A question arises concerning how a *useful* decomposition can be achieved. Although there is not (yet) a golden rule that can be obeyed, we have identified some useful decomposition criteria. In general, it helps to think in terms of interfaces and goals. One can start by identifying the interactions that service roles maintain with each other (i.e. the interactions that happen on the interfaces between any pair of roles). Rather than thinking about detailed interactions, one should think about the purpose or goal of those interactions. It also helps to think of the global states and/or phases that the service goes through, since these phases are normally the result of executing one or more sub-collaborations (see Paper 2). For example, the *BasicCall* service will go through several phases, such as *inviting*, *talking* and *disconnecting*, which could be described as separate sub-collaborations associated

with the interface between *caller* and *callee* (see Fig. 2.2(b)). In general, each sub-collaboration should have at least one main goal (concrete and meaningful), and may have zero or more alternative secondary goals. Sub-collaborations with more than one main goal could normally be further decomposed. For example, the main goal of the *Invite* collaboration in Fig. 2.2(a) would be to establish a connection between the components playing the *inviter* and *invitee* roles. When *Invite* is used as part of *BasicCall*, such goal becomes a sub-goal of *BasicCall*. In that context *Invite*'s main goal can be seen as connecting the components playing the *caller* and *callee* roles, so the end-users can talk. However, achieving this goal might not always be possible (e.g. because the *callee* is busy). In that case an alternative secondary goal would be to notify the *caller* and take some alternative actions.

To drive the decomposition process it also helps to identify the independent autonomous initiatives that roles may take to trigger services or service features, and describe the service collaboration they trigger separately. Being independent, such initiatives may happen simultaneously leading to the interleaving of the sequences of events that they trigger. By describing the behavior triggered by each initiative in a separate collaboration, the interleaving of events that need to be explicitly described in each collaboration gets reduced. This leads, in many cases, to purely sequential collaborations. The interleaving of events becomes clear when the different obtained collaborations are composed together using a choreography graph.

Decomposing in terms of goals, interfaces and initiatives naturally leads to two-party collaborations with one main goal, and whose roles define interface behavior. In general, these collaboration will only have one initiating role (i.e. a role able to initiate the collaboration), which is highly desirable. We note, however, that in some cases one may prefer collaborations with more than two roles and/or with more than one initiating role. Whenever two or more two-party collaborations pursue the same goal and require tight coordination of their behaviors, merging them into one single collaboration may be advisable. In cases where more than one role may independently initiate interactions having the same goal, we should consider creating one single collaboration where the initiatives taken by the roles to start the collaboration are properly and explicitly coordinated.

**Definition of collaboration structure.** We model the service structurally as a UML collaboration, describing the structure of roles taking part in the sub-collaborations identified in the previous step. Each sub-collaboration is represented as a collaboration use and its roles are bound to the roles of the main service collaboration. The result is a collaboration showing the sub-collaborations in which each service role is involved, and the sub-role(s) that it plays in them. See Fig. 2.2 for the *BasicCall* collaboration.

**Collaboration choreography construction.** At this point we know the sub-collaborations that service roles must participate in order to provide the service. We know the purpose (i.e. goal) of those sub-collaborations, and we may even know their detailed behavior if, for example, we were reusing any predefined collaboration. We do not know yet, however, the order in which these sub-collaborations should

be executed, so that their global, joint behavior matches the intended behavior for the service collaboration. In the SOA paradigm the description of the distributed sequencing and timing of services is called a *choreography* [Erl05]. We adopt this term and refer to the execution ordering (i.e. behavioral composition) of the sub-collaborations of a composite collaboration as a choreography.

In Paper 1 we used Use Case Maps [BC96, Buh98] to specify choreographies, while in Papers 3 and 4 we used so-called goal sequence diagrams, which are inspired by UML activity diagrams. In Papers 6 and 5, as well as in the examples that follow in the rest of this text, actual UML activity diagrams have been used. We note, however, that our use of activity diagrams is not completely compliant with the UML standard, as explained in Appendix 10.A on page 182.

The specific notation used to specify choreographies is a secondary concern, as long as it is expressive enough to describe the most common orderings or compositions that we can find among collaborations. We have identified five types of compositions that allow to describe the most usual orderings between any two collaborations,  $C_1$  and  $C_2$ :

- **Sequential composition:**  $C_2$  is executed after  $C_1$  (i.e.  $C_2$  is causally dependent on  $C_1$ ).
- **Parallel composition:**  $C_1$  and  $C_2$  are executed concurrently (i.e.  $C_1$  and  $C_2$  are independent).
- **Alternative composition:** Either  $C_1$  or  $C_2$  is executed (i.e.  $C_1$  and  $C_2$  are mutually-exclusive).
- **Invocation composition:**  $C_1$ , after reaching a point in its execution where a predicate *pred* holds, invokes  $C_2$ .  $C_1$  then is suspended and waits for  $C_2$  to execute all or part of its behavior before resuming. This normally represent a goal dependency between  $C_1$  and  $C_2$ , such that  $C_1$  can only achieve its own goal if  $C_2$  achieves its corresponding one.
- **Interruption composition:**  $C_2$  interrupts  $C_1$ .

A choreography graph provides an overview of the phases that a service goes through. Each activity in the graph represents a service phase where a certain sub-collaboration takes place. Building the choreography graph provides an opportunity to explicitly specify the ordering of sub-collaborations, which in turn enables the detection of unforeseen interactions between them.

With the choreography we are describing how the initiatives that service roles take to start sub-collaborations are to be coordinated at runtime. We may associate a *triggering event* with each collaboration. A collaboration would then be initiated if it is *enabled* (i.e. the necessary pre-condition for it to be executed is true) and its triggering event is observed. The triggering event of a collaboration may be the termination of another collaboration. It may also be an external system event (e.g. some user input) or a time-out that is not part of the collaboration being modeled. In these cases the role initiating the collaboration may seem to take spontaneously,

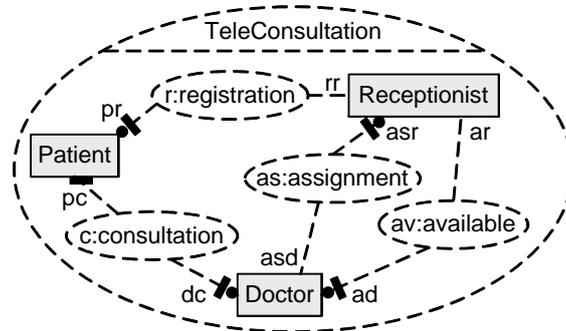


Figure 2.3: UML Collaboration for the TeleConsultation service

and on its own, an initiative to start the collaboration. It is crucial to coordinate these autonomous initiatives properly. Otherwise, unexpected behaviors may arise (see Section 2.3).

To build the choreography graph of a service collaboration it is important to identify service roles having the ability to take autonomous, independent initiatives leading to the execution of one or more sub-collaborations. In the case of purely sequential services, where only one service role can take the initiative to interact with the other roles, the choreography graph becomes simpler. It essentially describes one single thread of control where the different sub-collaborations follow each other sequentially.

In the more general case services may have several service roles able to take independent autonomous initiatives. The associated choreography will then have several concurrent threads of control that need to be properly coordinated. In these cases it is often difficult to consider all initiatives at once. Instead, one may first consider each initiative separately, and build a partial choreography graph with the sub-collaborations that are directly or indirectly triggered by that initiative. Once this is done for all initiatives, the resulting partial choreographies can be composed. An example of a system providing a *TeleConsultation* service will help to understand this idea. The system consists of multiple patients and doctors, as well as one receptionist. Doctors notify their availability to the receptionist at the beginning of their shift, as well as after each consultation with a patient (*available* sub-collaboration). The patients can call the receptionist in order to get remote assistance from one of the doctors (*registration* sub-collaboration). If a doctor is available when a patient calls, the receptionist transfers the call to that doctor (*assignment* sub-collaboration). If no doctor is available, the patient will have to call again later. An occurrence of the *TeleConsultation* service will involve three roles, namely *Patient*, *Receptionist* and *Doctor* (see Fig. 2.3<sup>1</sup>). The *Receptionist* role is rather passive, and does not take any autonomous initiative to interact with the other roles. Note that although *Receptionist* starts the *assignment* sub-collaboration

<sup>1</sup>The solid circles and bars beside the roles are respectively used to identify the role that initiate and terminate each collaboration. They are not standard UML.

to transfer the patient call to a doctor, this is not the result of an autonomous initiative. This behavior is triggered by the termination of *registration*. The *Patient* and *Doctor* roles may, on the other hand, behave pro-actively. The *Patient* role may take an autonomous initiative to interact with the *Receptionist* via the *registration* sub-collaboration. After this, the *assignment* and *consultation* sub-collaborations will be executed, if a doctor is available. This behavior can be described with the choreography in Fig. 2.4(a)<sup>2</sup>. This choreography describes a partial view of the *TeleConsultation* service. It only describes the service behavior that is triggered by the *Patient* role. The *Doctor* role may also take an autonomous initiative to notify its availability to the *Receptionist* via the *available* sub-collaboration. Thereafter it may participate in the *assignment* and *consultation* sub-collaborations. This is described in the choreography of Fig. 2.4(b), which specifies the service behavior that is triggered by the *Doctor* role. This choreography also gives a partial view of the *TeleConsultation* service, which is complementary to the view offered by the choreography in Fig. 2.4(a). To obtain the complete view of the service we can merge these two partial choreographies. The result is the global choreography in Fig. 2.4(c). To obtain this choreography we note that some activities in the partial choreographies may execute concurrently (i.e. in any order). This is the case of *registration* (see Fig. 2.4(a)), which may be executed concurrently with *available* (see Fig. 2.4(b)). These activities should be composed in parallel in the global choreography. On the other hand, there are activities in both partial choreographies that refer to the same occurrence of a sub-collaboration, namely *assignment* and *consultation*. In the global choreography graph, the concurrent flows of control that the partial choreographies describe should be merged for the execution of these two activities.

**Specification of elementary collaboration behavior.** In the previous step the behavior of the composite service collaboration has been specified as a choreography of its sub-collaborations. To complete the service specification one needs to describe the behavior of each of those sub-collaborations. Some sub-collaborations may be composed of other smaller sub-collaborations. In that case, their behavior would also be given by a choreography. To specify the complete behavior (at the interaction level) of so-called elementary sub-collaborations (i.e. sub-collaborations that are not further decomposed), we propose to use UML 2 sequence diagrams, since they have proven to be a popular and useful notation for the description of collaborative behavior. Each role in the collaboration will be represented as a lifeline in the sequence diagram. Continuations may be used to identify states in the collaboration or in the role behaviors. These could then be used to relate the sequence diagram behavior with the pins of activities in the choreography graph (see Papers 1 and 3).

---

<sup>2</sup>This is an activity diagram where each activity has been adorned with the collaboration use executed on that activity. The solid circles and bars beside the roles are respectively used to identify the role(s) that initiate and terminate each collaboration.

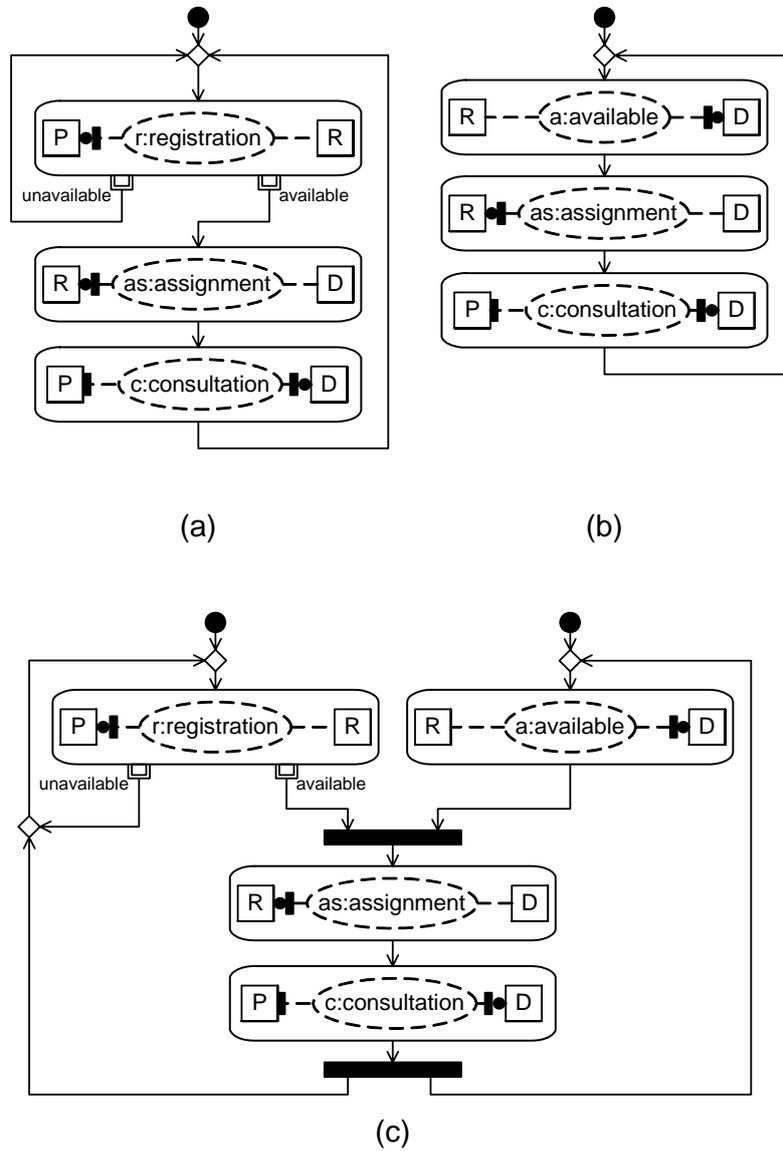


Figure 2.4: Choreography of TeleConsultation: (a) from the point of view of the Patient role; (b) from the point of view of the Doctor role; (c) global view

### 2.2.1 Behavioral interpretation of the specification

In this section we try to clarify how the behavior of collaborations and choreographies should be interpreted.

In [Krü00] Krüger distinguishes four possible interpretations of sequence diagrams with respect to what properties of the system under specification they describe: the *existential*, the *universal*, the *exact* and the *negated* interpretations. The *existential* interpretation considers that sequence diagrams specify *scenarios*. That is, they describe behavior (i.e. orderings of events) that may, but need not occur during the execution of the system (i.e. it should be possible in at least one run of the system). Other arbitrary behaviors are allowed before, during and after the specified behavior. The *universal* interpretation considers that the behavior described by a sequence diagram should occur in all executions of the system, but does not prevent other behaviors to also happen before, during or after the interactions described by the sequence diagram. The *exact* interpretation considers that the system should behave exactly as described by the sequence diagram. Other behaviors than the one specified are explicitly prohibited. Finally, the *negated* interpretation considers that sequence diagrams describe unwanted behavior.

In our approach, the sequence diagrams that describe the behavior of elementary collaborations describe the exact and complete behavior (at the abstract interaction level) of those elementary collaborations. In the same way, the choreographies describing the behavior of composite collaborations describe the exact and complete behavior of those composite collaborations. Let us discuss in some more detail what this means. As mentioned, we assume that the sequence diagram associated with an elementary collaboration describes the exact and complete behavior – with all possible outcomes – of that collaboration. Recall that a collaboration describes interactions between roles (not components). This means that the roles should behave exactly as described by the sequence diagram. As we know, we can compose collaborations. When we do this, the behavior of each sub-collaboration is interpreted as partial behavior from the point of view of the composite collaboration. That is, the behavior of a collaboration is complete if considered in isolation, but is partial in the context of a containing collaboration. It may happen that when two collaborations are composed, their behaviors are interleaved. This may seem to contradict the exact interpretation of the behavior of individual collaborations. It does not, however. The roles of each individual collaboration will observe the behavior exactly as specified for them. It is at the level of a composite role that the behaviors of several sub-roles may be interleaved.

This dichotomy between partial and complete behavior is a nice feature of collaborations. We can describe them completely, but since we describe the behavior of roles, and not the behavior of components, they still can be considered as partial behavior.

## 2.3 Realizability of a Service Model

A service model specifies the overall behavior of a service described as a collaboration between service roles. At the system design level, the service roles are assigned to the system components that will execute them. The local behavior of each component can then be obtained, in the form of a state machine, by projecting the global behavior described by the service model onto the given component (i.e. onto the service roles that will execute). We say that the design model obtained in this way is **directly realized** from the service model. If the behavior of a directly realized design model (i.e. the joint behavior of the synthesized system components) is equivalent to the overall behavior defined by the service model, we say that the service model is **directly realizable**. Unfortunately, not all service models are directly realizable. Consider, for example, the BasicCall service previously discussed. In this service, once connected and talking, both service roles can take the initiative to disconnect. This behavior may be specified as a choice between two disconnection collaborations, one of them initiated by the caller role, the other initiated by the callee role (see the partial choreography graph in Fig. 2.5(a)). Now consider two *UserAgents*, *A* and *B*, that respectively play the *caller* and *callee* roles in one occurrence of *BasicCall*. Assuming that the choreography of Fig. 2.5(a) describes the intended behavior for *BasicCall*, the two *UserAgents* should execute either collaboration *disc1* or collaboration *disc2*. However, a scenario is possible where both *UserAgents* simultaneously initiate the disconnection, that is, *A* initiates collaboration *disc1* and *B* initiates collaboration *disc1* (see Fig. 2.5(b)). The problem is that neither *A* nor *B* have enough information about the global system state, that is, they do not know whether the other component has already initiated a disconnection before they decide to initiate it. Both *UserAgents* are behaving correctly from their local points of view (i.e. according to the caller and callee roles as specified by the choreography). However, from a global point of view the resulting scenario does not correspond to either collaboration *disc1* or collaboration *disc2*. Such scenario is not explicitly described by the choreography, but is *implied* [AEY00]. As pointed out in [UKM04], implied scenarios may correspond to unwanted behaviors, such as deadlocks, but they may also represent acceptable behaviors that were overlooked during the requirements elicitation process. In both cases it is important to detect their presence, so that they can be handled correctly.

The problem of realizability of MSC-based specifications has received considerable attention during the last years (see Section 3.3). Some authors have focused on studying the complexity of checking whether a specification is realizable, and have proposed restricted classes of HMSCs for which realizability can be decided. These classes of HMSCs are usually too restrictive, and do not allow to specify behaviors that are common and useful in many services (e.g. non-local choices of competing initiatives – see Paper 5). Other authors focus on the detection of implied scenarios, which are the effects of realizability problems (i.e. the behaviors that can be observed in a realized system as a consequence of a realizability problem). We have sought to find the underlying causes that lead to realizability problems. We believe that knowing the nature of those problems helps to avoid them, and to find a correct

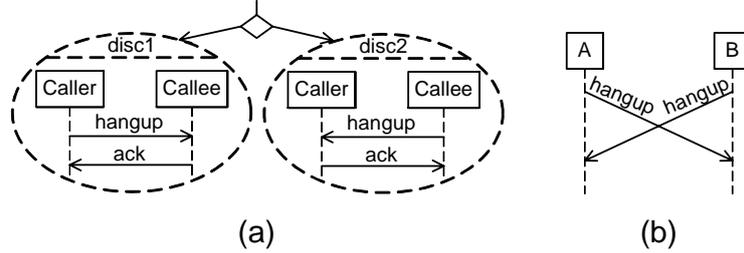


Figure 2.5: (a) Non-local choice; (b) Implied scenario

solution when they cannot be avoided. We studied this using the notion of direct realizability. This is a strict notion of realizability that requires the service roles to be synthesized without adding extra coordination messages or message contents, and requires the joint behavior of the service roles to be deadlock-free. In some cases a specification might become directly realizable by adding extra coordination messages or additional data in messages. Some authors have studied notions of realizability where this is done by default. We consider these measures as solutions to realizability problems, which could be adopted by the designer when necessary, depending on the specific service requirements.

In Paper 5 we have discussed under which circumstances a choreography is directly realizable. For each composition operator, we have studied the problems of direct realizability that may occur, how they may be detected, and what kind of additional mechanisms could be introduced into the direct realization of a choreography in order to assure that the resulting distributed behavior conforms to the choreography's specified behavior. These mechanisms include the addition of extra coordination messages in the behaviors of the service roles, as well as additional data in some messages. Provided we know which service roles initiate and terminate each sub-collaboration in a choreography, we are in many situations able to identify problems just by looking at the order that the choreography defines for those sub-collaborations without considering the detailed interactions. In other cases, we are able to identify potential problems at the choreography level, but need to consider detailed interactions of the sub-collaborations to determine whether the problems actually exist. In Paper 6 we present algorithms for the detection of some of the realizability problems.

## 2.4 Service role synthesis

After the service modeling activity is finished, we have a complete model (at the abstract interaction level) of the global behavior of one service occurrence. We are, however, ultimately interested in the complete local behaviors of the service roles, which will be assigned to system components for execution. For a given service role, its local behavior can be automatically synthesized in the form of a state machine from the information provided by the choreography graph and the sequence diagrams

describing the detailed behavior of elementary collaborations. The intuitive idea is the following:

1. First the behavior of each collaboration in the choreography graph (described by a sequence diagram) is projected on the lifeline of the specific service role (i.e. the actions of other service roles are ignored)
2. Then, the behaviors obtained in the previous step are composed following the ordering defined by the choreography for the collaborations.

Role synthesis from Use Case Maps-based choreographies was discussed in Paper 1. In the Appendix A on Part III, we discuss the generation of hierarchical state machines for roles from activity diagram-based choreographies.

If realizability problems were discovered during the analysis of the service model, additional coordination constructs (e.g. sending or receiving actions) may be added to the directly synthesized state machines in order to ensure proper coordination between the service roles.

## 2.5 System composition

During the service modeling phase, the focus was on specifying the services offered by the system under development. In the system composition phase the focus is on designing the complete behavior of each of the system components, which collaborate to provide the previously modeled services. In order to design the behavior of a system component it is necessary to know:

- the service collaborations in which the component participates, and the role(s) it plays in them;
- whether the component may simultaneously participate in multiple occurrences of a given service collaboration; and
- whether the roles played by the component may interact in unexpected ways if executed concurrently.

In the following we discuss these three issues in more detail.

**System diagram.** A so-called *system diagram* can be used to show the system structure (i.e. the system components and their relationships), the services provided by the system, and the assignment of service roles to system components. A system diagram is essentially a UML structured class with inner parts, where the structured class represents the system itself, and the internal parts represent the system components, with type and multiplicity. Connectors between the parts may be used to describe communication relationships between the system components. The services provided by the system are represented by collaboration uses defining the appropriate binding of service roles to parts (i.e. system components). Figure 2.6 shows

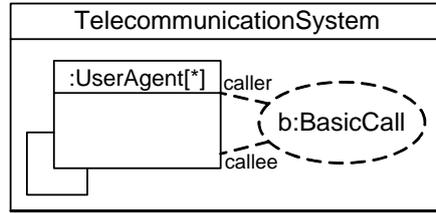


Figure 2.6: System diagram for the telecommunication system

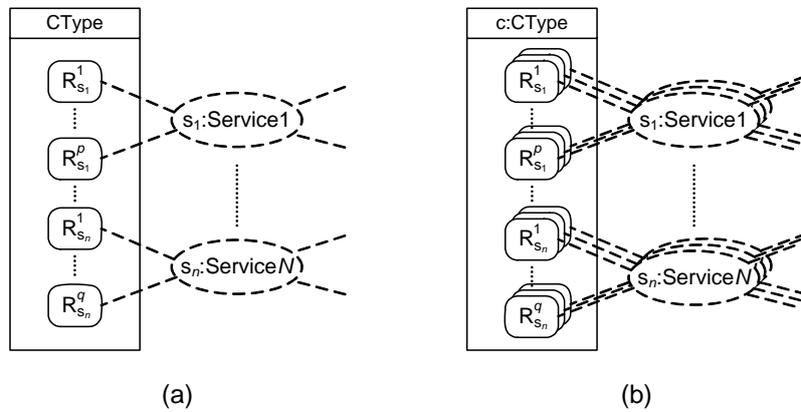


Figure 2.7: (a) Generic binding of roles to a system component type. (b) Illustration of generic role instantiation at runtime

the system diagram for the telecommunication system. It has one internal part of type *UserAgent*, which has an unbounded multiplicity. The *BasicCall* service that this system provides has been represented as a collaboration use that binds both roles of *BasicCall*, namely *caller* and *callee*, to the *UserAgent* part. This diagram expresses that the telecommunication system consists of a set of components of type *UserAgent*, which may collaborate, playing either the *caller* or the *callee* role, to provide a *BasicCall* service.

We note that, in general, a system component may participate in different service collaborations, playing one or more roles in each of them, as illustrated in Fig. 2.7(a).

**Simultaneous collaboration occurrences.** Sometimes, a system component may simultaneously participate (or at least be requested to do so) in several occurrences of a given service collaboration. For example, in the telecommunication system a *UserAgent* may be requested to join an occurrence of the *BasicCall* collaboration when it is already participating in another occurrence of that collaboration. The behavior of the component should then be designed so that the component is able to execute multiple instances of a given role type, as illustrated in Fig. 2.7(b).

This may be achieved, to a large extent, by adding extra coordination functionality to the system component, outside the role behaviors. For example, a controller state machine may be used to deal with requests for starting new roles (e.g. using a generic role request mechanism as the one discussed in Paper 7). For each granted request the controller state machine would create a dedicated instance of the appropriate role state machine. This is a generic solution to handle multiple service sessions, as explained in [BH93].

System components that may simultaneously participate (or be requested to do so) in multiple occurrences of a given service collaboration can be identified by analyzing the multiplicities of parts in the system diagram. For this, it is important to differentiate between *initiating* roles (i.e. roles performing the first actions in a collaboration) and *non-initiating* (or *offered*) roles (i.e. roles whose behavior in a collaboration starts with a message reception). The following two cases can be differentiated:

- A system component  $c_1$  plays one or more *non-initiating/offered* roles in a service collaboration, and that collaboration can be initiated by a system component  $c_2$  that has multiplicity greater than one (i.e. multiple instances of such component may exist). Different instances of  $c_2$  may simultaneously initiate several occurrences of the service collaboration and  $c_1$  may be requested to join all of them. We note that  $c_1$  cannot prevent such requests from being made and will have to handle them in one or another way. Consider, for example, our telecommunication system. According to the system diagram in Fig. 2.6, there are multiple *UserAgents* in the system with the ability to initiate, possibly simultaneously, a *BasicCall* collaboration by executing the initiating *caller* role. A given *UserAgent* may then receive multiple concurrent requests to play the non-initiating *callee* role. A question arises whether a *UserAgent* should be allowed to play several roles simultaneously. Would that make any sense? And if several roles are executed concurrently, is it possible that these roles interact in undesired ways?
- A system component plays one or more *initiating* roles whose execution is triggered by an external system event. It is then possible that a system component that is already participating in an occurrence of a collaboration tries to initiate a new collaboration occurrence in response to an external event. Of course, whether this may actually happen in practice depends on the actual system domain, that is, on how the environment may actually behave. Consider again our telecommunication system. The *caller* role is an initiating role whose execution is triggered by an external event, namely the end-user initiating a call. Obviously, the actions of an end-user cannot be controlled, so a decision should be made on how to deal with the possibility of an end-user initiating a call while its *UserAgent* is already busy participating in a *BasicCall* occurrence. If we consider that such end-user behavior is acceptable, we should design *UserAgents* so that they can simultaneously execute multiple instances of the *caller* role. Otherwise, *UserAgents* should be designed to reject (or ignore) the end-user initiative to start a new call.

**Unexpected role interactions.** In the service modeling phase, services were modeled separately, and only one isolated occurrence of each service was considered. Potential dependencies (e.g. via shared resources) between different services, and between different occurrences of a service, were not taken into account during the service modeling phase, but should be considered at this stage.

We have identified two cases in which roles simultaneously played by a system component in different service collaborations, or in different occurrences of the same service collaboration, may interact in unexpected ways:

- If two roles concurrently played in different collaborations accept messages with the same signature, a message may be consumed by a role that is not the intended receptor (i.e. the message may be consumed in a service or occurrence of a service to which the message does not belong). This kind of interactions may be easily avoided by marking messages with the id of the service occurrence/session that they belong to.
- Role interactions are also possible due to shared resources. For example, the roles being simultaneously played by a *UserAgent* in our telecommunication system would be sharing a limited resource, that is, the end-user (represented in the system by the *UserAgent* itself). Since human beings are normally not able to maintain multiple conversations simultaneously, the access to the end-user should be coordinated. One may then decide that *UserAgents* should reject all incoming requests to play a *callee* role if they are already busy playing another role. Alternatively one may decide that *UserAgents* should place such requests into some sort of waiting queue. In both cases *UserAgents* may be designed with a controller state machine in charge of handling incoming call requests, and creating (or selecting, if already running) a *callee* state machine to deal with granted requests. The specification of the *BasicCall* service may need to be modified to adapt the *caller* and *callee* roles to both the rejection and queue scenarios.

The teleconsultation system also provides examples of role interactions due to shared resources. Consider the *ReceptionistAgent* component in the system diagram of Fig. 2.8. This system component should behave according to the *Receptionist* service role in the *TeleConsultation* collaboration. In addition, the *ReceptionistAgent* should be able to deal with concurrent invitations to join several occurrences of the *TeleConsultation* collaboration. This results from the fact that there are potentially many *PatientAgents* and *DoctorAgents* that play initiating roles, so they may take independent initiatives to interact with the *ReceptionistAgent*. In order to decide how the *ReceptionistAgent* component should handle multiple initiatives coming from *PatientAgents* and *DoctorAgents* it is important to analyze their potential interactions. On one hand, the *ReceptionistAgent* is a shared resource from the point of view of the *PatientAgent* components, which will be competing for that resource. On the other hand, the ultimate goal of *PatientAgents* is to contact a *DoctorAgent*, so *DoctorAgents* are also shared resources from the point of view of *PatientAgents*, and the *ReceptionistAgent* should act as a kind of resource allocator.

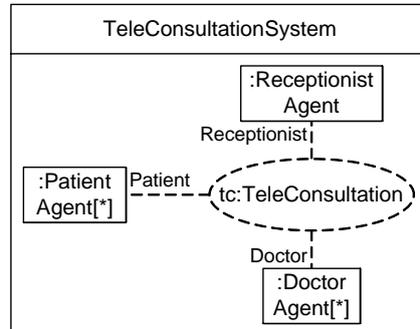


Figure 2.8: System diagram for the teleconsultation system

What happens if the *ReceptionistAgent* is already busy with a *PatientAgent* and receives a request, from another *PatientAgent*, to join a new occurrence of *TeleConsultation*? The answer depends on the actual service requirements. For example, one may decide to reject such request. This corresponds to what happens in the choreography of Fig. 2.4 when there are not available *DoctorAgents*. Alternatively, one may decide to place the request in a waiting queue. In this case we may need two queues, one for *PatientAgents* not yet attended by the *ReceptionistAgent*, and other for *PatientAgents* already attended by the *ReceptionistAgent* but waiting for a *DoctorAgent* to become available. As we have seen with the telecommunication and teleconsultation examples, role interactions due to shared resources can sometimes be easily identified by inspecting the system diagram. In other cases the interactions may not be so obvious and might be difficult to detect. In Papers 3 and 4 we presented an automatic detection technique based on the joint analysis of the sequences of (sub-)roles that a system component may play in different collaboration occurrences, and on the use of pre- and post-conditions associated to roles.

**Dynamic Role Binding.** So far we have discussed the design of system components so that they can participate, possibly simultaneously, in several service collaborations of the same or of different type. The challenge is to design the system components so that they are able to execute the appropriate role(s) in each service collaboration without undesired interactions with other roles. This can be seen as a problem of dynamic role binding<sup>3</sup>. In general, to bind a role to a system component, so that the latter executes it, some conditions should hold. From a strictly technical point of view, these conditions may be more or less intrinsic to the role (i.e. they should always hold, regardless of the context in which the role is executed), or they may depend on the current situation in terms of other active roles and be necessary to avoid undesired interactions with these other roles. The users of a system may also have their own preferences about when a system component may play a certain

<sup>3</sup>“Binding” is here taken to mean that the role is allowed to execute as part of a collaboration involving other roles it interacts with. Another term could be role *linking* or *invocation*.

role (e.g. in the telecommunication system, a user may not want its associated *User-Agent* to play a *callee* role after 10 pm). In general, user profiles, as well as service context and dependencies may be taken into account when making the decision to bind a role. In Paper 7 we discussed the use of policies to control the binding of roles to system components, in an attempt to make this process as general, scalable and flexible as possible. A policy framework was proposed with three types of policies, namely

- *Role-binding* policies, aimed at constraining the binding itself of roles to system components;
- *Collaboration* policies, to express constraints that should hold for a service collaboration as a whole when running; and
- *Feature-selection* policies, to control the triggering of context-dependent service features.

---

## Related Work

### 3.1 Work on service modeling

A number of service-oriented modeling approaches have been proposed in recent years, such as [KGM<sup>+</sup>04, DGS04, KKKR05, KH06, KBH07, ZZL07]. Krüger et al. [KGM<sup>+</sup>04] recognize, as we do, that service behavior emerge from the collaboration between roles played by components. They propose a scenario-based approach to service engineering consisting of several steps. First, use cases are used to provide a large-scale, scenario-based view of the system. From the use cases, sets of roles and services are identified. Service structure and role bindings are described with, so-called, role and deployment domain models, while the detailed behavior of services, and their relationships, are specified with an extended MSC language [Krü00]. A special operator called *join* is used to compose service scenarios with overlapping interaction patterns. This operator synchronizes the service scenarios on their shared messages, and otherwise interleaves non-shared messages. In our approach we propose a concise encapsulation of service functionality, so that no overlapping exists between the behaviors of any two service collaborations. While the use of the join operator may lead to more compact models, it hides subtle dependencies that may exist between the services. In our approach, such dependencies must be explicitly specified. We believe this contributes to a better understanding of the specification and to the detection of potential conflicts.

Deubler et al. [DGS04] also suggest a combination of use-cases and sequence diagrams for service specification, but do not recognize the crosscutting nature of services. They rather consider services to be “small self-contained functional entities responsible for a number of activities belonging together”.

The works in [KKKR05], with focus on modeling web service collaboration protocols, and in [KH06, KBH07, San07, ZZL07], with focus on modeling distributed reactive systems, are all based on UML 2 collaborations, which are used to specify structural properties of services. The main differences between these works, and our own, lie on how behavioral aspects are described. Kramler et al. [KKKR05] use activity diagrams to specify the behavior of elementary collaborations in terms of (web service) transactions. Each transaction is in turn described by another activity diagram, or by a sequence diagram. However the authors do not address the composition and choreography of collaborations, which is a central part in our work. Kraemer et al. [KH06, KBH07] also use activity diagrams to specify both the behavior of elementary collaborations and their composition. In their activity

diagrams each elementary collaboration is represented with a unique *call behavior action*, which may be (re)visited by a control token multiple times. In our approach, the choreography graph is intended to describe the evolution of a service by showing the phases it goes through. The same elementary collaboration may take place in different phases, so several call behavior actions in the choreography graph may refer to the same elementary collaboration. Our choreography graphs may therefore be seen as an unfolding of the activity diagrams of Kraemer et al. In [San07] Sanders propose to use so-called collaboration goal sequences, based on UML interaction overview diagrams, in order to describe the runtime ordering of the sub-collaborations of a main composite service collaboration. The intention is to show the order in which the goals of the individual collaborations has to be achieved in order for the main service goal to be fulfilled. Invocation and interrupting composition of collaborations are not supported. Zhang et al. [ZZL07] also use UML interaction overview diagrams to describe the composition of sequence diagrams specifying the behavior of elementary two-party collaborations. As in the case of [San07], they do not support invocation and interrupting composition of collaborations.

Other modeling approaches have been suggested that, despite not being service-oriented, may well be used for service modeling. Rößler et al. [RGG01] suggest a collaboration-based specification approach for distributed systems. They created a specific language, CoSDL, to describe both the behavior of collaborations and their composition. CoSDL is highly inspired by SDL, so it fails to provide the high-level service view offered by UML collaborations and choreographies. Amyot [Amy01] proposes a methodology to specify services using a combination of Use Case Maps, to describe scenarios and their relationships, and LOTOS, for formal validation of the specification. In Paper 1 we also used Use Case Maps, but rather than to describe scenarios, we used them to describe the choreography of collaborations. Whittle [Whi07] introduces Use Case Charts, a precise notation that allows specifying use cases at three levels of abstraction. First, the relationships between use cases are specified with an extended UML activity diagram. Each individual use case is then described as a collection of scenarios, whose relationships are again described with an extended UML activity diagram. Finally, the behavior of each scenario is detailed with a sequence diagram. Use Case Charts are able to describe sequential, concurrent and alternative relationships between scenarios/use cases, as well as preemption and suspension. Negative behaviors can also be described. Use Case Charts pursue the same goal as the choreographies presented in this work, that is, specify explicitly the relationships between “pieces” of crosscutting behavior. The main difference from our approach lies in the way such “pieces” of crosscutting behavior are structured and interpreted. Use Case Charts specify relationships between existential scenarios, which may partially overlap. We specify relationships between collaborations, which describe the complete and exact crosscutting behavior needed to achieve a certain goal.

## 3.2 Work on synthesis

The synthesis problem, as considered in this thesis, is related to the work on synthesis of communication protocols from service specifications (see [PS91] for a survey). In this context a service specification describes the actions that are executed at different "service access points", as well as their temporal order. The goal is then to derive the messages that the different system components must exchange to provide the specified service (i.e. to guarantee that the service actions are executed in the specified order). The assumption in protocol synthesis is that the service specification is complete. The problem is then to properly coordinate the system components to provide the exact specified behavior.

More recently, a considerable amount of work has been devoted to the synthesis of component behaviors from scenario-based specifications (see [AE03, LDD06] for a survey and comparison). The most obvious difference between the proposed approaches is the choice they make regarding the source scenario notation (e.g. (H)MSCs, LSCs, UML sequence diagrams) and the target construction model (e.g. UML and ROOM statecharts, SDL state-machines, Label Transition Systems(LTSs)). Although this choice may look as a matter of taste, it has important implications on the synthesis process and its final result. Recall that the goal of the synthesis process is to obtain a distributed implementation able to behave according to the specified behavior. For that a synthesis algorithm has to put together the slices of behavior that each component executes in each one of the source scenarios. The quality of the result (i.e. whether the synthesized components behave as specified both locally and globally), as well as the complexity of the process will therefore depend on the completeness of information provided by the specification (i.e. scenarios and their relationships) and the assumptions made about the communication service used by the target model (i.e. synchronous/asynchronous communication, one or several input buffers per component).

Some of the synthesis approaches that have been proposed take as input a set of scenarios without any extra information about their relationships. This is the case for [KM94, MS01, HK02, BSL04, HKP05, BHS05, SD06]. Both [KM94] and [MS01] consider as input for their algorithms a set a plain scenarios, described with either trace diagrams [KM94] or sequence diagrams [MS01]. In both cases the scenarios represent examples of system execution. The proposed algorithms compose the behavior of the source scenarios by inferring and merging common states based on sent and received messages. In [KM94], which proposes a fully automatic approach, states that are logically different may be merged, resulting in overgeneralized state machines (i.e. with more behavior than the specified one). To avoid such overgeneralization, [MS01] requires interaction with the designer to accept or reject the conjectures made by its learning algorithm. The synthesis approaches presented in [HK02], [BSL04] and [SD06] consider LSCs as the scenario notation. LSCs allow to specify both universal and existential scenarios. Universal scenarios describe behavior that must occur in all system executions (i.e. is mandatory), while existential scenarios describe behavior that may occur in a given system execution (i.e. is optional). Both types of scenarios allow additional events not present in

the scenarios to happen before, after and between those events contained in the scenarios. This is a main difference with other synthesis approaches, which normally interpret scenarios as describing the exact behavior of the participating components at a given time (i.e. no other behavior can happen). The use of both existential and universal scenarios, and the absence of explicit mechanisms to inter-relate the scenarios makes the synthesis from LSCs a complex task. To reduce such complexity the authors of [HK02] propose an alternative approach in [HKP05], which requires the interaction of the designer. The authors of [BSL04] also propose a more efficient algorithm in [BHS05]. This new algorithm is sound, but not complete. This means that it may fail to synthesize specifications that are realizable, but any synthesized system will be correct. To guarantee that the joint behavior of the synthesized components is according to the specified one, [HK02] allow components to share all their information with each other, and [BSL04] assumes every component can sense every event in the system. Both approaches are however unrealistic. [HKP05] and [SD06] add synchronization messages to guarantee that the distributed components remain coordinated on the transitions between scenarios. The algorithm in [BHS05] will stop if the specified behavior cannot be properly distributed without additional messages or data.

Like the approaches discussed above, the synthesis algorithms presented in [KGSB99], [WS00] and [KEK01] take as input a set of unordered scenarios. However, they require the designer to provide extra information that can be used to relate the source scenarios by identifying common states in them. In [KGSB99] state information is directly included in the input MSCs by means of conditions. Both [WS00] and [KEK01] use pre- and post-conditions, expressed in OCL, to give semantic to the messages of UML 1.4 sequence diagrams and collaboration diagrams<sup>1</sup>. The algorithm of [KEK01] is able to detect and avoid situations leading to overgeneralization of the synthesized components.

A third group of synthesis approaches takes as input for their algorithms scenarios that have been explicitly composed by the designer [LMR98, MZ99, UKM03, ZHJ04]. Both [LMR98], [MZ99] and [UKM03] use HMSCs to compose a set of MSCs from which ROOM statecharts, SDL state-machines, or LTSs, respectively, are synthesized. The work in [UKM03] also allows states to be identified in the MSCs by means of conditions. The work in [ZHJ04] considers UML 2 sequence diagrams as the scenario notation, and exploits the composition operators that this notation offers to compose basic sequence diagrams. Reference [LMR98] (and presumably [ZHJ04]) requires the composed scenarios to be mutually exclusive (i.e. the same event cannot be contained in two scenarios), while this is not required by [MZ99] and [UKM03]. Common to all these works is that the joint behavior of the components that are synthesized may differ from the specified one.

We have proposed a synthesis approach that is able to generate hierarchical state machines from a choreography of collaborations, which defines the runtime ordering of such collaborations. In this respect our approach is similar to other synthesis methods that require explicit composition of scenarios. However, we consider collaborations describing complete and exact behavior, rather than existential scenarios.

---

<sup>1</sup>These are called communication diagrams in UML 2

Our synthesis algorithm does not try to avoid implied scenarios. It rather assumes as input a realizable choreography (i.e. the choreography have already been analyzed in search of realizability problems). Our algorithm is strongly influenced by the work of Whittle [WJ06], who has also presented an algorithm able to generate hierarchical state machines from Use Case Charts. Kraemer et al. [KH07, KBH07] have described an algorithm that transforms activity diagrams describing collaborations into state machines. Our synthesis algorithm also take as input an activity diagram (i.e. the choreography graph), but this diagram specifies only the relationships between collaborations, whose detailed behavior is described using sequence diagrams.

### 3.3 Work on realizability and implied scenarios

The realizability of specifications of reactive systems was first studied, in general terms, in [ALW89]. In the context of MSC-based specifications it was first considered in [AEY00], where the authors relate the problem of realizability to the notion of implied scenarios. They consider a specification given as a set of MSCs describing asynchronous interactions, and analyze it to check if it implies any non-specified MSC. Intuitively, a realizable specification does not contain implied scenarios. The authors propose two notions of realizability, depending on whether the realization is required to be deadlock-free (safe realizability) or not (weak realizability). This work was extended in [AEY05] to consider realizability of bounded HMSCs [AY99]. Reference [Loh03] extends in turn the work of [AEY05] and provides some complexity results for the class of globally-cooperative HMSCs [Mor02, GMSZ06], which is less restrictive than the class of bounded HMSCs. Realizability of HMSCs with synchronous communication is considered in [UKM04]. The authors present a technique to detect implied scenarios from a specification describing both positive, as well as negative scenarios. The realizability notion considered in [AEY05] and [Loh03] does not allow adding data into messages or adding extra synchronization messages. This is seen as a very restrictive notion of realizability by some authors, who propose a notion of realizability where additional data can be incorporated into messages [MKS00, BM03, GMSZ06]. The authors of [MKS00] study safe realizability, with additional message contents, of regular (finite state) HMSCs with FIFO channels. This work is extended in [BM03], where non-FIFO communication is considered. The authors identify a subclass of HMSCs, so-called coherent HMSCs, which are safely realizable with additional message contents. However, checking whether an HMSC is coherent is in general hard. [GMSZ06] discusses two classes of unbounded HMSCs that are always realizable with data. In particular, they show that so-called local-choice HMSCs are always safely realizable with additional message contents. The authors, however, fail to explain precisely how the message contents should be generated, as we explained in Paper 5. Moreover, with this class of HMSCs some useful and common behaviors cannot be specified. A subclass of local-choice HMSCs that are safely realizable without additional message contents was studied in [HJ00].

Other authors have studied conditions for realizability of Compositional MSCs [MRW06] and pathologies in HMSCs [BAL97, H el01] and UML sequence diagrams [BBJ<sup>+</sup>05] that prevent their realization.

Our work on realizability has been quite pragmatic. We have focused on studying the problems that make a collaboration-oriented service model non-realizable. We have done this by studying the way collaborations can be composed in choreography graphs and the problems that may arise from the composition. We have analyzed the actual nature of realizability problems, which has allowed us to discuss appropriate solutions for them. In our study, we have considered the notion of direct-realizability, where no additional synchronization messages or additional message contents are added to realize the system. We consider such measures as solutions to realizability problems that should be applied depending on each specific situation, possibly guided by a designer.

---

## Summary of the papers

In this chapter we provide a short overview of each of the papers included in Part II and specify this author's contribution to each of them. These papers document the main results of this thesis work.

**Paper 1:** Humberto Nicolás Castejón. Synthesizing state-machine behavior from UML collaborations and Use Case Maps. In *Proc. 12th SDL Forum*, volume 3530 of *LNCS*, pages 339-359. Springer, June 2005.

This paper presents our first attempt at using UML 2 collaborations for compositional service specification. Through a small case study, we show how a service can be described as a collaboration consisting of other subordinate collaborations. We then discuss the importance of explicitly describing the dependencies between those sub-collaborations, and show how Use Case Maps (UCMs) can be used for this purpose. UCMs describe the causal flow of behavior of a system by ordering the system's responsibilities (e.g. tasks, actions, etc) along a path, and by linking causes (e.g. pre-conditions or triggering events) to effects (e.g. post-conditions). In the paper, UCMs are used to describe the progress of individual collaborations in terms of achievement of their service goals, and to relate different collaborations. Patterns for sequential dependencies, and for goal dependencies (i.e. cases where a collaboration depends on the success of other collaboration(s) in order to achieve its own goal) are given. Finally, we present an algorithm to synthesize the state-machine behavior of service roles from the joint information provided by UML collaborations, sequence diagrams (describing the behavior of elementary collaborations) and UCM diagrams.

*Contribution of this thesis' author:* Sole author.

**Paper 2:** Richard Torbjørn Sanders, Humberto Nicolás Castejón, Frank Alexander Kraemer, and Rolv Bræk. Using UML 2.0 collaborations for compositional service specification. In *Proc. ACM/IEEE 8th Intl. Conf. on Model Driven Engineering Languages and Systems (MoDELS'05)*, volume 3713 of *LNCS*, pages 460-475. Springer, October 2005.

This paper studies the suitability of UML 2 collaborations for model-driven service engineering. Collaborations are shown to have many interesting properties for the compositional specification of services. The paper shows how the structure of

services can be described with collaborations, and how the behavior of a collaboration can be specified at different levels of abstraction for the purpose of service specification. The composition of collaborations is discussed, and so-called (*collaboration*) *goal sequence diagrams* are presented to describe relationships between the sub-collaborations of a large collaboration. The concept of choreography, used in this thesis, and the concept of goal sequence, as presented in the paper, are closely related. The goal sequences presented in the paper can be seen as choreographies that describe only the ordering of sub-collaborations needed to achieve the goal of a service collaboration (i.e. goal sequences do not describe complete behavior). UML interaction overview diagrams are proposed in the paper to describe goal sequences. Interrupting and invocation compositions of collaborations can therefore not be specified.

*Contribution of this thesis' author:* The paper was written in close collaboration with the three other authors. This author was responsible for approximately 30% of the work.

**Paper 3:** Humberto Nicolás Castejón and Rolv Bræk. A collaboration-based approach to service specification and detection of implied scenarios. In *Proc. of 5th Int. Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM'06)*. ACM Press, May 2006.

This paper presents a service modeling approach based on UML 2 collaborations. A central element in this approach are, again, collaboration goal sequences, whose original syntax and semantics are modified and extended, so that collaboration invocation relationships can be described. In the second part of the paper we show how implied scenarios can be detected in collaboration-based specifications by analysing sequences of roles extracted from the collaboration goal sequence diagram. In the paper, role sequences are first analyzed individually. The implied scenarios detected this way follow from realizability problems associated with one individual occurrence of a service collaboration. The proposed analysis fails to detect all realizability problems (see Appendix A in the paper). A more complete study of these problems is presented in Paper 5. The second analysis technique proposed in the paper is able to detect implied scenarios resulting from the unexpected interaction of several concurrent collaboration occurrences (of the same or different collaboration type).

*Contribution of this thesis' author:* Main author, responsible for approximately 90% of the work.

**Paper 4:** Humberto Nicolás Castejón and Rolv Bræk. Formalizing collaboration goal sequences for service choreography. In *Proc. 26th IFIP WG 6.1 Intl. Conf. on Formal Methods for Networked and Distributed Systems (FORTE'06)*, volume 4229 of *LNCS*, pages 275-291. Springer, September 2006.

This paper extends the work in Paper 3 by providing formal syntax and semantics to collaboration goal sequences. Formal semantics is given by mapping goal sequences to hierarchical coloured Petri-nets (HCPNs). This allows to use general purpose tools available for HCPNs for the detection of implied scenarios.

*Contribution of this thesis' author:* Main author, responsible for approximately 90% of the work.

**Paper 5:** Humberto Nicolás Castejón, Gregor von Bochmann, and Rolv Bræk. Using collaborations in the development of distributed services. Technical report, Avantel 2/2008 ISSN 1503-4097, NTNU, February 2008. This is an extended and revised version of: Humberto Nicolás Castejón, Gregor von Bochmann, and Rolv Bræk. Realizability of collaboration-based service specifications<sup>1</sup>. In *Proc. 14th Asia-Pacific Soft. Eng. Conf. (APSEC'07)*. IEEE Computer Society, December 2007.

The paper discusses the use of collaborations in the development of services. It provides a discussion on the semantics of service collaborations, as well as possible notations for describing collaboration choreographies, with an emphasis on identifying concepts that allow specifying and analyzing the high level service flow without prematurely binding the detailed interactions. A main contribution of this paper is an extensive analysis of the type of realizability problems that may appear in choreography-based service specifications. Realizability problems are classified in terms of the composition operators that can be used in a choreography. The actual nature of these problems, as well as possible solutions, are discussed. Design guidelines are also provided. The paper ends with a discussion on the derivation of component behaviors from choreography-based service specifications.

*Contribution of this thesis' author:* The paper was written in close collaboration with the two other authors. This author was responsible for approximately 60% of the work, and main contributor to Section 3 (i.e. the study of realizability).

**Paper 6:** Humberto Nicolás Castejón, Gregor von Bochmann, and Rolv Bræk. Investigating the realizability of collaboration-based service specifications. Technical report, Avantel 3/2007 ISSN 1503-4097, NTNU, September 2007.

This paper has two main contributions. The first contribution is a formal syntax for choreography graphs and sequence diagrams, as well as a formal semantics based on partial orders. In particular, formal semantics are provided for sequential, parallel, alternative, invocation and interruption compositions of collaborations. The second contribution is a set of algorithms for the detection of race conditions and choice propagation problems in a choreography. The paper presents also a preliminary

---

<sup>1</sup> *Contribution of this thesis' author to the APSEC paper:* Main author, responsible for approximately 90% of the work.

version of our work on realizability problems. The reader may skip this part (i.e. Section 4 of the paper) in favor of Paper 5.

*Contribution of this thesis' author:* Main author, responsible for approximately 90% of the work.

**Paper 7:** Humberto Nicolás Castejón and Rolv Bræk. Dynamic role binding in a service oriented architecture. In *Proc. IFIP Intl. Conf. on Intelligence in Communication Systems (INTELLCOMM'05)*, volume 190 of *IFIP International Federation for Information Processing*. Springer, October 2005.

This paper discusses mechanisms to achieve modularity and flexibility in the deployment and execution of services. The paper recognizes that service modularity requires a separation between service roles (i.e. service logic) and system components. It then identifies the process of *dynamic role binding* (i.e. dynamically creating and releasing links between system components that execute appropriate service roles) as crucial to achieve such separation, while being able to handle context dependency, personalization, resource limitations and compatibility validation. A service-oriented architecture is presented that supports service modularity and dynamic role binding. This architecture provides a natural way to structure service-execution policies for flexible management of the dynamic role binding process. A policy framework is introduced.

*Contribution of this thesis' author:* The paper was written in close collaboration, with this thesis' author as the main author, responsible for approximately 75% of the work.

---

## Conclusions

In this thesis we have presented a service-oriented and model-driven approach to engineering distributed reactive systems. The approach integrates model creation and analysis tightly, resulting in an iterative process where the understanding of the modeled behavior, as well as the confidence in its correctness, increases with each iteration.

In the following we summarize and discuss the work that has been done. Thereafter we identify limitations and suggests directions for future work.

### 5.1 Summary and Discussion

We summarize and discuss here our results on three specific areas: service modeling, service analysis and the problem of dynamic role binding.

#### Service Modeling

In the proposed modeling approach the behavior of a system is first decomposed in terms of the services that the system provides. Each of these services is then modeled separately. UML 2 collaborations are used to represent an individual occurrence of each service as a structure of collaborating service roles, which interact to achieve the service's goal. Complex service collaborations are decomposed into sub-collaborations representing more elementary services or service features. Such decomposition tend to result in elementary collaborations associated with interfaces, whose behavior can be completely described using sequence diagrams. The full behavior of a composite collaboration is described by means of a choreography graph, which defines the global execution ordering of its sub-collaborations. The service models are then used as input for the construction of a design model, which describes the system architecture, as well as the complete local behavior of each system component type. For that purpose the system is modeled as a UML structured class, where the inner parts, with type and multiplicity, represent the system components, and the connectors between the parts represent the communication channels. The complete local behavior of each system component type is obtained in two steps. First, for each service model the local behavior of each service role is automatically synthesized in the form of a communicating state machine. Thereafter, roles are assigned to the component types that will play them. The complete local behavior of each component type is then designed as a composition of the roles it plays. At

this point, the need for coordination mechanisms between different role instances and types is considered, in case a component may participate in multiple concurrent occurrences of the same service, or of different services.

The main motivation for our work in this area has been twofold. First, most traditional modeling approaches do not model services explicitly. They rather focus on the behavior of system components. However, as we showed in the Introduction chapter, the behavior of these components depends on the services they are involved in (recall that service functionality arises from the interaction among components). Having a clear and explicit view of services is thus crucial for a flexible and efficient engineering of reactive systems. After all, reactive systems exist to provide services to their environment [WJ01]. In our modeling approach, services are explicitly modeled as UML 2 collaborations, and the behavior of components is semi-automatically derived from the service models. Our work is therefore related to other approaches where component behavior is (semi-)automatically synthesized from scenarios (e.g. MSC or sequence diagrams) describing crosscutting system behavior. Collaborations offer powerful means to structure and reuse such crosscutting behavior, and to provide a high-level overview of it. In the early stages of the development process they allow to focus on the main purpose or goal of the interactions to take place in the system, without getting buried in the details of those interactions. Moreover, some unexpected behaviors can already be detected by analysing the choreography of collaborations, without considering the detailed behavior of those collaborations.

The second reason that motivated our work was the usability of current modeling methods. Formal semantics is a highly desirable property of any modeling language or approach. It makes the meaning of the created models precise, so they can be automatically analyzed. However, industry practitioners usually find formal methods difficult to learn and use, and often resort to informal methods, such as UML. The result are imprecise models that usually contain implementation details and are hard to analyze [Hei98, Hei05]. In addition, there is also a general lack of guidelines for the construction of correct and high-quality models. Our work tries to address these issues in several ways. On one hand, we use UML elements for the construction of service and design models<sup>1</sup>, but we have sought to give a precise semantics to these models. On the other hand, our modeling approach aims to be *constructive* (i.e. aims to generate systems without errors) rather than *corrective* (i.e. aims to detect and correct the errors that are nonetheless made) [BH93]. To achieve this goal it is important to break down the complexity of the modeling process, and to generate models that are easy to understand by humans. We do this by applying the well-known principles of separation of concerns, abstraction and modularity at different levels:

- System behavior is decomposed into services, and services are in turn decomposed into more elementary services or service features. Services and service features are then modeled separately. This allow to focus at any time on the

---

<sup>1</sup>We use UML collaborations, sequence diagrams and state machines. We also use activity diagrams to describe choreographies, although in a way that is not completely compliant with the UML standard, as explained in Appendix 10.A on page 182.

behavior that is needed to achieve a certain goal. An explicit (re-)composition of the services and service features is then required, which forces one to highlight dependencies and to detect possible interactions between services.

- Services are modeled independently of the components that provide them. For each service we focus on the role behaviors that are needed to provide such service, and abstract away from the actual system components that will eventually execute those roles.
- In a system, potentially many occurrences of the same service may simultaneously coexist. Initially, we abstract away from this possibility, and focus on modeling one isolated occurrence of each service, that is, only the role behaviors needed for an occurrence of a service are modeled. It is during the construction of the design model that we deal with the possibility of having multiple concurrent occurrences of the service running in the system.

The proposed modeling approach has been applied to different use cases with positive results. In some cases, we had some a priori understanding of the services being modeled, and the potential problems that we could face during their specification and design. We could then test the ability of our approach to identify the known problems. In other cases, we undertook the modeling process of some services starting from a rough description of their requirements. We were then able to gain a better understanding of the service functionality during the construction of the service models, and to detect problems that were a priori unknown. While these results helped build confidence in the practical value of our modeling approach, a more comprehensive evaluation of the approach by means of industrial size case studies is still needed.

## Service Analysis

The modeling approach presented in this thesis has its place in the early stages of the development cycle, where the set of requirements of the system under development may not yet be complete. We have therefore investigated techniques for early analysis of the service and design models that help to discover undesired behaviors, as well as acceptable behaviors that were overlooked while studying the problem domain.

The proposed analysis approach is closely tied to the modeling approach and its inherent separation of concerns. First, each individual service model is analyzed in search of realizability problems. Then, during the construction of the design model, potential interactions between the roles that any component may play are analyzed and coordination measures taken.

The realizability analysis assumes a situation where the service roles synthesized from a service model are executed by distributed system components communicating by means of asynchronous message passing. It then determines whether the joint behavior of the set of service roles exactly corresponds to the global behavior specified by the service model. Realizability problems lead to implied scenarios [AEY00], that

is, behaviors that are not explicitly specified in the service model. Some authors have proposed techniques to find implied scenarios in MSC-based specifications. These techniques are able to determine whether there exist a realizability problem in a specification and to eventually show the consequences of such problem. They fail, however, to determine the nature or cause of the problem. We believe that having a clear understanding of the actual nature of realizability problems is essential, not only to adopt the most appropriate resolution when they are detected, but also to avoid them in the first place. This has motivated us to investigate, for each possible way of composing collaborations in a choreography, the realizability problems that may arise and the underlying reasons leading to them. We have also discussed possible ways to resolve them, and a set of algorithms have been proposed to detect realizability problems.

In the proposed modeling approach, service collaborations are modeled separately without initially considering any potential dependencies between them (e.g. due to shared resources). Moreover, in order to model each service collaboration explicitly only one occurrence of the collaboration is considered. In some cases, however, the final system should allow multiple occurrences of the same collaboration to run simultaneously. Those collaboration occurrences might not be totally independent. Therefore, if a system component is to play roles in different service collaborations and/or in multiple occurrences of the same service collaboration, unexpected interactions between those roles may arise. Such interactions should be detected before the role behaviors are composed into the full behavior of the system component. To deal with them, appropriate coordination mechanisms may need to be added to the component's behavior, and the original service models may need to be modified. As Jackson points out in [Jac05] in the context of problem frames, we believe this is not a disadvantage of the proposed separation of concerns, but rather an indication that dealing with all dependencies from the beginning would have been very difficult. In this thesis we have proposed an automatic method to detect role interactions due to shared resources. This method makes use of pre- and post-conditions associated with role executions, and does not require the detailed behavior of roles to be specified.

## Dynamic Role Binding

Service discovery and adaptation, and service personalization based on the service context and end-user preferences are increasingly important mechanisms. These mechanisms require system components to learn new behavior or adapt their current one according to the context and user preferences. This can be achieved by letting system components dynamically select the appropriate roles to be executed as part of a service. We call this *dynamic role binding*. We have proposed a service-oriented architecture and a policy framework where policies can be used to control the dynamic binding of roles to system components. These policies allow to express constraints that need to be satisfied for a role to be executed. They also allow to select alternative roles when those constraints cannot be satisfied. A prototype of the proposed policy-based and service-oriented architecture has been implemented as part of [Pha05].

## 5.2 Limitations and Future work

The modeling approach presented in this thesis could be extended in several directions:

- *Data handling.* Although the approach covers the synthesis of role behaviors, the focus is on control and interactions (i.e. message exchanges), while data handling is abstracted away. The state machines that are synthesized should therefore be considered as early prototypes for simulation purposes. To obtain more realistic state machines we need to add support for boolean conditions and data operations in the description of elementary collaborations, and extend the synthesis algorithm accordingly.
- *Synthesis of full component behavior.* While the behavior of roles can be automatically synthesized from a choreography graph, building the complete behavior of system components, as a composition of coordinated roles, has to be done manually. In cases where the roles are completely independent, this might be as simple as creating a state machine with several orthogonal regions and adding the state machine behavior of each role to one of the orthogonal regions. In other cases mechanisms to coordinate the roles behaviors may need to be added, such as a controller state machine that would decide whether or not to instantiate the role state machines when requested. We believe that, for a given domain, a number of useful coordination patterns could be identified. It would then be possible for a designer to select the most appropriate pattern from a library, and have it automatically instantiated and composed with the role behaviors to build the complete behavior of components.
- *Incremental specification and refinement.* In this thesis we have considered that services and components' behavior are modeled from scratch. This may however not always be the case in real projects. It would be interesting to study how functionality can be incrementally added to an existing service model in a modular way, and how that functionality can be incrementally deployed. Finding rules for systematic refinement of collaboration behavior is also an interesting area for future work. Here, the work on refinement of MSCs done by Krüger [Krü00], and the work on refinement of UML interaction diagrams done as part of the STAIRS method [Run07, HHRS05] may be used as a starting point.
- *Partial choreography views.* For complex choreography graphs it could be handy to work with partial views of the graph, each of them describing the global collaboration behavior from the point of view of a given role (as illustrated with the TeleConsultation example). For an effective use of views, mechanisms for automatically checking the consistency between several partial views and for merging them are needed.

The work on realizability could also be extended on several directions:

- *Algorithms.* The algorithms for the detection of race conditions and ambiguous/race propagation should be extended to detect those problems in the presence of interrupting and invocation compositions. Support for a more compositional analysis should also be investigated. The current algorithms assume that the collaborations referred to in the choreography graphs are elementary. If this is not the case, they require the choreography graph to be flattened.
- *Automatic resolution of conflicts.* In this thesis we have already identified and discussed some possible solutions for the different realizability problems. We envision a library of domain-specific solutions, from which a designer could choose the most appropriate. Work needs to be done on how to add the resolution behavior into the service and/or design models, so it can be considered during the automatic synthesis of component behavior.

In the area of dynamic role binding there also some possible extensions:

- *Granularity of feature-selection policies.* In the current implementation of the service-oriented and policy-based architecture discussed in Paper 7, feature-selection policies are used to replace the requested roles with other alternative roles. Another interesting approach would be to use those policies to enable or disable specific parts of the behavior of a (composite) role.

In general, work has to be done in all the aforementioned areas in order to implement the proposed solutions. We envision a tool that allows the modeling of services as explained in this thesis, the analysis of the created models and the automatic generation of state machines.

---

## References

- [AE03] Daniel Amyot and Armin Eberlein. An evaluation of scenario notations and construction approaches for telecommunication systems development. *Telecommunication Systems*, 24(1):61–94, 2003.
- [AEY00] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of message sequence charts. In *Proc. of the 22nd Int. Conf. on Software Engineering (ICSE'00)*, 2000. ACM Press.
- [AEY05] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Realizability and verification of MSC graphs. *Theoretical Computer Science*, 331(1):97–114, 2005.
- [AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.
- [ALW89] Martín Abadi, Leslie Lamport, and Pierre Wolper. Realizable and unrealizable specifications of reactive systems. In *Proc. of the 16th Intl. Colloquium on Automata, Languages and Programming (ICALP'89)*, volume 372 of *LNCS*, pages 1–17, 1989. Springer.
- [Amy01] Daniel Amyot. *Specification and Validation of Telecommunications Systems with Use Case Maps and LOTOS*. PhD thesis, School of Information Technology and Engineering (SITE), University of Ottawa, September 2001.
- [AY99] Rajeev Alur and Mihalis Yannakakis. Model checking of message sequence charts. In *Proc. of the 10th Intl. Conf. on Concurrency Theory (CONCUR'99)*, volume 1664 of *LNCS*, pages 114–129, 1999. Springer.
- [BAL97] Hanene Ben-Abdallah and Stefan Leue. Syntactic detection of process divergence and non-local choice in message sequence charts. In *Proc. of the 2nd Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, volume 1217 of *LNCS*, pages 259–274, 1997. Springer.
- [BBJ<sup>+</sup>05] Paul Baker, Paul Bristow, Clive Jervis, David King, Robert Thomson, Bill Mitchell, and Simon Burton. Detecting and resolving seman-

- tic pathologies in uml sequence diagrams. In *Proc. of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-13)*, pages 50–59, 2005. ACM Press.
- [BC96] R. J. A. Buhr and R. S. Casselman. *Use case maps for object-oriented systems*. Prentice-Hall, Inc., 1996.
- [BDC<sup>+</sup>89] T.F. Bowen, F.S. Dworack, C.H. Chow, N. Griffeth, G.E. Herman, and Y.J. Lin. The feature interaction problem in telecommunications systems. In *Proc. of the 7th Int'l Conf. Soft. Eng. for Telecommunications Switching Systems (SETSS'89)*, pages 59–62, 1989.
- [BH93] Rolv Bræk and Øystein Haugen. *Engineering Real Time Systems. An object-oriented methodology using SDL*. Prentice Hall, Englewood Cliffs, NJ, 1993.
- [BHS05] Yves Bontemps, Patrick Heymans, and Pierre-Yves Schobbens. Lightweight formal methods for scenario-based software engineering. In *Proc. of the 2003 Intl. Dagstuhl Work. on Scenarios: Models, Transformations and Tools*, volume 3466 of *LNCS*, pages 174–192, 2005. Springer.
- [BKM07] Manfred Broy, Ingolf H. Krüger, and Michael Meisinger. A formal model of services. *ACM Transactions on Software Engineering and Methodology*, 16(1):5, 2007.
- [BM03] Nicolas Baudru and Rémi Morin. Safe implementability of regular message sequence chart specifications. In *Proc. of the 4th ACIS Intl. Conf. on Soft. Eng., Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'03)*, pages 210–217, 2003.
- [Boc78] Gregor Bochmann. Finite state description of communication protocols. *Computer Networks*, 2:361–372, 1978.
- [Bræ79] Rolv Bræk. Unified system modeling and implementation. In *Proc. of the International Switching Symposium (ISS)*. ISS Committee, 1979.
- [Bræ99] Rolv Bræk. Using roles with types and objects for service development. In *proc. of the IFIP TC6 WG6.7 5th International Conference on Intelligence in Networks (SMARTNET)*, volume 160 of *IFIP Conference Proceedings*, pages 265–278, Pathumthani, Thailand, 1999. Kluwer.
- [BS80] Gregor Bochmann and Carl A. Sunshine. Formal methods in communication protocol design. *IEEE Transactions on Communications*, 28(4), April 1980.
- [BSL04] Yves Bontemps, Pierre-Yves Schobbens, and Christof Löding. Synthesis of open reactive systems from scenario-based specifications. *Fundamenta Informaticae*, 62(2):139–169, July 2004.

- [Buh98] R. J. A. Buhr. Use case maps as architectural entities for complex systems. *IEEE Transactions of Software Engineering*, 24(12):1131–1155, 1998.
- [DGS04] Martin Deubler, Johannes Grünbauer, and Chris Salzmänn. Towards a model-based and incremental development process for service-based systems. In *Proc. of the IASTED Intl. Conf. on Software Engineering (IASTED SE'04)*, pages 183–188. IASTED/ACTA Press, February 2004.
- [Erl05] Thomas Erl. *Service Oriented Architecture: Concepts, Technology and Design*. Number ISBN 0-13-185858-0. Prentice Hall, 2005.
- [FK01] Kathi Fisler and Shriram Krishnamurthi. Modular verification of collaboration-based software designs. In *Proc. of the 8th European software engineering conference held jointly with 9th ACM SIG-SOFT international symposium on Foundations of software engineering (ESEC/FSE-9)*, pages 152–163, 2001. ACM Press.
- [GMSZ06] Blaise Genest, Anca Muscholl, Helmut Seidl, and Marc Zeitoun. Infinite-state high-level mscs: Model-checking and realizability. *Journal of Computer and System Sciences*, 72(4):617–647, 2006.
- [Hei98] Constance L. Heitmeyer. On the need for practical formal methods. In *Proc. of the 5th Intl. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'98)*, volume 1486 of *LNCS*, pages 18–26, 1998. Springer.
- [Hei05] Constance L. Heitmeyer. A panacea or academic poppycock: Formal methods revisited. In *proc. of the 9th IEEE Intl. Symp. on High Assurance Systems Engineering (HASE'05)*, pages 3–7, 2005.
- [Hél01] Loïc Hélouët. Some pathological message sequence charts, and how to detect them. In *Proc. of the 10th Intl. SDL Forum*, volume 2078 of *LNCS*, pages 348–364, 2001. Springer.
- [HHRS05] Øystein Haugen, Knut Husa, Ragnhild Runde, and Ketil Stølen. STAIRS towards formal design with sequence diagrams. *Software and Systems Modeling*, 4(4):355–357, November 2005.
- [HJ00] Loïc Hélouët and Claude Jard. Conditions for synthesis of communicating automata from HMSCs. In *Proc. of the 5th Intl. Workshop on Formal Methods for Industrial Critical Systems (FMICS'00)*. GMD FOKUS, 2000.
- [HK02] David Harel and Hillel Kugler. Synthesizing state-based object systems from LSC specifications. *Intl. Journal of Foundations of Computer Science*, 13(1):5–51, 2002.

- [HKP05] David Harel, Hillel Kugler, and Amir Pnueli. Synthesis revisited: Generating statechart models from scenario-based requirements. In *Formal Methods in Software and Systems Modeling*, volume 3393 of *LNC3*, pages 309–324, 2005. Springer.
- [HP85] D. Harel and A. Pnueli. *On the development of reactive systems*, volume 13 of *Nato Asi Series F: Computer And Systems Sciences*, pages 477–498, 1985. Springer.
- [IT99] ITU-T. *ITU Recommendation Z.Z.120: "Message Sequence Chart (MSC-2000)"*. ITU, Geneva, 1999.
- [Jac05] Michael Jackson. Problem frames and software engineering. *Information & Software Technology*, 47(14):903–912, 2005.
- [KBH07] Frank Alexander Kraemer, Rolv Bræk, and Peter Herrmann. Synthesizing components with sessions from collaboration-oriented service specifications. In *Proc. of the 13th Intl. SDL Forum*, volume 4745 of *LNC3*, pages 166–185, Paris, 2007. Springer.
- [KEK01] Ismail Khriiss, Mohammed Elkoutbi, and Rudolf K. Keller. Automatic synthesis of behavioral object specifications from scenarios. *Journal of Integrated Design & Process Science*, 5(3):53–77, 2001.
- [KGM<sup>+</sup>04] Ingolf H. Krüger, Diwaker Gupta, Reena Mathew, Praveen Moorthy, Walter Phillips, Sabine Rittmann, and Jaswinder Ahluwalia. Towards a process and tool-chain for service-oriented automotive software engineering. In *Proc. of the ICSE 2004 Workshop on Software Engineering for Automotive Systems (SEAS)*, 2004.
- [KGSB99] Ingolf Krüger, Radu Grosu, Peter Scholz, and Manfred Broy. From MSCs to statecharts. In *Proc. of the IFIP WG10.3/WG10.5 Intl. workshop on Distributed and parallel embedded systems (DIPES'98)*, pages 61–71. Kluwer Academic Publishers, 1999.
- [KH06] Frank Alexander Kraemer and Peter Herrmann. Service specification by composition of collaborations—an example. In *Proc. of the 2nd Intl. Workshop on Service Composition (SERCOMP'06)*, pages 129–133, 2006. IEEE CS.
- [KH07] Frank A. Kraemer and Peter Herrmann. Transforming collaborative service specifications into efficiently executable state machines. In *Proc. of the 6th Intl. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'07)*, volume 7 of *Electronic Communications of the EASST*, 2007.
- [KKKR05] Gerhard Kramler, Elisabeth Kapsammer, Gerti Kappel, and Werner Retschitzegger. Towards using UML 2 for modelling web service collaboration protocols. In *Proc. of the 1st Intl. Conf. on Interoperability of Enterprise Software and Applications (INTEROP-ESA'05)*, 2005.

- [KM94] Kai Koskimies and Erkki Mäkinen. Automatic synthesis of state machines from trace diagrams. *Software - Practice and Experience*, 24(7):643–658, 1994.
- [KM03] Ingolf H. Krüger and Reena Mathew. Component synthesis from service specifications. In *proc. of the 2003 Intl. Dagstuhl Workshop on Scenarios: Models, Transformations and Tools*, volume 3466 of *LNCS*, pages 255–277, 2005. Springer.
- [Kri00] Ingolf H. Krüger. *Distributed system design with message sequence charts*. PhD thesis, Institut für Informatik, Technische Universität München, 2000.
- [LDD06] Hongzhi Liang, Juergen Dingel, and Zinovy Diskin. A comparative survey of scenario-based to state-based model synthesis approaches. In *Proc. of the 5th Intl. workshop on Scenarios and State Machines: models, algorithms, and tools (SCESM'06)*, pages 5–12, 2006. ACM Press.
- [LMR98] S. Leue, L. Mehrmann, and M. Rezai. Synthesizing ROOM models from message sequence chart specifications. Technical report, Dept. of Electrical and Computer Engineering, April 1998.
- [Loh03] Markus Lohrey. Realizability of high-level message sequence charts: closing the gaps. *Theoretical Computer Science*, 309(1-3):529–554, 2003.
- [MKS00] Madhavan Mukund, K. Narayan Kumar, and Milind A. Sohoni. Synthesizing distributed finite-state systems from MSCs. In *Proc. of the 11th Intl. Conf. on Concurrency Theory (CONCUR'00)*, volume 1877 of *LNCS*, pages 521–535. Springer, 2000.
- [Mor02] Rémi Morin. Recognizable sets of message sequence charts. In *proc. of the 19th Annual Symposium on Theoretical Aspects of Computer Science (STACS'02)*, volume 2285 of *LNCS*, pages 523–534, 2002.
- [MRW06] Arjan Mooij, Judi Romijn, and Wieger Wesselink. Realizability criteria for compositional msc. In *Proc. of the 11th Intl. Conf. on Algebraic Methodology and Software Technology (AMAST'06)*, volume 4019 of *LNCS*. Springer, 2006.
- [MS01] Erkki Mäkinen and Tarja Systä. MAS - an interactive synthesizer to support behavioral modelling in UML. In *Proc. of the 23rd Intl. Conf. on Software Engineering (ICSE'01)*, pages 15–24. IEEE Computer Society, 2001.
- [MZ99] Nikolai Mansurov and D. Zhukov. Automatic synthesis of SDL models in use case methodology. In *proc. of the 9th Intl. SDL Forum (SDL'99)*, pages 225–240. Elsevier, 1999.
- [OMG05] Object Management Group (OMG). *UML 2.0 Superstructure Spec.*, July 2005.

- [OMG07] Object Management Group (OMG). *UML 2.1.1 Superstructure Spec.*, February 2007.
- [Pha05] Quoc Tuan Pham. Policy-based service personalization. Master's thesis, Dept. of Telematics, Norwegian University of Science and Technology (NTNU), 2005.
- [PS91] Robert L. Probert and Kassem Saleh. Synthesis of communication protocols: Survey and assessment. *IEEE Transactions on Computers*, 40(4):468–476, 1991.
- [RGG01] Frank Rößler, Birgit Geppert, and Reinhard Gotzhein. Collaboration-based design of SDL systems. In *Proc. of the 10th Intl. SDL Forum (SDL'01)*, volume 2078 of *LNCS*, pages 72–89. Springer-Verlag, 2001.
- [Run07] Ragnhild Kobro Runde. *STAIRS – Understanding and Developing Specifications Expressed as UML Interaction Diagrams*. PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, 2007.
- [San07] Richard Sanders. *Collaborations, Semantic Interfaces and Service Goals – a new way forward for Service Engineering*. PhD thesis, Department of Telematics, Norwegian Univ. Science and Technology, Trondheim, Norway, March 2007.
- [SD06] Jun Sun and Jin Song Dong. Design synthesis from interaction and state-based specifications. *IEEE Transactions on Software Engineering*, 32(6):349–364, 2006.
- [TOP] TOPCASED project. URL: <http://www.topcased.org>.
- [UKM03] Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Synthesis of behavioral models from scenarios. *IEEE Transactions on Software Engineering*, 29(2):99–115, 2003.
- [UKM04] Sebastián Uchitel, Jeff Kramer, and Jeff Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Transactions on Software Engineering and Methodology*, 13(1):37–85, 2004.
- [VL85] Chris A. Vissers and Luigi Logrippo. The importance of the service concept in the design of data communications protocols. In *Proc. of the IFIP WG6.1 5th Intl. Conf. on Protocol Specification, Testing and Verification*, pages 3–17. North-Holland Publishing Co., 1985.
- [Whi07] Jon Whittle. Precise specification of use case scenarios. In *Proc. of the 10th Intl. Conf. on Fundamental Approaches to Software Engineering (FASE'07)*, volume 4422 of *LNCS*, pages 170–184, 2007. Springer

- [WJ01] Roel Wieringa and David N. Jansen. Techniques for reactive system design: The tools in TRADE. In *Proc. of the 13th Intl. Conf. on Advanced Information Systems Engineering (CAiSE'01)*, volume 2068 of *LNCS*, pages 93–107, 2001. Springer-Verlag.
- [WJ06] Jon Whittle and Praveen K. Jayaraman. Generating hierarchical state machines from use case charts. In *Proc. of the 14th IEEE Intl. Conf. on Requirements Engineering (RE'06)*, 2006.
- [WS00] Jon Whittle and Johann Schumann. Generating statechart designs from scenarios. In *Proc. of the 22nd Intl. Conf. on Software Engineering (ICSE'00)*, pages 314–323. ACM Press, 2000.
- [Zav01] Pamela Zave. Feature-oriented description, formal methods, and DFC. In *Proc. of the FIREworks Workshop on Language Constructs for Describing Features*, pages 11–26. Springer-Verlag, 2001.
- [Zav03] Pamela Zave. Feature disambiguation. In *Proc. of Feature Interactions in Telecommunications and Software Systems VII*, pages 3–9, Ottawa, Canada, 2003.
- [ZHJ04] Tewfik Ziadi, Loïc Hérouët, and Jean-Marc Jézéquel. Revisiting statechart synthesis with an algebraic approach. In *Proc. of the 26th Intl. Conf. on Software Engineering (ICSE'04)*, pages 242–251, 2004.
- [ZZL07] Pengcheng Zhang, Yu Zhou, and Bixin Li. A service-oriented methodology supporting automatic synthesis and verification of component behavior model. In *Proc. of the 8th ACIS Intl. Conf. on Soft.Eng., Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'07)*, pages 511–516. IEEE Computer Society, 2007.



**Part II**  
**Research Papers**



A series of papers representing the main body of the thesis' work is presented in the following. Papers 1 to 4, as well as Paper 7 were all published in international, peer-reviewed conferences. Papers 5 and 6 were published as technical reports. These two reports complement and/or extend the work of a paper (not included here) that was published in the international, per-reviewed APSEC'07 conference.

The format of the papers has been adapted to be included in this thesis, but the papers' content has not been modified and is presented here as it was originally published. An exception is Paper 6, where a minor error has been corrected and an endnote added to inform the reader. In other papers we have added extra material at the beginning or the end to clarify some issues.



---

## Paper 1

### **Synthesizing state-machine behavior from UML collaborations and Use Case Maps**

By Humberto Nicolás Castejón.

Published in the *Proceedings of the 12th International SDL Forum*, volume 3530 of *LNCS*, pages 339-359. Springer, June 2005.

The original publication is available at *www.springerlink.com* via [http://dx.doi.org/10.1007/11506843\\_24](http://dx.doi.org/10.1007/11506843_24)



# Synthesizing State-machine Behaviour from UML Collaborations and Use Case Maps

Humberto Nicolás Castejón Martínez

*NTNU, Department of Telematics, N-7491 Trondheim, Norway*  
humberto.castejon@item.ntnu.no

## Abstract

Telecommunication services are provided as the joint effort of components, which collaborate in order to achieve the goal(s) of the service. UML2.0 collaborations can be used to model services. Furthermore, they allow services to be described modularly and incrementally, since collaborations can be composed of subordinate collaborations. For such an approach to work, it is necessary to capture the exact dependencies between the subordinate collaborations. This paper presents the results of an experiment on using Use Case Maps (UCMs) for describing those dependencies, and for synthesizing the state-machine behaviour of service components from the joint information provided by the UML collaborations and the UCM diagrams.

## 6.1 Introduction

Telecommunication services are provided as the joint effort of active objects, which collaborate in order to achieve a goal for their environment. Initiatives may originate from any side, be simultaneous and possibly conflicting. This is what makes tele-services interesting, but at the same time particularly challenging to design.

Traditional service engineering approaches have been object-oriented. They have focused on modeling the total behaviour of objects, normally in terms of state-machines. The disadvantage of focusing on the complete behaviour of objects is that we only get a partial view of the services we want to design, which makes it difficult to understand and analyze them. Since telecommunication services are the result of collaborations among objects pursuing a goal, a collaboration-oriented approach to service engineering seems more suitable [RGG01, FK01]. A collaboration view helps to see the service as a whole, to define what roles are played by which objects, and to express what service goal combinations must be met for the successful provision of the service.

UML2.0 collaborations [OMG04, RJB04] are intended to describe partial functionalities involving interactions among participating roles played by objects. Therefore, they fit well with our understanding of service. An interesting characteristic of UML collaborations is that they can be bound to a specific context, becoming collaboration uses, which in turn can be used in the definition of larger collaborations. This feature enables a compositional and incremental design of services, which is desirable, but which will only succeed if the dependencies between the collaborations that are composed are explicitly captured [BC01].

This paper presents our approach to incremental service modeling using UML2.0 collaborations and motivates the need for explicitly expressing collaboration dependencies in this approach (see Sect. 6.2). It continues with the results of an experiment on using Use Case Maps (UCMs) [BC96, Buh98] for describing such dependencies (see Sect. 6.3) and synthesizing the state-machine behaviour of service components from the joint information provided by the UML collaborations and the UCM diagrams (see Sect. 6.4). The paper finishes with a comparison between our synthesis approach and other existing work (see Sect. 6.5), and with a summary of the presented work (see Sect. 6.6).

## 6.2 Goal-Oriented Service Collaborations

In our service engineering approach we model services by means of UML2.0 collaborations. They describe a structure of roles that collaborate to collectively accomplish some task, that is, to achieve some goal. The collaboration roles specify the properties that object instances must have in order to participate in the collaboration. The UML standard allows to associate behaviour with collaborations in several forms, like, for example, as sequence diagrams involving the collaborating roles, and as state-machines for the roles. Since our approach is collaboration-oriented, we prefer to describe the behaviour of collaborations as sequence diagrams that show the interactions between roles, rather than using state-machines for the roles.

Fig. 6.1 shows a UML collaboration diagram describing a UserLogon service. From the diagram we can see that there are five roles involved in the collaboration (represented by boxes). We can also see the relationships that are needed between these roles to achieve the goal of the collaboration. For example, Terminal is associated with TerminalClientSession, which in turn is a part of TerminalAgent and it is associated with UserTerminalSession. The diagram also shows one interesting aspect of UML collaborations: they can contain other sub-collaborations in their definition, expressed as collaboration uses. When this happens, the roles of the collaboration uses are bound to the roles of the container collaboration (e.g. *rr1*'s requested role is bound to TerminalAgent). Collaboration uses enable a modular design with small collaborations as units of reuse. Modularity is a well-proven approach to break down the complexity of systems; here we use it to structure services. It also promotes separation of concerns and reuse. These aspects are reflected in the UserLogon collaboration, which contains four sub-collaborations, namely *rr1*, *rr2*, *lo* and *ua*. The two first, *rr1* and *rr2*, are instances of a RoleRequest collaboration, while *lo* and *ua* are instances of a Logon and a UserAuthenticate collaboration, respectively. We

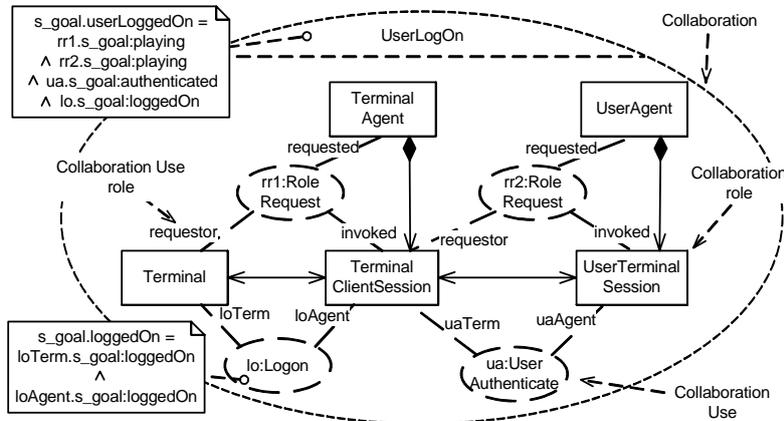


Figure 6.1: UML 2.0 Collaboration for UserLogon Service

can see that the RoleRequest collaboration has been reused. We also appreciate how separation of concerns has been achieved by separately defining the interactions between Terminal and TerminalClientSession, and between TerminalClientSession and UserTerminalSession. Indeed, although logon and authentication protocols are related, they are not exactly the same. For example, it may be perfectly possible for two different logon protocols to make use of the same authentication protocol.

We have just seen the benefits of defining collaborations in terms of other smaller collaborations. However, when looking at Fig. 6.1 we can guess how UserLogon works, but we do not exactly know how it does it. Even if we know how each of the four small sub-collaborations works in isolation, we do not know how they work together. Does *rr1* happen before *lo* or afterwards? Does *lo* finish before *ua* starts or do they overlap? Can *lo* succeed with independence of what happens to *ua* or does it depend on its result? These are questions that we have to answer if we really aim at composing collaborations, and we can do it by explicitly describing the dependencies between collaborations, that is, their inter-relationships.

Sanders [SB04] has proposed associating goals with services considered as collaborations as a means to express liveness properties. *Event goals* (e\_goals) are desired events, while *state goals* (s\_goals) are properties of collaboration global states that we wish to reach and entail combinations of role goals. Sanders found that service goals may also be used to express the dependencies that exist between collaborations. These goals become then synchronization points between collaborations. For example, we may say that when *rr1*'s goal is achieved, *lo* is enabled, that is, it can happen. Following this approach the problem of showing the dependencies between collaborations turns into the problem of showing the dependencies between their goals, but we still miss a good solution to show such dependencies. Sanders, for example, defined the goal of UserLogon (*s\_goal:userLoggedOn*) as a logical AND-operation over the goals of its subordinate collaborations, as depicted in Fig. 6.1. However, while such an expression reveals that UserLogon only succeeds if all its

subordinate collaborations also succeed, it still does not tell us the order in which the collaboration goals are achieved. To overcome that limitation Sanders also experimented with several UML concepts to describe goal dependencies, such as activity diagrams and interaction overview diagrams. These diagrams are good at expressing sequential and parallel relationships, but they do not meet all our needs, since they fail to express finer relationships between intermediate goals, as those existing when two collaborations overlap. For example, in UML activity diagrams activities can be nested, so if we represent collaborations as activities, it would be possible to show (to some extent) that, for example, *lo* only succeeds if *ua* also succeeds, by nesting *ua* inside *lo*. However it does not seem possible to show, for example, that a collaboration starts, after certain time enables a second collaboration, and from then on both run in parallel (see Fig. 6.3c).

Since UML diagrams do not meet all our needs, we have analysed Use Case Maps to see if they offer better support for expressing goal dependencies, since they are well known for their ability to explicitly capture inter-scenario relationships. The result has been promising, since we have been able to successfully describe several types of dependencies. Moreover, we have experimented with the synthesis of state-machines for collaboration roles using the UCM information to guide the process, and the results are again promising. We take a closer look at these two aspects in the next sections.

### 6.3 UCMs for Describing the Goal-based Progress of Collaborations and their Inter-Relationships

Use Case Maps (UCMs) [BC96, Buh98] are a scenario-based graphical notation used to describe causal relationships between responsibilities (e.g. tasks, actions, etc), which may be bound to abstract components. Basically, UCMs order responsibilities along a path and link causes (e.g. preconditions or triggering events) to effects (e.g. post-conditions). With UCMs several scenarios can be easily integrated in a single diagram. This is quite useful for showing interactions between the scenarios and understanding their combined behaviour.

In Sect. 6.2 we argued that the dependencies between collaborations can be expressed in terms of their goals. That is, by relating the event and state goals of different collaborations we can effectively capture their inter-relationships. Our aim is to use UCMs to:

1. describe the goal-based progress of each collaboration (i.e. the causal relationships between its event and state goals) in isolation;
2. integrate the individual UCMs into more elaborated diagrams that show the dependencies between the individual collaborations.

An example is given in Fig. 6.2, where separated UCMs for the RoleRequest, User-Authenticate and Logon collaborations are shown on the upper-side, and their in-

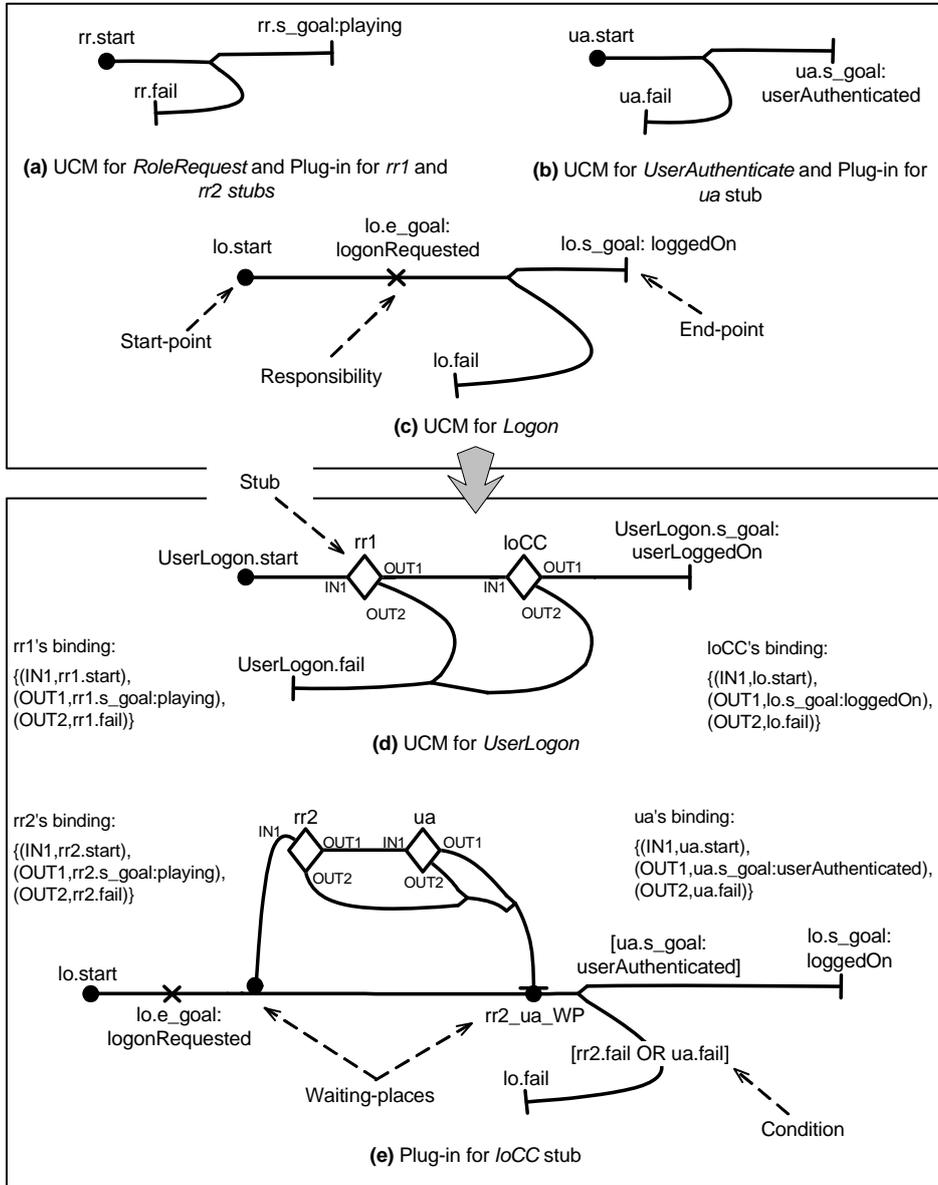


Figure 6.2: UCMs for the UserLogon Service

tegration into more complex UCMs for the UserLogon collaboration is shown on lower-side. We will explain the dependencies expressed by these UCMs in Sect. 6.3.2, but before we will briefly explain the basic UCM elements and how we use them.

### 6.3.1 Basic UCM Notation

It is not the scope of this paper to explain how UCMs work, so we will just briefly explain the UCM notational elements needed to understand the figures and concepts presented here. Those notational elements are highlighted in Fig. 6.2. For a more detailed explanation of them, and of UCMs in general, please refer to [BC96, Buh98].

In the UCM notation *paths* (depicted as lines) represent scenario flows, so we have used them to represent the lifeline of collaborations. They connect *start-points* with *responsibilities* and *end-points*. A start-point (labeled with «*collaboration\_name.start*») is a pre-condition or triggering cause that symbolizes the beginning of a collaboration, while an end-point is a post-condition representing one of its possible outcomes in terms of achievement or not of its goal(s). End-points representing achievement of state goals are labeled with «*collaboration\_name.s\_goal:goal\_name*», while those representing failure are labeled with «*collaboration\_name.fail*». Fig. 6.2a exemplifies the use of these notational elements. The figure shows a simple collaboration that, after starting, it can reach any of two final states: one representing the achievement of its goal (labeled *rr.s\_goal:playing*); the representing failure on achieving its goal (labeled *rr.fail*).

Responsibilities are intended to represent generic tasks or actions. We use them, however, to represent event goals, that is, to show that collaboration achieve some progress (see Fig. 6.2c). Therefore, we interpret responsibilities as “tasks to achieve some progress”. We understand that this use of responsibilities is not completely rigorous, but we think it is acceptable<sup>1</sup>.

*Static stubs* can be used to better structure a large diagram. They are containers for sub-maps (called plug-ins) and, in our approach, represent collaboration uses. This is an elegant way of representing the composition of a collaboration from other subordinate collaborations. An example of the use of stubs can be seen in Fig. 6.2d, where the UserLogon collaboration is composed of two other collaborations, namely *rr1* and *loCC*. The plug-ins for these collaborations are shown in Figs. 6.2a and 6.2e respectively. Fig. 6.2d (at the sides) also shows how the bindings between the inputs and outputs of a stub and the start- and end-points of its plug-in are defined.

*Waiting places* are points where the path waits for an event to happen (e.g. an arrival along a tangentially connected path or a connected end-point). They constitute points where interactions with the environment or other paths can happen, so they can be used to couple collaborations and so express causal dependencies between them. As guard conditions for waiting places we use logical expressions in terms of event and/or state goals of the triggering collaboration. The use of waiting places is illustrated in Fig. 6.2e, where two waiting places are shown. The first one (to the left) is activated when there is an arrival on the *lo*'s path, that is, after *lo* achieves *logonRequested* progress. When this happens, the collaboration represented by the *rr2* stub is enabled. The second waiting place (to the right) is used to make the *lo* collaboration wait for the outcome of *ua* and/or *rr2*.

*AND-/OR- forks* and *joins* can be used to, respectively, split and merge paths.

---

<sup>1</sup>If this interpretation was not acceptable we may use *open waiting places* (see [BC96]) instead of responsibilities

### 6.3.2 Dependency Patterns

A side-effect of decomposing the interactions between components into small collaborations is, as we have already pointed out, that the dependencies between the resulting collaborations must be explicitly captured. The majority of these dependencies can be classified as: sequential dependencies, if they impose a temporal ordering between collaborations; or as goal dependencies, if the goal of a collaboration depend on the goal(s) of other collaboration(s).

We can express collaboration dependencies using UCMs. To do it, we have to couple the UCMs that represent each individual collaboration according to the patterns that we present below.

#### Sequential Dependencies.

Sequential dependencies impose a temporal ordering between collaborations. If a collaboration  $c2$  depends sequentially on other collaboration  $c1$ , we say that  $c1$  enables  $c2$ . Two UCM patterns can be used to express sequential dependencies between collaborations. The selection of the appropriate pattern is made according to the nature of the condition that enables the dependent collaboration.

If the collaboration  $c2$  is enabled when  $c1$  achieves (or not) its goal, the appropriate end-point of  $c1$  is connected with the start-point of  $c2$  (see Fig. 6.3a). In situations where stubs are used to represent collaborations, the appropriate output of the “enabling” stub is interconnected with the input of the “enabled” stub (see Figs. 6.3b and 6.2d).

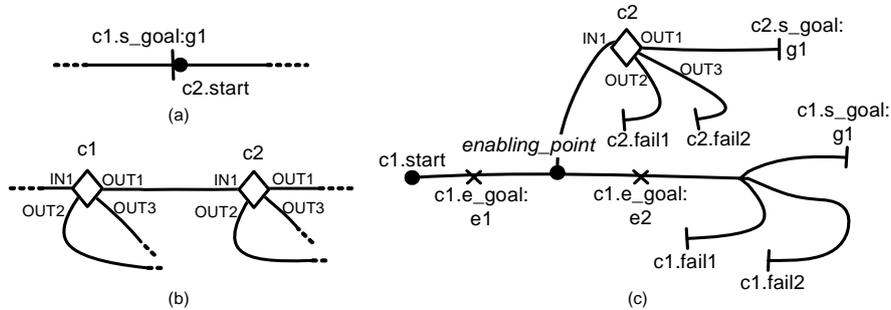


Figure 6.3: Sequential Dependencies

If the collaboration  $c2$  is enabled when  $c1$  achieves some progress (i.e. reaches an event goal), the start-point of  $c2$  is tangentially connected to the path of  $c1$ , just after the responsibility representing the event goal that enables  $c2$ . Note that if a stub is used to represent  $c2$ , its start-point is not directly connected to the path of  $c1$ . However, this connection happens indirectly through an auxiliary path with its start-point connected tangentially to  $c1$ 's path, and its end-point merged with the input of the stub. This is shown in Fig. 6.3c, where  $c2$ , represented by a stub, is

enabled when  $c1$  achieves event goal  $e1$  (setting its value to *true*). Note also that in order for the interconnection to be effective, the *enabling\_point* waiting place<sup>2</sup> must have its guard condition set to  $c1.e\_goal:e1 == true$ . After  $c2$  is enabled, both collaborations,  $c1$  and  $c2$ , run concurrently.

### Goal Dependencies.

A goal dependency exists when a collaboration depends on the success of other collaboration(s) in order to achieve its own goal(s). This dependency can be either *total* or *partial*. When a collaboration  $C$  has no own behaviour, but its behaviour has been completely specified by reusing other collaborations, we talk about total goal dependency. In this case,  $C$ 's goal is completely specified in terms of the other collaborations' goals (e.g.  $C.goal = c1.goal \wedge c2.goal$ ). However, if the achievement of  $C$ 's goal not only depends on the achievement of other collaborations' goals, but also on the progress achieved by  $C$  itself, we talk about partial goal dependency.

We can show a total goal dependency using a UCM for the main collaboration that does not include any responsibility, and in which the subordinate collaborations are stubs. This is illustrated in Fig. 6.2d, where the UserLogon collaboration is composed of two other collaborations, namely *rr1* (see Fig. 6.2a) and *loCC* (see Fig. 6.2e). The interpretation of UserLogon's UCM is as follows. When UserLogon starts, *rr1* is automatically enabled and runs to completion. If it fails to achieve its goal, so does UserLogon. But if *rr1* succeeds, *loCC* is enabled, which also runs to completion. In the same way as before, if *loCC* fails, so does UserLogon, but if it succeeds, UserLogon achieves its goal. Therefore, this UCM tells us both the execution order of the collaborations and the goal dependency that UserLogon maintains with *rr1* and *loCC*.

According to Fig. 6.2d,  $UserLogon.fail = rr1.fail \vee lo.fail$ . We may have given a different meaning to *UserLogon.fail* (or even have defined several types of failure) just by connecting the outputs of the stubs in a different manner, with help of OR-joins and AND-joins.

Note also that UserLogon is a composition of *rr1* and *loCC*, where *loCC* is in turn a composition of *rr2* and *ua*. This exemplifies how collaborations can be nested in several levels.

A partial goal dependency can be illustrated applying the patterns depicted in Fig. 6.4 or Fig. 6.5. The key aspect behind both patterns is the interconnection of the collaboration UCMs by means of waiting places.

Fig. 6.4 shows a case in which a subordinate collaboration  $c2$  is enabled when the main collaboration  $c1$  achieves certain progress. Then  $c1$  waits for  $c2$  to run to completion and, depending on  $c2$ 's goal outcome,  $c1$  either succeeds itself or not. This is the same type of dependency expressed in Fig. 6.2e, where the *lo* collaboration is partially goal dependent on *rr2* and *ua* collaborations.

There are four aspects that deserve explanation in this pattern:

---

<sup>2</sup>*enabling\_point* is actually a start-point, but start-points are special waiting places for a stimulus to start a path

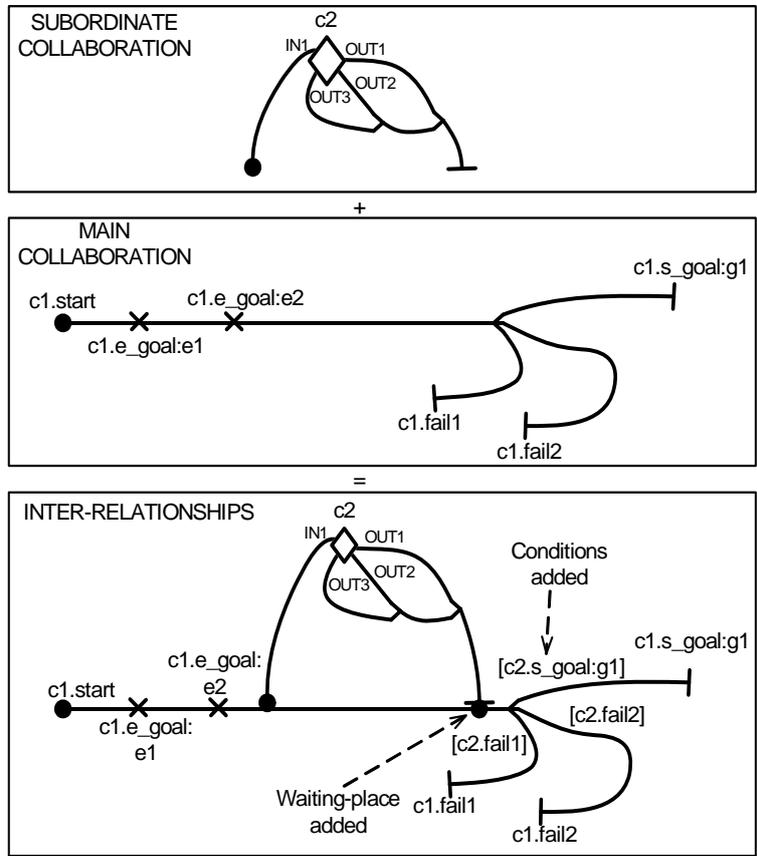


Figure 6.4: Partial Goal Dependency (I)

1. There is no interaction between *c1* and *c2* other than the one at the beginning and at the end of *c2*. That is, the subordinate collaboration, once enabled, runs without interruption. It is therefore that we use a stub to represent it.
2. The subordinate collaboration is actually sequentially dependent on the main collaboration. To express that its start-point is tangentially connected to the path of the main collaboration, just after the responsibility that represents the enabling event goal.
3. A waiting place is added to the main collaboration's path at the point where the end-point of the subordinate collaboration must be connected. The main collaboration waits there for the subordinate one to finish.
4. Conditions expressed in terms of the subordinate collaboration success or failure are added to the OR-joins of the main collaboration. By doing this we join pre-conditions (related to the subordinate collaboration) with post-conditions (related to the main collaboration).

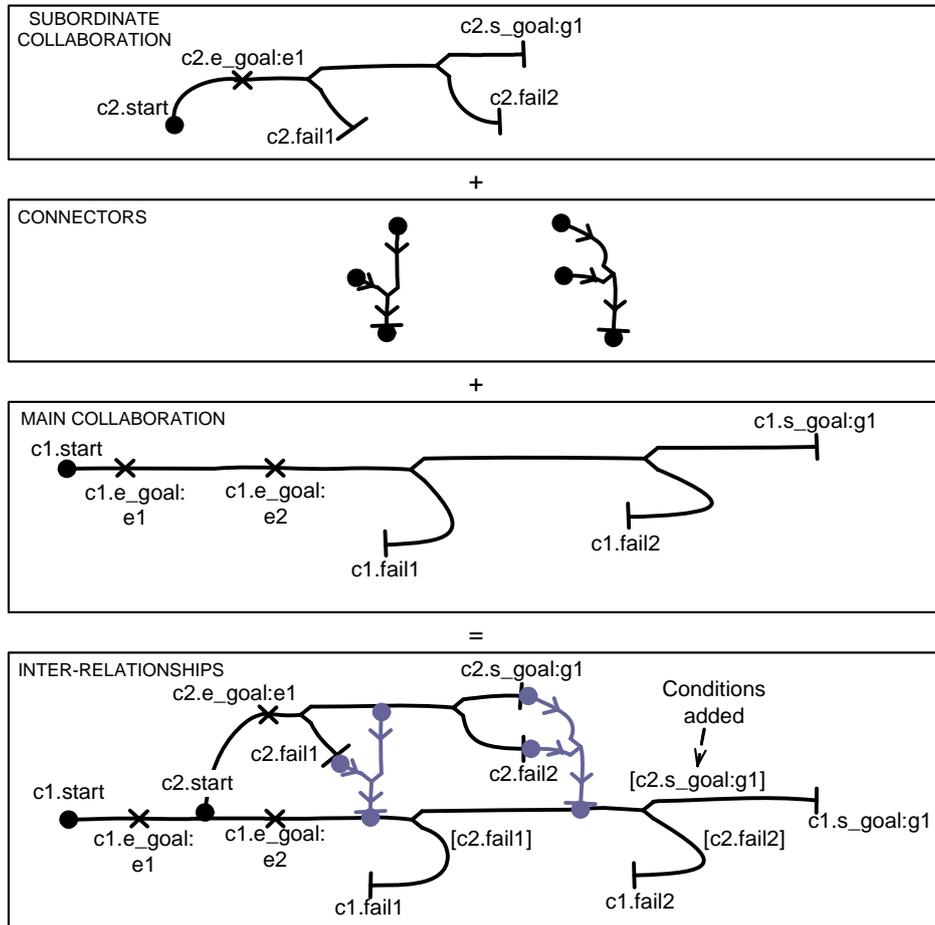


Figure 6.5: Partial Goal Dependency (II)

The collaboration composition presented in Fig. 6.5 is slightly more complicated. Here the collaborations do not only interact at the beginning and at the end of the subordinate collaboration, but also at intermediate points of the latter collaboration's lifetime. The only difference thus with the previous case is that we have to interconnect the two collaborations at those intermediate points. This is done by using the connectors illustrated in Fig. 6.5. They are paths with one or more start-points, one end-point, and one waiting place connected to the end-point. The start-points are connected to the enabling collaboration and the waiting-place is inserted into the path of the enabled collaboration. Note that both the main and the subordinate collaborations can adopt, at different times, the roles of enabling and enabled collaborations, depending on the concrete interactions that take place between them.

## 6.4 Towards Automatic Synthesis of State-Machines

The proposed service engineering process ends up with the translation from the collaboration-oriented view, where a service is described as a collaboration, into the object-oriented view, where the total behaviour of the service objects participating in the service provision is described as state-machines. This translation process basically consists on building the state-machines of the collaboration roles and binding them to instances of objects. This could be a trivial step if a collaboration was not decomposed into smaller sub-collaborations. However, when decomposition is used, as it is the case for the UserLogon collaboration presented in Fig. 6.1, the process is not so trivial. In the figure we see that an object playing the TerminalClientSession role will indeed play four sub-roles<sup>3</sup>: *rr1*, *lo*, *rr2* and *ua*; each one in a different sub-collaboration. Therefore, we have to compose the state-machines of those four sub-roles in order to synthesize the behaviour of TerminalClientSession. We need to know then the order in which the roles are played, and if their executions overlap or not. This information can be extracted from the UCM describing the UserLogon collaboration (see Fig. 6.2d).

The synthesis process we present here allows to mechanically generate the aforementioned state-machines. In Sect. 6.2 we mentioned that, in our approach, the behaviour of each collaboration is described with sequence diagrams. These diagrams are taken as input for the synthesis process, as well as the UCM representing the service collaboration, which shows the dependencies between its sub-collaborations. In the following we will refer to this UCM as “the composite UCM”.

For each collaboration role, the process for synthesizing the state-machine of an object playing that role consists of four steps. These are explained below, and exemplified with the synthesis of a fraction of the state-machine of an object playing the TerminalClientSession role:

1. Determine the sub-collaborations<sup>4</sup> the object participates in. This is necessary because the composite UCM may contain information about other collaborations not relevant for this object, which should be ignored.  
Store the collaboration names, together with the name of the role the object plays in each collaboration, in a table, which we will refer to as the *Role Table*. Table 1 is the Role Table for TerminalClientSession.
2. For each collaboration and role in Table 1, project its associated sequence diagram into the lifeline of the role. This is done to obtain, for each role, an automaton (still with goal information) describing its behaviour in the collaboration. Note that this automaton may be stored in the collaboration, to be reused in the future. This process is shown in Fig. 6.6 for some of the sub-roles of TerminalClientSession.

<sup>3</sup>The UML standard does not use the word *sub-role* when talking about collaboration use roles that are bound to collaboration roles. However the informal interpretation is that of roles of a role, or just sub-roles as we like to call them

<sup>4</sup>For the sake of simplicity, the prefix *sub* will be omitted in the following, but the reader should be aware that when we say “collaboration” we really mean “sub-collaboration”

Table 6.1: Role Table for the TerminalClientSession

Collaboration	Collaboration Role
<i>rr1</i>	<i>invoked</i>
<i>lo</i>	<i>loAgent</i>
<i>rr2</i>	<i>requestor</i>
<i>ua</i>	<i>uaTerm</i>

3. Use the composite UCM (see Fig. 6.2) to guide the composition of the automata generated in step 3 into a state-machine. The UCM is traversed and the automata (as a whole or in parts) are added to the final state-machine attending to the events we find in the UCM's paths. This is done according to the algorithm described in the Appendix.
4. As a final step, suppress any state existing between consecutive input and output transitions.

It should be noted that the synthesized state-machine is not complete, because it does not include internal actions, which have to be added at a later stage by the designer. We plan to look at how this can be done in future work.

#### 6.4.1 An Example

In this section we illustrate how the state-machine of an object playing the Terminal-ClientSession role can be intuitively synthesized from the joint information provided by the composite UCM for UserLogon and the automata for the *invoked*, *requestor*, *uaTerm* and *loAgent* roles.

Looking at Fig. 6.2d, we see that just after UserLogon starts, the *rr1* stub is found. Its plug-in (see Fig. 6.2a) indicates that the *rr1* collaboration starts. Since TCS participates in *rr1*, playing the *invoked* role, we study the details of the plug-in. It indicates that *rr1* starts and runs to completion without interruptions, so we add the whole automaton for the *invoked* role to the TCS state-machine (see Fig. 6.7, step 1). After the *rr1* stub, we find the *loCC* stub. When we look at its plug-in (see Fig. 6.2e) we see that *lo* starts. TCS also participates in this collaboration, where it plays the *loAgent* role. We therefore look at the details of the plug-in and see that a responsibility corresponding to *logonRequested* event goal is reached. As a result, we take, from the *loAgent* automaton, the transitions and states placed between the start symbol and the transition marked with *e\_goal:logonRequested* (inclusive) and add them to the TCS state-machine (see Fig. 6.7, step 2). Following the UCM we see that a new path containing two stubs, namely *rr2* and *ua*, is triggered, while *lo* waits at the *rr2\_uaWP* waiting place. These two stubs represent two collaborations which TCS participates in, playing the *requestor* and *uaTerm* roles in them, respectively. Therefore, we add the whole automaton for both the *requestor* and the *uaTerm* roles to the TCS state-machine. Note, that the *requestor*'s automaton is added after the transition marked with *e\_goal:logonRequested* (see Fig. 6.7, step 3), since the UCM's interpretation is that *e\_goal:logonRequested* enables

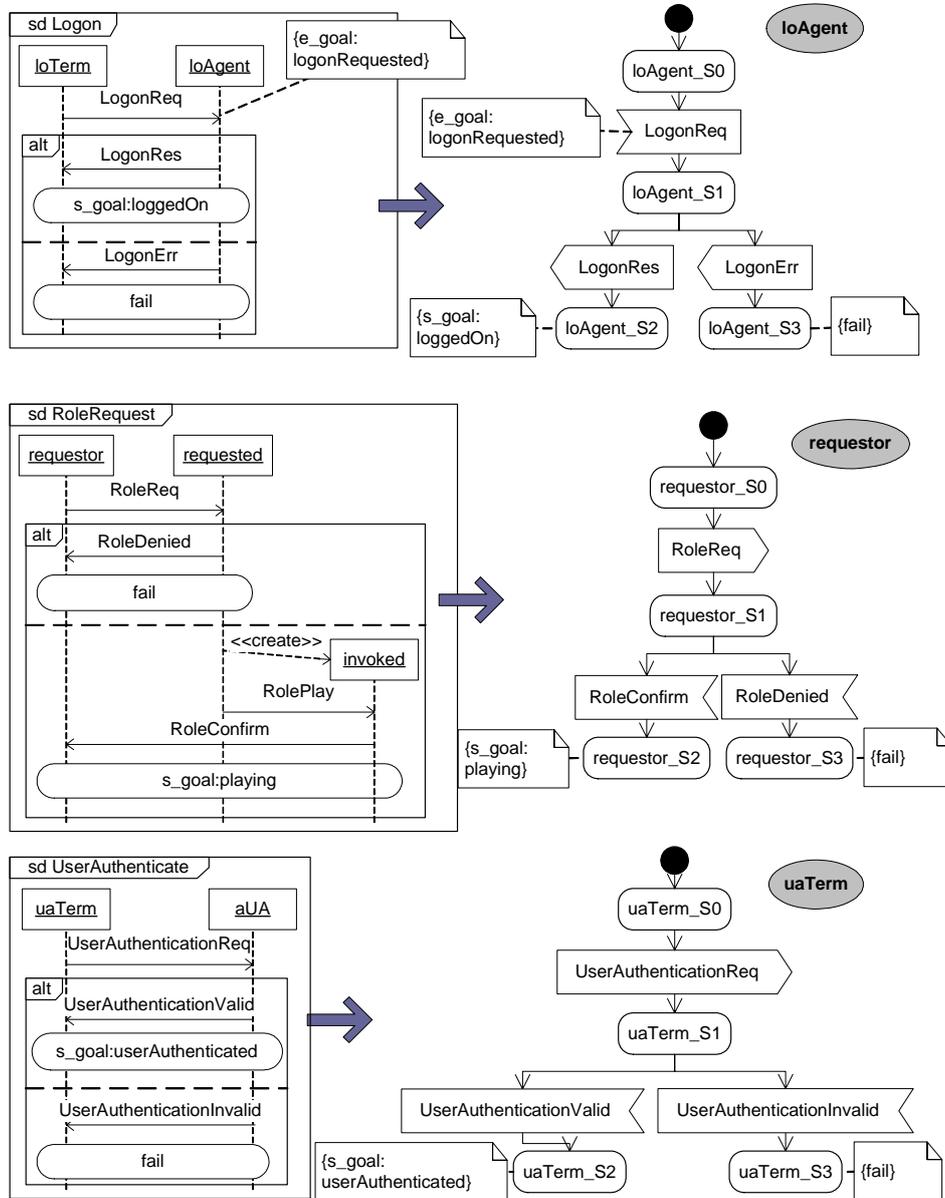


Figure 6.6: Projection of Interactions into Collaboration Roles

*rr2* and thus the *requestor* role. The *uaTerm*'s automaton is added, in turn, after the state marked with *s\_goal:playing* (see Fig. 6.7, step 4), since the UCM tells us that *rr2.s\_goal:playing* enables *ua*. Following the UCM we arrive to the *rr2.uaWP* waiting place where *lo* was waiting. That means that *lo* is enabled again. The

UCM indicates that if *ua.s\_goal:userAuthenticated* was achieved, *lo* achieves its own goal. Thus we take, from the *loAgent* automaton, the transitions and states placed between the transition marked with *e\_goal:logonRequested* (i.e. the point where we stopped last time) and the state marked with *s\_goal:loggedOn* (inclusive) and add them to the TCS state-machine. The addition is performed at the state marked with *ua.s\_goal:userAuthenticated* in the TCS state-machine (see Fig. 6.7, step 5). In much the same way, we take, from the *loAgent* automaton, the transitions and states placed between the transition marked with *e\_goal:logonRequested* and the state marked with *fail* (inclusive) and add them to the TCS state-machine, at the states marked with *rr2.fail* or *ua.fail* (see Fig. 6.7, step 6).

The synthesis of the state-machine for TCS is now finished. However the resulting state-machine is not totally correct. As a final step we need to suppress any state existing between consecutive input and output transitions. This is also shown in Fig. 6.7.

## 6.5 Related Work and Discussion

The idea of synthesizing state-machines/state-charts from scenario models is not new, as demonstrated by the number of existing publications in this area. Quite a few papers have been published proposing automatic synthesis approaches that make use of extra information to guide the synthesis process, for example [LMR98, MZ99, WS00, KGSB99, UKM03]. Our approach is however not currently automated, but there is nothing that prevents its automatization.

Leue et al. [LMR98] use HMSCs to explicitly compose a set of MSCs from which ROOM statecharts are synthesized. Mansurov and Zhukov[MZ99] also use HMSCs in their synthesis of SDL state-machines. HMSCs abstract away the details of MSCs and give a high-level view of the relation between scenarios. The disadvantage, however, of using HMSCs (and their UML counterparts Interaction Overview Diagrams) is their lack of support for describing composition of overlapping scenarios, like for example those described by the Logon (*lo*) and UserAuthenticate (*ua*) collaborations (see Figs. 6.1 and 6.6). To express the composition of these two collaborations with a HMSC we should split the sequence diagram associated with the Logon collaboration in two diagrams. By using UCMs to describe the goal-oriented progress of collaborations we also abstract away the details of sequence diagrams, while we are able to describe the composition of overlapping collaborations.

Krüger et al. [KGSB99] adopt a different approach for the synthesis of statecharts from a set of MSCs. Instead of explicitly describing the composition of MSCs, state information is included in them, so different MSCs are related on the basis of similar states. This can be compared to our use of state and event goals, which we include in the sequence diagrams associated with the collaborations (see Fig. 6.6) to help during the synthesis process. However, in our approach, the state and event goals are not shared between sequence diagrams belonging to different collaborations, as it would be required in order to apply the Krüger et al.'s approach. In contrast, we relate the goals of different collaboration by means of UCMs. Our approach promotes, thus, reuse and separation of concerns between scenarios, at the time that makes explicit their inter-relationships.

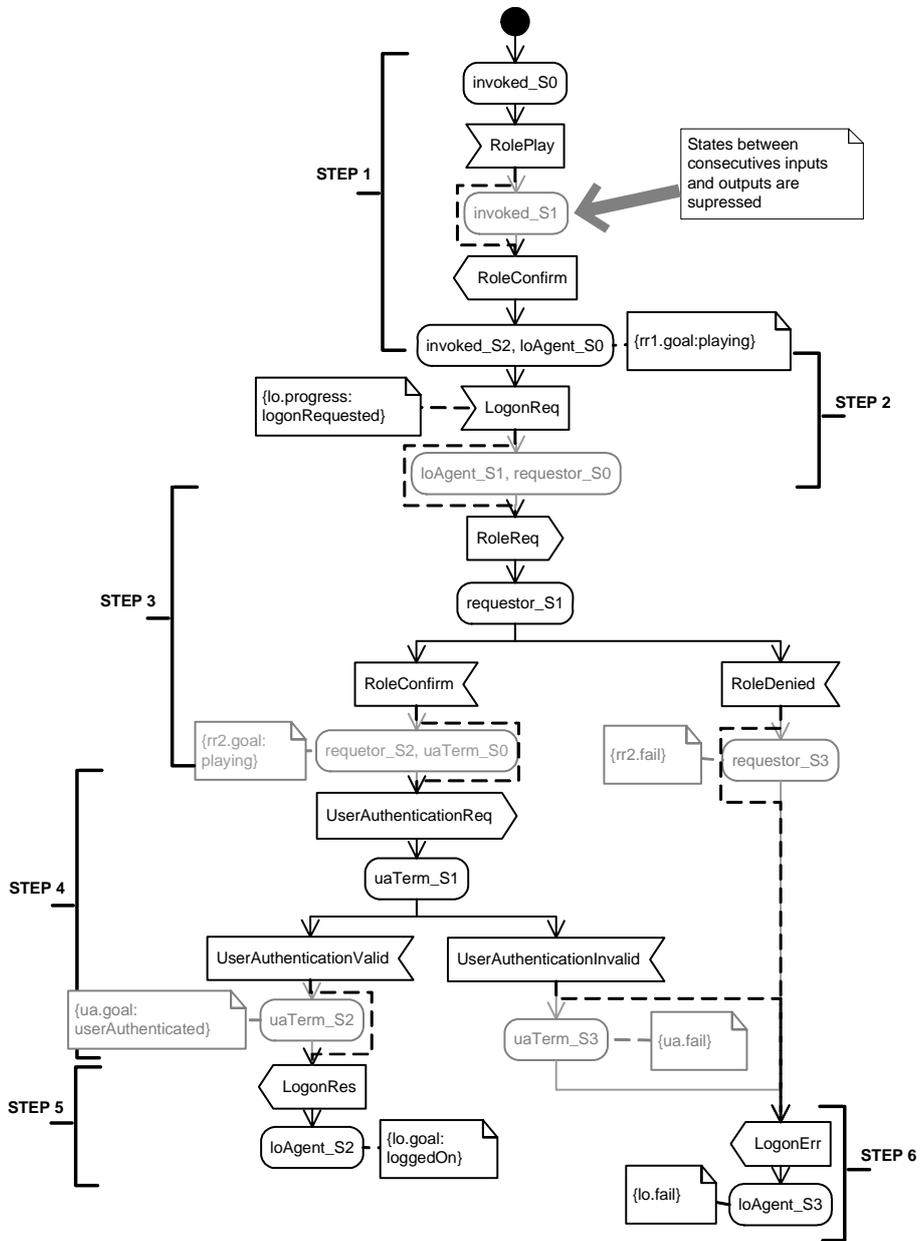


Figure 6.7: Synthesized State-machine for TerminalClientSession

The approach by Whittle and Schumann [WS00] also advocates for including extra information in the scenarios in order to relate them. Pre- and post-conditions, expressed in OCL, are used to give semantic to the messages of UML sequence di-

agrams, from which UML state-charts are generated. The proposed synthesis algorithm does not, however, support overlapping scenarios. This is the main drawback of this approach. Other disadvantage is its low-level of abstraction, since constraints are specified on a per-message basis. Its scalability could also be argued, since its application to large systems with many scenarios and interactions will probably be a tedious work. On the contrary, with UCMs it is easier to inter-relate the scenarios (i.e. collaborations) of large systems in a structured way.

Uchitel et al. [UKM03] present an MSC language with semantics based on scenario composition, state identification and label transition systems (LTS). They further present an approach for synthesizing label transition systems (LTS) from a set of scenarios described in their MSC language. This approach, as ours, tries to combine the benefits of approaches using scenario composition, such as [LMR98] and [MZ99], with the benefits of approaches using state identification, such as [KGSB99] and [WS00]. Moreover, the authors show how their approach can be used to support other synthesis approaches and make their assumptions explicit. The drawback of Uchitel et al.'s approach is, however, its lack of support for overlapping scenarios.

A semi-automatic approach for the synthesis of UML state-charts from a set of UML sequence diagrams is given by Mäkinen and Systä in [MS01]. In their approach no extra information is used to guide the synthesis process. UML sequence diagrams are considered to represent example cases that can be treated in any order. If an ambiguity is found during the synthesis, the user is consulted. This approach recognizes the difficulty of precisely defining the dependencies between scenarios, which, by its nature, are incomplete and many times overlapping. Specially interesting in this approach is the ability to discover ambiguities in a set of scenarios. Its drawback, however, is the total absence of extra information to guide the synthesis process, which makes it too dependent on the user. It would be interesting to study how the approach we present here may benefit from the ability to discover ambiguities of Mäkinen and Systä's approach.

The work presented here is not the first one that uses UCMs for the synthesis of state-machines from scenarios. In [Bor99], [Sal01] and [HAW03] UCMs are also used for that purpose. The differences with our approach lies, however, on the concrete use of UCMs that is done. We use UCMs to describe the dependencies between collaborations at a high level of abstraction. In contrast, Sales [Sal01] uses UCMs to describe SDL state-machines, while both Bordeleau [Bor99] and He et al. [HAW03] use UCMS, at an initial stage, to capture the requirements of services. Then UCMs are translated into MSCs, which are finally used to synthesize SDL state-machines. The approach by He et al. [HAW03] is fully automated, thanks partially to the use of the UCMNav tool [Mig98, UCM], which permits to graphically construct UCMs and translate them into MSCs, as well as export the UCMs as XML files. These files could be used in the automatization of our approach.

## 6.6 Conclusions

We have presented a service modeling approach that uses UML2.0 collaborations, sequence diagrams and UCMs in a complementary way. UML collaborations are used

to describe services as a structure of roles collaborating to perform a task or achieve a goal. They help to get a high-level view of services. The low-level details of the collaborations are then given in the form of associated sequence diagrams annotated with goal information. A strong feature of UML collaborations is the possibility to compose them from other smaller sub-collaborations (by using collaboration uses). This allows for a modular approach that promotes reuse and separation of concerns. However, we argue that for such an approach to work, collaboration dependencies must explicitly be described. We use UCMs for this purpose. They are used to describe causal relationships between the event and state goals of isolated collaborations and to effectively relate goals of different collaborations.

Several patterns for the illustration of goal and sequential dependencies between collaborations using UCMs have been proposed. They are not intended to cover all possible cases of dependencies, but just as a starting point in this way.

An experiment has been performed to synthesize the state-machine of a collaboration role from other smaller roles. As input for this process we have used the information provided by the UML collaborations, in the form of sequence diagrams, and the dependency information provided by a UCM. The results have been satisfactory for small services, as the one presented here. However, we need to experiment with other more complex services to really understand the potential of our synthesis approach.

The work presented here is, however, at an early stage of maturity. Further research has to be done in several directions. We are currently working with the improvement and implementation of the synthesis algorithm. We are also investigating rules for choosing appropriate event and state goals, as well as studying the formalization of goals expressions in terms of temporal logic. We would like to study further the classification of dependencies and their illustration with UCMs, so as to make their scope larger. Finally, we would like to extend our approach to the generation of Hierarchical State Machines, as we believe they provide better support for evolving systems.

## Acknowledgements

The author would specially like to thank Rolv Bræk, Frank Kræmer and Richard Sanders for their useful comments on this work.

## References

- [BC96] R. J. A. Buhr and R. S. Casselman. *Use case maps for object-oriented systems*. Prentice-Hall, Inc., 1996.
- [BC01] Francis Bordeleau and Jean-Pierre Corriveau. On the importance of inter-scenario relationships in hierarchical state machine design. In Heinrich Hußmann, editor, *Proc. of the 4th Intl. Conf. on Fundamental Approaches to Software Engineering (FASE'01)*, volume 2029 of *Lecture Notes in Computer Science*, pages 156–170. Springer-Verlag, 2001.

- [Bor99] Francis Bordeleau. *A Systematic and Traceable Progression from Scenario Models to Communicating Hierarchical State Machines*. PhD thesis, Department of Systems and Computer Engineering, Faculty of Engineering, Carleton University, Ottawa, 1999.
- [Buh98] R. J. A. Buhr. Use case maps as architectural entities for complex systems. *IEEE Transactions of Software Engineering*, 24(12):1131–1155, 1998.
- [FK01] Kathi Fisler and Shriram Krishnamurthi. Modular verification of collaboration-based software designs. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 152–163. ACM Press, 2001. Conference Chair-A. Min Tjoa and Conference Chair-Volker Gruhn.
- [HAW03] Yong He, Daniel Amyot, and Alan W. Williams. Synthesizing SDL from use case maps: An experiment. In Rick Reed and Jeanne Reed, editors, *SDL Forum*, volume 2708 of *Lecture Notes in Computer Science*, pages 117–136. Springer, 2003.
- [KGSB99] Ingolf Krüger, Radu Grosu, Peter Scholz, and Manfred Broy. From MSCs to statecharts. In *DIPES '98: Proceedings of the IFIP WG10.3/WG10.5 international workshop on Distributed and parallel embedded systems*, pages 61–71. Kluwer Academic Publishers, 1999.
- [LMR98] S. Leue, L. Mehrmann, and M. Rezai. Synthesizing ROOM models from message sequence chart specifications. Technical report, Dept. of Electrical and Computer Engineering, April 1998.
- [Mig98] Andrew Miga. Application of use case maps to system design with tool support. Master's thesis, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, 1998.
- [MS01] Erkki Mäkinen and Tarja Systä. MAS - an interactive synthesizer to support behavioral modelling in UML. In *Proc. 23rd Intl. Conf. on Software Engineering (ICSE'01)*, pages 15–24. IEEE Computer Society, 2001.
- [MZ99] Nikolai Mansurov and D. Zhukov. Automatic synthesis of SDL models in use case methodology. In *9th Intl. SDL Forum (SDL'99)*, pages 225–240. Elsevier, 1999.
- [OMG04] Object Management Group (OMG). *UML 2.0 Superstructure Spec.*, 2004.
- [RGG01] Frank Rößler, Birgit Geppert, and Reinhard Gotzhein. Collaboration-based design of SDL systems. In Rick Reed and Jeanne Reed, editors, *SDL '01: Proceedings of the 10th International SDL Forum Copenhagen on Meeting UML*, volume 2078 of *Lecture Notes in Computer Science*, pages 72–89. Springer-Verlag, 2001.

- [RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2nd edition, 2004.
- [Sal01] Igor Sales. A bridging methodology for internet protocols standards development. Master's thesis, School of Information Technology and Engineering (S.I.T.E.), Faculty of Engineering, University of Ottawa, Ontario, 2001.
- [SB04] Richard Torbjørn Sanders and Rolv Bræk. Modeling peer-to-peer service goals in UML. In *Proc. of the 2nd Int. Conf. on Soft. Eng. and Formal Methods (SEFM'04)*. IEEE CS, 2004.
- [UCM] Use Case Maps Web Page and UCM User Group. URL: <http://www.usecasemaps.org>.
- [UKM03] Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Synthesis of behavioral models from scenarios. *IEEE Trans. Softw. Eng.*, 29(2):99–115, 2003.
- [WS00] Jon Whittle and Johann Schumann. Generating statechart designs from scenarios. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 314–323. ACM Press, 2000. Chairman-Carlo Ghezzi and Chairman-Mehdi Jazayeri and Chairman-Alexander L. Wolf.

## Appendix 6.A Synthesizing Algorithm

The algorithm presented here has not been tested thoroughly yet, so it may contain some inconsistencies. The algorithm steps should therefore be taken as guidelines, rather than as strict steps.

We present guidelines for a recursive algorithm. First, state-machines for the inner-most stubs are synthesized. These are stubs whose plug-ins do not contain other stubs. The state-machines are synthesized following steps 1 - 9 (see below). Once the state-machines for the inner-most stubs are synthesized, the state-machines for their container UCMs, up to the composite UCM, can be synthesized following steps again steps 1 - 9 (see below). Note that state-machines are only synthesized for those UCMs whose start-point refers to one of the collaborations in the Role Table.

The algorithm uses the following variables:

- *currentRole*: stores the name of the collaboration role we are dealing with.
- *currentUCM*: stores the name of the currently active collaboration
- *currentRoleState[currentRole]*: array that for each collaboration role stores the name of the last automaton's state added to the object's state-machine. Initialized to "start".

- *currentSMState[currentUCM]*: array that for each collaboration/UCM stores the name of the state where other states and transitions can be added. Initialized to “start”.
- *ucmsCurrentPoint[currentUCM]*: array that for each UCM stores the last processed element. Initialized to “start-point”.

And it consists of the following 9 steps:

1. Set *currentRole* to the collaboration role that the object plays in the collaboration that the UCM/stub represents and *currentUCM* to that UCM/stub. Go to step 2.
2. Traverse *currentUCM*'s path, starting at *ucmsCurrentPoint[currentUCM]*, until a responsibility, a waiting place (either belonging to the path or tangentially connected to it), an OR-fork, an end-point or a stub is found. If a responsibility is found go to step 3. If a waiting place is found go to step 4. If an OR-fork is found go to step 6. If an end-point is found go to step 8. And if a stub is found go to step 9.
3. For the *currentRole*'s automaton, take the states and transitions between (but not including) *currentRoleState[currentRole]* and the transition marked with the responsibility's event goal. Add these states and transitions, together with the event goal transition and its succeeding state, to the *currentUCM*'s state-machine at *currentSMState[currentUCM]*. Update *currentRoleState[currentRole]* and *currentSMState[currentUCM]*, and set *ucmsCurrentPoint[currentUCM]* pointing to the just handled responsibility. Go to step 2.
4. If the waiting place is tangentially connected to the path (i.e. other collaboration is enabled), a search for a second waiting place, this time inserted in the current path, is performed. If it is found, a partial goal dependency pattern has been encountered. Go to step 5. If it is not found, or a new tangentially connected waiting place is found, a sequential dependency pattern has been encountered. The enabling collaboration and the enabled one run then concurrently. A composite state with concurrent sub-states (or two orthogonal regions in UML) should preferably be used to represent this behaviour. This treatment is left as further work.
5. If the path between the first and the second waiting place is not empty (i.e. any responsibility, stub or other element is found) both the enabling and the enabled collaborations run concurrently for a while. At the time of writing this paper we have not yet decided the best way of dealing with this situation. This is left as further work.  
Otherwise, if the path between the first and the second waiting place is empty, set *ucmsCurrentPoint[currentUCM]* pointing to the second waiting place and synthesize an automaton for the just enabled collaboration, according to steps

- 1 - 9 (the automaton is not necessarily synthesized for the whole collaboration, but maybe just for a part of the collaboration, which is represented by a fragment of its UCM enclosed between two waiting places). Add the synthesized automaton to the *currentUCM*'s state-machine. To do it, eliminate the start symbol of the automaton and merge each of its succeeding states with a state of *currentUCM*'s state-machine in the following way: if the automaton state is labeled, merge it with a state of *currentUCM*'s state-machine with the same label; if the automaton state is not labeled, merge it with the state pointed by *currentSMState[currentUCM]*. Update *currentSMState[currentUCM]*, so it points to the last added state which is not labeled with any state goal or fail, and go to 2.
6. Set *ucmsCurrentPoint[currentUCM]* pointing to the OR-fork. Check if *currentRoleState[currentRole]* precedes a choice. If so, go to 7. If not, traverse the *currentRole*'s automaton, starting at *currentRoleState[currentRole]*, searching for a choice. If a choice is found, take the *currentRole*'s automaton states and transitions between *currentRoleState[currentRole]* and the state preceding the choice and add them (except the first state) to the *currentUCM*'s state-machine at *currentSMState[currentUCM]*. Update *currentSMState[currentUCM]* and go to 7. If a choice is not found, it means that the OR-fork describes aspects of other collaboration role. Take then all the *currentRole*'s automaton states and transitions from *currentRoleState[currentRole]* and add them to the *currentUCM*'s state-machine at *currentSMState[currentUCM]*. The state-machine for *currentUCM* is finished.
  7. For each of the fork's outgoing paths, synthesize an automaton according to steps 1 - 9. Eliminate the start symbol and label the first state with the guard condition of the path. If there is no guard condition, the state is not labeled. Return to previous active step.
  8. For the *currentRole*'s automaton, take all the states and transitions from (and including) *currentRoleState[currentRole]* to the state marked with the end-point's goal/fail. Add these states and transitions to the *currentUCM*'s state-machine at *currentSMState[currentUCM]*. If there are no more paths in the UCM, the state-machine for *currentUCM* is finished, otherwise return to previous active step.
  9. If the stub does not represent a collaboration in Role Table, bypass it and go to step 2. Otherwise, add the stub's state-machine (without the start state) to the *currentUCM*'s state-machine. If the stub is enabled by an event goal (i.e. a responsibility), the addition is done at the state succeeding the transition marked with the event goal. If the stub is enabled by a state goal (i.e. and end-point), the addition is done at the state marked with the state goal. If the stub is enabled by a start-point labeled with a start label, the addition is done at the *currentUCM*'s state-machine start state. Go to step 2.



---

## Paper 2

### Using UML 2.0 collaborations for compositional service specification.

By Richard Torbjørn Sanders, Humberto Nicolás Castejón, Frank Alexander Kraemer and Rolv Bræk.

Published in the *Proceedings of the ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS'05)*, volume 3713 of *LNCIS*, pages 460-475. Springer, October 2005.

The original publication is available at [www.springerlink.com](http://www.springerlink.com) via [http://dx.doi.org/10.1007/11557432\\_35](http://dx.doi.org/10.1007/11557432_35)

#### Notes

In this paper a service was defined as a collaboration between service roles played by objects that deliver functionality to the end-users. The main emphasis was therefore on the collaborating nature of services. In Section 1.1 we have presented a new definition where the emphasis is on functionality, and the collaboration among service roles is seen as a necessary vehicle to deliver such functionality.



# Using UML 2.0 Collaborations for Compositional Service Specification

Richard Torbjørn Sanders<sup>a</sup>, Humberto Nicolás Castejón<sup>b</sup>, Frank Alexander Kraemer<sup>b</sup> and Rolv Bræk<sup>b</sup>

<sup>a</sup>*SINTEF ICT, N-7465 Trondheim, Norway*

`richard.sanders@sintef.no`

<sup>b</sup>*NTNU, Department of Telematics, N-7491 Trondheim, Norway*

`{humberto.castejon, kraemer, rolv.braek}@item.ntnu.no`

## Abstract

Collaborations and collaboration uses are features new to UML 2.0. They possess many properties that support rapid and compositional service engineering. The notion of collaboration corresponds well with the notion of a service, and it seems promising to use them for service specification. We present an approach where collaborations are used to specify services, and show how collaborations enable high level feature composition by means of collaboration uses. We also show how service goals can be combined with behavior descriptions of collaborations to form what we call semantic interfaces. Semantic interfaces can be used to ensure compatibility when binding roles to classes and when composing systems from components. Various ways to compose collaboration behaviors are outlined and illustrated with telephony services.

## 7.1 Introduction

Service development or service engineering is currently receiving considerable attention and starting to become a discipline in its own right. Driven by the belief that future revenues will have to come from new services, a tremendous effort is being invested in new platforms, methods and tools to enable rapid development and incremental deployment of convergent services, i.e. integrated communication, multimedia and information services delivered transparently over a range of access

and transport networks. The Service Oriented Architecture (SOA) and Service Oriented Computing (SOC), building on web services, are exponents of this trend in the business domain. A general challenge for service engineering, be it business or ICT applications, is to enable services and service components to be rapidly developed, and to be deployed and composed dynamically without undesirable service interactions. This is a challenging problem largely due to fundamental properties of services, i.e.:

- A service is a *partial functionality*. It can be combined with other services to provide the full functionality offered to a user.
- A service execution normally involves several *collaborating* components (i.e. a service is not simply an interface to an object).
- Components can participate in several services, simultaneously or alternately.
- Services are partially dependent on each other, on shared resources and on user preferences.

In order to support model driven service engineering, corresponding modeling concepts are needed. This is where UML 2.0 collaborations come in, since they possess many properties that make them attractive for this purpose.

First of all the concept of UML collaboration corresponds closely with the concept of a service as explained above. We actually define a *service* as a collaboration between *service roles* played by objects that deliver functionality to the end users. Note that this definition is quite general and covers both client-server and peer-to-peer services as described in [BF04].

Secondly, UML collaboration uses provide a means to structure complex collaborations and give an overview not provided by other notations, while at the same time being precise. Collaborations have much the same simplicity and appeal as use cases, and can be used for the much same purposes, but provide additional benefits for service engineering, as will be presented in the following. Service specification using collaborations and collaboration uses fits well with the preferred view of marketers and end-users, while at the same time supporting the difficult engineering tasks of service and system designers.

Thirdly, a collaboration role can be bound to several different classifiers by means of collaboration uses. This provides the desired flexibility to bind service roles to components, the only UML requirement being that the classifier is *compatible* with the type of the role(s) bound to it. A precise definition of compatibility is left as a semantic variation in UML 2.0, but it is clear that this should entail the observable behavior on interfaces of a component.

This leads to a fourth motivation for collaborations – they lend themselves nicely to the definition of so-called *semantic interfaces* [SBvBA05]. As we shall see, a two-party collaboration can define a pair of complementary semantic interfaces. Compared to traditional syntactical interfaces known from web services, CORBA, Java and UML, semantic interfaces also define the visible interface behavior and the goals of the collaboration. This extends the notion of compatibility beyond static

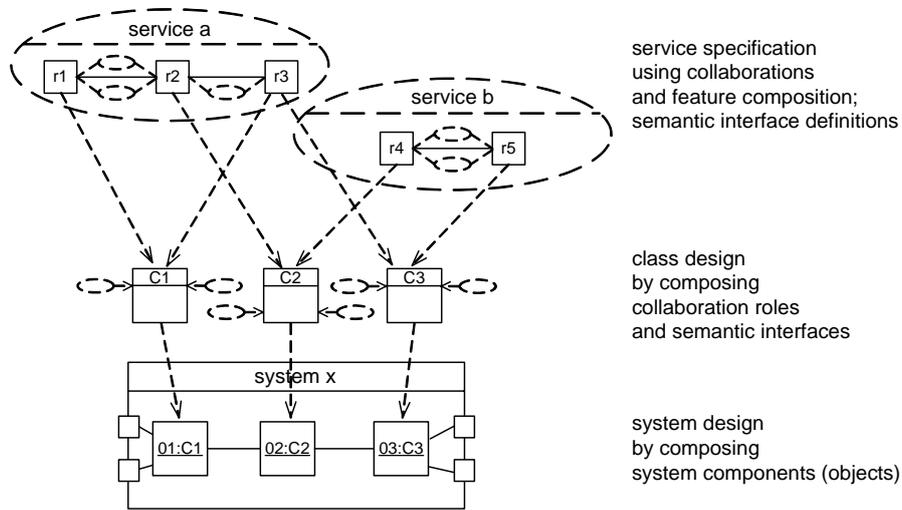


Figure 7.1: Service engineering overview

signature matching to include safety and liveness properties. It also provides an efficient means to perform such compatibility checks at design time and even at runtime.

Finally, it may be argued that the crosscutting view of collaborations is valuable in its own right [RGG01]. It enables us to focus on the joint behavior of objects rather than on each object individually and, not the least, to focus on the purposes and goals of the joint behavior in terms of desirable global states, called service goals in [SB04a]. A service goal can be expressed in OCL, and is a property that identifies essential progress, thus characterizing a desired or successful outcome of a service invocation. It can be argued that service goals are closer to capturing and expressing the user needs than specifying how they are achieved in terms of detailed interactions. Moreover, goal expressions define liveness properties that must be satisfied by compatible components.

Fig. 7.1 provides a principal overview of service engineering using collaborations.

Our service engineering approach is both collaboration-oriented and compositional. It is collaboration-oriented because we model services as collaborations between roles played by distributed components, and it is compositional because we build services from other smaller services. We treat collaborations and collaboration roles as units of reuse.

We consider the following composition cases:

1. Composition of two-party services and semantic interfaces from two-party collaborations.
2. Composition of multi-party services from two-party or n-party collaborations.
3. Class design by composing service roles and semantic interfaces.

Class design is out of the scope of this paper. Here we focus on the use of collaborations for service specification. It is our belief that class design can become a more mechanical process supported by tools if it takes collaborations and semantic interfaces as input. Our experience so far indicates that this is the case [Cas05, Flo03]. However, further work is still needed to confirm this with certainty.

### 7.1.1 Structure of the Paper

In section 7.2 we present how service structures can be described in UML, and how service behavior can be described. We introduce the concept of service goals, and discuss how they can be defined in service structures and in the behavioral descriptions. We introduce what lies in a semantic interface, and discuss compatibility between roles and classifiers.

In section 7.3 we discuss the composition of two-party collaborations used for defining semantic interfaces, as well as composing multi-party services from subordinate collaborations, and indicate directions toward class design. Finally we conclude.

## 7.2 Collaborations, Goals and Semantic Interfaces

### 7.2.1 Collaboration Structure

When used for service specification, the structure of a collaboration identifies the service roles that collaborate to provide the service, as well as their multiplicity and interconnections. Fig. 7.2 depicts a collaboration called `UserCall` specifying the structure of a classical telephone call service. This collaboration diagram tells us that exactly two roles, `A` and `B`, of type `Caller` and `Callee` respectively, are needed to provide a `UserCall` service, and that a communication path between instances playing those roles must exist.

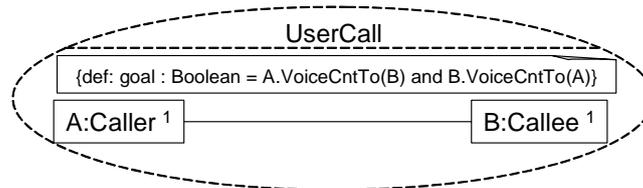


Figure 7.2: The `UserCall` service specified as a collaboration with a goal expression

Specifying a service as a collaboration enables roles to be identified and described without introducing undue bindings to implementation details. Thus a service can be specified and understood as a behavioral component of its own, independent of systems components that implement them.

As we shall see, the behavior of collaborations can be described at several levels of detail. Furthermore, collaborations can themselves be used as components in collaboration compositions, thus becoming units of reuse.

### 7.2.2 Collaboration Goals

The diagram in Fig. 7.2 also shows a goal that should be reached by the `UserCall` collaboration. It is represented by an OCL predicate over properties of the two participating roles. In this case it is a simple logical addition of the role goals of `A` and `B`, to show that `A` has a voice connection to `B` and `B` has a voice connection to `A`:

$$\text{VoiceCnt}(A,B) = A.\text{VoiceCntTo}(B) \text{ and } B.\text{VoiceCntTo}(A)$$

Goal expressions like this can be made very high level, protocol independent and close to the essential purpose of a service as seen from a user point of view. They are actually formal requirements expressions. In this respect they are not new; the novelty lies in the natural binding to the different service specification diagrams, such as collaborations and sequence diagrams. Furthermore, a goal expression represents a liveness property that should hold in actual collaboration uses and therefore constitutes part of the required compatibility of role binding. This illustrates one asset of UML collaborations: they are natural places to express crosscutting properties of services.

### 7.2.3 Collaboration Behavior

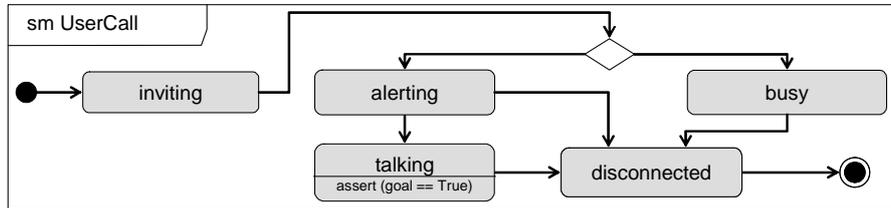
Since UML collaborations inherit from both structured classifiers and behaviored classifiers, they have a large range of expression forms at their disposal. In addition to expressing structural relationships, it is possible to express all forms of behavioral aspects of collaborations, such as interactions, activities and state machines. The UML standard [OMG05] and reference book [RJB04] focus mainly on the structural features of collaborations, and provide few guidelines on how the behavior of a collaboration is described, nor do they explain how collaboration behavior is related to the behavior of its constituent parts, i.e. the roles and role classifiers.

In the following we suggest how the behavior of a collaboration can be described for the purpose of service specification. We first specify the main states a collaboration goes through with a state diagram. This helps to abstract away details and focus on the goal of the collaboration. Thereafter detailed interactions for the collaboration are provided in the form of sequence diagrams.

#### Collaboration States.

The states (or phases) of a collaboration may be described in a state diagram (or activity diagram), as illustrated in Fig. 7.3.

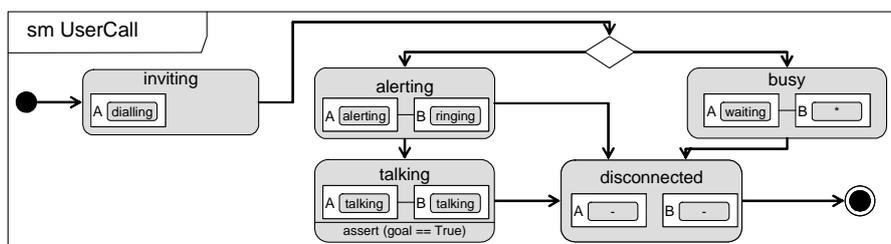
This state diagram describes well known situations in the progress of a basic telephone call. The transitions between the states are represented by arcs, but we have chosen not to define exactly what causes them. For instance the transition

Figure 7.3: State machine diagram for collaboration `UserCall`

from `alerting` to `disconnected` can be due to the `caller` hanging up, the `callee` not answering before a timeout, or the network malfunctioning. Leaving such details undefined can be desirable in a high level service specification.

But what do states of a collaboration mean? Given that a collaboration is not instantiated as an object, no entity is ever in a collaboration state. Rather, a collaboration state is a conceptual state expressing certain situations or conditions on the combined states of the roles `A` and `B` during the collaboration, see Fig. 7.4. It may be considered as a liveness property of the collaboration.

The possibility to focus on the joint behavior and goals rather than the individual role behavior is an important asset of collaborations. The role behaviors must somehow be aligned with each other; we indicate a way of doing so in Fig. 7.4. One must ensure that the role behaviors are dual, i.e. they are fully compatible with respect to safety properties, and that they can reach the joint collaboration states and goals and thereby satisfy liveness properties. A two-party collaboration satisfying these properties defines a pair of semantic interfaces [SBvBA05].

Figure 7.4: State machine diagram for `UserCall` with role states and service goal expression (UML enhancement illustrating role states in collaboration states)

By describing state machines for both the collaboration and the role classifiers, a certain amount of redundancy is added, and the question of compatibility between them arises. This can either be considered as a problem to be avoided, or as a feature that can be put to use. In our view validating consistency between the role behavior and the collaboration behavior is an opportunity that should not be missed.

**Interactions.**

Interaction diagrams are often partial descriptions that are not meant to describe complete behavior, unlike state machine diagrams. For the purpose of service specification interactions for a collaboration should at least focus on the successful cases, i.e. those that lead to the achievement of service goals.

In Fig. 7.5 we have described interactions that lead to the achievement of the service goal of a collaboration called `Invite`. The goal of this collaboration is to bring the collaborating instances to the `talking` state. The goal is indicated by an adornment in the continuation label `talking`.

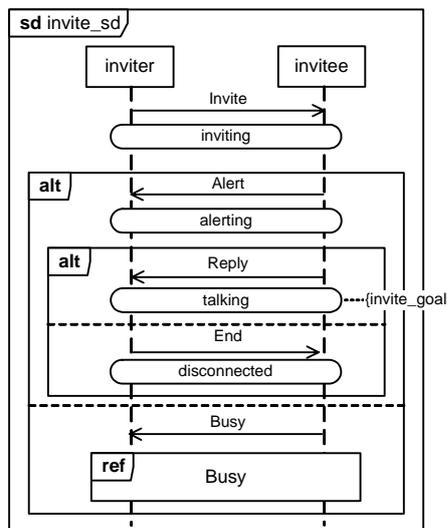


Figure 7.5: Sequence diagram for collaboration `Invite`

**7.2.4 Semantic Interfaces and Compatibility**

In principle, components can participate in any service as long as they can play their part of the service. Therefore, the specification of a service should not bind the service roles to specific classifiers [Bræ99]. In [SB04] we used association classes to specify services, but they fail to meet the requirements for flexible role binding. This is because with associations the binding is determined by the classifiers at the association ends. Collaborations do not have this limitation. With the help of collaboration uses, collaborations roles can be bound to any classifiers that are compatible with the role types. This is shown in Fig. 7.6, where the same classifier, `UserAgent`, is bound to two different roles, `A` and `B`. This is possible as long as the `UserAgent` class is compatible with both collaboration roles. Our interpretation of compatibility is that the `UserAgent` must have visible interface behavior that is goal equivalent with the behavior of both roles, implying that the roles of the collaboration can be achieved.



Figure 7.6: Binding roles to component classes in a collaboration use

This can be put to use by defining a pair of *semantic interfaces* in a two-way collaboration like `UserCall`, as proposed in [SBvBA05]. The semantic interfaces include goal expressions and role behaviors for the two collaboration roles. Such role behavior can be seen as a kind of protocol state machine specifying only the input/output behavior visible on the interface. It can be derived from a general state machine by making a projection of its behavior on the interface in question. In the case of the `UserAgent` in Fig. 7.6, compatibility can be checked in two steps. First we verify that the collaboration goals of `UserCall` are reachable given the roles `A` and `B`. Then we check that the projected behaviors of `UserAgent` on each side of the connection defined by `UserCall` are goal equivalent to the respective behaviors of `A` and `B`. This enables a compositional and scalable validation approach where the most computation intensive work (making projections and comparing behaviors) can be done at design time. When dynamically binding roles to system components at runtime, validation need not be repeated.

The UML standard [OMG05] says that “a collaboration is often defined in terms of roles typed by interfaces”. Unfortunately an interface typing a role can only describe either a *provided* interface, or a *required* interface, but not a combination. This is a limitation. We want role classifiers to describe both the required and the provided interface behavior in a single modeling unit. Typing a role by two interfaces, a required and a provided one, is not legal in the current version of UML, nor would this result in a unified interface description. Similarly, a protocol state machine attached to an interface only constrains the sequence of operation calls to a component, and can not be used to describe a two-way interface.

The limitations of interfaces may be overcome, however, if UML allowed describing interface behavior in terms of state machines that model the (projected) input/output behavior of a component on the interface, such as the Port State Machines (PoSM) proposed by Mencl [Men04]. This is indeed close to the port state machines of ROOM [SGW94], and should be included in UML. Goal compatibility between a component and a port state machine could then be defined in terms of behavior projection.

Given that the behavior of a collaboration role is described in a state machine diagram enriched with service goals, it is relatively straightforward to validate safety and liveness compatibility between a classifier and a semantic interface to which it is bound [Flo03, FB05, SB04], thus ascertaining goal equivalence between objects and roles.

## 7.3 Composition from Collaborations

### 7.3.1 Composition of Two-party Services and Semantic Interfaces from Two-party Collaborations

With collaboration uses we can express how services can be composed from elementary service features, as illustrated in Fig. 7.7.

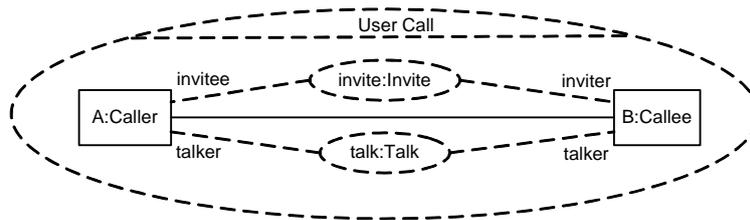


Figure 7.7: `UserCall` composed of elementary features (subordinate collaboration uses)

In Fig. 7.7 the `UserCall` collaboration is decomposed into smaller features, `invite` and `talk`, represented as collaboration uses. These are related to the distinct states of the `UserCall` service (see Fig. 7.3) and to the sequence diagram for `Invite` (see Fig. 7.5). To simplify the example, we have grouped the states for `UserCall` so that the goal of the `invite` collaboration is to bring the `UserCall` collaboration to the state `talking`, upon which the `talk` collaboration use takes over. However, it is not clear from Fig. 7.7 what relationship there is between `invite` and `talk`, that is, if their interactions are interleaved or if they represent a sequence.

It is of central importance to service engineering to make the sequence of goals and the relationships between collaborations explicit. This may be done in several ways. One possibility is showing dependencies between the subordinate collaboration uses and/or their roles in the collaboration diagram itself. Another possibility is to utilize pre- and post-conditions. A third possibility is to use interaction overview diagrams or activity diagrams to express goal sequences, as suggested in Fig. 7.8a below.

Interaction overview diagrams are a form of activity diagram, and thus the token passing semantics of the latter apply. To express goal relationships, the following interpretation of the tokens is employed: a token being passed represents that a goal is achieved, while an input token implies that a subsequent collaboration use (i.e. a service) is enabled. This can be exploited by mechanisms supporting the dynamic discovery of service opportunities [SBvBA05, SB04a]. Note that what happens if the goal is not achieved is not described – the focus is on the achievement of goals. However, if the goal is not achieved in a referenced collaboration, the goal sequence is interrupted.

With this interpretation, Fig. 7.8a specifies that after `invite` has achieved its service goal, the subordinate collaboration use `talk` is enabled. Note that this relationship applies in the context of their use, i.e. in the collaboration `UserCall`.

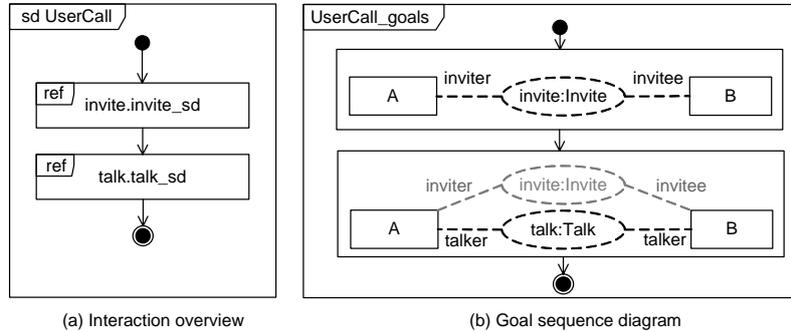


Figure 7.8: Overview of the subordinate collaboration uses of `UserCall`

It is not stated in the specification of the subordinate collaborations `Invite` and `Talk`, which are thus free to be used in other collaboration contexts.

A minor grammatic enhancement to UML, which is to include an illustration of the situation with respect to the involved collaborations (see Fig. 7.8b), seems attractive. This is what we have called a *goal sequence diagram* [SB04]. The second rectangle in Fig. 7.8b illustrates how the roles of `Invite` and `Talk` are bound in the context of `UserCall`. They are statically bound in the `UserCall` collaboration of Fig. 7.7, and simply referred to in Fig. 7.8b. Goal sequence diagrams do not change the semantics of UML, and what is illustrated in Fig. 7.8b corresponds to what is expressed in Fig. 7.8a. Goal sequence diagrams illustrate the evolution of the collaboration structure. For instance, two shades of coloring are employed for the referenced collaboration uses: black color (e.g. for `talk`) illustrates that the collaboration use is active, while grey color (e.g. for `invite`) is for preceding collaboration uses that do not have to exist any longer. For the simple example in Fig. 7.8 the added value of the goal sequence diagram is not striking; Fig. 7.10 is perhaps a more convincing case.

Illustrating situations has been also suggested by Diethelm & al. [DGMZ02]; they use communication diagrams to illustrate use cases and to illustrate do-actions in states.

Two-party collaborations can be composed to form semantic interfaces, which define role behavior and goals of a pair of complementary roles. Limiting such collaborations to a pair of roles is chosen to simplify the validation approach, which is based on validation of object behavior projections and goals over a binary association, as mentioned previously. It also simplifies composition, as components can be composed of composite states that correspond to the semantic interfaces [FB03].

This restriction does not hinder multi-party services to be defined; they can be composed from two-party collaborations with semantic interfaces, as well as from subordinate multi-party collaborations, as shown below. However, this complicates the validation and composition process, as several interfaces have to be validated or composed, and the relationships between the interfaces must be known. Goal sequence diagrams seem to be promising when it comes to composition, as illustrated in the next section.

### 7.3.2 Composition of Multi-party Services

An example that illustrates the potential of composing collaborations from subordinate collaborations is found in Fig. 7.9, where the `UserCall` service with the call transfer feature is described.

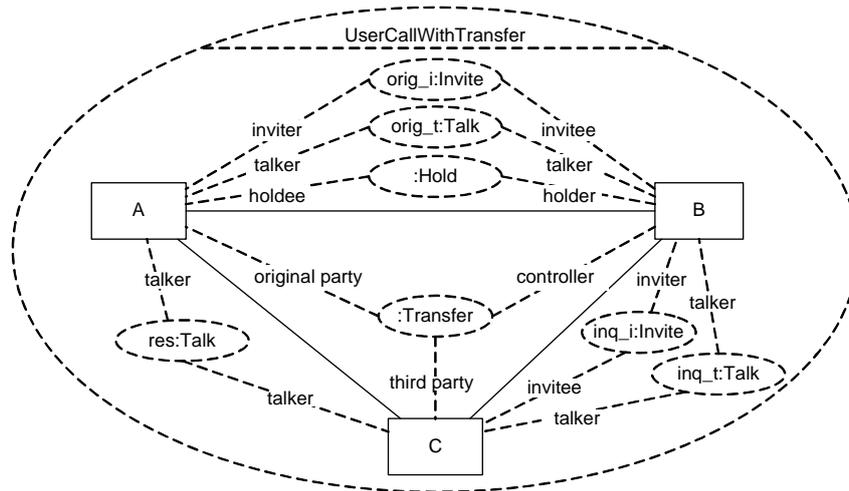


Figure 7.9: The collaboration `UserCallWithTransfer`

Fig. 7.9 demonstrates how subordinate collaborations such as `Invite` and `Talk` may be reused in new settings, due to the flexible role binding of collaboration uses. Such reuse is a very attractive aspect of collaborations, and can help to give an intuitive understanding of a complex situation, as illustrated here. Call transfer is a classical challenge for service designers to understand and describe succinctly. From Fig. 7.9 it is apparent that several call invitations are involved. However, the precise ordering of the subordinate collaboration uses can not be understood from Fig. 7.9 alone. A goal sequence diagram for the `UserCallWithTransfer` service, as suggested in Fig. 7.10a, is one possibility of describing this.

Fig. 7.10a describes the ordering of collaboration uses required for the overall service goal of the transfer feature to be achieved. The goal sequence diagram combined with the collaboration diagram of the service (see Fig. 7.9) provides a compact and fairly intuitive description of a complex service. It has been common practice among telecom service engineers to make informal sketches to the same effect as an aid in service design. UML collaborations provide an opportunity to formalize and better support this practice. The goal sequence demonstrates how UML promotes reuse of units of behavior in the form of collaboration uses, and documents the evolution of the static structure depicted in the collaboration diagram. One particularly interesting aspect of the goal sequence diagram in Fig. 7.10a is that it shows situations in which a role, e.g. B, is simultaneously playing two or more sub-roles, e.g. `holder` and `inviter` in the fourth step of the sequence. Note that the simplicity of collaboration structures may be deceiving. Call transfer may look simple in Fig.

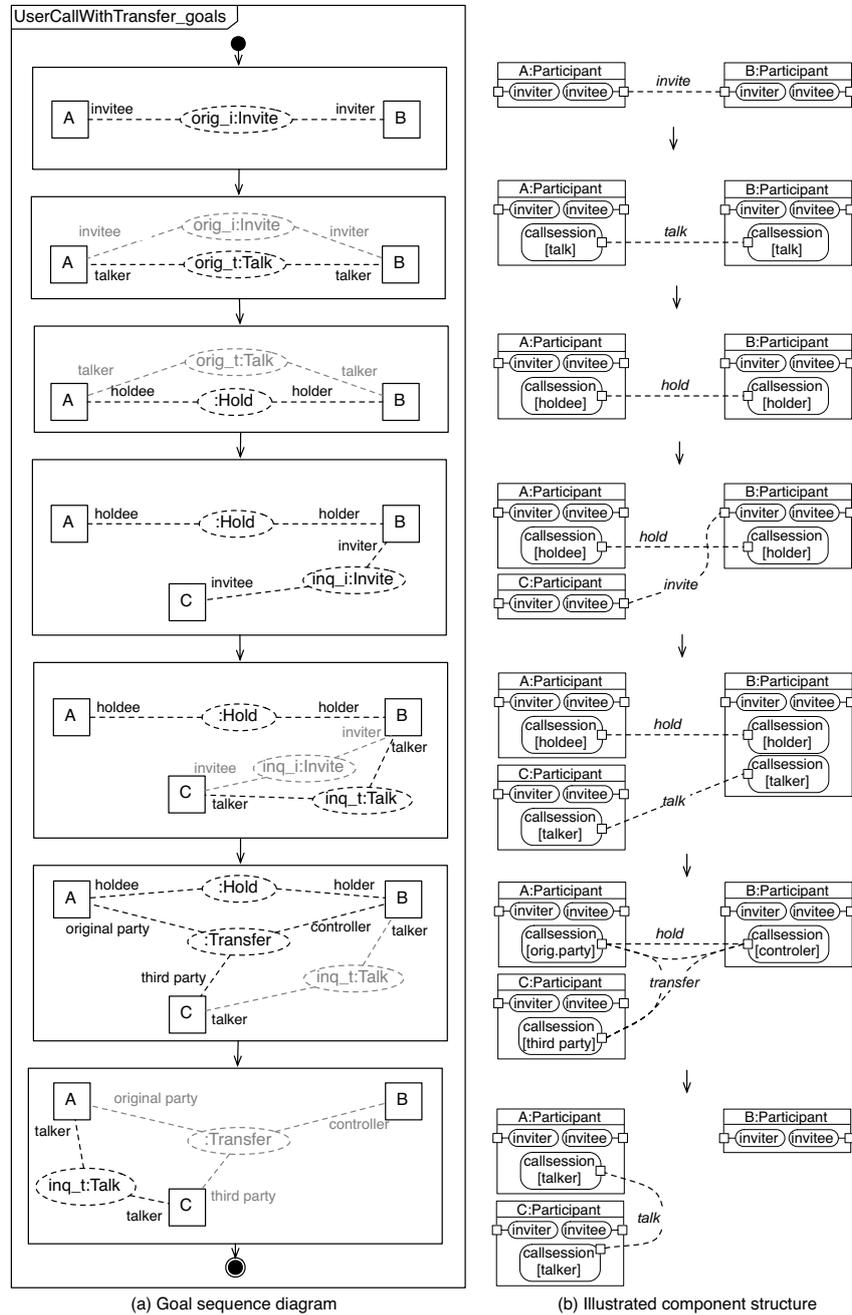


Figure 7.10: Goal sequence for UserCallWithTransfer with related component structure

7.9, but when fully elaborated the underlying sequences and role behaviors can be quite complex.

There are limits to what goal sequence diagrams are capable of expressing. For instance, it is not possible to describe goal dependencies among overlapping collaborations. This is the case, for example, of a **log-on** collaboration that requires a user authentication as part of its operation. It is desirable to model **log-on** and **authenticate** as separate collaborations to achieve reuse, and allow **log-on** to be combined with alternative authentication patterns. However, we cannot express with goal sequence diagrams that **authenticate** is enabled when **log-on** achieves a sub-goal, and that **authenticate** must achieve its goal before further progress in **log-on** is possible. An alternative notation, Use Case Maps [ITU04a], has been shown to have the necessary expressive power [Cas05].

### 7.3.3 Towards Class Design

The specification of service functionality in collaborations is beneficial beyond the specification phase and can have direct influence on the design of classes and state machines. Analyzing the collaborations and the goal sequences tells us which roles a class must play over time, which requests for roles can arrive in which situations and which connections must be established to reach the goals of the implemented services. Modeling service specifications can help class design, as we now shall see.

Fig. 7.10b illustrates the coarse structure of a class **Participant** that implements all three roles **A**, **B** and **C** of **UserCallWithTransfer**. The sub-roles **invitee** and **inviter** are implemented as separate state machines, since call requests can arrive at any time. When a call request from another component is received, **invitee** creates a new instance of the state machine **callsession** to handle the request. The sub-roles **talk**, **hold** and **transfer** can be implemented by composite states inside **callsession**, as these roles are played alternately. The figure also illustrates the connections between the state machines of the components and how they evolve as the service progresses towards the achievement of its goal.

To complete class design one must consider all collaboration roles bound to the class. The **Participant** class, for example, may take part in several collaborations other than **UserCallWithTransfer**, as it is shown in Fig. 7.11. In that case **Participant** must be compatible with the four roles **ua**, **A**, **B** and **ub**, and class design must take this into account.

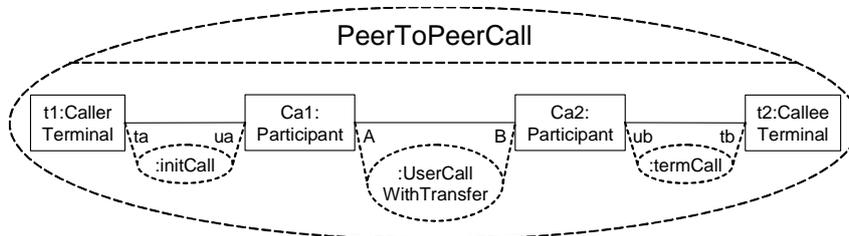


Figure 7.11: Service composed of elementary services

## 7.4 Discussion

### 7.4.1 Related Work

The understanding that services involve collaboration between distributed components is not new; indeed, this was recognized since the early days of telecommunications. In terms of modeling the interaction of collaborations, various dialects of interaction diagrams existed prior to the first standardization of the ITU-T MSC language [ITU04b] in 1994. A slightly different approach was taken in the use cases of OOSE [JCJØ92], where interactions were described textually. However, interactions alone do not really cover the structural aspects of the roles and the flexible binding of roles to classifiers.

Collaborative designs such as protocols have traditionally been specified by state diagrams, using combinations of informal descriptions and formal models, e.g. using SDL [ITU02] or similar ([Int89, Har87, SGW94]). But while state diagrams describe complete object behavior, the overall goals and the joint behavior tend to be blurred.

The concept of role was already introduced in the end of the 70's in the context of data modeling [BD77] and emerged again in the object-oriented literature. Using roles for functional modeling of collaborations was of primary concern in the OORAM methodology [RWL96], and was one of the inputs influencing the UML work on collaborations in OMG. Within teleservice engineering it has been a long-standing convention to describe telephone services using role names like A and B. In [Bræ99] we classified different uses of the role concept, and pointed out that UML 1.x was too restrictive, since a *ClassifierRole* could bind to only one class, so they were not independent concepts that could be re-used in different classes.

Rössler & al. [RGG01] suggested collaboration based design with a tighter integration between interaction and state diagram models, and created a specific language, CoSDL, to define collaborations [RGG02]. CoSDL was aligned to SDL-96. Floch [Flo03] also proposed a notation for collaboration structure diagrams, where components were designed in SDL-2000 [ITU02].

With UML 2.0, it is now possible to model collaborations in a standardized language, increasingly supported by tools. Modeling collaborating services with UML 2.0 collaborations has earlier been suggested by Haugen and Møller-Pedersen [HMP03]. They pointed out that there might be limitations in binding collaboration uses to classifier parts; these issues must be clarified, and binding to parts should preferably be supported. In the FUJABA approach described in [BGHS04], so-called coordination patterns are used for similar purposes as our semantic interfaces. They use a model checker to provide incremental verification based on the coordination patterns.

### 7.4.2 Further Work

A number of issues presented in this article need to be clarified and researched, and experiments in real projects must be undertaken before all problems are solved. We are currently applying these techniques on several practical service engineering cases including access control services, call control, and mobile information services.

Compatibility rules between role classifiers and the objects and classes bound by collaboration uses is a semantic variation point in UML. The research on semantic interfaces [SBvBA05] is a promising starting point for compatibility checking between complementary roles. Additional work on validating compatibility between roles and class designs, with tool support for composition, is being undertaken.

An experimental tool suite is currently being developed as part of the Teleservice Lab at the department of Telematics at NTNU, based on the Eclipse platform. The EU funded project *Semantic Interfaces for Mobile Services*, SIMS, to commence in 2006, will develop tool support for designing and validating collaborations, taking existing prototypes [Als04] as a starting point and validating the approach among industrial users.

## 7.5 Conclusion

This article has suggested ways of exploiting UML 2.0 for service engineering, and has discussed opportunities and limitations that lie in the current standard [OMG05] in that respect. Our conclusion is that UML 2.0 collaborations seem to be a very useful expression form, as it allows one to define pieces of collaborating role behavior that can be bound to role players in a very flexible way.

Useful validation opportunities arise once criteria for role compatibility have been defined. Collaborations can be used to define semantic interfaces, which in turn can be used for compatibility checks and to support composition. We have argued for the inclusion of port state machines in UML as a more general description of semantic interface behavior than the existing protocol state machine mechanisms that have been defined in UML 2.0.

Furthermore we have suggested how minor notational enhancements can be introduced to represent collaboration situations in order to support high level feature composition; this is more of a tool issue than a language issue, but has methodological implications that are important. Finally, we have demonstrated how collaboration uses provide means to define complex multi-party services on a high level.

In contrast to the common practice of modeling complete service sequences involving all participating roles, our approach encourages decomposition into interface behaviors represented as two-way collaborations. The result is smaller and more reusable interface behaviors that can be validated separately, thereby addressing compositionality and scalability. The disadvantage is that behavior composition needs special attention, e.g. using goal sequences as elaborated in [Cas05].

## References

- [Als04] Rune Alsnes. Role validation tool. Master's thesis, NTNU, 2004.
- [BD77] Charles W. Bachman and Manilal Daya. The role concept in data models. In *Proc. of the 3rd Int. Conference on Very Large Data Bases, Tokyo, Japan*. IEEE Computer Society, 1977.

- [BF04] Rolv Bræk and Jacqueline Floch. ICT convergence: Modeling issues. In *Proc. of the 4th Int. SDL and MSC (SAM) Workshop, Ottawa, Canada*. LNCS 3319, Springer, 2004.
- [BGHS04] Sven Burmester, Holger Giese, Martin Hirsch, and Daniela Schilling. Incremental design and formal verification with UML/RT in the FU-JABA real-time tool suite. In *Proc. of the Int. Workshop on Specification and Validation of UML models for Real Time and embedded Systems (SVERTS), associated with UML2004*, Lisbon, Portugal, October 2004.
- [Bræ99] Rolv Bræk. Using roles with types and objects for service development. In *IFIP 5th Int. Conf. on Intelligence in Networks (SMARTNET)*, Pathumthani, Thailand, 1999. Kluwer.
- [Cas05] Humberto Nicolás Castejón. Synthesizing state-machine behaviour from UML collaborations and Use Case Maps. In *Proc. of the 12th Int. SDL Forum, Norway*. LNCS 3530, Springer, June 2005.
- [DGMZ02] Ira Diethelm, Leif Geiger, Thomas Maier, and Albert Zündorf. Turning collaboration diagram strips into storycharts. In *Workshop on Scenarios and state machines: models, algorithms, and tools; ICSE'02, Orlando, Florida, USA*, 2002.
- [FB03] Jacqueline Floch and Rolv Bræk. Using SDL for modeling behavior composition. In *Proc. of the 11th Int. SDL Forum, Stuttgart, Germany*. LNCS 2708, Springer, 2003.
- [FB05] Jacqueline Floch and Rolv Bræk. A compositional approach to service validation. In *Proc. of the 12th Int. SDL Forum, Grimstad, Norway*. LNCS 3530, Springer, 2005.
- [Flo03] Jacqueline Floch. *Towards Plug-and-Play Services: Design and Validation using Roles*. PhD thesis, Dep. of Telematics, Norwegian Univ. Sci. and Tech., Trondheim, Norway, February 2003.
- [Har87] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.
- [HMP03] Øystein Haugen and Birger Møller-Pedersen. The fine arts of service modeling. Technical report, Internal report. ARTS, 2003. <http://www.pats.no/projects/ARTS/arts.html>.
- [Int89] International Organization for Standardization (ISO). *Estelle: a formal description technique based on an extended state transition model*. ISO9074, 1989.
- [ITU02] ITU-T Recommendation Z.100. *Specification and Description Language (SDL)*, 2002.

- [ITU04a] ITU-T Draft Recommendation Z.152. *URN - Use Case Maps notation (UCM)*, 2004.
- [ITU04b] ITU-T Recommendation Z.120. *Message Sequence Charts (MSC)*, 2004.
- [JCJØ92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Øvergaard. *Object-Oriented Software Engineering: A Case Driven Approach*. Addison-Wesley, 1992.
- [Men04] Vladimir Mencl. Specifying component behavior with port state machines. *Electr. Notes Theor. Comput. Sci.*, 101:129–153, 2004.
- [OMG04] Object Management Group. *UML 2.0 Superstructure Specification*, April 2004.
- [RGG01] Frank Rößler, Birgit Geppert, and Reinhard Gotzhein. Collaboration-based design of SDL systems. In *Proc. of the 10th Int. SDL Forum, Copenhagen, Denmark*. LNCS 2078, Springer, 2001.
- [RGG02] Frank Rößler, Birgit Geppert, and Reinhard Gotzhein. CoSDL: An experimental language for collaboration specification. In *Proc. of the 3rd Int. SDL and MSC (SAM) Workshop, Aberystwyth, UK*. LNCS 2599, Springer, 2002.
- [RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2nd edition, 2004.
- [RWL96] Trygve Reenskaug, P. Wold, and O. A. Lehne. *Working with Objects: The OOram Software Engineering Method*. Prentice Hall, 1996.
- [SB04a] Richard Torbjørn Sanders and Rolv Bræk. Discovering service opportunities by evaluating service goals. In *Proc. of the 10th EUNICE and IFIP Workshop on Advances in Fixed and Mobile Networks*, Tampere, Finland, 2004.
- [SB04b] Richard Torbjørn Sanders and Rolv Bræk. Modeling peer-to-peer service goals in UML. In *Proc. of the 2nd Int. Conf. on Soft. Eng. and Formal Methods (SEFM'04)*. IEEE Computer Society, 2004.
- [SBvBA05] Richard Torbjørn Sanders, Rolv Bræk, Gregor von Bochmann, and Daniel Amyot. Service discovery and component reuse with semantic interfaces. In *Proc. of the 12th Int. SDL Forum, Grimstad, Norway*. LNCS 3530, Springer, June 2005.
- [SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.



---

## Paper 3

### A collaboration-based approach to service specification and detection of implied scenarios.

By Humberto Nicolás Castejón and Rolv Bræk.

Published in the *Proceedings of the 5th International Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM'06)*. ACM Press, May 2006.

The original publication is available at *portal.acm.org* via <http://doi.acm.org/10.1145/1138953.1138962>

#### Notes

- In this paper a service was defined as a collaboration between service roles played by objects that deliver functionality to the service users. The main emphasis was therefore on the collaborating nature of services. In Section 1.1 we have presented a new definition where the emphasis is on functionality, and the collaboration among service roles is seen as a necessary vehicle to deliver such functionality.
- An appendix has been included at the end of the paper to discuss some issues discovered after the original publication.

Is not included due to copyright

---

## Paper 4

### **Formalizing collaboration goal sequences for service choreography.**

By Humberto Nicolás Castejón and Rolv Bræk.

Published in the *Proceedings of the 26th IFIP WG 6.1 Intl. Conf. on Formal Methods for Networked and Distributed Systems (FORTE'06)*, volume 4229 of *LNCS*, pages 275-291. Springer, September 2006.

The original publication is available at [www.springerlink.com](http://www.springerlink.com) via [http://dx.doi.org/10.1007/11888116\\_21](http://dx.doi.org/10.1007/11888116_21)



# Formalizing Collaboration Goal Sequences for Service Choreography

Humberto Nicolás Castejón and Rolv Bræk

*NTNU, Department of Telematics, N-7491 Trondheim, Norway*  
{humberto.castejon,rolv.braek}@item.ntnu.no

## Abstract

Methods for service specification should be simple and intuitive. At the same time they should be precise and allow early validation and detection of inconsistencies. UML 2.0 collaborations enable a systematic and structured way to provide overview of distributed services, and decompose cross-cutting service behaviour into features and interfaces by means of collaboration-uses. To fully take advantage of the possibilities thus opened, a way to compose (i.e. choreograph) the joint collaboration behaviour is needed. So-called collaboration goal sequences have been introduced for this purpose. They describe the behavioural composition of collaboration-uses (modeling interface behaviour and features) within a composite collaboration. In this paper we propose a formal semantics for collaboration goal sequences by means of hierarchical coloured Petri-nets (HCPNs). We then show how tools available for HCPNs can be used to automatically analyse goal sequences in order to detect implied scenarios.

## 9.1 Introduction

Many authors have identified the cross-cutting nature of distributed services (e.g. [RGG01, KGM<sup>+</sup>04]) i.e. that services in the general case, involve several collaborating components playing different roles, that each may participate in several services. For service engineering, this implies a need to specify services in terms of their roles and cross-cutting service behaviour, then to specify the detailed behaviour of each service role and, finally, to compose the behaviour of service roles into complete, coordinated and correct component behaviours. UML 2.0 collaborations [OMG05] provide language concepts and mechanisms that partially support this and are therefore very promising from a service engineering point of view. Being both structural and behavioural classifiers in UML 2.0, collaborations can be used to define a service

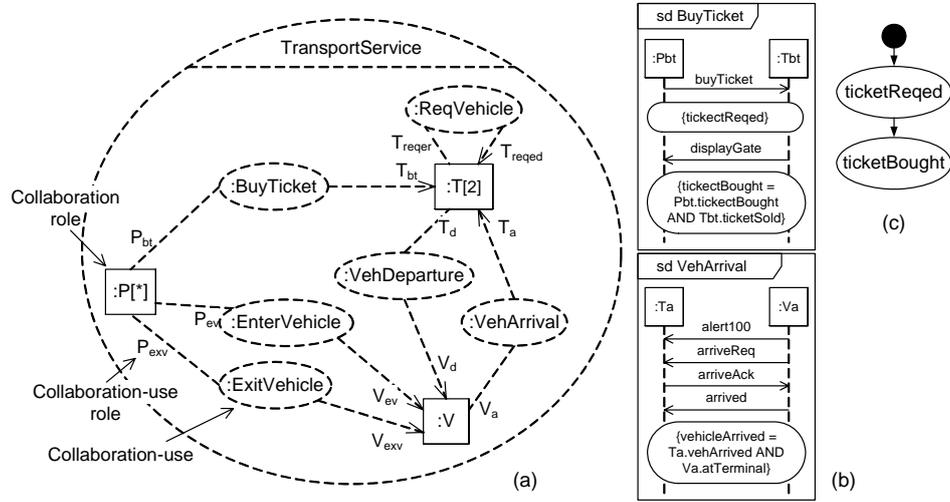


Figure 9.1: (a) Transport service as a UML 2.0 collaboration; (b) Sequence diagrams for BuyTicket and VehArrival sub-collaborations; (c) Service-goal tree for BuyTicket

as a structure of roles with associated cross-cutting behaviour defined using e.g. sequence diagrams. Detailed role behaviour can be defined using e.g. state machines. UML collaborations can be bound to specific contexts (e.g. larger collaborations) by means of collaboration-uses. This feature enables a compositional and incremental specification of services.

As an example consider a simple transport service (inspired by a case study from [UKM04]) in which one vehicle transports one passenger at a time between two terminals. Figure 9.1a depicts this service as a UML 2.0 collaboration. This collaboration identifies three roles, namely  $P$  (Passenger),  $T$  (Terminal) and  $V$  (Vehicle); as well as seven sub-collaborations representing interfaces and features of the service. These sub-collaborations are specified as UML collaboration-uses, whose roles are bound to the *TransportService*'s roles (e.g. *BuyTicket*'s role  $T_{bt}$  is bound to *TransportService*'s role  $T$ ). Bound roles are classified as either *initiating* (i.e. takes the initiative to start the collaboration) or *offered* (i.e. accepts the initiative), indicated by an arrow head with offered roles. For the sake of clarity, in the following we will refer to  $P$ ,  $T$  and  $V$  as service-roles, and to  $T_{bt}$ ,  $T_d$  and the like as sub-roles (of  $T$ ,  $P$  or  $V$ ). The *TransportService*'s sub-collaborations have been identified from the following service requirements. In order to travel, a passenger must buy a ticket at one of the terminals (collaboration-use *BuyTicket*). When this happens, if the vehicle is waiting at the terminal, the departure gate is indicated, and the passenger can enter the vehicle (*EnterVehicle*). The terminal then dispatches the vehicle (*VehDeparture*) and, after arriving at the second terminal (*VehArrival*), the passenger disembarks (*ExitVehicle*). If the vehicle is not at the terminal where the passenger buys the ticket, that terminal requests the vehicle from the other terminal (*ReqVehicle*), which dispatches the vehicle towards the requesting terminal. When the vehicle arrives, the departure gate is displayed and the service continues

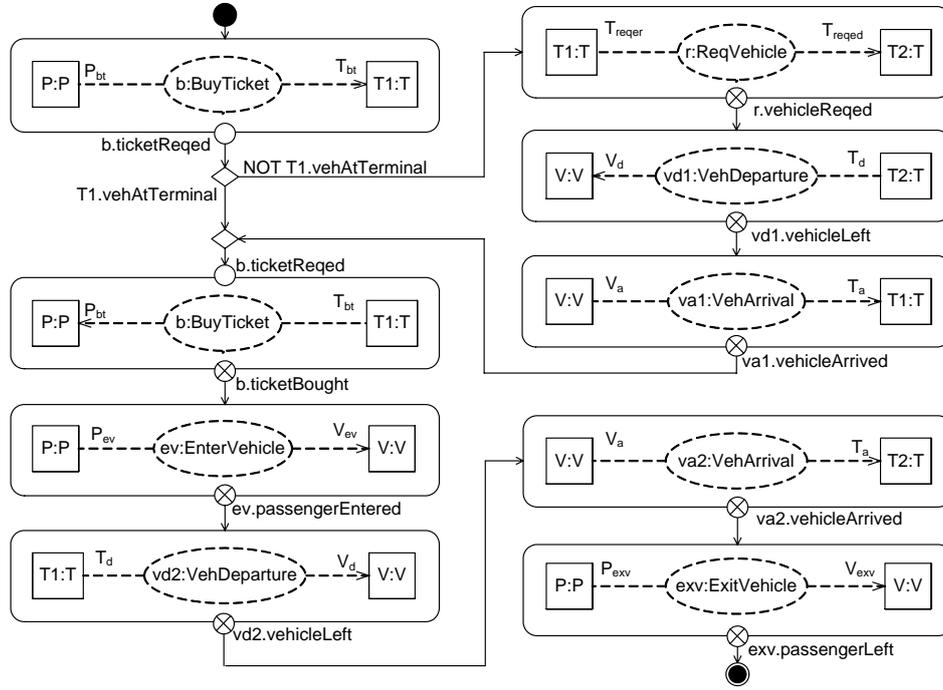
as explained before. In order to support validation and composition, service-goals [SB04] are associated with each of the identified sub-collaborations. These goals are expressed in terms of predicates over properties of the collaborations. Two types of service-goals can be described: event-goals, denoting desired events; and state-goals, which are properties of global collaboration states that we wish to reach, and which entail combinations of role goals. The ordered sequence of goals for an individual collaboration can be described with a *service-goal tree*, which is a directed graph with an initial node, zero or more intermediary nodes representing event-goals, and one or more leaf nodes representing state-goals. Figure 9.1c shows the service-goal tree for *BuyTicket*, with an event-goal (i.e. *ticketReqed*) and a state-goal (i.e. *ticketBought*). Goal trees describe the behaviour of elementary collaborations at a high-level of abstraction, since the interactions are not detailed. These interactions can be specified in sequence diagrams annotated with goal information (by means of continuations), such as the ones presented for *BuyTicket* and *VehArrival* in Fig. 9.1b.

What remains in Fig. 9.1 is to specify the overall cross-cutting behaviour of the *TransportService* collaboration, that is, how its sub-collaborations interact. This kind of behaviour will be distributed among the collaboration roles and is traditionally referred to as a *choreography* in SOA. *Collaboration goal sequences* have been proposed by Sanders [SB04, SCKB05], and extended in [CB06a], to describe the choreography of collaborations. They capture the liveness aspects of composite service collaborations by describing the execution order of their sub-collaborations, and by showing the interactions between these sub-collaborations in terms of goal achievement (hence the name *collaboration goal sequences*). While *service-goal trees* describe the sequence of goals for individual collaborations, *collaboration goal sequences* specify the sequence of goals for their composition. The information provided by the goal trees and the goal sequence should therefore be consistent. In the following we will assume this is the case.

In this paper, we present the formal syntax of goal sequences and provide semantics to them by means of hierarchical coloured Petri-nets (HCPNs) [Jen97] (see Sect. 9.2). We also show how a general purpose tool for HCPNs (i.e. CPN Tools [CPN05]) can be used to analyse goal sequences for the detection of implied scenarios (see Sect. 9.3). These scenarios are a direct consequence of concurrency and correspond to service behaviour that has not been explicitly described in the specification of the service, but that will be present in any implementation of it [AEY00]. The proposed detection approach avoids a global analysis of the service specification, limiting thus the effect of the state-explosion problem. We end with related work and some discussion in Sects. 9.4 and 9.5.

## 9.2 Collaboration Goal Sequences

Collaboration goal sequences complement UML collaborations for the specification of services by describing the execution dependencies that exist between the sub-collaborations (i.e. features) of the service. As an example, Fig. 9.2 depicts the goal sequence for the *TransportService* collaboration. The actual meaning of this diagram will become clear in the following, when we explain the syntax and semantics of goal sequences.

Figure 9.2: Goal sequence for the *TransportService* collaboration

### 9.2.1 Syntax for Goal Sequences

The goal sequences presented here are inspired by UML activity diagrams. Conceptually, they show an ordering of service phases for a service collaboration  $C$ . Each of these phases corresponds to an activity (i.e. round-cornered rectangle) in the goal sequence. In each phase or activity, a specific sub-collaboration of  $C$  is active (so-called activity's *active collaboration*). This is represented by adorning the activity with a collaboration-use, whose roles are bound to instances of  $C$ 's roles. For example, in Fig. 9.2, the *BuyTicket* collaboration is active in the first activity. This is expressed by adorning that activity with a  $b:BuyTicket$  collaboration-use, whose roles (i.e.  $P:P$  and  $T1:T$ ) are bound to instances of *TransportService*'s roles (i.e.  $P:P$  and  $T1:T$ ). The arrow in the binding identifies the offered role. In a goal sequence, the same sub-collaboration may be active in several activities. In some cases these activities represent different phases of that sub-collaboration, while in other cases they represent different occurrences of the sub-collaboration. In the former cases activities are annotated with the same collaboration-use, such as the two first activities to the left in Fig. 9.2. They represent different phases of *BuyTicket* (i.e. before and after requesting the ticket) and are therefore annotated with the same collaboration-use (i.e.  $b:BuyTicket$ ). In the latter cases, activities are annotated with distinct collaboration-uses, as for instance  $va1:VehArrival$  and  $va2:VehArrival$  in Fig. 9.2.

Each activity has one or more exit-points, and may or may not have one entry-point. Both entry- and exit-points represent execution points at which an activity's active collaboration interact with other collaborations. They are labeled with predicates describing goals of the active collaboration. Exit-points can be of two different types. An empty-circle ( $\circ$ ) is used for *suspension* exit-points. They are annotated with event-goals, and correspond to execution points of an active collaboration where the latter can be (or must be) suspended for another collaboration to be started (or resumed). In Fig. 9.2 a suspension exit-point is used in the first activity. The activity's active collaboration (i.e. *b:BuyTicket*) will therefore be suspended when the *ticketRequed* event-goal of *BuyTicket* holds. A crossed-circle ( $\otimes$ ) is used for *end-of-execution* exit-points. They are annotated with state-goals, and represent the end of execution of an active collaboration. Entry-points are drawn as empty circles and annotated with event-goals. They represent the execution point at which a previously suspended active collaboration is to be resumed. When an activity does not have an entry-point, its active collaboration starts execution from its initial state.

Edges (i.e. directed connections between activities) and control-flow nodes (i.e. decision, merge, fork, join, initial and final nodes) are respectively used to allow and coordinate the flow of control among activities. An activity can only have one incoming edge, so multiple incoming edges must be AND- or OR-joined.

According to the concrete syntax just described, the formal syntax of goal sequences can be defined as:

**Definition 9.1** (collaboration goal sequence). A collaboration goal sequence, for a collaboration  $C$ , is a tuple  $GS = (N, E, g_d, m_{\text{exp-a}}, R_{GS}, AC, m_{\text{a-ac}}, m_{\text{enp-a}}, l_{\text{ep-pred}}, \text{exp}_{\text{type}})$  where

- (i)  $N$  is a set of nodes. It is partitioned into an initial node ( $n_0$ ) and sub-sets of activities ( $N_A$ ), entry-points ( $N_{\text{EnP}}$ ), exit-points ( $N_{\text{Exp}}$ ), control flow nodes ( $N_{\text{Flow}}$ ) and final nodes ( $N_{\text{Fi}}$ ). In turn,  $N_{\text{Flow}}$  is partitioned into decision ( $N_{\text{D}}$ ), merge ( $N_{\text{M}}$ ), fork ( $N_{\text{F}}$ ) and join ( $N_{\text{J}}$ ) nodes.
- (ii)  $E \subseteq (N_{\text{Exp}} \cup N_{\text{Flow}} \cup \{n_0\}) \times (N_A \cup N_{\text{EnP}} \cup N_{\text{Fi}} \cup N_{\text{Flow}})$  is a set of directed edges between nodes.
- (iii)  $g_d$  is a guard function for decision nodes' outgoing edges. It is defined from  $\{(s, t) \in E : s \in N_{\text{D}}\}$  into boolean expressions.
- (iv)  $R_{GS} = \{(id, type) : type \in R_C\}$  is a set of role instances, with  $R_C$  being the set of roles of collaboration  $C$ .
- (v)  $AC = \{(id, type, B)\}$  is a set of *active collaborations*, that is, a collaboration-use representing a specific occurrence of one of  $C$ 's sub-collaborations. For each  $ac \in AC$ ,  $id$  is the name of the collaboration-use;  $type$  is the name of the collaboration that actually describes the collaboration-use (i.e. one of  $C$ 's sub-collaborations); and  $B \subseteq R_{\text{type}} \times R_{GS}$  is a set of role bindings, where  $R_{\text{type}}$  is the set of roles of the sub-collaboration named  $type$ .

- (vi)  $m_{a-ac} : N_A \rightarrow AC \times CL$  is a non-injective function that maps active collaborations to activities and classifies the active collaboration's roles as *initiating* or *offered* roles within the context of the mapping (i.e. for the given activity). More formally,  $CL$  is a set of binary relations, such that if  $m_{a-ac}(n_a) = (ac, cl)$ , then  $cl = \{(r, typ) : r \text{ is a role of the collaboration with name } ac.type \text{ and } typ \in \{INIT, OFF\}\}$ .
- (vii)  $m_{enp-a} : N_{EnP} \rightarrow N_A$  and  $m_{exp-a} : N_{Exp} \rightarrow N_A$  are functions mapping entry- and exit-points to activities.
- (viii)  $l_{ep-pred} : (N_{EnP} \cup N_{Exp}) \rightarrow P$  is an injective function labeling each entry and exit-point of an activity with a state predicate of the activity's active collaboration.
- (ix)  $exp_{type} : N_{Exp} \rightarrow \{END, SUSPENSION\}$  is a function that classifies exit-points either as *end-of-execution* or as *suspension* ones.

### 9.2.2 Semantics for Goal Sequences

Goal sequences are given a token-game semantics. Intuitively, when an activity receives an input token, its active collaboration is enabled. If the token is directly received from an edge (i.e. not via an entry-point), the active collaboration can begin execution from its initial state. Otherwise, if the token is received through an entry-point, the active collaboration can resume execution from the state represented by the event-goal labeling the entry-point. The active collaboration in reality begins or resumes its execution when one of its roles takes the appropriate initiative. Thereafter, it evolves until an interaction point with other collaborations is eventually reached. That is, the active collaboration runs until the predicate of one of its activity's exit-points holds. When this happens, the control token is passed on to the next activity or control node. According to this semantics, the intended behaviour of the *TransportService* collaboration, as specified by its goal sequence (Fig. 9.2), closely reflects the requirements. Initially the *BuyTicket* collaboration is started and thereafter suspended after the ticket is requested. At that point, a check is performed to determine if the vehicle is at the terminal (i.e. at  $T1$ ). If the result is positive, *BuyTicket* is finished and *EnterVehicle* is enabled, followed by *VehDeparture*, *VehArrival* and *ExitVehicle*. If the vehicle was not at  $T1$ , this role initiates *ReqVehicle* to request the vehicle from  $T2$ . *VehDeparture* is then executed, followed by *VehArrival*, which allows *BuyTicket* to be resumed. Thereafter the service progresses as explained before.

Formal semantics for goal sequences is provided by mapping them into hierarchical coloured Petri-nets (HCPNs). The selection of HCPNs as the semantic domain has been mainly motivated by two facts. First, Petri-nets in general, and HCPNs in particular, have been extensively studied, and quite a number of quality tools exist that support and automate their analysis. Second, the mapping of goal sequences into HCPNs is rather intuitive, as will become clear later on. Due to space limitations we will assume that the reader is familiar with traditional Petri-nets and will only give a short introduction to (H)CPNs.

Coloured Petri-nets (CPNs) [Jen97] extend traditional Petri-nets by associating a *colour* or data type with each token. In this way, tokens are distinguishable from each other, unlike in traditional Petri-nets. Places has also an associated data type (or *colour domain*) determining the kind of tokens they can contain. Transitions can modify the type and value of their output tokens. In addition, they can have an associated guard stating conditions over its input tokens, that must be satisfied for the transition to become enabled.

**Definition 9.2** (CPN). A non-hierarchical CPN is a tuple  $CPN = (\Sigma, P, T, A, N, C, G, E, I)$  [Jen97] where  $\Sigma$  is a finite set of non-empty *types*,  $P$  is a finite set of *places*,  $T$  is a finite set of *transitions*,  $A$  is a finite set of *arcs*,  $N : A \rightarrow (P \times T) \cup (T \times P)$  is a *node* function,  $C : P \rightarrow \Sigma$  is a *colour* function,  $G$  is a *guard* function mapping boolean guards to transitions,  $E$  is an *arc expression* function labeling arcs, and  $I$  is an *initialisation* function for places.

In a hierarchical CPN it is possible to define *substitution transitions*, which can be decomposed into so-called *subpages* (i.e. subnets). Each subpage has a number of places called *port places*, through which the subpage communicates with its surroundings. The relationship between a substitution transition and its subpage is specified by describing a *port assignment*, which couples the port places of the subpage with the surrounding places, or so-called *socket places*, of the substitution transition. Port and socket places can be classified as input (i.e. accept tokens), output (i.e. deliver tokens) or I/O (i.e. both accept and deliver tokens) places.

**Definition 9.3** (HCPN). A hierarchical CPN is a tuple  $HCPN = (S, SN, SA, PN, PT, PA, FS, FT, PP)$  where  $S$  is a finite set of *pages* (i.e. subnets),  $SN$  is a set of *substitution transitions*,  $SA : SN \rightarrow S$  is a *page assignment* function,  $PN$  is a set of *port nodes*,  $PT : PN \rightarrow \{\text{in, out, i/o, general}\}$  is a port type function,  $PA$  is a port assignment function mapping, for a given substitution transition, its sockets with its subnet's ports,  $FS$  is a finite set of *fusion sets*,  $FT$  is a *fusion type* function, and  $PP$  is a multi-set of *prime pages* [Jen97].

### Informal Mapping

The main idea behind the mapping of goal sequences to HCPNs is to map the collaboration-uses of a goal sequence to substitution transitions, and decompose them into subnets describing the behaviour of those collaboration-uses.

Given a goal sequence describing the behaviour of a collaboration  $C$  (composed of a set of sub-collaborations), we map each collaboration-use of the goal sequence into a substitution transition. This means that several activities may be mapped into the same substitution transition, if they are annotated with the same collaboration-use (e.g. the two activities annotated with  $b:BuyTicket$  in Fig. 9.2 are mapped to the same substitution transition). The mapping of activities and their collaboration-uses is illustrated in Fig. 9.3b. Note that entry-points, as well as *suspension* exit-points of an activity are mapped into I/O socket places of the corresponding substitution transition, while *end-of-execution* exit-points are mapped into output socket places. Therefore, socket places represent event- and state-goals (i.e. the goals labeling

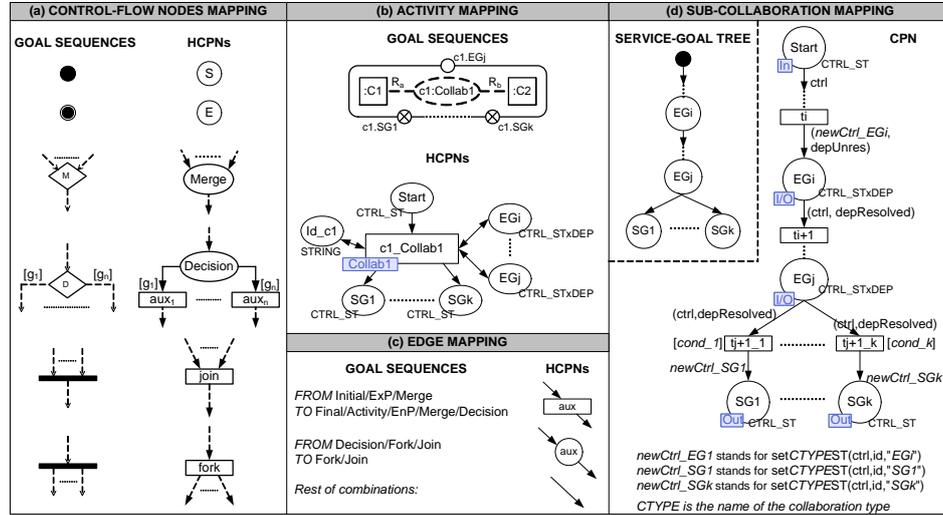


Figure 9.3: Mapping of goal sequence elements to HCPN elements

the entry- and exit-points). In addition, an input socket is added, representing the starting point of the collaboration, as well as an *id* I/O socket, which is used to uniquely identify the specific collaboration-use the substitution transition represents. The colours used for socket places are CTRL\_ST and CTRL\_STxDEP, which are two custom defined data-types. CTRL\_ST represents *C*'s global state, and is composed of the individual states of *C*'s sub-collaborations. CTRL\_STxDEP is a Cartesian product of CTRL\_ST and DEP. The latter is an enumeration with two values: `depUnres` (for dependency unresolved) and `depResolved` (for dependency resolved). The CTRL\_STxDEP type is used to cope with suspend-resume dependencies, which require sub-collaborations to give away the control token while in the middle of execution (i.e. at *suspension* exit-points in the goal sequence). To enforce this behaviour, all tokens leaving I/O socket places (except the *id* one) must be marked with `depUnres`, while all arriving tokens must be marked with `depResolved`.

The initial node, as well as the final and merge nodes of the goal sequence are mapped into places, while join and fork nodes are mapped into normal transitions. The mapping of a decision node yields a place interconnected to as many transitions as the node has outgoing edges. The guards of these edges are then assigned to the transitions. Edges become net arcs, possibly with auxiliary transitions or places so as to respect the bipartite nature of Petri nets. All these mappings are summarized in Figs. 9.3a and 9.3c.

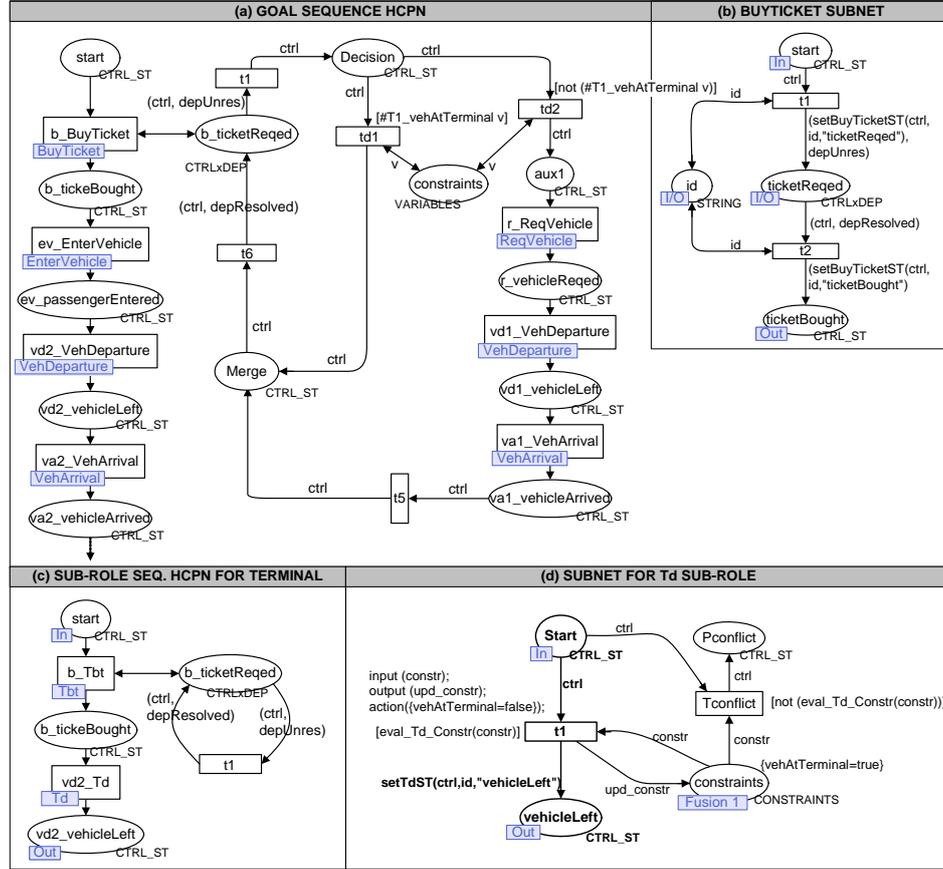
The translation of activities, edges and control-flow nodes, we have just explained, yields the main net of the final HCPN. For the mapping to be complete, we need to describe the decomposition of substitution transitions into subnets. These subnets will describe the behaviour of the goal sequence's collaboration-uses that the substitution transitions represent. As the collaboration-uses of the goal sequence (e.g. *va1:VehArrival* in Fig. 9.2) are occurrences of the sub-collaborations

of  $C$  (e.g. *VehArrival* in Fig. 9.1a), the subnets will describe the behaviour of those sub-collaborations. Several substitution transitions may be assigned the same subnet, if they represent collaboration-uses of the same type (i.e. occurrences of the same sub-collaboration of  $C$ ).

We are not interested on subnets describing detailed behaviour, but rather aim at high-level, abstract behaviour descriptions. Service goal trees (*SGTs*) provide such descriptions, so we use them as input for the mapping of sub-collaborations into subnets (see Fig. 9.3d). The *SGT* nodes are translated into net places, and the *SGT* arcs into net arcs plus an auxiliary transition. Places are characterized as port places: the *Start* place becomes an input port, places representing event-goals (*EG*) become I/O (i.e. bidirectional) ports, and those representing state-goals (*SG*) become output ports. Then, when coupling the subnet's ports with the sockets of a substitution transition, those ports and sockets representing the same goal are interconnected. The *Start* place, as well as those places representing state-goals are typed with the `CTRL_ST` colour, while the `CTRL_STxDEP` colour is used for places representing event-goals. Custom defined functions are used to modify the state of the collaboration (represented by `CTRL_ST`) as the control token travels from the *Start* input port to the output port(s). At each point in time the value of the token reflects the place the token has reached, thus reflecting the event-/state-goal that has been achieved. In addition, all tokens arriving at an I/O port are marked with `depUnres`, while all tokens leaving an I/O port are marked with `depResolved`. This ensures that the control token leaves the net at I/O ports, in order to satisfy suspend-resume dependencies.

All transitions of the subnet will be unguarded, except possibly those leading to output ports (i.e. places representing state-goals). If several transitions lead to different output ports from the same place, as illustrated in Fig. 9.3d, guards may be imposed on those transitions if a deterministic choice is wanted. These guards would determine the conditions to achieve each of the goals. They can be constructed from the information provided by the goal sequence, since the latter describes the relationships between sub-collaboration goals (i.e. it tells us the goal that a sub-collaboration must achieve in order for another sub-collaboration to achieve its own goal). The process to determine these guards is explained in the next section.

Figure 9.4a partially shows the HCPN resulting from the mapping of the *TransportService*'s goal sequence. Each one of the collaboration-uses in Fig. 9.2 has been mapped to a substitution transition. Note that the two activities referring to *b:BuyTicket* correspond now to a single substitution transition (i.e. *b\_BuyTicket*). This substitution transition has one I/O socket (i.e. *b\_ticketReqed*) representing both the suspension exit-point of the first activity and the entry-point of the second activity to the left in Fig. 9.2. Figure 9.4b depicts the subnet describing the behaviour of *BuyTicket*. This is the subnet assigned to the *b\_BuyTicket* substitution transition, and closely resembles the service-goal tree in Fig. 9.1c.

Figure 9.4: Nets for the *TransportService* case study

### Formal Semantics.

For the mapping of goal sequences, we define two semantic functions:  $[[\_]]_{\text{CPN}}$ , which maps elementary sub-collaborations into non-hierarchical CPNs; and  $[[\_]]_{\text{HCPN}}$ , which maps collaboration goal sequences into HCPNs.  $[[\_]]_{\text{CPN}}$  takes a service-goal tree and a collaboration goal sequence, and returns a CPN representing the collaboration whose goals are described by the service-goal tree. A service-goal tree is defined as:

**Definition 9.4** (Service-goal tree). A service-goal tree is a directed graph  $SGT = (cId, GN, GA)$  where:  $cId$  is the name of the collaboration whose goals  $SGT$  describes;  $GN = \{start\} \cup EG \cup SG$  is a set of nodes, with  $start$  being the initial node,  $EG$  being a set of intermediary nodes representing event-goals, and  $SG$  being a set of final nodes representing sate-goals; and  $GA \subseteq N \times N$  is a set of directed arcs between nodes, such that  $\forall (s, t) \in GA : [s \notin SG \wedge t \neq start]$ .

According to the mapping explained in the previous section, and given  $SGT = (cId, GN, GA)$  and  $GS = (N, E, g_d, m_{exp-a}, R_{GS}, AC, m_{a-ac}, m_{enp-a}, l_{ep-pred}, exp_{type})$ , we define  $\llbracket SGT, GS \rrbracket_{CPN} = (\Sigma, P, T, A, N, C, G, E, I)$ , where:

$$\begin{aligned}
 \Sigma &= \{CTRL\_ST, CTRL\_ST \times DEP, STRING\} \\
 P &= GN \cup \{Id\} \quad T = \{t_{ga} : ga \in GA\} \\
 A &= \{sourceTOt_{ga}, t_{ga}TOtarget : ga = (source, target) \in GA\} \\
 &\quad \cup \{IdTOt_{ga}, t_{ga}TOId : Id \in P, ga \in GA\} \\
 N(a) &= (source, target), \text{ if } a \text{ is in the form } sourceTOtarget \\
 C(p) &= \begin{cases} CTRL\_ST, & \text{if } p \in SG \cup \{start\} \\ CTRL\_ST \times DEP, & \text{if } p \in EG \\ STRING, & \text{if } p = Id \end{cases} \\
 E(a) &= \begin{cases} ctrl, & \text{if } (a = s\_t_{ga}) \wedge (s = start) \\ (setCTYPEnST(ctrl, id, "\text{\textasciitilde}gn"), depUnres), & \text{if } (a = t_{ga-t}) \wedge (t \in EG) \\ (ctrl, depResolved), & \text{if } (a = s\_t_{ga}) \wedge (s \in EG) \\ setCTYPEnST(ctrl, id, "\text{\textasciitilde}"), & \text{if } (a = t_{ga-t}) \wedge (t \in SG) \\ iid, & \text{if } [(a = s\_t_{ga}) \wedge (s = Id)] \vee [(a = t_{ga-t}) \wedge (t = Id)] \end{cases}
 \end{aligned}$$

No initial marking ( $I$ ) is defined for the resulting CPN. The guard function  $G$  assigns *true* to all transitions of the CPN, except possibly to those leading to places  $p \in SG$ . To describe how the guards are assigned to those transitions, let us consider the example net in Fig. 9.3d. To determine the set of conditions  $cond_i$ , we just need to search for entry-points in the goal sequence that are labeled with  $EG_j$ . For each entry-point, if its associated activity  $A$  has several exit-points, then the conditions are set to *true* (i.e. representing a non-deterministic choice). Otherwise, if the state-goal labeling  $A$ 's exit-point is  $SG_n$ , then  $cond_n$  is set to the value of the goal labeling the exit-point of the activity immediately preceding  $A$ . Note that  $cond_n$  may actually be a boolean expression of goals, if several activities lead to  $A$  through a control flow node.

As a convention, in the following we will use the notation  $T.E$ , meaning element  $E$  of tuple  $T$ , in order to access the elements of a tuple. We can now define  $\llbracket - \rrbracket_{HCPN}$ , which takes a goal sequence  $GS = (N, exp_{type}, m_{enp-a}, m_{exp-a}, R_{GS}, AC, m_{a-ac}, l_{ep-pred}, E, ga)$  and a set of service-goal trees  $\{SGT_{ac} = (cId, GN, GA), SGT_{ac}.cId = ac.type, ac \in GS.AC\}$  (i.e. one for each sub-collaboration referred to by  $GS$ ), and returns a  $HCPN = (S, SN, SA, PN, PT, PA, FS, FT, PP)$ . We start by introducing the set of subnets ( $S_{AC}$ ), the set of transitions due to the mapping of decision nodes and edges ( $T_D$  and  $T_{edges}$ ), the set of places due to the mapping of arcs ( $P_{edges}$ ), the set of arcs connecting *id* places to substitution transitions ( $A_{Id}$ ), and the set of arcs connecting

the *constraint* place to transitions generated by decision nodes ( $A_{\text{constr}}$ ) as:

$$\begin{aligned}
S_{AC} &= \{[(SGT_{ac}, GS)]_{CPN} : SGT_{ac}.cId = ac.type, ac \in AC\} \\
T_D &= \{t_d : d \in N_D, (d, -) \in E\} \\
T_{\text{edges}} &= \{t_{(es, et)} : (es, et) \in E, es \in \{n_0\} \cup N_M \cup N_{\text{Exp}}, \\
&\quad et \in N_{FI} \cup N_A \cup N_{\text{EnP}} \cup N_M \cup N_D\} \\
P_{\text{edges}} &= \{p_{(es, et)} : es \in T_D \cup N_F \cup N_J, et \in N_F \cup N_J, (es, et) \in E\} \\
A_{Id} &= \{ac.id\_Id \text{TO} ac.id\_ac.type, ac.id\_ac.type \text{TO} ac.id\_Id : ac \in AC\} \\
A_{\text{constr}} &= \{constr \text{TO} t_d, t_d \text{TO} constr : constr \in P, t_d \in T_D\}
\end{aligned}$$

Now we define the main net ( $s_{\text{main}}$ ) describing the interconnection of substitution transitions (representing collaboration uses of the goal sequence). This net is a  $CPN = (\Sigma, P, T, A, N, C, G, E, I)$  described as:

$$\begin{aligned}
\Sigma &= \{CTRL\_ST, CTRL\_ST \times DEP, STRING, VARIABLES\} \\
P &= \{n_0\} \cup N_{FI} \cup N_D \cup N_M \cup \{constr\} \cup \{ac.id\_Id : ac \in AC\} \\
&\quad \cup \{ac.id\_p_{sac} : p_{sac} \in P_{sac}, sac \in S_{AC}\} \cup P_{\text{edges}} \\
T &= \{ac.id\_ac.type : ac \in AC\} \cup T_D \cup T_{\text{edges}} \\
A &= A_{Id} \cup A_{\text{constr}} \cup \{es \text{TO} t_{(es, et)}, t_{(es, et)} \text{TO} et : t_{(es, et)} \in T_{\text{edges}}\} \\
&\quad \cup \{es \text{TO} p_{(es, et)}, p_{(es, et)} \text{TO} et : p_{(es, et)} \in P_{\text{edges}}\} \\
&\quad \cup \{es \text{TO} et : (es, et) \in E, t_{(es, et)} \notin T_{\text{edges}}, p_{(es, et)} \notin P_{\text{edges}}\} \\
N(a) &= (source, target), \text{ if } a \text{ is in the form } source \text{TO} target \\
C(p) &= \begin{cases} CTRL\_ST, & \text{if } p \in \{n_0\} \cup N_{FI} \cup N_D \cup N_M \cup P_{\text{arcs}} \\ STRING, & \text{if } p \text{ is in the form } ac.id\_Id \\ VARIABLES, & \text{if } p = constr \\ C(p'), & \text{if } p \text{ is a socket connected to port } p' \end{cases} \\
E(a) &= \begin{cases} (ctrl, depUnres), & \text{if } a = source \text{TO} target \\ & \text{and } source \text{ is a socket connected to an i/o port} \\ (ctrl, depResolved), & \text{if } a = source \text{TO} target \\ & \text{and } target \text{ is a socket connected to an i/o port} \\ varbl, & \text{if } a \in A_{\text{constr}} \\ id, & \text{if } a \in A_{Id} \\ ctrl, & \text{otherwise} \end{cases} \\
G(t) &= \begin{cases} g_D(e), & \text{if } t = t_d \in T_D, e = (d, -) \in E, d \in N_D \\ true, & \text{otherwise} \end{cases}
\end{aligned}$$

The initialisation function ( $I$ ) of  $s_{\text{main}}$  assigns to the starting place (i.e.  $p = n_0$ ) a token of type CTRL\_ST. This token describes the initial state of the composite collaboration described by  $GS$ , where all state predicates representing the goals of the collaboration are set to *false*.

Finally, we define  $\llbracket GS, \{SGT_{ac}\} \rrbracket_{\text{HCPN}} = (S, SN, SA, PN, PT, PA, FS, FT, PP)$ , where:

$$\begin{aligned}
 S &= \{s_{\text{main}}\} \cup S_{\text{AC}} & SN &= \{ac.id\_ac.type : ac \in AC\} \\
 PN &= \bigcup_{s_{ac} \in S_{\text{AC}}} P_{s_{ac}} & FS &= \emptyset & PP &= \{s_{\text{main}}\} \\
 SA(t) &= s_{ac}, \text{ if } t = ac.id\_ac.type, ac \in AC, s_{ac} \in S_{\text{AC}} \\
 PT(p) &= \begin{cases} in, & \text{if } p = start \\ out, & \text{if } p \in GT_{ac}.SG \\ i/o, & \text{if } p \in GT_{ac}.EG \cup \{Id\} \end{cases}, \forall p \in P_{s_{ac}}, \forall s_{ac} = \llbracket (SGT_{ac}, GS) \rrbracket_{\text{CPN}} \in S_{\text{AC}} \\
 PA(t) &= (ac.id\_p, p), \forall p \in PN, \forall ac.id\_p \in P_{s_{\text{main}}}
 \end{aligned}$$

### 9.3 Detection of Implied Scenarios

A goal sequence describes the intended behaviour of a service from a global perspective, and can be used to synthesize state-machines for the service-roles. The actual service behaviour is performed by the components playing those roles. Since components only have a local view of the service, unexpected interactions may arise. These are the so-called implied scenarios [AEY00], which correspond to service behaviour that has not been explicitly specified, but follows implicitly, and will be present in any implementation of the service. An implied scenario may capture some overlooked positive behaviour, but it may also represent undesired behaviour. Detecting implied scenarios is therefore important.

In the context of the collaboration-based service specification approach treated here, an implied scenario may arise due to the existence of multiple initiatives, from the service-roles, to engage in sub-collaborations. In the collaboration goal sequence these initiatives are ordered in some desired sequence. However, this ordering may not be guaranteed at runtime due to the independence between the initiatives of different service-roles. Therefore, all possible orderings should be analyzed in order to determine if undesired behaviours may arise. Fortunately, this can be done without performing a global analysis of the service collaboration. It suffices to analyse, separately, the sub-role sequences that each service-role may execute. These sub-role sequences can be obtained from the collaboration goal sequence. For example, the following sub-role sequence:  $V_{\text{ev}} \rightarrow V_{\text{d}} \rightarrow V_{\text{a}} \rightarrow V_{\text{exv}}$ ; can be extracted from the goal sequence in Fig. 9.2 for the  $V$  service-role.

Separate sub-role sequences are extracted for each (instance of a) service-role (e.g.  $T1:T$ ,  $T2:T$ , ...). This can be done by invoking the *VISIT* algorithm (see Algorithm 1), with  $i = 0$  and  $n = n_0$ , for each service-role ( $rType$ ), and for each instance of that role ( $rIns$ ). This algorithm traverses the goal sequence's graph ( $GSG$ ) with a depth-first search method, looking for occurrences of  $rIns$ . While traversing the  $GSG$  forwards, the algorithm creates a role-sequence graph ( $RSG$ ) that includes only those activities (and their associated entry-/exit-points) *related* to  $rIns$ .  $RSGs$  have the same syntax and semantics as  $GSGs$ . If a fork node is found, the algorithm adds it to the  $RSG$  and continues the search through one of the fork's outgoing edges. When a decision node is found, one of its outgoing edges is also

**Algorithm 1:** VISIT( $GSG, n, rIns, RSG[rIns, i]$ )

---

```

/* All variables except adjNodes and nextN are global */
/* All elements of the visited array are initialized to 0 before first call
to VISIT */
1 visited[n]++; adjNodes[n] = GETADJACENTNODES(n, GSG)
2 while adjNodes[n] ≠ ∅ do
3   nextN = adjNodes[n].pop()
4   if visited[nextN] < 2 then
5     if ((n ∈  $N_{EnP}$ ) ∨ (n ∈  $N_{Exp}$ ) ∨ (n ∈  $N_A$ )) ∧ RELATED(n, rIns) then
6       | ADDTOGRAPH(n, RSG[rIns, i])
7     else if (n ∈  $N_F$ ) ∨ (n ∈  $N_J$ ) then
8       | ADDTOGRAPH(n, RSG[rIns, i]) and update forks[rIns, i]/joins[rIns, i]
9     else if n ∈  $N_D$  then
10      | update decisions[rIns, i]
11     else if n = n0 then
12      | ADDTOGRAPH(n, RSG[rIns, i])
13      | VISIT(GSG, nextN, rIns, RSG[rIns, i])

// There are no (more) adjacent nodes
14 visited[n] = visited[n] − 1
15 if n ∈  $N_{FI}$  then
16   | // Final node
17   | ADDTOGRAPH(n, RSG[rIns, i]); RSG[rIns, i + 1] = RSG[rIns, i]; i++
17 REMOVEFROMGRAPH(n, RSG[rIns, i]) // Backtracking

```

---

chosen to continue the search, but the decision node is not added to the *RSG* (since at runtime only one of the branches can be executed). Instead, different *RSGs* will be generated for each of the decision's branches (e.g. in our case study two *RSGs* are generated for  $T1:T$ , one for *vehAtTerminal* and other for *NOT vehAtTerminal*). In order to know the decision node's branch a *RSG* corresponds to, the branch's guard is saved in a dedicated table (*decisions*). Once a final node is found, a sub-role sequence has been obtained. From there, a copy of the *RSG* is done and the algorithm begins the backtracking phase. During this phase the previously added nodes are removed from the *RSG* until a decision or fork node with unvisited edges is found. If this happens, one of the unvisited edges is selected and the *GSG* is again traversed forwards (so new nodes are added to the *RSG*). Otherwise, if the initial node is reached during backtracking, the extraction process ends. Note that if fork (resp. join) nodes were found while traversing the *GSG*, the generated *RGSs* describe a path through only one of the outgoing (resp. incoming) edges of these nodes. The individual *RGSs* sharing fork (resp. join) nodes must therefore be merged at the end. To help in this process, each time a fork (resp. join) node is found, information about the traversed edge is saved in a dedicated table (*forks*; resp. *joins*). Note also that loops are traversed only once (i.e. only one iteration is performed). This is achieved by annotating in a table (*visited*) the number of times each node is visited. With this restriction we avoid infinite role sequences, while we ensure that all possible non-repetitive sequences of sub-roles are considered.

Once the sub-role sequences have been obtained, their analysis can start. For each service-role, its sub-role sequences are first analysed individually, and thereafter their interactions are studied. In the individual analysis, we look for any set of two or more consecutive offered sub-roles (i.e. offered sub-roles connected by edges and/or join/fork nodes) that the sequence may contain. Consecutive offered sub-roles may represent a conflict, if they are played in collaborations with different parties, and these collaborations maintain some kind of dependency (e.g. one of them should not finish before the other does). In that case, the dependency might be violated, since the initiatives to start the collaborations are taken by different parties. In the *TransportService* example this happens for the *V* service-role. According to their sub-role sequences,  $V_{ev}$  is to be played in *EnterVehicle* before  $V_d$  in *VehDeparture* (see Fig. 9.2). However there is no way for *T*, which takes the initiative in *VehDeparture*, to know if *Passenger* (*P*) has taken the initiative to start *EnterVehicle*, and when this has finished (i.e. the condition *ev.passEntered* is not visible for *Terminal*). Thus *T* may request *V* to play  $V_d$  before *P* has requested it to play  $V_{ev}$ .

After the individual analysis, we study how the sub-role sequences of a single service-role interact with each other, if executed concurrently. Intuitively, we first constrain the execution of sub-roles by imposing pre- and post-conditions, and then build the cross-product of the sub-role sequences to detect constraint conflicts. For that purpose, sub-role sequences are semantically mapped into HCPNs. This mapping follows the same guidelines as the goal sequence mapping detailed in Sect. 9.2.2, the only difference being substitution transitions labeled with sub-role names, rather than with active collaboration names. As an example, consider Fig. 9.4c, which depicts the HCPN for the sub-role sequence obtained when the *TransportService*'s goal sequence is projected onto *T1:T* and *T1.vehAtTerminal* is *true*. Figure 9.4d presents the subnet representing role  $T_d$  (part in boldface).

The execution constraints (i.e pre- and post-conditions) to be imposed on sub-roles follow from the requirements and the service domain. For example, in our case study we can further restrict the execution of role  $T_d$  (from *VehDeparture*) by setting *VehAtTerminal* and *NOT VehAtTerminal* as part of  $T_d$ 's pre- and post-condition, respectively. In our HCPN model constraints are represented as boolean tokens that reside in a place shared by all the sub-role sequence nets. Since HCPNs do not allow guards to be imposed on substitution transitions (which, remember, represent sub-roles), the pre-condition for the execution of a sub-role is instead specified as a guard on the first transition of the subnet describing the sub-role behaviour. If the guard is satisfied, the transition fires and it updates the value of the constraints according to the post-condition. This is illustrated in Fig. 9.4d for the  $T_d$  sub-role, where the result of calling function *evalTdConstr(constr)* has been imposed as guard of transition *t1*. This function processes the value of the *constr* token, which represents the constraints, and returns *true* if *VehAtTerminal* is *true*. The value of *VehAtTerminal* is updated when *t1* fires, by its code segment. Note that in addition to the *constraints* place, a *Tconflict* transition and a *Pconflict* place have been added to the subnet of  $T_d$ . Note also that *Tconflict* can only be fired when *t1* can not, that is, when *VehAtTerminal* is *false*. In such a case, *Tconflict* "steals" the tokens from the *Start* and *constraints* places forcing a dead-marking to be reached.

This behaviour reflects our desire of a (potential) conflict to be reported if a sub-role cannot be immediately executed when it receives the control token, because its pre-condition is not satisfied.

At the end, the sub-role sequences are composed in parallel and the reachability graph of the resulting net is constructed and analysed in search of dead-markings, which would represent potential conflicts. In order to test our analysis method, we used CPN Tools [CPN05] to analyse an extended version of the *TransportService* (with a control center for mediation between the terminals). A reachability graph with 37 nodes and 58 arcs was generated for the analysis of the sub-role sequences of the *Terminal* (T) service-role. This analysis revealed two implied scenarios: a passenger may miss the vehicle after buying the ticket, if the vehicle is dispatched following a request from the control center; or the vehicle may depart with the passenger before a control center's request has been completely processed. A reachability graph of similar size was generated for the *Vehicle* (V) service-role. As a comparison, the detection method by Uchitel et al. [UKM04], which is of exponential complexity with the number of service-roles, needs to build a safety property for the same case study of 4414 states, if heuristics are used. Although no formal conclusions can be obtained from this comparison, we believe the results show the potential of our approach.

## 9.4 Related Work

Service-oriented specification has been addressed in several works. Rößler et al. [RGG01] suggested collaboration based design with a tighter integration between interaction and state diagram models, and created a specific language, CoSDL, to define collaborations. CoSDL is inspired by SDL, so it fails at providing the cross-cutting service composition offered by UML collaborations and goal sequences. Krüger et al. [KGM<sup>+</sup>04] propose an approach to service engineering that has many commonalities with our own. They consider, as we do, services as collaborations between roles played by components, and use a combination of Use Cases and an extended MSC language to describe them. Liveness is expressed by means of the operators provided by their MSC language, while service structure and role binding are described with, so-called, role and deployment domain models. In our approach UML collaboration diagrams are used to provide a unified way of describing service structure and role bindings, and to provide a framework for expressing liveness with goal sequences. Goal sequences provide interesting opportunities for analysis, as we have discussed.

The concept of implied scenarios was first introduced by Alur et al. in [AEY00], where they presented an algorithm to detect this kind of scenarios from MSC specifications. This work was later extended by Uchitel et al. [UKM04], who proposed an approach for the incremental specification (using both MSCs and HMSCs) of systems, driven by the detection of implied scenarios. The main drawback of Uchitel et al.'s work is, however, the state explosion problem (although they limit it by applying heuristics). Muncchini has proposed an approach for the detection of implied scenarios based on the analysis of HMSCs [Muc03]. His work builds over a previous work of Uchitel et al., and avoids the state explosion problem. Our method also

limits the state explosion problem and it is applicable to UML collaboration-based specifications, while Munccini's approach applies to HMSC-based specifications.

## 9.5 Discussion And Conclusions

UML 2.0 collaborations provide very useful structuring mechanisms for specifying cross-cutting service behaviours. They enable: (a) an attractive structured overview; (b) structural decomposition into features, by means of collaboration-uses; (c) re-usability; and (d) definition of semantic interfaces for dynamic discovery, binding and compatibility checks [SBvBA05]. Still, a proper way to describe the choreography or joint behaviour of the sub-collaborations of a composite collaboration is needed. Collaboration goal sequences can be used to fill this gap. They help to understand and document the relationships and execution dependencies between sub-collaborations, in terms of their goals. Moreover, they can be analysed in order to detect inconsistencies and implied scenarios at an early stage of service specification.

Formal semantics for goal sequences based on hierarchical coloured Petri-nets has been presented here that allows their automated analysis using general purpose tools available for HCPNs. The detection of implied scenarios is done in two phases. First, sub-role sequences are extracted from the goal sequence and individually analysed. Then the cross-product of the sub-role sequences of each service-role is built to examine how they interact. The proposed analysis suffers little from the state explosion problem since the sub-role sequences of each service-role are analysed separately, so the complexity is linear with the number of service-roles. In addition, the analysis is done at a high-level of abstraction (i.e. with role sequences and not message sequences). The proposed implied scenario detection approach demonstrates, in addition, that we have much to gain from the explicit description of features dependencies, and from the analysis and understanding of concurrency on interfaces.

Although we can use HCPN-tools for the analysis of goal sequences, their mapping into HCPNs is still performed manually. Thus, a short-term objective is to provide tool support for the mapping, so the whole process can be automatized. Another interesting issue we plan to work on is how to address the elimination of the implied scenarios. One possibility might be to specify negative goal sequences (as the the negative scenarios in [UKM04]).

## Acknowledgements

We would like to thank Gregor von Bochmann, Cyril Carrez and the anonymous reviewers for their valuable comments on this work.

## References

- [AEY00] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of message sequence charts. In *22nd Int. Conf. on Software Engineering (ICSE'00)*, 2000.

- [CB06] Humberto N. Castejón and Rolv Bræk. A collaboration-based approach to service specification and detection of implied scenarios. In *Proc. of 5th int. workshop on Scenarios and state machines: models, algorithms and tools (SCESM'06)*. ACM Press, 2006.
- [CPN05] CPN Group. CPN Tools Manual. Technical report, Univ. of Aarhus, Denmark, 2005. Available at <http://wiki.daimi.au.dk/cpntools/cpntools.wiki>.
- [Jen97] Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume 1 of *Monographs in Theoretical Computer Science*. Springer-Verlag, 1997.
- [KGM<sup>+</sup>04] Ingolf H. Krüger, Diwaker Gupta, Reena Mathew, Praveen Moorthy, Walter Phillips, Sabine Rittmann, and Jaswinder Ahluwalia. Towards a process and tool-chain for service-oriented automotive software engineering. In *Proc. of the ICSE 2004 Workshop on Software Engineering for Automotive Systems (SEAS)*, 2004.
- [Muc03] Henry Muccini. Detecting implied scenarios analyzing non-local branching choices. In *6th Intl. Conf. Fundamental Approaches to Software Engineering (FASE'03)*, volume 2621 of *LNCS*, pages 372–386, 2003.
- [Obj05] Object Management Group (OMG). *UML 2.0 Superstructure Spec.*, 2005.
- [RGG01] Frank Rößler, Birgit Geppert, and Reinhard Gotzhein. Collaboration-based design of SDL systems. In *Proc. of 10th Intl. SDL Forum*, volume 2078 of *LNCS*, pages 72–89. Springer, 2001.
- [SB04] Richard Torbjørn Sanders and Rolv Bræk. Modeling peer-to-peer service goals in UML. In *Proc. of the 2nd Int. Conf. on Soft. Eng. and Formal Methods (SEFM'04)*. IEEE CS, 2004.
- [SBvBA05] Richard Torbjørn Sanders, Rolv Bræk, Gregor von Bochmann, and Daniel Amyot. Service discovery and component reuse with semantic interfaces. In *Proc. 12th Intl. SDL Forum*, volume 3530 of *LNCS*, 2005.
- [SCKB05] Richard T. Sanders, Humberto N. Castejón, Frank A. Kraemer, and Rolv Bræk. Using UML 2.0 collaborations for compositional service specification. In *ACM/IEEE 8th Intl. Conf. on Model Driven Eng. Languages and Systems (MoDELS)*, volume 3713 of *LNCS*, pages 460–475, 2005.
- [UKM04] Sebastián Uchitel, Jeff Kramer, and Jeff Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Trans. Softw. Eng. Methodol.*, 13(1):37–85, 2004.

---

## Paper 5

### Using collaborations in the development of distributed services.

By Humberto Nicolás Castejón, Gregor von Bochmann and Rolv Bræk.

Published as Technical Report Avantel 2/2008, ISSN 1503-4097, NTNU, February 2008.

This is an extended and revised version of: Humberto Nicolás Castejón, Gregor von Bochmann, and Rolv Bræk. Realizability of collaboration-based service specifications. In *Proc. 14th Asia-Pacific Soft. Eng. Conf. (APSEC'07)*. IEEE Computer Society, December 2007.

#### Notes

An appendix has been included at the end of the paper to discuss some issues discovered after the original publication.



# Using Collaborations in the Development of Distributed Services

Humberto Nicolás Castejón<sup>a</sup>, Gregor von Bochmann<sup>b</sup> and Rolv Bræk<sup>a</sup>

<sup>a</sup>*NTNU, Department of Telematics, N-7491 Trondheim, Norway*  
{humberto.castejon, rolv.braek}@item.ntnu.no

<sup>b</sup>*School of Inf. Technology and Engineering, University of Ottawa, Ottawa, Canada*  
bochmann@site.uottawa.ca

## Abstract

This paper is concerned with the compositional specification of services using UML 2 collaborations, activity and interaction diagrams. It addresses the problem of realizability: given a global specification, can we construct a set of communicating system components whose joint behavior is precisely the specified global behavior? We approach the problem by looking at how the sequencing of sub-collaborations and local actions may be described using UML activity diagrams. We identify the realizability problems for each of the sequencing operators, such as strong and weak sequence, choice of alternatives, loops, and concurrency. Possible solutions to the realizability problems are discussed. This brings a new look at already known problems: we show that given some conditions, certain problems can already be detected at an abstract level, without looking at the detailed interactions of the sub-collaborations, provided that we know the components that initiate and terminate the different sub-collaborations.

## 10.1 Introduction

For several decades now it has been common practice to specify and design reactive systems in terms of loosely coupled components modeled as communicating state machines [Boc78, Bræk79], using languages such as SDL [ITU00] and more recently UML [OMG07]. This has helped to substantially improve quality and modularity, mainly by providing means to define complex, reactive behavior precisely in a way that is understandable to humans and suitable for formal analysis as well as automatic generation of executable code.

However, there is a fundamental problem. In many cases, the behavior of services provided by a system is not performed by a single component, but by several collaborating components. This has been recognized by several authors, such as [Bræ99], [Krü03] and [BKM07], and is sometimes referred to as the “crosscutting” nature of services [FK01, KM03]. Often each component takes part in several different services, so in general, the behavior of services is composed from partial component behaviors, while component behaviors are composed from partial service behaviors. By structuring according to components, the behavior of each individual component can be defined precisely and completely, while the behavior of a service is fragmented. In order to model the global behavior of a service more explicitly one needs an orthogonal view where the collaborative behavior is in focus.

Interaction sequences such as MSC [ITU98], and UML Sequence diagrams [OMG07] are commonly used to describe collaborative behavior, and have proven to be very valuable. They are however, not without limitations. They are expressed in terms of message exchanges, which at an early stage of development may be too detailed. Due to the large number of interaction scenarios that are possible in realistic systems, it is normally too cumbersome to define them all, and therefore only typical/important scenarios are specified. In addition, there are problems related to the realizability of interaction scenarios, i.e. finding a set of local component behaviors whose joint execution leads precisely to the global behavior specified in the scenarios. Some authors have studied the realizability problem in the context of implied scenarios [UKM04, AEY05], i.e. unspecified scenarios that will be generated by any set of components implementing the specified scenarios. Other authors have studied pathologies in interaction sequences, e.g. non-local choices [BAL97], that prevent their realization.

We have asked ourselves if there are better ways to model services. Is it possible to model service behavior more completely? Can it be done in a more structured way without revealing more interaction detail than necessary? Is it possible to support composition and to detect and remove realization problems? And is it possible to derive detailed implementations automatically from service models?

We have found that a promising step forward is to adopt a collaboration-oriented approach, where the main structuring units are collaborations. This is made practically possible by the new UML 2 collaboration concept [OMG07]. The underlying ideas, however, date back to before the UML era [RAB<sup>+</sup>92, RWL96]. Collaborations model the concept of a service very nicely. They define a structure of partial object behaviors, called roles, and enable a precise definition of the service behavior using interaction diagrams, activity diagrams and state machines as explained in [SCKB05], [CB06b] and [KH06]. They also provide a way to compose services by means of collaboration uses and to bind roles to components. In this way, UML 2 collaborations directly support (crosscutting) service modeling and service composition. As we shall see in the following, this opens many interesting opportunities. Fig. 10.1 illustrates the main models involved in the collaboration oriented approach being discussed in the following:

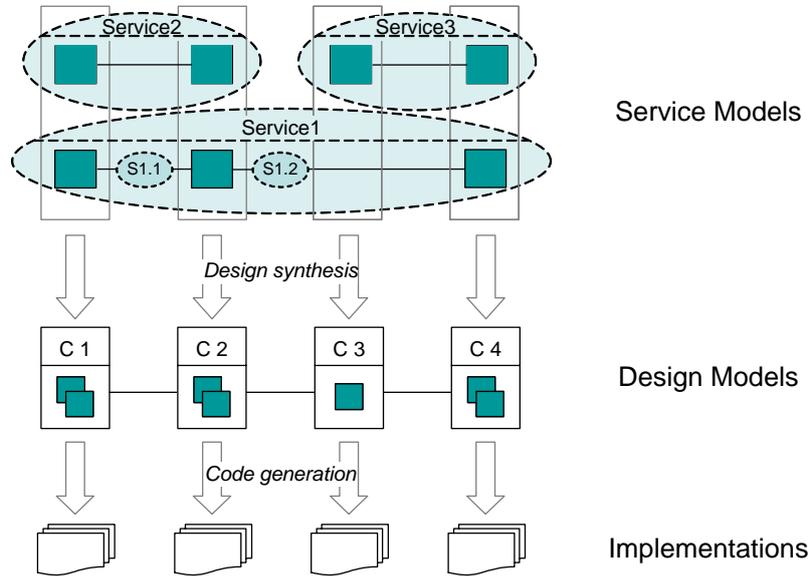


Figure 10.1: Collaboration oriented development

- (i) Service models are used to formally specify and document services. Collaborations provide a structural framework for these models that can embody both the role behaviors and the interactions between the roles needed to fulfill the service.
- (ii) Design models are used to formally specify and document system structure and components providing the services. They are expressed in terms of communicating state machines, typically using UML2 active objects or SDL agents (processes). Each of these will realize one or more collaboration roles.
- (iii) Implementations are executable code automatically generated from the design models.

This paper is concerned with the crucial first steps of expressing service models using UML 2 collaborations and deriving well-formed design models expressed as communicating state machines. The ensuing steps from design component models to implementations and dynamic deployment on service platforms can be solved in different ways, see for instance [San00] and [BM05], and are not discussed further here.

An important property of collaborations is that it is possible and convenient to compose/decompose collaborations structurally into sub-collaborations, by means of collaboration uses. These refer to separately defined collaborations and provide a mechanism for abstraction and collaboration reuse. In order to define the behavior of collaborations, we have found it useful to distinguish between the behavior of *composite collaborations* and *elementary collaborations* (collaborations that are not

further decomposed into sub-collaborations). The elementary collaborations that result from the decomposition process are often quite simple, reusable and possible to define completely in terms of interaction sequences. Binary collaborations can in many cases be associated with interfaces and their sub-collaborations with phases or features of the interface behavior. Assuming the behavior of elementary collaborations are completely defined using interaction diagrams or other notations, the question then is how to define the overall behavior of composite collaborations in terms of sub-collaboration behaviors. In the SOA domain this kind of behavior is called “choreography” [Erl05], a term we will use in the following. Several notations may be used to define the choreography of sub-collaborations (i.e. their global execution ordering). We have found UML2 Activity diagrams a good candidate, as they provide many of the composition operators needed for the purpose. This will be elaborated in Section 10.2.

Interestingly, the operations needed to define a choreography also enable us to identify and classify the underlying reasons leading to realization problems. Many of these can be found by analyzing choreographies at the level of its sub-collaborations without needing to go into interaction details. When this is not possible, potential problem spots can be pinpointed so that detailed interaction analysis can be focused on those. In Section 10.3 we present our results in this area.

In Section 10.4 we discuss the feasibility of automatically synthesizing components from collaborations, i.e. to automate the step from service models to design models. We foresee a process where choreographies defined using activity diagrams are used directly. This points towards a highly automated process form collaboration oriented service models to executable services.

## 10.2 Using Collaborations to Model Services

### 10.2.1 A case study: TeleConsultation

We consider as an example a telemedicine consultation service. A patient is being treated over an extended period of time for an illness that requires frequent tests and consultations with a doctor at the hospital to set the right doses of medicine. Since the patient may stay at home and the hospital is a considerable distance away from the patient’s home, the patient has been equipped with the necessary testing equipment at home. The patient will call the hospital on a regular basis to have remote tests done and consult with a doctor. A consultation may proceed as follows:

- (i) The patient calls the telemedicine reception desk to ask for a consultation session with one of the doctors. The receptionist will register the information needed, and then see if any of the appropriate doctors is available.
- (ii) If no doctor is available, the patient will be put on hold, possibly listening to music, until a doctor is available. If the patient does not want to wait he/she may hang up (and call back later).

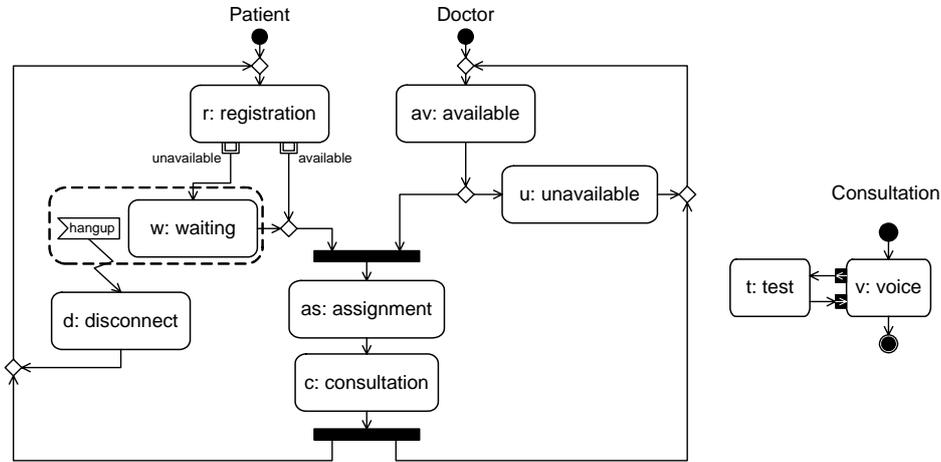


Figure 10.2: Activity diagrams describing a TeleConsultation

- (iii) When a doctor becomes available while the patient is still waiting, the doctor is assigned to the patient and may use the supplied information to look up the patient journal.
- (iv) A voice connection is established between the patient and the doctor allowing the consultation to take place.
- (v) During the consultation the doctor may decide to perform some remote tests using the equipment located at the patient’s site. The doctor evaluates the results and advises the patient about the further treatment. Either the doctor or the patient may end the consultation call.
- (vi) After the consultation call is ended the doctor may spend some time updating the patient journal and doing other necessary work before signaling that he/she is available for a new call. The doctor may signal that he/she is unavailable when leaving office for a longer period, or going off-duty.

Each of these points may be considered an activity. An UML activity diagram describing the order of execution of these activities is given in Fig. 10.2. The fact that the patient and the doctor behave concurrently and may take initiatives independently of each other is reflected by the use of two initial nodes. The result is a diagram with two concurrent parts that are joined for the assignment and consultation activities.

### 10.2.2 Collaboration structure

An important aspect in requirements specification and service modeling, as well as in workflow modeling, is the identification of the actors involved in the different activities.

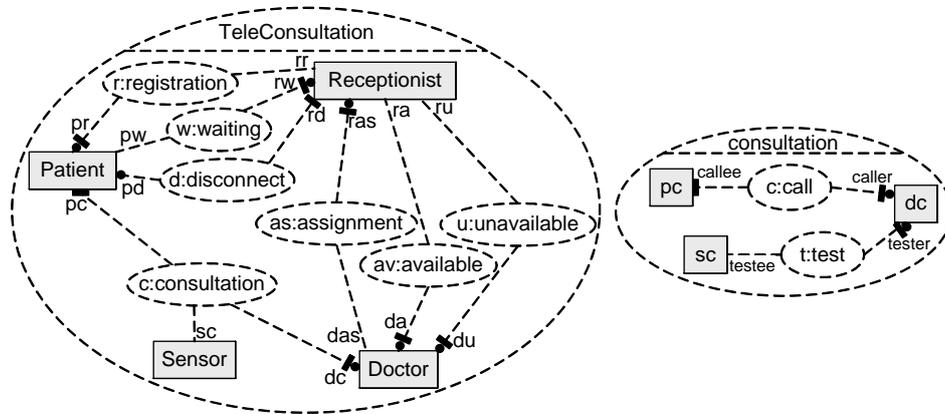


Figure 10.3: Roles and sub-collaborations in the hospital visit

Traditional UML use cases distinguish the *system* and actors that are part of the system environment. For high-level workflow modeling and domain analysis, one usually identifies various actors that participate without necessarily identifying a system. For the TeleConsultation example, we can identify the following actors: the patient, the doctor and the receptionist. It should be noted here that it is useful to make a distinction between real actors and the roles that actors play. In specifications and designs it does not make much sense to identify particular actors. One rather identifies roles that may be composed and bound to real actors in different ways. As illustrated in Fig. 10.2, roles (patient and doctor) may have different responsibilities and follow partly independent workflows with some joint activities.

A UML collaboration diagram is well suited to model role structures and identify sub-collaborations among the roles. This is illustrated for the TeleConsultation example in Fig. 10.3, where 4 roles and 7 sub-collaborations are identified. These sub-collaborations are modeled as UML collaboration uses, and their roles are bound to the roles of TeleConsultation<sup>1</sup>. When decomposing collaborations into sub-collaborations one tends to identify sub-collaborations that involve just two roles. Such *binary* collaborations can be used to define interfaces and interface behavior. This has some advantages: (1) the behaviors are relatively small, (2) they can be completely defined, and (3) they are units of reuse. In this example, all sub-collaborations except *consultation* are elementary and associated with an interface between two roles.

As explained in [SBvBA05] and [San07] binary collaborations can be used to define semantic interfaces that are used to type components in order to enable efficient discovery and compatibility checking at design time and runtime. Used in this way, collaborations are useful during the entire life cycle, not only in early service modeling.

<sup>1</sup>The “dots” and “bars” in this diagram are not standard UML. They are used to indicate the initiating and terminating roles of a collaboration use, as will be explained in Section 10.2.4

In service and workflow modeling one often distinguishes between actors and resources. Both may be associated with a given action and required for its correct execution. The main difference between these two entities is normally that actors may take initiatives, while resources are rather passive. For the modeling of collaborations, we consider both actors and resources as roles that participate in the execution of an action. In the TeleConsultation the sensors are rather passive resources. The doctors may be seen as shared resources from the point of view of patients, and the receptionist as a resource allocator. Contrary to sensors, the patients and the doctors can take independent initiatives, and this exemplifies that resources may also be active.

### 10.2.3 Collaboration behavior: Choreography

The activity diagram in Fig. 10.2 defines the global behavior for the TeleConsultation collaboration. The activities have been chosen so that each activity corresponds to a sub-collaboration in Fig. 10.3, and in this way it defines their choreography. Note how this diagram defines collaboration ordering in a visual way without going into the details of interactions.

UML Activity diagrams appear to be at a suitable level of abstraction for defining the choreography of sub-collaborations within a given composite collaboration. They can express sequential, alternative and concurrent behavior including the possibility of looping, as well as interruption and activity invocation (e.g. *voice* invoking *test* in the *consultation* collaboration), as illustrated in Fig. 10.2. The units of execution are atomic actions or activities. An activity can be further refined and described by a separate activity diagram that will be invoked when the activity becomes activated.

There is, however, one important feature of collaborations that is different from what is normally assumed of an activity. A collaboration involves several roles (participants). In activity diagrams, each activity normally involves only a single role. It must therefore be foreseen that an activity or action, representing a sub-collaboration within an activity diagram, involves more than one role. This information can be provided by annotating the diagram as shown in Fig. 10.4. UML is very open-ended concerning the precise notation for activity diagrams and allows this kind of notational extension. The important point here is the kind of information we would like to include in the models, not exactly how it is done. The same information might be supplied in different ways, as will be discussed in Section 10.2.5.

### 10.2.4 The nature of collaborations

A collaboration describes (joint) actions among a set of roles carried out in order to achieve a goal. The roles describe the behavior and properties that components should exhibit in order to participate in the collaboration. The order in which the actions are performed is enforced locally by the participants and globally by the exchange of messages between the participants. This is in general a partial order, as first explained by Lamport [Lam78]. A collaboration may include behavior alter-

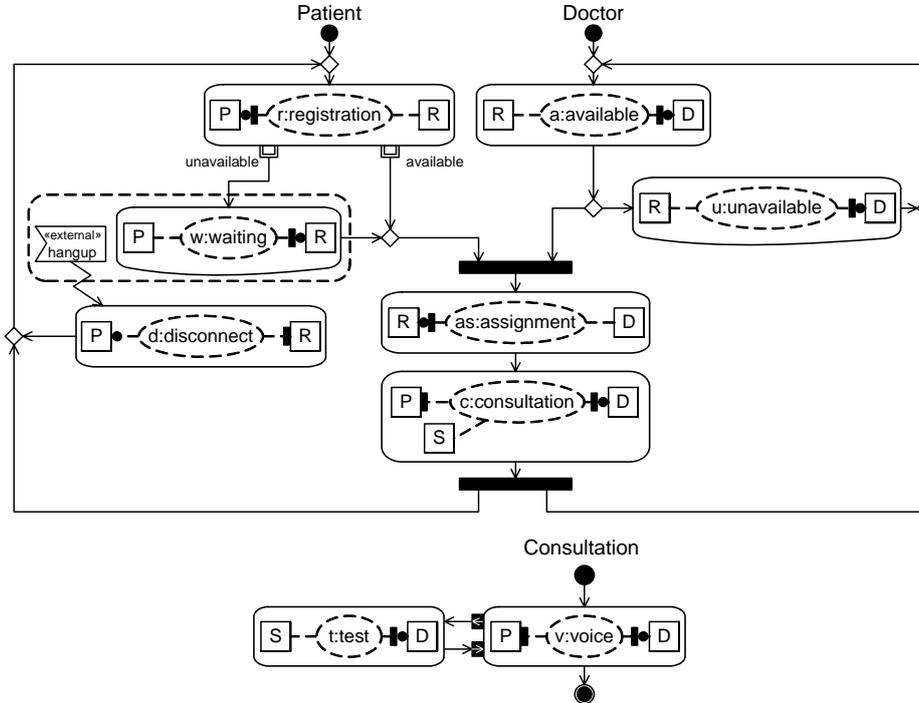


Figure 10.4: Choreography for the TeleConsultation collaboration.

natives leading to different outcomes, as for instance the registration collaboration in the TeleConsultation, see Fig. 10.4.

For each collaboration, we distinguish the *initiating actions* and the *terminating actions*. The initiating actions are those actions for which there is no earlier action in the collaboration according to the (partial) execution order defined for the collaboration. Similarly, the terminating actions are those for which there is no later action in the collaboration. The roles involved in the execution of an initiating (resp. terminating) action of a collaboration are called *initiating* (resp. *terminating*) *roles*. In Fig. 10.3 and Fig. 10.4 the initiating roles have been identified by a dot and the terminating roles by a bar. Which roles initiate and/or terminate the execution of a sub-collaboration is very important for the coordination of the temporal order of execution of different sub-collaborations, as discussed in more detail in Section 10.3.

How the roles of sub-collaborations are bound to roles of enclosing collaborations and eventually to components of a distributed system design is important for the realizability of a specified ordering. Collaboration structures such as Fig. 10.3 specify precisely the binding of **sub-roles** (i.e. roles of sub-collaborations) to the **composite-roles** of the collaboration structure. The diagram in Fig. 10.3, for instance, specifies that the sub-role *pr* from *registration* is bound to the composite-role *Patient* of TeleConsultation. We will say that a composite-role is the initiating

(resp. terminating) role of a sub-collaboration if it is bound to the initiating (resp. terminating) role of that sub-collaboration.

In the case of a sub-collaboration that has several terminating roles, the terminating actions performed by these different roles will normally not be synchronized. This is in contradiction to the semantics of UML activity diagrams which states that all the outputs of an activity will become available at the same time (when the activity terminates)<sup>2</sup>.

We note that a collaboration with more than one initiating role may be not so easy to realize because of the required coordination between the initiating roles for initiating the collaboration. As a general design guideline, it is therefore desirable to avoid collaborations with multiple initiating roles as far as possible. Related issues are further discussed in Section 10.3. Here are some examples:

- (i) Single initiating role: the patient (initiating role) registers with the receptionist.
- (ii) Several alternative initiating roles: each side of a consultation call may take the initiative to perform the call termination sub-collaboration.
- (iii) No initiating role identified: it is not specified whether the doctor or the sensor (triggered by the patient) initiates a test.
- (iv) Several terminating roles: in the voice collaboration the last action may be performed by the patient or the doctor, or both.

At a high-level of abstraction, we may characterize a collaboration by a pre-condition and a post-condition. A collaboration can only be initiated if its pre-condition is satisfied; we say then that the collaboration is *enabled*. The pre-condition describes the required system state for the collaboration to start. As a design guideline, we note that it is desirable that the enabling condition only depends on the state of the initiating roles. The post-condition represents a condition that will be true when the collaboration terminates; if the collaboration admits several alternative outcomes, the post-condition will be the logical OR between these alternatives. One may also specify “goals” in terms of states or events where the purpose of a collaboration is achieved (see [SBvBA05]), when different from the post-condition. In the TeleConsultation example pre- and post-conditions have not been illustrated. However, we can imagine that there would be an *available* predicate reflecting the availability of the doctor. This predicate should for example be true for *assignment* to be enabled, and would become false when this collaboration finishes.

Sometimes it is also useful to identify a *triggering event* for a collaboration. This is an event that leads to the execution of the collaboration if its precondition is true

---

<sup>2</sup>In UML, outputs of an action are identified by so-called pins, outputs of an activity are identified by so-called parameters of the activity. There are two exceptions to the rule that all outputs occur when the activity terminates: (a) Several alternate sets of outputs may be identified (only one of these sets of outputs will occur), and (b) so-called stream inputs and outputs may occur anytime during the execution of an activity.

when the triggering event occurs. In the case of sequential execution of two sub-collaborations, the triggering event of the second collaboration would normally be the termination of a terminating action of the first collaboration. In other cases, the triggering event may be the reception of an external input or a time-out that is not part of the collaboration being modeled. Such *external triggering events* may cause a role to initiate a collaboration seemingly spontaneously and on its own initiative. External triggering events are normally not specified explicitly, only the *initiatives*, i.e. the seemingly spontaneous actions they cause. It is important to identify such initiatives in service modeling for two main reasons: (1) most services and service features are initiated by external initiatives; (2) they give rise to concurrency and potential conflicts. Initiatives normally occur independently. They start threads of sequential behavior, which execute (partly) in parallel with the behavior triggered by other initiatives. In the TeleConsultation service, for example, the patient and the doctor take independent initiatives leading to the parallel paths in Fig. 10.2 and Fig. 10.4. The two paths may be considered as different views on the service; the patient view and the doctor view. These are brought together and coordinated during the assignment and consultation collaborations. The existence of independent initiatives and the need for their coordination is an essential property of the TeleConsultation service and many other services. Independent initiatives may give rise to conflicts, if they are not properly coordinated. This will be discussed further in Section 10.3.

### 10.2.5 Notation

We make in the following some comments about possible notations for representing the choreography of collaborations as described above. The objective is not so much to find a notation as to identify concepts that allow specifying and analyzing the high level flow without prematurely binding the detailed interactions, and that also allow gradual detailing towards interactions and localized behavior.

In Section 10.2.3 we have already discussed the suitability of UML activity diagrams for defining the choreography of sub-collaborations. They allow a compositional specification approach and can express sequential, alternative and concurrent behavior, as well as interruption and activity invocation.

An important aspect of a choreography is to show which roles participate in which collaborations and whether they are initiating or terminating roles. A possible graphical notation for representing multiple roles involved in a single activity was proposed in [Boc00] where resources and roles are represented as separate entities and their involvement in activities by a special type of arrow (see Fig. 10.5(a)). A variant of this is to let activities and roles overlap as illustrated in Fig. 10.5(b). This style can visualize the ordering of sub-roles within a role, and can also be used to localize control flows to particular roles when this is desirable. In the variant shown in Fig. 10.5(c), this is taken one step further by representing the global choreography as one enclosing activity with partitions corresponding to the roles, and sub-collaborations modeled as activities that cross-cuts the partitions. All flow lines are here local to roles and specify precisely how role behaviors are composed. This approach has been used in several case studies, and to demonstrate that design

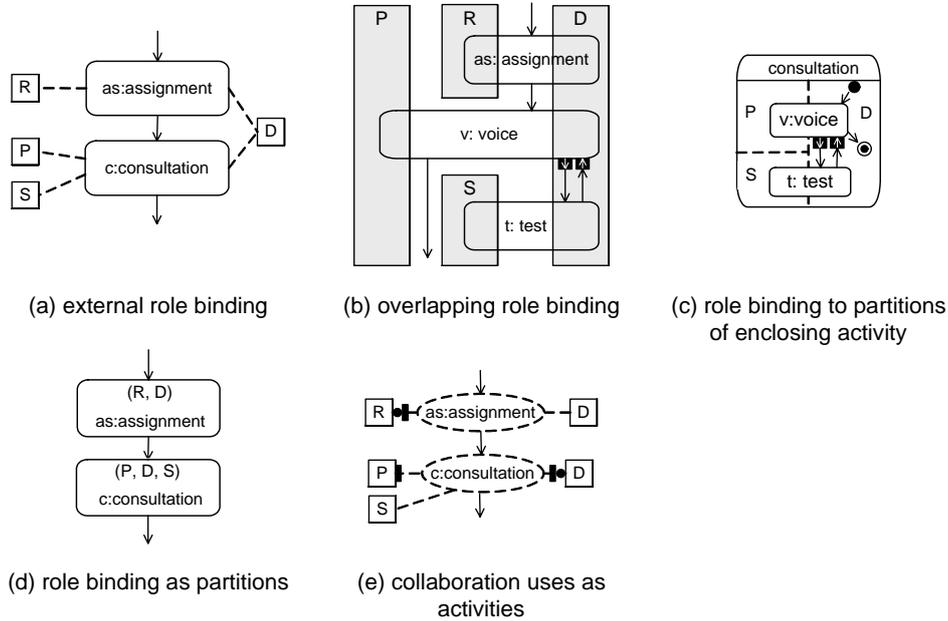


Figure 10.5: Alternative notations for role binding

components defined as state machines can be derived automatically [KBH07]. The UML specification suggests representing the partitions textually as illustrated in Fig. 10.5(d). The notation used in Fig. 10.4 is a visual enhancement of this showing the roles graphically as well as the active collaboration use. It is an extension of the notation originally proposed in [CB06a] and [CB06b]. It has the advantage that each activity has a clear boundary, which helps to organize larger diagrams. In the following we will use the simplified form shown in Fig. 10.5(e) for illustration purposes.

Note that the variants in Fig. 10.5(a) to Fig. 10.5(d) are formed simply by varying the representation and localization of roles relative to activity boundaries. The variants in Fig. 10.5(b) and Fig. 10.5(c) allow localizing control flows to roles. Semantically the only difference between these variants is the localization of control flow. Note that the variant in Fig. 10.5(b) resembles a sequence diagram, where messages are replaced by sub-collaborations. The variants in Fig. 10.4 and Fig. 10.5(e) have an explicit reference to collaboration uses, while in the other variants this is maintained by naming conventions.

High-Level Message Sequence Charts (HMSC) [ITU98] or UML Interaction Overview Diagrams (IOD) are other notations that also have a suitable level of abstraction for choreography, but they are tied to interactions and do not readily allow the same flexibility to combine notations. Moreover, they lack certain operators (e.g. preemption) and semantics needed to fully define collaborative behavior as also pointed out in [Krü03].

A textual notation might also be used to define the temporal ordering of actions and sub-collaborations within a given collaboration. In previous work [BG86, KHB96] a notation based on process algebras was used. If one uses Use Case Maps [Amy03] to model collaborations, the concept of sub-collaboration could be modeled as a "stub", and a participant as a "component"; however, also here it is assumed that each action (called "responsibility") is associated with at most one "component".

To specify the behavior of elementary collaborations there are several options. Activity diagrams may be used to define the behavior in a way that allow component behaviors to be derived automatically [KBH07]. If one chooses to define elementary collaborations using interaction diagrams, these may be referenced from the activities of a choreography [CB06a]. One may also refer to collaborations representing semantic interfaces with goals and behavior defined by two role state machines [SBvBA05]. Alternative approaches of combining sequence diagrams with other diagrams have been proposed by several authors, e.g. [RT03] and [Whi07].

Finally, we mention that it is often useful to introduce variables that are used to define guards for alternate choices or sub-activity invocations. They typically represent databases or state variables and are important for the description of the overall system behavior. At the early stages of development, these variables may be considered global variables (as in Use Case Maps [Amy03]). At the later stages, they must be allocated to particular system components or replaced by other means of keeping the pertinent information.

### 10.3 Ordering operators for choreography

In this section we discuss the sequencing operators that we consider important for describing the execution order of collaborations. These are the standard concepts of sequential execution, alternatives, concurrency, and interruption which are supported by most notations for workflow modeling, including UML Activity Diagrams and Use Case Maps. We discuss in the following some particular semantic features for these concepts which are not provided by the standard semantics of Activity Diagrams. We also introduce the concept of activity invocation, a variant of a remote procedure call. We note that some of these features are also discussed in Wohead's analysis of the control-flow perspective of UML Activity Diagrams [WvdAD<sup>+</sup>05].

#### 10.3.1 Realization problems

We consider in this paper that the model of a distributed service is defined by a composition of several sub-collaborations, as discussed in Section 10.2. The service model defines a global order about different actions that will be performed by different service roles. At the system design level, as shown in Fig. 10.1, the roles of the collaborations are assigned to certain system components and their local behavior may be defined by state machine models. One may attempt to obtain the specification for the behavior of a given system component by projecting the global service behavior specification onto that component, that is, ignoring all actions at the other components in the service model and deriving the order of the local actions at the

given component from the ordering of these actions in the service model. We say that a design model is **directly realized** from a given service model, if the behavior of each system component of the design is obtained by projecting the service model onto the given system component. If the behavior of the directly realized design model is equivalent to the overall system behavior defined by the service model, we say that the service model is **directly realizable**.

Unfortunately, in some cases the directly realized design will generate interaction scenarios not foreseen by the service model. The problem of these *implied scenarios* was originally studied for MSC-based specifications in [AEY00]. This problem is however not unique to MSCs, but inherent to any specification language where the behavior of a distributed system is described from a global perspective, while it is realized by independent components with only local knowledge.

We will discuss in this section under which circumstances a choreography is directly realizable. We will discuss for each composition operator what problems of direct realizability may occur, how they may be detected, and what kind of additional mechanisms could be introduced into the directly realized design model in order to assure that the resulting behavior conforms to the service model. These mechanisms include additional coordination messages, and additional parameters in the messages of the directly realized design. Provided we know the initiating and terminating roles, we are in many situations able to identify problems by looking only at the sub-collaboration ordering defined by the choreography. In other cases, we are able to identify *potential* problems at the choreography level, but need to consider detailed interactions of the sub-collaborations to see if the problems are *actual*, i.e. actually exist.

It is important to note that the question whether a choreography is directly realizable depends not only on the ordering defined by the choreography, but also on the characteristics of the underlying communication service that is used for the transmission of messages between the different system components. Important characteristics of the communication service are the type of transmission channels, and the type of input buffering at each component. We assume that there is no message loss, and distinguish between channels with *out-of-order delivery* (i.e. messages sent from a given source to a given destination may be received in a different order than they were sent) and channels with *in-order delivery*. Concerning the input buffering we distinguish between the following schemes of *message reordering for consumption*:

- (i) *No reordering*: Each component has a single FIFO buffer in which all received messages are stored until they are processed. Messages are consumed in a FIFO order.
- (ii) *Reorder between sources*: A component has separate FIFO buffers for messages received from different source components, and may locally determine from which source the next message should be consumed.
- (iii) *Full reorder*: A component may reorder received messages freely.

In the following we assume that each sub-collaboration of a choreography is directly realizable and discuss for each ordering operator different conditions for the direct realizability of the choreography.

### 10.3.2 Sequence

According to the semantics of UML activity diagrams, each activity is completely finished before the next one starts. In the most general case, this requires global coordination between all system components participating in the two activities. In many cases, such *strong sequencing* is what we want for collaborations too, but sometimes strong sequencing is too restrictive and may be replaced by *weak sequencing* where the sequential order is only enforced locally on each component without global coordination.

It is therefore necessary to allow weak sequencing as well as strong sequencing and to provide some notation for distinguishing between them. We can annotate sequential composition with a constraint of the form " $\{\text{weak}\}$ " and " $\{\text{strong}\}$ " for this purpose. By default we assume weak sequencing and only annotate edges in case of strong sequencing. Note that weak sequencing is the semantics defined for sequential execution in High-Level Message Sequence Charts and UML interaction overview diagrams.

Strong and weak sequential execution of sub-activities impose ordering constraints on the collaborations and may lead to various realization problems which are further discussed below.

#### Strong Sequence

Strong sequencing between two collaborations  $C_1$  and  $C_2$ , written  $C_1 \circ_s C_2$ , requires  $C_1$  to be completely finished, for all its roles, before  $C_2$  can be initiated. It requires a direct precedence relation between the terminating action(s) of  $C_1$  and the initiating action(s) of  $C_2$ , so that the latter can only happen after the former are finished. The situation is particularly simple in the case of a localized sequence composition as defined below.

**Definition 10.1 (localized sequence composition).** A sequence composition  $C_1 \circ C_2$  is a localized sequence composition if all terminating actions of  $C_1$  and all initiating actions of  $C_2$  are located at the same composite role.

In the case of a localized sequence composition, there is no semantic difference between strong and weak sequencing. In this case, the initiator of  $C_2$  can know when  $C_1$  is completely finished. We have the following proposition.

**Proposition 10.2.** *A localized sequence composition of two directly realizable collaborations is directly realizable.*

Note that this property can be checked at the choreography level, i.e. by considering the initiating and terminating roles, without considering detailed interactions. In Fig. 10.4, for example, the condition is not satisfied anywhere so there is no localized sequence composition in the diagram.

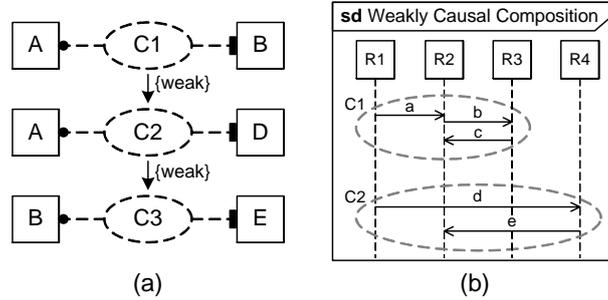


Figure 10.6: Problematic weak sequential compositions

If the condition is not satisfied and strong sequencing is required, coordination messages must be added from  $C_1$ 's terminating composite-roles to  $C_2$ 's initiating-composite roles. This could be done automatically by a synthesis algorithm [BG86].

### Weak Sequence

Weak sequencing of two sub-collaborations  $C_1$  and  $C_2$ , written  $C_1 \circ_w C_2$ , basically requires each composite role in  $C_2$  to be completely finished with previous collaborations before it may initiate participation in  $C_2$ . This means that the actions in the two collaborations are sequenced on a per-role basis. This corresponds to the semantics of MSC and UML Interaction Diagrams.

Weak sequencing introduces concurrency, since the actions of the composed collaborations may partially overlap. Although such concurrency may be desirable for performance or timing reasons (i.e. a role may initiate a new collaboration if the actions in that collaboration are independent of the actions that have yet to be executed in the first collaboration), it comes at a price, since it may lead to specifications that are counter-intuitive and/or not directly realizable, as illustrated in Fig. 10.6.

According to the basic weak sequence semantics, role  $B$  in Fig. 10.6(a) may initiate collaboration  $C_3$  as soon as it has finished with  $C_1$ . As a result, collaborations  $C_2$  and  $C_3$  may be executed in any order in the realized system. This is counter-intuitive to the specification, which we assume reflects the designer's intention (i.e. that  $C_3$  should be executed after  $C_2$ , with some allowed overlapping). If the designer's intention was that the collaborations should be concurrently executed, this should be explicitly specified by means of parallel composition. Note that in this case there are no problems with realizability since the roles of  $C_2$  are completely disjoint from the roles of  $C_3$  and may execute independently. A tool should nonetheless issue a warning that the collaborations may not behave as intended and suggest replacing the sequence with a parallel composition. Note that the composition  $C_2 \circ_w C_3$  has two initiating roles, i.e.  $A$  and  $B$ , that may be executed concurrently. As a guideline such initial concurrency should be avoided in order to ensure a minimal amount of causality between initiatives.

**Definition 10.3 (weak-causality).** A weak sequential composition of two collaborations,  $C_1 \circ_w C_2$ , is weakly-causal if  $C_2$  has a single initiating composite role and this role participates in  $C_1$ .

The weak-causality property ensures that the initial actions of  $C_1$  and  $C_2$  are ordered sequentially. This can be checked at the collaboration level. We note that weak-causality is enforced in the so-called local-HMSCs of [GMSZ06].

Consider the weak sequential composition of  $C_1$  and  $C_2$  in Fig. 10.6(b). This composition is weakly causal, but not directly realizable. Component  $R1$  may initiate collaboration  $C_2$  just after sending message  $a$  in  $C_1$ . Therefore, the actions in  $C_2$  may overlap with the actions in  $C_1$  that follow the sending of message  $a$ . For example, message  $e$  may be received at  $R2$  before message  $c$ , or even before message  $a$ . This message reception order has not been explicitly specified, and therefore it is an implied scenario. Note that such problems may only occur when a composite role (here  $R2$ ) participates in both  $C_1$  and  $C_2$  and plays a non-initiating sub-role in  $C_2$ .

**Proposition 10.4.** A weakly causal composition of two directly realizable collaborations,  $C_1 \circ_w C_2$  is directly realizable if no composite role participating in  $C_1$  participates with a non-initiating role in  $C_2$ .

This property can be easily checked at the choreography level and represents a situation where weak sequencing is unproblematic. In the opposite case, where a non-initiating role in  $C_2$  also participates in  $C_1$ , there is a *potential race* condition.

In the literature about MSCs, the possibility that messages may be received in a different order than the one specified is usually called a **race condition** [AHP96]. In general, a race condition can occur when the specification requires a receiving event to happen after another receiving event or a sending event, and both events are located at the same component. The reason lies in the controllability of events. While a component can control when its sending events should happen, it cannot control the timing of its receiving events.

The actual occurrence of races highly depends on the underlying communication service being used. Channels with in-order delivery prevent races in the communication between a pair of roles, but do not prevent races when more than two roles are involved. This is the case for the situation in Fig. 10.6(b). Such races may in general be resolved by means of input buffering that can reorder between sources (unless choices are involved, as we will see in Section 10.3.3).

We note that race conditions may not only appear between *directly* composed collaborations, but also between *indirectly* composed ones. This is because a role in a collaboration that is composed in weak sequence can remain active during several succeeding collaboration steps. For example, in the TeleConsultation service a race condition exists between *registration* and *consultation* at composite role  $P$  (see Fig. 10.4). In this case it is the weak sequencing between *registration* and *assignment* that makes such race possible, since the sub-role played by  $P$  in *registration* may still be active (i.e. not finished) while *assignment* is executed and when *consultation* is initiated. We therefore say that there is **indirect weak sequencing** between

*registration* and *consultation*. This “propagation” of weak sequencing makes it more difficult to avoid races.

A property that helps to reduce the number of races and facilitates their detection is *send-causality*, which requires all sending events to be totally ordered.

**Definition 10.5 (send-causal composition<sup>3</sup>).** A composition  $C_1 \circ_w C_2$  is send-causal if the composite role initiating  $C_2$  is either the terminating role of  $C_1$  or the role that performs the last sending event of  $C_1$ .

**Definition 10.6 (send-causal collaboration).** A collaboration  $C$  is send-causal if:

- (i) it is a single message transmission, or
- (ii) it is formed by, possibly repeated, send-causal compositions  $C = C_1 \circ_w C_2$  where  $C_1$  and  $C_2$  are send causal.

It has been shown in [CBB07] that when send-causality is enforced, races may only occur between two or more consecutive receiving events (i.e. not between a sending event and a receiving event).

**Proposition 10.7.** *In a send-causal collaboration, race conditions may only exist between two or more consecutive receiving events.*

If  $C = C_1 \circ_w C_2$  is send-causal a *potential race condition* exists on a composite role  $R$  in  $C$  if the sub-role that  $R$  plays in  $C_1$  ends with a message reception (i.e. is a terminating role) and the sub-role  $R$  plays in  $C_2$  starts with another message reception (i.e. is a non-initiating role). Whether the potential race condition is an *actual race* or not depends on the underlying communication service, and on whether messages are received from the same or from different components. For example, in the TeleConsultation service the collaborations *available* and *assignment* are composed in weak sequence (see Fig. 10.4). Role  $D$  plays a terminating sub-role in *available*, while it plays a non-initiating sub-role in *assignment*. Therefore, a potential race condition exists at  $D$  between the receptions of the last message in *available* and the first message in *assignment*. This race is only *actual* in the case of out-of-order delivery. Note that we can identify this potential race simply by considering the initiating and terminating roles in the choreography in Fig. 10.4.

**Proposition 10.8.** *A send-causal weak sequential composition of a sequence of directly-realizable collaborations  $C = C_1 \circ_w C_2 \dots \circ_w C_n$  ( $n > 1$ ) is directly realizable*

- (i) *over a communication service with in-order delivery if the following condition is satisfied: if a composite role plays a terminating role in a collaboration  $C_i$  ( $1 \leq i < n$ ) followed by a non-initiating role in another collaboration  $C_j$  ( $i < j \leq n$ ), then the last message it receives in  $C_i$  and the first one it receives in  $C_j$  are sent by the same peer-composite role; or*

---

<sup>3</sup>For the sake of simplicity, we assume here that each sub-collaboration has only a single initiating event and a single last sending event, but the definition could be easily generalized to consider multiple ones.

- (ii) *over a communication service with out-of-order delivery only if no composite role plays a terminating sub-role followed by a non-initiating sub-role.*

For binary sub-collaborations we can easily identify which composite role sends the first and last messages, if we know which composite roles are the initiating and terminating roles. Using Proposition 10.8, we can determine whether a collaboration is directly realizable and identify actual races at the choreography level without considering the detailed interactions. In the case of n-ary collaborations, we can perform the same early analysis, but only potential races can be discovered.

One interesting aspect of the specification with collaborations is that we can get information about potential races from the diagram describing the structural composition of collaborations, see Fig. 10.3. In such diagrams we can see whether a component participates in several collaborations, and whether it plays at least one terminating and one non-initiating role in them. If that is the case, a potential race exists. This information could then be used to direct the analysis of the behavioral specification (i.e. the choreography). For example, from the collaboration diagram for *TeleConsultation* (see Fig. 10.3(a)) we can see that *Patient* participates both in *registration* and *consultation*, playing a terminating role in the first one and a non-initiating role in the second one. From this information we can conclude that a potential race exists at *Patient* between those two collaborations. We could then check whether a path from *registration* to *consultation* exists in the choreography. If that is the case, the race is actual. Now, if we consider the collaboration diagram for *consultation* it is easy to see that there will not be races between *voice* and *test* at *Doctor*, since the latter does not play a non-initiating role in any of them.

One of our motivations is to provide guidelines for constructing specifications with as few conflicts as possible and whose intuitive interpretation corresponds to the behavior allowed by the underlying semantics. We therefore propose, as a general specification guideline, that all elementary collaborations be send-causal. Weak sequencing of collaborations should also be send-causal, unless there is a good reason to relax this requirement. In the following we assume that all elementary collaborations are send-causal.

### Resolution of Race Conditions

Race conditions can be resolved in several ways. Some authors [Mit05, CKS05] have proposed mechanisms to automatically eliminate race conditions by means of synchronization messages. We note that when the send-causality property is satisfied, a synchronization message should be used to transform the weak sequencing leading to the race into strong sequencing. If synchronization messages are added in other places new races may be introduced. For example, in the *TeleConsultation* service (see Fig. 10.4) the race condition between *registration* and *consultation* at composite role *P* may be eliminated by introducing strong sequencing between *registration* and *assignment*.

Other authors (e.g. [KZ05, MRW06]) tackle the resolution of race conditions at the design and implementation levels. They differentiate between the reception and consumption of messages. This distinction allows messages to be consumed in an

order determined by the receiving component, independently of their arrival order. In general, this reordering may be implemented by first keeping all received messages in a (unordered) pool of messages. When the behavior of the component expects the reception of one or a set of alternative messages, it waits until one of these messages is available in the message pool. Khendek et al. [KZ05] use the SDL Save construct to specify such message reordering. This technique can be used to resolve races between messages received from the same source (i.e. in the case of channels with out-of-order delivery), as well as races between messages received from different sources. It corresponds to the full reordering for consumption capability mentioned in Section 10.3.1. Finally, races may also be eliminated if an explicit consumption of messages in all possible orders is specified (i.e. similar to co-regions in MSCs). We note that in the presence of choices, message reordering may only be possible if the messages to be reordered are marked with the *id* of the collaboration instance that they belong to (see Section 10.3.3).

We believe that the resolution of races heavily depends on the specific application domain and requirements, as well as on the context in which they happen. In some cases the addition of synchronization messages is not an option, and a race has to be resolved by reordering for consumption. In other cases, such as when races lead to race propagation problems (see Section 10.3.3) a strict order between receptions is required, so components should be synchronized by extra messages. At any rate, all race conditions should be brought to the attention of the designer once discovered. She could then decide, first, whether the detected race entails a real problem (e.g. there is no race at  $P$  between *registration* and *consultation* if all channels have the same latency). Then, she could decide whether reordering for consumption is acceptable or synchronization messages need to be added or the specification has to be revised.

### 10.3.3 Alternatives

We consider here the case that at some point of the execution of a collaboration, two or more alternative sub-collaborations may be performed. Alternative composition is specified by means of choice operators, and describes alternatives between different execution paths. In a choice one or more *choosing* composite roles decide the alternative of the choice to be executed, based on the (implicit or explicit) conditions associated with the alternatives. Choosing roles are initiating roles. The other *non-choosing* composite roles involved in the choice follow the decision made by the choosing roles (i.e. execute the alternative chosen by the latter). Non-choosing roles are non-initiating roles. It is thus important that:

- (i) The choosing roles, if several, agree on the alternative to be executed. We call this the **decision-making process**.
- (ii) The decision made by the choosing roles is correctly propagated to the non-choosing roles. We call this the **choice-propagation process**.

In the following we study how each of these aspects affect the direct realizability of a choice. We note that a choice can be seen as a sequential composition with one inlet

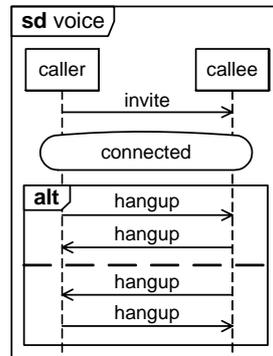


Figure 10.7: A sequence diagram for the voice collaboration illustrating mixed initiatives with common goals

and a set of alternative outlets. The propositions and guidelines for sequential composition, given in Section 10.3.2, apply to every path through the choice. However, we will see how the choice-propagation process affects the resolution of races.

We assume collaborations to be weakly-causally composed, and therefore consider that the set of choosing roles is the union of the initiating roles of all collaborations immediately following the decision node.

### Decision-making Process

We may distinguish the following situations:

- (i) The enabling conditions of the alternatives are mutually exclusive; only one of the sub-collaborations can be initiated.
- (ii) The enabling conditions of several alternatives could be true; if the initiating composite roles of these sub-collaborations are different and there is no coordination between these roles, several alternatives may be initiated concurrently. We call this situation *mixed initiatives*. In many cases this is due to uncoordinated external triggering events, represented by independent initiatives in the collaborations, see Fig. 10.7. We distinguish the following two sub-cases:
  - a) The different sub-collaborations have different goals; only one of them should succeed. We call this situation *competing initiatives*.
  - b) The different sub-collaborations have the same goal; there is no conflict between them at the semantic level, however, there is a conflict at the level of message exchanges. Example: the doctor and the patient initiates the terminating collaboration of a voice call at the same time, see Fig. 10.7. We call this situation *mixed initiatives with common goals*.

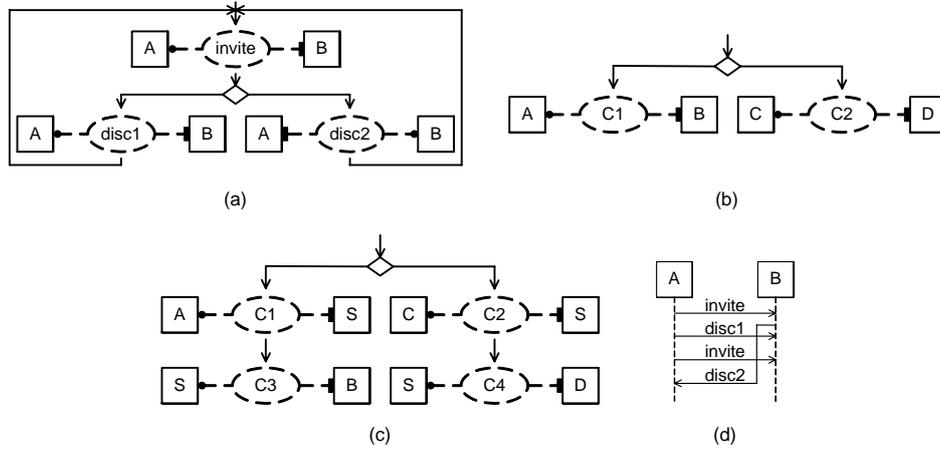


Figure 10.8: (a) Example of a non-local choice; (b) Non-local choice where a mixed initiative conflict cannot be detected; (c) Non-local choice where a mixed initiative conflict can be detected; (d) Possible scenario resulting from (a)

**Local Choice.** Deciding the alternative to be executed becomes simple if there is only one choosing role, and the enabling conditions and triggering events for the alternatives are local to that role (i.e. they are expressed in terms of observable predicates, and events). Choices with this property are called **local**. It is easy to see that the decision making process of local choices are directly realizable, since the decision is made by a single role based only on its local knowledge.

**Non-local choice.** The decision-making process gets complicated when there is more than one choosing role, as in Fig. 10.8(a), where there are two choosing roles, namely *A* and *B*. From a global perspective, our intention is that once the choice node is reached, either role *A* initiates collaboration *discA* with *B*, or role *B* initiates collaboration *discB* with *A*. We are assuming then that there is an implicit synchronization between *A* and *B*, which allows them to agree on the alternative to be executed. However, in a directly realized system, components *A* and *B* will not be able to synchronize and may decide to initiate both collaborations simultaneously resulting in a mixed initiative.

Choices involving more than one choosing role are usually called **non-local choices** [BAL97]. They are normally considered as pathologies that can lead to misunderstanding and unspecified behaviors, and algorithms have been proposed to detect them in the context of HMSCs (e.g. [BAL97, H el01]). Despite the extensive attention they have received, there is no consensus on how they should be treated. We believe this might be motivated by a lack of understanding of their nature. Some authors (e.g. [BAL97]) consider them as the result of an underspecification and suggest their elimination. This is done by introducing explicit coordination, as a refinement step towards the design. Other authors look at non-local choices as

an obstacle for realizability and propose a restricted version of HMSCs, called *local HMSCs* [HJ00, GMSZ06], that forbid non-local choices. Finally, there are authors [GY84, MGR05] that consider non-local choices to be inevitable in the specification of distributed systems with autonomous processes. They propose to address them at the design level, and propose a generic implementation approach for non-local choices.

A non-local choice shows up at the choreography level as a choice where the alternatives have different initiating roles. We may avoid the problem of mixed initiatives by coordinating these initiating roles (e.g. either with additional messages or with additional message contents). This would make the choice local in practice. Unfortunately, such coordination is not always feasible. If the alternatives are triggered by independent external events (represented by independent initiatives), we call the choice an **initiative choice**. In these choices the occurrence of mixed initiatives is unavoidable. In the TeleConsultation service, for example, two collaborations are enabled after the execution of *available: assignment* and *unavailable* (see Fig. 10.4). The triggering events for these come from the end-users (i.e. the actual doctor and receptionist) that operate independently and are not coordinated. It makes little sense to coordinate components *D* and *R* in order to obtain a local choice, since this would imply the coordination of the end-users' initiatives. Such non-local choice is simply unavoidable. It is an initiative choice.

Any role involved in two or more alternatives of an initiative choice may be potentially used to detect a mixed initiative and initiate the resolution. For such roles, the mixed initiatives reveal themselves in the role behavior as choices between an initiating and a non-initiating sub-role, or between two non-initiating sub-roles played in collaborations with different peers. Note that if two alternatives with different choosing roles have no common roles, a mixed initiative conflict can not be detected (see, for example, Fig. 10.8(b)). If the intention is that they shall be mutually-exclusive, an arbiter role should be introduced. Such arbiter role would act as an intermediary between the choosing roles and the non-choosing roles, and could detect a mixed initiative conflict (e.g. the choice in Fig. 10.8(c) results from adding an arbiter role *S* to the choice in Fig. 10.8(b)).

Situations of initiative choices were discussed by Gouda et al. [GY84] and Mooij et al. [MGR05]. These authors propose some resolution approaches. In the domain of communication protocols, Gouda et al. [GY84] propose a resolution approach for two competing alternatives (i.e. two choosing components), which gives different priorities to the alternatives. Once a conflict is detected, the alternative with lowest priority is abandoned. With motivation from a different domain, where Gouda's approach is not satisfactory, Mooij et al. [MGR05] propose a resolution technique that executes the alternatives in sequential order (according to their priorities), and is valid for more than two choosing components. We conclude that the resolution approach to be implemented depends on the specific application domain. We therefore envision a catalog of domain specific resolution patterns from which a designer may choose the one that better fits the necessities of her system. We note that any potential resolution should also address the problem of orphan messages, see Section 10.3.5, which is not considered in either [GY84] or [MGR05].

### Choice-propagation Process

The decision made by the choosing component must be properly propagated to the non-choosing components, in order for them to execute the right alternative. In each alternative, the behavior of a non-choosing component begins with the reception of a sequence of messages, which we call the *triggering trace*. Thereafter, the component may send and receive other messages. It is the triggering traces that enable a non-choosing component to determine the alternative chosen by the choosing component. In some cases, however, a non-choosing component may not be able to determine the decision made by the choosing component. As an example, we consider the local choice in Fig. 10.9(a). For the component  $R3$ , the triggering traces for both alternatives are the same (i.e. the reception of message  $x$ ). Therefore, upon reception of  $x$ ,  $R3$  cannot determine whether  $R1$  decided to execute collaboration  $C_1$  or  $C_2$ . That is,  $R1$ 's decision is ambiguously propagated to  $R3$ . We say a choice has an **initial ambiguous propagation** if there is a non-choosing component for which the triggering traces *specified* in two alternatives have a common prefix. Note that according to this definition, triggering traces such as  $(?x, ?y)$  and  $(?x, ?z)$  cause initial ambiguous propagation since in any direct realization, the choice cannot be made immediately after  $?x$ . An easy solution in this case would be to delay the choice (i.e. extract  $?x$  from the choice). Choices with ambiguous propagation are not directly realizable. They are similar to the non-deterministic choices defined in [MRW06]. Unfortunately, ambiguous propagation cannot be detected at the choreography level as it depends on the detailed interactions of the sub-collaborations. In order to avoid ambiguous propagation, [BG86] suggested the introduction of a message parameter that indicates to which branch of the choreography the message belongs.

If any of the alternatives contains a weak sequence with a race condition, the race may make the propagation ambiguous. Consider the choice in Fig. 10.9(b). In this case there is a race in the weak sequence of  $U$  and  $V$ . The choice is followed by either  $V$  or  $U^+V$  and may result in the situation depicted in Fig. 10.9(c). This example shows that in the presence of race conditions the triggering trace *observed* by a non-choosing component may differ from the specified one. Therefore, whenever race conditions may appear in any of the alternatives, we need to consider the potentially observable triggering traces in the analysis of choice propagation. For example, in Fig. 10.9(b) the specified triggering traces for  $R3$  are  $(?b^+, ?c)$  and  $(?c)$ . However,  $R3$  may observe triggering traces such as  $(?c, ?b)$  or  $(?b, ?c, ?b)$ . We say a choice has a **race propagation** if there is ambiguous propagation due to races. Choices with race propagation are not directly realizable. They are similar to the race choices defined in [MRW06].

Choices without ambiguous or race propagation are said to have **proper decision propagation**. These choice propagations are directly realizable.

### Resolution of Race Propagation

To resolve the problem of race propagation we need to resolve the race(s) that lead to it. However, if we try to remove the race conditions by means of message reordering for consumption (e.g. by means of separate input buffers), the race propagation

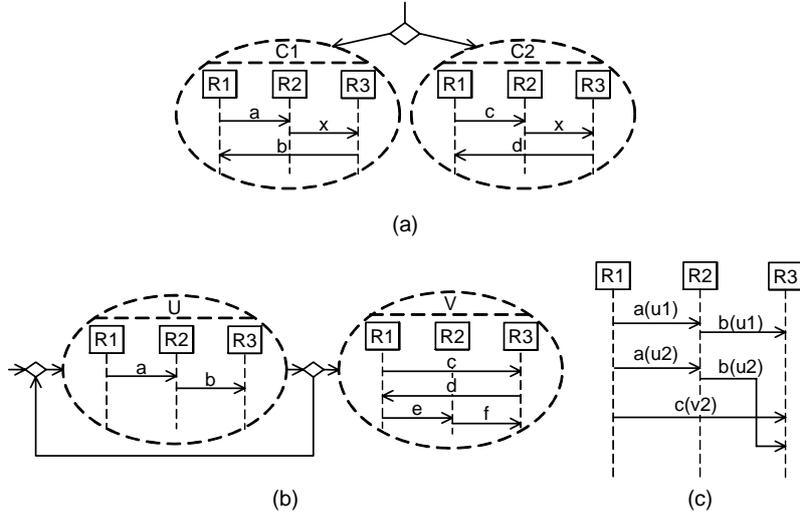


Figure 10.9: Choices with (a) ambiguous propagation and (b) race propagation; (c) Behavior implied by (b)

problem may still persist. This is because, in general, a component would not be able to determine whether a received message should be immediately consumed as part of one alternative, or be kept for later consumption in another alternative as illustrated by the race propagation in Fig. 10.10(a). To make the message reordering work, we need to mark the messages with the id of the collaboration *instance* they belong to. In order to obtain such an id, we need to unfold the branches of the choice in the choreography graph, so that they do not share any activity. Then, we need to assign a different id to each activity referring to the same collaboration (e.g. Fig. 10.10(b) shows the unfolding of the choice in Fig. 10.10(a) and the assignment of distinct collaboration ids). Fig. 10.10(c) shows a possible scenario during the execution of the choice in Fig. 10.10(a). If the messages are not marked,  $R_3$  cannot determine whether it should consume message  $d$  immediately after receiving it (i.e. in case  $C_2$  has been directly executed after the decision node), or whether it should keep it on the buffer until it receives message  $b$ . Marking message  $d$  with  $C_2$  (i.e. the id of the collaboration “type” it belongs to) would not help. It has to be marked with the id of the actual collaboration instance it belongs to (i.e.  $C_2'$ ).

When loops are involved, we need to consider the number of iterations of the loop in order to create the collaboration ids. Consider, for example, the choice in Fig. 10.9(b). An unfolding of this choice would give an infinite number of alternative paths:  $U_1 \rightarrow V_1$ ,  $U_1 \rightarrow U_2 \rightarrow V_2$ ,  $U_1 \rightarrow U_2 \rightarrow U_3 \rightarrow V_3$ , and so on. In order to assign a proper id to each instance of  $U$  (or, in general, to each instance of a collaboration inside a loop) we just need to use the iteration number. The id assigned to each instance of  $V$  (or, in general, to each instance of a collaboration following the loop) depends on the total number of iterations that have been executed. The messages in

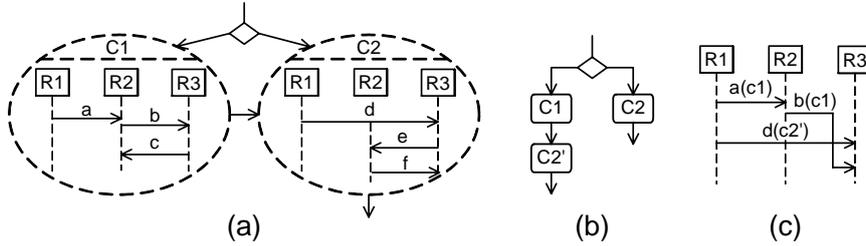


Figure 10.10: (a) Choice with race propagation; (b) Unfolding of (a); (c) Behavior implied by (a)

the scenario of Fig. 10.9(c) have been marked following these principles. Therefore, when  $R3$  receives message  $c(v2)$  it may determine that there is still one message  $b$  on the communication medium, and wait for it without consuming  $c$  (for this  $R3$  needs to keep the count of  $b$  messages that it has received).

In [GMSZ06] the realizability of local-HMSCs is studied. The authors propose to implement the behavior of each component by means of a simple linear algorithm. This algorithm is based on the idea of marking messages with the id of the HMSC node they belong to. This is basically the same idea that we have just discussed for the resolution of race choices. Indeed, although the authors do not explicitly study the race propagation problem, their solution should in principle avoid such problems. The authors do not explain, however, the way to achieve a unique id for each HMSC node. They might consider this as something trivial, although we have shown it is not so trivial. Moreover, they propose marking all messages that are exchanged, and not only those involved in a race propagation. Components therefore have to check the data carried by *all* messages, and decide whether to consume them or not. We believe this unnecessarily increases the amount of processing needed upon message reception. We prefer to detect the cases of race propagation. Then, if we want to resolve the problematic propagation by means of message reordering<sup>4</sup>, we design the components so that they only mark the involved messages, and apply message reordering only to them. Alternatively, we may decide to resolve the race propagation in a probably simpler way, that is, by transforming the responsible weak sequencing into strong sequencing (see e.g. [NHT98]).

### 10.3.4 Merge

When two or more preceding flows merge into a single successor flow, this may be seen as a set of sequential compositions where each preceding flow is composed with the succeeding flow. The propositions and guidelines given in Section 10.3.2 apply to each flow composition.

<sup>4</sup> This avoids not only race propagation, but ambiguous propagation in general

### 10.3.5 Loop

Loops can be used to describe the repeated execution of a (composite) collaboration, which we call the *body* collaboration. A loop can be seen as a shortcut for strong or weak sequential composition of several executions of the same body collaboration, combined with a choice and a merge (see e.g. Fig. 10.9(b)). This means that the rules for strong/weak sequencing with choices and merges must be applied. We note that all executions of a loop involve the same set of components. This fact makes the chances for races high when weak sequencing is used even though the weak-causality property is always satisfied. Strong sequencing should therefore be preferred in loops. When strong sequencing is specified between any two executions of the body collaboration (e.g. to be sure that one iteration is completely finished before the next one starts), the body collaboration should be initiated and terminated by the same component. When send-causal weak sequencing is specified, the component initiating the loop-body collaboration should be the one sending or receiving the last message exchanged in the collaboration.

Loops may give rise to so-called *process divergence* [BAL97], characterized by a component sending an unbounded number of messages ahead of the receiving component. This may happen with weak sequencing if the communication between any two of the participants in the body collaboration is unidirectional. They may also give rise to so-called *orphan* messages, i.e. messages sent in one iteration and received in a later iteration. Consider the specification in Fig. 10.8(a), and imagine that each collaboration consists of only one message. Then the scenario in Fig. 10.8(d) is possible, where message *discB* is sent as a response to the first *invite* message, but it is received by *A* after having sent the second *invite*. Component *A* may then consume message *discB* as a response to the second *invite* message, leading to an undesired behavior. In this scenario, *discB* is a so-called *orphan* message.

Situations similar to loops occur if several occurrences of the same collaboration may be weakly sequenced (e.g. several consecutive sessions of a service).

### 10.3.6 Concurrency

Concurrency means that several sub-collaborations are executed independently from one another, possibly at the same time. We use forks and joins to describe concurrency, and we require they are properly nested as in UML Interaction Overview Diagrams. Concurrent sub-collaborations are directly realizable as long as they are completely independent (i.e. their executions do not interfere with each other). This is clearly the case when there is no overlap among the roles. When a role participates in several concurrent collaborations it must be possible to distinguish messages from the different collaborations, otherwise messages belonging to one collaboration may be consumed within a different collaboration.

In the TeleConsultation example, the receptionist participates in two concurrent flows, and this indicates that the flows are partially dependent. In this case the receptionist serves to coordinate the doctor and the patient. Concurrent activities often involve shared resources for which there is competition that require coordination. Seen from the patient, the doctors are shared resources and the coordination is performed by the receptionist.

Indirect dependencies may also exist through passive shared resources, and shared information. In this case, appropriate coordination has to be added between the collaborations, which will normally be service-specific. In [CB06a] and [CB06b] we discussed the automatic detection of problems due to shared resources, between concurrent instances of the same composite service collaboration. This detection approach makes use of pre- and post-conditions associated with sub-collaborations, and could also be used to detect interactions between concurrent collaborations composed using forks and joins.

In a fork, a preceding flow is followed sequentially by a set of two or more succeeding flows running concurrently. The opposite takes place in a join; a set of two or more preceding flows running concurrently is followed sequentially by a single succeeding flow. For each of the sequential flow compositions in the set of compositions defined by a fork/join the conditions for (weak/strong) sequential composition explained in Section 10.3.2 apply.

For strong sequencing, all the collaborations immediately succeeding a fork must be initiated by the role terminating the collaboration preceding the fork. Similarly, all the collaborations immediately preceding a join must terminate at the component initiating the collaboration succeeding the join. If this is not the case, coordination messages may be added before the join/fork to ensure strong sequencing [BG86].

### 10.3.7 Interruption

We consider here the interruption of a sub-collaboration  $C$  by another sub-collaboration  $C_{\text{int}}$  that may become enabled, for instance as soon as  $C$  is initiated, or when  $C$  reaches a certain state.  $C_{\text{int}}$  requires a triggering event to be initiated, normally in the form of a request coming from an external user or another active agent. In the *TeleConsultation* the observation of the external event *hangup* performed by the patient results in the interruption of the *waiting* collaboration by the *disconnect* collaboration.

As noted in [KHB96], a semantics for cancellation with immediate termination of all activities in the interrupted process is not directly realizable in a distributed system. Instead, one has to assume that the cancellation takes some time to propagate to all participants in the interrupted sub-collaboration, which means that certain activities of the interrupted process may still proceed for some time after the cancellation has been initiated. For example, a client may send a request to a server and, shortly after that, decide to send a cancellation message. While this message is on the way, the server would continue processing the request, and may even send a response back to the client before it receives the cancellation message. The client would then receive an unexpected response message. Similarly, the server would receive a non-awaited cancellation message.

Interruption composition is akin to mixed initiatives where the preempting collaboration has priority. Interruption implies that resolution behavior must be added. However, with interruptions the existence of mixed initiatives is clearly visible in the choreography. The detection is thus easy at the choreography level.

### 10.3.8 Activity invocation

In many cases, a given collaboration  $A$  needs to invoke another collaboration  $B$  in order to carry out some task. In the *TeleConsultation*, the doctor invokes the *test* collaboration while in the middle of the *voice* collaboration, as illustrated in Fig. 10.2 and Fig. 10.4.

We propose to model this situation in activity diagrams using two "stream" control flow arrows, one representing the request for service and the other representing the results returned<sup>5</sup>. If a collaboration  $A$ , invoking a collaboration  $B$ , suspends its own behavior while waiting for the results of  $B$ , then this collaboration invocation corresponds to the semantics of a procedure call, which is a case of strong sequencing. This is directly realizable if collaboration  $B$  is initiated and terminated by a single role that also participates in  $A$ . If this is not the case, additional coordination messages are needed to ensure strong sequencing.

Activity invocations should be checked to ensure that no invocation cycles are created (e.g.  $A$  invokes  $B$ , which in turns invokes  $C$ , which in turns invokes  $A$ ). These cycles may lead to deadlocks. Mixed initiatives may also appear if the invoking collaboration does not suspend its behavior. An example can be found in the *TeleConsultation* service. Assume that the behavior of *voice* is given by the sequence diagram in Fig. 10.7, and that the *test* collaboration is invoked when *connected* is true. Then a result from *test* may be received when callee has already sent a *hangup* message.

We note here that streaming pins allow information to be exchanged between concurrent activities and provides a general mechanism to model information exchange between collaborations that are executing in parallel, as has been demonstrated by Kraemer and Herrmann [KBH07]. This possibility is not elaborated here, but we remark that such interchange is directly realizable if localized within one role, as indicated in Fig. 10.5(b) and Fig. 10.5(c).

### 10.3.9 Related work on realizability

The realizability of specifications of reactive systems was first studied, in general terms, in [ALW89]. In the context of MSC-based specifications it was first considered in [AEY00], where the authors relate the problem of realizability to the notion of implied scenarios. They consider a specification given as a set of MSCs describing asynchronous interactions, and analyze it to check if it implies any non-specified MSC. Intuitively, a realizable specification does not contain implied scenarios. The authors propose two notions of realizability, depending on whether the realization is required to be deadlock-free (*safe realizability*) or not (*weak realizability*). This work was extended in [AEY05] to consider realizability of *bounded* HMSCs [AY99]. Reference [Loh03] extends in turn [AEY05] and provides some complexity results for a less restrictive class of HMSCs. Realizability of HMSCs with synchronous communication is considered in [UKM04]. The authors present a technique to detect implied scenarios from a specification describing both positive, as well as negative

<sup>5</sup>One may use several "stream" control flows for representing different types of results that could be obtained, such as normal and exceptional cases.

scenarios. The realizability notion considered in [AEY05] and [Loh03] does not allow adding data into messages or adding extra synchronization messages. This is seen as a very restrictive notion of realizability by some authors, who propose a notion of realizability where additional data can be incorporated into messages [MKS00, BM03, GMSZ06]. The authors of [MKS00] study safe realizability, with additional message contents, of regular (finite state) HMSCs with FIFO channels. This work is extended in [BM03], where the authors consider non-FIFO communication, and identify a subclass of HMSCs, so-called *coherent* HMSCs, which are safely realizable with additional message contents. However, checking whether an HMSC is coherent is in general hard. Reference [GMSZ06] discusses two classes of unbounded HMSCs. They claim that so-called *local-choice* HMSCs are always safely realizable with additional message contents<sup>6</sup>. A subclass of *local-choice* HMSCs that are safely realizable without additional message contents was studied in [HJ00].

Other authors have studied conditions for realizability of Compositional MSCs [MRW06] and pathologies in HMSCs [BAL97, H el01] and UML sequence diagrams [BBJ<sup>+</sup>05] that prevent their realization. None of these works discusses the nature of the realization problems.

### 10.3.10 System composition

In service engineering it is desirable that services can be modeled as independently as possible and then be composed in a modular way at design time and/or runtime. So far we have discussed the composition of a service defined as a collaboration among roles.

In general, a system may provide many different services, and many occurrences of the same service may be running concurrently. A collaboration, such as the TeleConsultation, may just be one of many collaborations to be realized in a given system, and a given system may run several TeleConsultations concurrently. In order to be executed in a system, each role must be bound to a component that can execute it, either statically or dynamically. In general a role may be assigned to many different components and each component may be assigned several different roles.

Using UML, the system structure and the binding of roles to components may be defined using composite classes with inner parts, as illustrated in Fig. 10.11. Clearly *system composition* has similarities with, but is not the same as the collaboration composition we have discussed so far. One will normally not define the complete system behavior explicitly as the choreography of an enclosing collaboration, but rather let it follow implicitly from the system structure and the binding of roles to the system components. System composition raises a number of issues and problems related to the composition and coordination of roles that will not be discussed further here. We only remark that, to a large extent, they may be handled outside the roles by additional coordination functionality in the components.

---

<sup>6</sup>Although their claim is true, the authors do not explain the proper format of message contents, as we discussed in Section 10.3.3

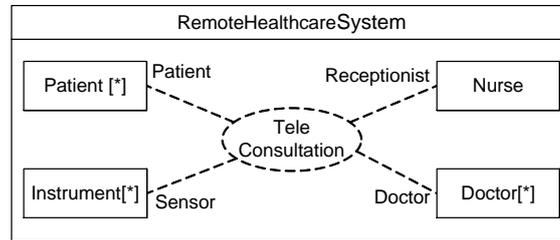


Figure 10.11: Actors with role bindings

## 10.4 Going from collaborations to component designs

We discuss in this section the possibility of deriving the behaviors of the components in a distributed design automatically from the behavior of a collaboration which we assume is given in the form described in Section 10.2. In general, each component will realize the behavior of at least one role identified in the collaboration. The behavior of each component will be given in terms of local actions to be performed and messages exchanged with the other components within the system. The term "protocol" [Boc90] denotes this behavior that must be satisfied for obtaining compatibility between the actions performed by the different system components.

During the past years, much research effort has been spent on the problem of deriving component behaviors from scenario-based specifications of the system behavior (for a survey, see [LDD06]). The system behavior is usually defined in terms of sequence diagrams or similar notations. In this context, most of the issues discussed in Section 10.3 must be addressed. As explained in Sections 10.2 and 10.3, collaborations provide useful structuring and composition mechanisms to describe and analyze the requirements that are the starting point for a systematic development process, and are therefore preferable to message sequence diagrams, which are at a relatively low level of abstraction. Nevertheless, message sequence scenarios can be derived from higher-level specifications in the form of activity diagrams or Use Case Maps [AHHC03], and then one could derive component behaviors in a second step. In the following, however, we do not follow that approach, but consider instead the direct derivation of component behavior from the specification of a collaboration.

### 10.4.1 Protocol derivation from service specification

Traditionally, an abstract view of a collaboration within a distributed system is called a service [BS80]. The specification of a service behavior describes actions that are executed at different "service access points" and their temporal order. A service access point corresponds to a role (or a participating component) in the definition of a collaboration. At this level of abstraction, the exchange of messages between the different roles is normally not shown. However, these messages are essential at

the level of the protocol specification which defines the behavior of each component in the distributed system. A body of work exists that describes algorithms for deriving a protocol specification from a given service specification. The service specification defines the temporal ordering of elementary actions that are associated with the components of the system. A protocol derivation algorithm, therefore, derives the necessary message exchanges between the different components in order to assure that the service actions are executed in an order consistent with the service specification. The initial work in [BG86] assumes that the service specification consists of elementary actions where the temporal ordering is defined by sequence, alternative and concurrency operators; the inclusion of message parameters for data transfer was added in [GB90]. In [KBK89], temporal orderings with regular recursion and sub-collaborations are introduced. General recursion is dealt with in [NHT98] and [KHB96], and the latter also deals with interruption.

The basic idea of these algorithms is to first identify for each sub-collaboration the roles (components) involved, and in particular the initiating and terminating roles (components). If two sub-collaborations should be executed in the temporal order of a strong sequence, then the protocol derivation algorithm introduces a **coordination message** from each terminating component of the first sub-collaboration to each initiating component of the second sub-collaboration. In many cases, the sub-collaborations have only a single initiating and terminating component; in this case a single coordination message is sufficient; and when both components are the same, no coordination message is required, since the sequencing can be enforced by the single component.

We note that most of these approaches only consider strong sequencing and assume that the service specification does not include a non-local choice. However, a non-local choice can be handled by introducing a conflict-resolution protocol between the components involved, for instance in the form of a circulating token, or by introducing priorities as suggested by Gouda [GY84], however, no general solution exists. Most derivation algorithms also assume that each component has separate input buffers for all its partners, and that there is no message overtaking.

## 10.4.2 Protocol derivation for Petri-nets

The problem of protocol derivation from service specifications has been studied also for the case that the service specification is given as a Petri net or some extended form of Petri nets [YOHT95, KG96]. This is of particular interest to us because the semantics of activity diagrams can be described naturally with Petri nets. An elementary action of a collaboration (described as an activity diagram) corresponds to a transition of the corresponding Petri net. Each transition of the Petri net is therefore associated with one of the roles of the collaboration. The Petri net tokens that pass from one transition to another represent coordination messages. Non-local choice remains a problem. We note that Petri net extensions have been considered, including pre- and post-conditions for transitions and variables that may be located at different components. The derived protocol includes mechanisms for checking non-local pre-conditions, updating of variables, and the control of access to shared resources [YEFBH03].

### 10.4.3 Semi-automatic designs of collaborations

From the above discussion, we get the following conclusion: Given the behavior of a collaboration described in terms of sub-collaborations and elementary actions and their allowed execution order, the problem of deriving the behavior of components that will realize this global behavior through message exchanges has been solved under the assumption that there is (a) no weak sequencing, and (b) no non-local choice or mixed initiatives.

For the many cases where these assumptions are not satisfied, further work is needed for finding appropriate solutions for the component behaviors. Concerning weak sequencing, a composition of sub-collaborations satisfying Propositions 10.4 or 10.8 in Section 10.3 has been shown to be directly realizable. In [CBB07] we provide proofs of this and also algorithms to check if the conditions for direct realizability of weak sequencing are satisfied or not.

We hope that the guidelines given in Section 10.3 will eventually lead to the semi-automated derivation of component behaviors from a service specification given in the form of sub-collaborations, elementary actions and their allowed execution order. This would lead to a semi-automatic process, where the designer has to choose domain-specific solutions to those problems for which no general solution is available, namely non-local choices and mixed initiatives.

## 10.5 Conclusions

In Section 10.1 we asked ourselves the following questions. Is it possible to model service behavior more completely? Can it be done in a structured way without revealing more interaction detail than necessary? Is it possible to support composition and to detect and remove realization problems? And is it possible to derive detailed implementations automatically from service models? We have shown here that a collaboration oriented approach based on UML 2 collaborations have potential to provide positive answers to several of these questions. In particular we have demonstrated how choreographies defined using activity diagrams can be used for service specification at a higher level than interactions and at the same time help to identify and resolve realization problems. To our best knowledge we are able to identify all the realization problems that have been reported in literature, many at the level of choreography, without needing to consider detailed interactions of sub-collaborations. We have also argued that our approach can be supported by tools that automatically generate correct implementations from service specifications. Evidence of this has been provided through several demonstrations by our groups and others.

## References

- [AEY00] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of message sequence charts. In *Proc. of 22nd Int. Conf. on Software Engineering (ICSE'00)*. ACM, 2000.

- [AEY05] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Realizability and verification of MSC graphs. *Theor. Comput. Sci.*, 331(1):97–114, 2005.
- [AHC03] Daniel Amyot, Xiangyang He, Yong He, and Dae Yong Cho. Generating scenarios from use case map specifications. In *Proc. of the 3rd Intl. Conf. on Quality Software (QSIC'03)*, pages 108–115, 2003. IEEE CS.
- [AHP96] Rajeev Alur, Gerard J. Holzmann, and Doron Peled. An analyzer for message sequence charts. *Software - Concepts and Tools*, 17(2):70–77, 1996.
- [ALW89] Martín Abadi, Leslie Lamport, and Pierre Wolper. Realizable and unrealizable specifications of reactive systems. In *Proc. of the 16th Intl. Colloquium on Automata, Languages and Programming (ICALP'89)*, pages 1–17, 1989. Springer-Verlag.
- [Amy03] Daniel Amyot. Introduction to the user requirements notation: learning by example. *Computer Networks*, 42(3):285–301, 2003.
- [AY99] Rajeev Alur and Mihalis Yannakakis. Model checking of message sequence charts. In *Proc. 10th Intl. Conf. on Concurrency Theory (CONCUR'99)*, volume 1664 of *LNCS*, pages 114–129. Springer, 1999.
- [BAL97] Hanene Ben-Abdallah and Stefan Leue. Syntactic detection of process divergence and non-local choice in message sequence charts. In *Proc. 2nd Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, 1997.
- [BBJ<sup>+</sup>05] Paul Baker, Paul Bristow, Clive Jervis, David King, Robert Thomson, Bill Mitchell, and Simon Burton. Detecting and resolving semantic pathologies in uml sequence diagrams. In *Proc. of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-13)*, pages 50–59, 2005. ACM Press.
- [BG86] Gregor Bochmann and Reinhard Gotzhein. Deriving protocol specifications from service specifications. In *Proc. of ACM SIGCOMM Symposium*, pages 148–156, 1986.
- [BKM07] Manfred Broy, Ingolf H. Krüger, and Michael Meisinger. A formal model of services. *ACM Trans. Softw. Eng. Methodol.*, 16(1):5, 2007.
- [BM03] Nicolas Baudru and Rémi Morin. Safe implementability of regular message sequence chart specifications. In *Proc. of ACIS 4th Intl. Conf. on Soft. Eng., Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'03)*, pages 210–217, 2003.

- [BM05] Rolv Bræk and Geir Melby. *Model Driven Service Engineering*, chapter of Model-driven Software Development. Volume II of Research and Practice in Software Engineering. Springer, 2005.
- [Boc78] Gregor Bochmann. Finite state description of communication protocols. *Computer Networks*, 2:361–372, 1978.
- [Boc90] Gregor von Bochmann. Protocol specification for OSI. *Computer Networks and ISDN Systems*, 18(3):167–184, 1989/1990.
- [Boc00] Gregor v. Bochmann. Activity nets: A UML profile for modeling workflow and business processes. Technical report, SITE, University of Ottawa, 2000.
- [Bræ79] Rolv Bræk. Unified system modeling and implementation. In *International Switching Symposium (ISS)*. ISS Committee, 1979.
- [Bræ99] Rolv Bræk. Using roles with types and objects for service development. In *IFIP TC6 WG6.7 Fifth International Conference on Intelligence in Networks (SMARTNET)*, volume 160 of *IFIP Conference Proceedings*, pages 265–278, Pathumthani, Thailand, 1999. Kluwer.
- [BS80] Gregor Bochmann and Carl A. Sunshine. Formal methods in communication protocol design. *IEEE Trans. on Communications*, 28(4), April 1980.
- [CB06a] Humberto N. Castejón and Rolv Bræk. A collaboration-based approach to service specification and detection of implied scenarios. In *Proc. of 5th int. workshop on Scenarios and state machines: models, algorithms and tools (SCESM'06)*. ACM Press, 2006.
- [CB06b] Humberto N. Castejón and Rolv Bræk. Formalizing collaboration goal sequences for service choreography. In *Proc. of the 26th IFIP WG 6.1 Intl. Conf. on Formal Methods for Networked and Distributed Systems (FORTE'06)*, volume 4229 of *LNCS*, pages 275–291, Paris, France, September 2006. Springer-Verlag.
- [CBB07] Humberto N. Castejón, Gregor Bochmann, and Rolv Bræk. Investigating the realizability of collaboration-based service specifications. Technical report, Avantel 3/2007 ISSN 1503-4097, NTNU, 2007.
- [CKS05] Chien-An Chen, Sara Kalvala, and Jane Sinclair. Race conditions in message sequence charts. In *Proc. of 3rd Asian Symposium on Programming Languages and Systems (APLAS'05)*, volume 3780 of *LNCS*, pages 195–211. Springer, 2005.
- [Erl05] Thomas Erl. *Service Oriented Architecture: Concepts, Technology and Design*. Number ISBN 0-13-185858-0. Prentice Hall, 2005.

- [FK01] Kathi Fisler and Shriram Krishnamurthi. Modular verification of collaboration-based software designs. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 152–163. ACM Press, 2001.
- [GMSZ06] Blaise Genest, Anca Muscholl, Helmut Seidl, and Marc Zeitoun. Infinite-state high-level mscs: Model-checking and realizability. *J. Comput. Syst. Sci.*, 72(4):617–647, 2006.
- [GB90] Reinhard Gotzhein and Gregor von Bochmann. Deriving protocol specifications from service specifications including parameters. *ACM Trans. Comput. Syst.*, 8(4):255–283, 1990.
- [GY84] Mohamed G. Gouda and Yao-Tin Yu. Synthesis of communicating finite state machines with guaranteed progress. *IEEE Trans. on Communications*, Com-32(7):779–788, July 1984.
- [Hél01] Loïc Hélouët. Some pathological message sequence charts, and how to detect them. In *10th Intl. SDL Forum*, volume 2078 of *LNCS*, pages 348–364. Springer-Verlag, 2001.
- [HJ00] Loïc Hélouët and Claude Jard. Conditions for synthesis of communicating automata from HMSCs. In *Proc. of 5th Intl. Workshop on Formal Methods for Industrial Critical Systems (FMICS'00)*. GMD FOKUS, 2000.
- [ITU98] ITU-T. Message sequence charts (MSC). Technical report, Recommendation Z.120. International Telecommunications Union, 1998.
- [ITU00] ITU-T. Specification and description language (SDL). Technical report, Recommendation Z.100. International Telecommunications Union, 2000.
- [KBH07] Frank Alexander Kraemer, Rolv Bræk, and Peter Herrmann. Synthesizing components with sessions from collaboration-oriented service specifications. In *Procs. of 13th SDL Forum*, volume 4745 of *LNCS*, pages 166–185, Paris, 2007. Springer.
- [KG96] Hakim Kahlouche and Jean-Jacques Girardot. A stepwise refinement based approach for synthesizing protocol specifications in an interpreted petri net model. In *Proc. of INFOCOM'96*, pages 1165–1173, 1996.
- [KH06] Frank Alexander Kraemer and Peter Herrmann. Service specification by composition of collaborations—an example. In *Procs. of 2nd Intl. Workshop on Service Composition (SERCOMP'06)*, pages 129–133, 2006. IEEE CS.

- [KHB96] Christian Kant, Teruo Higashino, and Gregor von Bochmann. Deriving protocol specifications from service specifications written in LOTOS. *Distributed Computing*, 10(1):29–47, 1996.
- [KM03] Ingolf H. Krüger and Reena Mathew. Component synthesis from service specifications. In *Intl. Dagstuhl Workshop on Scenarios: Models, Transformations and Tools*, volume 3466 of *LNCS*, pages 255–277. Springer, 2003.
- [Krü03] Ingolf Krüger. Capturing overlapping, triggered, and preemptive collaborations using MSCs. In *Proc. of the 6th Intl. Conf. on Fundamental Approaches to Software Engineering (FASE'03)*, volume 2621 of *LNCS*, pages 387–402. Springer, 2003.
- [KBK89] F. Khendek, G. von Bochmann, and C. Kant. New results on deriving protocol specifications from service specifications. *SIGCOMM Comput. Commun. Review*, 19(4):136–145, 1989.
- [KZ05] Ferhat Khendek and Xiao Jun Zhang. From msc to sdl: Overview and an application to the autonomous shuttle transport system. In Springer, editor, *2003 Dagstuhl Workshop on Scenarios: Models, Transformations and Tools*, volume 3466 of *LNCS*, pages 228–254, 2005.
- [Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
- [LDD06] Hongzhi Liang, Juergen Dingel, and Zinovy Diskin. A comparative survey of scenario-based to state-based model synthesis approaches. In *Proc. of the 5th Intl. workshop on Scenarios and State Machines: models, algorithms, and tools (SCESM'06)*, pages 5–12, 2006. ACM Press.
- [Loh03] Markus Lohrey. Realizability of high-level message sequence charts: closing the gaps. *Theor. Comput. Sci.*, 309(1-3):529–554, 2003.
- [MGR05] Arjan J. Mooij, Nicolae Goga, and Judi Romijn. Non-local choice and beyond: Intricacies of MSC choice nodes. In *Proc. Intl. Conf. on Fundamental Approaches to Soft. Eng. (FASE'05)*, volume 3442 of *LNCS*. Springer, 2005.
- [Mit05] Bill Mitchell. Resolving race conditions in asynchronous partial order scenarios. *IEEE Trans. Softw. Eng.*, 31(9):767–784, 2005.
- [MKS00] Madhavan Mukund, K. Narayan Kumar, and Milind A. Sohoni. Synthesizing distributed finite-state systems from MSCs. In *Proc. 11th Intl. Conf. on Concurrency Theory (CONCUR'00)*, volume 1877 of *LNCS*, pages 521–535. Springer, 2000.

- [MRW06] Arjan Mooij, Judi Romijn, and Wieger Wesselink. Realizability criteria for compositional msc. In *Proc. of 11th Intl. Conf. on Algebraic Methodology and Software Technology (AMAST'06)*, volume 4019 of *LNCS*. Springer, 2006.
- [NHT98] A. Nakata, T. Higashino, and K. Taniguchi. Protocol synthesis from context-free processes using event structures. In *Proc. of the 5th Intl. Conf. on Real-Time Computing Systems and Applications (RTCSA '98)*, page 173. IEEE CS, 1998.
- [OMG07] Object Management Group (OMG). *UML 2.1.1 Superstructure Spec.*, February 2007. Accesible at <http://www.omg.org/cgi-bin/apps/doc?ptc/06-04-02.pdf>.
- [RAB<sup>+</sup>92] T. Reenskaug, E.P. Andersen, A.J. Berre, A. Hurlen, A. Landmark, O.A. Lehne, E. Nordhagen, E. Ness-Ulseth, G. Oftedal, A.L. Skaar, and P. Stenslet. OORASS: Seamless support for the creation and maintenance of object-oriented systems. *Journal of Object-oriented Programming*, 5(6):27–41, 1992.
- [RT03] Abhik Roychoudhury and P. S. Thiagarajan. Communicating transaction processes: An msc-based model of computation for reactive embedded systems. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 789–818, 2003.
- [RWL96] Trygve Reenskaug, P. Wold, and O. A. Lehne. *Working with Objects: The OOram Software Engineering Method*. Prentice Hall, 1996.
- [San00] Richard Sanders. Implementing from SDL. *Teletronikk*, 96(4), 2000.
- [San07] Richard Sanders. *Collaborations, Semantic Interfaces and Service Goals – a new way forward for Service Engineering*. PhD thesis, Department of Telematics, Norwegian Univ. Science and Technology, Trondheim, Norway, March 2007.
- [SBBA05] Richard Torbjørn Sanders, Rolv Bræk, Gregor von Bochmann, and Daniel Amyot. Service discovery and component reuse with semantic interfaces. In *Proc. 12th SDL Forum*, volume 3530 of *LNCS*, Grimstad, Norway, June 2005. Springer.
- [SCKB05] Richard Torbjørn Sanders, Humberto N. Castejón, Frank Alexander Kraemer, and Rolv Bræk. Using UML 2.0 collaborations for compositional service specification. In Lionel Briand and Clay Williams, editors, *ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 3713 of *LNCS*, pages 460–475, Montego Bay, Jamaica, October 2005. Springer-Verlag.

- [UKM04] Sebastián Uchitel, Jeff Kramer, and Jeff Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Trans. Softw. Eng. Methodol.*, 13(1):37–85, 2004.
- [Whi07] Jon Whittle. Precise specification of use case scenarios. In *10th Intl. Conf. on Fundamental Approaches to Software Engineering (FASE'07)*, volume 4422 of *LNCS*, pages 170–184, 2007.
- [WvdAD<sup>+</sup>05] Petia Wohed, Wil M. P. van der Aalst, Marlon Dumas, Arthur H. M. ter Hofstede, and Nick Russell. Pattern-based analysis of the control-flow perspective of UML activity diagrams. In *24th Intl. Conf. on Conceptual Modeling (ER'05)*, volume 3716 of *LNCS*, pages 63–78. Springer, 2005.
- [YEFBH03] Hirozumi Yamaguchi, Khaled El-Fakih, Gregor von Bochmann, and Teruo Higashino. Protocol synthesis and re-synthesis with optimal allocation of resources based on extended petri nets. *Distributed Computing*, 16(1):21–35, 2003.
- [YOHT95] H. Yamaguchi, K. Okano, T. Higashino, and K. Taniguchi. Synthesis of protocol entities specifications from service specifications in a petri net model with registers. In *Proc. of the 15th Intl. Conf. on Distributed Computing Systems (ICDCS'95)*, page 510. IEEE Computer Society, 1995.

## Appendix 10.A Remarks to the Paper

This appendix did not appear in the original publication. It is included here to discuss some issues regarding the use of activity diagrams that were discovered after the publication time.

In this paper we suggested using UML activity diagrams to model collaboration choreographies. An activity diagram describing a choreography consists of control nodes, as well as call behavior actions allowing to directly invoke (collaboration) behaviors elsewhere described. Since in UML both interactions as well as activities are a kind of behavior, both of them can be invoked from a call behavior action. This means that, in principle, call behavior actions can be used in a choreography graph to invoke the behavior of both elementary collaborations, described by means of UML sequence diagrams (a kind of interaction diagram), as well as the behavior of composite collaborations, described by means of UML activity diagrams (specifying the choreography of their sub-collaborations). This is what we want. However, an obstacle arises in order to support the weak sequencing of collaborations, as well as the invocation of collaborations by other ones (i.e. invocation compositions). In both cases we need a call behavior action (invoking the behavior of one collaboration) to be started while another one is still running. That is, we need a call behavior action

to offer tokens while its invoked behavior is still running (or suspended, but not finished). When the invoked behavior is described by an activity, this is perfectly possible. In that case we could:

- (i) Provide the activity with *streaming* parameters, which allow the behavior described by the activity to accept input values, and provide output values, while in the middle of its execution (i.e. not only at the beginning and at the end of the execution).
- (ii) Provide the call behavior action invoking that activity with pins, which would be associated with the activity's parameters. Pins connected to output streaming parameters would then receive a token whenever a value was produced via the corresponding parameter.

Unfortunately, streaming parameters are only defined for activities (and more specifically for the package of CompleteActivities – see page 394 of [OMG07]). UML interactions can have parameters, but not with streaming capabilities. Therefore, if the behavior invoked by a call behavior action is described using an interaction (i.e. as in the case of elementary collaborations), the call behavior action may neither receive nor offer tokens while its invoked behavior is being executed. This means that we cannot have weak sequencing semantics and invocation compositions in choreographies described with UML activity diagrams when the behaviors of the collaborations are described using interaction diagrams. Not if we want to be totally compliant with the UML semantics for activity diagrams. This is paradoxical. On the one hand, the UML standard allows to invoke behaviors described using interaction diagrams from activity diagrams. On the other hand, the standard imposes (implicitly) a strong sequence semantics, while weak sequencing is the default semantics for interaction diagrams. We believe the UML standard should be more clear and precise on this issue. If interactions may be invoked from activities, weak sequence semantics should be possible, either by direct support, or by allowing sequence diagrams to have streaming parameters. Otherwise activities should not be allowed to invoke behaviors described using interactions.

As a result, we propose to use the syntax of activity diagrams, but with the semantics described in Paper 6.

We note that it could maybe be possible to define a UML profile where interactions are extended so that they can have streaming parameters. This option should however be carefully studied.



---

## Paper 6

### **Investigating the realizability of collaboration-based service specifications.**

By Humberto Nicolás Castejón, Gregor von Bochmann and Rolv Bræk.

Published as Technical Report Avantel 3/2007, ISSN 1503-4097, NTNU, September 2007.

#### **Notes**

A minor correction has been made to Property 11.3.1. This correction is explained in an endnote.



# Investigating the Realizability of Collaboration-based Service Specifications

Humberto Nicolás Castejón<sup>a</sup>, Gregor von Bochmann<sup>b</sup> and  
Rolv Bræk<sup>a</sup>

<sup>a</sup>*Dept. of Telematics, Norwegian University of Science and Technology, Trondheim,  
Norway*

`{humberto.castejon, rolv.braek}@item.ntnu.no`

<sup>b</sup>*School of Inf. Technology and Engineering, University of Ottawa, Ottawa, Canada*  
`bochmann@site.uottawa.ca`

## Abstract

This report is concerned with compositional specification of services using UML 2 collaborations, activity and interaction diagrams. It provides formal syntax and semantics for so-called choreography graphs, used to describe the complete behavior of composite collaborations. It then addresses the problem of realizability: given a global specification, can we construct a set of communicating state machines whose joint behavior is precisely the specified one? We approach the problem by looking at how collaboration behaviors may be composed using UML activity diagrams-based choreographies. We classify realizability problems from the point of view of each composition operator, and discuss their nature and possible solutions. This brings a new look at already known problems. We show that given some conditions, some problems can already be detected at an abstract collaboration level, without needing to look into detailed interactions. We present algorithms to detect some of the discussed problems.

## 11.1 Introduction

For several decades now it has been common practice to specify and design reactive systems in terms of loosely coupled components modeled as communicating state machines [Boc78, Bræk79], using languages such as SDL [IT00] and UML [OMG07]. This has helped to substantially improve quality and modularity, mainly by providing means to define complex, reactive behavior precisely in a way that is understandable to humans and suitable for formal analysis as well as automatic generation of executable code.

However, there is a fundamental problem. In many cases, application/service behavior is not performed by a single component, but by several collaborating components. This is referred to as the “crosscutting” nature of services by different authors [RGG01, FK01, KM03]. Often each component takes part in several different services, so in general, the behavior of services is composed from partial component behaviors, while component behaviors are composed from partial service behaviors. By structuring according to components, the behavior of each individual component can be defined precisely and completely, while the behavior of a service is fragmented. In order to model the global behavior of a service more explicitly one needs an orthogonal view where the collaborative behavior is in focus. Interaction sequences such as MSC [IT99], and UML Sequence diagrams [OMG07] are commonly used for this purpose, but normally only to describe typical/important use cases and not complete behaviors. Normally when using interaction sequences, it is very cumbersome to define all the intended scenarios. In addition, there are problems related to the *realizability* of interaction scenarios, i.e. finding a set of local component behaviors whose joint execution leads precisely to the global behavior specified in the scenarios. The realizability of MSC-based specifications has been extensively studied by different authors (e.g. [AEY00, UKM04, AEY05, BS05]). Conditions for realizability have been proposed for HMSCs [HJ00] and Compositional MSCs [MRW06], as well as restricted classes of HMSCs that are known to be always realizable [GMSZ06]. Some authors have studied pathologies in HMSCs [BAL97, H el01] that prevent their realization. Other authors have considered realizability notions that allow additional message contents [BM03, GMSZ06].

A promising step forward is to adopt a *collaboration-oriented* approach, where the main structuring units are collaborations. This is made practically possible by the new UML 2 collaboration concept [OMG07]. The underlying ideas, however, date back to before the UML era [RAB<sup>+</sup>92, RWL96]. Collaborations model the concept of a service very nicely. They define a structure of partial object behaviors, called roles, and enable a precise definition of the service behavior using interaction diagrams, activity diagrams and state machines as explained in [SCKB05, CB06a, CB06b]. They also provide a way to compose services by means of collaboration uses and to bind roles to components. In this way, UML 2 collaborations directly support (crosscutting) service modeling and service composition. As we shall see in the following, this opens many interesting opportunities. Figure 11.1 illustrates the main models involved in the collaboration oriented approach being discussed in the following:

- Service models are used to formally specify and document services. Collaborations provide a structural framework for these models that can embody both the role behaviors and the interactions between the roles needed to fulfill the service.
- Design models are used to formally specify and document system structure and components realizing the services. They are expressed in terms of com-

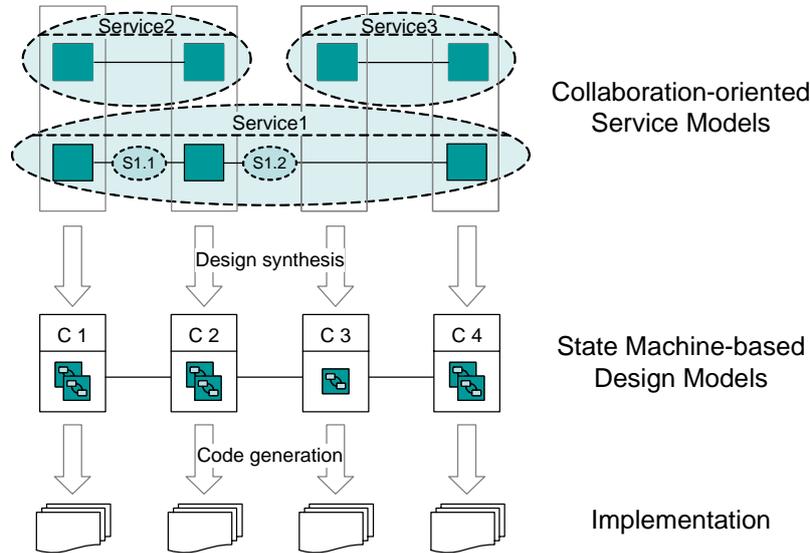


Figure 11.1: Collaboration oriented development

municating state machines, using UML 2 active objects. Each of these will realize one or more collaboration roles.

- Implementations are executable code automatically generated from the design models.

This paper is concerned with the crucial first steps of expressing service models using UML 2 collaborations and deriving well-formed design models expressed as communicating state machines. The ensuing steps from design component models to implementations and dynamic deployment on service platforms can be solved in different ways, see for instance [San00, BM05], and are not discussed further here.

An important property of collaborations is that it is possible and convenient to compose/decompose collaborations structurally into sub-collaborations, by means of collaboration uses. These refer to separately defined collaborations and provide a mechanism for collaboration reuse. In order to define the behavior of collaborations, we have found it useful to distinguish between the behavior of *composite collaborations* and *elementary collaborations* (collaborations that are not further decomposed into sub-collaborations). The elementary collaborations that result from the decomposition process are often quite simple, reusable and possible to define completely using interaction sequences. Binary collaborations can in many cases be associated with interfaces and their sub-collaborations with features of the interface. The question then is how to define the overall behavior of composite collaborations in terms of sub-collaboration behaviors? In the web service domain this kind of behavior is called “choreography” [Erl05], a term we will use in the following. Several notations may be used to define the choreography of sub-collaborations (i.e. their

global execution ordering). We have found UML 2 Activity diagrams a good candidate, as they provide many of the composition operators needed for the purpose. While HMSCs normally describe collection of scenarios, and therefore represent incomplete and existential behavior, our choreographies describe the exact behavior of a service according to the service designer's intentions (i.e. the service should behave exactly as described, no other behaviors are allowed). The local behavior for a given component of the choreography can be obtained by applying the ordering defined by the choreography's activity diagram to the role behaviors bound to the component in question.

We say that a choreography is directly realizable if the joint execution of the local behaviors of all components leads precisely to the global behavior specified by the choreography. Note that some choreographies that are not directly realizable may still be realized by adding extra coordination messages or additional data in messages. We consider these measures as solutions to realization problems, which could be adopted by the designer depending on the context and service domain. Note also that the realizability of a choreography depends not only on the ordering defined by the activity diagram of the choreography, but also on the characteristics of the underlying communication service used for the transmission of messages. Important characteristics of the communication service are the type of transmission channels, and the type and number of input buffers of each component. We assume there is no message loss, and distinguish between channels with out-of-order delivery (i.e. messages sent from a given source to a given destination may be received in a different order than they were sent) and channels with in-order delivery. Components may have either a single input FIFO buffer (i.e. one buffer for all received messages) or separate input FIFO buffers (i.e. one buffer for messages received from each different peer).

In the rest of the paper we study the direct realizability of a choreography from the point of view of the operators used to compose the sub-collaborations. In our discussion we assume that each sub-collaboration of a choreography is directly realizable. Then, for each composition operator (i.e. sequential, alternative, parallel, interruption) we study the problems that can lead to difficulties of realization. We investigate the actual nature of these problems and discuss possible solutions to prevent or remedy them.

### 11.1.1 Outline

The paper is structured as follows. In Section 11.2 the proposed service modeling approach is illustrated with help of an example, and the syntax and semantics of choreographies is presented in Section 11.3. The problem of realizability of choreographies is discussed in Section 11.4. Section 11.5 presents a set of algorithms to detect some of the realizability problems discussed in the previous section. We finally conclude with Section 11.6.

## 11.2 Service Specification Approach: An Example

We exemplify our service specification approach by means of a simple shuttle service (inspired by a case study from [UKM04]) in which one vehicle transports one passenger at a time between two terminals. Figure 11.2 depicts this service as a UML 2.0 collaboration. This collaboration identifies three roles, namely *P* (Passenger), *T* (Terminal) and *V* (Vehicle); as well as seven sub-collaborations representing interfaces and features of the service. These sub-collaborations are specified as UML collaboration-uses, whose roles are bound to the *ShuttleService*'s roles (e.g. *BuyTicket*'s role  $T_{bt}$  is bound to *ShuttleService*'s role *T*). For the sake of clarity, in the following we will refer to *P*, *T* and *V* as service-roles, and to  $T_{bt}$ ,  $T_d$  and the like as sub-roles (of *T*, *P* or *V*). The *ShuttleService*'s sub-collaborations have been identified from the following service requirements. In order to travel, a passenger must buy a ticket at one of the terminals (collaboration-use *BuyTicket*). When this happens, if the vehicle is waiting at the terminal, the departure gate is indicated, and the passenger can enter the vehicle (*EnterVehicle*). The terminal then dispatches the vehicle (*VehDeparture*) and, after arriving at the second terminal (*VehArrival*), the passenger disembarks (*ExitVehicle*). If the vehicle is not at the terminal where the passenger buys the ticket, that terminal requests the vehicle from the other terminal (*ReqVehicle*), which dispatches the vehicle towards the requesting terminal. When the vehicle arrives, the departure gate is displayed and the service continues as explained before.

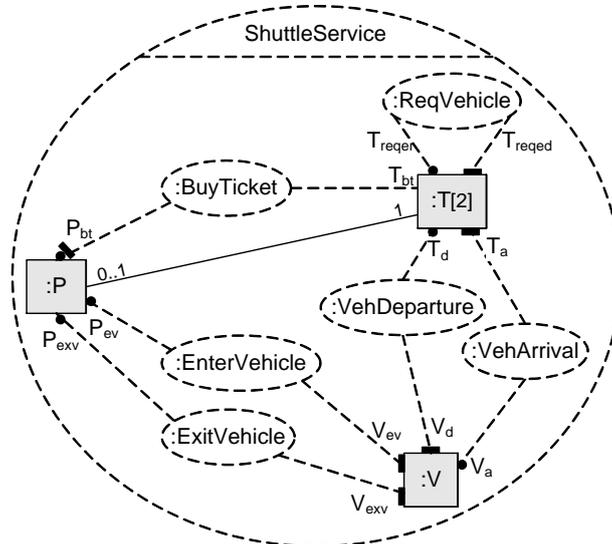


Figure 11.2: UML collaboration for the *ShuttleService*

The complete and exact behavior of each elementary sub-collaboration is de-

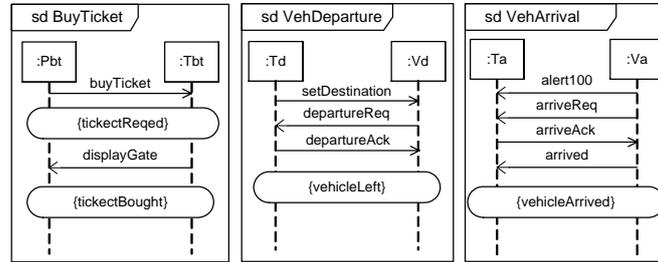


Figure 11.3: Sequence diagrams describing some elementary collaborations of *ShuttleService*

scribed by means of sequence diagrams. Figure 11.3 shows the sequence diagrams describing *BuyTicket*, *VehDeparture* and *VehArrival*.

What remains is to specify the overall cross-cutting behavior of the *ShuttleService* collaboration, that is, the choreography describing how its sub-collaborations are ordered and interact. We use UML 2 activity diagrams to describe the choreography of collaborations. They capture the liveness aspects of composite service collaborations by describing the execution order of their sub-collaborations. The choreography for *ShuttleService* is depicted in Fig. 11.4. Note that we have annotated each activity with a pictorial representation of the collaboration-use the activity refers to.

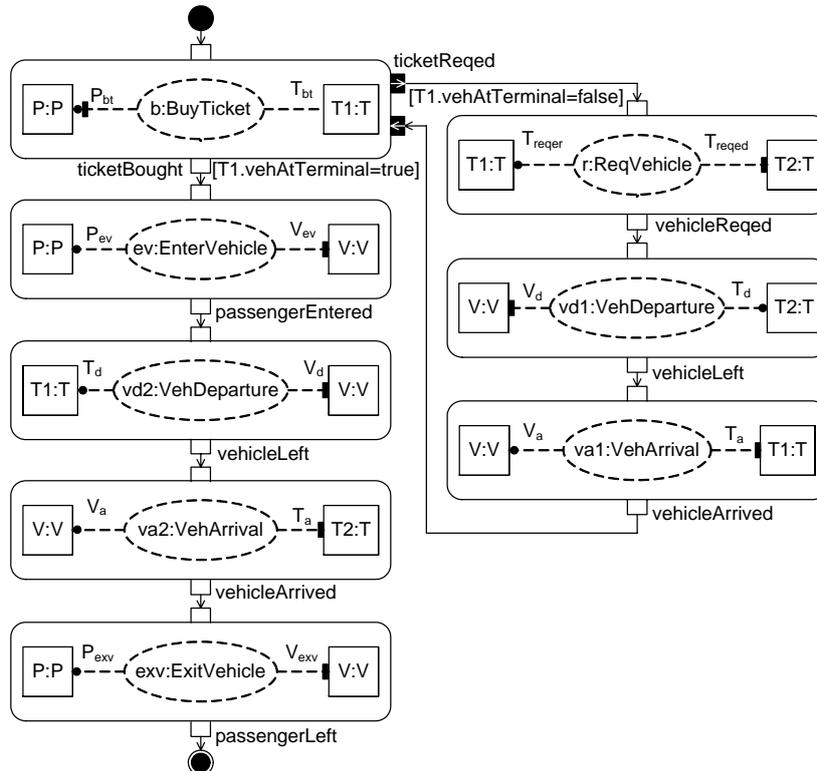


Figure 11.4: Choreography of *ShuttleService*

## 11.3 Syntax and Semantics of Choreographies

A choreography graph specifies the full behavior of a composite collaboration by defining the global execution ordering of its sub-collaborations. We consider here choreography graphs represented by means of UML activity diagrams, and assume that the sub-collaborations referred to by the choreography are elementary collaborations whose behavior is described by means of UML sequence diagrams. In the following we describe the syntax and semantics of choreography graphs. Before that we describe the syntax and semantics of sequence diagrams.

### 11.3.1 Syntax and Semantics for Sequence Diagrams

We consider here a restricted version of UML 2 sequence diagrams, which we use to specify the behavior of elementary collaborations. We used the syntax proposed in the UML standard, but provide a semantics based on partially ordered sets (posets). In the following we define and provide a semantics to basic sequence diagrams. Thereafter, we focus on composite sequence diagrams.

**Definition 11.1 (Basic Sequence Diagram).** A basic sequence diagram defines a labeled directed acyclic graph that can be described by a tuple  $bSD = (E, <_e, <_m, \mathcal{P}, \mathcal{M}, \Sigma, loc, lbl, rcv, snd)$ , where:

- $E = S \cup R \cup \Phi$  is a set of events partitioned into sending events ( $S$ ), receiving events ( $R$ ) and predicate events ( $\Phi$ )
- $\mathcal{P}$  is a set of lifelines
- $\mathcal{M}$  is a set of messages
- $\Sigma = \Sigma_c \cup \Sigma_p$  is a set of communication actions ( $\Sigma_c$ ) and predicates ( $\Sigma_p$ ). Elements of  $\Sigma_c$  are of the form  $\langle !m, p, q \rangle$  or  $\langle ?m, p, q \rangle$ , with  $p, q \in \mathcal{P}, m \in \mathcal{M}$ . We read  $\langle !m, p, q \rangle$  as “ $p$  sends message  $m$  to  $q$ ”, and  $\langle ?m, p, q \rangle$  as “ $p$  receives message  $m$  from  $q$ ”
- $loc : E \rightarrow \mathcal{P}$  is a mapping that associates each event with a lifeline
- $lbl : E \rightarrow \Sigma_c \cup \Sigma_p$  is a labeling function associating each event with a communicating action or a predicate
- $rcv : S \rightarrow R$  and  $snd : R \rightarrow S$  are bijective functions that respectively match each sending event with its corresponding receiving event, and each receiving event with its corresponding sending event. We have  $snd \equiv rcv^{-1}$
- $<_e \subseteq E \times E$  is an acyclic relation between events, called the *visual* order, that satisfies:
 
$$<_e = \left( \bigcup_{p \in \mathcal{P}} <_p \right) \cup \{ (s, rcv(s)) : s \in S \},$$
 where for each  $p \in \mathcal{P}, <_p$  is a total order (i.e. an antisymmetric, transitive and total binary relation) on  $E_p = \{ e \in E : loc(e) = p \}$  (i.e. events on a lifeline are totally ordered)

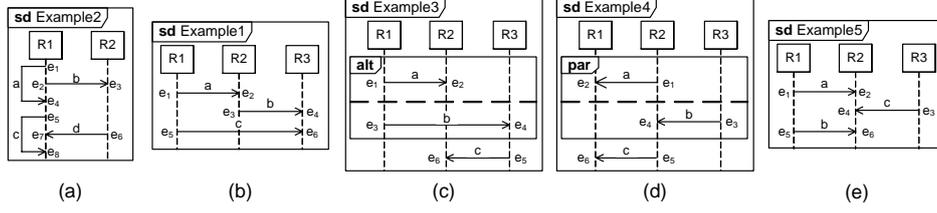


Figure 11.5: Some basic sequence diagrams (with conflicts)

- $<_m \subseteq S \times S$  is a total order on sending events, called the *message order*. To construct  $<_m$  we imagine a vertical line  $L$  aligned with the sequence diagram and project each sending event on  $L$ . We say  $s_1 <_m s_2$  if  $s_1$  is located on  $L$  higher than  $s_2$ . To ensure that  $<_m$  is a total order, two sending events cannot be drawn on the same imaginary horizontal line.

For any sequence diagram we require the following property:

**Property 11.3.1 (Non-crossing messages).** Messages do not graphically cross each other. That is:

- Messages exchanged between different lifelines are represented by horizontal arrows.
- Messages sent by a lifeline  $p \in \mathcal{P}$  to itself (*self-receiving* messages) satisfy this condition:  $\forall e_1 = \langle !m, p, p \rangle, e_2 = rcv(e_1), \nexists e$  such that  $e_1 <_p e <_p e_2$ <sup>1</sup>.

In this property, condition (i) is required to avoid cycles that could lead to causal relations where a sending message is indirectly dependent on its receiving event. Condition (ii) is required because, in general, a specification with an order  $\langle !m, p, p \rangle <_p e_1 <_p \dots <_p e_n <_p rcv(\langle !m, p, p \rangle)$  is not directly realizable. As an example consider the diagram in Fig. 11.5(a). Here  $R1$  first sends message  $a$  to itself and before receiving it, it sends message  $b$  to  $R2$ . This specification does not satisfy condition (ii) in Property 11.3.1, since we have  $e_1 = \langle !a, R1, R1 \rangle <_p e_2 <_p e_4 = \langle ?a, R1, R1 \rangle$ . It is easy to see that, in general<sup>1</sup>, this order cannot be guaranteed by any directly realized system, since the reception of message  $a$  cannot be controlled, and may therefore happen before the sending of message  $b$ . In the lower part of the diagram a similar situation happens between messages  $c$  and  $d$ .

Both the visual order ( $<_e$ ) and the message order ( $<_m$ ) can be obtained from the graphical representation of the sequence diagram. The visual order captures the order of events on each lifeline, as well as the order induced by message transmissions. The message order captures the vertical order of messages. For example, in Fig. 11.5(b) the visual order is  $<_e = \{(e_1, e_5), (e_2, e_3), (e_4, e_6), (e_1, e_2), (e_3, e_4), (e_5, e_6)\}$ , while the message order is  $<_m = \{(e_1, e_3), (e_3, e_5), (e_1, e_5)\}$ , which indirectly describes the vertical order between messages  $a$ ,  $b$  and  $c$ .

<sup>1</sup>This order can only be guaranteed if the design is such that the two sending events  $e_1$  and  $e_2$  are performed in an atomic transition.

The ordering of events dictated by the visual order does not necessarily reflect the semantics of the sequence diagram. The visual order may impose ordering between events that cannot be guaranteed in a directly realized system. To describe the semantics of sequence diagrams we use a weaker partial order, called the *causal partial order* (or *causal poset*), which only orders two events if they necessarily happen in that order in any execution of the directly realized system. The causal order takes into account the particularities of the communication architecture to be used in the realized system, in particular whether channels with out-of-order delivery or channels with in-order delivery are to be used.

**Definition 11.2.** A **causal order for channels with out-of-order (or non-fifo) delivery** ( $\prec_{\text{nf}}$ ) is the reflexive-transitive closure of the immediate precedence relation  $<_{\text{nf}}$  (i.e.  $\prec_{\text{nf}} = (<_{\text{nf}})^*$ ), where  $e <_{\text{nf}} e'$  if any of the following holds:

- $e \in S \wedge \text{rcv}(e) = e'$  (i.e.  $e'$  is the receiving event associated to the sending event  $e$ )
- $e' \in S \cup \Phi \wedge e <_p e'$ , for  $p \in \mathcal{P}$  (i.e.  $e'$  is a sending event, or a predicate event, and  $e$  is a visual predecessor of  $e'$  on the same lifeline)

The above definition reflects that in a channel with out-of-order delivery, the order in which messages are received (at a certain input buffer) may not be the same as the order in which message were sent (i.e. messages may overtake each other on the channel). With out-of-order delivery channels, only two classes of event orderings can be guaranteed. First, a receiving event will always happen after its sending event. Second, a system component may always control when to perform a sending event. This means that a sending event  $s$  will always happen after any other events that precede  $s$  on the same lifeline have been performed.

**Definition 11.3.** A **causal order for channels with in-order (or fifo) delivery** ( $\prec_{\text{f}}$ ) is the reflexive-transitive closure of the immediate precedence relation  $<_{\text{f}}$  (i.e.  $\prec_{\text{f}} = (<_{\text{f}})^*$ ), where  $e <_{\text{f}} e'$  if one of the following holds:

- $e <_{\text{nf}} e'$  (i.e.  $e$  is an immediate predecessor of  $e'$  under out-of-order semantics)
- $e, e' \in R \wedge e <_p e' \wedge \text{snd}(e) <_q \text{snd}(e')$ , for  $p, q \in \mathcal{P}$  (i.e. messages cannot overtake each other on the channel, so any two messages sent by a system component to another will arrive in the correct order)

We can now define two semantic functions that assign a  $\Sigma$ -labeled causal poset to a basic sequence diagram.

**Definition 11.4.** The semantics of a basic sequence diagram  $bSD = (E, <_v, <_m, \mathcal{P}, \mathcal{M}, \Sigma, \text{loc}, \text{lbl}, \text{rcv}, \text{snd})$  can be described with a semantic function  $\llbracket - \rrbracket_{\text{bSD}}^x$ , with  $x \in \{\text{nf}, \text{f}\}$ , such that

- $\llbracket bSD \rrbracket_{\text{bSD}}^{\text{nf}} = (E, \prec_{\text{nf}}, \text{lbl})$ , in the presence of out-of-order delivery channels
- $\llbracket bSD \rrbracket_{\text{bSD}}^{\text{f}} = (E, \prec_{\text{f}}, \text{lbl})$ , in the presence of in-order delivery channels.

In general, when the specific type of channel is not important for the discussion, we will use  $\llbracket - \rrbracket_{\text{bSD}}$  as a generic semantic function for basic sequence diagrams.

Basic sequence diagrams can be composed to obtain more complex behaviors. In UML 2 this is possible by means of interaction operators. We consider four operators: **seq** (for weak sequential composition), **alt** (for alternative composition), **par** (for parallel composition) and **loop** (for iterative composition). The weak sequential composition of two sequence diagrams consists in their lifeline-by-lifeline concatenation, such that for each instance, the events of the first diagram precede the events of the second diagram. Events on different lifelines are interleaved. In the parallel composition of two sequence diagrams their events are interleaved. The alternative composition of two sequence diagrams describes a choice between them, such that in any run of the system events will be ordered according to only one of the diagrams. That is, alternative composition introduces alternative orderings of events. The semantics of an alternative composition of basic sequence diagrams is therefore defined by a set of posets. The iterative composition of a sequence diagram can be seen as the weak sequential composition of a number of instances of that sequence diagram.

The syntax of a composite sequence diagram (SD) is defined by the following BNF-grammar:

$$SD \stackrel{\text{def}}{=} \text{bSD} \mid (SD_1 \text{ seq } SD_2) \mid (SD_1 \text{ alt } SD_2) \mid (SD_1 \text{ par } SD_2) \mid \text{loop}(\text{min}, \text{max}) SD_1$$

We describe the semantics of a composite sequence diagram by means of a *set* of causal posets (one for each possible alternative behavior described by the sequence diagram). As we did for basic sequence diagrams, we consider two semantic functions  $\llbracket - \rrbracket_{\text{SD}}^{\text{nf}}$  and  $\llbracket - \rrbracket_{\text{SD}}^{\text{f}}$  that assign a *set* of  $\Sigma$ -labeled posets to a composite sequence diagram.

We introduce now two operations on posets that we need for the definition of  $\llbracket - \rrbracket_{\text{SD}}^x$ , with  $x \in \{\text{nf}, \text{f}\}$ . The first one is *weak sequencing* of two  $\Sigma$ -labeled posets (over  $\mathcal{P}, \mathcal{M}$ ), which results in a new  $\Sigma$ -labeled poset where, for each lifeline, the events of the first poset precede the events of the second poset. The second operation is *concurrency* of two  $\Sigma$ -labeled posets, which results in a new poset with interleaved events.

**Definition 11.5 (Weak sequencing).** Let  $p_1 = (E_1, \prec_1, \text{lbl}_1)$  and  $p_2 = (E_2, \prec_2, \text{lbl}_2)$  be two  $\Sigma$ -labeled posets (over  $\mathcal{P}, \mathcal{M}$ ) with disjoint sets of events<sup>2</sup>. Their weak sequencing,  $p_1 \circ_w p_2$ , is a new  $\Sigma$ -labeled poset defined as  $p_1 \circ_w p_2 = (E_1 \cup E_2, \prec_{1 \circ_w 2}, \text{lbl}_1 \cup \text{lbl}_2)$ , where  $\prec_{1 \circ_w 2} = (\prec_1 \cup \prec_2 \cup \{(e_1, e_2) \in E_1 \times E_2 : \text{loc}(e_1) = \text{loc}(e_2)\})^*$ .

**Definition 11.6 (Concurrency).** Let  $p_1 = (E_1, \prec_1, \text{lbl}_1)$  and  $p_2 = (E_2, \prec_2, \text{lbl}_2)$  be two  $\Sigma$ -labeled posets (over  $\mathcal{P}, \mathcal{M}$ ) with disjoint sets of events<sup>2</sup>. Their parallel composition,  $p_1 \parallel p_2$ , is a new  $\Sigma$ -labeled poset defined as  $p_1 \parallel p_2 = (E_1 \cup E_2, \prec_1 \cup \prec_2, \text{lbl}_1 \cup \text{lbl}_2)$ .

Now we can define the semantics of a composite sequence diagram, along the lines in [KL98], as follows:

<sup>2</sup>If they are not disjoint, they are renamed

**Definition 11.7.** The semantics of a composite sequence diagram  $SD$  can be described with a semantic function  $\llbracket - \rrbracket_{SD}^x$ , with  $x \in \{\text{nf}, \text{f}\}$ , such that

$$\begin{aligned} \llbracket bSD \rrbracket_{SD}^x &\stackrel{def}{=} \{ \llbracket bSD \rrbracket_{bSD}^x \} \\ \llbracket SD_1 \text{ seq } SD_2 \rrbracket_{SD}^x &\stackrel{def}{=} \{ p_1 \circ_w p_2 : p_1 \in \llbracket SD_1 \rrbracket_{SD}^x, p_2 \in \llbracket SD_2 \rrbracket_{SD}^x \} \\ \llbracket SD_1 \text{ par } SD_2 \rrbracket_{SD}^x &\stackrel{def}{=} \{ p_1 \parallel p_2 : p_1 \in \llbracket SD_1 \rrbracket_{SD}^x, p_2 \in \llbracket SD_2 \rrbracket_{SD}^x \} \\ \llbracket SD_1 \text{ alt } SD_2 \rrbracket_{SD}^x &\stackrel{def}{=} \llbracket SD_1 \rrbracket_{SD}^x \cup \llbracket SD_2 \rrbracket_{SD}^x \\ \llbracket \text{loop}(min, max) SD \rrbracket_{SD}^x &\stackrel{def}{=} \bigcup_{min \leq i \leq max} \llbracket \Delta_i . SD \rrbracket_{SD}^x \end{aligned}$$

where

$$\begin{aligned} \Delta_0 . SD &\stackrel{def}{=} \{ (\emptyset, \emptyset, \emptyset) \} \\ \Delta_n . SD &\stackrel{def}{=} SD \text{ seq } \Delta_{n-1} . SD, n > 0 \end{aligned}$$

In general, when the specific type of channel is not important for the discussion, we will use  $\llbracket - \rrbracket_{SD}$  as a generic semantic function for sequence diagrams.

For the analysis of sequence diagrams it is useful to distinguish their initiating and terminating events. For a sequence diagram  $SD$ , we denote its multi-set of *initiating events* as  $init(SD) = \{min(ps) : ps \in \llbracket SD \rrbracket_{SD}\}$ , where  $min(ps) = \{e \in E : \nexists e' \in E, e' \prec e\}$  is the set of minimum events (i.e. events non-causally dependent on other events) of the  $\Sigma$ -labeled poset  $ps$ . The initiating events of  $SD$  will be the minimum sending events for each possible alternative described by the sequence diagram. Similarly, we denote multi-set of *terminating events* for a sequence diagram  $SD$  as  $term(SD) = \{max(ps) : ps \in \llbracket SD \rrbracket_{SD}\}$ , where  $max(ps) = \{e \in E : \nexists e' \in E, e \prec e'\}$  is the set of maximum events (i.e. events that do not precede any other events) of the  $\Sigma$ -labeled poset  $ps$ . The terminating events of  $SD$  will be the maximum receiving events for each possible alternative described by the sequence diagram. Finally, we denote the multi-set of *terminating sending events* of  $SD$  as  $term_{snd}(SD) = \{max_{snd}(ps) : ps \in \llbracket SD \rrbracket_{SD}\}$ , where  $max_{snd}(ps) = \{s \in S : \nexists s' \in S, s \prec s'\}$  is the set of maximum sending events (i.e. sending events that do not precede any other sending events, just receiving events) of the  $\Sigma$ -labeled poset  $ps$ . The terminating sending events of  $SD$  will be the maximum sending events for each possible alternative described by the sequence diagram.

### 11.3.2 Syntax and Semantics for Choreography Graphs

We present now the syntax and semantics for choreography graphs.

#### Choreography Syntax

Each of the activities in the activity diagram of a choreography can be seen as a phase in the execution of a service collaboration  $C$ . In each phase or activity, a spe-

cific sub-collaboration of  $C$  is active (so-called activity's *active collaboration*). This is represented by adorning the activity with a collaboration-use, whose roles are bound to instances of  $C$ 's roles. For example, in Fig. 11.4, the *BuyTicket* collaboration is active in the first activity. This is expressed by adorning that activity with a  $b:BuyTicket$  collaboration-use, whose roles (i.e.  $P_{bt}$  and  $T_{bt}$ ) are bound to instances of *ShuttleService*'s roles (i.e.  $P:P$  and  $T1:T$ ). The solid circles and bars beside the roles are respectively used to identify the role that initiate and terminate each collaboration. Each activity has one input pin representing the starting execution point of the activity's active collaboration, and one or more output pins representing alternative end-of-execution points of the active collaboration. These pins are represented as small empty rectangles attached to the boundary of the activity node. If several alternative end-of-execution pins exists, each of them is surrounded by an additional rectangle (see Fig. 11.12 on page 218 for an example). Activities may have additional output pins, describing execution points where the active collaboration is suspended to invoke another collaboration, as well as additional input pins, describing execution points at which a previously suspended active collaboration is to be resumed. Pins used for invoking and resuming an activity's active collaboration are represented as small rectangles with an arrow inside. Both input and output pins represent execution points at which an activity's active collaboration interact with other collaborations. They are labeled with predicates describing goals of the active collaboration.

Edges (i.e. directed connections between activities) and control-flow nodes (i.e. decision, merge, fork, join, initial and final nodes) are respectively used to allow and coordinate the flow of control among activities. An activity can only have one incoming edge, so multiple incoming edges must be AND- or OR-joined.

According to the concrete syntax just described, the formal syntax of goal sequences can be defined as follows:

**Definition 11.8 (Choreography).** A choreography of the sub-collaborations of a collaboration  $C$  is a directed graph defined by the tuple  $CH = (V, \mathcal{E}, \mathcal{R}_{int}, \searrow_{ch}, g_e, m_{exp-a}, R_{CH}, AC, m_{a-ac}, m_{p-a}, pin, l_{pred}, p_{type})$  where

- (i)  $V$  is a set of nodes. It is partitioned into an initial node ( $v_0$ ) and sub-sets of activities ( $V_A$ ), input pins ( $V_{InP}$ ), output pins ( $V_{OutP}$ ), control flow nodes ( $V_{FLOW}$ ), accept event actions ( $V_{EA}$ ) and final nodes ( $V_{FI}$ ). In turn,  $V_{FLOW}$  is partitioned into decision ( $V_D$ ), merge ( $V_M$ ), fork ( $V_F$ ) and join ( $V_J$ ) nodes.
- (ii)  $\mathcal{E} \subseteq (V_{OutP} \cup V_{FLOW} \cup V_{EA} \cup \{v_0\}) \times (V_A \cup V_{InP} \cup V_{FI} \cup V_{EA} \cup V_{FLOW})$  is a set of directed edges between nodes, which is partitioned into normal edges ( $\mathcal{E}_n$ ) and interrupting edges ( $\mathcal{E}_{int}$ ).
- (iii)  $\mathcal{R}_{int}$  is a set of interruptible regions (i.e. regions containing nodes that can be interrupted).
- (iv)  $\searrow_{ch} \subseteq \mathcal{R}_{int} \times (\mathcal{R}_{int} \cup V)$  is a hierarchy relation among interruptible regions and nodes. We write  $reg \searrow_x$  if  $x$  is a node or an interruptible region that is directly contained by the interruptible region  $reg$ .

- (v)  $g_e$  is a guard function for edges. It is defined from  $\mathcal{E}_n$  into boolean expressions.
- (vi)  $R_{CH} = \{(id, type) : type \in R_C\}$  is a set of role instances, with  $R_C$  being the set of roles of collaboration  $C$ .
- (vii)  $AC$  is a set of *active collaborations*, that is, a collaboration-use representing a specific occurrence of one of  $C$ 's sub-collaborations. For each  $(id, type, B) \in AC$ ,  $id$  is the name of the collaboration-use;  $type$  is the name of the collaboration that actually describes the collaboration-use (i.e. one of  $C$ 's sub-collaborations); and  $B \subseteq R_{type} \times R_{CH}$  is a set of role bindings, where  $R_{type}$  is the set of roles of the sub-collaboration named  $type$ .
- (viii)  $m_{a-ac} : V_A \rightarrow AC$  is a non-injective function that maps active collaborations to activities.
- (ix)  $m_{p-a} : V_{InP} \cup V_{OutP} \rightarrow V_A$  is a function mapping input and output pins to activities, and  $pin : V_A \rightarrow \mathcal{P}(V_{InP} \cup V_{OutP})$  is a function that returns the set of pins attached to a given activity.
- (x)  $l_{pred} : V_{InP} \cup V_{OutP} \rightarrow Pre$  is an injective function labeling each input and output pin of an activity with a state predicate of the activity's active collaboration.
- (xi)  $p_{type} : V_{InP} \cup V_{OutP} \rightarrow \{START, END, INVOCATION, RESUMPTION\}$  is a function that classifies pins as either *starting*, *end-of-execution*, *invocation* or *resumption* ones.

### Choreography Semantics

The semantics of a choreography can be intuitively understood in terms of a token-game. At a high level of abstraction, when an activity receives an input token, its active collaboration is enabled. If the token is received on the activity's starting input pin, the active collaboration can begin execution from its initial state. Otherwise, if the token is received through a resuming input pin, the active collaboration can resume execution from the state represented by the event-goal labeling the pin. The active collaboration in reality begins or resumes its execution when one of its roles takes the appropriate initiative. Thereafter, it evolves until an interaction point with other collaborations is eventually reached. That is, the active collaboration runs until the predicate of one of its activity's output pins holds. When this happens, the control token is passed on to the next activity or control node. According to this semantics, the intended behavior of the *ShuttleService* collaboration, as specified by its choreography (see Fig. 11.4), closely reflects the requirements. Initially the *BuyTicket* collaboration is started and thereafter suspended after the ticket is requested. At that point, a check is performed to determine if the vehicle is at the terminal (i.e. at  $T1$ ). If the result is positive, *BuyTicket* is finished and *EnterVehicle* is enabled, followed by *VehDeparture*, *VehArrival* and *ExitVehicle*. If the vehicle was not at  $T1$ , this role initiates *ReqVehicle* to request the vehicle from  $T2$ . *VehDeparture* is then executed, followed by *VehArrival*, which allows *BuyTicket* to be resumed. Thereafter the service progresses as explained before.

The above high-level semantics, which describes the intended behavior of a service collaboration from the point of view of the service designer, was formalized in [CB06b]. This semantics abstract away from individual roles, and implicitly considers that the sequencing between collaborations is strong. That is, when a collaboration passes the control token to the next collaboration through an end-of-execution pin, the behavior of the former collaborations is assumed to be completely finished, for all its participating roles. In the present work we consider a weak sequencing semantics, since it better reflects the actual behavior. Instead of assuming only one control token, we may think that there is one specific token for each role instance appearing in the choreography graph. In order to perform a sending or receiving event, a role needs its token. As soon as a role is finished with its participation on an activity's active collaboration, its token can be sent out to the next activity and the role can start participating in the active collaboration of the new activity. This means that, at any point in time, the execution of two active collaborations may partially overlap. This behavior can be described with a Petri net. However, we will not use Petri nets to formalize the semantics of choreography graphs. Instead we will use partial orders, as we did for sequence diagrams. We note that runs of a Petri net can be described as partial orders over events, where the events correspond to the firing of the net transitions [Kie97].

Paths that start at the initial node of a choreography graph and end at any of the final nodes correspond to finite executions of the service collaboration modeled by the choreography. Infinite paths (due to loops) starting at the choreography's initial node correspond to infinite executions of the service collaboration. A labeled poset can be obtained for each of the finite and infinite execution paths of the choreography. The semantics of a choreography is then defined as the (possibly infinite) set of labeled posets obtained from all the choreography's finite and infinite executions paths.

The execution paths of a choreography can be obtained using a depth first search technique. The labeled poset corresponding to a given execution path can be obtained by applying some simple guidelines. We detail those guidelines for each of the possible ways in which activities (i.e. collaborations) can be composed or ordered in the choreography graph<sup>3</sup>. In the explanation we consider that the behavior of an activity  $C_i$  is described by a collaboration, whose behavior is in turn described by a sequence diagram  $SD_i$ .

**Sequential composition.** When the end-of-execution pin of an activity  $C_1$  is connected (directly, or via one or more control nodes) to the starting pin of an activity  $C_2$ ,  $C_1$  and  $C_2$  are composed in weak sequence. This is the case for activities  $C_1$  and  $C_2$  in Fig. 11.6(a), and  $C_3$  and  $C_1$  in Fig. 11.6(b). When the activities are directly connected (e.g. Fig. 11.6(a)), or connected through decision and merge nodes, the semantics of the composition corresponds to the `seq` operator defined for sequence diagrams. The semantics for the composition in Fig. 11.6(a) is thus  $\llbracket SD_1 \text{ seq } SD_2 \rrbracket_{SD}$ .

---

<sup>3</sup>We do not consider here *alternative* composition, which just defines several execution paths. Decision and merge nodes in the choreography graph are used to select one or another path, but they are otherwise ignored in order to build the labeled poset for the selected execution path.

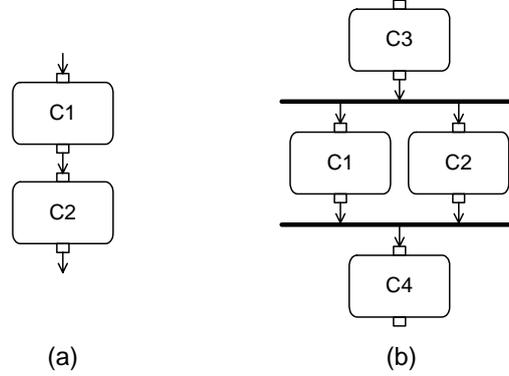


Figure 11.6: Example of sequential and parallel composition of activities in a choreography graph.

**Parallel composition.** Activities inside a fork-join pair are composed in parallel (i.e. they are executed concurrently). Since we require proper nesting of fork and join nodes<sup>4</sup>, the semantics of such composition corresponds to the **par** operator defined for sequence diagrams. The semantics for the composition of  $C_1$  and  $C_2$  in Fig. 11.6(b) is thus  $\llbracket SD_1 \text{ par } SD_2 \rrbracket_{SD}$ . The semantics for the whole composition presented in Fig. 11.6(b) is  $\llbracket (SD_3 \text{ seq } (SD_1 \text{ par } SD_2)) \text{ seq } SD_4 \rrbracket_{SD}$ . We note that fork-join pairs are first processed, and then composed with any preceding and/or succeeding activities.

**Interruption composition.** In this kind of composition an activity  $C_2$  interrupts another activity  $C_1$ . It is represented as in Figures 11.7(a) and 11.7(b). In Fig. 11.7(a) an accept event action is enabled whenever  $C_1$  reaches a point in its execution where predicate  $pred$  holds. From that point of time on, if the event  $e$  associated to the accept event action is observed, activity  $C_1$  is interrupted and activity  $C_2$  starts execution. We assume that event  $e$  is an external stimulus from the environment observed by the role(s) initiating  $C_2$ . If  $C_1$  finishes execution before event  $e$  is observed, the interruptible region (i.e. the dashed rectangle) is abandoned via a normal edge. The accept event action is then disabled (i.e. interruption is no longer possible) and activity  $C_4$  is started. In the case of Fig. 11.7(b), activity  $C_1$  may be interrupted as soon as it starts execution, since the accept event action becomes enabled as soon as the interruptible region is entered.

To define the semantics of interruption we need to introduce the notion of prefix, and prefix with a fixed part, of a labeled poset.

<sup>4</sup>All outgoing edges of a fork node should lead to the same join node, and all incoming edges of a join node should come from the same fork node. An exception is the following. If the join node is connected to a final node, the former could be removed, and let all outgoing edges of the fork lead to final nodes.

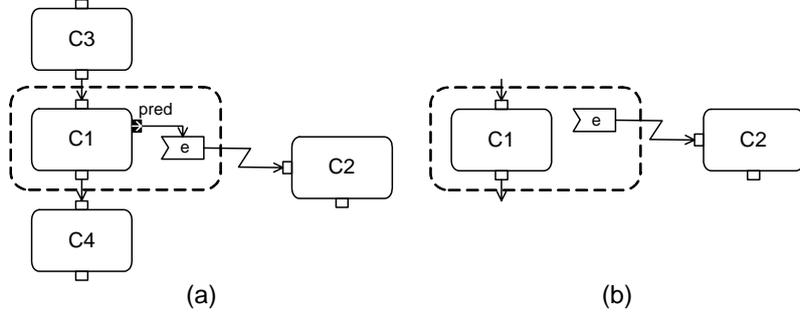


Figure 11.7: Examples of interrupting composition of activities in a choreography graph.

**Definition 11.9 (Prefix).** A labeled poset  $(E', \prec', lbl')$  is a *prefix* of a labeled poset  $(E, \prec, lbl)$  describing the semantics of a sequence diagram  $SD$  iff the following conditions hold:

- $E' \subset E$
- if  $e \in S'$ , then  $rcv(e) \in E'$  (i.e. if the prefix contains a sending event, it also contains its corresponding receiving event)
- $\forall e, f$ , if  $e \prec f \wedge f \in E'$ , then  $e \in E'$  (i.e. if an event  $f$  is part of the prefix, all the events that precede  $f$  in the original poset are also in the prefix)
- $\prec' = \prec \cap (E' \times E')$
- $lbl' = lbl \upharpoonright E'$ , where  $\upharpoonright$  denotes restriction

Note that the empty poset  $[\epsilon]$  is a prefix of each poset. Note also that this definition differs from the one in [KL98] in two respects. First, we require the set of events of the prefix to be strictly included in the set of events of the original poset. This means that a poset cannot be a prefix of itself. Second, for each sending event contained in the prefix we require its matching receiving event to be also contained in the prefix. For a poset  $ps$  we denote the set of all its prefixes as  $prefix(ps)$ .

**Definition 11.10 (Prefix with fixed part).** A labeled poset  $(E', \prec', lbl')$  is a *prefix with a fixed part* of a labeled poset  $(E, \prec, lbl)$ , with upper limit for the fixed part a predicate event  $e_{pred}$  with label  $pred$  (i.e.  $lbl(e_{pred}) = pred$ ), iff the following conditions hold:

- $(E', \prec', lbl')$  is a prefix of  $(E, \prec, lbl)$
- $e_{pred} \in E'$  and  $\forall f \in E, f \prec e_{pred}$ , we have  $f \in E'$

A prefix with a fixed part cannot be an empty poset. It will always contain a fixed part consisting of the predicate event  $e_{pred}$  and all its predecessor events in the

original poset. For a poset  $ps$  and a predicate  $pred$  labeling one of  $ps$ 's events, we denote the set of all its prefixes with a fixed part as  $fprefix(ps, pred)$ .

We let  $C_2 \text{ int } C_1$  informally mean that activity  $C_2$  interrupts activity  $C_1$  from the beginning of  $C_1$  (as in Fig. 11.7(b)). The semantics for this type of interrupting composition is defined as follows:

$$\llbracket C_2 \text{ int } C_1 \rrbracket = \{ ps_1 \circ_w ps_2 : ps' \in \llbracket SD_1 \rrbracket_{SD}, ps_1 \in fprefix(ps', pred), ps_2 \in \llbracket SD_2 \rrbracket_{SD} \}$$

Valid posets for the interruption are those formed by the weak sequencing of a prefix of one of  $C_1$ 's posets and one of  $C_2$ 's posets.

We now let  $C_2 \text{ int}(pred) C_1$  informally mean that activity  $C_2$  interrupts activity  $C_1$  from the point in the execution of  $C_1$  where predicate  $pred$  holds (as in Fig. 11.7(a)). We also let  $e_{pred}$  be the predicate event with label  $pred$  (i.e.  $lbl(e_{pred}) = pred$ ). The semantics for this type of interrupting composition can then be defined as follows:

$$\llbracket C_2 \text{ int}(pred) C_1 \rrbracket = \{ ps_1 \circ_w ps_2 : ps' = (E, \prec, lbl) \in \llbracket SD_1 \rrbracket_{SD} \text{ such that } e_{pred} \in E, ps_1 \in fprefix(ps', pred), ps_2 \in \llbracket SD_2 \rrbracket_{SD} \}$$

Since  $SD_1$  may have several associated posets (in case it contains alternatives or loops), we select those  $ps'$  posets that contain  $e_{pred}$ . Valid posets for the interruption are those formed by the weak sequencing of a prefix with a fixed part of  $ps'$  and one of  $C_2$ 's posets. The fixed part is the behavior the needs to be executed in order for predicate  $pred$  to hold.

We note that the above semantics describe cases where the interruption actually happens, that is, where an interrupting edge is traversed. The semantics for the choreography in Fig. 11.7(a) would be

$$\llbracket (SD_3 \text{ seq } SD_1) \text{ seq } SD_4 \rrbracket_{SD} \cup \llbracket SD_3 \text{ seq } (C_2 \text{ int}(pred) C_1) \rrbracket$$

**Invocation composition.** In the most general case, this type of composition implies that while in the middle of its execution,  $C_1$  invokes  $C_2$ . Thereafter, the behaviors of  $C_1$  and  $C_2$  may proceed independently. Here we consider a more specific case of invocation (so-called *invocation with feedback*), where  $C_1$ , after reaching a point in its execution where a predicate  $pred$  holds, invokes  $C_2$ .  $C_1$  then is suspended and waits for  $C_2$  to execute all or part of its behavior before resuming. This normally represent a goal dependency between  $C_1$  and  $C_2$ , such that  $C_1$  can only achieve its own goal if  $C_2$  achieves its corresponding one. An example is shown in Fig. 11.8(a).

Invocation composition is achieved with help of invocation and resumption pins. For each invocation output pin labeled with a predicate  $pred$ , there should be a corresponding resumption input pin labeled with the same predicate (see Fig. 11.8). These pins correspond to execution points that are represented by means of predicate events in the sequence diagram of the invoking activity. For each pair of invocation/resumption pins of an invoking activity that are labeled with a predicate  $pred$ , the sequence diagram describing the behavior of that activity should contain a predicate event labeled with the same predicate  $pred$ . Since the invoking activity is

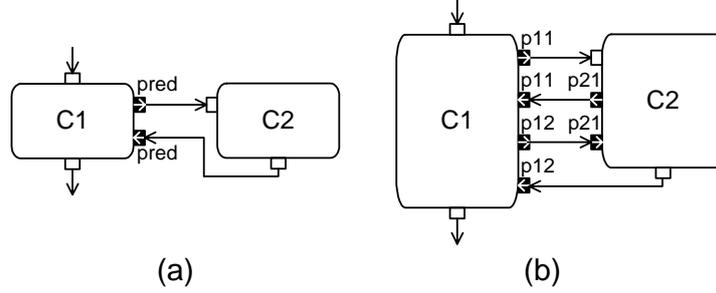


Figure 11.8: Examples of invocation composition of activities in a choreography graph.

required to get suspended after the invocation, such predicate event may not appear inside an operand of a **par** combined fragment.

To define the semantics of invocation we need to introduce the notion of segment of a labeled poset.

**Definition 11.11 (Segment).** A labeled poset  $(E', \prec', lbl')$  is a *segment* of a labeled poset  $ps = (E, \prec, lbl)$  with predicates  $pred_1$  and  $pred_2$  as lower and upper limits (written  $seg(ps, pred_1, pred_2)$ ), iff the following conditions hold:

- Let  $e_1, e_2 \in E$  such that  $lbl(e_1) = pred_1$  and  $lbl(e_2) = pred_2$ , and let  $E_{pr} = \{f \in E : f \prec e_2\}$  and  $E_{sc} = \{f \in E : e_1 \prec f\}$ ,
  - if  $pred_1 = start$ , then  $E' = \{e_2\} \cup E_{pr}$  (i.e. the segment contains  $e_2$  and all events of the original poset that are predecessors of  $e_2$ )
  - if  $pred_2 = end$ , then  $E' = E_{sc}$  (i.e. the segment contains all events of the original poset that are successors of  $e_1$ )
  - if  $pred_1 \neq start \wedge pred_2 \neq end$ , then  $E' = \{e_2\} \cup (E_{sc} \cap E_{pr})$  (i.e. the segment contains  $e_2$  and all events of the original poset that are successors of  $e_1$  and predecessors of  $e_2$ )
  - if  $pred_1 = start \wedge pred_2 = end$ , then  $E' = E$  (i.e. the segment is the original poset)
- if  $e \in S'$ , then  $rcv(e) \in E'$  (i.e. if the segment contains a sending event, it also contains its corresponding receiving event)
- $\prec' = \prec \cap (E' \times E')$
- $lbl' = lbl \upharpoonright E'$ , where  $\upharpoonright$  denotes restriction

In the above definition, *start* and *end* are special purpose predicates that should not label any of the predicate events of the original poset.

We let  $C_1 \text{ inv}(\Psi_1, \Psi_2) C_2$  informally mean that activities  $C_1$  and  $C_2$  invoke each other (with  $C_1$  performing the first invocation) at the execution points indicated

by the predicates in  $\Psi_1$  and  $\Psi_2$ .  $\Psi_1$  (resp.  $\Psi_2$ ) is an ordered set containing the predicates that hold at execution points where  $C_1$  (resp.  $C_2$ ) invokes  $C_2$  (resp.  $C_1$ ). We now explain how to obtain  $\Psi_1$  for each  $ps'_1 \in \llbracket SD_1 \rrbracket_{SD}$  (the same procedure can be used to obtain  $\Psi_2$ ):

- (i) Get the predicates labeling the invocation pins of  $C_1$  that are connected to pins of  $C_2$ , that is,  $Pr = \{l_{\text{pred}}(ip) : ip \in \text{pin}(C_1), p_{\text{type}}(ip) = \text{INVOCATION}, (ip, x) \in \mathcal{E}, m_{p-a}(x) = C_2\}$
- (ii) Get the total ordered set  $t_{\text{pred}}$  of predicate events in  $ps'_1 = (E'_1, \prec'_1, lbl'_1)$  that are labeled with predicates from  $Pr$  (i.e. events  $e \in E'_1$  such that  $lbl'_1(e) \in Pr$ , and their order relations). Note that it is a total order since the predicate events denoting invocation points cannot appear inside **par** combined fragments of  $C_1$ 's sequence diagram to enforce suspension.
- (iii) Replacing each event in  $t_{\text{pred}}$  with its label, and adding the special purpose predicates *start* and *end* as first and last elements, respectively, we obtain  $\Psi_1$

For the invocation composition in Fig. 11.8(a) we would have  $\Psi_1 = \{start, pred, end\}$  and  $\Psi_2 = \{start, end\}$ , while for the invocation composition in Fig. 11.8(b) we would have  $\Psi_1 = \{start, p_{11}, p_{12}, end\}$  and  $\Psi_2 = \{start, p_{21}, end\}$ .

In general, we assume that  $\Psi_1 = \{pred_1^1, \dots, pred_1^n\}$  (with  $pred_1^1 = start$  and  $pred_1^n = end$ ) and  $\Psi_2 = \{pred_2^1, \dots, pred_2^{n-1}\}$  (with  $pred_2^1 = start$  and  $pred_2^{n-1} = end$ ), where  $n > 1$ . The semantics for  $C_1 \text{ inv}(\Psi_1, \Psi_2) C_2$  can then be defined as follows:

$$\begin{aligned} \llbracket C_1 \text{ inv}(\Psi_1, \Psi_2) C_2 \rrbracket &= \{seg(ps'_1, pred_1^1, pred_1^2) \circ_w seg(ps'_2, pred_2^1, pred_2^2) \circ_w \dots \\ &\quad \dots seg(ps'_2, pred_2^{n-2}, pred_2^{n-1}) \circ_w seg(ps'_1, pred_1^{n-1}, pred_1^n) : \\ &\quad ps'_1 \in \llbracket SD_1 \rrbracket_{SD}, ps'_2 \in \llbracket SD_2 \rrbracket_{SD}\} \end{aligned}$$

## 11.4 Realizability of Choreographies

In the following sections we study the direct realizability of a choreography from the point of view of the operators used to compose the sub-collaborations. In our discussion we assume that each sub-collaboration referred to in a choreography is directly realizable. Starting from single messages and applying the operators and rules described in the following will ensure this. For the sequential, alternative, parallel and interruption composition operators we study the problems that can lead to difficulties of realization. We investigate the actual nature of these problems and discuss possible solutions to prevent or remedy them.

### 11.4.1 Sequential Composition

Sequential composition imposes a causal dependency or partial order between the events of the composed sub-collaborations. In the following the notions of strong and weak sequential composition are discussed.

### Strong Sequencing

Strong sequencing between two collaborations  $C_1$  and  $C_2$ , written  $C_1 \circ_s C_2$ , requires  $C_1$  to be completely finished, for all its components, before  $C_2$  can be initiated. It requires a direct precedence relation between the terminating action(s) of  $C_1$  and the initiating action(s) of  $C_2$ , so that the latter can only happen after the former are finished. This leads to the following:

**Proposition 11.12.** *The strong sequential composition of two directly realizable collaborations  $C_1$  and  $C_2$ ,  $C_1 \circ_s C_2$ , is directly realizable if all terminating actions of  $C_1$  and all initiating actions of  $C_2$  are located at the same component.*

The above proposition requires  $C_1$  to terminate at the component initiating  $C_2$ . This is the only way the initiator of  $C_2$  can know when  $C_1$  is completely finished. If this condition is not satisfied, coordination messages must be added from  $C_1$ 's terminating components to  $C_2$ 's initiating components, in order to guarantee the strong sequencing. This could be done automatically by a synthesis algorithm [BG86].

### Weak Sequencing

Weak sequencing of two sub-collaborations  $C_1$  and  $C_2$ , written  $C_1 \circ_w C_2$ , does not require  $C_1$  to be completely finished before  $C_2$  can be initiated. Any component can start participating in  $C_2$  as soon as it has finished with  $C_1$  (without waiting for the other components to finish as well), which means that the actions from both collaborations are sequenced on a per-component basis. This is the sequential composition semantics used in HMSCs and UML Interaction Overview Diagrams, but not in standard UML activity diagrams. The semantics of choreography graphs presented in the previous section assumes weak sequencing.

Weak sequencing introduces a certain degree of concurrency, since the executions of the composed collaborations may partially overlap. Although such concurrency may be desirable for performance or timing reasons (i.e. a component may initiate a new collaboration if the actions in that collaboration were independent of the actions that have yet to be executed in the first collaboration), it comes at a price, since it may lead to specifications that are not directly realizable and even counter-intuitive. The specification in Fig. 11.9(a) is an example of a counter-intuitive composition. According to the weak sequence semantics, component  $B$  may initiate collaboration  $C_3$  as soon as it has finished with  $C_1$ . As a result, collaborations  $C_2$  and  $C_3$  may be executed in any order in the realized system. This is counter-intuitive to the specification, which we assume reflects the designer's intention (i.e. that  $C_3$  should be executed after  $C_2$ , with some allowed overlapping). If the designer's intention was that the collaborations should be concurrently executed, this should rather be explicitly specified by means of parallel composition.

To avoid the aforementioned problem, when two collaborations are composed in weak sequence the component initiating the second collaboration should participate in the first collaboration (e.g. as in the composition of  $C_1$  and  $C_2$  in Fig. 11.9(a)). We say a sequential composition with this property is weakly-causal:

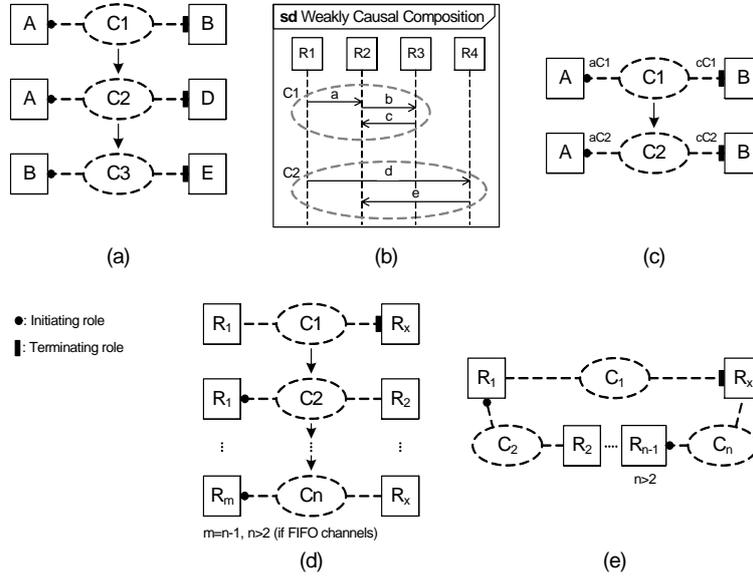


Figure 11.9: Weak Sequence Problems

**Definition 11.13 (weak-causality).** The weak sequential composition of two collaborations,  $C_1 \circ_w C_2$ , is *weakly-causal* if the initiator of  $C_2$  participates in  $C_1$ .

A weak sequential composition without the weak-causality property (e.g. the composition of  $C_2$  and  $C_3$  in Fig. 11.9(a)) can be made weakly-causal by means of a synchronization message sent from one of the participants of the first collaboration to the initiator of the second collaboration<sup>5</sup>.

Weak-causality is a necessary condition for direct realizability of weak sequential composition. However, it is not strong enough to be a sufficient condition. For example, consider the weak sequential composition of  $C_1$  and  $C_2$  in Fig. 11.9(b). This composition is weakly-causal, but it is not directly realizable. Component  $R1$  may initiate collaboration  $C_2$  just after sending message  $a$  in  $C_1$ . Therefore, the actions in  $C_1$  that follow the sending of message  $a$  may overlap with those performed in  $C_2$  by the same components. For example, message  $e$  may be received at  $R2$  before message  $c$ , or even before message  $a$ . Obviously, this message reception order has not been explicitly specified. We note that weak-causality is enforced in the so-called local-HMSCs of [GMSZ06].

In the literature about MSCs, the possibility that messages may be received in a different order from the one specified is usually called a **race condition** [AHP96]. In general, race conditions can occur when a receiving event is specified to happen before another event (i.e. either a receiving or a sending one), and both events are located on the same component. The reason lies in the controllability of events.

<sup>5</sup>The component sending this message should be chosen among the components that participate in both collaborations (if any), in order to minimize the risk of introducing race conditions.

While a component can always control when its sending events should happen (e.g. it can wait for one or more messages to be received before sending a message), it cannot control the timing of its receiving events. The occurrence of races highly depends on the underlying communication service that is used. If no assumption is made about the communication service, races can only be prevented if all message transmissions are strongly sequenced. This condition might be quite restrictive. We now present a less restrictive condition that does not prevent all races, but reduces their number and facilitates their detection, compared with weak-causality. This condition, which we call *send-causality*, requires all sending events to be ordered, except those that have been explicitly specified (with parallel composition) to happen concurrently.

**Definition 11.14 (send-causal composition).**  $C_1 \circ_w C_2$  is *send-causal* if 1)  $C_1$  and  $C_2$  are send-causal, and 2) the component initiating  $C_2$  is the one that performs either the last sending event of  $C_1$  or the receiving event corresponding to that sending event.

An elementary collaboration is send-causal if can be decomposed into a choreography of sub-collaborations, each of them consisting of exactly one message, where all sequential compositions in the choreography are send-causal. We can give a more formal definition based on sequence diagrams as follows:

**Definition 11.15 (send-causal elementary collaboration).** An elementary collaboration is *send-causal* if its associated sequence diagram is send-causal.

**Definition 11.16 (send-causal sequence diagram).** A sequence diagram is *send-causal* if any of the following conditions is satisfied:

- (i) If the diagram is a basic sequence diagram, the following holds:  $\forall s, s' \in \mathcal{S}$ , if  $s <_m s' \wedge \exists s'' \in \mathcal{S}, s <_m s'' <_m s'$  then  $loc(s') = loc(s) \vee loc(s') = loc(rcv(s))$ .
- (ii) If the diagram is a composite sequence diagram, the following holds:
  - All its basic sequence sub-diagrams are send-causal
  - Whenever two sub-diagrams  $SD_1$  and  $SD_2$  are composed in weak sequence (i.e.  $SD_1 \text{ seq } SD_2$ ), the following is satisfied:  $\forall \mathcal{T}_S \in \text{term}_{\text{snd}}(SD_1), \forall \mathcal{I} \in \text{init}(SD_2), \forall s_t \in \mathcal{T}_S, \forall s_i \in \mathcal{I}, loc(s_i) = loc(s_t) \vee loc(s_i) = loc(rcv(s_t))$ .

Note that in condition (ii) of Definition 11.16 we have implicitly considered the possibility that a composite sequence diagrams may describe alternative or parallel behaviors. In such situation, for each alternative behavior, or each parallel behavior, we require that the send-causality property holds.

It can be shown that when send-causality is enforced, races may only occur between two or more consecutive receiving events (i.e. not between a sending event and a receiving event).

**Proposition 11.17.** *In a send-causal composition race conditions may only exist between two or more consecutive receiving events.*

*Proof.* See Appendix 11.A. □

**Corollary 11.18.** *A send-causal composition is directly realizable over a communication service with in-order delivery and separate input buffers.*

One of our motivations is to provide guidelines for constructing specifications with as few conflicts as possible and whose intuitive interpretation corresponds to the behavior allowed by the underlying semantics. We therefore propose, as a general specification guideline, that all elementary collaborations be send-causal. Weak sequencing of collaborations should also be send-causal, unless there is a good reason to relax this requirement. In the following we assume that all elementary collaborations are send-causal.

A *potential* race condition exists between two weakly sequenced collaborations,  $C_1 \circ_w C_2$ , if there is a component that participates in both collaborations playing roles that may partially overlap. Due to Proposition 11.17, if the sequencing is send-causal this may only happen when the role that the component plays in  $C_1$  ends with a message reception (i.e. it is a terminating role) and the role it plays in  $C_2$  starts with another message reception (i.e. it is a non-initiating role). Whether a potential race condition is an *actual* race or not depends on the underlying communication service, and on whether messages are received from the same or from different components. For example, in Fig. 11.9(c) a potential race condition exists at component  $B$  between the receptions of the last message in  $C_1$  and the first message in  $C_2$ , but it is only actual in the case of out-of-order delivery.

We note that race conditions may not only appear between *directly* composed collaborations (Fig. 11.9(c)), but also between *indirectly* composed ones, as shown in Fig. 11.9(d). In this specification it is the weak sequencing between  $C_1$  and  $C_2$  that makes the potential race between  $C_1$  and  $C_n$  possible. We therefore say that there is **indirect weak sequencing** between  $C_1$  and  $C_n$ . This “propagation” of weak sequencing makes it more difficult to avoid races.

We have the following result:

**Proposition 11.19.** *The send-causal weak sequential composition of a set of directly-realizable collaborations is directly realizable*

- *over a communication service with in-order delivery if the following condition is satisfied: if a component plays a terminating role in a collaboration  $C_1$  followed by a non-initiating role in another collaboration  $C_n$ , then the last message it receives in  $C_1$  and the first one it receives in  $C_n$  are sent by the same peer-components.*
- *over a communication service with out-of-order delivery only if no component plays a terminating role followed by a non-initiating role.*

Working with binary collaborations we can easily know which component sends the first and last messages of a collaboration, if we know which components play

the initiating and terminating roles. Due to Proposition 11.19, actual races can then be detected at an early specification stage, when the detailed behavior of each collaboration has not yet been specified, but only the selection of their initiating and terminating roles has been done. In the case of n-ary collaborations, we can perform the same early analysis, but only potential races can be discovered.

One interesting thing of the specification with collaborations is that we can get information about potential races from the diagram describing the structural composition of collaborations (see e.g. Fig. 11.9(d)). In such diagram we can see whether a component participates in several collaborations, and whether it plays at least one terminating and one non-initiating role in them. If that is the case, a potential race exists. This information could then be used to direct the analysis of the behavioral specification (i.e. the choreography).

**Resolution of Race Conditions.** Race conditions can be resolved in several ways. Some authors [Mit05, CKS05] have proposed mechanisms to automatically eliminate race conditions by means of synchronization messages. We note that when the send-causality property is satisfied, the synchronization message should be used to transform the weak sequencing leading to the race into strong sequencing. If synchronization messages are added in other places new races may be introduced.

Other authors tackle the resolution of race conditions at the design and implementation levels. They differentiate between the reception and consumption of messages. This distinction allows messages to be consumed in an order determined by the receiving component, independently of their arrival order. We call this *message reordering for consumption*. In general, this reordering may be implemented by first keeping all received messages in a (unordered) pool of messages. When the behavior of the component expects the reception of one or a set of alternative messages, it waits until one of these messages is available in the message pool. Khendek et al. [KZ05] use the SDL Save construct to specify such message reordering. This technique can be used to resolve races between messages received from the same source (i.e. in the case of channels with out-of-order delivery), as well as races between messages received from different sources. In the latter case, a communication service with separate input buffers would also resolve the races. Finally, races may also be eliminated if an explicit consumption of messages in all possible orders is implemented (i.e. similar to co-regions in MSCs).

We believe that the resolution of races heavily depends on the specific application domain and requirements, as well as in the context which they happen in. In some cases the addition of synchronization messages is not an option, and a race has to be resolved by reordering for consumption. In other cases, such as when races lead to race propagation problems (see Section 11.4.2) a strict order between receptions is required, so components should be synchronized by extra messages. At any rate, all race conditions should be brought to the attention of the designer once discovered. She could then decide, first, whether the detected race entails a real problem (e.g. in Fig. 11.9(d) there is no race if all channels have the same latency). Then, she could decide whether reordering for consumption is acceptable or synchronization messages need to be added or the specification has to be revised.

## Loops

Loops can be used to describe the repeated execution of a (composite) collaboration, which we call the body collaboration. A loop can therefore be seen as a shortcut for strong or weak sequential composition of several executions of the same body collaboration. This means that the rules for strong/weak sequencing must be applied. We note that all executions of a loop involve the same set of components (weak-causality property is thus always satisfied). This fact makes the chances for races high when weak sequencing is used. Strong sequencing should therefore be preferred for loop bodies in the general case.

Loops may give rise to so-called *process divergence* [BAL97], characterized by a component sending an unbounded number of messages ahead of the receiving component. This may happen if the communication between any two of the participants in the body collaboration is unidirectional (i.e. only happens in one direction).

As we will see in the next section, loops may also affect the realizability of choices.

### 11.4.2 Alternative Composition

Alternative composition is specified by means of choice operators, and describes alternatives between different execution paths. In a choice one or more *choosing* components decide the alternative of the choice to be executed, based on the (implicit or explicit) conditions associated with the alternatives. The other *non-choosing* components involved in the choice follow the decision made by the choosing components (i.e. execute the alternative chosen by the later ones). It is therefore important that:

- (i) The choosing components, if several, agree on the alternative to be executed. We call this the **decision-making process**.
- (ii) The decision made by the choosing components is correctly propagated to the non-choosing components. We call this the **choice-propagation process**.

In the following we study how each of these aspects affect the realizability of a choice. We assume that the set of choosing components is the union of the initiating components of all the choice alternatives.

#### Decision-making Process

The intuitive interpretation of a choice is that only one of the alternative behaviors is to be eventually executed. Deciding which alternative to be executed becomes simple if there is only one choosing component, and the conditions for the alternatives are local to that component (i.e. they are expressed in terms of observable predicates). Choices with this property are called **local**. It is easy to see that local choices are realizable (up to the decision-making process), since the decision is made by a single component based only on its local knowledge.

The decision-making process gets complicated when there is more than one choosing component. This is the case in the choice of Fig. 11.10(a), where there are two choosing components, namely *A* and *B*. From a global perspective, we may think

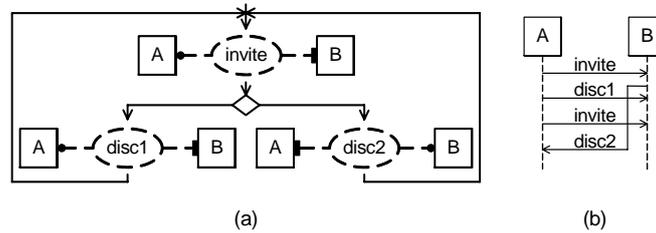


Figure 11.10: Competing-initiatives choice

that once the decision node is reached, either component *A* initiates collaboration *disc1* with *B*, or component *B* initiates collaboration *disc2* with *A*. We are assuming then that there is an implicit synchronization between *A* and *B*, which allows them to agree on the alternative to be executed. However, in a directly realized system, components *A* and *B* will not be able to synchronize and they may decide to initiate both collaborations simultaneously.

Choices involving more than one choosing component are usually called **non-local choices** [BAL97]. They are normally considered as pathologies that can lead to misunderstanding and unspecified behaviors, and algorithms have been proposed to detect them in the context of HMSCs (e.g. [BAL97, H el01]). Despite the extensive attention they have received, there is no consensus on how they should be treated. We believe this is due to a lack of understanding of their nature. Some authors (e.g. [BAL97]) consider them as the result of an underspecification and suggest their elimination. This is done by introducing explicit coordination, as a refinement step towards the design. Other authors look at non-local choices as an obstacle for realizability and propose a restricted version of HMSCs, called *local HMSCs* [HJ00, GMSZ06], that are always realizable. These HMSCs forbid non-local choices. Finally, there are authors [GY84, MGR05] that consider non-local choices to be almost inevitable in the specification of distributed systems with autonomous processes. They propose to address them at the implementation level, and propose a generic implementation approach of non-local choices.

The problem with non-local choices is the existence of several *uncoordinated* components that have the possibility to make an independent decision in the directly realized system. As a solution, we may think of making the choice local by coordinating these components (i.e. either with additional messages or with additional message contents), so that they make a common decision. Such coordination may however not be feasible in all contexts and application domains. Consider, for example, the specification of a personal communication service where both end-users can take the initiative to disconnect. This could be specified as a non-local choice between two disconnection collaborations, each of them initiated by a different component (see Fig. 11.10(a)). The decision made by any of the components to initiate one of the disconnection collaborations is not totally controlled by that component, but it is triggered by the respective end-user. It makes therefore little sense to coordinate the components in order to obtain a local choice, since this would imply the coordination of the end-users' initiatives. Such non-local choice is simply unavoidable.

We refer to non-local choices where the coordination of the choosing components is not feasible as **competing-initiatives choices**. A characteristic of them is that all the alternative collaborations are simultaneously enabled, and will be triggered by events that cannot be controlled by the initiating components, such as an end-user initiative or a time-out. As a result, the alternative collaborations cannot be prevented from being simultaneously triggered. If this happens, it should be detected as soon as possible and resolved by means of a proper conflict resolution. Any component involved in two or more alternatives may be potentially used to detect the initiative conflict and initiate the resolution. For such components, the competing initiatives reveal themselves in the components' role sequences as choices between an initiating and a non-initiating role, or between two non-initiating roles played in collaborations with different peers.

A side effect of competing-initiatives choices is the existence of **orphan** messages. Consider again the specification in Fig. 11.10(a), which describes the repetitive execution of collaboration *invite* followed by either *disc1* or *disc2*. Now imagine that each collaboration consists only of one message. Then the scenario in Fig. 11.10(b) is possible, where message *disc2* is sent as a response to the first *invite* message, but it is received by *A* after having sent the second *invite*. Component *A* may then consume message *disc2* as a response to the second *invite* message, leading to an undesired behavior. In this scenario the collaboration occurrence where *disc2* is sent is considered finished while *disc2* is still in the system (i.e. not consumed). This message becomes thus orphan, with the danger of being consumed in a latter occurrence of the same collaboration. To avoid this messages should be marked (e.g. with a session id), so they are only consumed within the right collaboration instance.

Competing-initiatives choices correspond to the non-local choices discussed by Gouda et al. [GY84] and Mooij et al. [MGR05]. These authors propose some resolution approaches. In the domain of communication protocols, Gouda et al. [GY84] proposes a resolution approach for two competing alternatives (i.e. two choosing components), which gives different priorities to the alternatives. Once a conflict is detected, the alternative with lowest priority is abandoned. With motivation from a different domain, where Gouda's approach is not satisfactory, Mooij et al. [MGR05] propose a resolution technique that executes the alternatives in sequential order (according to their priorities), and is valid for more than two choosing components. We conclude that the resolution approach to be implemented depends on the specific application domain. We therefore envision a catalog of domain specific resolution patterns from which a designer may choose the one that better fits the necessities of her system. We note that any potential resolution should also address the problem of orphan messages, which is not considered in either [GY84] or [MGR05].

### Choice-propagation Process

The fact that a choice is local does not guarantee its realizability. The decision made by the choosing component must be properly propagated to the non-choosing components, in order for them to execute the right alternative. In each alternative, the behavior of a non-choosing component begins with the reception of a sequence of messages, which we call the *triggering trace*. Thereafter, the component may

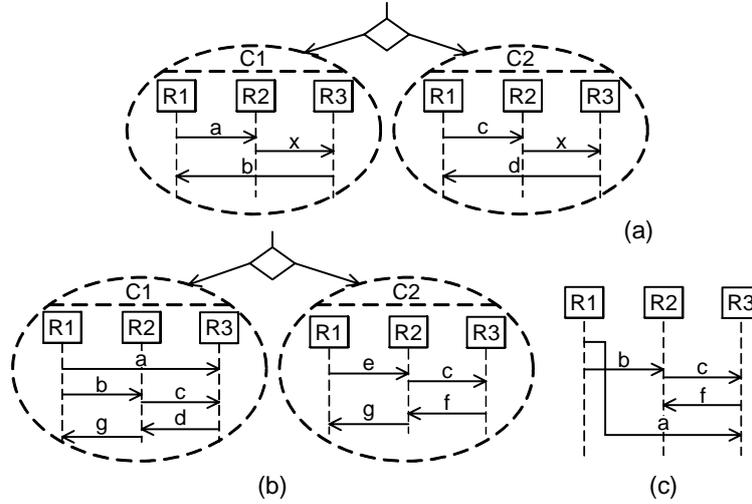


Figure 11.11: (a) Non-deterministic and (b) Race choice propagation; (c) Behavior implied by (b)

send and receive other messages. It is the triggering traces that enable a non-choosing component to determine the alternative chosen by the choosing component. In some cases, however, a non-choosing component may not be able to determine the decision made by the choosing component. As an example, we consider the local choice in Fig. 11.11(a). For the component  $R3$ , the triggering traces for both alternatives are the same (i.e. the reception of message  $x$ ). Therefore, upon reception of  $x$ ,  $R3$  cannot determine whether  $R1$  decided to execute collaboration  $C1$  or  $C2$ . That is,  $R1$ 's decision is ambiguously propagated to  $R3$ . We say a choice has an **ambiguous propagation** if there is a non-choosing component for which the triggering traces specified in two alternatives have a common prefix<sup>6</sup>. Choices with ambiguous propagation are not directly realizable. They are similar to the non-deterministic choices defined in [MRW06].

Now consider the choice in Fig. 11.11(b). It is a local choice and, according to the triggering traces specified for any of the two non-choosing components, the propagation should not be ambiguous. Still, this choice is not directly realizable. A race condition between messages  $a$  and  $c$  in  $C1$  may lead to the scenario of Fig. 11.11(c), where  $R1$  and  $R2$  execute  $C1$ , while  $R3$  executes  $C2$ . This example shows that in the presence of race conditions the triggering trace observed by a non-choosing component may differ from the specified one. Therefore, whenever race conditions may appear in any of the alternatives, we need to consider the potentially observable

<sup>6</sup>Note that this definition considers that there is ambiguous propagation with the following triggering traces:  $\{?x, ?y\}$  and  $\{?x, ?z\}$ . This is true in any directly realized system, since the choice cannot be made immediately after  $?x$ . An easy solution in this case would be to delay the choice (i.e. extract  $?x$  from the choice). Note, however, that this solution would not always be appropriate (e.g. with the following triggering traces:  $\{?x\}$  and  $\{?x, ?z\}$ ).

triggering traces in the analysis of choice propagation (e.g.  $\{?a, ?c\}$  and  $\{?c, ?a\}$  for  $R3$  in collaboration  $C_1$  – Fig. 11.11(b)). We say a choice has a **race propagation** if there is ambiguous propagation due to races. Choices with race propagation are not directly realizable. They are similar to the race choices defined in [MRW06].

To resolve the problem of race propagation we need to eliminate the race(s) that lead to it. However, if we try to remove the race conditions by means of message reordering for consumption (e.g. by means of separate input buffers), the race propagation problem may still persist. This is because, in general, a component would not be able to determine whether a received message should be immediately consumed as part of one alternative, or be kept for later consumption in another alternative (e.g. race propagation in Fig. 11.11(b) cannot be solved with separate input buffers). To make the message reordering work, we need to mark the messages with the collaboration instance<sup>7</sup> they belong to [BG86]. This not only avoids race propagation, but also ambiguous propagation in general. In [GMSZ06], although choice propagation is not explicitly discussed, the authors propose marking all messages (i.e. not only those involved in a race propagation) as just explained, in order to realize local-HMSCs specifications. Components then have to check the data carried by messages upon each reception. We believe this unnecessarily increases the amount of processing that each component has to do upon message reception. We would prefer to detect the cases of race propagation and either remove the race condition(s) by transforming the responsible weak sequencing into strong sequencing, or apply message reordering together with marking only to the messages involved.

Neither ambiguous nor race choice propagation can be detected at the collaboration level<sup>8</sup>, we need to consider the detailed behavior of the sub-collaborations involved in the choice.

A choice without ambiguous or race propagation is said to have **proper decision propagation**. A local choice with proper decision propagation is directly realizable.

### 11.4.3 Interruption

The interruption semantics requires a collaboration  $C$  be interrupted once another preempting collaboration  $C_{int}$  is initiated. In a distributed asynchronous system the interruption may take some time to propagate to all participants in the interrupted collaboration. This means that certain components may still proceed executing their behavior in  $C$  for some time after  $C_{int}$  has been initiated. For example, a client may send a request to a server and, shortly after that, decide to send a cancellation message. While this message is on the way, the server would continue processing the request, and may even send a response back to the client before it receives the cancellation message. The client would then receive a response message that it does not expect. Similarly, the server would receive a non-awaited cancellation message.

As competing-initiatives choices, interruption compositions suffer from a problem of initiatives (from the interrupted and the interrupting collaborations) that compete

<sup>7</sup>If the choice is part of the body of a loop, the iteration number should be considered

<sup>8</sup>In the case of race propagation we may detect the existence of a race at the collaboration level, but could not determine if that race affects the propagation.

with each other. They are therefore not directly realizable, in the general case. Note, however, that the presence of competing initiatives is visible with interruptions, and so the detection is easy at the choreography level. We refer to Section 11.4.2 for a discussion on resolution of competing initiatives situations and related problems.

#### 11.4.4 Parallel Composition

A parallel composition is directly realizable as long as the composed collaborations are completely independent (i.e. their executions do not interfere with each other). Unfortunately, sometimes there are implicit dependencies that may lead to unspecified behaviors. This is the case if a component participates in several concurrent collaborations that use the same message types. Messages belonging to one collaboration may then be consumed within a different collaboration.

Implicit dependencies may also exist through shared resources. In this case, appropriate coordination has to be added between the collaborations, which will normally be service-specific. In [CB06b] we discussed the automatic detection of interactions, due to shared resources, between concurrent instances of the same composite service collaboration. This detection approach makes use of pre- and post-conditions associated with sub-collaborations, and could also be used to detect interactions between collaborations composed in parallel with forks.

#### Forks, Joins and Sequential Composition.

We note that the rules for (weak/strong) sequential composition should be applied, as in any other context, in the presence of forks and joins. If strong sequencing is required, all the collaborations immediately following a fork should be initiated by the component terminating the collaboration preceding the fork. Similarly, all the collaborations immediately preceding a join should terminate at the component initiating the collaboration following the join. If weak sequencing is required, each collaboration immediately following a fork should be initiated by a component participating in the collaboration preceding the fork. Likewise, the component initiating the collaboration following a join should participate in each of the collaborations immediately preceding the join.

We also note that if synchronization behavior is needed in order to guarantee the strong or weak sequencing as explained above (or to remove race conditions), such behavior should be added to the affected branch after the fork, or before the join, in order to prevent interactions with the collaborations in the other branches.

#### 11.4.5 Conflicts between Concurrent Collaboration Instances

So far we have discussed the conflicts that may appear when sub-collaborations are composed within the scope of an enclosing collaboration. In a running system there will normally be many collaborations executing in parallel. One will normally not define the complete system behavior explicitly as one collaboration, but rather let it be implied from the binding (and composition) of roles to components. Here we

briefly discuss the problem of undesired interactions between concurrent instances of the same composite collaboration (e.g. concurrent sessions of a service).

If several instances of a collaboration are executed at the same time, and they involve disjoint sets of components (i.e. the roles of each collaboration instance are bound to different components), they will run independently, without interactions. The situation is however different if one component participates in two or more collaboration instances. In that case, undesired interactions between the collaborations may arise if the roles played by the component in those collaborations need to access shared resources. To avoid such interactions the roles should be properly coordinated. Depending on the kind of resource and on the concrete service requirements, a different mechanism may be needed for their coordination. One may also distinguish between static role binding, which is resolved at design time, and dynamic role binding, which is resolved at runtime. For example, an FTP server may maintain concurrent sessions with different clients. Since the memory at the server is a shared and limited resource, the total number of such concurrent sessions should be restricted to a maximum. A session manager could then be used that would dynamically bind new session roles or reject session requests once the maximum was reached. In a telephone service, where a user may receive a call while already talking to someone, a call-waiting functionality may be a better solution.

In [CB06a, CB06b] we showed that the detection of conflicts between concurrent instances of a composite collaboration can be automatized. This is not elaborated further here, and we just explain the main lines behind the proposed analysis approach. It is based on the fact that a component participates in a composite collaboration instance by playing a certain sequence of sub-roles (i.e. one sub-role for each sub-collaboration in the collaboration choreography the component participates in). When the same component participates in several collaboration instances, it plays several sequences of sub-roles. Undesired interactions may then arise between sub-roles belonging to different sequences (i.e. played in different collaboration instances) due to shared resources. In the proposed analysis approach, collaboration pre-conditions are used to specify the status and availability of the resources needed to execute a collaboration. Post-conditions describe the status and availability of resources after the collaboration execution. The analysis of interactions between the sequences of sub-roles played by a component is performed by constructing all possible interleavings of such sequences. If an interleaving contains two consecutive sub-roles such that the post-condition of the first one contradicts/falsifies the pre-condition of the second one, an interaction is reported.

## 11.5 Algorithms

In the following we present algorithms for the detection of race conditions and choice propagation problems. The input for these algorithms is a slightly modified version of the choreography graph. In a choreography an activity may describe two or more alternative behaviors, each one of them given by a different poset. Depending on the behavior that is executed, a different output pin may be used to pass on the focus of control to another activity (see left side of Fig. 11.12). Activities with these

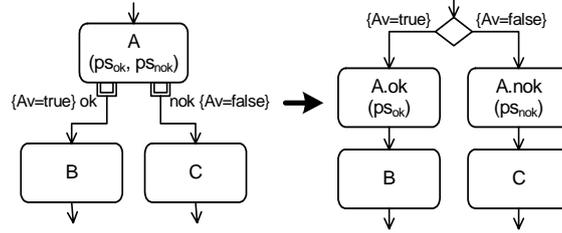


Figure 11.12: Replacement of collaborations with behavior described by several posets

characteristics are replaced with a choice node and a set of new activities. Each new activity will be associated to one of the posets of the original activity. The interconnection of these new activities with the other activities in the graph is made according to the original interconnections ( see right side of Fig. 11.12).

We also assume that the choreography graph does not contain interrupting and invocation compositions.

### 11.5.1 Detection of Race Conditions

For the detection of races it is useful to know when a role  $p$  gets synchronized at a sending event  $s$ , so that no races may happen at  $p$  between a visual predecessor of  $s$  and any of its visual successors. Formally, given a visual order  $<$  and its associated causal order  $\prec$ , we say that a role  $p$  gets synchronized at a sending event  $s \in S_p$  if for any  $e_1 \in \{e \in E_p : e <_p s\}$  and any  $e_2 \in \{e \in E_p : s <_p e\}$ , it holds that  $e_1 \prec e_2$ .

In an MSC, a race condition exists when two events are related by the MSC's visual order but not by its causal order. Building the visual and causal orders requires a transitive closure operation that has quadratic complexity on the number of events. Unfortunately, the detection of races in HMSCs is not so simple. Since an HMSC describes a collection of possible scenarios, it may happen that a race exists in one execution of the HMSC but it is compensated in another execution. That is, two events may be said to happen in one order in one of the HMSCs executions, and in the opposite order in another execution. As a result, the events are not in race. For the general class of HMSCs, the race detection problem is undecidable [MP00, Mus00].

A choreography of collaborations describes exact behavior, rather than existential behavior. Therefore, a race condition will exist in a choreography if it exists in any of its executions. The straightforward approach for the detection of races in a choreography requires to construct the visual and causal orders for all possible execution sequences, which might be computationally costly. In the following we present a set of algorithms for the detection of race conditions that do not require building the visual and causal orders for all possible execution sequences. We assume that all elementary collaborations are send-causal, that is, that their behaviors are

described by sequence diagrams satisfying the send-causality property.

In the main algorithm (see Algorithm 2), we first look for races between events of the same elementary collaboration. For each elementary collaboration  $c$ , we construct the causal order of its events ( $\prec^c$ ). We then check, for each role  $p$ , if there exist any pair of receiving events that are ordered by the total order associated to  $p$  (i.e.  $r_1 <_p r_2$ ) but they are unordered according to the causal order of the collaboration (i.e.  $r_1 \not\prec^c r_2$ ). If that is the case, a race exist between  $r_1$  and  $r_2$  (lines 1-4 of Algorithm 2). In case the sequence diagram of the elementary collaboration contains loops, we consider just one iteration of the loop. If an event inside a loop is in race, we proceed according to the *GetRaceType* procedure (see page 227). Note that we only check for races between receiving events. This is because we assume that all elementary collaborations are send-causal, and according to Proposition 11.22, a receiving event cannot be in race with a sending event when the send-causality property is satisfied. We may have actually used the results of Propositions 11.24 and 11.25 to detect race conditions without constructing the causal order of each elementary collaboration. We nevertheless built the causal orders since they will be needed in other parts of the race detection process.

Once all elementary collaborations have been analyzed, we search for races in the choreography graph, between events that belong to different collaborations. We propose two techniques to detect races in a choreography graph. One is used to detect races when the weak sequencing of collaborations in the choreography graph is send-causal, while the other is used when the weak sequencing is weakly-causal. We discuss these two techniques in the following.

---

**Algorithm 2:** DetectRaces
 

---

**Data:** A composite collaboration  $C$ ; A choreography graph  $(V, E)$

**Result:** A table *EventsInRace* containing in position  $(e_1, e_2)$  a set of collaboration sequences that lead to a race between events  $e_1$  and  $e_2$

```

1 foreach elementary sub-collaboration  $c$  of  $C$  do
2   Construct causal order  $\prec^c$  for  $c$ 
3   foreach role  $p$  of  $c$  and each pair of receiving events  $r_1, r_2 \in E_p^c$  do
4     if  $r_1 <_p r_2 \wedge r_1 \not\prec^c r_2$  then  $EventsInRace[r_1][r_2] \leftarrow \{c\}$ 
5 DetectRacesWithSendCausality()
6 DetectRacesWithWeakCausality()
7 DetectRacesInChainedActSeqs()

```

---

### Races with Send-causal Weak Sequencing

When the weak sequencing of collaborations is send-causal, no sending event may be involved in a race condition (see Propositions 11.20 and 11.22). Only receiving events may be in race with other receiving events. Moreover, based on Propositions 11.24 and 11.25, only the visual order of events specified for a role  $p$  (i.e.  $<_p$ ) needs to be considered in order to detect races between the receiving events performed by that role. That is, with send-causality the detection of races can be performed on a per-role basis, without taking into account the global causal order.

Propositions 11.24 and 11.25 show that two receiving events from the same role

$p$ ,  $r_1$  and  $r_2$ , may be in race only if, according to  $p$ 's visual order,  $p$  does not perform any sending event between the two receiving events (i.e. if  $r_1 <_p r_2$ , and there is not a sending event  $s$  such that  $r_1 <_p s <_p r_2$ ). Therefore, if a choreography describes a sequence of activities<sup>9</sup>  $v_1 \cdot v_2 \cdot \dots \cdot v_n$  ( $1 < n$ ), a race between  $r_1 \in R_p^{v_1}$  (i.e. a receiving event performed by role  $p$  in  $v_1$ ) and  $r_n \in R_p^{v_n}$  may only be possible if the two following conditions are satisfied:

- $p$  plays a terminating sub-role in  $v_1$  (i.e.  $p$  finishes its participation in  $v_1$  with a receiving event) and a non-initiating sub-role in  $v_n$  (i.e.  $p$  starts its participation in  $v_n$  with a receiving event); and
- for each activity  $v_i$ , with  $1 < i < n$ , either  $p$  does not participate in  $v_i$  (i.e.  $R_p^{v_i} = \emptyset$ ) or  $p$  only executes receiving events in  $v_i$  (i.e.  $S_p^{v_i} = \emptyset$ ).

We propose to detect races in two steps. First, algorithm *DetectRacesWithSendCausality* (on page 229) traverses the choreography graph and finds activity sequences of the form  $v_1 \cdot v_2 \cdot \dots \cdot v_n$  ( $n > 1$ ), where  $v_1 \circ_w v_2$  is send-causal<sup>10</sup> and  $p$  only participates in  $v_1$ , playing a terminating sub-role, and in  $v_n$ , playing a non-initiating sub-role in  $v_n$ . For each of these sequences, a check is performed to detect potential races between  $v_1$  and  $v_n$  at role  $p$ <sup>11</sup>. Thereafter, algorithm *DetectRacesInChainedActSeqs* (on page 243) tries to “chain” the activity sequences obtained by *DetectRacesWithSendCausality*. That is, given a sequence of activities  $v_1 \cdot v_2 \cdot \dots \cdot v_n$  ( $n > 1$ ), where  $p$  has no sending event in  $v_n$ <sup>12</sup>, the algorithm looks for any other sequences that starts with  $v_n$ . If a sequence  $v_n \cdot v_{n+1} \cdot \dots \cdot v_m$  ( $m > n$ ) is found, a check is performed to find races between the events of  $v_1$  and  $v_m$ . After that, the process starts again, taking now the concatenated sequence, that is,  $v_1 \cdot v_2 \cdot \dots \cdot v_n \cdot v_{n+1} \cdot \dots \cdot v_m$ , as the initial sequence. To better understand the detection process, consider the sequence diagram in Fig. 11.13, which illustrates a sequence of collaborations<sup>13</sup>. For role  $R2$ , the *DetectRacesWithSendCausality* algorithm would find two sequences of collaborations with the aforementioned characteristics, namely  $v_1 \cdot v_2 \cdot v_3$  and  $v_3 \cdot v_4$  (note that  $R2$  plays a terminating sub-role in  $v_4$ , but  $v_4 \circ_w v_5$  is not send-causal). It would then check for races between the events of  $v_1$  and  $v_3$ , and between the events of  $v_3$  and  $v_4$ . Assuming communication channels with in-order delivery, a race would be detected between  $?m_4$  (in  $v_3$ ) and  $?m_6$  (in  $v_4$ ). The *DetectRacesInChainedActSeqs* algorithm would determine that sequences  $v_1 \cdot v_2 \cdot v_3$  and  $v_3 \cdot v_4$  can be chained (note that  $R2$  has no sending event in  $v_3$ ). It would then check for races between the events of  $v_1$  and  $v_4$ . As a result, a race between  $?m_1$  (in  $v_1$ ) and  $?m_6$  (in  $v_4$ ) would be detected.

<sup>9</sup>For the sake of simplicity we will use the terms activity and (sub-)collaboration interchangeably, since activities in the choreography refer to occurrences of sub-collaborations

<sup>10</sup>The sequencing of the other activities might be either send-causal or weakly-causal.

<sup>11</sup>For the sake of brevity, we will talk about races between two activities  $v$  and  $w$  at a role  $p$ . This should be understood as races between an event  $e_v \in E_p^v$  and an event  $e_w \in E_p^w$ .

<sup>12</sup>This is a requisite to chain sequences obtained by the *DetectRacesWithSendCausality* algorithm. However, as we will see, sequences where  $p$  performs a sending event in  $v_n$  may be chained with sequences obtained by the *DetectRacesWithWeakCausality* algorithm.

<sup>13</sup>The dashed rounded-rectangles are included just for illustration purposes, but are not standard UML

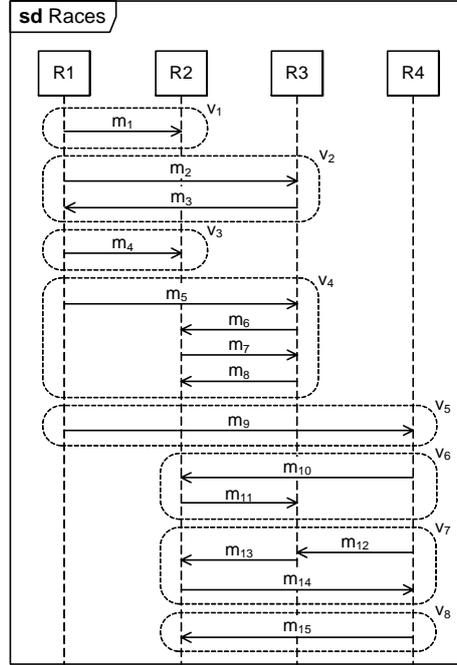


Figure 11.13: Races with send-causal and weakly-causal compositions

In the following we explain in more detail each of the procedures that are used as part of the *DetectRacesWithSendCausality* algorithm.

**Algorithm *DetectRacesWithSendCausality* (on page 229).** This algorithm performs a separate analysis of the choreography graph for each role  $p$  that may be subject to potential races, that is, any role that plays a terminating sub-role in at least one sub-collaboration, and a non-initiating sub-role in at least other (possibly the same) sub-collaboration. Roles that play terminating sub-roles, but do not play non-initiating sub-roles, or roles that do not play any terminating sub-role are not subject to races<sup>14</sup>, so they are not considered during the analysis of the choreography. This information can be obtained from the collaboration diagram.

For each role  $p$  that is subject to potential races, and from each activity  $v_1$  where  $p$  plays a terminating sub-role, the algorithm invokes the *VisitSuccessorSC* procedure (on page 230) to perform a depth-first search (DFS) [AHU74] on the choreography graph. The idea is to find sequences of activities  $v_1 \cdot v_2 \cdot \dots \cdot v_n$  ( $n > 1$ ) where  $p$  only participates in  $v_1$  and  $v_n$  and, thereby, check if there is any race between  $v_1$  and  $v_n$  at role  $p$ .

<sup>14</sup>They may actually have races inside a given collaboration, but not between two collaborations.

**Procedure *VisitSuccessorSC* (on page 230).** This is a recursive procedure that performs a depth-first search [AHU74] on the choreography graph to find sequences of activities  $v_1 \cdot v_2 \cdot \dots \cdot v_n$  ( $n > 1$ ) where  $p$  only participates in  $v_1$  and  $v_n$ . Once such a sequence is found, that is, once an activity  $v_n$  where  $p$  participates is found, procedure *CheckRacesSC* is invoked (line 5). This procedure checks whether there is any race between  $v_1$  and  $v_n$  at role  $p$  (see details on page 227). When *CheckRacesSC* returns, the backtracking process is initiated until a node with unvisited successors is found.

We note that, for a given activity  $v_1$  and a role  $p$ , a possibly infinite number of activity sequences  $v_1 \cdot v_2 \cdot \dots \cdot v_n$  ( $n > 1$ ), where  $p$  only participates in  $v_1$  and  $v_n$ , may be found in the choreography graph. Fortunately, only a subset of these sequences is of interest for our purposes. We note the following:

- (i) In order to detect a race between two activities  $v_1$  and  $v_n$  at role  $p$ , it is sufficient to find one sequence of the form  $v_1 \cdot v_2 \cdot \dots \cdot v_n$  ( $n > 1$ ), where  $p$  only participates in  $v_1$  and  $v_n$ . This has a positive implication on loops: we just need to traverse the body of a loop once, thus avoiding infinite sequences. Therefore, during each traversal of the graph, each activity is visited as most once. For example, in the case of the choreography in Fig. 11.14(a) the algorithm would only return the sequence  $v_1 \cdot v_2 \cdot v_3$ , which is sufficient to detect races between  $v_1$  and  $v_3$ .
- (ii) Given a sequence of activities  $v_1 \cdot v_2 \cdot \dots \cdot v_n$ , a race between  $v_1$  and  $v_n$  is only possible if  $v_1$  and  $v_2$  are composed in weak sequence. Making strong the sequencing between  $v_1$  and  $v_2$  would eliminate any potential race between  $v_1$  and  $v_n$ , but only when  $v_n$  is reached via  $v_2$ . The race may still be possible if  $v_n$  is reached through another path. This means that, given two activities  $v_1$  and  $v_n$ , we are interested in the set of all sequences that start at  $v_1$ , end at  $v_n$ , and have a second activity that is different from the second activity of any other sequence in the set. For example, in the choreography of Fig. 11.14(c) we are interested in two sequences, namely  $v_1 \cdot v_2 \cdot v_3$  and  $v_1 \cdot v'_2 \cdot v_3$ , while in the choreography of Fig. 11.14(b) we are only interested in one sequence, either  $v_1 \cdot v_2 \cdot v_3 \cdot v_5$  or  $v_1 \cdot v_2 \cdot v_4 \cdot v_5$ .

To obtain the set of desired sequences, procedure *VisitSuccessorSC* allows decision nodes to be revisited multiple times, while merge nodes may be revisited only in certain cases. Whenever a merge node is visited, its *visited* flag is set to *true*. In a traditional DFS algorithm, such flag would be reset during backtracking. This does not happen in the proposed algorithm (line 45). Instead, when a decision node is visited, and before visiting any of its successor nodes, the algorithm checks whether *AuxSeq* (i.e. the ordered set of visited activities in the current path) contains only one element (i.e.  $v_1$ ). If that is the case, a different “second activity” will be visited. It is then that the *visited* flag of merge nodes is set to *false* (lines 13-15), so that  $v_n$  can be visited again (if that is possible through the new path). For example, in the choreographies of Figs. 11.14(a) and 11.14(b), the *visited* flag of  $m_1$  would not be reset when visiting  $d_1$ , since *AuxSeq* =  $\{v_1, v_2\}$  at that point. In the case of the choreographies in Figs. 11.14(c) and 11.14(d), *AuxSeq* =  $\{v_1\}$  when visiting  $d_1$ , so the *visited* flag for  $m_1$  would be reset.

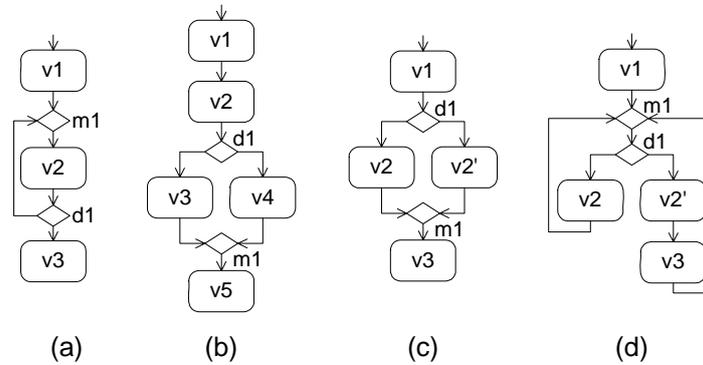


Figure 11.14: Examples of combinations of merge and decision nodes in a choreography graph

The treatment of fork and join nodes by the *VisitSuccessorSC* procedure deserves some explanation. In the following we assume that fork and join nodes are properly nested. That is, all outgoing edges of a fork node lead to the same join node (in the following called the *companion join*), and all incoming edges of a join node come from the same fork node<sup>15</sup>. We note that each branch of a fork (corresponding to each of the fork’s outgoing edges) may define several execution paths. For example, in Fig. 11.15 the fork’s right branch defines two execution paths, namely  $v_4 \cdot v_5$  and  $v_4 \cdot v_6$ .

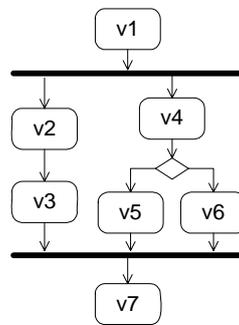


Figure 11.15: Properly nested fork and join nodes in a choreography graph

When a fork node is visited, a search is performed in each of the fork’s branches for activities where  $p$  participates. This is done by invoking the *TraverseForkBranch* procedure for each of the fork’s successor nodes (line 25). This procedure returns three values:

<sup>15</sup>An exception is the following. If the join node is connected to a final node, the former could be removed, and let all outgoing edges of the fork lead to final nodes.

- *continue*: it is a boolean predicate that is *true* when  $p$  does not participate in one or more (possibly all) of the execution paths of the traversed branch, and gets synchronized<sup>16</sup> in the remaining paths. For example, consider that  $p$  does not participate in any of the activities inside the fork-join pair in Fig. 11.15. Then *continue* would be *true* for both branches of the fork. If  $p$  only participates in  $v_6$  and gets synchronized, *continue* would still be *true* for the right branch, but if it does not get synchronized, then *continue* would be *false* for the right branch.
- *auxSynch*: it is a boolean predicate that is *true* when  $p$  gets synchronized in all execution paths of the traversed branch.
- *lastPMRs*: it is a set containing, for each of the execution paths of the traversed branch, the index of the last entry created in  $PMRSeqs_{sc}$ . Note that *TraverseForkBranch* invokes *VisitSuccessorSC*, which may in turn invoke *CheckRacesSC*, where new entries can be added to the  $PMRSeqs_{sc}$  table.

Once all the branches of the fork has been traversed, and if *contAfterFork* is *true* (this only happens if *continue* was *true* for all the fork's branches – see line 28), *VisitSuccessorSC* continues traversing the choreography graph from the fork's companion join node (line 41). Otherwise, if *contAfterFork* is *false*, *VisitSuccessorSC* starts the backtracking process. This may happen in two cases:

- $p$  gets synchronized in one or more of the fork's branches (i.e. *auxSynch* is *true* for each of those branches and *synch* is *true* – see line 27). No races are then possible between activities that precede the fork and activities that succeed the fork's companion join. In this case the *DetectRacesInChainedActSeqs* algorithm should not try to chain activity sequences whose first activity precedes the fork with activity sequences whose last activity succeeds the fork's companion join. For example, consider that, after analysing the graph in Fig. 11.15, *VisitSuccessorSC* stores three activity sequences in  $PMRSeqs_{sc}$ , namely  $v_1 \cdot v_2$ ,  $v_2 \cdot v_3$  and  $v_3 \cdot v_7$ . Assume also that  $p$  gets synchronized in the fork's right branch. Now, the *DetectRacesInChainedActSeqs* algorithm would chain the  $v_1 \cdot v_2$  and  $v_2 \cdot v_3$  sequences, but should not try to chain the resulting sequence (i.e.  $v_1 \cdot v_2 \cdot v_3$ ) with  $v_3 \cdot v_7$ . This is because  $p$  gets synchronized in one of the fork's branches, and no races can thus happen between events of  $v_1$  and events of  $v_7$ . To ensure that *DetectRacesInChainedActSeqs* behaves as expected, the last entries created in  $PMRSeqs_{sc}$  for each of the fork's branches, whose indexes are stored in *lastPMRs*, are “marked”. In the previous example the entry containing  $v_2 \cdot v_3$  in would be marked. This is done with help of the table *synchPMR*, which associate each element in *lastPMRs* with the first activity in *AuxSeq*, which precedes the fork (lines 32-34).

---

<sup>16</sup>A role gets synchronized at a sending event if no races may happen between events preceding and succeeding that sending event. See the explanation of procedure *GetsSynchronized* on page 225 for more details.

- b)  $p$  does not get synchronized in any of the fork's branches, and  $p$  participates in at least one of the activities of one of the fork's branches. In this case races between activities preceding the fork and activities succeeding the fork's companion join are possible, but will be detected by the *DetectRacesInChainedActSeqs* algorithm. Consider the example in Fig. 11.15 and imagine that  $p$  plays initiating sub-roles in  $v_1$ ,  $v_2$  and  $v_7$  ( $p$  does not participate in the other activities). Starting from  $v_1$ , *VisitSuccessorSC* would traverse the fork's left branch, find the sequence  $v_1 \cdot v_2$ , and check for races between  $v_1$  and  $v_2$ . Then it would traverse the fork's right branch. After that, since *continue* would be *false* for the left branch, it would start the backtracking process. Thereafter, starting from  $v_2$ , *VisitSuccessorSC* would find the sequence  $v_2 \cdot v_3 \cdot v_7$ , and check for races between  $v_2$  and  $v_7$ . Potential races between  $v_1$  and  $v_7$  would be found by the *DetectRacesInChainedActSeqs* algorithm when chaining the two sequences  $v_1 \cdot v_2$  and  $v_2 \cdot v_3 \cdot v_7$ .

**Procedure *TraverseForkBranch* (on page 231).** This procedure first invokes the *VisitSuccessorSC* procedure in order to find an activity in a fork's branch where a role  $p$  participates. There situations can then be differentiated:

- $p$  does not participate in any of the execution paths of the fork's branch. Then *joinFound* will be *true* and  $k = 0$ . *TraverseForkBranch* will thus return with  $(true, false, \emptyset)$ .
- $p$  does not participate in some of the execution paths of the fork's branch, and gets synchronized in the other paths. Then *joinFound* will be *true* and  $k > 0$ . *TraverseForkBranch* will again return with  $(true, false, \emptyset)$ , since  $F$  will be an empty set.
- $p$  participates in some of the execution paths. For each of those paths an entry in the  $PMRSeqs_{sc}$  table should have been created by the *CheckRacesSC* procedure, and the index of such entry should have been stored in the *forkPMR* array. Then, for each entry in  $PMRSeqs_{sc}$ , *TraverseForkBranch* is recursively invoked, but this time starting from the successor node of the last activity of the activity sequence stored in  $PMRSeqs_{sc}$ . As a result of the recursion, all the execution paths of the branch will be traversed until either the fork's companion join node or an activity where  $p$  gets synchronized is found. At the end of this process, *lastPMRs* will contain the index of the last entry created in  $PMRSeqs_{sc}$  in each of the execution paths. *synch* will be *true* if  $p$  got synchronized in all execution paths, and *contAfterFork* will be *true* if  $p$  did not participate in one or more (possibly all) of the execution paths, and got synchronized in the remaining paths.

**Procedures *GetsSynchronized* (on page 233) and *SynchronizedRoles* (on page 233).** We say that a role  $p$  gets synchronized at a sending event  $s$ , if no races may happen at  $p$  between a visual predecessor of  $s$  and any of its visual successors. More formally, given a visual order  $<$  and its associated causal order

$\prec$ , we say that a role  $p$  gets synchronized at a sending event  $s \in S_p$  if for any  $e_1 \in \{e \in E_p : e < s\}$  and any  $e_2 \in \{e \in E_p : s < e\}$ , it holds that  $e_1 \prec e_2$ . Intuitively, procedure *GetsSynchronized* determines whether a role  $p$  gets synchronized, at a specific point, in some of the execution paths described by the choreography graph.

Consider that the choreography describes a sequence of activities  $v_1 \cdot \dots \cdot v_i \cdot v_{i+1} \cdot \dots$  ( $i > 0$ ). In general, this sequence of activities defines several alternative execution paths, since each activity may describe several alternative behaviors. Each of those execution paths is represented by a visual order  $(E, <)$  and a causal order  $(E, \prec) = ps_1 \circ_w \dots \circ_w ps_i \circ_w ps_{i+1} \circ_w \dots$  (where each  $ps_i$  is one of the causal partial orders associated with activity  $v_i$ ). A given activity may appear in several sequences of activities, and each of these sequences may define several execution paths. Given an activity  $v_i$ , a causal order  $ps_i = (E_i, \prec_i)$ , and a role  $p$ , procedure *GetsSynchronized* returns *true* if  $p$  gets synchronized at a sending event  $s \in E_i$  in all the choreography's execution paths that contain the behavior described by  $ps_i$ . Otherwise, it returns *false*.

We explain in the following the process of checking whether a role gets synchronized in a given execution path whose visual order is  $(E, <)$  and its causal order is  $(E, \prec) = ps_1 \circ_w \dots \circ_w ps_i \circ_w ps_{i+1} \circ_w \dots \circ_w ps_n$ . Recall that we assume that the sequential composition of activities in the choreography graph is weakly-causal, and that each individual activity is send-causal. This means that any  $ps_i$  in  $(E, \prec)$  is send-causal, and any  $ps_i \circ_w ps_{i+1}$  is weakly-causal. We consider initially the case where activities describe sequential behaviors. Later, we generalize the result to consider the possibility of concurrent behaviors.

Assume that role  $p$  executes at least one sending event in the behavior described by  $ps_i$  (otherwise  $p$  does not get synchronized). Let  $s_p \in E_i$  be the minimum sending event of role  $p$  in  $ps_i$  (i.e. there is no other sending event  $s' \in E_i$  such that  $s' \prec_i s_p$ ). Consider now two events located at  $p$ ,  $e_1$  and  $e_2$ , such that  $e_1 < s_p$  (i.e.  $e_1$  is a visual predecessor of  $s_p$ ) and  $s_p < e_2$  (i.e.  $e_2$  is a visual successor of  $s_p$ ). We know that  $e_1 \prec s_p$ , since a role will not execute a sending until all events that are specified to happen before that sending have been processed. Therefore, to determine whether  $e_1 \prec e_2$  (and thereby determining whether  $p$  gets synchronized at  $s_p$ ), we just need to check whether  $s_p \prec e_2$ . Two cases can be differentiated. If  $e_2 \in E_u$ , given that  $ps_i$  is send-causal, and according to Proposition 11.22, we have that  $s_p \prec e_2$ . If otherwise  $e_2 \notin E_i$ , then it should be the case that  $e_2 \in E_{i+1} \cup \dots \cup E_n$ . Let us take a closer look at this case. Let  $s_j^{min}$  be the minimum sending event in  $ps_j$  ( $1 \leq j \leq n$ ). Let now  $q$  be the initiating role of  $ps_{j+1}$ , that is, the role executing the minimum sending event in  $ps_{j+1}$  (i.e.  $loc(s_{j+1}^{min}) = q$ ), and let  $e_j^{max-q}$  be the maximum event in  $ps_j$  of  $q$ <sup>17</sup>. Since  $ps_j$  is send-causal we have that  $s_j^{min} \prec e$ , for any  $e \in E_j$ , and therefore  $s_j^{min} \prec e_j^{max-q}$  or  $s_j^{min} = e_j^{max-q}$ . Now, since  $e_j^{max-q} \prec s_{j+1}^{min}$  (due to the definition of weak sequencing), we have that  $s_j^{min} \prec s_{j+1}^{min} \prec e'$ , for any  $e' \in E_{j+1}$ . It is easy then to see that  $s_j^{min} \prec e_x$ , for any  $e_x \in E_j \cup \dots \cup E_n$  and any  $j \in \{1 \dots n\}$ . In particular, it must be the case that  $e_i^{max-q} \prec s_{i+1}^{min} \prec e_2$ , where  $loc(e_i^{max-q}) = loc(s_{i+1}^{min}) = q$ . Therefore, to determine

<sup>17</sup>We note that  $q$  will always participate in  $ps_j$ , since  $ps_j \circ_w ps_{j+1}$  is weakly-causal

whether  $s_p \prec e_2$ , we just need to check whether  $s_p \prec e_i^{\max-q}$  or  $s_p = e_i^{\max-q}$ . This is what procedure *SynchronizedRoles* does.

Procedure *SynchronizedRoles* takes also into account that the causal orders may describe concurrent behaviors. In general,  $p$  may have a set *Min* of minimum sending events in  $ps_i$ . There might also be a set  $I$  of initiating roles in  $ps_{i+1}$ , and each of them may have a set  $Max_q$  of maximum events in  $ps_i$ . Then, in order for role  $p$  to get synchronized, it is necessary that for each role  $q \in I$ , there is at least one minimum sending  $s_p \in Min$  and one maximum event  $e_i^{\max-q} \in Max_q$  such that  $s_p \prec e_i^{\max-q}$  or  $s_p = e_i^{\max-q}$ . This ensures that  $s_p \prec s_{i+1}^{\min}$  for each role  $q$ .

**Procedure *GetRaceType* (on page 233).** In case the behavior of an activity contains loops, procedure *CheckRacesSC* considers only one iteration for each of the loops. When a race is detected between two events and one of the events, or both, are inside a loop, a question arises whether all potential instances of those events (when several loop iterations are considered) will be in race. Procedure *GetRaceType* answers that question by classifying a race as *type1* and/or *type2*. This procedure assumes send-causality (i.e. for any loop, the sequential composition of its body with itself, and with the preceding and succeeding behaviors, is send-causal) and, given a receiving event  $r_1$  that is in race with another receiving event  $r_2$  (i.e.  $r_1 <_p r_2$  but  $r_1 \not\prec r_2$ ), differentiates two main cases:

- a)  $r_1$  is a loop event. If there is a sending event inside all loops that contain  $r_1$ , then only the instance of  $r_1$  corresponding to the last iteration of the loops will be in race with  $r_2$ . Otherwise, each instance of  $r_1$  will be in race with  $r_2$ , and the race between  $r_1$  and  $r_2$  is said to be a *type1* loop-race (i.e. *type1* = *true*).
- b)  $r_2$  is a loop event. If there is a sending event inside all loops that contain  $r_2$ , then  $r_1$  is in race only with the instance of  $r_2$  corresponding to the first iteration of the loops. Otherwise,  $r_1$  will be in race with all instances of  $r_2$ , and the race between  $r_1$  and  $r_2$  is said to be a *type2* loop-race (i.e. *type2* = *true*).

**Procedure *CheckRacesSC* (on page 232).** This procedure checks whether there is any race at role  $p$  between events in  $v_1$  (i.e. the first activity in the current sequence of activities) and events in  $u$  (i.e. the first activity after  $v_1$  where role  $p$  participates). The procedure takes into account that activities may describe alternative behaviors, so each activity may have several associated posets, as well as concurrent behaviors, so  $p$  may have several minimum and maximum sendings in each poset. It also takes into account that activities may describe loops. Only one iteration of each loop is considered to build the visual orders used to detect races. The effect of multiple loop iterations is considered by invoking procedure *GetRaceType*.

For race detection the procedure obtains, for each poset of  $v_1$  and each poset of  $u$ , the sets  $R_{\text{race}}^{v_1}$  and  $R_{\text{race}}^u$  of receiving events that might be in race.  $R_{\text{race}}^{v_1}$  (line 4) contains receiving events from  $v_1$  that do not have any sending event as a successor, while  $R_{\text{race}}^u$  (line 13) contains receiving events from  $u$  that do not have any sending event as a predecessor. If non-FIFO channels are used for communication, all these events will be in race. Otherwise, if FIFO channels are used, a receiving event

$r_1 \in R_{\text{race}}^{v_1}$  will be in race with a receiving event  $r_2 \in R_{\text{race}}^u$  if their associated sending events are not located at the same role. Whenever a race is found between two events  $r_1$  and  $r_2$ , the value of *AuxSeq* (i.e. the current sequence of visited activities) is stored in table *EventsInRace* (line 18). In addition, procedure *GetRaceType* is invoked to determine whether the events that are in race are inside a loop and, if so, check whether all potential instances of those events (when several loop iterations are executed) are or not in race. The result of this procedure is stored in the *RaceType* table (line 19).

In addition to checking the existence of races between  $v_1$  and  $u$ , procedure *CheckRacesSC* determines whether races may exist at role  $p$  between  $v_1$  and other activities that may be executed after  $u$ . This information will then be used by procedure *TraverseForkBranch* and algorithm *DetectRacesInChainedActSeqs*.

Races between  $v_1$  and any successor activity of  $u$  might be possible in the following three cases:

- a) If  $u$  describes several alternative behaviors, and  $p$  does not participate in some of them (line 6).
- b) If  $p$  only executes receiving events in one of the possible behaviors described by  $u$  (line 20).
- c) If  $p$  executes a sending event in one of the possible behaviors described by  $u$ , but it does not get synchronized at that sending event<sup>18</sup> (line 24).

In the three cases above, procedure *CheckRacesSC* stores in the *PMRSeqs<sub>sc</sub>* table data that will be used by *DetectRacesInChainedActSeqs* for the actual detection of races (namely *AuxSeq*,  $R_{\text{race}}^{v_1}$ ,  $R_{\text{race}}^u$  and a boolean value specifying whether there was any race between events of  $v_1$  and  $u$ ). In addition, the index of the entry created in *PMRSeqs<sub>sc</sub>* is stored in the *forkPMR* set and in the *PMR<sub>rcv</sub>* set (in cases a and b) or in the *PMR<sub>snd</sub>* set (in case c). Also in the three cases above, a boolean entry in the *synchSB* array is created and set to *false*, meaning that role  $p$  does not get synchronized in the current execution path. Otherwise, if no one of the three above cases applies, an entry in the *synchSB* array is created and set to *true*, meaning that  $p$  gets synchronized in the current execution path.

---

<sup>18</sup>Note that this implies that the sequential composition of  $u$  with one of its succeeding activities in the choreography is weakly-causal.

---

**Algorithm 3:** DetectRacesWithSendCausality

---

**Data:** A choreography graph  $(V, E)$ **Result:** A table *EventsInRace* containing in position  $(e_1, e_2)$  a set of collaboration sequences that lead to a race between events  $e_1$  and  $e_2$ ; Sets *PMR* and *PMRSeqs* that are useful for the detection of races involving several collaboration sequences

```

1 foreach role  $p$  playing both terminating and non-initiating sub-roles do
2   foreach  $v_1 \in V$  where  $p$  plays a terminating sub-role do
3     forall  $v \in V$  do  $visited[v] \leftarrow false$ 
4      $i \leftarrow 0$ ;  $VisitedMerge \leftarrow \emptyset$ 
5      $AuxSeq \leftarrow \{v_1\}$  // Ordered sequence of activities
6      $v \leftarrow$  successor of  $v_1$ 
7      $VisitSuccessorSC(v, false, false, \emptyset, 0)$ 

```

---

---

**Procedure VisitSuccessorSC( $v, forkFound, joinFound, synchSB, k$ )**


---

```

1  visited[v] ← true
2  if v is an activity node then
3    AuxSeq ← AuxSeq ∪ {v}
4    if p participates in v then
5      (synchSB, k) ← CheckRacesSC(v, synchSB, k)
6      AuxSeq ← AuxSeq - {v}; visited[v] ← false // Backtrack
7      return (joinFound, synchSB, k)
8  else if v is a merge node then
9    VisitedMerge ← VisitedMerge ∪ {v}
10 else if v is a decision node then
11   visited[v] ← false // Decision nodes can always be revisited
12   foreach u successor of v do
13     if |AuxSeq| = 1 then
14       forall w ∈ VisitedMerge do visited[w] ← false
15       VisitedMerge ← ∅
16     if !visited[u] then
17       (joinFound, synchSB, k) ← VisitSuccessorSC(u, forkFound, joinFound, synchSB, k)
18   return (joinFound, synchSB, k)
19 else if v is a join node ∧ forkFound then
20   vjoin ← v; visited[v] ← false
21   return (true, synchSB, k) // joinFound is set to true
22 else if v is a fork node then
23   forkFound ← true; vjoin ← null
24   synch ← false; contAfterFork ← true; allLastPMRs ← ∅; AuxSeqold ← AuxSeq
25   foreach successor u of v do
26     (continue, auxSynch, lastPMRs) ← TraverseForkBranch(u, ∅, ∅)
27     allLastPMRs ← allLastPMRs ∪ lastPMRs
28     synch ← synch ∨ auxSynch
29     contAfterFork ← contAfterFork ∧ continue
30     AuxSeq ← AuxSeqold
31   forkFound ← false; joinFound ← false
32   if !contAfterFork then
33     if synch = true then
34       v1 ← first element of AuxSeq
35       forall (u, p, j) ∈ allLastPMRs do synchPMR[u][p][j] ← v1
36       synchSB[k] ← true; k++
37     else
38       // In case the just visited fork is inside another fork-join pair
39       foreach (u, p, j) ∈ allLastPMRs do
40         forkPMR[k] ← (u, p, j); synchSB[k] ← false; k++
41   visited[v] ← false // Backtrack
42   return (false, synchSB, k)
43   AuxSeq ← AuxSeq ∪ {v}
44   v ← vjoin // Continue traversing graph from companion join
45 u ← successor of v
46 if !visited[u] then
47   (joinFound, synchSB, k) ← VisitSuccessorSC(u, forkFound, joinFound, synchSB, k)
48 /* Backtrack
49 if v is NOT a merge node then visited[v] ← false
50 return (joinFound, synchSB, k)

```

---

\*/

---

**Procedure** *TraverseForkBranch*( $v, prevPMR, lastPMRs$ )
 

---

```

/* forkPMR and PMRSeqssc are global variables whose data is set in the
CheckRacesSC procedure */
1 synch ← true; contAfterFork ← false
2 (joinFound, synchSB, k) ← VisitSuccessorSC( $v, true, false, \emptyset, 0$ )
3 if joinFound then synch ← false
4 if joinFound ∧ prevPMR =  $\emptyset$  then contAfterFork ← true
5 if !joinFound ∨ (joinFound ∧ k > 0) then
6   lastPMRs ← lastPMRs − prevPMR
7    $F \leftarrow \{forkPMR[j] : j \in 0 \dots k - 1 \wedge synchSB[j] = false\}$ 
8   foreach ( $w, q, j \in F$ ) do
9     lastPMRs ← lastPMRs ∪ {( $w, p, j$ )}
10    u ← last activity of  $PMRSeqs_{sc}[w][q][j].AuxSeq$ 
11    AuxSeq ← {u}; iold ← i; i ← 0
12    x ← successor of u
13    (continue, auxSynch, lastPMRs) ← TraverseForkBranch( $x, \{(w, q, j)\}, lastPMRs$ )
14    i ← iold
15    synch ← synch ∧ auxSynch
16    if !auxSynch then contAfterFork ← false
17 return (contAfterFork, synch, lastPMRs)

```

---

---

**Procedure CheckRacesSC( $u, \text{synchSB}, k$ )**


---

**Data:** Activity  $u$  with which  $v_1$  (first act. in  $\text{AuxSeq}$ ) could be in race

**Result:**  $\text{EventsInRace}$  and  $\text{RaceType}$  are updated if a race is found; Entries in  $\text{PMR}_{\text{rcv/snd}}$  and  $\text{PMRSeqs}_{\text{sc}}$  are created if  $v_1$  might be in race with an activity following  $u$  (to be used by the  $\text{DetectChainRaces}$  algorithm); Entries in  $\text{forkPMR}$  and  $\text{synch}$  are created for use in the  $\text{TraverseForkBranch}$  procedure

```

1  $v_1 \leftarrow$  first element of  $\text{AuxSeq}$ 
2 foreach visual order ( $E_p^{v_1}, <_p^{v_1}$ ) of  $v_1$  do
3    $S_{\text{max}}^{v_1} \leftarrow \{s \in S_p^{v_1} : \exists s' \in S_p^{v_1}, s <_p^{v_1} s'\}$  // max sending events in  $ps_p^{v_1}$ 
   /* Obtain set  $R_{\text{race}}^{v_1}$  of receiving events from  $ps_p^{v_1}$  that could be in race
   (i.e. all receiving events, except those that according to  $<_p^{v_1}$  precede
   any of the maximum sending events) */
4    $R_{\text{race}}^{v_1} \leftarrow R_p^{v_1} - \{r \in R_p^{v_1} : \exists s_{\text{max}} \in S_{\text{max}}^{v_1}, r <_p^{v_1} s_{\text{max}}\}$ 
5   foreach visual order ( $E_p^u, <_p^u$ ) of  $u$  do
6     if  $E_p^u = \emptyset$  then /*  $p$  does not participate in this alternative of  $u$  */
7        $\text{PMRSeqs}_{\text{sc}}[v_1][p][i] \leftarrow (\text{AuxSeq}, R_{\text{race}}^{v_1}, \emptyset, \text{false})$ ;  $\text{PMR}_{\text{rcv}} \leftarrow \text{PMR}_{\text{rcv}} \cup \{(v_1, p, i)\}$ 
8        $i++$ 
9        $\text{forkPMR}[k] \leftarrow (v_1, p, i)$ ;  $\text{synchSB}[k] \leftarrow \text{false}$ ;  $k++$ 
10    else
11       $\text{race} \leftarrow \text{false}$ 
12       $S_{\text{min}}^u \leftarrow \{s \in S_p^u : \exists s' \in S_p^u, s' <_p^u s\}$  // min sending events in  $ps_p^u$ 
      /* Obtain set  $R_{\text{race}}^u$  of receiving events from  $ps_p^u$  that could be in
      race (i.e. all receiving events, except those that according to
       $<_p^u$  happen after any of the minimum sending events) */
13       $R_{\text{race}}^u \leftarrow R_p^u - \{r \in R_p^u : \exists s_{\text{min}} \in S_{\text{min}}^u, s_{\text{min}} <_p^u r\}$ 
      // Check races
14      foreach  $r_{v_1} \in R_{\text{race}}^{v_1}$  do
15        foreach  $r_u \in R_{\text{race}}^u$  do
16          if non-FIFO OR (FIFO AND  $\text{loc}(\text{snd}(e_{v_1})) \neq \text{loc}(\text{snd}(e_u))$ ) then
17             $\text{race} \leftarrow \text{true}$ 
18             $\text{EventsInRace}[r_{v_1}][r_u] \leftarrow \text{EventsInRace}[r_{v_1}][r_u] \cup \{\text{AuxSeq}\}$ 
            /* Check if the race involves events inside loops */
19             $\text{RaceType}[r_{v_1}][r_u] \leftarrow \text{GetRaceType}(r_{v_1}, r_u, S_{\text{max}}^{v_1}, S_{\text{min}}^u)$ 
20      if  $S_{\text{min}}^u = \emptyset$  then //  $p$  has only receiving events in  $E_p^u$ 
21         $\text{PMRSeqs}_{\text{sc}}[v_1][p][i] \leftarrow (\text{AuxSeq}, R_{\text{race}}^{v_1}, R_{\text{race}}^u, \text{race})$ 
22         $\text{PMR}_{\text{rcv}} \leftarrow \text{PMR}_{\text{rcv}} \cup \{(v_1, p, i)\}$ ;  $i++$ 
23         $\text{forkPMR}[k] \leftarrow (v_1, p, i)$ ;  $\text{synchSB}[k] \leftarrow \text{false}$ ;  $k++$ 
24      else if NOT GetsSynchronized( $p, u, ps_p^u$ ) then
25         $\text{PMRSeqs}_{\text{sc}}[v_1][p][i] \leftarrow (\text{AuxSeq}, R_{\text{race}}^{v_1}, R_{\text{race}}^u, \text{race})$ 
26         $\text{PMR}_{\text{snd}} \leftarrow \text{PMR}_{\text{snd}} \cup \{(v_1, p, i)\}$ ;  $i++$ 
27         $\text{forkPMR}[k] \leftarrow (v_1, p, i)$ ;  $\text{synchSB}[k] \leftarrow \text{false}$ ;  $k++$ 
28      else
        /* Role  $p$  gets synchronized. No possibility of race at  $p$ 
        between  $v_1$  and any activity following  $u$ , so no  $\text{PMR}$  and
         $\text{PMRSeqs}$  entries needed. */
29       $\text{synchSB}[k] \leftarrow \text{true}$ ;  $k++$ 
30 return ( $\text{synchSB}, k$ )

```

---

---

**Procedure GetRaceType( $r_{v_1}, r_u, S_{max}^{v_1}, S_{min}^u$ )**


---

```

1 (type1,type2) ← (false,false) // No special race
2 if  $r_{v_1}$  is inside one or more nested loops then
3   /* There will be one instance of  $r_{v_1}$  for each loop iteration */
4   if  $\exists s_{max} \in S_{max}^{v_1}$  such that  $s_{max}$  is contained by all nested loops that contain  $r_{v_1}$  then
5     /* All instances of  $r_{v_1}$  are in race with  $r_u$  */
6     type1 ← true
7 if  $r_u$  is inside one or more nested loops then
8   /* There will be one instance of  $r_u$  for each loop iteration */
9   if  $\exists s_{min} \in S_{min}^u$  such that  $s_{min}$  is contained by all nested loops that contain  $r_u$  then
10    /*  $r_{v_1}$  is in race with all instances of  $r_u$  */
11    type2 ← true
12 return (type1,type2)

```

---



---

**Procedure GetsSynchronized( $p, u, ps_u$ )**


---

```

Result: true if  $p$  gets synchronized for all successor posets of  $ps_u$ . false otherwise
1 result ← true
2 foreach successor activity  $w$  of  $u$  do
3   foreach poset  $ps_w$  of  $w$  do
4     if SynchronizedRoles( $ps_u, ps_w, \{p\}$ ) =  $\emptyset$  then
5       result ← false
6 return result

```

---



---

**Procedure SynchronizedRoles( $ps_1, ps_2, \mathcal{R}$ )**


---

```

/* We assume  $ps_1 = (S_1 \cup R_1, <_1)$  */
1  $\mathcal{R}_{synch} \leftarrow \emptyset$ 
/* Get subset  $\mathcal{R}_{snd}$  of roles from  $\mathcal{R}$  that have a sending event in  $ps_1$  */
2  $\mathcal{R}_{snd} \leftarrow \{loc(s) : loc(s) \in \mathcal{R} \wedge s \in S_1\}$ 
3 if  $\mathcal{R}_{snd} \neq \emptyset$  then
4   if  $ps_1 \circ_w ps_2$  is weakly-causal then
5      $I \leftarrow \{loc(e) : e \in min(ps_2)\}$  /* Initiating roles of  $ps_2$  */
6     foreach  $q \in I$  do
7        $Max[q] \leftarrow \{e : e \in max(ps_1) \wedge loc(e) = q\}$  /* Max events in  $ps_1$  of  $q$  role */
8     foreach  $p \in \mathcal{R}_{snd}$  do
9        $Min \leftarrow \{s : s \in min((S_1, <_1)) \wedge loc(s) = p\}$  /* Min sendings in  $ps_1$  of  $p$  */
10      if  $\forall q \in I, \exists m \in Max[q], \exists s \in Min$  such that  $s <_1 m \vee s = m$  then
11         $\mathcal{R}_{synch} \leftarrow \mathcal{R}_{synch} \cup \{p\}$ 
12   else /* Send-causal sequencing */
13      $\mathcal{R}_{synch} \leftarrow \mathcal{R}_{snd}$ 
14
15 return  $\mathcal{R}_{synch}$ 

```

---

### Races with Weakly-causal Weak Sequencing

When the weak sequencing of collaborations is weakly-causal, the global causal order of events has to be considered for race detection. Consider again the example in Fig. 11.13. The sequential composition  $v_4 \circ_w v_5$  is weakly-causal. It is easy to see that events happening in  $v_4$  after  $!m_5$  (i.e. the events performed by  $R_2$  and  $R_3$ ) may be in race with other events in  $v_5, v_6, v_7$  and  $v_8$ . Let us focus on role  $R_2$ , which performs three events in  $v_4$  after  $!m_1$ , namely  $?m_6, !m_7$  and  $?m_8$ . Just by looking at the local ordering of events in the lifeline of  $R_2$  we cannot determine whether, for example,  $?m_6$  is in race with  $?m_{10}, ?m_{13}$  or  $?m_{15}$ . To find this out we need to consider the causal order between the events of  $R_2$  and the events performed by the other roles. Fortunately, the total number of events that we need to consider in order to build such causal order can be limited. In this case, for example, we do not need to consider the events of  $v_8$  (and of any other collaboration that may succeed  $v_8$ ) in order to detect races at  $R_2$ . This is because  $R_2$  gets synchronized at sending event  $!m_{14}$  in  $v_7$  (see explanation on page 225), so no races may happen at  $R_2$  between events preceding and succeeding  $!m_{14}$  in  $R_2$ 's lifeline.

In the following we explain in more detail the *DetectRacesWithWeakCausality* algorithm and each of the procedures that are used by this algorithm.

**Algorithm *DetectRacesWithWeakCausality* (on page 237).** For each causal poset  $ps_1 = (E_1, <_1)$  of an activity  $v_1$ , and each causal poset  $ps_2$  of an activity  $v_2$ , such that  $ps_1 \circ_w ps_2$  is weakly-causal, this algorithm finds race conditions involving events of  $E_1$ . For that, it first obtains the set  $\mathcal{R}$  of roles that are subject to potential races, that is, roles whose events may be in race with other events (lines 4-9). Intuitively, these are roles that may execute an event from  $E_1$  when the behavior described by  $ps_2$  has already been started. Given an initiating-role  $q$  of  $ps_2$ , and a maximum event  $m$  of  $q$  in  $ps_1$  (if  $m$  is a receiving event, its associated sending event is considered instead), a role may execute an event  $e \in E_1$  after  $q$  has initiated  $ps_2$  if  $e$  will always be executed after  $m$  (i.e.  $m <_1 e$ ), or if  $e$  and  $m$  may be executed in any order (i.e.  $m \not<_1 e$  and  $e \not<_1 m$ ). We note, however, that the algorithm only considers roles that may execute a sending event from  $ps_1$  when  $ps_2$  has already been initiated (line 9). This is because races affecting a role that may execute receiving events from  $ps_1$ , but not sending events, when  $ps_2$  has already been initiated, are already detected by the *DetectRacesWithSendCausality* algorithm.

Once the set of roles subject to potential races has been obtained, the algorithm invokes the *VisitSuccessorWC* procedure to perform a depth-first search on the choreography graph. This procedure returns an array *SEQS* (via a global variable) whose elements are tuples of the form  $(seq, ps, \mathcal{R}')$ , where  $seq = v_1 \cdot v_2 \cdot \dots \cdot v_n$  is a sequence of activities,  $ps = ps_1 \circ_w ps_2 \circ_w \dots \circ_w ps_n$  ( $n \geq 2$ ) is the causal poset corresponding to one of the execution paths associated to  $seq$ , and  $\mathcal{R}' \subseteq \mathcal{R}$  is the set or roles that did not get synchronized in any of the posets in  $ps$ . Normally,  $\mathcal{R}'$  will be the empty set, unless a final node or an infinite loop was found in the choreography (see the explanation of *VisitSuccessorWC*).

After *VisitSuccessorWC* returns, the *CheckRacesWC* procedure is invoked to detect races with help of the causal posets previously obtained.

**Procedure *VisitSuccessorWC* (on page 238).** This is a recursive procedure that performs a depth-first search [AHU74] in the choreography graph.

Control nodes, as well the posets of activity nodes, can be “visited” at most twice. This avoids infinite sequences in the presence of loops, while it ensures that all combinations of activities that are interesting for the detection of races are considered.

Each time an activity node is visited, each one of its posets is visited. The procedure then checks whether any role in  $\mathcal{R}$  got synchronized in the previously visited poset (when that poset is weak sequenced with the current one) and, if so, it removes the roles that got synchronized from  $\mathcal{R}$  (line 7). When all roles in  $\mathcal{R}$  get eventually synchronized (and if the *done* flag is *false*), the current sequence of visited activities, and the causal poset corresponding to the sequence of visited posets are stored in the *SEQS* array (by invoking the *ProcessSeq* procedure in line 10). The *done* flag is done to avoid having several equal entries in the *SEQS* array, in case  $\mathcal{R}$  becomes empty for several posets of the same activity. If not all roles got synchronized, the current activity and poset are added to the sequences of visited activities and posets, and the traversal of the graph continues by recursively invoking *VisitSuccessorWC* (line 13). The backtracking process starts in the following cases:

- If a final node is found or all roles in  $\mathcal{R}$  get synchronized. In that case, the *ProcessSeq* procedure is invoked to store in the *SEQS* array the current sequence of visited activities, as well as the causal poset corresponding to the sequence of visited posets.
- If a join node is found. This is done as part of the special treatment of fork-join pairs, as it is explained later on.
- In the presence of loops, when a third attempt to traverse the body of the loop is made. In this case the *ProcessSeq* procedure might or not be invoked. Consider the loop in Fig. 11.16(a). After visiting  $d_1$  twice, *VisitSuccessorWC* may try to visit  $m_1$  a third time with no success. It would then go back to  $d_1$  and continue thereafter to  $v_3$  without invoking *ProcessSeq*. Consider now the loop in Fig. 11.16(b). Here, after visiting  $v_2$  twice, *VisitSuccessorWC* may try to visit  $m_1$  a third time. As in the previous example, it would not succeed and the backtracking process would be initiated. However, before that, *VisitSuccessorWC* would invoke the *ProcessSeq* procedure (line 14). As a result, the activity sequence  $v_1 \cdot v_2 \cdot v_2$  and the causal poset  $ps_1 \circ_w ps_2 \circ_w ps_2$  would be stored in the *SEQS* array.

When a fork node is visited, the *MapForkWC* procedure is invoked (line 26). This procedure returns two sets, *SynchPaths* and *UnsynchPaths*, whose elements are tuples of the form  $(seq, ps, \mathcal{R}')$  (i.e. the same type of tuples stored in the *SEQS* array). The elements in *SynchPaths* describe execution paths through the fork where all roles in  $\mathcal{R}$  get synchronized. For each of these paths the *ProcessSeq* procedure is invoked (line 32) and the backtracking process is initiated. The elements in *UnsynchPaths* describe execution paths through the fork where not all roles in  $\mathcal{R}$  get synchronized. For each of these paths, the *VisitSuccessorWC* procedure is invoked (line 37) to continue traversing the choreography graph from the fork’s companion join node.

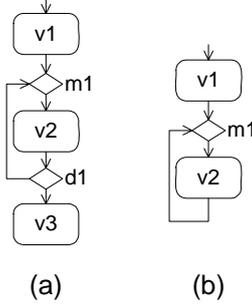


Figure 11.16: Two examples of loops in a choreography graph

**Procedure *MapForkWC* (on page 239).** For each branch of a fork-join pair, this procedure traverses all the possible execution paths, and generates a causal poset and a sequence of activities for each of them (lines 1-1). Thereafter, it groups the paths in sets  $CP$  of concurrent paths (the *getsComb* function, shown below, is used for this – line 9). For example, two sets of concurrent paths would be generated for the fork-join pair in Fig. 11.15, one for  $v2 \cdot v3$  and  $v4 \cdot v5$ , and another for  $v2 \cdot v3$  and  $v4 \cdot v6$ . The procedure generates a causal order for each set of concurrent paths, as well as a corresponding activity sequence, and stores them in the *SynchPaths* set, if all roles in  $\mathcal{R}$  get synchronized in at least one of the concurrent paths, or in the *UnsynchPaths* set, otherwise.

$$getCombs(\mathbb{S}) = \begin{cases} \emptyset, & \text{if } \mathbb{S} = \emptyset \\ \{\{s\} : s \in \mathbb{S}\}, & \text{if } \mathbb{S} = \{S\} \\ \{\{e\} \cup Comb : e \in S \wedge Comb \in getCombs(S')\}, & \text{if } \mathbb{S} = S' \cup \{S\} \end{cases}$$

**Procedure *CheckRacesWC* (on page 240).** This procedure analyzes the causal posets generated by the *VisitSuccessorWC* procedure (and stored in the global variable *SEQS*) in order to find race conditions. Each of those causal posets describes an execution path through a sequence of activities  $v_1 \cdot v_2 \cdot \dots \cdot v_n$ , and is of the form  $(E_{\prec}, \prec) = ps_1 \circ_w ps_2 \circ_w \dots \circ_w ps_n$  ( $n \geq 2$ ), where  $ps_1 \circ_w ps_2$  is weakly-causal.

*CheckRacesWC* checks whether any event  $e \in E_1$  (if  $ps_1 = (E_1, \prec_1)$ ) is in race with any receiving event  $r \in E_{\prec} - E_1$ . This is done by checking whether  $e$  is a causal predecessor of  $r$ . If it is not, the events are in race, and the activity sequence where such race may happen is stored in the *EventsInRace* table. Note that the same poset may appear twice in  $ps_1 \circ_w ps_2 \circ_w \dots \circ_w ps_n$ , if its activity is inside a loop in the choreography graph. In that case the second occurrence of the poset will have their events relabeled and marked as loop events. This fact is used by the *CheckRacesWC* procedure to determine the “type of loop race” (see explanation on page 227).

**Algorithm 10:** DetectRacesWithWeakCausality

---

**Data:** A choreography graph  $(V, E)$   
**Result:**  $x$

```

1 foreach pair  $(v_1, v_2)$  of activities whose sequential composition is weakly-causal do
   /*  $\llbracket SD \rrbracket_{SD}^{WC}$  instantiates each loop of SD with two iterations */
2   foreach poset  $ps_1 = (S_1 \cup R_1, \prec_1, t_1) \in \llbracket SD_{v_1} \rrbracket_{SD}^{WC}$  of  $v_1$  do
3     foreach poset  $ps_2 \in \llbracket SD_{v_2} \rrbracket_{SD}^{WC}$  of  $v_2$  such that  $ps_1 \circ_w ps_2$  is weakly causal do
4       /* Obtain the set  $\mathcal{R}$  of roles that are subject to potential races
5         due to weak causality */
6        $\mathcal{R} \leftarrow \emptyset$ ;  $MaxEv \leftarrow \emptyset$ 
7       foreach initiating-role  $q$  of  $ps_2$  do
8         foreach maximum event  $m \in \max(ps_1)$  such that  $loc(m) = q$  do
9           if  $m$  is a receiving event then  $m \leftarrow snd(m)$ 
10           $MaxEv \leftarrow MaxEv \cup \{m\}$ 
11           $\mathcal{R} \leftarrow \mathcal{R} \cup \{loc(s) \neq q : s \in S_1 \wedge (m \prec_1 s \vee (m \not\prec_1 s \wedge s \not\prec_1 m))\}$ 
12
13       if  $\mathcal{R} \neq \emptyset$  then
14         /* Now we start a DFS on the graph, looking for activities
15         where roles subject to races participate */
16         forall  $v \in V$  do
17           if  $v$  is an activity node then
18              $visited[ps_v] \leftarrow 0$ , for each poset  $ps_v$  of  $v$ 
19           else
20              $visited[v] \leftarrow 0$ 
21
22          $forks \leftarrow 0$ ;  $i \leftarrow 0$ ;  $SEQS \leftarrow \emptyset$ 
23          $AuxSeqAct \leftarrow \{v_1, v_2\}$ ;  $AuxSeqPS \leftarrow \{ps_1, ps_2\}$ 
24          $visited[ps_1] ++$ ;  $visited[ps_2] ++$ 
25          $seqProcessed \leftarrow false$ 
26          $VisitSuccessorWC(u, \mathcal{R}, ps_2)$  //  $u$  is  $v_2$ 's successor
27          $CheckRacesWC(v_1, S_1 \cup R_1, MaxEv, \mathcal{R})$ 

```

---

---

**Procedure VisitSuccessorWC**( $v, \mathcal{R}, ps_{prev}$ )
 

---

```

1  if  $v$  is an activity node then
2     $\mathcal{R}_{bck} \leftarrow \mathcal{R}$  /* Backup to have fresh data for each poset          */
3     $done \leftarrow false$ 
4    foreach poset  $ps_v \in \llbracket SD_v \rrbracket_{SD}^{WC}$  of  $v$ , such that  $visited[ps_v] < 2$  do
5       $\mathcal{R} \leftarrow \mathcal{R}_{bck}$ 
6       $visited[ps_v] ++$ 
7      /* Check if any role gets "synch" with a sending event          */
8       $\mathcal{R} \leftarrow \mathcal{R} - SynchronizedRoles(ps_{prev}, ps_v, \mathcal{R})$ 
9      if  $\mathcal{R} = \emptyset \wedge !done$  then
10        $done \leftarrow true$ 
11        $ProcessSeq(AuxSeqPS, AuxSeqAct, \mathcal{R})$ 
12     else if  $\mathcal{R} \neq \emptyset$  then
13       /* NOTE: If  $visited[ps_v] = 2$ , relabel events of  $ps_v$  as loop events */
14        $AuxSeqPS \leftarrow AuxSeqPS \cup \{ps_v\}; AuxSeqAct \leftarrow AuxSeqAct \cup \{v\}$ 
15        $VisitSuccessorWC(u, \mathcal{R}, ps_v)$  //  $u$  is  $v$ 's successor
16       if !seqProcessed then  $ProcessSeq(AuxSeqPS, AuxSeqAct, \mathcal{R})$ 
17        $AuxSeqPS \leftarrow AuxSeqPS - \{ps_v\}; AuxSeqAct \leftarrow AuxSeqAct - \{v\}$ 
18      $visited[ps_v] --$ 
19   else if  $visited[v] < 2$  then
20      $visited[v] ++$ 
21     if  $v$  is an activity-final or flow-final node then
22        $ProcessSeq(AuxSeqPS, AuxSeqAct, \mathcal{R})$ 
23     else if  $v$  is a join node AND forks  $> 0$  then
24        $ProcessSeq(AuxSeqPS, AuxSeqAct, \mathcal{R})$ 
25        $v_{join} \leftarrow v$ 
26     else if  $v$  is a fork node then
27       forks  $++$ ;  $v_{join} \leftarrow null$ 
28       ( $SynchPaths, UnsynchPaths$ )  $\leftarrow MapForkWC(v, \mathcal{R}, ps_{prev})$ 
29       forks  $--$ 
30       if  $v_{join} = null$  then
31         /* Either all fork branches got synchronized before reaching the
32         associated join node, or all branches ended up in a final node.
33         In any case we are finished.          */
34          $SynchPaths \leftarrow SynchPaths \cup UnsynchPaths$ 
35          $UnsynchPaths \leftarrow \emptyset$ 
36       foreach ( $forkActs, ps_{fork}, \mathcal{R}_{fork}$ )  $\in SynchPaths$  do
37          $ProcessSeq(AuxSeqPS \cup \{ps_{fork}\}, AuxSeqAct \cup \{forkActs\}, \mathcal{R}_{fork})$ 
38        $v_{join\_succ} \leftarrow$  successor of  $v_{join}$ 
39       foreach ( $forkActs, ps_{fork}, \mathcal{R}_{fork}$ )  $\in UnsynchPaths$  do
40          $AuxSeqPS \leftarrow AuxSeqPS \cup \{ps_{fork}\}; AuxSeqAct \leftarrow AuxSeqAct \cup \{forkActs\}$ 
41         seqProcessed  $\leftarrow false$ 
42          $VisitSuccessorWC(v_{join\_succ}, \mathcal{R}_{fork}, ps_{fork})$ 
43     else
44       foreach  $u$  successor of  $v$  do
45         seqProcessed  $\leftarrow false$ 
46          $VisitSuccessorWC(u, \mathcal{R}, ps_{prev})$ 
47      $visited[v] --$ 
48   return

```

---

---

**Procedure ProcessSeq**(*AuxSeqPS*, *AuxSeqAct*,  $\mathcal{R}$ )

---

```

1 SEQS[i]  $\leftarrow$  (AuxSeqAct, CausalOrderSeq(AuxSeqPS),  $\mathcal{R}$ )
2 i ++
3 seqProcessed  $\leftarrow$  true
4 return

```

---



---

**Procedure MapForkWC**(*v<sub>fork</sub>*,  $\mathcal{R}$ , *ps<sub>prev</sub>*)

---

```

1 AuxSeqPSold  $\leftarrow$  AuxSeqPS; AuxSeqActold  $\leftarrow$  AuxSeqAct; iold  $\leftarrow$  i
2 Paths  $\leftarrow$   $\emptyset$ ;
3 foreach successor u of vfork do
4   AuxSeqPS  $\leftarrow$   $\emptyset$ ; AuxSeqAct  $\leftarrow$   $\emptyset$ ; i  $\leftarrow$  iold
5   VisitSuccessorWC(u,  $\mathcal{R}$ , psprev)
6   /* SEQS is a global variable updated by VisitSuccessorWC */
7   Paths  $\leftarrow$  Paths  $\cup$  { {SEQS[j] : iold  $\leq$  j < i} }
7 AuxSeqPS  $\leftarrow$  AuxSeqPSold; AuxSeqAct  $\leftarrow$  AuxSeqActold; i  $\leftarrow$  iold
8 SynchPaths  $\leftarrow$   $\emptyset$ ; UnsynchPaths  $\leftarrow$   $\emptyset$ 
9 foreach CP  $\in$  getCombs(Paths) do
10  psfork  $\leftarrow$  CausalOrderPar({ (ps : (seq, ps,  $\mathcal{R}_p$ )  $\in$  CP) } )
11  forkActs  $\leftarrow$   $\bigparallel_{(seq, ps, \mathcal{R}_p) \in CP}$  seq
12  if  $\exists$ (seq, ps,  $\mathcal{R}_p$ )  $\in$  CP such that  $\mathcal{R}_p = \emptyset$  then
13    /* If roles get synchronized in a path, they do it for the set of
14    concurrent paths */
15    SynchPaths  $\leftarrow$  SynchPaths  $\cup$  { (forkActs, psfork,  $\emptyset$ ) }
14  else
15    UnsynchPaths  $\leftarrow$  UnsynchPaths  $\cup$  { (forkActs, psfork, {  $\mathcal{R}_p$  : (seq, ps,  $\mathcal{R}_p$ )  $\in$  CP }) }
16 return (SynchPaths, UnsynchPaths)

```

---

---

**Procedure** CheckRacesWC( $v_1, E_1, \text{MaxEv}, \mathcal{R}$ )
 

---

```

1   $PMR_{wc} \leftarrow \emptyset; n \leftarrow 0$ 
2  foreach  $j \in \{0 \dots i\}$  do
3     $(ActSeq, (E_{\prec}, \prec), \mathcal{R}') \leftarrow SEQS[j]$ 
4     $R_{tail} \leftarrow \{e \in (E_{\prec} - E_1) : e \text{ is a receiving event}\}$ 
5    foreach  $p \in \mathcal{R}$  do
6      foreach  $e \in E_1$ , such that  $loc(e) = p$  do
7        foreach  $r \in R_{tail}$ , such that  $loc(r) = p$  do
8          if  $e \neq r$  then
9            if  $e$  or  $r$  are marked as a "loop event" then
10              /*  $e_{type}$  (resp.  $r_{type}$ ) is the event type of which  $e$ 
11                 (resp.  $r$ ) is an instance */
12               $(type1, type2) \leftarrow RaceType[e_{type}][r_{type}]$ 
13              if  $e$  is marked as a "loop event" then
14                 $type1 \leftarrow true$ 
15              if  $r$  is marked as a "loop event" then
16                 $type2 \leftarrow true$ 
17               $RaceType[e_{type}][r_{type}] \leftarrow (type1, type2)$ 
18            else
19               $EventsInRace[e][r] \leftarrow EventsInRace[e][r] \cup \{SEQS[j].ActSeq\}$ 
20
21     $PMRSeqs_{wc}[v_1][p][n] \leftarrow SEQS[j]$ 
22     $PMR_{wc} \leftarrow PMR_{wc} \cup \{(v_1, p, n)\}$ 
23     $n++$ 

```

---

### Races in Chained Activity Sequences

**Algorithm *DetectRacesInChainedActSeqs* (on page 243).** Given an activity sequence  $seq_1 = v_1 \cdot v_2 \cdot \dots \cdot v_n$  ( $n > 1$ ), obtained by the *DetectRacesWithSendCausality* algorithm<sup>19</sup>, the *DetectRacesInChainedActSeqs* algorithm looks for other activity sequences whose first activity is  $v_n$ , that is, sequences of the form  $seq_2 = v_n \cdot v_{n+1} \cdot \dots \cdot v_m$  ( $m > n$ ). Thereafter, it looks for races between  $v_1$  and  $v_m$ , if  $seq_2$  was obtained by the *DetectRacesWithSendCausality* algorithm. Otherwise, if  $seq_2$  was obtained by the *DetectRacesWithWeakCausality* algorithm, it looks for races between  $v_1$  and any of the activities in  $seq_2$ . In the former case the matching and detection processes are repeated again, now with  $seq_1 = v_1 \cdot v_2 \cdot \dots \cdot v_n \cdot v_{n+1} \cdot \dots \cdot v_m$ .

The matching and detection processes described above are indeed performed by the *CheckChainsSC* and *CheckChainsWC* procedures, which we explain in the following.

**Procedure *CheckChainsSC* (on page 244).** This procedure tries to chain the activity sequences that were obtained by the *DetectRacesWithSendCausality* algorithm (so it assumes that the sequential composition of activities is send-causal). When two or more sequences are chained, the procedure checks for races between the first and the last activities of the resulting sequence. The procedure also detects whether the activities in races are inside loops. If two activities  $v_1$  and  $v_2$ , such that  $v_1$  is executed before  $v_2$ <sup>20</sup>, are in race, and  $v_1$  is inside a loop, *LoopRaceType1*[ $v_1$ ][ $v_2$ ] is set to *true*. This means that if an event  $e_1$  from  $v_1$  is in race with and event  $e_2$  from  $v_2$ , all instances of  $e_1$  (due to the loop iterations) will be in race with  $e_2$ . If  $v_2$  is inside a loop, *LoopRaceType2*[ $v_1$ ][ $v_2$ ] is set to *true*. This means that if an event  $e_1$  from  $v_1$  is in race with and event  $e_2$  from  $v_2$ ,  $e_1$  will be in race with all instances of  $e_2$ . To detect the existence of loops, when an activity sequence is considered for concatenation, its last activity is marked as “visited”, as well as added to the *visitedSeq* set, which is an ordered set of visited activities (line 8). Activities inside loops are stored in the *ActInsideLoop* set. In addition, the procedure uses the *ActInRace* set to keep record of the activities that are in race.

The procedure receives as input an activity sequence  $seq_1$ , which starts with an activity  $v$  and ends with an activity  $u$ . Using the indexes stored in  $PMR_{rcv}$  and  $PMR_{snd}$ , the procedure finds activity sequences that start with  $u$ . When a sequence  $seq_2$  starting with  $u$  is found, its last activity (in the following  $w$ ) is extracted. The procedure then checks whether  $w$  has already been visited (line 7). If not, it marks it as visited and checks whether  $v$  and  $w$  are the same activity. If that is the case,  $v$  is inside a loop, so it is added to the *ActInsideLoop* set. If any activity is in race with  $v$  the appropriate *LoopRaceType1* flag is set to *true* (lines 9-11). After that, the procedure checks whether there are any races between events of  $v$  and

<sup>19</sup>Recall that in the *CheckRacesSC* procedure (pages 227 and 232), activity sequences (together with additional data) were stored in the  $PMRSeqs_{sc}$  table, and that the indexes of the entries created in that table were stored in either the  $PMR_{rcv}$  set or the  $PMR_{snd}$  set. Activity sequences obtained by the *DetectRacesWithSendCausality* algorithm can thus be retrieved via the elements in  $PMR_{rcv}$  and  $PMR_{snd}$ .

<sup>20</sup>If the execution is started at the choreography graph’s initial node

events of  $w$  (lines 12-15). If any race is detected, an entry in the *EvenstInRace* table is created, and  $w$  is added to the *ActInRace* set (line 16). The *LoopRaceType1* flag is also set to *true* if  $v$  is inside a loop. Once the race detection process is finished, the procedure checks whether it is allowed to chain the activity sequence resulting from concatenating  $seq_1$  and  $seq_2$  with other activity sequences (line 18)<sup>21</sup>. If allowed, the procedure invokes either *CheckChainsSC* (i.e. a recursive invocation) or *CheckChainsWC* to continue with the chaining process. The former happens if role  $p$  does not execute any sending event in  $w$ . In that case, the index  $(u, p, j)$  pointing to  $seq_2$  belongs to  $PMR_{rcv}$ .

If  $w$  was already visited, a loop has been detected. In that case all activities that were visited after the first visit to  $w$  are added to the *ActInsideLoop* set. Also, if any of those activities is in race with  $v$ , the appropriate *LoopRaceType2* flag is set to *true*.

**Procedure *CheckChainsWC* (on page 245).** Given an activity sequence  $seq_1$  (obtained by the *DetectRacesWithSendCausality* algorithm), which starts with an activity  $v$  and ends with an activity  $u$ , this procedure tries to find activity sequences obtained by the *DetectRacesWithSendCausality* algorithm that start with  $u$  (this is done using the indexes stored in  $PMR_{wc}$ ). Given an activity sequence  $seq_2$  that starts with  $u$ , and whose causal partial order is  $(R_{\prec} \cup S_{\prec}, \prec)$ , the procedure looks for races between any receiving event  $r_v$  from  $v$  (i.e.  $r_v \in R_{\text{race}}^v$ ) and any receiving event  $r$  from  $seq_2$  (i.e.  $r \in R_{\prec}$ ). For that, the procedure obtains the set of minimum sending events of  $p$  in  $seq_2$  (by construction of  $seq_2$  those sending events should belong to  $u$ 's set of events). Role  $p$  may have more than one minimum sending event if there are concurrent sendings. Given a maximum sending event  $m$ , and a receiving event  $r \in R_{\prec}$ , it is easy to see that if  $m \prec r$ , no  $r_v \in R_{\text{race}}^v$  can be in race with  $r$  (since  $r \prec m$  will always be true). Therefore, the procedure only looks for receiving events  $r \in R_{\prec}$  that have no maximum sending of  $p$  as causal predecessor.

---

<sup>21</sup>See explanation on page 224.

**Algorithm 15:** DetectRacesInChainedActSeqs**Data:**  $PMR, PMRSeqs$  from Algorithm 3; Choreography graph  $(V, E)$ **Result:** A table  $EventsInRace$  containing in position  $(e_1, e_2)$  a set of collaboration sequences that lead to a race between events  $e_1$  and  $e_2$ 


---

```

1 foreach role  $p$  do  $ActsInsideLoop[p] \leftarrow \emptyset$ 
2 forall  $(v, p, i) \in PMR_{rcv}$  such that  $v \neq \text{synch}PMR[v][p][i]$  do
3    $(seq_1, R_{\text{race}}^v, R_{\text{race}}^u, \text{race}) \leftarrow PMRSeqs_{sc}[v][p][i]$ 
4    $visitedSeq \leftarrow \emptyset$ 
5   forall  $u \in V$  do  $visited[u] \leftarrow \text{false}$  // Mark all choreography nodes as
   non-visited
6    $x \leftarrow$  last element of  $seq_1$ 
7    $visited[x] \leftarrow \text{true}; visitedSeq \leftarrow \{x\}$ 
8   if  $v = x$  then
9      $ActsInsideLoop[p] \leftarrow ActsInsideLoop[p] \cup \{v\}$ 
10   $ActsInRace \leftarrow \emptyset$ 
11  if  $\text{race}$  then  $ActsInRace \leftarrow \{x\}$ 
12   $CheckChainsSC(seq_1, v, p, R_{\text{race}}^v)$ 
13 forall  $(v, p, i) \in PMR_{snd}$  such that  $v \neq \text{synch}PMR[v][p][i]$  do
14    $(seq_1, R_{\text{race}}^v, R_{\text{race}}^u, \text{race}) \leftarrow PMRSeqs_{sc}[v][p][i]$ 
15    $u \leftarrow$  last element of  $seq_1$ 
16    $CheckChainsWC(seq_1, u)$ 

```

---

---

**Procedure** CheckChainsSC( $seq_1, v, p, R_{race}^v$ )
 

---

```

1   $visited_{old} \leftarrow visited$ ;  $visitedSeq_{old} \leftarrow visitedSeq$ ;
2   $u \leftarrow$  last element of  $seq_1$ 
3  forall  $(u, p, j) \in PMR_{snd} \cup PMR_{rcv}, j \geq 0$  do
4  |    $visited \leftarrow visited_{old}$ ;  $visitedSeq \leftarrow visitedSeq_{old}$ ;
5  |    $(seq_2, R_{race}^u, R_{race}^w, race) \leftarrow PMRSeqs_{sc}[u][p][j]$ 
6  |    $w \leftarrow$  last element of  $seq_2$ 
7  |   if  $\neg visited[w]$  then
8  |   |    $visited[w] \leftarrow true$ ;  $visitedSeq \leftarrow visitedSeq \cup \{w\}$ 
9  |   |   if  $w = v$  then
10 |   |   |    $ActsInsideLoop[p] \leftarrow ActsInsideLoop[p] \cup \{v\}$ 
11 |   |   |   forall  $x \in ActsInRace$  do  $LoopRaceType1[v][x] \leftarrow true$ 
12 |   |   |   foreach  $r_v \in R_{race}^v$  do
13 |   |   |   |   foreach  $r_w \in R_{race}^w$  do
14 |   |   |   |   |   if non-FIFO channels OR (FIFO channels AND
15 |   |   |   |   |   |    $loc(snd(r_v)) \neq loc(snd(r_w))$ ) then
16 |   |   |   |   |   |   |    $EventsInRace[r_v][r_w] \leftarrow EventsInRace[r_v][r_w] \cup \{(seq_1 - \{u\}) \cup seq_2\}$ 
17 |   |   |   |   |   |   |    $ActInRace \leftarrow ActInRace \cup \{w\}$ 
18 |   |   |   |   |   |   |   if  $v \in ActInsideLoop[p]$  then  $LoopRaceType1[v][w] \leftarrow true$ 
19 |   |   |   |   |   |   |   |
20 |   |   |   |   |   |   |   |   if  $(u, p, j) \in PMR_{rcv}$  then
21 |   |   |   |   |   |   |   |   |    $CheckChainsSC((seq_1 - \{u\}) \cup seq_2, v, p, R_{race}^v)$ 
22 |   |   |   |   |   |   |   |   |   else
23 |   |   |   |   |   |   |   |   |   |    $CheckChainsWC((seq_1 - \{u\}) \cup seq_2, p, R_{race}^v)$ 
24 |   |   |   |   |   |   |   |   |   |   |
25 |   |   |   |   |   |   |   |   |   |   |    $X \leftarrow \{w\} \cup \{x : x \text{ appears after } w \text{ in } visitedSeq\}$ 
26 |   |   |   |   |   |   |   |   |   |   |    $ActsInsideLoop[p] \leftarrow ActsInsideLoop[p] \cup X$ 
27 |   |   |   |   |   |   |   |   |   |   |   forall  $x \in ActsInRace \cap X$  do  $LoopRaceType2[v][x] \leftarrow true$ 
28 |   |   |   |   |   |   |   |   |   |   |   return

```

---

---

```

Procedure CheckChainsWC( $seq_1, p, R_{race}^v$ )
1  $u \leftarrow$  last element of  $seq_1$ 
2 forall  $(u, p, j) \in PMR_{wc}, j \geq 0$  do
3    $(seq_2, (R_{\prec} \cup S_{\prec}, \prec), \mathcal{R}) \leftarrow PMRSeqs_{wc}[u][p][j]$ 
4    $M \leftarrow \{m : m \in \min((S_{\prec}, \prec)) \wedge loc(m) = p\}$  // Minimum sending event(s) of  $p$  in  $E_{\prec}$ 
5   foreach  $r_v \in R_{race}^v$  do
6     foreach  $r \in R_{\prec}$  such that  $loc(r) = p \wedge m \neq r, \forall m \in M$  do
7       if non-FIFO channels OR (FIFO channels AND  $loc(snd(r_v)) \neq loc(snd(r))$ )
8         then
9           if  $r$  is marked as a “loop event” then
10            /* We assume  $r_{type}$  is the event type of which  $r$  is an instance */
11             $(type1, type2) \leftarrow RaceType[r_v][r_{type}]$ 
12             $type2 \leftarrow true$ 
13            if  $v \in ActsInsideLoop[p]$  then  $type1 \leftarrow true$  else  $type1 \leftarrow false$ 
14             $RaceType[r_v][r_{type}] \leftarrow (type1, type2)$ 
15          else
16             $EventsInRace[r_v][r] \leftarrow EventsInRace[r_v][r] \cup \{(seq_1 - \{u\}) \cup seq_2\}$ 
17
18 return

```

---

### 11.5.2 Detection of Ambiguous and Race Propagation

Ambiguous propagation happens when the triggering traces *specified* for a non-choosing component in two different alternatives of a choice have a common prefix. Race propagation happens if there can be an ambiguous propagation due to the existence of races. That is, if the triggering traces that may be *observed* at run-time in two different alternatives, due to the effect of races, have a common prefix.

Obviously, in order to determine whether two triggering traces have a common prefix it is sufficient to compare their first elements. This is indeed enough to either assert or negate the existence of ambiguous or race propagation. However, in case of a propagation problem, knowing the complete triggering traces and their complete common prefix helps to determine the most appropriate resolution. For example, consider these two pairs of triggering traces, both of them giving rise to ambiguous propagation:  $\{(?a), (?a)\}$  and  $\{(?a, ?b, ?c), (?a, ?b, ?d)\}$ . Whith the second pair we may opt for a design solution where the decision on which alternative to follow is postpone until the reception of either message  $c$  or  $d$ . We can do this by “extracting”  $?a, ?b$  from the choice in the local behavior of the non-choosing component. Obviously, this solution is not valid for the first pair of triggering traces.

We present in the following an algorithm for the detection of both ambiguous and race propagation problems. In the case of ambiguous propagation, the algorithm is able to obtain the maximum common prefix of two triggering traces leading to ambiguous propagation, even in the presence of loops. In the case of race propagation, the algorithm considers only one iteration of loops. This means that race propagation problems will always be detected, but the maximum common prefix of the triggering traces leading to a race propagation problem may not always be obtained.

For the algorithms we assume that a single state machine will be synthesized for each role in the choreography graph, and that such state machine will have one single input buffer to store the messages received from all the other roles. In the case that the same role participates in two concurrent collaborations (i.e. inside a fork-join pair), we consider that the messages received and sent by the role in both collaborations are interleaved (since we consider one single state machine with one single buffer for the role). However, a decision could be made to create the role state machine in such a way that messages belonging to each of the concurrent collaborations are treated by separate orthogonal sub-state machines. In that case, messages would no longer interleave. The proposed algorithms could be easily adapted to consider this case. We just need to mark the roles in concurrent collaborations as implemented by different state-machines. A decision could also be made to provide state machines with different input buffers for different peer roles. In that case, to construct correct triggering traces, the algorithm should be modified to consider only messages that can be received on the same buffer.

**Algorithm *DetectChoicePropagationProblems* (on page 254).** For each choice node  $v_{\text{ch}}$ , and each non-choosing role  $p$  on  $v_{\text{ch}}$ , this algorithm invokes the *ChoreographyToFSA* procedure to convert (a part of) the choreography graph into an “equivalent” (from the point of view of  $p$ ) finite state automaton, whose transitions are labeled by receiving and sending events executed by  $p$ . The resulting automaton may also have  $\varepsilon$ -transitions (i.e. silent transitions). Those  $\varepsilon$ -transitions are removed (see Appendix 11.B.2), and the resulting automaton is split into as many automata as branches has the choice under analysis. Each of the new automata may be used to generate the triggering traces of one of the choice’s branches.

These “branch” automata are then passed as input to the *DetectAmbiguousPropagation* and *DetectRacePropagation* procedures, which will analyse them for the detection of ambiguous and race propagation problems.

**Procedure *DetectAmbiguousPropagation* (on page 254).** This procedure receives as input a set  $\mathbb{A}$  of automata, where each automaton  $\mathcal{A}_i \in \mathbb{A}$  describes (part of) the behavior of a non-choosing role  $p$  in  $i^{\text{th}}$  branch of the choice under analysis. More specifically,  $\mathcal{A}_i$  describes the behavior of  $p$ , up to the its first sending event (if any), in each of the possible execution paths through the  $i^{\text{th}}$  branch. That is,  $\mathcal{A}_i$  describes  $p$ ’s *specified* triggering traces in the  $i^{\text{th}}$  branch of the choice under analysis.

Given two automata  $\mathcal{A}_1 = (Q_1, \Sigma_1, \delta_1, q_{01}, F_1)$  and  $\mathcal{A}_2 = (Q_2, \Sigma_2, \delta_2, q_{02}, F_2)$ , an ambiguous propagation will exist if one of the strings accepted by  $\mathcal{A}_1$  has a common prefix with one of the strings accepted by  $\mathcal{A}_2$ . This can be easily checked by constructing their intersection automaton  $(Q, \Sigma, \delta, q_0, F) = \mathcal{A}_1 \cap \mathcal{A}_2$ , where  $Q = Q_1 \times Q_2$ ,  $\Sigma = \Sigma_1 \cap \Sigma_2$ ,  $\delta = \{((q_1, q_2), e, (q'_1, q'_2)) : (q_1, e, q'_1) \in \delta_1, (q_2, e, q'_2) \in \delta_2\}$ ,  $q_0 = (q_{01}, q_{02})$  and  $F = F_1 \times F_2$ .

Note that before intersecting the automata, their transitions are relabeled, so that instead of events, their associated messages (i.e. the message sent or received) are used as transition labels.

If the set of transitions of the intersection automaton is non-empty (i.e.  $\delta \neq \emptyset$ ), there is a problem of ambiguous propagation. To obtain the sequence(s) of events leading to ambiguous propagation, the intersection automaton is converted into an equivalent regular expression. Note, however, that the intersection automaton may have unreachable states. These states should be eliminated before converting the automaton into a regular expression. In addition,  $\varepsilon$ -transitions should be added from each state with no output transitions to a common final state.

**Procedure *DetectRacePropagation* (on page 255).** This procedure receives as input a set  $\mathbb{A}$  of automata, where each automaton  $\mathcal{A}_i \in \mathbb{A}$  describes (part of) the behavior of a non-choosing role  $p$  in  $i^{\text{th}}$  branch of the choice under analysis. More specifically,  $\mathcal{A}_i$  describes the behavior of  $p$ , up to the its first sending event, in each of the possible execution paths through the  $i^{\text{th}}$  branch. That is,  $\mathcal{A}_i$  describes  $p$ 's *specified* triggering traces in the  $i^{\text{th}}$  branch of the choice under analysis. The procedure also gets as input the *containerNode* table, which for each sending event at the end of a triggering trace stores the activity node where the sending event can be found, or the fork node of a fork-join pair containing the activity where the sending event can be found.

To detect a race propagation this procedure obtains the *observed* triggering traces (lines 1-18), that is, the triggering traces observed by  $p$  in the presence of races. Once they have been obtained, the procedure checks whether any two of those traces (corresponding to different branches of the choice under analysis) have a common prefix (line 19).

To obtain the observed triggering traces the procedure proceeds as follows. Each automaton is converted into an equivalent regular expression (line 3), following the technique described in Appendix 270. The resulting regular expression may describe loops. The *SeparateInAltSubexpressions* procedure (line 3) initializes those loops, so at most one iteration is considered. For that purpose, each sub-expression  $\alpha_1 \cdot \alpha_2^* \cdot \alpha_3$  is replaced with  $(\alpha_1 \cdot \alpha_3 | \alpha_1 \cdot \alpha_2 \cdot \alpha_3)$ , and each sub-expression  $\alpha^+$  is replaced with  $\alpha$ . The new regular expression will consist of the union of several sub-expressions, each of them describing  $p$ 's triggering trace in a given execution path. Each of those sub-expression will be processed separately. For each sub-expression  $re$ , the terminating sending event  $s$ , if any, is extracted. All events in  $re$  (except  $s$ ) are then partially ordered (line 6): events in race become unordered; otherwise the total order dictated by  $re$  is respected. If  $re$  ended with a receiving event (i.e.  $s = \text{null}$ ), it means that a final node was reached in the execution path described by  $re$ . The resulting poset can thus already be used to generate the observed triggering traces, which correspond to the labeled-linearizations of the poset<sup>22</sup> (line 18). Otherwise, if  $re$  ended with a sending event  $s$ , races might be possible between receiving events from the triggering trace and receiving events that succeed  $s$  according to the visual order. The set  $E_{\text{ext}}$  of “external” events (i.e. not from the triggering trace) that are in race with events

<sup>22</sup>A *linearization* of a poset  $(E, <)$  is a word  $w = e_1 \cdots e_{|E|}$  over the alphabet  $E$ , such that if  $e_i < e_j$  then  $i < j$ . In [ON05] a technique is described to obtain all possible linearizations of a poset in an efficient way. A *labeled-linearization* is a linearization where each event has been replaced with its associated message (i.e. the message sent or received).

of the triggering trace can be obtained with help of the *EventsInRace* table (lines 11-13). However, we still need to determine the causal order between those events, and between them and the events of the triggering trace. This is done with help of the *GetPosetsForObservedTT* procedure, which returns a set of partial orders whose labeled-linearizations correspond to the observed triggering traces.

**Procedure *ChoreographyToFSA* (on page 256).** This procedure returns an automaton describing the behavior of role  $p$  in (part of) the choreography graph. It also returns a table *containerNode*, which stores the choreography nodes where some events can be found.

Starting from a decision node, this procedure traverses the choreography graph using a depth-first search technique [AHU74]. For each possible execution path starting at the decision node, the procedure stops searching when an activity is found where role  $p$  executes a sending event, or when a final node is reached. The nodes that are visited and the edges that are traversed are mapped into states and transitions of an automaton  $\mathcal{A}$ .

When an activity  $v$  where  $p$  participates is visited, function *fsa* is invoked to obtain an automaton describing the behavior of  $p$  in  $v$  (line 4). This new automaton is then concatenated with  $\mathcal{A}$  at certain *junction* states  $J$  (i.e. final states with incoming transitions that are labeled with a receiving event – see *Concatenation* on page 250), and the set of junction states is updated. If the resulting automaton has any final state with an input transition labeled with a sending event  $s$ , an entry in the *containerNode* table is created. Thereafter, if all the final states of the automaton have input transitions labeled with sending events (i.e. the new set of junction states is empty – see line *refjunctionempty*), the backtracking process is initiated. Otherwise, the graph traversal continues.

When a merge node is found (line 9), a new state is added to  $\mathcal{A}$  and each of the junction states is connected to the newly added state by means of  $\epsilon$ -transitions. Decision nodes are not explicitly mapped into states of  $\mathcal{A}$ . Instead, the junction states at the time a decision node is visited will act as “decision” states (i.e. the automata obtained by traversing each of the branches of the decision node will be concatenated at those junction states).

When a fork node is visited (line 11) the *ForkToFSA* procedure is invoked. This procedure returns an automaton corresponding to the nodes found within the fork and its companion join node. This automaton is then concatenated with  $\mathcal{A}$ . If the set of new junction states is empty (i.e. a sending event was found in all paths within the fork-join pair) or if a join node was not found (because all the fork’s branches ended up in a final node), the backtracking process is initiated. Otherwise, the graph traversal continues from the fork’s companion join node.

Each node in the choreography graph is visited at most once. It may happen that an attempt is made to re-visit a merge node, or to re-visit the decision node of the choice under analysis (i.e. the very first node been visited). The latter may happen if that node is inside a loop, as in the choreography of Fig. 11.17(a). In the first case  $\epsilon$ -transitions are added from the current junction states to the state corresponding to the merge node (i.e. the state added the first time the merge node

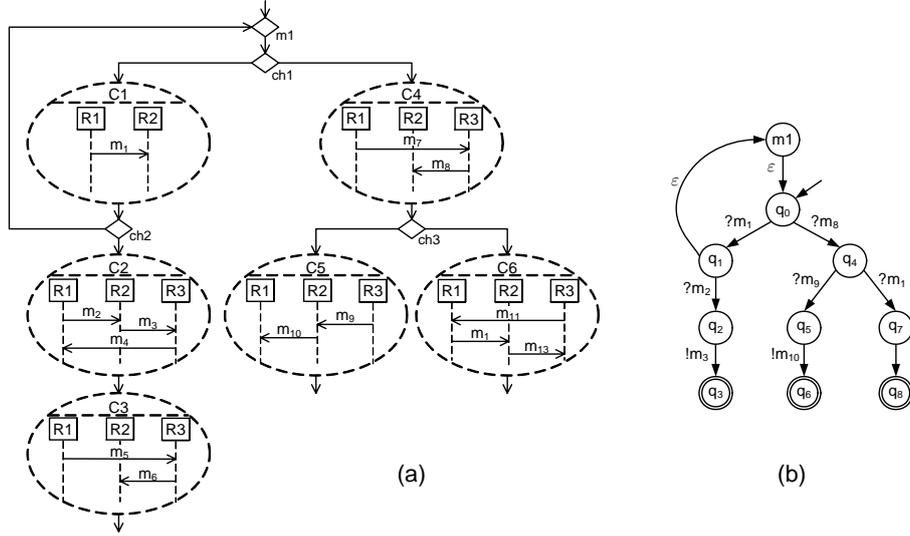


Figure 11.17: (a) Choice (ch1) with race propagation; (b) Automaton describing the significant part of  $R2$ 's behavior in (a) for detection of race propagation

was visited)(line 28). In the second case,  $\epsilon$ -transitions are added from the current junction states to the initial state (line 30).

Figure 11.17 shows a choreography graph and the automaton created by procedure *ChoreographyToFSA* for role  $R2$ .

**Function *f<sub>sa</sub>*.** This function takes an expression describing a sequence diagram<sup>23</sup> and a role  $p$ , and returns an automaton  $\mathcal{A}$  describing the behavior of  $p$  in the given sequence diagram. It is defined as

$$f_{sa}(SD, p) = \begin{cases} \mathcal{A}_{bSD}, & \text{if } SD := bSD \\ f_{sa}(SD_1, p) \cdot f_{sa}(SD_2, p), & \text{if } SD := SD_1 \text{ seq } SD_2 \\ f_{sa}(SD_1, p) \cup f_{sa}(SD_2, p), & \text{if } SD := SD_1 \text{ alt } SD_2 \\ f_{sa}(SD_1, p) \times f_{sa}(SD_2, p), & \text{if } SD := SD_1 \text{ par } SD_2 \\ f_{sa}(SD_1, p)^*, & \text{if } SD := \text{loop}(0, n) SD_1, n > 0 \\ f_{sa}(SD_1, p) \cdot f_{sa}(SD_1, p)^*, & \text{if } SD := \text{loop}(n, m) SD_1, 0 < n \leq m \end{cases}$$

where  $\mathcal{A}_{bSD}$  is an automaton describing the behavior of  $p$ , up to its first sending event (if any), in the  $bSD$  basic sequence diagram. To obtain  $\mathcal{A}_{bSD}$ , the basic sequence diagram is first projected onto the lifeline of  $p$ . The result is a totally ordered set of events  $(E, <)$ . From that set we are only interested in the first sending event (if

<sup>23</sup>Recall that a sequence diagram can be described by an expression consisting of basic sequence diagrams (bSDs) identifiers and a combination of *seq*, *alt*, *par* and *loop* operators (i.e. an expression conforming to the BNF-grammar described in Sect. 193)

any) and all its preceding events. That is, assuming that  $E = R \cup S$  (with  $S$  the set of sending events), we are interested on the totally ordered set  $(E', <)$ , where

$$E' = \begin{cases} E, & \text{if } S = \emptyset \\ \{e \in E : (e < s \vee e = s), s \in S \wedge \nexists s' \in S, s' < s\}, & \text{otherwise} \end{cases}$$

The ordered set  $(E', <)$  is then converted into an automaton  $\mathcal{A}_{\text{bSD}}$  such that:

- If  $E' = \emptyset$  (i.e.  $p$  does not participate in the basic sequence diagram),  $\mathcal{A}_{\text{bSD}} = \mathcal{A}_\emptyset$ , where  $\mathcal{A}_\emptyset = (\{q_0\}, \emptyset, \emptyset, q_0, \{q_0\})$  is the so-called *empty* automaton.
- Otherwise,  $\mathcal{A}_{\text{bSD}} = (Q, E, \delta, q_0, F)$ , with
 
$$Q = \{q_0\} \cup \{q_e : e \in E\}$$

$$\delta = \{(q_0, e, q_e) : e \in E \wedge \nexists f \in E, f < e\} \cup \{(q_e, f, q_f) : e, f \in E, e < f \wedge \exists g \in E, e < g < f\}$$

$$F = \{q_e : e \in E \wedge \nexists f \in E, e < f\}$$

The operators used by the *fsa* function to compose the automata are defined as follows. Let  $\mathcal{A}_1 = (Q_1, \Sigma_1, \delta_1, q_{01}, F_1)$  and  $\mathcal{A}_2 = (Q_2, \Sigma_2, \delta_2, q_{02}, F_2)$  be two automata with disjoint sets of states, and let  $J_i = \{q_f \in F_i : \exists (q, r, q_f) \in \delta_i \text{ and } r \text{ is a receiving event}\}$ ,  $i \in \{1, 2\}$ , be the set of *junction* states of  $\mathcal{A}_i$  (i.e. the set of final states that have an incoming transition labeled with a receiving event). We then define the following four basic operations on automata:

**Concatenation.** For the concatenation of two automata  $\mathcal{A}_1$  and  $\mathcal{A}_2$ , for each output transition with label  $e$  from the initial state of  $\mathcal{A}_2$ , new transitions are added, with the same label  $e$ , from the junction states of  $\mathcal{A}_1$  to the successors of the initial state of  $\mathcal{A}_2$ . If the initial state of  $\mathcal{A}_2$  is not a final state, it is removed, together with its output transitions. More formally, the concatenation  $\mathcal{A}_1 \cdot \mathcal{A}_2$  of two automata  $\mathcal{A}_1$  and  $\mathcal{A}_2$  is an automaton  $\mathcal{A}$  such that:

- $\mathcal{A} = \mathcal{A}_1$ , if  $J_1 = \emptyset \vee \mathcal{A}_2 = \mathcal{A}_\emptyset$
- $\mathcal{A} = \mathcal{A}_2$ , if  $\mathcal{A}_1 = \mathcal{A}_\emptyset$
- Otherwise,  $\mathcal{A} = (Q, \Sigma_1 \cup \Sigma_2, \delta, q_{01}, F_2)$ , with
 
$$Q = \begin{cases} Q_1 \cup Q_2, & \text{if } q_{02} \in F_2 \\ Q_1 \cup Q_2 - \{q_{02}\}, & \text{otherwise} \end{cases}$$

$$\delta = \delta_1 \cup (\delta_2 \cap (Q \times \Sigma_2 \times Q)) \cup \{(q_1, e, q_2) \in J_1 \times \Sigma_2 \times Q_2 : (q_{02}, e, q_2) \in \delta_2\}$$

**Union.** For the union of two automata, a new initial state is created and concatenated with each of the automata. More formally, the union  $\mathcal{A}_1 \cup \mathcal{A}_2$  of two automata  $\mathcal{A}_1$  and  $\mathcal{A}_2$  is automaton  $\mathcal{A}$ , such that:

- $\mathcal{A} = \mathcal{A}_1$ , if  $\mathcal{A}_2 = \mathcal{A}_\emptyset$
- $\mathcal{A} = \mathcal{A}_2$ , if  $\mathcal{A}_1 = \mathcal{A}_\emptyset$

- Otherwise,  $\mathcal{A} = (Q, \Sigma_1 \cup \Sigma_2, \delta, q_0, F_1 \cup F_2)$ , where

$q_0$  is a new state

$$Q = \begin{cases} Q_1 \cup Q_2, & \text{if } q_{01} \in F_1 \text{ and } q_{02} \in F_2 \\ Q_1 \cup Q_2 - \{q_{01}, q_{02}\}, & \text{if } q_{01} \notin F_1 \text{ and } q_{02} \notin F_2 \\ Q_1 \cup Q_2 - \{q_{0i}\} & \text{if } q_{0i} \in F_i, i \in \{1, 2\} \end{cases}$$

$$\delta = (\delta_1 \cap (Q \times \Sigma_1 \times Q)) \cup (\delta_2 \cap (Q \times \Sigma_2 \times Q)) \cup \{(q_0, e, q_1) \in \{q_0\} \times \Sigma_1 \times Q_1 : (q_{01}, e, q_1) \in \delta_1\} \cup \{(q_0, e, q_2) \in \{q_0\} \times \Sigma_2 \times Q_2 : (q_{02}, e, q_2) \in \delta_2\}$$

**Kleene closure.** For the Kleene closure of an automaton, the initial state is also made a final state. The original final states are removed, and their input transitions are connected to the initial state. More formally, the Kleene closure  $\mathcal{A}_1^*$  of an automaton  $\mathcal{A}_1$  is an automaton  $\mathcal{A}$  such that:

- $\mathcal{A} = \mathcal{A}_1$ , if  $F_1 = J_1 \vee \mathcal{A}_1 = \mathcal{A}_0$
- Otherwise,  $\mathcal{A} = (Q, \Sigma_1, \delta, q_{01}, F)$ , with
 
$$Q = Q_1 - J_1 \cup \{q_{01}\}$$

$$\delta = \delta_1 \cup \{(q, e, q_{01}) : (q, e, q_j) \in \delta_1, q_j \in J_1\}$$

$$F = \{q_{01}\} \cup (F_1 - J_1)$$

**Cross-product.** The cross-product  $\mathcal{A}_1 \times \mathcal{A}_2$  of two automata  $\mathcal{A}_1$  and  $\mathcal{A}_2$  is the automaton  $\mathcal{A}$  returned by the *CrossProductFSA* procedure. That is,  $\mathcal{A} = \text{CrossProductFSA}(\mathcal{A}_1, \mathcal{A}_2)$ . The result is an automaton describing the interleaving of the transitions of the original automata. The interleaving is however stopped as soon as a transition labeled with a sending event is found.

**Procedure *ForkToFSA* (on page 257).** This procedure returns an automaton describing the behavior of a role  $p$  on the part of a choreography graph included between a fork node and its companion join node. It also returns a *containerNode* table, which for each sending event labeling an input-transition of a final automaton state (i.e. a sending event at the end of a triggering trace) stores the activity node where the sending event can be found, or the fork node of a fork-join pair containing the activity where the sending event can be found.

The procedure converts each branch of the fork into an automaton with help of the *ChoreographyToFSA* procedure. The resulting automaton may have several final states if several execution paths are possible through the fork's branch. In that case the *SplitFSA* procedure is invoked to split the automaton into as many automata as final states had the former automaton. Each of the new automata will describe the behavior of  $p$  on one of the execution paths through the branch. If any of the automata has a final state with input-transitions labeled with a sending event, an entry in the *containerNode* table is created storing the fork node under analysis.

After all branches have been processed, their associated automata are grouped in sets of concurrent automata (i.e. automata corresponding to concurrent execution paths) with help of the *getsComb* function (see page 236 for details on this function).

A cross-product automaton is then obtained for each group of concurrent automata. The union of all the cross-product automata, and the *containerNode* table, are returned by the *ForkToFSA* procedure.

**Procedure *GetPosetsForObservedTT* (on page 259).** For a given role  $p$ , this procedure takes a poset  $(E_{tt}, \prec_{tt})$ , describing the causal order between the events of a certain triggering trace, a set  $E_{ext}$  of all “external” events (i.e. not from the triggering trace) that are in race with events from the triggering trace. Those external events may not all be executed in the same execution path. For each possible execution path, this procedure finds out which external events are executed in the given path and creates a causal poset (in the following *OTT poset*) relating those events and the events from the triggering trace. The set of all OTT posets is then returned.

In order to properly group the external events and create the observed triggering trace’s posets, a choreography node  $v_0$  is also provided as input for the procedure. This node might be either an activity node or a fork node. Let us first look at the former case.

If  $v_0$  is an activity node,  $v_0$  is the activity where the sending event triggered by the triggering trace can be found. Since all external events will be successor (or concurrent) events of that sending event, the procedure uses  $v_0$  as a starting point to traverse the choreography graph in search of the external events. Since the behavior of  $v_0$  may be described by more than one poset, the procedure first needs to determine which of those posets should be considered as the actual starting point to begin searching for external events. The set  $PS_{match}$  of selected posets will contain those posets with the highest number of events in common with the triggering trace (line 3). Consider, for example, that the behavior of  $p$  in  $v_0$  is the one illustrated in Fig. 11.18, and that  $E_{tt} = \{\dots, ?a, ?b, ?c\}$  (i.e. the triggering trace is, for example,  $\dots ?a ?c ?b$  and triggers  $!s_1$ ). Then, of the two posets describing the behavior of  $v_0$  the procedure will only select the poset that includes events  $?c$  and  $!s_2$ .

Once the set  $PS_{match}$  has been obtained, the procedure determines the subset of external events that may be found in activities succeeding  $v_0$  (line 5). If that subset is not empty, the *VisitSuccessorRP* procedure is invoked (line 10). That procedure traverses the choreography graph from  $v_0$ ’s successor using a DFS technique. If an activity where external events can be found is visited by *VisitSuccessorRP*, those external events are added into a poset. *VisitSuccessorRP* returns a set  $PS_{tail}$  of posets, each of them describing the causal order of the external events found on a certain execution path.

After *VisitSuccessorRP* returns (if it was invoked), the procedure analyses the poset in poset  $PS_{match}$ . For each poset  $ps_0 \in PS_{match}$ , the procedure checks whether any external event can be found in  $ps_0$  (line 13). At this point it is important to note that not all external events that could be found in  $ps_0$  should be included in the OTT poset under construction. Consider again the example in Fig. 11.18, and imagine that  $?a$  is in race with  $?e$ , but that this race can only happen when  $?d$  is executed. Such race could therefore not happen for the given triggering trace. Although event  $?a$  would be part of  $E_{ext}$ , it should not be included in the OTT poset. In general, for a poset  $ps_0$ , only external events that are not causal successor of any minimum

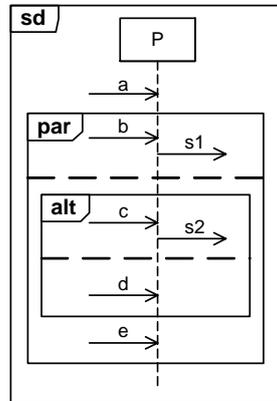


Figure 11.18: Behavior of role  $p$  in  $v_0$

sending of  $ps_0$  should be included in the OTT poset. To determine the causal order between such events and the triggering trace events, two cases are considered. For those triggering trace events that can also be found in  $ps_0$ , the causal order relations dictated by that poset are considered (line 14). For each triggering trace event  $r_1$  preceding the events of  $ps_0$ , a causal order relation  $r_1 \prec_{\text{ott}} r_2$  is created for each selected external event  $r_2$ , if  $r_1$  and  $r_2$  are not in race (line 15).

Once  $ps_0$  has been processed, the procedure checks whether role  $p$  gets synchronized on a sending event belonging to  $ps_0$  (line 16). If  $p$  does not get synchronized (line 5 of *UpdatePoset* procedure), each of the posets in  $PS_{\text{tail}}$  are concatenated with the OTT poset. If  $p$  gets synchronized, the triggering trace events from activities preceding  $v_0$  will not be in race with events from activities succeeding  $v_0$ . However, triggering trace events from  $v_0$  might be in race with events from activities succeeding  $v_0$ . If there is any triggering trace event from  $v_0$  in race with some external events  $E$  (line 9), only the parts of the posets in  $PS_{\text{tail}}$  that deal with any of the  $E$  events are concatenated with the OTT poset. If  $E = \emptyset$ , the  $PS_{\text{tail}}$  posets are not considered to create the OTT posets.

As we already said,  $v_0$  might also be a fork node. In that case,  $v_0$  is the fork of a fork-join pair that contains the activity where the sending event triggered by the triggering trace can be found, but that does not contain all the events of the triggering trace. In this case, procedure *MapForkWC* (see page 236) is invoked to obtain a set of causal posets for all the activities contained by the fork-join pair. After that, the processing is similar as the case where  $v_0$  is an activity.

**Algorithm 18:** DetectChoicePropagationProblems

---

```

1 foreach choice node  $v_{ch} \in V$  do
2   foreach non-choosing role  $p$  in  $v_{ch}$  do
3     forall  $v \in V$  do  $visited[v] = false$ 
4      $q_0 \leftarrow NewFSAState()$ 
5      $\mathcal{A} \leftarrow (\{q_0\}, \emptyset, \emptyset, q_0, \{q_0\})$ 
6      $(\mathcal{A}, containerNode) \leftarrow ChoreographyToFSA(p, v_{ch}, \mathcal{A}, \{q_0\})$ 
7      $\mathcal{A} \leftarrow RemoveEpsilonTransitions(\mathcal{A})$ 
      /* We split  $\mathcal{A}$  in a set of automata, one for each branch of the
      choice. */
8     forall  $q \in Q$  such that  $\exists(q_0, e, q) \in \delta, e \in \Sigma$  do
9        $Q_{cl} \leftarrow Closure(q)$  /* All states reachable from  $q$ , incl.  $q$  */
10       $F' \leftarrow Q_{cl} \cap F; Q' \leftarrow Q_{cl} \cup \{q_0\}; \delta' \leftarrow \{(q_1, e, q_2) \in \delta : q_1, q_2 \in Q'\}$ 
11       $\mathbb{A} \leftarrow \mathbb{A} \cup \{(Q', \Sigma, \delta', q_0, F')\}$ 
12       $DetectAmbiguousPropagation(\mathbb{A})$ 
13       $DetectRacePropagation(p, containerNode, \mathbb{A})$ 
14 end

```

---

**Procedure** DetectAmbiguousPropagation( $\mathbb{A}$ )

---

```

1  $RE \leftarrow \emptyset$ 
2 foreach  $\mathcal{A} \in \mathbb{A}$  do  $\mathcal{A} \leftarrow RelabelTransitions(\mathcal{A})$ 
3 foreach  $\mathcal{A}_1 \in \mathbb{A}$  do
4    $\mathbb{A} \leftarrow \mathbb{A} - \{\mathcal{A}_1\}$ 
5   foreach  $\mathcal{A}_2 \in \mathbb{A}$  do
6      $\mathcal{A}_{prefix} \leftarrow \mathcal{A}_1 \cap \mathcal{A}_2$  // Intersection automaton
7     if  $\mathcal{A}_{prefix}.\delta \neq \emptyset$  then
8       /* Eliminate non-reachable states in  $\mathcal{A}_{prefix}$  and add  $\epsilon$ -transition
      from states w/o output transitions to a common final state */
       $RE \leftarrow RE \cup \{ConvertFSAToRE(\mathcal{A}_{prefix})\}$ 

```

---

---

**Procedure DetectRacePropagation**( $p, \text{containerNode}, \mathbb{A}$ )
 

---

```

1  foreach  $\mathcal{A} \in \mathbb{A}$  do
2     $\mathcal{L}[\mathcal{A}] \leftarrow \emptyset$ 
3     $RE \leftarrow \text{SeparateInAltSubexpressions}(\text{ConvertFSAToRE}(\mathcal{A}))$ 
4    foreach  $re \in RE$  do
5       $s \leftarrow$  sending event at the end of  $re$  or null if  $re$  ends with a receiving event
6       $(E_{tt}, \prec_{tt}) \leftarrow$  events in  $re$  (except  $s$ ) are partially ordered
7      if  $s = \text{null}$  then
8         $\Upsilon \leftarrow \{(E_{tt}, \prec_{tt})\}$ 
9      else
10         /* Get set  $E_{\text{ext}}$  of "external" events in race with events from the
11         triggering trace */
12          $E_{\text{ext}} \leftarrow \emptyset$ 
13         foreach  $e_1 \in E_{tt}$  do
14           foreach  $e_2$  such that  $\text{EventsInRace}[e_1, e_2] \neq \emptyset \wedge e_2 \notin E_{tt}$  do
15              $E_{\text{ext}} \leftarrow E_{\text{ext}} \cup \{e_2\}$ 
16         if  $E_{\text{ext}} = \emptyset$  then
17            $\Upsilon \leftarrow \{(E_{tt}, \prec_{tt})\}$ 
18         else
19           /* Get set  $\Upsilon$  of causal posets describing causal relations
20           between events in  $E_{tt} \cup E_{\text{ext}}$  */
21            $\Upsilon \leftarrow \text{GetPosetsForObservedTT}(p, \text{containerNode}[s], E_{\text{ext}}, E_{tt}, \prec_{tt})$ 
22          $\mathcal{L}[\mathcal{A}] \leftarrow \mathcal{L}[\mathcal{A}] \cup \text{GetLinearizations}(\Upsilon)$ 
23 if  $\exists l \in \mathcal{L}[\mathcal{A}_1], \mathcal{A}_1 \in \mathbb{A} \wedge \exists l' \in \mathcal{L}[\mathcal{A}_2], \mathcal{A}_2 \in \mathbb{A}$  such that  $l$  and  $l'$  have a common prefix then
24    $\left[ \text{Report race propagation} \right.$ 

```

---

---

**Procedure** ChoreographyToFSA( $p, v, \mathcal{A}, J$ )
 

---

```

1  containerNode  $\leftarrow \emptyset$ 
2  visited[v]  $\leftarrow true$ 
3  if  $v$  is an activity node where  $p$  participates then
4      /* Let  $SD_v$  be the sequence diagram describing  $v$ 's behavior */
5       $(\mathcal{A}, J) \leftarrow ConcatenateFSA(\mathcal{A}, fsa(SD_v, p), J)$ 
6      foreach sending event  $s$  such that  $(q, s, q_f) \in \delta, q_f \in F - J$  do
7          containerNode[s]  $\leftarrow v$ 
8      if  $J = \emptyset$  then
9          visited[v]  $\leftarrow false$ ; return  $(\mathcal{A}, containerNode)$ 
10     else if  $v$  is a merge node then
11          $q_v \leftarrow NewFSAState(v); Q \leftarrow Q \cup \{q_v\}; \delta \leftarrow \delta \cup \{(q_j, \epsilon, q_v) : q_j \in J\}; F \leftarrow F \cup \{q_v\}; J \leftarrow \{q_v\}$ 
12     else if  $v$  is a join node then
13          $v_{join} \leftarrow v$ ; visited[v]  $\leftarrow false$ ; return  $(\mathcal{A}, containerNode)$ 
14     else if  $v$  is a fork node then
15          $v_{join} \leftarrow null$ 
16          $(\mathcal{A}_{fork}, containerNode_{aux}) \leftarrow ForkToFSA(p, v)$ 
17         containerNode  $\leftarrow containerNode \cup containerNode_{aux}$ 
18          $(\mathcal{A}, J) \leftarrow ConcatenateFSA(\mathcal{A}, \mathcal{A}_{fork}, J)$ 
19         if  $J = \emptyset \vee v_{join} = null$  then
20             visited[v]  $\leftarrow false$ 
21             return  $(\mathcal{A}, containerNode)$ 
22         /* We continue traversing the graph from the join node associated to the
23            fork (i.e.  $v_{join}$ ). Note that we assume proper nesting of fork/join
24            nodes */
25          $v \leftarrow v_{join}$ 
26     foreach  $u$  successor of  $v$  do
27         if !visited[u] then
28              $(\mathcal{A}, containerNode_{aux}) \leftarrow ChoreographyToFSA(p, u, \mathcal{A}, J)$ 
29             containerNode  $\leftarrow containerNode \cup containerNode_{aux}$ 
30         else
31             if  $u$  is a merge node then
32                  $\delta \leftarrow \delta \cup \{(q_j, \epsilon, q_u) : q_j \in J\}$ 
33             else
34                  $\delta \leftarrow \delta \cup \{(q_j, e, q_0) : q_j \in J\}$ 
35     if  $v$  is NOT a merge node then
36         visited[v]  $\leftarrow false$ 
37     return  $(\mathcal{A}, containerNode)$ 

```

---

**Procedure ForkToFSA( $p, v_{\text{fork}}$ )**


---

```

1  $\mathbb{A}_{\text{all}} \leftarrow \emptyset$ 
2 foreach successor  $v$  of  $v_{\text{fork}}$  do
   | /* An FSA is built for each branch of the fork */
3    $q_0 \leftarrow \text{NewFSAState}(v)$ 
4    $\mathcal{A}_v \leftarrow (\{q_0\}, \emptyset, \emptyset, q_0, \{q_0\})$ 
5    $(\mathcal{A}_v, \text{containerNode}) \leftarrow \text{ChoreographyToFSA}(p, v, \mathcal{A}_v, \{q_0\})$ 
6    $\mathcal{A}_v \leftarrow \text{RemoveEpsilonTransitions}(\mathcal{A}_v)$ 
   | /* If  $\mathcal{A}_v$  has several final states,  $\text{SplitFSA}()$  returns one FSA for each
   | final state */
7   if  $|\mathcal{A}_v.F| > 1$  then
8   |  $\mathbb{A}_{\text{all}} \leftarrow \mathbb{A}_{\text{all}} \cup \{\text{SplitFSA}(\mathcal{A}_v)\}$ 
9   else
10  |  $\mathbb{A}_{\text{all}} \leftarrow \mathbb{A}_{\text{all}} \cup \{\{\mathcal{A}_v\}\}$ 
11  foreach  $s$  such that  $(q, s, q_f) \in \mathcal{A}_v.\delta, q_f \in \mathcal{A}_v.F$  and  $s \in \mathcal{A}_v.\Sigma_v$  is a sending event do
12  |  $\text{containerNode}[s] \leftarrow v_{\text{fork}}$ 
13  $\mathcal{A}_{\text{fork}} \leftarrow (\{q_0\}, \emptyset, \emptyset, q_0, \{q_0\})$ 
14 foreach  $\mathbb{A} \in \text{getCombs}(\mathbb{A}_{\text{all}})$  do
15 |  $\mathcal{A}_{\text{fork}} \leftarrow \mathcal{A}_{\text{fork}} \cup \text{CrossProductFSA}(\mathbb{A})$ 
16 return  $(\mathcal{A}_{\text{fork}}, \text{containerNode})$ 

```

---

**Procedure CrossProductFSA( $\mathbb{A}$ )**


---

```

Data: Set  $\mathbb{A}$  of automata
Result: FSA  $(Q, \Sigma, \delta, q_0, F)$  corresponding to the cross-product of the FSAs in  $\mathbb{A}$ 
1 if  $\mathbb{A} = \{\mathcal{A}\}$  then return  $\mathcal{A}$ 
2  $\mathcal{A}_1 \leftarrow$  any element of  $\mathbb{A}$ ;  $\mathbb{A} \leftarrow \mathbb{A} - \{\mathcal{A}_1\}$ 
3 while  $\mathbb{A} \neq \emptyset$  do
4 |  $\mathcal{A}_2 \leftarrow$  any element of  $\mathbb{A}$ ;  $\mathbb{A} \leftarrow \mathbb{A} - \{\mathcal{A}_2\}$ 
5 |  $Q_{\text{aux}} \leftarrow \{(q_{01}, q_{02})\}$ 
6 | while  $Q_{\text{aux}} \neq \emptyset$  do
7 | |  $(q_1^s, q_2^s) \leftarrow$  Any element of  $Q_{\text{aux}}$ ;  $Q_{\text{aux}} \leftarrow Q_{\text{aux}} - \{(q_1^s, q_2^s)\}$ 
8 | | foreach transition  $(q_1^s, e_1, q_1^d) \in \delta_1$  do
9 | | |  $q_{\text{new}} \leftarrow (q_1^d, q_2^s)$ ;  $\delta \leftarrow \delta \cup \{(q_1^s, q_2^s), e_1, q_{\text{new}}\}$ ;  $Q \leftarrow Q \cup \{q_{\text{new}}\}$ 
10 | | | if  $e_1$  is a sending event  $\vee q_{\text{new}} \in F_1 \times F_2$  then  $F \leftarrow F \cup \{q_{\text{new}}\}$ 
11 | | | else  $Q_{\text{aux}} \leftarrow Q_{\text{aux}} \cup \{q_{\text{new}}\}$ 
12 | | foreach transition  $(q_2^s, e_2, q_2^d) \in \delta_2$  do
13 | | |  $q_{\text{new}} \leftarrow (q_1^s, q_2^d)$ ;  $\delta \leftarrow \delta \cup \{(q_1^s, q_2^s), e_2, q_{\text{new}}\}$ ;  $Q \leftarrow Q \cup \{q_{\text{new}}\}$ 
14 | | | if  $e_2$  is a sending event  $\vee q_{\text{new}} \in F_1 \times F_2$  then  $F \leftarrow F \cup \{q_{\text{new}}\}$ 
15 | | | else  $Q_{\text{aux}} \leftarrow Q_{\text{aux}} \cup \{q_{\text{new}}\}$ 
16 |  $\mathcal{A}_1 \leftarrow (Q, \Sigma, \delta, (q_{01}, q_{02}), F)$ 
17 return  $\mathcal{A}_1$ 

```

---

---

**Procedure SplitFSA( $\mathcal{A}$ )**


---

```

/* Assume  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$  */
1  $\mathbb{A} \leftarrow \emptyset$ 
2 foreach  $q_f \in F$  do
   /* Get set  $Q_{\text{rcl}}$  of all states from which  $q_f$  can be reached, incl.  $q_f$  */
3    $Q_{\text{rcl}} \leftarrow \text{ReverseClosure}(q_f)$ 
4    $\delta' \leftarrow \{(q_1, e, q_2) \in \delta : q_1, q_2 \in Q_{\text{rcl}}\}$ 
5    $\Sigma' \leftarrow \{e : (q_1, e, q_2) \in \delta'\}$ 
6    $\mathbb{A} \leftarrow \mathbb{A} \cup \{(Q_{\text{rcl}}, \Sigma', \delta', q_0, \{q_f\})\}$ 
7 return  $\mathbb{A}$ 

```

---

---

**Procedure** GetPosetsForObservedTT( $p, v_0, E_{\text{ext}}, E_{\text{tt}}, \prec_{\text{tt}}$ )

---

```

1  $\Upsilon \leftarrow \emptyset$ 
2 if  $v_0$  is an activity node then
   /* Get the posets of  $v_0$  that have the most common events with the
   triggering trace. Only one iteration of loops is considered. */
3  $PS_{\text{match}} \leftarrow \{(E_1, \prec_1) \in \llbracket SD_{v_0} \rrbracket_{\text{SD}} : \exists (E_2, \prec_2) \in (\llbracket SD_{v_0} \rrbracket_{\text{SD}} - \{(E_1, \prec_1)\}), |E_{\text{tt}} \cap E_2^p| >$ 
    $|E_{\text{tt}} \cap E_1^p|\}$ 
4  $PS_{\text{tail}} \leftarrow \emptyset$ 
5  $E'_{\text{ext}} \leftarrow E_{\text{ext}} - \{R_0^p : (R_0 \cup S_0, \prec_0) \in PS_{\text{match}}\}$ 
6 if  $E'_{\text{ext}} \neq \emptyset$  then
7   foreach  $v \in V$  do
8     if  $v$  is an activity node then  $visited[ps_v] \leftarrow false$ , for each poset  $ps_v$  of  $v$ 
9     else  $visited[v] \leftarrow false$ 
10   $PS_{\text{tail}} \leftarrow VisitSuccessorRP(p, u, E'_{\text{ext}}, (\emptyset, \emptyset))$  //  $u$  is  $v_0$ 's successor
11  foreach  $(R_0 \cup S_0, \prec_0) \in PS_{\text{match}}$  do
12     $Min \leftarrow \{s \in \min((S_0, \prec_0)) : loc(s) = p\}$  // Minimum sendings of  $p$  in  $ps_0$ 
13     $F \leftarrow \{r \in E_{\text{ext}} \cap R_0^p : \exists s \in Min, s \prec_0 r\}$ 
14     $\prec_{\text{ott}} \leftarrow \prec_{\text{tt}} \cup \{(r_1, r_2) \in \prec_0 : (r_1, r_2 \in F) \vee (r_1 \in E_{\text{tt}} \cap R_0^p \wedge r_2 \in F)\}$ 
15     $\prec_{\text{ott}} \leftarrow \prec_{\text{ott}} \cup \{(r_1, r_2) : r_1 \in (E_{\text{tt}} - R_0^p) \wedge r_2 \in F \wedge EventsInRace[r_1][r_2] = \emptyset\}$ 
16     $synched \leftarrow CheckSynchronization(p, v_0, (R_0 \cup S_0, \prec_0))$ 
17     $\Upsilon \leftarrow \Upsilon \cup UpdatePoset(synched, PS_{\text{tail}}, \prec_{\text{ott}}, E_{\text{tt}}, F, E'_{\text{ext}}, R_0^p, Min)$ 
18 else
   /*  $v_0$  is a fork node */
19  $v_{\text{join}} \leftarrow null$  // Global variable updated inside  $VisitSuccessorWC$ 
20  $(SynchPaths, UnsynchPaths) \leftarrow MapForkWC(v_0, \{p\}, (\emptyset, \emptyset))$ 
   /* Get the posets in  $PS_{\text{fork}}$  that have the most common events with the
   triggering trace */
21  $PS_{\text{fork}} \leftarrow SynchPaths \cup UnsynchPaths$ 
22  $PS_{\text{match}} \leftarrow \{(seq, (E_1, \prec_1), \mathcal{R}) \in PS_{\text{fork}} : \exists (E_2, \prec_2) \in (PS_{\text{fork}} - \{(seq, (E_1, \prec_1), \mathcal{R})\}), |E_{\text{tt}} \cap E_2^p| >$ 
    $|E_{\text{tt}} \cap E_1^p|\}$ 
23  $PS_{\text{tail}} \leftarrow \emptyset$ 
24  $E'_{\text{ext}} \leftarrow E_{\text{ext}} - \{R_0^p : (seq, (R_0 \cup S_0, \prec_0), \mathcal{R}) \in PS_{\text{match}}\}$ 
25 if  $E'_{\text{ext}} \neq \emptyset$  then
26   foreach  $v \in V$  do
27     if  $v$  is an activity node then  $visited[ps_v] \leftarrow false$ , for each poset  $ps_v$  of  $v$ 
28     else  $visited[v] \leftarrow false$ 
29    $PS_{\text{tail}} \leftarrow VisitSuccessorRP(p, u, E'_{\text{ext}}, (\emptyset, \emptyset))$  //  $u$  is  $v_{\text{join}}$ 's successor
30   foreach  $(seq, (R_0 \cup S_0, \prec_0), \mathcal{R}) \in PS_{\text{match}}$  do
31      $Min \leftarrow \{s \in \min((S_0, \prec_0)) : loc(s) = p\}$  // Minimum sendings of  $p$  in  $ps_0$ 
32      $F \leftarrow \{r \in E_{\text{ext}} \cap R_0^p : \exists s \in Min, s \prec_0 r\}$ 
33      $\prec_{\text{ott}} \leftarrow \prec_{\text{tt}} \cup \{(r_1, r_2) \in \prec_0 : (r_1, r_2 \in F) \vee (r_1 \in E_{\text{tt}} \cap R_0^p \wedge r_2 \in F)\}$ 
34      $\prec_{\text{ott}} \leftarrow \prec_{\text{ott}} \cup \{(r_1, r_2) : r_1 \in (E_{\text{tt}} - R_0^p) \wedge r_2 \in F \wedge EventsInRace[r_1][r_2] = \emptyset\}$ 
35      $\Upsilon \leftarrow \Upsilon \cup UpdatePoset((\mathcal{R} = \emptyset), PS_{\text{tail}}, \prec_{\text{ott}}, E_{\text{tt}}, F, E'_{\text{ext}}, R_0^p, Min)$ 
36 return  $\Upsilon$ 

```

---

---

**Procedure UpdatePoset** (*synched*,  $PS_{\text{tail}}$ ,  $\prec_{\text{ott}}$ ,  $E_{\text{tt}}$ ,  $F$ ,  $E_{\text{ext}}$ ,  $R_0^p$ , *Min*)
 

---

```

1  $\Upsilon \leftarrow \emptyset$ 
2 if  $PS_{\text{tail}} = \emptyset$  then
3    $\Upsilon \leftarrow \Upsilon \cup \{(E_{\text{tt}} \cup F, \prec_{\text{ott}})\}$ 
4 else
5   if !synched then
6     foreach  $((E_{\text{tail}}, \prec_{\text{tail}}), \text{synchB}) \in PS_{\text{tail}}$  do
7        $\prec_{\text{ott}} \leftarrow \prec_{\text{ott}} \cup \prec_{\text{tail}} \cup \{(r_1, r_2) : r_1 \in E_{\text{tt}} \cup F \wedge r_2 \in E_{\text{tail}} \wedge \text{EventsInRace}[r_1][r_2] = \emptyset\}$ 
8        $\Upsilon \leftarrow \Upsilon \cup \{(E_{\text{tt}} \cup F \cup E_{\text{tail}}, \prec_{\text{ott}})\}$ 
9     else if  $\exists r \in E_{\text{tt}} \cap R_0^p$  such that
10       $\forall s \in \text{Min}, s \neq r$  and  $\text{EventsInRace}[r][r'] \neq \emptyset$ , for any  $r' \in E_{\text{ext}}$  then
11       foreach  $((E_{\text{tail}}, \prec_{\text{tail}}), \text{synched}) \in PS_{\text{tail}}$  do
12          $E_{\text{tail}} \leftarrow \{r_2 \in E_{\text{tail}} : \text{EventsInRace}[r_1][r_2] \neq \emptyset \text{ for any } r_1 \in E_{\text{tt}} \cap R_0^p\}$ 
13          $\prec_{\text{ott}} \leftarrow \prec_{\text{ott}} \cup (\prec_{\text{tail}} \cap (E_{\text{tail}} \times E_{\text{tail}}))$ 
14          $\prec_{\text{ott}} \leftarrow \prec_{\text{ott}} \cup \{(r_1, r_2) : r_1 \in E_{\text{tt}} \cap R_0^p \wedge r_2 \in E_{\text{tail}} \wedge \text{EventsInRace}[r_1][r_2] = \emptyset\}$ 
15          $\Upsilon \leftarrow \Upsilon \cup \{(E_{\text{tt}} \cup F \cup E_{\text{tail}}, \prec_{\text{ott}})\}$ 
16     else
17        $\Upsilon \leftarrow \Upsilon \cup \{(E_{\text{tt}} \cup F, \prec_{\text{ott}})\}$ 
18 return  $\Upsilon$ 

```

---

---

**Procedure VisitSuccessorRP**( $p, v, E_{\text{ext}}, ps_{\text{aux}}$ )
 

---

```

1  if  $v$  is an activity node then
2    if  $p$  participates in  $v$  then
3       $ps_{\text{old}} \leftarrow ps_{\text{aux}}; \Upsilon \leftarrow \emptyset$ 
4      foreach poset  $ps_v = (R_v \cup S_v, \prec_v)$  of  $v$  such that  $!visited[ps_v]$  do
5         $ps_{\text{aux}} \leftarrow ps_{\text{old}}; visited[ps_v] \leftarrow true$ 
6        /* Consider  $ps_{\text{aux}} = (E_{\text{aux}}, \prec_{\text{aux}})$  and  $ps_v = (R_v \cup S_v, \prec_v)$  */
7         $F \leftarrow \{r \in R_v^p \cap E_{\text{ext}}\}$ 
8         $E_{\text{ext}} \leftarrow E_{\text{ext}} - F$ 
9         $E_{\text{aux}} \leftarrow E_{\text{aux}} \cup F$ 
10        $\prec_{\text{aux}} \leftarrow \prec_{\text{aux}} \cup \{(r_1, r_2) : r_1 \in E_{\text{aux}}, r_2 \in F, EventsInRace[r_1][r_2] = \emptyset\}$ 
11        $\prec_{\text{aux}} \leftarrow \prec_{\text{aux}} \cup \{(r_1, r_2) \in \prec_v, r_1, r_2 \in F\}$ 
12       if  $E_{\text{ext}} = \emptyset$  then
13          $\Upsilon \leftarrow \Upsilon \cup \{(ps_{\text{aux}}, false)\}$ 
14       else
15         if CheckSynchronization( $p, v, ps_v$ ) = true then
16            $\Upsilon \leftarrow \Upsilon \cup \{(ps_{\text{aux}}, true)\}$ 
17         else
18            $\Upsilon \leftarrow \Upsilon \cup VisitSuccessorRP(p, u, E_{\text{ext}}, ps_{\text{aux}})$  //  $u$  is  $v$ 's successor
19        $visited[ps_v] \leftarrow false$ 
20     else
21        $\Upsilon \leftarrow VisitSuccessorRP(p, u, E_{\text{ext}}, ps_{\text{aux}})$  //  $u$  is  $v$ 's successor
22   return  $\Upsilon$  // Backtrack
23 else
24   /* Control node */
25   if  $v$  is a final node then
26     return  $\{(ps_{\text{aux}}, false)\}$ 
27   else if  $v$  is a join node then
28      $v_{\text{join}} \leftarrow v$  // Global variable used inside MapForkRP()
29     return  $\{(ps_{\text{aux}}, false)\}$ 
30   else if  $v$  is a fork node and  $!visited[v]$  then
31      $visited[v] \leftarrow true$ 
32      $\Upsilon \leftarrow TraverseForkRP(p, v, E_{\text{ext}}, ps_{\text{aux}})$ 
33      $visited[v] \leftarrow false$ 
34     return  $\Upsilon$ 
35   else
36      $ps_{\text{old}} \leftarrow ps_{\text{aux}}$ 
37     foreach  $u$  successor of  $v$  do
38       if  $E_{\text{ext}} \neq \emptyset$  then
39          $ps_{\text{aux}} \leftarrow ps_{\text{old}}$ 
40          $\Upsilon_{\text{aux}} \leftarrow VisitSuccessorRP(p, u, E_{\text{ext}}, ps_{\text{aux}})$ 
41          $E_{\text{ext}} \leftarrow E_{\text{ext}} - \{E_{\text{aux}} : (E_{\text{aux}}, \prec_{\text{aux}}) \in \Upsilon_{\text{aux}}\}$ 
42          $\Upsilon \leftarrow \Upsilon \cup \Upsilon_{\text{aux}}$ 
43     return  $\Upsilon$  // Backtrack

```

---

---

**Procedure TraverseForkRP**( $p, v_{\text{fork}}, E_{\text{ext}}, ps_{\text{aux}}$ )
 

---

```

1  $v_{\text{join}} \leftarrow \text{null}$  // Global variable updated when a join node is visited
2  $E_{\text{ext}}^{\text{new}} \leftarrow E_{\text{ext}}$ 
3 foreach successor  $u$  of  $v_{\text{fork}}$  do
4   if  $E_{\text{ext}}^{\text{new}} \neq \emptyset$  then
5      $\Upsilon \leftarrow \text{VisitSuccessorRP}(p, u, E_{\text{ext}}, (\emptyset, \emptyset))$ 
6      $\Upsilon_{\text{fork}} \leftarrow \Upsilon_{\text{fork}} \cup \Upsilon$ 
7      $E_{\text{ext}}^{\text{new}} \leftarrow E_{\text{ext}}^{\text{new}} - \{E : ((E, \prec), \text{synchB}) \in \Upsilon, \text{ for any } \prec, \text{synchB}\}$ 
8  $\Upsilon \leftarrow \emptyset$ 
9 foreach  $\Upsilon_{\text{path}} \in \text{getCombs}(\Upsilon_{\text{fork}})$  do
10   $(E_{\text{forkP}}, \prec_{\text{forkP}}) \leftarrow \text{CausalOrderPar}(\{ps : (ps, \text{synchB}) \in \Upsilon_{\text{path}}, \text{ for any } \text{synchB}\})$ 
11   $\prec_{\text{aux}} \leftarrow \prec_{\text{aux}} \cup \prec_{\text{forkP}} \cup \{(r_1, r_2) : r_1 \in E_{\text{aux}} \wedge r_2 \in E_{\text{forkP}} \wedge \text{EventsInRace}[r_1][r_2] = \emptyset\}$ 
12   $E_{\text{aux}} \leftarrow E_{\text{aux}} \cup E_{\text{forkP}}$ 
13  if ( $v_{\text{join}} = \text{null}$ )  $\vee$  ( $E_{\text{ext}}^{\text{new}} = \emptyset$ )  $\vee$  ( $\exists (ps, \text{true}) \in \Upsilon_{\text{path}}, \text{ for any } ps$ ) then
14     $\Upsilon \leftarrow \Upsilon \cup \{(E_{\text{aux}}, \prec_{\text{aux}}), \text{true}\}$ 
15  else
16     $v_{\text{join\_succ}} \leftarrow \text{successor of } v_{\text{join}}$ 
17     $\Upsilon \leftarrow \Upsilon \cup \text{VisitSuccessorRP}(p, v_{\text{join\_succ}}, E_{\text{ext}} - E_{\text{forkP}}, (E_{\text{aux}}, \prec_{\text{aux}}))$ 
18 return  $\Upsilon$ 

```

---



---

**Procedure CheckSynchronization**( $p, v, ps_v$ )
 

---

```

1  $\text{synched} \leftarrow \text{false}$ 
2 if  $p$  has a sending in  $ps_v$  then
3    $\text{synched} \leftarrow \text{true}$ 
4   foreach successor activity  $u$  of  $v$  do
5     foreach poset  $ps_u$  of  $u$  do
6        $I \leftarrow \{loc(e) : e \in \text{min}(ps_u)\}$  // Initiating roles of  $ps_u$ 
7        $\text{foreach } q \in I \text{ do } \text{Max}[q] \leftarrow \{e : e \in \text{max}(ps_v) \wedge loc(e) = q\}$ 
8        $\text{Min} \leftarrow \{s : s \in \text{min}((S_v, \prec_v)) \wedge loc(s) = p\}$  // Min sendings in  $ps_v$  of  $p$ 
9       if  $\exists q \in I$  such that  $\forall m \in \text{Max}[q], \exists s \in \text{Min}, s \prec_v m$  then
10         $\text{synched} \leftarrow \text{false}$ 
11      else
12         $\text{visited}[ps_u] \leftarrow \text{true}$ 
13 return  $\text{synched}$ 

```

---

## 11.6 Conclusions

We have outlined a collaboration-oriented service specification approach, where UML 2 collaborations are used to specify services. The behavior of elementary collaborations is described by means of UML sequence diagrams, while the behavior of composite collaborations is described with help of a choreography graph (following the notation of UML activity diagrams) that defines the execution order of its sub-collaborations. We have provided a formal syntax and semantics to choreography graphs and sequence diagrams in terms of partial orders.

We have discussed realizability of choreographies in terms of the composition operators: weak and strong sequence, alternative, interruption and parallel. For each composition operator we have studied the problems that can lead to difficulties of realization. We have investigated the actual nature of these problems and discussed possible solutions to prevent or remedy them. The result of our study is a better understanding of the actual nature of realizability problems. Not surprisingly, we have seen that implicit concurrency and competing initiatives are at the heart of most problems. The send-causality property identified in this paper helps to build specifications that are more intuitive and less prone to conflicts, since it forces concurrency to be explicitly specified (i.e. by means of parallel composition or interruption). We have shown that some problems can already be detected at an abstract collaboration level, without needing to look into detailed interactions. We have also shown that generic solutions to the discussed problems are not valid. The same type of problem may require different resolutions in different contexts.

Finally, we have presented a set of algorithms for the detection of the problems discussed in this paper, and are currently working on their implementation.

## References

- [AEY00] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of message sequence charts. In *22nd Int. Conf. on Software Engineering (ICSE'00)*, 2000.
- [AEY05] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Realizability and verification of MSC graphs. *Theor. Comput. Sci.*, 331(1):97–114, 2005.
- [AHP96] Rajeev Alur, Gerard J. Holzmann, and Doron Peled. An analyzer for message sequence charts. *Software - Concepts and Tools*, 17(2):70–77, 1996.
- [AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.
- [BAL97] Hanene Ben-Abdallah and Stefan Leue. Syntactic detection of process divergence and non-local choice in message sequence charts. In *Proc. 2nd Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, 1997.

- [BG86] Gregor Bochmann and Reinhard Gotzhein. Deriving protocol specifications from service specifications. In *Proc. of ACM SIGCOMM Symposium*, pages 148–156, 1986.
- [BM03] Nicolas Baudru and Rémi Morin. Safe implementability of regular message sequence chart specifications. In *Proc. of ACIS 4th Intl. Conf. on Soft. Eng., Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'03)*, pages 210–217, 2003.
- [BM05] Rolv Bræk and Geir Melby. *Model Driven Service Engineering*, chapter of Model-driven Software Development. Volume II of Research and Practice in Software Engineering. Springer, 2005.
- [Boc78] Gregor Bochmann. Finite state description of communication protocols. *Computer Networks*, 2:361–372, 1978.
- [Bræ79] Rolv Bræk. Unified system modeling and implementation. In *International Switching Symposium (ISS)*. ISS Committee, 1979.
- [BS05] Yves Bontemps and Pierre-Yves Schobbens. The complexity of live sequence charts. In *Proc. of 8th Intl. Conf. on Foundations of Software Science and Computational Structures (FoSSaCS '05)*, pages 364–378, 2005.
- [CB06a] Humberto N. Castejón and Rolv Bræk. A collaboration-based approach to service specification and detection of implied scenarios. In *Proc. of 5th int. workshop on Scenarios and state machines: models, algorithms and tools (SCESM'06)*. ACM Press, 2006.
- [CB06b] Humberto N. Castejón and Rolv Bræk. Formalizing collaboration goal sequences for service choreography. In *Proc. of the 26th IFIP WG 6.1 Intl. Conf. on Formal Methods for Networked and Distributed Systems (FORTE'06)*, volume 4229 of *LNCS*, pages 275–291, Paris, France, September 2006. Springer-Verlag.
- [CKS05] Chien-An Chen, Sara Kalvala, and Jane Sinclair. Race conditions in message sequence charts. In *Proc. of 3rd Asian Symposium on Programming Languages and Systems (APLAS'05)*, volume 3780 of *LNCS*, pages 195–211. Springer, 2005.
- [Erl05] Thomas Erl. *Service Oriented Architecture: Concepts, Technology and Design*. Number ISBN 0-13-185858-0. Prentice Hall, 2005.
- [FK01] Kathi Fisler and Shriram Krishnamurthi. Modular verification of collaboration-based software designs. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 152–163. ACM Press, 2001.

- [GMSZ06] Blaise Genest, Anca Muscholl, Helmut Seidl, and Marc Zeitoun. Infinite-state high-level MSCs: Model-checking and realizability. *J. Comput. Syst. Sci.*, 72(4):617–647, 2006.
- [GY84] Mohamed G. Gouda and Yao-Tin Yu. Synthesis of communicating finite state machines with guaranteed progress. *IEEE Trans. on Communications*, Com-32(7):779–788, July 1984.
- [Hél01] Loïc Hérouët. Some pathological message sequence charts, and how to detect them. In *10th Intl. SDL Forum*, volume 2078 of *LNCS*, pages 348–364. Springer-Verlag, 2001.
- [HJ00] Loïc Hérouët and Claude Jard. Conditions for synthesis of communicating automata from HMSCs. In *Proc. of 5th Intl. Workshop on Formal Methods for Industrial Critical Systems (FMICS'00)*. GMD FOKUS, 2000.
- [HMU00] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2000.
- [IT99] ITU-T. *ITU Recommendation Z.Z.120: "Message Sequence Chart (MSC-2000)"*. ITU, Geneva, 1999.
- [IT00] ITU-T. *ITU Recommendation Z.100: "The Specification and Description Language (SDL)"*. ITU, Geneva, 2000.
- [Kie97] Astrid Kiehn. Observing partial order runs of petri nets. In *Foundations of Computer Science: Potential - Theory - Cognition, to Wilfried Brauer on the occasion of his sixtieth birthday*, pages 233–238, London, UK, 1997. Springer-Verlag.
- [KL98] J. P. Katoen and L. Lambert. Pomsets for message sequence charts. In H. König and P. Langendörfer, editors, *Formale Beschreibungstechniken fuer verteilte Systeme, 8. GI/ITG-Fachgespräch, Cottbus, Germany*, pages 197–207. Shaker Verlag, 1998.
- [KM03] Ingolf H. Krüger and Reena Mathew. Component synthesis from service specifications. In *2003 Dagstuhl Workshop on Scenarios: Models, Transformations and Tools*, volume 3466 of *LNCS*, pages 255–277. Springer, 2005.
- [KZ05] Ferhat Khendek and Xiao Jun Zhang. From MSC to SDL: Overview and an application to the autonomous shuttle transport system. In *2003 Dagstuhl Workshop on Scenarios: Models, Transformations and Tools*, volume 3466 of *LNCS*, pages 228–254. Springer, 2005.
- [MGR05] Arjan J. Mooij, Nicolae Goga, and Judi Romijn. Non-local choice and beyond: Intricacies of MSC choice nodes. In *Proc. Intl. Conf. on Fundamental Approaches to Soft. Eng. (FASE'05)*, volume 3442 of *LNCS*. Springer, 2005.

- [Mit05] Bill Mitchell. Resolving race conditions in asynchronous partial order scenarios. *IEEE Trans. Softw. Eng.*, 31(9):767–784, 2005.
- [MP00] Anca Muscholl and Doron Peled. Analyzing message sequence charts. In *SAM*, pages 3–17, 2000.
- [MRW06] Arjan Mooij, Judi Romijn, and Wieger Wesselink. Realizability criteria for compositional msc. In *Proc. of 11th Intl. Conf. on Algebraic Methodology and Software Technology (AMAST'06)*, volume 4019 of *LNCS*. Springer, 2006.
- [Mus00] Anca Muscholl. Compositional issues on message sequence charts. In *Proc. Workshop on Logic and Algebra in Concurrency*, 2000.
- [OMG07] Object Management Group (OMG). *UML 2.1.1 Superstructure Spec.*, February 2007.
- [ON05] Akimitsu Ono and Shin-Ichi Nakano. Constant time generation of linear extensions. In *15th Intl. Symp. on Fundamentals of Computation Theory (FCT'05)*, pages 445–453, 2005.
- [RAB<sup>+</sup>92] T. Reenskaug, E.P. Andersen, A.J. Berre, A. Hurlen, A. Landmark, O.A. Lehne, E. Nordhagen, E. Ness-Ulseth, G. Oftedal, A.L. Skaar, and P. Stenslet. OORASS: Seamless support for the creation and maintenance of object-oriented systems. *Journal of Object-oriented Programming*, 5(6):27–41, 1992.
- [RGG01] Frank Rößler, Birgit Geppert, and Reinhard Gotzhein. Collaboration-based design of SDL systems. In *Proc. of the 10th Intl. SDL Forum*, volume 2078 of *LNCS*, pages 72–89. Springer-Verlag, 2001.
- [RWL96] Trygve Reenskaug, P. Wold, and O. A. Lehne. *Working with Objects: The OOram Software Engineering Method*. Prentice Hall, 1996.
- [San00] Richard Sanders. Implementing from SDL. *Teletronikk*, 96(4), 2000.
- [SCKB05] Richard Torbjørn Sanders, Humberto N. Castejón, Frank Alexander Kraemer, and Rolv Bræk. Using UML 2.0 collaborations for compositional service specification. In *ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 3713 of *LNCS*, pages 460–475, Montego Bay, Jamaica, October 2005. Springer-Verlag.
- [UKM04] Sebastián Uchitel, Jeff Kramer, and Jeff Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Trans. Softw. Eng. Methodol.*, 13(1):37–85, 2004.
- [Woo87] Derick Wood. *Theory of Computation*. John Wiley & Sons, Inc., New York, 1987.

## Appendix 11.A Propositions and Proofs

We show here that in sequence diagrams with both the send-causality and the non-crossing messages properties, race conditions may only occur between two or more consecutive receiving events. We introduce first some useful propositions, which will help on the demonstration.

**Proposition 11.20.** *In a basic sequence diagram with the send-causality property all sending events are causally ordered (i.e. there is a total causal order on sending events).*

*Proof.* We prove it for out-of-order delivery semantics, so the result is also valid for in-order delivery semantics.

Consider two consecutive sending events according to  $<_m$ , that is,  $s, s' \in S, s <_m s' \wedge \exists s'' \in S, s <_m s'' <_m s'$ . Then, from the definition of send-causal sequence diagram (Definition 11.16), we have that either  $loc(s') = loc(s)$  or  $loc(s') = loc(rcv(s))$ . If  $loc(s') = loc(s) = p$ , then  $s <_p s'$ , and from the definition of causal order with out-of-order delivery (Definition 11.2), we conclude  $s \prec_{nf} s'$ . Otherwise, if  $loc(s') = loc(rcv(s)) = p$ , then  $rcv(s) <_p s'$ . Again, by Definition 11.2, we conclude  $s \prec_{nf} s'$ . Since any two consecutive sending events are causally ordered, and by transitivity of  $\prec_{nf}$ , we conclude that all sending events are causally ordered.  $\square$

**Corollary 11.21.** *A send-causal basic sequence diagram has a unique initiating event.*

Note that, in the absence of parallel composition, send-causality imposes a total causal order of sending events for each alternative behavior. If a sequence diagram describes a parallel composition by means of a *par* construct, the sending events within each operand of the construct are related by a total causal order. In addition, the sending events preceding (resp. succeeding) the *par* construct are causal predecessors (resp. successors) of all sending events within the *par* construct.

Now we demonstrate that in a send-causal sequence diagram, if a receiving event  $r \in R$  is specified to happen after a sending event  $s \in S$  on the same lifeline  $p \in \mathcal{P}$  (i.e.  $s <_p r$ ), then  $r$  is always causally dependent on  $s$ , with independence of the communication architecture (i.e.  $r$  will always happen after  $s$  in a any realized system).

**Proposition 11.22.** *In a send-causal sequence diagram satisfying the non-crossing messages property, the following is always true: given a sending event  $s \in S$  and a receiving event  $r \in R$  located on the same lifeline  $p \in \mathcal{P}$  (i.e.  $loc(s) = loc(r) = p$ ), we have that  $s <_p r \Rightarrow s \prec_{nf} r \wedge s \prec_f r$ .*

*Proof.* We prove it for the out-of-order delivery causal order ( $\prec_{nf}$ ), since it corresponds to a communication architecture without restrictions. The results will then be valid for any more restrictive order (e.g.  $\prec_f$ ). We consider first the case where  $s$  and  $r$  are located within the same basic sequence sub-diagram. By Proposition 11.20, we know that  $s \prec_{nf} snd(r)$ . We conclude, therefore, that  $s \prec_{nf} r$ .

We consider now two basic sequence sub-diagrams,  $SD_1$  and  $SD_2$ , such that  $SD_1 \text{ seq } SD_2$ . We assume  $s$  is located in  $SD_1$  and  $r$  is located in  $SD_2$ . We know that:

- (i) Either  $s$  is a maximum sending event of  $SD_1$  (i.e.  $s \in \mathcal{T}_s, \mathcal{T}_s \in \text{term}_{\text{snd}}(SD_1)$ ), or, by Proposition 11.20, we have that  $s \prec_{nf} s_t, \forall s_t \in \mathcal{T}_s, \mathcal{T}_s \in \text{term}_{\text{snd}}(SD_1)$ .
- (ii) Either  $\text{snd}(r)$  is a minimum event of  $SD_2$  (i.e.  $\text{snd}(r) \in \mathcal{I}, \mathcal{I} \in \text{init}(SD_2)$ ), or, by Proposition 11.20, we have that  $s_i \prec_{nf} \text{snd}(r), \forall s_i \in \mathcal{I}, \mathcal{I} \in \text{init}(SD_2)$ .

By Definition 11.16, we also know that  $\forall \mathcal{T}_s \in \text{term}_{\text{snd}}(SD_1), \forall \mathcal{I} \in \text{init}(SD_2), \forall s_t \in \mathcal{T}_s, \forall s_i \in \mathcal{I}, \text{loc}(s_i) = \text{loc}(s_t) \vee \text{loc}(s_i) = \text{loc}(\text{rcv}(s_t))$ . With this information, and applying the same reasoning as the one used in the proof of Proposition 11.20, we conclude that  $s \prec_{nf} r$ .

The above results can be easily generalized, by induction on the composite structure of the sequence diagram, to prove that in all cases  $s \prec_{nf} r$ .  $\square$

Given Proposition 11.22 and the fact that races may exist between consecutive receiving events (see, e.g., the race between  $e_4$  and  $e_6$  in Fig. 11.5(a)) we have the result we were looking for:

**Proposition 11.23.** *In a sequence diagram satisfying the send-causality property and the non-crossing messages property, a potential race condition exists between two receiving events  $r_1$  and  $r_2$ , located at the same lifeline  $p$ , if  $r_1 <_p r_2$  and there is not a sending event  $s \in S$  such that  $r_1 <_p s <_p r_2$ .*

### 11.A.1 Race Conditions in Send-Causal Sequence Diagrams

We study now the necessary conditions for a race to actually exist in a send-causal sequence diagram. Such conditions depend on the communication architecture. We consider two architectures, one with out-of-order delivery channels and other with in-order delivery channels.

#### Race conditions with in-order delivery channels

Messages cannot overtake each other in channels with in-order delivery. Therefore, there are not races between receiving events located on the same lifeline  $p$  if their associated sending events are both located on the same lifeline  $q$ . However, a race exists if the sending events are located on different lifelines and there is no sending event located on  $p$  between the receiving events.

**Proposition 11.24.** *Given a sequence diagram satisfying the send-causality property and the non-crossing messages property, and a communication architecture with in-order delivery channels, a race condition exists between two receiving events,  $r$  and  $r'$ , located on the same lifeline  $p \in \mathcal{P}$ , iff  $r <_p r' \wedge \text{loc}(\text{snd}(r)) \neq \text{loc}(\text{snd}(r')) \wedge \nexists s \in S, r <_p s <_p r'$ .*

*Proof.* ( $\Leftarrow$ ) We assume that  $r <_p r' \wedge \text{loc}(\text{snd}(r)) \neq \text{loc}(\text{snd}(r')) \wedge \bar{A}s \in \mathcal{S}, r <_p s <_p r'$  and prove that there is a race between  $r$  and  $r'$  (i.e.  $r \not\prec_f r'$ ). We do this by contradiction. Let us assume that  $r \prec_f r'$ . Then we should have that  $r \prec_f \text{snd}(r')$ . We recall that, according to Definition 11.3,  $\prec_f = (\prec_f)^*$ . It is easy to see that  $r \not\prec_f \text{snd}(r')$ , since  $r$  and  $\text{snd}(r')$  do not satisfy any of the conditions on Definition 11.3. Therefore, there must exist one or more events such that  $r <_f e_1 <_f \dots e_n <_f \text{snd}(r')$ . Since  $\text{loc}(\text{snd}(r)) \neq \text{loc}(\text{snd}(r'))$ , the relation  $r <_f e_1$  can only be true if  $e_1 \in \mathcal{S} \wedge r <_p e_1$ . This contradicts our initial assumption:  $\bar{A}s \in \mathcal{S}, r <_p s$ ; so we conclude that  $r \not\prec_f r'$ .

( $\Rightarrow$ ) We consider two receiving events,  $r$  and  $r'$ , that are located on the same lifeline  $p \in \mathcal{P}$  and that are in race, so that  $r \not\prec_f r'$ . By the definition of race condition, we know that  $r <_p r'$ . We also know that in-order delivery semantics do not allow message overtaking, so if  $\text{loc}(\text{snd}(r)) = \text{loc}(\text{snd}(r'))$  there cannot be a race. Therefore it must be  $\text{loc}(\text{snd}(r)) \neq \text{loc}(\text{snd}(r'))$ . We just need to prove that  $\bar{A}s \in \mathcal{S}, r <_p s <_p r'$ . Let us assume that  $\exists s \in \mathcal{S}, r <_p s <_p r'$ . Then, by Definition 11.3, we have that  $r \prec_f s$ , and by Proposition 11.22, we have that  $s \prec_f r'$ . This implies  $r \prec_f r'$ , which is a contradiction. Therefore  $\bar{A}s \in \mathcal{S}, r <_p s <_p r'$ .  $\square$

### Race conditions with out-of-order delivery channels.

With out-of-order delivery channels races always exist between receiving events located on the same lifeline  $p$  if there is no sending event located also  $p$  between the receiving events.

**Proposition 11.25.** *Given a sequence diagram satisfying the send-causality property and the non-crossing messages property, and a communication architecture with out-of-order delivery channels, a race condition exists between two receiving events,  $r$  and  $r'$ , located on the same lifeline  $p \in \mathcal{P}$ , iff  $r <_p r' \wedge \bar{A}s \in \mathcal{S}, r <_p s <_p r'$ .*

*Proof.* Since the causal order imposed by out-of-order delivery channels is less restrictive than the causal order imposed by in-order delivery channels (i.e.  $\prec_{nf} \subseteq \prec_f$ ), races will exist with out-of-order delivery channels whenever they exist with in-order delivery channels. The case of races between receiving events associated to messages sent by different lifelines is therefore proved by 11.24. We just need to prove that there are also races between receiving events associated to messages sent by the same lifeline. This can be easily done following the same reasoning as with Proposition 11.24, and having into account that messages may overtake each other in an out-of-order delivery channel.  $\square$

## Appendix 11.B Automata Theory

A finite state automaton is a tuple  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ , where  $Q$  is a finite set of states,  $\Sigma$  is a set of transition labels,  $\delta \subseteq Q \times \Sigma \times Q$  is a set of transitions,  $q_0 \in Q$  is the initial state, and  $F \subseteq Q$  is the set of final (or accepting) states.

### 11.B.1 Converting an FSA into a regular expression

There exist several methods for the conversion of an FSA into an equivalent regular expression (i.e. a regular expression accepting the same language as the FSA). Here we describe the state-elimination technique from [Woo87].

The intuitive idea behind the state-elimination technique is to bypass an state  $q \in Q - \{q_0, q_f\}$  by replacing that state, together with its incoming, outgoing and self-looping transitions, with new transitions. These new transitions are labeled with regular expressions, such that the resulting automaton accepts the same language as the original one.

As input for the state-elimination process we assume an automaton  $\mathcal{A} = (Q, \Sigma, \delta, q_0, q_f)$  with the following properties (which facilitate the elimination process):

- $\mathcal{A}$  has one single initial state, and this state has no input or self-looping transitions. If the original automaton does not fulfill this requisite, a new initial state  $q'_0$  can be added and connected to  $q_0$  by an  $\varepsilon$ -transition (i.e.  $Q' = Q \cup \{q'_0\}$ ,  $\delta' = \delta \cup \{(q'_0, \varepsilon, q_0)\}$ ).
- $\mathcal{A}$  has one single final state, and this state has no output or self-looping transitions. If this is not the case for the original automaton, the original final states are converted into normal states and connected to a new final state  $q'_f$  by  $\varepsilon$ -transitions (i.e.  $Q' = Q \cup \{q'_f\}$ ,  $\delta' = \delta \cup \{(q, \varepsilon, q'_f) : q \in F\}$ ,  $F' = \{q'_f\}$ ).
- Each state  $q \in Q - \{q_0, q_f\}$  has a unique self-looping transition  $(q, \beta, q) \in \delta$ . This is a merely accessory property and required to ease the explanation below. Note that if a state  $q \in Q - \{q_0, q_f\}$  has not self-looping transitions, we can assume a self-looping silent transition (i.e.  $\beta = \varepsilon$ ). And if  $q$  has several self-looping transitions, we can merge them.

Formally, a step of the state elimination technique can be seen as the transformation of an automaton  $\mathcal{A} = (Q, \Sigma, \delta, q_0, q_f)$  into a new automaton  $\mathcal{A}' = (Q - \{q\}, \Sigma', \delta', q_0, q_f)$ , where  $q \in Q - \{q_0, q_f\}$  is the state being eliminated.  $\mathcal{A}'$  is then used as input for the next step. The elimination process is repeated until we obtain an automaton  $\mathcal{A}_E = (\{q_0, q_f\}, \Sigma_E, \{(q_0, E, q_f)\}, q_0, q_f)$  that consists of exactly two states, the initial and final ones, and one transition between them. Such transition is labeled with a regular expression  $E$ , which accepts exactly the same language as the original automaton  $\mathcal{A}$ .

For the elimination of a state  $q \in Q - \{q_0, q_f\}$  we proceed as follows. For each input transition  $(p_i, u_i, q) \in \delta$ , with  $1 \leq i \leq n$  and  $n \geq 1$ , each output transition  $(q, v_j, r_j) \in \delta$ , with  $1 \leq j \leq m$  and  $m \geq 1$ , and the self-looping transition  $(q, w, q) \in \delta$ , we create a new transition  $(p_i, u_i \cdot w^* \cdot v_j, r_j)$ . We then obtain  $\Sigma'$  and  $\delta'$  as indicated below:

$$\begin{aligned} \delta' &= \delta - (\{(p_i, u_i, q), (q, v_j, r_j) : 1 \leq i \leq n \wedge n \geq 1 \wedge 1 \leq j \leq m \wedge m \geq 1\} \cup \{(q, w, q)\}) \\ &\quad \cup \{(p_i, u_i \cdot w^* \cdot v_j, r_j) : 1 \leq i \leq n \wedge n \geq 1 \wedge 1 \leq j \leq m \wedge m \geq 1\} \\ \Sigma' &= \Sigma \cup \{u_i \cdot w^* \cdot v_j : 1 \leq i \leq n \wedge n \geq 1 \wedge 1 \leq j \leq m \wedge m \geq 1\} \end{aligned}$$

If  $\delta'$  contains more than one transition between any two states, we merge those transitions. Formally, given two transitions  $(q_1, u, q_2), (q_1, v, q_2) \in \delta'$ , we remove them from  $\delta'$  and add a new transition  $(q_1, u | v, q_2)$ . We also add the label  $u | v$  to  $\Sigma'$ .

Figure 11.19(a) shows the automaton obtained after running the *ChoreographyTo-FSA* procedure on the choreography graph of Fig. 11.17(a). Figures 11.19(b) and 11.19(c) illustrate some steps of the state elimination technique applied to that automaton. The automaton in Fig. 11.19(b) results after eliminating states  $m_1, q_0, q_3, q_6$  and  $q_8$ . Note that since the two first properties discussed above were not satisfied by the automaton in Fig. 11.19(a), a new initial state,  $q'_0$ , and a new final state,  $q_f$ , were added. After eliminating all states, except the initial and final ones, the automaton in Fig. 11.19(c) is obtained. Merging the transitions of that automaton would give a new automaton with a single transition, whose label would correspond to a regular expression accepting the same language as the FSA in Fig. 11.19(a).

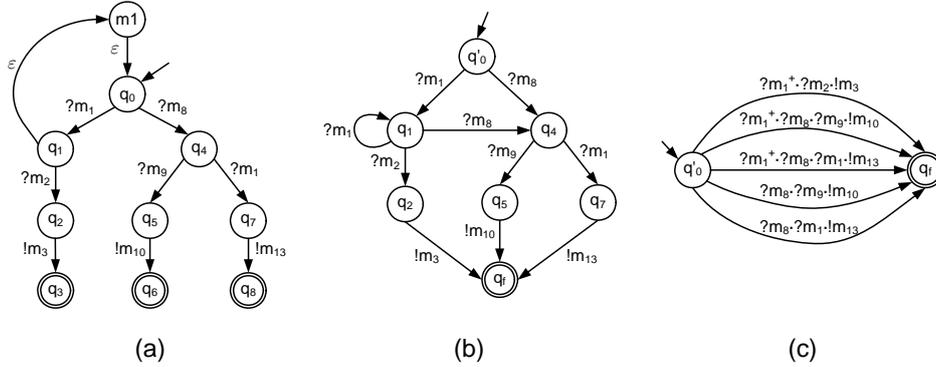


Figure 11.19: (a) FSA for the choreography of Fig. 11.17(a); (b) FSA resulting after eliminating all  $\varepsilon$ -transitions in (a); (c) FSA resulting after eliminating all non-initial and non-final states in (b)

### 11.B.2 Eliminating $\varepsilon$ -transitions

To eliminate these  $\varepsilon$ -transitions the technique described in [HMU00] can be used. This technique consists of three basic steps:

- a) Identify all states  $Q_\varepsilon$  with output  $\varepsilon$ -transitions.
- b) For each state  $p \in Q_\varepsilon$ , find all states  $Q'$  that can be reached using only  $\varepsilon$ -transitions (this can be done with a DFS technique). For each state  $q \in Q'$ , if there is a transition to state  $r$  on input  $e$  (i.e. a non  $\varepsilon$ -transition), then create a new transition  $(p, e, r)$  from  $p$  to  $r$  on input  $e$ . If any state  $q \in Q'$  is a final state, then make  $p$  a final state.
- c) Remove all  $\varepsilon$ -transitions and all unreachable states.

## Notes

<sup>1</sup>The second condition for the *non-crossing messages* property was wrongly formulated in the original technical report. In the original text it could be read

$$\forall e = \langle !m, p, p \rangle, \exists e' \neq rcv(\langle !m, p, p \rangle), e <_p e'$$

We note that such condition would be right if  $<_p$  was an immediate precedence relation, rather than a total order.

---

## Paper 7

### **Dynamic role binding in a service oriented architecture.**

By Humberto Nicolás Castejón and Rolv Bræk.

Published in the *Proceedings of the IFIP Intl. Conf. on Intelligence in Communication Systems (INTELLCOMM'05)*, volume 190 of *IFIP International Federation for Information Processing*. Springer, October 2005.

The original publication is available at [www.springerlink.com](http://www.springerlink.com) via [http://dx.doi.org/10.1007/0-387-32015-6\\_11](http://dx.doi.org/10.1007/0-387-32015-6_11)



# Dynamic Role Binding in a Service Oriented Architecture

Humberto Nicolás Castejón and Rolv Bræk

*NTNU, Department of Telematics, N-7491 Trondheim, Norway*  
{humberto.castejon,rolv.braek}@item.ntnu.no

## Abstract

Many services are provided by a structure of service components that are dynamically bound and performed by system components. Service modularity requires that service components can be developed separately, deployed dynamically and then used to provide situated services without undesirable service interactions. In this paper we introduce a two-dimensional approach where service components are roles defined using UML 2.0 collaborations and system components are agents representing domain entities such as users and terminals. The process of dynamic role binding takes place during service execution and provides general mechanisms to handle context dependency, personalisation, resource limitations and compatibility validation. A policy framework for these mechanisms is outlined.

## 12.1 Introduction

A *service* may generally be defined as an identified partial functionality provided by a system to an end user, such as a person or other system. The most general form of service involves several system components collaborating on an equal basis to provide the service to one or more users. This understanding of service is quite general and covers both client-server and peer-to-peer services as described in [BF04]. A common trait of many services is that the structure of collaborating components is dynamic. Links between components are created and deleted dynamically and many services and service features depend on whether the link can be established or not, and define what to do if it cannot be established (e.g. busy treatment in a telephone call). Indeed, setting up links is the goal of some services. For example, the goal of a telephone call is to establish a link between two system components, so that the users they represent can talk to each other. In the past this problem has often been addressed in service specific ways. It may however, be generalised to a

problem of *dynamic role binding*, i.e. requesting system components to play roles, such as for example requesting a `UserAgent` to play the `b`-subscriber in a call. The response to such a request may be to alert the end-user (if free and available), to reject the call, to forward it or to provide some waiting functionality (if busy). Which feature to select depends on what is subscribed, what other features are active, what resources are available, what the current context is and what the preferences of the user are. By recognising dynamic role binding as a general problem, we believe it is possible to find generic and service independent solutions. In fact, many crucial mechanisms can be associated with dynamic role binding: service discovery; feature negotiation and selection; context dependency resolution; compatibility validation of collaborating service components, and dependency resolution.

Modularity is a well-known approach for easing service development. Service modularity requires a separation of service components from system components, allowing the former ones to be specified and designed separately from the latter ones, then be incrementally deployed and finally be linked dynamically during service execution to provide actual services without undesirable service interactions. We will show that dynamic role binding mechanisms are crucial to achieve the desired separation and modularity and still be able to manage the complex mutual dependencies between service components and system components. It is desirable that such dependencies are not hard coded, but represented by information that can be easily configured and interpreted by general mechanisms, i.e. by some kind of policies.

In this paper we present a service architecture where service modularity and dynamic linking is supported by means of roles and general mechanisms for dynamic role binding. In Sect. 12.2 the main elements of the architecture are presented: agents mirroring the environment as system components; UML 2.0 collaborations and collaboration roles as service-modelling elements; and UML active classes as service components. In Sect. 12.3 we show how the proposed architecture provides structure to service-execution policies and how dynamic role binding enables policy-driven feature selection with compatibility guarantees. Finally we conclude with a summary of the presented work.

## 12.2 Agent and Role Based Service Architecture

Fig. 12.1 suggests an architecture for service-oriented systems which is characterized by horizontal and vertical composition. On the horizontal axis, system components are identified that may reside in different computing environments. This axis reflects domain entities (such as users, user communities and terminals) and resources that must be represented in a service providing system regardless of what services it provides. On the vertical axis, several services and service components are identified that depend on the system components of the architecture. This two-dimensional picture illustrates the crosscutting nature of services which is a well-known challenge in service engineering [Bræ99, Flo03, KGM<sup>+</sup>04, BCS00].

In the following sections we will present our particular realization of this architecture.

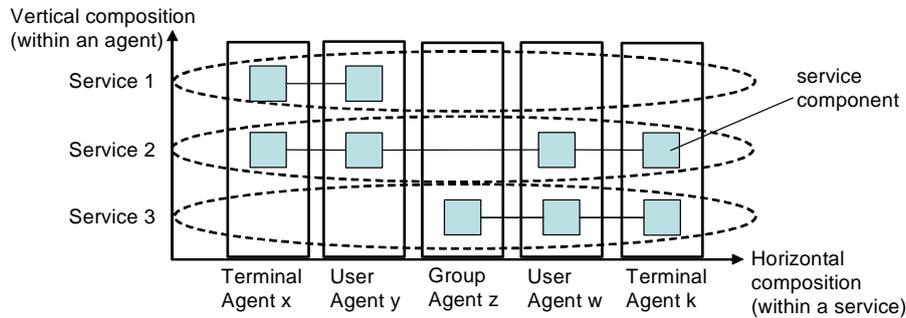


Figure 12.1: Service Oriented Architecture

### 12.2.1 Agents as System Components

In [BF04], Bræk and Floch identify two principal system architectures: the *agent oriented* and the *server oriented*. The agent oriented architecture follows the principle that a system should be structured to mirror objects in the domain and environment it serves [BH93]. This is a general principle known to give stable and adaptable designs. Agents may represent and have clear responsibilities for serving domain/environment entities and resources and thereby provide a single place to resolve dependencies. In the case of personalised communication services accessible over a number of different terminal types, this mirroring leads to a structure of **TerminalAgents** and **UserAgents** as illustrated in Fig. 12.1. In addition there may be agents corresponding to user communities (e.g. the **GroupAgent** in Fig. 12.1), service enablers and shared service functionality. Several authors have proposed similar architectures, for example [ZWO<sup>+</sup>95] and [AKGM00].

Note that such an agent structure reflects properties of the domain being served and not particular implementation details, nor particular services. It is therefore quite stable and service independent. At the same time agents provide natural containers for properties and policies of domain entities like users, terminals and service enablers.

### 12.2.2 UML Collaborations as Services and Roles as Service Components

As illustrated in Fig. 12.1, a service is a partial functionality provided by a collaboration between service components executed by agents to achieve a desired goal for the end users. UML 2.0 *collaborations* [OMG05] are well suited for service modelling as they are intended to describe partial functionalities provided by collaborating roles played by objects. An interesting characteristic of UML collaborations is that they can be applied as *collaboration uses* and employed as components in the definition of larger collaborations. This feature enables a compositional and incremental design of services, as we explain in [SCKB05, Cas05]. For instance, Fig. 12.2 shows a collaboration specifying a **UserCall** service in terms of collaboration roles linked

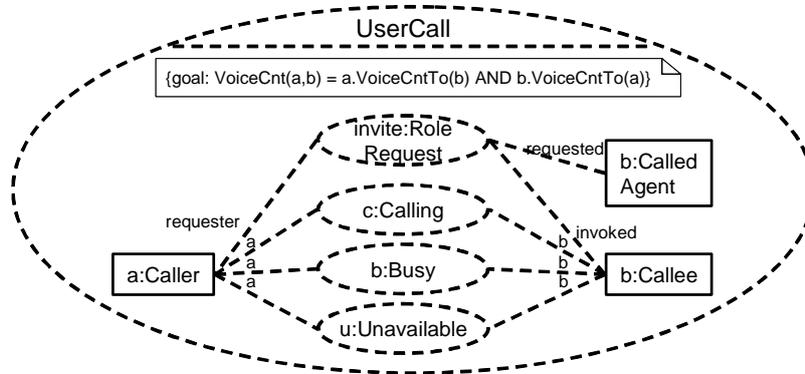


Figure 12.2: The UserCall Service as a UML 2.0 Collaboration

by collaboration uses, which correspond to different phases and features of the service. It also shows a simple goal expression representing a desired goal state for the collaboration. Behaviour descriptions can be associated with the collaboration to precisely define the service behaviour, including precise definitions of the visible interface behaviour that objects must show in order to participate in the collaboration. Collaborations thereby provide a mechanism to define *semantic interfaces* that can be used for service discovery and to ensure compatibility with respect to safety and liveness (i.e. reaching the desired goal states) when linking service components, as we discuss in [SBvBA05].

Ideally, service models should be independent of particular system structures. It can be argued, however, that it is necessary and beneficial to take a minimum of architectural aspects into account [Zav03]. The challenge is to do this at an abstraction level that fits the nature of the services without unduly binding design solutions and implementations. In our architecture (see Fig. 12.1) the horizontal axis represents the agent structure and the vertical axis represents the services modelled as collaborations with roles that are bound to agents. The service-independent agent structure is therefore instrumental, since it helps to identify and shape roles, without introducing undue bindings to implementation details. At the same time it provides an architectural framework for role composition, role binding and role execution.

In this service oriented architecture, service specific behaviour is the responsibility of (service-) roles while domain specific behaviour and policies are the responsibility of agents. Interactions between roles and agents are needed primarily in the process of creating and releasing dynamic links, that is, the process of *dynamic role binding*.

Roles need to be mapped to well defined service components, which can then be deployed and composed in agents to provide (new) services without causing safety or liveness problems. We do this by defining service components as UML active classes with behaviour defined by state machines. Note that service components may implement one or more UML collaboration roles composed by means of collaboration uses, as it is roughly illustrated in Figs. 12.2 and 12.4. Each of these collaboration

roles will correspond to a different service or service feature. We assume that service components are typed with semantic interfaces with well defined feature sets. This information is exploited when service components are dynamically linked within a service collaboration in order to ensure their compatibility in terms of safety and liveness criteria, as explained in [SBvBA05]. In addition, it is necessary to ensure that the service components can actually be bound to the intended agents. Their feature sets may also be restricted and dynamically selected during the binding process. These aspects will be discussed in the following sections.

Note that, for the sake of simplicity, in the rest of the paper we will use the word *role* to name service components.

### 12.2.3 Dynamic Role Binding

Dynamic Role binding has three distinct phases:

- a) *Agent identification*, which aims at identifying an agent by consulting a name-server or performing a service discovery. Some service features are related to the agent identification, e.g. aliasing, business domain restrictions or originating and terminating screening features in telephony.
- b) *Role request*, which aims at creating a dynamic link according to a semantic interface with an agreed feature set. This means to request the agent identified in phase 1 to play a role with a certain feature set, which can be negotiated. The role with the agreed feature set is finally invoked. The role request pattern [BHM02] described in Fig. 12.3 provides one partial solution to this. Using this protocol any role can request an agent to play a complimentary role. If the agent is able to play the requested role, it invokes it and a link is dynamically established between the requesting and the requested roles, so that they can collaborate. This is illustrated in Fig. 12.2, where a **UserAgent** is requested (invited) to play the **b** role in a **UserCall** collaboration. The response of the **UserAgent** in this case is to enable one of three features, represented as collaboration uses: **Calling**, **Busy** or **Unavailable**.
- c) *Role release*, which signals that a role is finished and has released whatever resources it had occupied.

Once a role is invoked it can proceed autonomously, until it reaches a state where interaction with agents is again required for one of the following reasons:

- it needs to bring a new role into the collaboration (i.e. create another dynamic link). In this case it first needs to identify the agent that should play the role and then initiate a role request to this agent, as explained above;
- it needs to check what feature or feature set to select at a certain point in its behaviour, if this depend on agent policy (e.g. if mid-call telephony services are allowed);

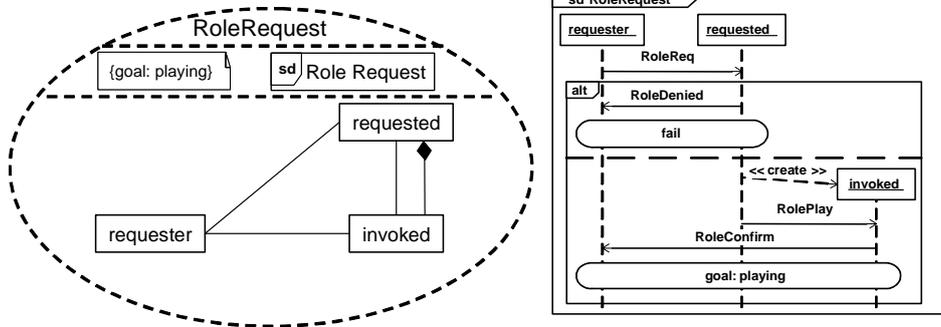


Figure 12.3: Role Request Pattern

- it needs to signal to its own agent that it is available for additional linking, in response to an incoming role request (e.g. to perform a call waiting feature); or
- it is finished and performs role release.

Note that all cases except the second are related to dynamic role binding. Also note that a large proportion of features are discovered, selected and initiated in connection with dynamic role binding.

## 12.3 Governing Service Execution with Policies

Dynamic role binding is clearly a very central mechanism in service execution. As we have already pointed out, associated with dynamic role binding are such key issues as: service discovery, feature negotiation and selection, context dependency resolution, compatibility validation and feature interaction detection/avoidance. The challenge is to find general, scalable and adaptable ways of dealing with those issues.

In general a role can only be bound to an agent without undesired side-effects if certain (pre-/post-) conditions hold. By explicitly expressing these conditions as constraints, we may check them upon role-binding and only allow the role to be invoked if they are satisfied. It is also important to give users the possibility to express their preferences to control the selection of features when, for example, the requested service can not be delivered. In the following we will use the term *policy* to cover both general role binding constraints and user preferences. In doing that we adhere to the usual definition of policy that can be found in the computer-science literature: *a rule or information that modifies or defines a choice in the behavior of a system* [LS99].

The agent architecture discussed in the previous section provides a natural way to structure policies into three groups:

- *Role-binding* policies, which constrain the binding of roles to agents at runtime.

- *Collaboration* policies, which express constraints that must hold for a collaboration (i.e. a service) as a whole when it is executed. They aim at preventing actions that may compromise the intentions and goals of the collaboration.
- *Feature-selection* policies, which control the triggering of context-dependent service features.

We will take a closer look at each of these policies in the following.

### 12.3.1 Role-binding and Collaboration Policies

Role-binding policies represent conditions that must be satisfied for a role to be bound to an agent. These policies may be associated with agent types, so they shall hold for all instances of that type, or they may be defined for specific agent instances (usually describing user preferences and/or user permissions). Finally, role-binding policies may also be associated with role types and they shall hold for all instances of that role type.

The role-binding policies associated with a role type define constraints that the role imposes on any agent it may be bound to and, thereby, indirectly on system resources. For example, the role-binding policy of a role may require that role to be bound to a **TerminalAgent** representing a specific terminal type with specific capabilities (e.g. a PDA).

The role-binding policies associated with an agent represent, on the contrary, constraints that the agent imposes on the roles it can play. When these policies are associated with an agent type, they represent constraints on the type and multiplicity of the roles that can be bound to the agent, as well as other constraints imposed by the service provider (e.g. that the user must hold a valid subscription to play a certain role). When they are associated with a particular agent instance, they represent user preferences and/or user permissions specifying when that particular agent should or should not play a certain role. These preferences/permissions can be seen to express context dependency (e.g. on location, calendar, presence or availability). For example, a user may define a role-binding policy for her **UserAgent** to express that it should only participate in a **UserCall** service, playing the **callee** role, if the invitation was received between 8 am and 11 pm.

Collaboration policies express constraints that must hold for a collaboration (i.e. a service) as a whole when it occurs. We may associate collaboration policies with a UML collaboration, so that they shall hold for all occurrences of that collaboration. For instance, a collaboration policy may be associated with a conference-call collaboration to prohibit the agent playing the conference-controller role to temporarily interrupt its participation in the service. This policy would specifically prohibit the conference-controller role to invoke the *hold* feature. Agent instances may also hold specialised collaboration policies that, in this case, shall only be satisfied for those occurrences of the collaboration where the agent participates. These policies may then represent user preferences. An important use of such user-defined collaboration policies is to constrain who participates in a service session. For personal communication services the identity of the agents participating in a service is important (e.g.

a calling user wants a specific user's **UserAgent** to play the **b** role - see Fig. 12.2). It is not only important who must be invited to a service, but also who cannot be invited. Some users may not want to talk to certain people, or they may not like, for example, to talk to a machine. We can easily solve this problem using collaboration policies by constraining the type and/or id of agents that can participate in a service session, as well as the roles they can play. For instance, if a user does not want to be redirected to an automatic-response machine, she may define a collaboration policy for the **UserCall** service constraining the participation of **IVRAgents**<sup>1</sup>. This policy would be held by her **UserAgent**, thus only affecting **UserCall** services in which she may participate.

Role-binding and collaboration policies are checked upon a role request by both the **requester** and the **requested** agents. The **requester** agent checks the collaboration policies associated to the service being (or to be) executed before the role request is sent. This is done to confirm that inviting the **requested** agent would not violate those policies (e.g. that a **UserAgent** representing an undesired user would not be invited when performing a forwarding). At the reception of the role request, the **requested** agent checks first the collaboration policies for conformance on joining the collaboration (e.g. to ensure that all other participating agents are welcome). Thereafter, it checks the role-binding policies concerning the requested role, which is only bound if those policies are satisfied.

Policies defined for a collaboration (and its roles) are “inherited“ when that collaboration is employed, as a collaboration use, in the specification of other collaborations. Therefore, when a collaboration is bound to a set of agents for its execution, all policies defined for the collaboration itself and for its sub-collaborations must hold. This is illustrated in Fig. 12.4. The upper part shows a collaboration specifying a **FullCall** service. This service is a collaboration between four roles and it is composed from the **UserCall** service described in Fig. 12.2 and two uses of a **TermCall** service, which specifies the collaboration between **UserAgents** and **TerminalAgents**. Policies have been defined for each of the roles and collaboration uses in **FullCall** (P2-P8). In addition a policy (P1) has been defined for the **FullCall** collaboration as a whole. The lower part of the figure illustrates a set of **UserAgents** and **TerminalAgents**, representing users and terminals, performing the **FullCall** service. It is important to note that for this collaboration use (i.e. **fcx:FullCall**) all and each of the policies defined for the **FullCall** collaboration must hold. This is indicated by the annotation **P1+P2+...+P8**. Moreover, each agent holds a set of role-binding, collaboration and feature selection policies, called **P10-P13** in the figure, that also must hold in the execution of **fcx:FullCall**. Therefore, if for example **at:TermCaller** had some collaboration policy that would not hold when inviting **bt:TermCallee**, **fcx:FullCall** could not be completed.

### 12.3.2 Feature Selection Policies

A feature can be defined as a unit of functionality in a base service. In general, we can differentiate two types of features, depending on how they are selected and

<sup>1</sup>IVR stands for Interactive Voice Response machine

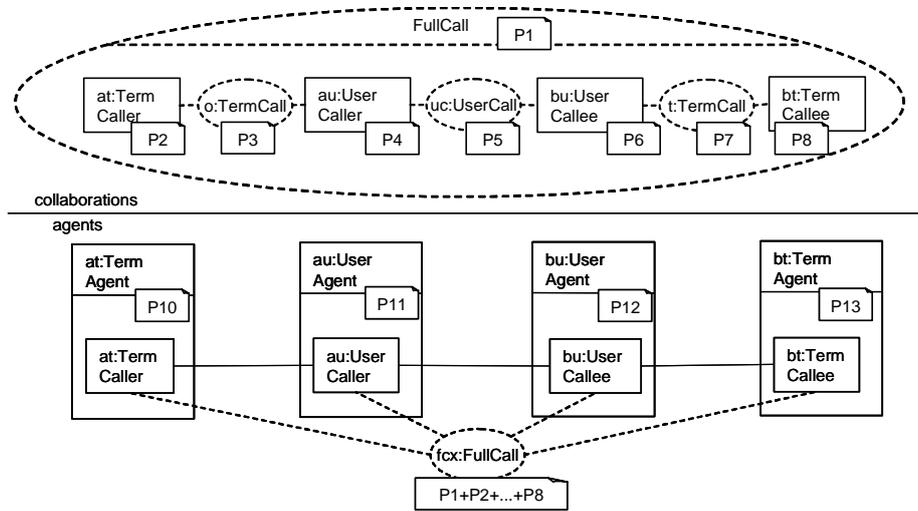


Figure 12.4: Role Binding

triggered:

- features that are triggered within a role as part of its behaviour (e.g. call-transfer)
- features that are triggered upon role-binding depending on the agent's context and policy (e.g. call-forward on busy subscriber)

We refer to the first type of features as *mid-role-triggered* (or just *mid-role*), and to the second as *role-binding-triggered*. Mid-role-triggered features are selected as part of the role behaviour. If they may be disabled by policy decisions this should be agreed during a negotiation phase before the role is bound. Alternatively, the role may consult its containing agent concerning actual policies before invoking mid-role features. One example of such a feature is call-forward on no-answer. It describes an alternative behavior to the `UserCall` service and it is triggered when the latter does not achieve its goal (i.e. contacting the end-user).

Role-binding-triggered feature selection occurs when an agent receives a role request. In that case the agent (1) checks its feature selection policies to determine if, in the current context, there is an alternative feature to be selected, without even trying to invoke the requested role. This may be the case if the following feature selection policy existed: “when the `b` role is requested in a `UserCall` and the (called) user is not at home, always select call-forward instead”. This checking returns either the requested role (if no feature selection policy was satisfied) or an alternative one. The selected role is then target (2) for checking the collaboration and role binding policies to decide whether it may be actually invoked. If yes, a confirmation message is sent back to the requesting role. Note that if the role that is finally selected to be invoked is not the originally requested one, the confirmation message may be

replaced by a negotiation phase (not shown in Fig. 12.3). If otherwise collaboration and/or role binding policies are not satisfied, (3) a search is again performed for a substitute role that may be invoked, and, if found, the process is repeated from (2), until a role with specific features is agreed and invoked. In addition, if an invoked role does not achieve its goal during the service execution, a search for an alternative role, implementing a mid-role-triggered feature, can be made once more (e.g. to invoke call-forward on no-answer).

From the above explanation three generic events can be distinguished that trigger the selection of features describing alternative behavior. These events are:

- *OnRoleRequest*,
- *OnUnsuccessfulRoleBinding*, and
- *OnNonAchievedGoal* event.

Feature selection policies can then be defined, by for example end-users, as event-condition-action (ECA) rules, where the event is one of the three just mentioned, the condition is expressed in terms of the context and the action is the selection of a feature.

Note that up to now we have just talked about the use of feature selection policies to select features of a base service. However their potential is actually greater than that. There is nothing that prevents us from using feature selection policies to specify any service as an alternative to another one. That is, we may specify which event and condition leads to the substitution of a role X for a role Y, where roles X and Y are not necessarily related. In this case, the role at the requesting side must most likely be also substituted. A negotiation between the parties would then be necessary.

The use of policies for service-execution management and personalization is not novel. For example, the Call Processing Language (CPL) [LWS04] is used to describe and control Internet telephony services. With CPL users can themselves specify their preferences for service execution. Reiff-Marganiec and Turner [RMT03] also propose the use of policies to enhance and control call-related features. The novelty of our work lies in the structuring of policies we make, based on the proposed service architecture.

## 12.4 Conclusion

We have presented a two-dimensional service oriented architecture where service components are roles defined using UML 2.0 collaborations and system components are agents representing domain entities such as users and terminals. Service modularity is achieved by the separation of service components from system components, and by general policy-driven mechanisms for dynamic role binding that handle context dependency, personalisation, resource limitations and compatibility validation. Central parts of this architecture, such as the role request pattern and a simple form of XML-based role-binding policy, have been implemented in ServiceFrame

[BHM02] and have been used to develop numerous demonstrator services within the Program for Advanced Telecommunication Services (PATS) research program [PAT], which is a cooperation between the Norwegian University of Science and Technology (NTNU), Ericsson, Telenor and Compaq (now Hewlett-Packard). These experiments have confirmed that dynamic role binding is central not only to traditional telecom services, but also to a wide range of convergent services, and that explicit support for role-binding helps to manage the complexity of such services. The use of more advanced role-binding policies specified as BeanShell [Nie97] scripts has also been studied in [Stø03]. At the time of writing this paper, ServiceFrame has been extended with support for java-based role-binding, collaboration and feature selection policies that can be specified by both end-users and service providers to handle context dependency [Pha05].

An interesting problem that has not been treated is undesirable interactions between two or more roles simultaneously played by an agent in different services. This is known as the feature interaction problem. We believe that our policy-driven mechanisms for dynamic role binding can help to avoid such interactions, if the agent maintains the consistency between the policies imposed in different services. We are also investigating in this direction.

## Acknowledgements

We wish to thank NTNU and the ARTS project for partial funding this work. G. Melby and K.E. Husa at Ericsson have been extremely valuable through numerous discussions and by implementing the core of the architecture. We are also grateful to the many MSc students that have contributed by making services using these principles and by proposing and prototyping role binding mechanisms. Finally we appreciate all the input from our close colleagues J. Floch, R. T. Sanders, F. Krämer and H. Samset.

## References

- [AKGM00] Magdi Amer, Ahmed Karmouch, Tom Gray, and Serge Mankovski. Feature-interaction resolution using fuzzy policies. In *Feature Interactions in Telecommunications and Software Systems VI*, pages 94–112, Glasgow, Scotland, UK, 2000.
- [BCS00] F. Bordeleau, J. P. Corriveau, and B. Selic. A scenario-based approach to hierarchical state machine design. In *ISORC '00: Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 78. IEEE Computer Society, 2000.
- [BF04] Rolv Bræk and Jacqueline Floch. ICT convergence: Modeling issues. In Daniel Amyot and Alan W. Williams, editors, *System Analysis and Modeling (SAM), 4th International SDL and MSC Workshop*, volume 3319 of *LNCS*, pages 237–256, Ottawa, Canada, 2004. Springer.

- [BH93] Rolv Bræk and Øystein Haugen. *Engineering Real Time Systems. An object-oriented methodology using SDL*. Prentice Hall, 1993.
- [BHM02] Rolv Bræk, Knut Eilif Husa, and Geir Melby. Service-Frame: WhitePaper. White paper, Ericsson Norarc, 2002. <http://www.pats.no/devzone/platforms/ServiceFrame/doc/ServiceFrameWhitepaperv8.pdf>.
- [Bræ99] Rolv Bræk. Using roles with types and objects for service development. In *IFIP TC6 WG6.7 Fifth International Conference on Intelligence in Networks (SMARTNET)*, volume 160 of *IFIP Conference Proceedings*, pages 265–278, Pathumthani, Thailand, 1999. Kluwer.
- [Cas05] Humberto N. Castejón. Synthesizing state-machine behaviour from UML collaborations and use case maps. In Andreas Prinz, Rick Reed, and Jeanne Reed, editors, *Proc. 12th SDL Forum*, volume 3530 of *LNCS*, pages 339–359, Grimstad, Norway, June 2005. Springer.
- [Flo03] Jacqueline Floch. *Towards Plug-and-Play Services: Design and Validation using Roles*. PhD thesis, Department of Telematics, Norwegian Univ. Science and Technology, Trondheim, Norway, February 2003.
- [KGM<sup>+</sup>04] Ingolf H. Krüger, Diwaker Gupta, Reena Mathew, Praveen Moorthy, Walter Phillips, Sabine Rittmann, and Jaswinder Ahluwalia. Towards a process and tool-chain for service-oriented automotive software engineering. In *Proceedings of the ICSE 2004 Workshop on Software Engineering for Automotive Systems (SEAS)*, 2004.
- [LS99] Emil C. Lupu and Morris Sloman. Conflicts in policy-based distributed systems management. *IEEE Trans. Softw. Eng.*, 25(6):852–869, 1999.
- [LWS04] J. Lennox, X. Wu, and H. Schulzrinne. Call processing language (CPL): A language for user control of internet telephony services. RFC 3880, IETF, October 2004.
- [Nie97] P. Niemeyer. Beanshell - lightweight scripting for java. <http://www.beanshell.org/>, 1997.
- [OMG05] Object Management Group (OMG). *UML 2.0 Superstructure Spec.*, July 2005.
- [PAT] Program for Advanced Telecom Services (PATS). <http://www.pats.no>.
- [Pha05] Quoc Tuan Pham. Policy-based service personalization. Master’s thesis, Dept. of Telematics, Norwegian University of Science and Technology (NTNU), 2005.
- [RMT03] Stephan Reiff-Marganiec and Kenneth J. Turner. A policy architecture for enhancing and controlling features. In *Feature Interactions in*

*Telecommunications and Software Systems VII*, pages 239–246, Ottawa, Canada, 2003.

- [SBvBA05] Richard Torbjørn Sanders, Rolv Bræk, Gregor von Bochmann, and Daniel Amyot. Service discovery and component reuse with semantic interfaces. In *Proc. 12th SDL Forum*, volume 3530 of *LNCS*, Grimstad, Norway, June 2005. Springer.
- [SCKB05] Richard Torbjørn Sanders, Humberto N. Castejón, Frank Alexander Kraemer, and Rolv Bræk. Using UML 2.0 collaborations for compositional service specification. In Lionel Briand and Clay Williams, editors, *ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 3713 of *LNCS*, pages 460–475, Montego Bay, Jamaica, October 2005. Springer-Verlag.
- [Stø03] Alf Kristian Støyle. Flexible user agent. Technical report, Dept. of Telematics, Norwegian University of Science and Technology (NTNU), 2003.
- [Zav03] Pamela Zave. Feature disambiguation. In *Feature Interactions in Telecommunications and Software Systems VII*, pages 3–9, Ottawa, Canada, 2003.
- [ZWO<sup>+</sup>95] Israel Zibman, Carl Woolf, Peter O’Reilly, Larry Strickland, David Willis, and John Visser. Minimizing feature interactions: An architecture and processing model approach. In *Feature Interactions in Telecommunications III*, pages 65–83, Kyoto, Japan, 1995.



**Part III**  
**Appendices**



## Synthesis of Role State Machines

In Paper 1 we presented an algorithm for the synthesis of role state machines from a Use Case Maps-based choreography. The state machines generated were flat and interruption composition was not supported. Here we present a new algorithm for the synthesis of role state machines from activity diagram-based choreography graphs. The five types of composition discussed in Section 2.2, namely sequential, parallel, alternative, invocation and interruption composition, are supported. The algorithm generates hierarchical state machines.

The formal syntax and semantics of choreography graphs were presented in Paper 6. To help with the understanding of the synthesis algorithm we reproduce here their syntax definition.

**Definition A.1 (Choreography).** A choreography of the sub-collaborations of a collaboration  $C$  is a directed graph defined by the tuple  $CH = (V, \mathcal{E}, \mathcal{R}_{\text{int}}, \searrow_{\text{ch}}, g_e, m_{\text{exp-a}}, R_{\text{CH}}, AC, m_{\text{a-ac}}, m_{\text{p-a}}, pin, l_{\text{pred}}, p_{\text{type}})$  where

- a)  $V$  is a set of nodes. It is partitioned into an initial node ( $v_0$ ) and sub-sets of activities ( $V_A$ ), input pins ( $V_{\text{InP}}$ ), output pins ( $V_{\text{OutP}}$ ), control flow nodes ( $V_{\text{FLOW}}$ ), accept event actions ( $V_{\text{EA}}$ ) and final nodes ( $V_{\text{FI}}$ ). In turn,  $V_{\text{FLOW}}$  is partitioned into decision ( $V_D$ ), merge ( $V_M$ ), fork ( $V_F$ ) and join ( $V_J$ ) nodes.
- b)  $\mathcal{E} \subseteq (V_{\text{OutP}} \cup V_{\text{FLOW}} \cup V_{\text{EA}} \cup \{v_0\}) \times (V_{\text{InP}} \cup V_{\text{FI}} \cup V_{\text{EA}} \cup V_{\text{FLOW}})$  is a set of directed edges between nodes, which is partitioned into normal edges ( $\mathcal{E}_n$ ) and interrupting edges ( $\mathcal{E}_{\text{int}}$ ).
- c)  $\mathcal{R}_{\text{int}}$  is a set of interruptible regions (i.e. regions containing nodes that can be interrupted).
- d)  $\searrow_{\text{ch}} \subseteq \mathcal{R}_{\text{int}} \times (\mathcal{R}_{\text{int}} \cup V)$  is a hierarchy relation among interruptible regions and nodes. We write  $reg \searrow x$  if  $x$  is a node or an interruptible region that is directly contained by the interruptible region  $reg$ .
- e)  $g_e$  is a guard function for edges. It is defined from  $\mathcal{E}_n$  into boolean expressions.
- f)  $R_{\text{CH}} = \{(id, type) : type \in R_C\}$  is a set of role instances, with  $R_C$  being the set of roles of collaboration  $C$ .
- g)  $AC$  is a set of *active collaborations*, that is, a collaboration-use representing a specific occurrence of one of  $C$ 's sub-collaborations. For each  $(id, type, B) \in$

$AC$ ,  $id$  is the name of the collaboration-use;  $type$  is the name of the collaboration that actually describes the collaboration-use (i.e. one of  $C$ 's sub-collaborations); and  $B \subseteq R_{type} \times R_{CH}$  is a set of role bindings, where  $R_{type}$  is the set of roles of the sub-collaboration named  $type$ .

- h)  $m_{a-ac} : V_A \rightarrow AC$  is a non-injective function that for a given activity returns its associated active collaboration.
- i)  $m_{p-a} : V_{InP} \cup V_{OutP} \rightarrow V_A$  is a function mapping input and output pins to activities, and  $pin : V_A \rightarrow \mathcal{P}(V_{InP} \cup V_{OutP})$  is a function that returns the set of pins attached to a given activity.
- j)  $l_{pred} : V_{InP} \cup V_{OutP} \rightarrow Pre$  is an injective function labeling each input and output pin of an activity with a state predicate of the activity's active collaboration.
- k)  $p_{type} : V_{InP} \cup V_{OutP} \rightarrow \{START, END, INVOCATION, RESUMPTION\}$  is a function that classifies pins as either *starting*, *end-of-execution*, *invocation* or *resumption* ones.

We require proper nesting of fork and join nodes in the choreography graph. That is, all outgoing edges of a fork node should eventually lead to the same join node, and all incoming edges of a join node should come from the same fork node. We also assume that invocation, resumption and end pins are always labeled with a predicate.

The state machines generated by the proposed algorithm are hierarchical finite state machines, which we define as follows.

**Definition A.2.** A hierarchical finite state machine is a tuple  $HFSM = (Q, \mathcal{R}, \searrow, T, G, A, \delta, q_0, J)$ , where

- $Q$  is a finite set of states, which is partitioned into *simple* states ( $Q_{sim}$ ), *composite* states ( $Q_{comp}$ ), *junction* pseudostates ( $Q_{jun}$ ), *fork* pseudostates ( $Q_{fork}$ ), *join* pseudostates ( $Q_{join}$ ), *initial* pseudostates ( $Q_{init}$ ) and *final* pseudostates ( $Q_{end}$ )
- $\mathcal{R}$  is a set of regions
- $\searrow \subseteq (Q_{comp} \times \mathcal{R}) \cup (\mathcal{R} \times Q)$  is a hierarchy relation among regions and states. We write  $q_{comp} \searrow reg$  if  $reg$  is a region of the composite state  $q_{comp}$ , and  $reg \searrow q$  if the state  $q$  is directly contained by the region  $reg$ .
- $T$  is a set of event triggers (e.g. message receptions)
- $G$  is a set of guard conditions (i.e. boolean expressions)
- $A$  is a set of actions, such as the sending of a message or a variable assignment
- $\delta \subseteq Q \times T \times G \times A \times Q$  is a set of transitions between states. A transition is fired when its triggering event is observed and its guard condition (if any) evaluates as *true*. After firing, the associated action (if any) is executed. If a

triggering event is not specified, the transition fires when the guard condition is *true*. If neither a triggering event nor a guard are specified, the transition may fire spontaneously.

- $q_0 \in Q$  is the so-called *initial* state, but not necessarily a UML initial pseudo-state
- $J \subseteq Q$  is a set of *connecting* states, which are special purpose states used to composed together state machines

In certain cases it is useful to obtain the *parent* of a region or state, that is, the state or region containing them. For this purpose we define the function *parent* :  $Q \cup \mathcal{R} \rightarrow Q_{\text{comp}} \cup \mathcal{R}$ , such that

$$\text{parent}(x) = \begin{cases} \text{reg} \in \mathcal{R}, & \text{if } x \in Q \wedge \text{reg} \searrow x \\ q \in Q_{\text{comp}}, & \text{if } x \in \mathcal{R} \wedge q \searrow x \\ \text{null}, & \text{if } x \in Q_{\text{top}} \end{cases}$$

In the definition of the *parent* function,  $Q_{\text{top}}$  is the set of all top-level states, where a top-level state is defined as follows.

**Definition A.3.** A *top-level state*  $q$  is a state that is not contained by any region (i.e.  $\nexists \text{reg} \in \mathcal{R}$ , such that  $\text{reg} \searrow q$ ).

In the following we present and explain the synthesis algorithm. We discuss first the synthesis of state machines from UML sequence diagrams, before addressing the synthesis from choreography graphs.

## A.1 Synthesis from UML 2 Sequence Diagrams

Basic sequence diagrams can be composed to obtain more complex behaviors. In UML 2 this is possible by means of interaction operators. We consider four operators: **seq** (for weak sequential composition), **alt** (for alternative composition), **par** (for parallel composition) and **loop** (for iterative composition). The weak sequential composition of two sequence diagrams consists in their lifeline-by-lifeline concatenation, such that for each instance, the events of the first diagram precede the events of the second diagram. Events on different lifelines are interleaved. In the parallel composition of two sequence diagrams their events are interleaved. The alternative composition of two sequence diagrams describes a choice between them, such that in any run of the system events will be ordered according to only one of the diagrams. That is, alternative composition introduces alternative orderings of events. The semantics of an alternative composition of basic sequence diagrams is therefore defined by a set of posets. The iterative composition of a sequence diagram can be seen as the weak sequential composition of a number of instances of that sequence diagram.

The syntax of a composite sequence diagram (SD) is defined by the following BNF-grammar:

$$SD \stackrel{\text{def}}{=} bSD \mid (SD_1 \text{ seq } SD_2) \mid (SD_1 \text{ alt } SD_2) \mid (SD_1 \text{ par } SD_2) \mid \text{loop}(\text{min}, \text{max}) SD_1$$

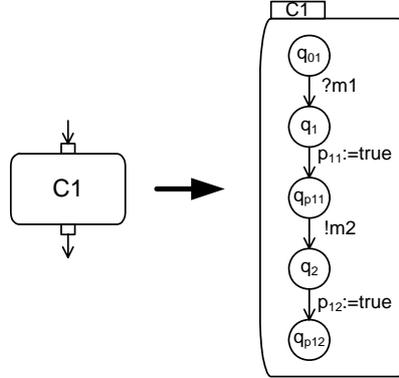


Figure A.1: Mapping of an activity into an HFSM

The *bSDToHFSM* procedure (on page 298) generates a state machine for a given role  $p$  from a basic sequence diagram *bSD*. The synthesis of the state machine is based on the projection of the sequence diagram events on the lifeline of the role. The result of the projection is a totally ordered set of events  $\pi(bSD, p) = (E, <)$ , where  $loc(e) = p$  for each  $e \in E$ , and  $<$  is a restriction of the visual order of *bSD* to the events in  $E$ . The set  $E$  is partitioned as usual into sending events ( $S$ ), receiving events ( $R$ ) and predicate events ( $\Phi$ ). Instead of working with the total order, the procedure obtains its linearization.

**Definition A.4.** A *linearization* of a poset  $(E, <)$  is a word  $w = e_1 \cdots e_{|E|}$  over the alphabet  $E$ , such that if  $e_i < e_j$  then  $i < j$ . We let  $w_i$  denote the  $i^{th}$  element of  $w$ , and  $|w|$  denote the size of  $w$  (i.e. its number of elements).

If the role does not participate in the sequence diagram, the so-called *empty* state machine  $SM_\emptyset = (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$  is returned. Otherwise a flat state machine is returned that consists of a sequence of states. The transitions between those states are labeled with either an event trigger corresponding to the reception of a message, an action describing the sending of a message, or a local action where a boolean predicate is set to *true*.

The *SDToHFSM* procedure (on page 297) generates a state machine for a given role  $p$  from a composite sequence diagram *SD*. This state machine consists of a composite state with a single region that contains the state machine-based behavior of  $p$  in *SD* (see Fig. A.1). The composite state is given as *id* the the *id* of the activity  $v$  whose behavior is described by *SD*. This way we can later on easily retrieve the state machine corresponding to the behavior of  $p$  in an activity. The *hfsm* function is used to obtain the actual state machine for role  $p$  from the sequence diagram (line 6). This is a recursive function defined as follows:

$$h fsm(SD, p) = \begin{cases} bSDToHFSM(SD, p), & \text{if } SD := bSD \\ h fsm(SD_1, p) \cdot h fsm(SD_2, p), & \text{if } SD := SD_1 seq SD_2 \\ h fsm(SD_1, p) \cup h fsm(SD_2, p), & \text{if } SD := SD_1 alt SD_2 \\ parToHFSM(SD_1, SD_2, p), & \text{if } SD := SD_1 par SD_2 \\ h fsm(SD_1, p)^*, & \text{if } SD := loop(0, m) SD_1, m > 0 \\ h fsm(SD_1, p) \cdot h fsm(SD_1, p)^*, & \text{if } SD := loop(n, m) SD_1, 1 \leq n \leq m \end{cases}$$

The *h fsm* function uses three operators to compose state machines: the concatenation operator ( $\cdot$ ), for sequential composition; the union operator ( $\cup$ ), for alternative composition; and the Kleene closure operator ( $*$ ), for iteration. These operators are similar to those presented in [ZHJ04]. The *h fsm* function also uses the *parToHFSM* procedure to compose state machines in parallel, and the *bSDToHFSM* procedure to obtain the state machine of basic sequence diagrams. We define the composition operators and explain the *parToHFSM* procedure in the following. Before that, let us explain the mapping into state machines of sequence diagrams describing loops. For this mapping, we abstract away from the actual number of iterations specified by the loop. If the minimum number of iterations is zero, a state machine is generated where the behavior described by the sequence diagram may be executed zero or more times (line 5 in the definition of *h fsm*). Otherwise, if the minimum number of iterations is greater than zero, a state machine is generated where the behavior described by the sequence diagram will be executed at least once (line 6 in the definition of *h fsm*).

Let  $SM_1 = (Q_1, \mathcal{R}_1, \searrow_1, T_1, G_1, A_1, \delta_1, q_{01}, J_1)$  and  $SM_2 = (Q_2, \mathcal{R}_2, \searrow_2, T_2, G_2, A_2, \delta_2, q_{02}, J_2)$  be two hierarchical finite state machines with disjoint sets of states. We define the composition operators used by the *h fsm* function as follows.

**Concatenation.** For the concatenation of two state machines  $SM_1$  and  $SM_2$ , we proceed as follows. If the initial state of  $SM_2$  is a fork pseudostate, we just add a silent transition from the connecting states of  $SM_1$  (i.e.  $J_1$ ) to the fork state. Otherwise, for each output transition with label  $(e, g, a)$  from the initial state of  $SM_2$ , new transitions are added with the same label  $(e, g, a)$  from the connecting states of  $SM_1$  to the successors of the initial state of  $SM_2$ . Then, if the initial state of  $SM_2$  is not a connecting state, it is removed, together with its output transitions (see Fig. A.2(a)). Otherwise, if the initial state of  $SM_2$  is a connecting state, nothing else is done (see Fig. A.2(b)). More formally, the concatenation  $SM_1 \cdot SM_2$  of two state machines  $SM_1$  and  $SM_2$  is a state machine  $SM$  such that:

- $SM = SM_1$ , if  $SM_2 = SM_\emptyset$
- $SM = SM_2$ , if  $SM_1 = SM_\emptyset$
- Otherwise,  $SM = (Q, \mathcal{R}_1 \cup \mathcal{R}_2, \searrow_1 \cup \searrow_2, T_1 \cup T_2, G_1 \cup G_2, A_1 \cup A_2, \delta, q_{01}, J_2)$ , where
  - If  $q_{02}$  is a fork pseudostate,
  - $Q = Q_1 \cup Q_2, \delta = \delta_1 \cup \delta_2 \cup \{(q_1, \emptyset, true, \emptyset, q_{02}) : q_1 \in J_1\}$

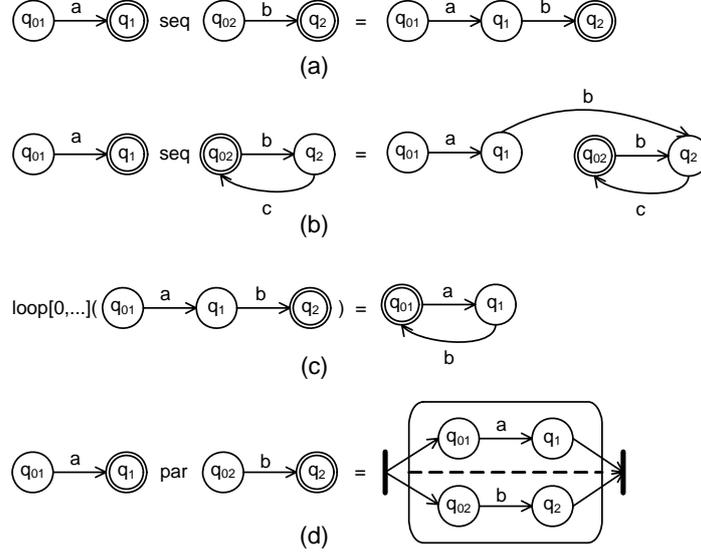


Figure A.2: Composition of state machines

– Otherwise,

$$Q = \begin{cases} Q_1 \cup Q_2, & \text{if } q_{02} \in J_2 \\ Q_1 \cup Q_2 - \{q_{02}\}, & \text{otherwise} \end{cases}$$

$$\delta = \delta_1 \cup (\delta_2 \cap (Q \times T_2 \times G_2 \times A_2 \times Q)) \cup \{(q_1, e, g, a, q_2) \in J_1 \times T_2 \times G_2 \times A_2 \times Q_2 : (q_{02}, e, g, a, q_2) \in \delta_2\}$$

**Union.** For the union of two state machines, a new initial state is created and concatenated with each of the state machines. More formally, the union  $SM_1 \cup SM_2$  of two state machines  $SM_1$  and  $SM_2$  is a state machine  $SM$ , such that:

- $SM = SM_1$ , if  $SM_2 = SM_\emptyset$
- $SM = SM_2$ , if  $SM_1 = SM_\emptyset$
- Otherwise, we let  $SM_0 = (\{q_0\}, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, q_0, \{q_0\})$  and proceed as follows to get  $SM$ 
  - a) Concatenate  $SM_0$  and  $SM_1$ :  $SM' = SM_0 \cdot SM_1$
  - b) Set the connecting states of  $SM'$  to  $\{q_0\}$  (i.e.  $J' = \{q_0\}$ ) and concatenate the new  $SM'$  with  $SM_2$ :  $SM'' = SM' \cdot SM_2$
  - c) Set the connecting states of  $SM''$  to  $J_1 \cup J_2$  (i.e.  $J'' = J_1 \cup J_2$ ) to obtain  $SM$

**Kleene closure.** For the Kleene closure of a state machine, the initial state is also made a connecting state. If one of the original connecting state is a join pseudostate (i.e. it belongs to  $Q_{\text{join}}$ ), a silent transition is added from the join state to the initial state. Otherwise, the original connecting states are removed, and their input transitions are connected to the initial state (see Fig. A.2(c)). More formally, the Kleene closure  $SM_1^*$  of a state machine  $SM_1$  is a state machine  $SM$  such that:

- $SM = SM_1$ , if  $SM_1 = SM_0$
- Otherwise,  $SM = (Q, \mathcal{R}_1, \searrow_1, T_1, G_1, A_1, \delta, q_{01}, \{q_{01}\})$ , where
 
$$Q = Q_1 - \{q \in J_1 : q \notin Q_{\text{join}}\} \cup \{q_{01}\}$$

$$\delta = \delta_1 \cup \{(q, e, g, a, q_{01}) : (q, e, g, a, q_j) \in \delta_1, q_j \in J_1, q_j \notin Q_{\text{join}}\} \cup \{(q_1, \emptyset, true, \emptyset, q_{01}) : q_1 \in J_1 \cap Q_{\text{join}}\}$$

The *parToHFSM* procedure (on page 298) generates a hierarchical state machine HFSM. For each of the input sequence diagrams, it obtains a state machine for role  $p$  (line 8) and copies the states, transitions and other elements of the state machine into the final HFSM (line 13). It also creates a composite state with two orthogonal regions and places each of the obtained state machines in one of the regions. In addition, it creates a fork pseudostate and adds silent transitions from it to the initial states of the state machines. It also creates a join pseudostate and adds silent transitions from the connecting states of each state machine to the join pseudostate (see Fig. A.2(d)).

---

**Procedure SDToHFSM( $v, SD, p$ )**


---

- 1  $q_v \leftarrow newCompositeState(v)$
  - 2  $Q \leftarrow \{q_v\}; \mathcal{R} \leftarrow \emptyset; \searrow \leftarrow \emptyset; T \leftarrow \emptyset; G \leftarrow \{true\}; A \leftarrow \emptyset; \delta \leftarrow \emptyset; J \leftarrow \emptyset$
  - 3  $reg \leftarrow newRegion()$
  - 4  $\mathcal{R} \leftarrow \mathcal{R} \cup \{reg\}$
  - 5  $\searrow \leftarrow \{(q_v, reg)\}$  // Add region to composite state
  - 6  $SM \leftarrow hfsm(SD, p)$
  - 7 **foreach** top-level state  $q$  of SM **do**
  - 8    $\lfloor \searrow \leftarrow \searrow \cup \{(reg, q)\}$  // Add state to region
  - 9 Update  $Q, \mathcal{R}, \searrow, T, G, A, \delta, J$  with elements from  $SM$
  - 10 **return**  $(Q, \mathcal{R}, \searrow, T, G, A, \delta, SM.q_0, J)$
-

---

**Procedure** bSDToHFSM( $bSD, p$ )
 

---

```

1  $w \leftarrow \text{linearization}(\pi(bSD, p))$ 
2 if  $|w| = 0$  then
   | /* p does not participate in bSD */
3   | return  $SM_\emptyset$ 
4  $q_0 \leftarrow \text{newState}()$ 
5  $\text{currState} \leftarrow q_0$ 
6  $Q \leftarrow \{q_0\}; T \leftarrow \emptyset; G \leftarrow \emptyset; A \leftarrow \emptyset; \delta \leftarrow \emptyset$ 
7 foreach  $i \in \{1 \dots |w|\}$  do
8   |  $q_{w_i} \leftarrow \text{newState}(w_i); Q \leftarrow Q \cup \{q_{w_i}\}$ 
9   | if  $w_i \in R$  then
10  |   |  $T \leftarrow T \cup \{?m : \text{lbl}(w_i) = \langle ?m, p_1, p_2 \rangle\}$ 
11  |   |  $\delta \leftarrow \delta \cup \{(currState, ?m, true, \emptyset, q_{w_i}) : \text{lbl}(w_i) = \langle ?m, p_1, p_2 \rangle\}$ 
12  | else if  $w_i \in S$  then
13  |   |  $A \leftarrow A \cup \{!m : \text{lbl}(w_i) = \langle !m, p_1, p_2 \rangle\}$ 
14  |   |  $\delta \leftarrow \delta \cup \{(currState, \emptyset, true, !m, q_{w_i}) : \text{lbl}(w_i) = \langle !m, p_1, p_2 \rangle\}$ 
15  | else
16  |   |  $act \leftarrow \text{"pred} := true\text{"}, \text{ where } \text{lbl}(w_i) = pred$ 
17  |   |  $A \leftarrow A \cup \{act\}$ 
18  |   |  $\delta \leftarrow \delta \cup \{(currState, \emptyset, true, act, q_{w_i})\}$ 
19  |   |  $\text{currState} \leftarrow q_{w_i}$ 
20 return  $(Q, \emptyset, \emptyset, T, G, A, \delta, q_0, \{currState\})$ 

```

---



---

**Procedure** parToHFSM( $SD_1, SD_2, p$ )
 

---

```

1  $\text{initForkState} \leftarrow \text{newForkPseudoState}()$ 
2  $\text{finalJoinState} \leftarrow \text{newJoinPseudoState}()$ 
3  $\text{enclosingState} \leftarrow \text{newCompositeState}()$ 
4  $Q \leftarrow \{\text{initForkState}, \text{finalJoinState}, \text{enclosingState}\}; \mathcal{R} \leftarrow \emptyset; \searrow \leftarrow \emptyset; T \leftarrow \emptyset; G \leftarrow \{true\};$ 
   |  $A \leftarrow \emptyset; \delta \leftarrow \emptyset; J \leftarrow \emptyset$ 
5  $\text{reg}_1 \leftarrow \text{newRegion}(); \text{reg}_2 \leftarrow \text{newRegion}()$ 
6  $\mathcal{R} \leftarrow \mathcal{R} \cup \{\text{reg}_1, \text{reg}_2\}$ 
7  $\searrow \leftarrow \{(\text{enclosingState}, \text{reg}_1), (\text{enclosingState}, \text{reg}_2)\}$  // Add regions to composite state
8  $SM_1 \leftarrow \text{hfsm}(SD_1, p); SM_2 \leftarrow \text{hfsm}(SD_2, p)$ 
9 foreach top-level state  $q$  of  $SM_1$  do
10  |  $\searrow \leftarrow \searrow \cup \{(\text{reg}_1, q)\}$  // Add state to region
11 foreach top-level state  $q$  of  $SM_2$  do
12  |  $\searrow \leftarrow \searrow \cup \{(\text{reg}_2, q)\}$ 
13 Update  $Q, \mathcal{R}, \searrow, T, G, A, \delta$  with elements from  $SM_1$  and  $SM_2$ 
14  $\delta \leftarrow \delta \cup \{(\text{initForkState}, \emptyset, true, \emptyset, SM_1.q_0), (\text{initForkState}, \emptyset, true, \emptyset, SM_2.q_0)\}$ 
15  $\delta \leftarrow \delta \cup \{(q, \emptyset, true, \emptyset, \text{finalJoinState}) : q \in SM_1.J\} \cup \{(q, \emptyset, true, \emptyset, \text{finalJoinState}) : q \in SM_2.J\}$ 
16 return  $(Q, \mathcal{R}, \searrow, T, G, A, \delta, \text{initForkState}, \{\text{finalJoinState}\})$ 

```

---

## A.2 Synthesis from Choreography Graphs

The *GetRoleStateMachine* algorithm (on page 303) is used to generate a hierarchical state machine HFSM for a role  $p$  from a choreography graph. Before discussing the details of that algorithm, let us explain the *ChoreographyToHFSM* procedure (on page 304). This procedure traverses the choreography graph using a depth-first search (DFS) [AHU74] technique and maps the visited nodes into elements of an HFSM. Normal edges are used to navigate from one node of the choreography to another node (i.e. interrupting edges are not traversed). A global variable *currState* is used to identify the state of the HFSM to which new states should be concatenated. When an activity is visited, the *SDToHFSM* procedure is invoked to obtain the state machine corresponding to that activity's sequence diagram (line 3), which is merged into the HFSM being created (line 4). A silent transition is then added from *currState* to the initial state of the activity's state machine (line 5). In case the activity has invocation pins, the *ProcessInvocation* procedure (to be explained later) is invoked to deal with a possible invocation composition. Once this procedure has finished, one of the successors  $u$  of the activity is visited (line 22). The *currState* variable is then updated to point to the state named with the predicate that labels the end-of-execution pin of the just processed activity that leads to  $u$  25. Merge and final nodes in the choreography are mapped into junction and final pseudostates in the HFSM (lines 7-11). Decision nodes in the choreography are not mapped into any pseudostate in the HFSM. Instead, the presence of a decision node will cause *currState* (at the time of visiting the decision node) to be the same for all the successor nodes of the decision node. When a fork node is visited, the *ForkToHFSM* procedure (to be explained later) is invoked (line 18) to obtain a state machine corresponding to the behavior of role  $p$  in all the nodes contained between the fork and its companion join node. That state machine is merged into the HFSM being created (line 19) and a silent transition is added from *currState* to the initial state of the state machine (line 20). The traversal of the graph continues then from the fork's companion join. The global variable  $v_{\text{join}}$  points to that join node<sup>1</sup>. This variable is updated when a join node is visited, which should happen during the execution of the *ForkToHFSM* procedure, as we will see.

The *ForkToHFSM* procedure (on page 305) follows the same principles as the *parToHFSM* procedure described in the previous section. The *parToHFSM* procedure uses the *h fsm* function to obtain the state machines to be composed in parallel. Instead, the *ForkToHFSM* procedure repeatedly invokes the *ChoreographyToHFSM* procedure to obtain state machines for each of each branches (lines 5-14). Figure A.3 illustrates the mapping performed by the *ForkToHFSM* procedure.

The *ProcessInvocation* procedure (on page 306) is used to deal with invocation compositions (see e.g. Fig. A.4(a) and Fig. A.4(b)). A given activity may invoke several other activities, which in turn may invoke other activities, and so on. Imagine, for example, that an activity  $v_1$  invokes two other activities,  $v_2$  and  $v_3$ , and that  $v_2$  invokes in turn an activity  $v_4$ . The procedure would first process the invocation  $v_1 \text{ inv } v_2$ . Then, by means of a recursive call, it would process the invocation

---

<sup>1</sup>This is true as long as forks and join nodes are properly nested.

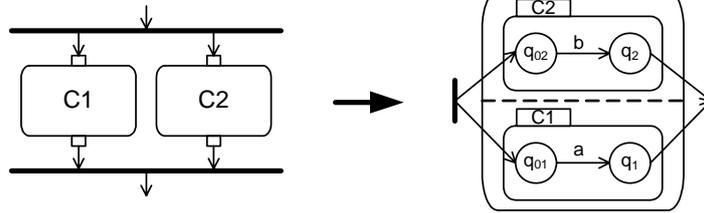


Figure A.3: Illustration of mapping of fork-join pairs into an HFSM

$v_2 \text{ inv } v_4$ . And after returning from the recursive call, it would process the invocation  $v_1 \text{ inv } v_3$ . For a given activity  $v$ , the procedure obtains first the set of activities that  $v$  invokes, and removes from that set the activity stored in the *alreadyInvoked* variable (initially empty) (lines 1-2). For example, in Fig. A.4(a) the set of invoked activities for  $v$  would be  $\{u\}$ . In this figure, both activities invoke each other, and this mutual invocation is handled while processing  $v$ . That is, the invocation of  $v$  from  $u$  is already processed while handling the invocations of  $v$ . Therefore, when *ProcessInvocation* is recursively invoked (line 26) to process the invocations of  $u$ , *alreadyInvoked* is set to  $v$ . Given one of the invoked activities,  $u$ , the procedure obtains the state machine for that activity and, if it is not the empty state machine, merges it with the main HFSM being constructed (lines 4 and 6). Now the procedure handles the invocation and resumptions by adding silent transitions between appropriate states of the two state machines of  $v$  and  $u$ . We will explain this process with help of the invocation composition in Fig. A.4(b). In that figure  $v$  invokes  $u$  at a point in its execution where predicate  $p_{11}$  holds, and gets suspended. Activity  $u$  executes and returns the control to  $v$ , which is resumed at the same point where it was suspended. Invocations are performed by means of invocation pins, which are labeled with predicates that should hold at the time of the invocation. These predicates are represented by states in the state machines (in the following called “predicate states” or “invocation states”). For example, in Fig. A.4(b),  $v$  invokes  $u$  by means of an invocation pin labeled with predicate  $p_{11}$ . This predicate is also labeling one of the states of  $v$ ’s state machine in Fig. A.4(c). For each invocation pin there is a companion resumption pin, and both of them are labeled with the same predicate (see Fig. A.4(b)). This means that resumption should happen at the same invocation state in the state machine at which the invocation took place. In practice, a new state is created for resumption (called “resumption state”) in the following way. For a predicate state  $q_{pred}$ , a new state  $q_{pred-res}$  is created. Then, for each transition  $tr$  with label  $l$  from  $q_{pred}$  to a state  $q$ , a transition from  $q_{pred-res}$  to  $q$  with label  $l$  is added, and the original transition  $tr$  is removed (see  $v$ ’s state machine in Fig. A.4(d)). For each invocation pin  $invPin$  of  $v$ , the *ProcessInvocation* procedure proceeds as follows. It first obtains the pin  $peerPin_1$  of  $u$  that is connected to  $invPin$  (line 10). The  $peerPin_1$  pin might be a start pin or a resumption pin. In the former case, a silent transition is added from the appropriate invocation state of  $v$ ’s state machine to the initial state of  $u$ ’s state machine (line 12), as shown in

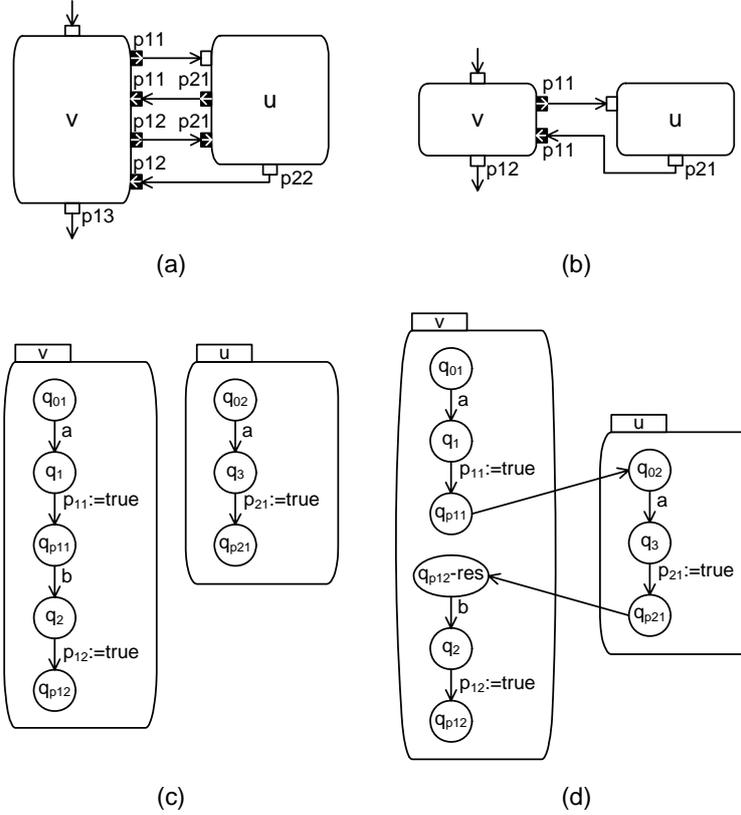


Figure A.4: (a) Mutual invocation; (b) Simple invocation; (c) State machines for each of the activities in (b); (d) State machine resulting from the invocation composition in (b)

Fig. A.4(d). If  $peerPin_1$  is a resumption pin, an appropriate resumption state is created in  $u$ 's state machine and a silent transition is added from the appropriate invocation state of  $v$ 's state machine to the newly created resumption state in  $u$ 's state machine (lines 14-18). After that, the procedure creates a resumption state in  $v$ 's state machine (line 19-21) and obtains  $resPin$ , the companion resumption pin of  $invPin$  (line 22). It then obtains the pin  $peerPin_2$  of  $u$  that is connected to  $resPin$  (line 23). The  $peerPin_2$  pin might be an invocation pin or an end pin. In both cases, the pins are associated to appropriate predicate states in the state machine. A silent transition is then added from the appropriate predicate state of  $u$ 's state machine to the appropriate resumption state of  $v$ 's state machine (see Fig. A.4(d)).

Now that we have explained the procedures used (directly or indirectly) by the *GetRoleStateMachine* algorithm, we can explain its operation. It creates first an HFSM describing the so-called nominal behavior of  $p$  in the choreography (i.e. all the behavior except the behavior executed due to interruptions). This is done by

invoking the *ChoreographyToHFSM* procedure from the initial node of the choreography (lines 1-4). After that, the behaviors executed when an interruption occurs are added. The whole process is illustrated in Fig. A.5. The process of adding the interrupting behaviors is as follows. First, a new composite state is created and added to the final HFSM for each interruptible region in the choreography (lines 5-7). Then, for each accept event action  $ea$  in the choreography<sup>2</sup>, the algorithm gets the node  $v_{\text{int}}$  connected to  $ea$  via an interrupting edge (line 9). That node corresponds to the beginning of the interrupting behavior (e.g. activity  $C_3$  in Fig. A.5(a)). The *ChoreographyToHFSM* procedure is then invoked from  $v_{\text{int}}$  to obtain a state machine for the interrupting behavior (line 10), which is merged with the final HFSM (line 17). If the *ChoreographyToHFSM* procedure returns a non-empty set of *finalStates*, the interrupting behavior is inside a fork-join pair, and should be placed inside the appropriate orthogonal region of the composite state corresponding to that fork-join pair. To do that the *getRegionForInterruptionBehavior* procedure is used (line 13). This procedure is not detailed here. Instead we explain its main principles. Using the  $v_{\text{join}}$  global variable (updated inside *ChoreographyToHFSM*) we can retrieve the join pseudostate associated to the composite state we are looking for, and, thereafter, we can retrieve the regions of the composite state itself (following the transitions connected to the join pseudostate). Then we need to identify one region that contains a composite state that corresponds to an activity contained by the interruptible region containing the accept event action we are dealing with (line 12). At this point we would have a situation as the one depicted in Fig. A.5(b). We still need to encapsulate the behavior that can be interrupted (i.e. the behavior corresponding to the activities inside the interruptible region) into the composite state that was created for the interruptible region (lines 18-21), and add a transition from that composite state to the beginning of the interrupting behavior (lines 22-26). The state machine for the choreography in Fig. A.5(a) is shown in Fig. A.5(c).

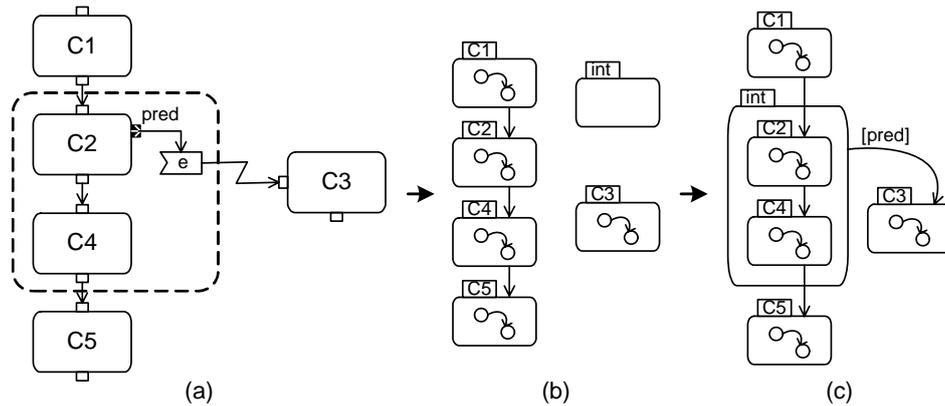


Figure A.5: (a) Interruption composition; (b) Partial mapping of (a) into an HFSM; (c) Complete mapping of (a) into an HFSM

<sup>2</sup>Accept event actions are used to trigger interruptions

**Algorithm 33:** GetRoleStateMachine

---

```

/* Obtain first the HFSM = (Q, R, \, T, G, A, δ, q0, J) for the "nominal" behavior
*/
1 forall v ∈ V do visited[v] = false
2 (HFSM, finalStates) ← ChoreographyToHFSM(v0, p, SMθ)
3 qinit ← newInitialPseudoState()
4 Q ← Q ∪ {qinit}; δ ← δ ∪ {(qinit, θ, true, θ, HFSM.q0)}
/* Now we obtain and add the "interrupting" behaviors */
5 foreach interruptible region intReg ∈ Rint do
6   qintReg ← newCompositeState(intReg)
7   Q ← Q ∪ {qintReg}
8 foreach accept event action ea ∈ VEA do
/* Get the node at the end of the interrupting edge */
9   vint ← mp-a(startPin), such that (ea, startPin) ∈ Eint */
10  (SMvint, finalStates) ← ChoreographyToHFSM(vint, p, SMθ)
11  if finalStates ≠ ∅ then
/* The interrupting behavior ended up at a join node, so it should be
placed inside a region of a composite state. vjoin is a global
variable updated inside ChoreographyToHFSM that points to the last
visited join node. qvjoin is the state corresponding to that node */
12  intReg ← interruptible region containing ea (i.e. such that intReg \ch ea)
/* Obtain the region of the HFSM where the interrupting behavior
should be placed */
13  reg ← getRegionForInterruptingBehavior(HFSM, SMvint, qvjoin, intReg) */
14  foreach top-level state q of SMvint do
15  | \ ← \ ∪ {(reg, q)} // Add state to region
16  | δ ← δ ∪ {(q, θ, true, θ, qvjoin) : q ∈ finalStates}
17  Copy elements of SMvint to HFSM
/* Place the behavior that can be interrupted inside one of the
composite states previously created */
18  reg ← newRegion(); R ← R ∪ {reg}; \ ← \ ∪ {(qintReg, reg)}
19  Vi ← {x : intReg \ch x}
20  foreach x ∈ Vi do
21  | \ ← \ ∪ {(reg, qx)}
/* Add a transition from composite state to first state of the
interrupting behavior */
22  if ∃(invPin, ea) ∈ En, for any invPin ∈ VOutP (i.e. ea has no incoming edge) then */
23  | δ ← δ ∪ {(qintReg, θ, true, θ, SMvint.q0)}
24  else
25  | pred ← lpred(invPin), such that (invPin, ea) ∈ En
26  | δ ← δ ∪ {(qintReg, θ, pred, θ, SMvint.q0)}

```

---

---

**Procedure** ChoreographyToHFSM( $v, p, HFSM$ )
 

---

```

1 visited[v] ← true
2 if v is an activity node where p participates then
   | /* Let SDv be the sequence diagram describing v's behavior */
3   | SMv ← SDToHFSM(v, SDv, p)
4   | Copy elements of SMv to HFSM
5   | δ ← δ ∪ {(currState, 0, true, 0, SMv.q0)}
6   | if v has invocation pins then HFSM ← ProcessInvocation(HFSM, SMv, v, p, 0)
7 else if v is a merge node then
8   | qv ← newJunctionPseudoState(v); Q ← Q ∪ {qv}; δ ← δ ∪ {(currState, 0, true, 0, qv)}
9   | currState ← qv
10 else if v is a final node then
11   | qv ← newFinalPseudoState(v); Q ← Q ∪ {qv}; δ ← δ ∪ {(currState, 0, true, 0, qv)}
12 else if v is a join node then
13   | vjoin ← v; visited[v] ← false
14   | return (HFSM, {currState})
15 else if v is a fork node then
16   | vjoin ← null
17   | currStateold ← currState
18   | SMv ← ForkToHFSM(v, p)
19   | Copy elements of SMv to HFSM
20   | δ ← δ ∪ {(currStateold, 0, true, 0, SMv.q0)}
   | /* We continue traversing the graph from the fork's companion join node
   | (i.e. vjoin). Note that we assume proper nesting of fork/join nodes */
21   | v ← vjoin
22 foreach u successor of v do
23   | if !visited[u] then
24     | if v is an activity then
25       | currState ← qpr where lpred(ep) = pr and ep is the end-of-execution pin of v
26       | that leads to u
27     | (HFSM, finalStatesaux) ← ChoreographyToHFSM(u, p, HFSM)
28     | finalStates ← finalStates ∪ finalStatesaux
29   | else if u is a merge node then
   | /* A silent transition is added from currState to the state
   | corresponding to the previously visited merge node */
30   | δ ← δ ∪ {(currState, 0, true, 0, qu)}
31 if v is NOT a merge node then
   | /* Merge nodes cannot be revisited. They are mapped into a state just
   | once. See also line 30 */
32   | visited[v] ← false
33 return (HFSM, finalStates)

```

---

---

**Procedure ForkToHFSM( $v_{fork}, p$ )**


---

```

1  $q_{v_{fork}\text{-init}} \leftarrow newForkPseudoState(v_{fork}\text{-init})$ 
2  $q_{v_{fork}} \leftarrow newCompositeState(v_{fork})$ 
3  $Q \leftarrow \{q_{v_{fork}\text{-init}}, q_{v_{fork}}\}; \mathcal{R} \leftarrow \emptyset; \searrow \leftarrow \emptyset; T \leftarrow \emptyset; G \leftarrow \{true\}; A \leftarrow \emptyset; \delta \leftarrow \emptyset; J \leftarrow \emptyset$ 
4  $F \leftarrow \emptyset$ 
5 foreach successor  $v$  of  $v_{fork}$  do
    /* An HFSM is built for each branch of the fork and placed in a separate
    orthogonal region */
6    $reg \leftarrow newRegion()$ 
7    $\mathcal{R} \leftarrow \mathcal{R} \cup \{reg\}$ 
8    $\searrow \leftarrow \{(q_{v_{fork}}, reg)\}$  // Add region to composite state
9    $(SM_v, finalStates) \leftarrow ChoreographyToHFSM(v, p)$ 
10  foreach top-level state  $q$  of  $SM_v$  do
11     $\searrow \leftarrow \searrow \cup \{(reg, q)\}$  // Add state to region
12  Update  $Q, \mathcal{R}, \searrow, T, G, A, \delta$  with elements from  $SM_v$ 
13   $\delta \leftarrow \delta \cup \{(q_{v_{fork}\text{-init}}, \emptyset, true, \emptyset, SM_v.q_0)\}$ 
14   $F \leftarrow F \cup finalStates$ 
    /*  $v_{join}$  is a global variable updated inside ChoreographyToHFSM */
15  $q_{v_{join}} \leftarrow newJoinPseudoState(v_{join})$ 
16  $Q \leftarrow Q \cup \{q_{v_{join}}\}$ 
17  $\delta \leftarrow \delta \cup \{(q, \emptyset, true, \emptyset, q_{v_{join}}) : q \in F\}$ 
18  $currState \leftarrow q_{v_{join}}$ 
19 return  $(Q, \mathcal{R}, \searrow, T, G, A, \delta, q_{v_{fork}\text{-init}}, \{currState\})$ 

```

---

---

**Procedure ProcessInvocation**(*HFSM, SM<sub>v</sub>, v, p, alreadyInvoked*)
 

---

```

1 InvokedActs ← {mp-a(x) : (inv, x) ∈  $\mathcal{E}$ , inv ∈ pin(v), ptype(inv) = INVOCATION, x ∈ VA}
2 InvokedActs ← InvokedActs − {alreadyInvoked}
3 foreach u ∈ InvokedActs do
4   SMu ← h fsm(SDu, p)
5   if SMu ≠ SM0 then
6     Copy elements of SMu to HFSM
7     INVv-u = {inv ∈ pin(v) : ptype(inv) = INVOCATION, (inv, x) ∈  $\mathcal{E}$ , mp-a(x) = u}
8     foreach invPin ∈ INVv-u do
9       /* lblv is the labeling function of the seq. diag. of v */
10      ev ← lblv−1(lpred(invPin))
11      /* Get pin on u connected to v's invocation pin */
12      peerPin1 ← x, such that (invPin, x) ∈  $\mathcal{E}$ 
13      if ptype(peerPin1) = START then
14        |  $\delta$  ←  $\delta \cup \{(q_{e_v}, \emptyset, true, \emptyset, SM_u.q_0)\}$ 
15      else
16        /* peerPin1 is a resumption pin on activity u */
17        eu ← lblu−1(lpred(peerPin1))
18        qeu-res ← newState(eu-res); Q ← Q ∪ {qeu-res} // Create resumption
19        state
20         $\delta$  ←  $\delta \cup \{(q_{e_v}, \emptyset, true, \emptyset, q_{e_u-res})\}$ 
21         $\delta$  ←  $\delta \cup \{(q_{e_u-res}, e, g, a, q) : (q_{e_u}, e, g, a, q) \in SM_u.\delta\}$ 
22         $\delta$  ←  $\delta - \{(q_{e_u}, e, g, a, q) \in SM_u.\delta\}$ 
23        // Create a resumption state on v's state machine
24        qev-res ← newState(ev-res); Q ← Q ∪ {qev-res}
25         $\delta$  ←  $\delta \cup \{(q_{e_v-res}, e, g, a, q) : (q_{e_v}, e, g, a, q) \in SM_v.\delta\}$ 
26         $\delta$  ←  $\delta - \{(q_{e_v}, e, g, a, q) \in SM_v.\delta\}$ 
27        /* Get resumption pin on v with same predicate as invPin */
28        resPin ← res ∈ pin(v) : ptype(res) = RESUMPTION ∧ lpred(res) = lpred(invPin)
29        /* Get pin on u connected to v's resumption pin */
30        peerPin2 ← x, such that (x, resPin) ∈  $\mathcal{E}$ 
31        eu ← lblu−1(lpred(peerPin2))
32         $\delta$  ←  $\delta \cup \{(q_{e_u}, \emptyset, true, \emptyset, q_{e_v-res})\}$ 
33      ProcessInvocation(HFSM, SMu, u, p, v)
34   return

```

---