Distributed System Design with Message Sequence Charts

Ingolf Heiko Krüger

Institut für Informatik
der Technischen Universität München

Distributed System Design with Message Sequence Charts

Ingolf Heiko Krüger

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Manfred Paul

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Manfred Broy
2. Univ.-Prof. Dr. Martin Wirsing
   Ludwig-Maximilians-Universität
   München

Die Dissertation wurde am 28. März 2000 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 20. Juli 2000 angenommen.

Abstract

The methodical mastery of interaction scenarios is a key factor for capturing and modeling system requirements of distributed, reactive systems. Message Sequence Charts (MSCs) and variants thereof are well-accepted as a graphical description technique for interaction scenarios. MSCs emphasize the inter-component coordination aspect of typically partial system executions; this complements the usually complete behavior description for individual components, as given by state-oriented automaton specifications.

The topic of this thesis is the seamless, methodically founded integration of MSCs into the development process for distributed, reactive systems.

The comparison of several MSC dialects and automaton models is followed by the definition and analysis of the formal syntax and semantics for the MSC notation used in this thesis. The stream-based system model, underlying the semantics definition, enables the integrated consideration of interaction-oriented and state-oriented system specifications; it also serves as the basis for the introduction of effective refinement notions for MSCs. Next, different MSC interpretations – in the range from scenario specification to complete behavior descriptions to the specification of unwanted behavior – are formally defined. In addition, the application of MSCs for the description of safety and liveness properties is analyzed.

Finally, two transformation procedures, supporting the transition from interaction scenarios to complete behavior specifications for individual components, are presented. The first one schematically extracts relational assumption/commitment specifications from MSCs. The second one turns MSCs syntactically into corresponding state automata. On the one hand this makes the component properties defined by MSCs accessible to formal analysis; on the other hand this constructively bridges the gap between interaction requirements and component implementations.

Kurzfassung

Die methodische Beherrschung von Interaktionsszenarien ist ein Schlüsselfaktor bei der Erfassung und Modellierung von Systemanforderungen für verteilte, reaktive Systeme. Message Sequence Charts (MSCs) und Varianten davon haben sich als grafische Beschreibungstechnik für Interaktionsszenarien etabliert. MSCs betonen den komponentenübergreifenden Koordinationsaspekt eines typischerweise partiellen Systemablaufs; dies ergänzt die komponentenlokale, jedoch meist vollständige Verhaltensbeschreibung, wie sie durch zustandsorientierte Automatenspezifikationen gegeben ist.

Gegenstand der vorliegenden Arbeit ist die durchgängige, methodisch fundierte Integration von MSCs in den Entwicklungsprozeß für verteilte, reaktive Systeme.

Nach einer vergleichenden Betrachtung verschiedener MSC-Dialekte und Automatenmodelle wird die formale Syntax und Semantik der in dieser Arbeit verwendeten MSC-Notation definiert und analysiert. Das der Semantikdefinition zugrundeliegende, strombasierte Systemmodell ermöglicht die integrierte Betrachtung interaktions- und zustandsorientierter Systemspezifikationen und dient als Basis für die Einführung effektiver Verfeinerungsbegriffe für MSCs. Im Anschluß daran werden unterschiedliche MSC-Interpretationen – von der Szenarienspezifikation über die vollständige Verhaltensbeschreibung, bis hin zur Spezifikation von Fehlverhalten – formal definiert. Zusätzlich wird die Anwendung von MSCs zur Beschreibung von Sicherheits- und Lebendigkeitseigenschaften untersucht.

Um den Übergang von Interaktionsszenarien zu vollständigen Verhaltensspezifikationen für einzelne Komponenten methodisch zu unterstützen, werden schließlich zwei Transformationsverfahren angegeben. Das erste extrahiert relationale Assumption/Commitment-Spezifikationen auf schematische Weise aus MSCs. Das zweite wandelt MSCs syntaktisch in korrespondierende Zustandsautomaten um. Dadurch werden einerseits die durch ein MSC definierten Komponenteneigenschaften einer formalen Analyse zugänglich, andererseits wird die Lücke zwischen Interaktionsanforderungen und Komponentenimplementierungen konstruktiv überbrückt.

Acknowledgments

First and foremost my sincere thanks to Professor Manfred Broy, who has invited me to work in his research group, and has provided me with the opportunity of carrying out the research leading to this thesis. His encouragement, guidance, and advice during the selection of the topic, as well as during the work on it that followed, were very valuable; he also commented on and suggested enhancements of draft versions of this thesis. Moreover, I am very grateful to Manfred Broy for the stimulating and challenging research environment he has established. I consider it a rare privilege to have the chance of working together, and of exchanging ideas with so many excellent colleagues and friends within a single research group.

My thanks also go to Professor Martin Wirsing, who, when asked, immediately accepted to serve on my dissertation committee; he also gave very useful feedback on draft versions of this document. I enjoyed, and significantly profited from the various discussions we had on MSCs and related topics.

I am especially indebted to Katharina Spies. She patiently read almost the entire thesis during its various stages of maturity, and kindly provided valuable suggestions for improvement; moreover, she was a never-failing source of encouraging comments. Thank you, Katharina! Michael von der Beeck and Bernhard Schätz also read substantial parts of the thesis, discussed its contents with me, and proposed several changes; I am much obliged to them for this, as well as for their being ready to listen and respond to the many questions I had. Furthermore, I am grateful to Ekkart Rudolph for reading and commenting on the section about MSC-96; the insight into ITU's MSC standard he shared with me significantly helped me to improve my understanding of "the idea behind" MSC-96 and MSC 2000.

I am very grateful to Manfred Broy, Radu Grosu, and Thomas Stauner for the inspiring and challenging collaboration on topics in the context of MSCs. This joint work has helped me shape and validate many of the ideas I present in this thesis.

I experienced the SysLab project as an excellent research environment for applying the concepts discussed in this thesis within the framework of systematic, object-oriented development techniques. I thank Ruth Breu, Franz Huber, Bernhard Rumpe, and Wolfgang

Schwerin not only for interesting discussions, especially in the context of our joint work on the methodic foundation of the UML's description techniques, but also for the fun it was working with them.

Max Breitling was always willing to discuss topics related to my work, even if it meant diving deeply into the corresponding formulae; thank you for this, and also for being the great office mate you are. I am also grateful to Peter Braun, Heiko Lötzbeyer, Stephan Merz, Jan Philipps, Alex Pretschner, Chris Salzmann, Alex Schmidt, Oscar Slotosch, as well as to all the other members of our research group for the active exchange of ideas on work-related areas and beyond.

Within a cooperation project with Siemens ICN I had the challenging opportunity to transfer some of my ideas into industrial practice. I am most grateful to the members of Professor Cornelis Hoogendoorn's group at Siemens ICN for the excellent atmosphere within the project; in particular, many thanks are due to Cornelis Hoogendoorn, Heinz Koßmann, Axel Pink, and Kurt Stadler for our valuable discussions on the practical application of MSCs.

Despite my less than adequate responsiveness to their suggestions with respect to the other interesting, challenging, and fun sides of life, many friends and colleagues have provided inspiration, words of cheer, and pleasant diversions on many occasions. In addition to being terrific friends, Markus Kaltenbach and Bernd Finkbeiner even found the time and patience to discuss issues of semantics, logics, and automata with me. Thanks to you all for all your help and companionship!

My parents Eva and Winfried Krüger have always lovingly supported, challenged, and encouraged me in my studies and my goals in general. I am profoundly grateful for all you have done for me!

Finally, and most importantly, my very special thanks go to Stephanie Pittner for her love and patience, even across geographical distance. Whenever I needed it most, you were there to provide emotional support, encouragement, and motivation. Your contagious warm-heartedness and sense of humor have turned days of hard work into days of pleasure and joy. Thank you!

Contents

1. Introduction **1**

1.1. Message Sequence Charts: Ready for Seamless System Development? . . . 2

1.2. Background and Motivation 5

1.2.1. System Classes 5

1.2.2. The Challenge: Developing Distributed Software Components . . . 7

1.2.3. System Views and Description Techniques 8

1.2.4. Seamless System Development – What Do we Need for it? 11

1.3. Contributions and Outline of this Thesis 14

1.4. Roadmaps through this Thesis 16

1.5. Related Work 18

2. MSC Notations – Introduction and Comparison **23**

2.1. Introduction 24

2.2. MSC-96 25

2.2.1. Basic MSC Notation 26

2.2.2. MSC Composition and Structuring 32

2.2.3. Miscellanea 46

2.3. Other MSC Dialects 50
2.3.1. OMSCs 50
2.3.2. Sequence and Collaboration Diagrams 54
2.3.3. EETs 61
2.3.4. Interworkings 65
2.3.5. HySCs 70
2.3.6. LSCs 70
2.3.7. MSC 2000 72
2.4. Comparison and Prospective Enhancements 73
2.5. Related Work 77
2.6. Summary 78

3. State-Based Description Techniques for Component Behavior 81

3.1. Introduction 82
3.2. Automata in the Development Process 85
3.3. Overview of Automaton Models 88
3.3.1. Moore/Mealy-Automata 88
3.3.2. Statecharts, ROOMCharts 91
3.3.3. ω-Automata, I/O-Automata, “Spelling” Automata 99
3.4. Related Work 103
3.5. Summary 103

4. YAMS – Yet Another MSC Semantics 105

4.1. Introduction 106
4.2. System Model and Mathematical Preliminaries 108
4.2.1. Notational Conventions 108
4.2.2. System Structure 110
4.2.3. System Behavior 111
4.3. Abstract Textual Syntax 112

4.4. Denotational MSC-Semantics 115
4.5. Discussion of the Semantics 131
4.5.1. Well-Definedness 131
4.5.2. General Observations 132
4.5.3. The Relationship between Sequential Composition and Interleaving 133
4.5.4. MSCs versus Temporal Logic 135
4.5.5. Adequacy of the Syntax and its Semantics 136
4.6. HMSCs 139
4.7. Example: the Abracadabra-Protocol 146
4.7.1. Informal “Requirements Specification” 147
4.7.2. “Roadmap” for the Major Use Cases 148
4.7.3. Successful Communication 149
4.7.4. Conflict and Conflict Resolution 149
4.7.5. Adding Progress/Liveness 150
4.7.6. Adding Preemption 151
4.8. Related Work 152
4.9. Summary 152

5. MSC Refinement **155**
5.1. Introduction 156
5.2. Binding References 160
5.3. Property Refinement 161
5.3.1. Refinement Rules 162
5.3.2. Compositionality 166
5.4. Message Refinement 167
5.4.1. Refinement Rule 171
5.4.2. Problems With Message Refinement 172
5.5. Structural Refinement 176
5.5.1. Refinement Rule 183
5.5.2. Relationship With MSC-96’s Instance Decomposition 183
5.6. Related Work 184
5.7. Summary 184

6. MSCs for Property-Oriented System Specifications 187
6.1. Introduction 188
6.2. MSC Interpretations 193
6.2.1. Existential MSC Interpretation 194
6.2.2. Universal MSC Interpretation 196
6.2.3. Exact MSC Interpretation 197
6.2.4. Negation: Unwanted Behaviors 200
6.3. Property Specification with MSCs: Safety and Liveness 202
6.3.1. Safety and Liveness 202
6.3.2. MSC Properties 204
6.4. Related Work 211
6.5. Summary 212

7. From MSCs to Component Specifications 215
7.1. Introduction 216
7.2. Relational Component Specifications 221
7.2.1. Basic Definitions 221
7.2.2. Causality, Realizability, and Nondeterminism 223
7.2.3. Composition 224
7.2.4. Component Refinement 226
7.2.5. Component Properties, Safety and Liveness of Components 228
7.3. From MSCs to A/C-Specifications 230
7.3.1. A/C Specifications 231
7.3.2. MSCs and Interaction Interfaces 234
7.3.3. From MSCs to A/C Specifications 237
7.3.4. MSC Refinement Revisited 241
7.3.5. Discussion 244
7.4. From MSCs to Automaton Specifications 245
7.4.1. Automaton Syntax and Semantics 246

7.4.2. Translation Scheme 250
7.4.3. Example: the Abracadabra-Protocol 267
7.4.4. Extensions 270
7.4.5. Methodological Issues 280
7.5. Related Work 285
7.5.1. A/C Specifications, Automaton Models 285
7.5.2. Work on the Transformation of MSCs to Automaton Specifications 286
7.6. Summary 292

8. Summary and Outlook **295**
8.1. Summary 296
8.2. Outlook 298

A. Syntactic And Semantic Extensions **303**
A.1. Instance Start and Stop 303
A.2. Timers 304
A.3. Message Parameters and Parametric MSCs 305
A.4. Actions 309
A.5. Gates 310

B. Proofs **311**
B.1. Properties of the MSC Semantics 312
B.1.1. Independence of Absolute Time 312
B.1.2. Properties of the MSC Operators 313
B.1.3. Well-Definedness of the Semantics 321
B.1.4. Sequential Composition versus Interleaving 323
B.2. Property Refinement Rules 324
B.3. MSCs for Property-Oriented System Specifications 328
B.3.1. Exact MSC Interpretation 328
B.3.2. Safety and Liveness 329
B.4. From MSCs to Component Specifications 355
B.4.1. Time Guardedness Of MSCs 355
B.4.2. Join Consistency 356

Bibliography **359**

CHAPTER 1

Introduction

The topic of this thesis is the methodical usage of Message Sequence Charts (MSCs) in the development process for distributed, reactive systems. In this chapter we motivate our interest in MSCs *beyond* their traditional application domain, i.e. the specification of interaction scenarios. In particular, we emphasize the following issues as important prerequisites for a seamless integration of MSCs into the overall development process: a thorough understanding of the semantics of MSCs and the properties they express, the availability of effective refinement and abstraction techniques, and transformations from MSCs to individual component specifications. Furthermore, we list the major contributions of this thesis, give its outline, and mention related work.

Contents

1.1. Message Sequence Charts: Ready for Seamless System Development? . . . 2

1.2. Background and Motivation . . . 5

1.3. Contributions and Outline of this Thesis . . . 14

1.4. Roadmaps through this Thesis . . . 16

1.5. Related Work . . . 18

1.1. Message Sequence Charts: Ready for Seamless System Development?

Message Sequence Charts (MSCs, for short) and similar notations for component interaction have gained wide acceptance for scenario-based specifications of component behavior. Due to their intuitive notation MSCs have proven useful as a communication tool between customers and developers of distributed systems, thus helping to reduce misunderstandings in early development stages; their predominant use today is in the requirements capture phase of the software development process.

In this section we give a concise overview of what distinguishes MSCs from other description techniques, as well as of what their role in the development process currently is and what it could be. Moreover, we summarize the contributions of this thesis with respect to a seamless integration of MSCs into an overall software and system development process. For a more detailed discussion of the background and context of this work we refer the reader to Sections 1.2 through 1.5.

What are MSCs?

Figure 1.1 shows an example of an MSC. It depicts a certain section of the communication among the four components W, X, Y, and Z within an imaginary distributed system. In this figure, labeled axes represent components, whereas labeled directed arrows indicate message exchange from the source (at the arrow's tail) to the destination component (at the arrow's head). Time advances from the top to the bottom of the figure; this induces a temporal order on the depicted messages. Intuitively, Figure 1.1 captures a situation where Y and Z, in turn, send the message *subscribe* to X. Then, X receives message *update* from W. Subsequently, X sends message *notify* to Y and Z (in that order). Upon receipt of message *notify*, component Y sends message *request* to W, and receives message *reply* in return.

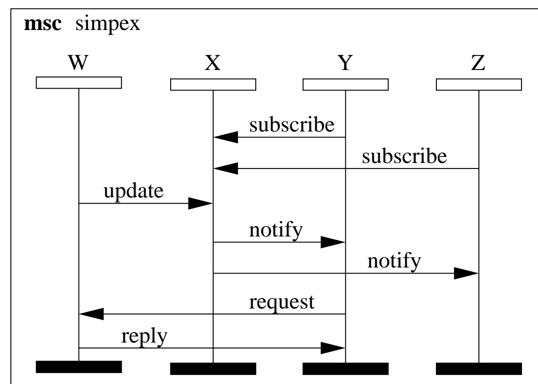


Figure 1.1.: Simple MSC

This simple example already allows us to illustrate one of the strengths of MSCs. They contain information on the distribution structure, as well as on the interaction behavior of the system under consideration. This combination helps make explicit the coordination aspect of system behavior, beyond the local scope of individual components. MSCs, such as the one in Figure 1.1, show one particular interaction pattern (or *scenario*) among the depicted components. In this sense, MSCs complement other forms of specification that capture the complete behavior of individual components.

The central contribution of this thesis is that we provide means for bridging the gap between scenarios of component coordination and the specification of individual component behavior. To put the methodical transition from scenarios to component specifications on solid, formal grounds we also contribute a thorough semantics definition, as well as effective refinement notions for MSCs; moreover, we give a detailed analysis of the properties expressed by MSCs throughout the development process.

The Role of MSCs in the Development Process

Over the past years several MSC-like description techniques have emerged (cf. Chapter 2). Despite their syntactic and semantic differences the common strength of all of these notations is the representation of component cooperation and coordination for achieving a certain goal in the system under consideration. The benefit added by MSCs to any portfolio of description techniques is that an MSC enables the visualization of only the *relevant (partial) behavior* of each depicted component with respect to the *specific* task or goal at hand. This is in contrast to other description techniques that stress *complete* behavior of *individual* components, such as automaton models (cf. Chapter 3).

In other words, MSCs represent projections of the complete system behavior on (part of) a certain task or service. The projection contains the relevant components together with their relevant behavior for the task under consideration. Instead of having to study the complete behavior descriptions of several individual components simultaneously to get an overview of what happens when the system executes the task, we can zoom in directly on the particularly interesting segment of each participating component's behavior by means of an appropriately chosen MSC.

Clearly, the representation of coordination and cooperation by means of interaction is of value within any development phase, be it analysis, specification, design, or implementation (cf. Section 1.2.4).

MSCs have traditionally been very popular as a means for documenting or illustrating interaction patterns or scenarios. In Chapter 6 we give a precise definition of the term "scenario"; for the remainder of this introduction it suffices to consider it as a synonym for a representation of a certain part of system behavior. Scenarios typically cover one (or a small number) of the possibly many different behaviors a system can display.

The usage of scenarios for capturing requirements has been suggested and studied extensively in the literature (cf. Section 1.5). Thus, MSCs – as a description technique for

representing interaction scenarios – can be advantageously applied during the analysis phase of the development process. During the design phase, scenarios sometimes serve as more detailed representations of the major services (or "use cases") of the system under consideration (cf., for instance, [JBR99]). Another common application for MSCs is the visualization of concrete system executions. Here, an MSC depicts the interactions among the system's components logged over a certain time interval during a run of the system. This can provide insight into the behavior of an already existing system. The same idea underlies the usage of MSCs for the validation of specifications or implementations. Test-case- and counter-example-representation by means of MSCs have also been proposed and studied in the literature (cf. Section 1.5).

What all of these applications of MSCs have in common is that they treat MSCs as representations of scenarios *only*. However, as we will see in Chapter 2, several MSC dialects offer means for composing scenarios, allowing elaborate specifications of system behavior. This is a first step towards a *constructive* application of MSCs in the development process. Here, the aim is not only to capture or document requirements and system traces by means of MSCs, but also to manipulate the captured requirements by refining them (to make the specification more and more specific), or even to use MSCs directly for the construction of corresponding component specifications and implementations.

Treating them as a "full-fledged" description technique from which there is even a direct transition to individual component specifications assigns a much more prominent role to MSCs, as compared to the one they have traditionally played.

Are MSCs Ready for Seamless System Development?

This new role of MSCs calls for their seamless integration into the overall software and system development process. Seamlessness, in this context, means that we aim at using MSCs – in addition to their traditional applications mentioned before – for any one of the following development tasks:

- *scenario elicitation*: the capturing of interaction requirements in the form of scenarios,
- *scenario composition* or *scenario completion*: the composition of different scenarios to transit from particular instances of behavior to complete behavior descriptions,
- *scenario refinement*: the adjustment of the level of detail of a scenario,
- *scenario transformation*: the derivation of other forms of specifications, in particular those for individual components, from scenarios.

Clearly, this requires a much more thorough understanding of MSCs than is needed if they only represent exemplary patterns of behavior. In this thesis we set out to establish a more seamless integration of MSCs into the development process by addressing the following four major topics:

- **Precise and Expressive MSC Syntax and Semantics:**
  Guided by existing and well-accepted notations (cf. Chapter 2) we define a precise syntax and semantics for the MSC dialect we use in this thesis. This dialect, which provides the required expressiveness for capturing *and* composing scenarios, allows us to use MSCs as a formal description technique throughout the development process.

- **MSC Refinement:**
  We provide effective abstraction and refinement mechanisms that allow us to switch easily between the levels of detail within a given specification.

- **MSCs for Exemplary and Complete System Behavior:**
  We precisely define various interpretations of MSCs; these interpretations enable the application of MSCs for the specification of exemplary scenarios, as well as for the description of complete component behavior.

- **MSC Transformation:**
  We define transformation procedures that allow us to *generate* component implementations from a specification of interaction requirements in the form of MSCs.

As a result, we will identify MSCs as an intuitive, sufficiently expressive, graphical description technique enabling us to capture, design and implement the interaction requirements of distributed systems.

The remainder of this introduction has the following structure. Section 1.2 contains a discussion of the context, as well as of further motivation of our work. We present a more detailed overview of the contributions, as well as an outline of this thesis in Section 1.3. Section 1.4 contains several roadmaps for reading this thesis; each roadmap corresponds to putting the reading focus on one of the four topics mentioned above. A brief discussion of related approaches appears in Section 1.5.

1.2. Background and Motivation

Mastery of the specification, design, and implementation of distributed systems is an important prerequisite for the development of by now ubiquitous products. These products range from technical systems (such as home appliances, cars, and entire production lines) to business applications (such as enterprise-wide supply-chain management systems, operating over the Internet). In the following paragraphs we briefly discuss the background and motivation behind our interest in the usage of MSCs in this context.

1.2.1. System Classes

To put our understanding of the target domain for the development techniques and methods we suggest on more solid grounds, we briefly define the system class we aim at. To

that end, we consider two major classifying coordinates: reactive versus transformational, and technical versus business systems. Our underlying assumption is that the systems we deal with are potentially *distributed*, i.e. composed of logically or physically separate components, as opposed to *monolithic* systems.

Once started, a *reactive system* operates continually as follows: if present, it accepts input from its environment, and produces (corresponding) output. Reactive systems never halt, although the output of these systems may well be always empty from a certain point in time onward. This system class includes the electronic control units (ECUs) in cars, as well as heart pacers, to name just two examples. *Transformational systems*, on the other hand, do not operate continually. Given a collection of input values they start execution, run for a certain (finite) amount of time, produce their result, and then stop execution. Examples of transformational systems are compilers, and database report generators. We consider transformational systems as special cases of reactive systems: from a certain time onward the system neither receives inputs, nor does it produce outputs (but keeps on "running").

The focus of *technical systems* is the control of technical processes; examples are the ECUs of a car, telecommunication switches, video cassette recorders (VCRs), and heart pacers. *Embedded systems*, a subclass of technical systems, have become of particular importance over the past ten years. An embedded system is a unit of hardware and software, connected to its environment via sensors and actuators; it controls technical processes in the environment, typically without immediate interaction with the user. As examples we mention digital clocks, cellular phones, washing machines, VCRs, and ECUs. Sometimes technical systems must process continuous instead of digital data, or must interact with their environment continuously, instead of at discrete time points. In this state of affairs, we speak of *hybrid systems*.

Flight and hotel reservation systems are classical examples of *business systems*. Here, the focus is on (mass) data storage and manipulation instead of on the control of technical processes.

In certain cases there is a significant overlap between technical and business systems. Telecommunication switches, for instance, often contain software and hardware for controlling the technical switching process, but also provide a billing functionality with the corresponding database accesses. In view of the increasing interconnections between individual systems we expect the boundaries between technical and business systems to vanish further.

In the remainder of this thesis we focus on the general class of distributed, reactive systems. This allows us, in particular, to target all kinds of systems in the range from technical to business systems with the techniques we propose. We do not focus on data transmission and manipulation as part of component interaction, although we also briefly discuss how to handle this aspect in our approach. Furthermore, we restrict ourselves to digital systems, and touch hybrid systems only briefly in our discussion of related work.

MSCs address both component distribution, and component communication; therefore, they are an interesting candidate for capturing and manipulating requirements of reactive systems.

1.2.2. The Challenge: Developing Distributed Software Components

Distributed computer systems have gained significant importance in our everyday lives over the past few decades. Today's upper class cars, for instance, contain networks of 60 to 80 electronic control units (ECUs) controlling safety-critical systems, such as motor management, airbags, and the anti blocking system (ABS), as well as comfort systems, like driver information systems, communication, radio, heating, and even seat control, to name just a few examples. Another – very prominent – distributed system of yet larger scale is the Internet with its millions of interconnected computers cooperating to run the TCP/IP protocol stack to enable applications such as email, and the World Wide Web (WWW). Even standard software on today's desktop computers displays a significant degree of functional or code distribution.

In each of these examples the crucial aspect, which enables the distributed systems to achieve and supply their functionality, is the cooperation of the individual system components: the ECUs in a car share signals, say, to prevent chair adjustments at high car speeds; the nodes of the Internet execute the relevant networking protocols to transport the application data.

Therefore, an important step of the development process for distributed systems is to describe *how* the components achieve their cooperation. In the case of the Internet, for instance, the cooperation of the different nodes of the network is fixed by the specification of the TCP/IP protocol. Each node follows that protocol to connect to, and exchange data with its partner nodes. More generally, the cooperation of the components in a distributed system is determined by the way the components communicate with one another, or, in other words, by the components' interfaces.

As a promising aid in the task of designing and implementing distributed systems *component oriented development* has become an important approach over the past few years. Here, the development focus is not only on providing the required functionality per component under development, and on encapsulating the inner workings of how this functionality is achieved. In addition, each component is equipped with an interface defining how the component's functionality can be accessed. Such an interface typically consists of two parts. The *syntactic* part defines the names of the services that the component provides, as well as the types of parameters it accepts when the services get invoked. The *semantic* part describes how the component reacts upon a service invocation. Together, the syntactic and the semantic part of a component's interface describe what communication protocol the environment must observe to cooperate with the component.

Middleware technologies such as CORBA, Java RMI, JavaBeans, and COM/DCOM provide the technical infrastructure for distributed component implementations. Each of these bases on a standard way of specifying the syntactic interface of the distributed components. Typically this consists of simple lists of service names and parameter types. The semantic component interface is, in general, only available through the executable implementation of the component.

For the development of distributed systems, however, the semantic component interfaces are essential if we aim at producing predictable and correct overall system behavior. Indeed, the design of semantic component interfaces – which involves capturing and refining the component's interaction requirements – so that all system components together can achieve the required functionality, is one of the decisive steps in distributed system development.

Academia and industry offer numerous approaches whose intention is to support the specification of semantic component interfaces; these approaches range from purely mathematical formalisms – invented with the intention of *proving* the correctness of systems with respect to their specifications – to purely pragmatic system descriptions in prose – without intention and hope for formal reasoning about the system under development.

Recently, *graphical description techniques* (of which MSCs are only one example) have emerged as a possible link between purely formal and purely pragmatic development methodologies; the promise here is that the graphical representation of system requirements and designs admits easier communication among developers, as well as better means for manipulation and reuse of requirements, specifications, and designs. A prerequisite for keeping this promise is, however, that the graphical description techniques, and the properties they allow the developer to represent are well-understood; otherwise the saying "a picture says more than a thousand words" can quickly turn into "a picture has more than a thousand different interpretations", which contradicts the intention of better communication and less erroneous specifications and designs right away.

In the remainder of this thesis we will study the use of MSCs as a graphical description technique for semantic component interfaces in distributed, reactive systems; this includes, in particular, a thorough definition and analysis of the semantics of MSCs and of the properties that MSCs express. The focus of MSC specifications is on the interaction among components. Clearly, this is only one of several important aspects of component-oriented development. In the following, we briefly discuss how MSCs complement other system views and description techniques. This gives a first impression on the role that MSCs play in an overall portfolio of specification techniques for distributed systems.

1.2.3. System Views and Description Techniques

The importance of software has increased dramatically over the past thirty years. Today, software plays the role of an enabling technology in virtually all product fields. Car manufacturers, for instance, can offer most of the advanced functionality of their models only because the software of the corresponding ECUs controls the technical processes "behind the scenes". Instead of tuning a car's mechanical or electric parts, the mechanic today plugs in a computer, and tunes the software of the corresponding ECU. Electronic commerce – to mention a second example – in the form we know it today was only possible because of the existence of software solutions connecting the virtual markets with every Internet-enabled device.

The gain in importance was accompanied with an increase of functional complexity, of size, as well as of interconnection. The software part of current telecommunication switches has several millions of code lines. The same holds for enterprise-wide supply chain management systems in the business domain. Even seemingly small embedded systems, such as cellular phones, contain a substantial amount of software for sending and receiving short messages and faxes, for a database of phone numbers, for games, and – of course – for handling the mobile-phone network-protocols.

As an approach to dealing with the complexity of software, in particular during the analysis, specification, and design phases of the development process (see also Section 1.2.4), *view-oriented* modeling techniques and notations have emerged; popular examples are the UML [Rat97, RJB99], Catalysis [DW98], ROOM [SGW94], SDL [EHS98], Syntropy [CD94], OMT [RBP+91], and the Booch method [Boo94]. Each of these proposes managing the complexity of software development by separating the two major modeling concerns: *system structure* and *system behavior*. For both, a variety of special-purpose textual and graphical description techniques, which highlight either the structural or the behavioral system aspects, has been developed. Each model, represented in one of the description techniques, conveys one particular view (sometimes also called a projection) on the overall system. If no confusion can arise we use the terms model and view interchangeably in the following.

Structure

Under the structural system aspects most of the mentioned modeling techniques and notations subsume the specification of the data structures on which the system operates, as well as the system's decomposition into (separate) components and their relations or connections. Typically, this includes the specification of syntactic component interfaces.

System structure diagrams (cf. Figure 1.2; labeled nodes represent distinct components, directed arrows represent uni-directional communication paths) model component distribution and interconnection.

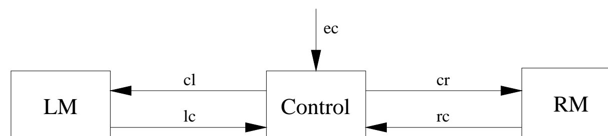


Figure 1.2.: System structure diagram

Common description techniques for data structures are entity/relationship- or class diagrams (cf. Figure 1.3; labeled nodes represent classes, lines represent associations between the classes).

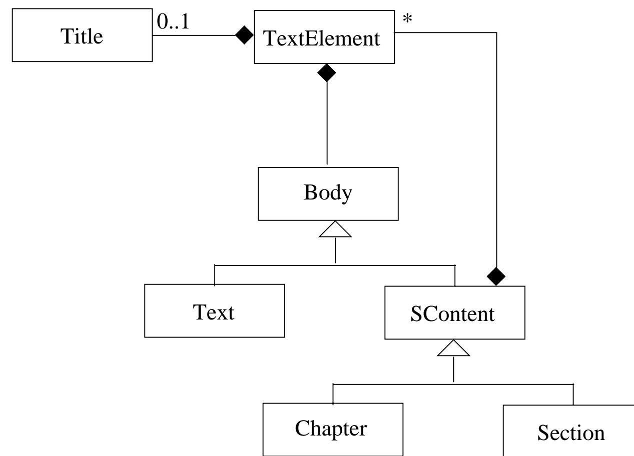


Figure 1.3.: Class diagram

Behavior

For the description of system behavior most development methods differentiate between the specification of the behavior of individual components, and the collaboration or coordination of multiple components.

The most prominent description techniques for individual component behavior in state-oriented systems are automaton models such as statecharts [Har87, HP98] (cf. Figure 1.4 (a)). They allow the developer to specify the relationship between state changes, their triggering events, and the actions taking place during a transition between two states.

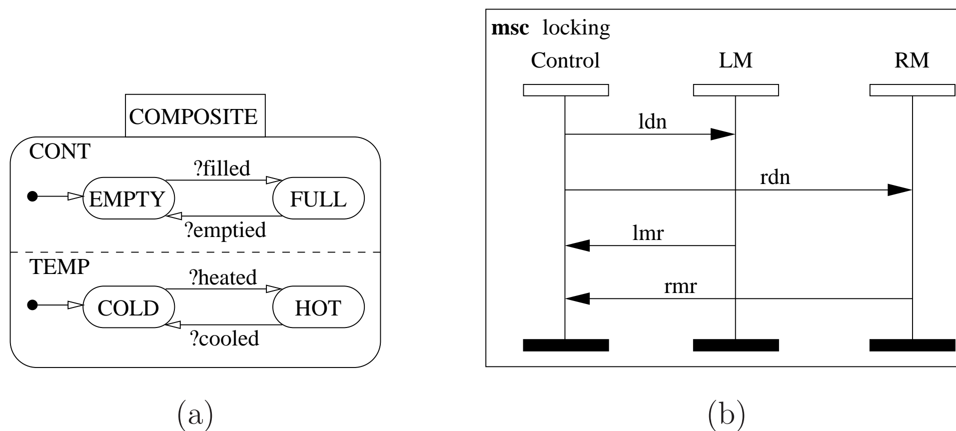


Figure 1.4.: Automaton (a) and Message Sequence Chart (b)

In recognition of the importance of the coordination aspect of distributed system behavior, the specification of interaction-oriented behavior specifications for multiple components has received an increasing amount of attention over the past decade. Graphical description techniques such as MSCs [IT96, IT98], event traces [Boo94, SHB96], and Sequence

Diagrams [Rat97, RJB99] have been developed to complement the local view on system behavior provided by automata (cf. Figure 1.4 (b)). MSCs and their relatives allow the developer to describe patterns of interaction among sets of components; these interaction patterns express how each individual component behavior integrates into the overall system to establish the desired functionality. Typically, one such pattern covers the interaction behavior for (part of) one particular service (or *scenario*) of the system.

Thus, automata and interaction-oriented description techniques span two different coordinates of the modeling task. Automata represent projections of the complete system behavior onto individual components, whereas MSCs represent projections of the complete system behavior onto particular (possibly partial) services.

In this thesis we focus on behavior specification. In particular, we study interaction-oriented collaboration specifications, and their relationship with state-oriented specifications for individual components in detail; we consider structural aspects only to the extent necessary for understanding interaction-oriented specifications.

Clearly, however, the separation of concerns induced by view-oriented development approaches is not without problems. Difficulties arising in the various phases of typical development processes, as well as steps towards solving these difficulties are the topic of the following section.

1.2.4. Seamless System Development – What Do we Need for it?

The definition of a precise semantics, of effective refinement and abstraction techniques, as well as of constructive transformations between models are important prerequisites for a seamless usage of graphical description techniques within the development process.

Software and system development processes typically consist of several phases, ranging from the capturing of requirements to the implementation and deployment of the final product. Only in rare cases can we expect such a development process to move linearly from requirements capture to implementation. More realistically, we will encounter *iterative* approaches in practice that visit some phases multiple times; sometimes entire phases are absent from concrete processes.

Figure 1.5 illustrates an iterative development process for the phases *analysis*, *specification*, *design*, and *implementation*. Under analysis we subsume everything related to capturing functional and nonfunctional requirements of the system under consideration. This can include, for instance, determining the major system components, as well as their major interaction patterns. The specification phase hosts the construction of a model for the system, such that the model meets the requirements captured during analysis. During the design phase the model becomes more specific. This can happen by means of refinement steps selecting one of a set of possible solutions for a certain problem. To name just one example for a design step, we mention the commitment to a specific form of communication

among system components. Implementation turns a model, obtained from the design phase, into an executable system.

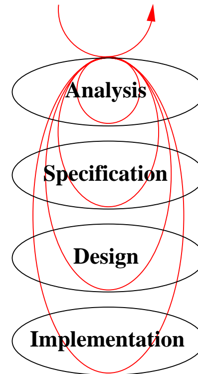


Figure 1.5.: Typical development process

There are many other, much more detailed presentations of development processes (cf., for instance, [JBR99, Kru99a, V97, SGW94, Boo94, Boe88, Roy87] and the references therein). However, for our purpose of illustrating the major difficulties arising in the combination of iterative, phase-oriented and view-oriented software development, the simple process of Figure 1.5 suffices.

The aim of software and system development along a process as sketched above is to obtain an implementation meeting the requirements captured during analysis. However, there are several obstacles in our way towards this goal. The first occurs right at the starting point of the process: how can we be sure of having captured all necessary requirements correctly? Once we have gathered the requirements, and are reasonably confident in their quality, how can we ensure that the models we construct during specification and design indeed meet the captured requirements?

If we adopt a view-oriented development approach we face further challenges. How can we establish the consistency of models representing different views? Determining the consistency of a service call (appearing in an automaton specification) with respect to a static service interface (as specified by a corresponding class diagram) is effectively manageable. The situation becomes more involved if we relate, say, interaction-oriented behavior specifications, which involve multiple components, with automaton specifications of individual components. How can we guarantee that each individual component can effectively participate in the interaction pattern under consideration?

In the following paragraphs we mention a few steps contributing to a seamless integration of description techniques into the development process.

Precise Semantics A helpful prerequisite for systematic view-oriented software development is a thorough understanding of the description techniques as such; every description

technique should have a precise, comprehensible meaning to avoid inconsistencies. In particular, an artifact should express unambiguous properties, constraints, or requirements during the entire development process. Only under this premise can we be sure to capture correct requirements with our view-oriented specifications.

The availability of a precise semantics is also crucial for subjecting models to verification technologies as provided by model checking and theorem provers. The purpose of applying these technologies could, for instance, be to *prove* the conformance of two different models (obtained either through different design steps, or from different system views) with respect to the same set of requirements.

Effective Refinement and Abstraction Techniques During the design phase we add more and more details to the model obtained from specification. In a view-oriented specification the system properties will be captured by means of multiple documents in a variety of description techniques. To increase the level of detail of one such property, we want to work with the description technique in which it is represented. Assume we have specified a certain system requirement by means of an MSC. To make this requirement more specific, we would welcome a set of effective refinement techniques for MSCs. This would allow us to adjust the level of detail without having to resort to other forms of behavior specification, such as automaton models. Similarly, the existence of adequate abstraction techniques can significantly increase the clarity and scalability of a specification.

Transformation between Models If we establish the consistency between different models *constructively*, we entirely avoid a particularly annoying discontinuity in the development process, viz. errors introduced by switching representations. Successful examples of this approach are compiler generators (which transform annotated grammars into parsers), as well as code generators in current development tools (which transform class diagrams and automaton models into prototypic source-code in the selected target programming-language).

As we have mentioned in Section 1.1, MSCs are particularly popular during the requirements capture and specification phase of the development process. Automata, on the other hand, are most popular during specification and design. Therefore, a "property preserving" transformation scheme from MSCs to automata could support development steps within and across development phases.

The combination of a thorough understanding of a description technique (based on a precise semantics), the existence of effective refinement and abstraction techniques, as well as constructive transformations from models in the description technique into other models significantly adds to the seamless integration of the description technique into the development process; this is accompanied by an increased traceability of requirements across the entire process, which opens the door for the application of quality assurance and validation technologies, such as verification and testing.

1.3. Contributions and Outline of this Thesis

Motivated by our observation that MSCs have the potential of supporting the specification and design of semantic component interfaces for reactive, distributed systems, we present the following contributions towards the seamless integration of MSCs into an overall software and system development process:

- we define and analyze a formal semantics for MSCs that
  - integrates the notions of interaction and state into a single mathematical framework,
  - enables the definition of multiple relevant MSC interpretations,
  - has increased expressiveness compared to the standard semantics of MSC-96;
- we define effective refinement notions for all aspects addressed by MSCs: system properties, messages, and component structure;
- we investigate the use of MSCs as scenarios and as a description technique for complete component behavior;
- we analyze, in detail, the properties we can specify with MSCs to obtain an understanding of the relationship between MSCs and other description techniques;
- we show how to transform MSCs into individual component specifications.

We develop these results along the following outline:

In Chapter 2 we give a thorough, yet informal introduction to the graphical syntax and semantics of several MSC-like description techniques. We compare these notations with respect to their underlying concepts, aims, expressiveness, and positioning within the development process. As a result we identify several deficits of existing MSC dialects; these deficits hinder the seamless usage of MSCs during system development. Most MSC dialects provide no means for combining overlapping interaction patterns, i.e. patterns showing the same components in different roles within the same overall sequence of interactions. Another problematic area is the specification of exceptional cases by means of MSCs; most MSC dialects lack proper notation and semantics for this. The most fundamental problem we observe, however, is the lack of methodical integration supplied with the notations: questions about how to refine MSC specifications, about how to interpret MSCs with respect to other system views, and about what properties MSCs express with respect to the system under consideration are rarely addressed in the MSC dialects we compare.

In Chapter 3 we consider Moore and Mealy Machines, statecharts and ROOMCharts, as well as ω- and I/O-automata as representatives of state-based description techniques for

reactive systems. This discussion highlights the role of automata as a means for specifying the complete behavior of individual components.

Chapter 4 contains a thorough foundation of the semantics of the MSC dialect used in this thesis. The semantics bases on a precise, mathematical system model for reactive systems, which allows us to integrate the notions of interaction and state in MSC specifications. With a few exceptions, the syntax of the chosen MSC dialect adheres to the ITU standard MSC-96 [IT96, IT98]. To solve the deficits identified in Chapter 2, we go beyond the standard syntax and semantics, and add several composition operators for MSCs. These operators allow us to deal systematically with overlapping and exceptional interaction patterns. Moreover, we can use them to add liveness and fairness constraints to MSC specifications.

The definition and investigation of effective refinement notions for MSCs is the topic of Chapter 5. Because MSCs address both behavioral and structural system aspects, we emphasize three separate forms of refinement: *property refinement*, *message refinement*, and *structural refinement*. Property refinement allows us to add detail to an MSC specification by reducing the set of behaviors represented by an MSC. This includes, for instance, removing alternatives and narrowing loop bounds. Message refinement allows us to adjust the level of detail of individual messages within MSCs. If needed, we can refine a single message into an entire protocol. Similarly, we can adjust the level of structural detail displayed in an MSC by means of structural refinement. This refinement notion allows us to make the hierarchical distribution structure of individual components explicit.

In Chapter 6 we take a closer look at what kinds of properties we can specify by means of MSCs. This adds to our understanding of the various roles played by MSCs throughout the development process. In a first step, we define and discuss four MSC interpretations with respect to the system under development: the *existential*, *universal*, *exact*, and *negative* interpretation. The existential interpretation forms the basis for using MSCs as scenario specifications: the depicted behavior can, but need not occur. An MSC under universal interpretation specifies behavior that must occur eventually in any system execution. The exact interpretation fixes the system's behavior to match precisely what the MSC specifies; this interpretation is the foundation for using MSCs as a description technique for complete component and system behavior. The specification of forbidden or undesirable behavior is the purpose of MSCs under the negative interpretation. In a second step, we focus on the distinction between safety and liveness properties in specifications for reactive systems. We analyze how safety and liveness properties propagate through MSC specifications. As a result we obtain that our MSC dialect specifies essentially liveness properties. Therefore, we can use these MSCs to complement safety-oriented specification techniques.

In Chapter 7 we develop another central result of our work. We present two transformation schemes for constructing individual component specifications from collections of MSCs. The first one yields *assumption/commitment* (A/C) specifications, which highlight the separate responsibilities of a component and its environment; we show how to extract A/C specifications schematically from an MSC. The other, more pragmatic one, produces state-

oriented automata. The component specifications we obtain satisfy – by construction – all the requirements captured by the MSCs. This establishes a smooth transition from MSCs to individual component specifications. The schematic A/C specifications cover the entire MSC syntax and semantics from Chapter 4, and make the semantic interfaces of individual components within MSC specifications accessible to formal reasoning. As an application, we investigate the implications of property refinement steps on MSCs with respect to the individual components. The construction of automaton-specifications from collections of MSCs gives us a pragmatic method for obtaining "jump-start" models of individual component behavior from interaction patterns. The automatic derivation of early component prototypes and test drivers is just one example for applications of this constructive method.

Chapter 8 contains an assessment of what we have achieved in this thesis, as well as a discussion of future work.

Details of the semantic extensions to the core MSC semantics of Chapter 4, as well as the proofs of the propositions we make in Chapters 4 through 7 appear in the Appendix.

1.4. Roadmaps through this Thesis

Chapters 4 through 7 contain the central results of our work. The detailed discussion of MSC dialects and automaton notations in Chapters 2 and 3 serve mostly as background information with respect to the methodical role of MSCs and automata, and as reference material on the syntaxes and intentions of the respective approaches. Readers who are familiar with the notational aspects of MSCs and automata are invited to skip Chapters 2 and 3, and to review the corresponding contents only "on demand".

The four major topics we deal with, i.e. MSC semantics, MSC refinement, MSCs for property specification, and the transformation from MSCs to component specifications, yield four different reading-paths through this thesis besides the usual sequential one (cf. Figures 1.6 through 1.9). Below, we give a brief overview of these alternative paths.

MSC Semantics For readers whose main interest is our syntactic and semantic treatment of MSCs we recommend studying the Chapters 2 and 4.

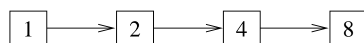


Figure 1.6.: Roadmap for studying MSC semantics

MSC Refinement For our treatment of MSC refinement we suggest to study Chapter 4 to get an impression of the underlying semantic model, followed by Chapter 5 where we introduce the refinement notions as such. To compare our refinement notions with the ones of MSC-96 and Interworkings, respectively, we propose also reading the corresponding sections of Chapter 2. For the relationship between MSC refinement and individual component refinement we refer the reader to Chapter 7 (especially Section 7.3.4).

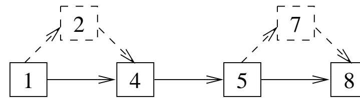


Figure 1.7.: Roadmap for studying MSC refinement

MSCs for Property Specifications Readers interested in various MSC interpretations, as well as in the kinds of properties (with respect to the safety/liveness classification) MSCs allow us to describe, the most relevant chapters are Chapter 4 (for the semantic bases of MSC interpretations and properties), and Chapter 6. For background information on the contrast between properties specified via MSCs, and properties of individual components we refer the reader to Chapter 7.

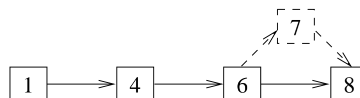


Figure 1.8.: Roadmap for studying property specification by means of MSCs

Transforming MSCs into Component Specifications Our exposition of the transformation from MSCs into relational component specifications in Chapter 7 is based on the syntax and semantics definition in Chapter 4, and on the "exact" MSC interpretation from Chapter 6.

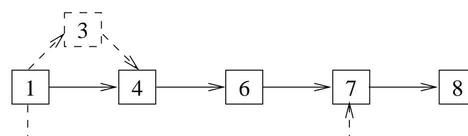


Figure 1.9.: Roadmap for studying the transformation of MSCs into component specifications

Because the transformation from MSCs into automaton specifications is purely syntactic its treatment in Section 7.4 is to a large extent independent of the concrete MSC semantics and interpretation. For concentrating on the syntactic transformation the reader might

want to skip Chapters 4 and 6, which provide the formal background for establishing the consistency between the MSCs and the resulting automata.

The automaton model we study in Chapter 7 shares features with several well-known automaton models, such as Mealy-, and ω-automata. To prepare the rather concise treatment of automaton syntax and semantics in Chapter 7 we refer the reader to Chapter 3.

1.5. Related Work

In the following, we give a brief overview of approaches from the literature at defining syntax and semantics of MSCs, at integrating them into development processes, and at their transformation into individual component specifications. At the end of this section we list the previously published material on which this thesis is partly based. In Chapters 2 through 7 we discuss the relationships with our work in more detail.

MSC Syntax and Semantics

Over the past 15 years, several notations for component interaction have been suggested in the literature. [IT96, IT98] defines the standard syntax and semantics for MSC-96. The semantics is given in a process-algebraic setting. In his PhD. thesis [Ren99] one of the contributors to [IT98] gives a thorough and accessible introduction to the history, syntax and semantics of MSC-96. In [GRG93, AHP96, MR94, MR96, Leu95, Fac95, SHB96, BHKS97a, Ber97, KW98] the authors define and discuss various semantics of MSC dialects. The underlying semantic models include Petri nets, partial order models, process-algebraic terms, variants of state machines, predicates over message traces, and timed rewriting logic. The focus of work is mainly the assignment of a semantics to individual MSCs, the discussion of different communication primitives, the detection of inconsistencies within MSCs, and the exploration of extensions to the standard MSC syntax and semantics. Standard MSCs, timing diagrams (cf. [SD93]) and statecharts (cf. [Har87, HP98]) have inspired the work on the MSC dialect LSC [DH99]. Here, the authors distinguish several interpretations for MSCs (similar to our discussion in Chapter 6), which allow, in particular, the definition of liveness properties. Some of these references (cf., for instance, [Fac95, Leu95, SHB96, Ber97, BHKS97a]) also place the integration of MSCs into development processes (see below) in the center of concern.

The UML [Rat97, RJB99] adopts Sequence Diagrams, a notation similar to Object Message Sequence Charts (OMSCs, cf. [BMR+96]) for the specification of component interaction. The syntax of OMSCs is, in turn, based on a small subset of [IT96]. However, besides some context conditions, UML does not provide a thorough foundation of the meaning of Sequence Diagrams, let alone their integration with other description techniques.

MSCs and Development Methods

The role of MSCs for capturing particular instances of behavior (scenarios), as opposed to complete component and system behavior, is in the center of concern of most contributions in the area of view-oriented system development. Refinement notions for MSCs and similar notations are rare; this is often a direct consequence of using MSCs for scenarios only.

Development methods and processes such as OMT [RBP+91], Objectory [JCJO92], Booch [Boo94], ROOM [SGW94], Syntropy [CD94], and Catalysis [DW98] suggest the use of (interaction) scenarios as a means for requirements capture in object-oriented analysis and design. In OMT, Objectory, and Booch variants of interaction diagrams express exemplary interaction scenarios. While Syntropy concentrates on statecharts as the description technique for scenarios, ROOM mentions standard MSCs explicitly. However, because the precise relationship between scenarios and complete actor behavior is left unspecified in ROOM, the former's integration into the development process is rather loose. In Catalysis, an extended variant of the UML's Sequence Diagrams (SDs, cf. [RJB99]) serves for capturing and refining use cases and operations. Here, the authors also state refinement notions similar to what we call message and structural refinement in Chapter 5. The authors of [FS97] and [Dou98] also describe the use of SDs in object-oriented system development. They mainly suggest SDs as a means for exemplary interaction descriptions. Yet, [Dou98] makes use of state information within SDs to relate state-based and interaction-based component specifications. In the "unified software development process" [JBR99] the authors recommend using SDs during the design phase of the development process. The purpose of the SDs here is to give a more detailed representation of the use cases identified during requirements capture and analysis. The authors also mention the advantages of SDs for determining an adequate decomposition of the overall system into interacting subsystems, as well as for identifying the corresponding subsystem interfaces.

Similar to the Syntropy approach, the author of [Kle98] uses transition systems as the basis of defining the semantics of scenarios. The author also distinguishes between existential and universal interpretations of scenarios as a specification technique, and provides property refinement rules (in the sense of Chapter 5) for the underlying transition systems to facilitate a scenario-based development process. Scenario and use case specification and refinement is also the topic of [MR96] and [BC96], respectively. In [WK96] the authors use rewriting logic as a formal basis for an extension of the method suggested in [JCJO92]. [WK96] includes, in particular, the definition of a formal notion of model refinement, based on bisimulation. As one application of this refinement notion the authors discuss the binding of message parameters to concrete values.

The authors of [HSG+94] use regular grammars and corresponding state machines to represent scenarios for individual components in sequential systems. A similar approach, independent of a particular graphical syntax, appears in [RKW95]; here the focus is on the synthesis of multiple use case descriptions during requirements analysis that can also serve as a preparation for verification and testing. Within the ESPRIT project CREWS (Cooperative Requirements Engineering With Scenarios, cf. [RA98, ARS98, ATS99] and the

references therein) scenarios and their transition to requirements specifications are considered. Although the authors also mention notions of scenario composition, and even touch on the issue of refinement, the focus of their work is mainly the rather informal capturing and managing of scenarios in natural language; this is in contrast to our aim of using MSCs as an intuitive, yet formal description technique seamlessly within the development process.

In [BHKS97a] we have investigated a variant of MSCs named extended event traces (EETs, cf. [SHB96]) as a means for specifying communication in software architectures. One of the applications of this work is providing a foundation for the usage of MSCs in pattern descriptions [GHJV95, BMR+96], which employ MSCs as a description technique for scenario documentation. The author of [Pae97] uses state transition diagrams as a basis for the specification of actor roles in software architectures. These roles are, in a sense, projections of the EETs used in [BHKS97a] onto individual actors. An approach to using scenarios in a form similar to the UML's Sequence Diagrams and EETs for capturing interaction requirements for object-oriented systems appears in [Bre99].

Several contributions deal with the combination of MSCs and validation technologies, such as testing and verification. [GHN93, NGH93], [SKGH98] and [GKSH99] are only a few examples of a large body of work on using MSCs for the specification of test cases. The author of [Hau97] presents transformations from a subset of standard MSCs to temporal logics with the aim of performing model-checking of the resulting formulae against system properties also specified in temporal logic. [BAL96] describes a tool prototype supporting the use of High-Level MSCs (cf. [IT96]) in scenario-based design of concurrent systems. Furthermore, the authors suggest to interpret MSC specifications with model-checking tools. The work by Holzmann et al. (cf. [Hol95, Hol96, AHP96]) is based on a partial order semantics for an MSC dialect. The semantics definition provides the option to "plug-in" various communication models (such as asynchronous, FIFO, and synchronous). This allows the authors to define and to detect race conditions between messages. Based on the ideas contained in [Hol95, Hol96, AHP96] a toolset has been developed that allows checking MSC specifications for race conditions automatically (cf. [UBE99]); in its most recent version this tool also addresses the construction of component specifications from MSCs. The model checker Spin [Hol97] uses MSCs to illustrate counter-examples for falsified properties.

From MSCs to Component Specifications

As we have mentioned above, the traditional and most frequently cited role of MSCs in the development process is the specification of particular interaction scenarios among several components. The methodical transition from collections of scenarios to complete component behavior is still an active area of research.

[BGH+98] (an extension of our work in [BHKS97a]) contains a discussion of such a transition in the context of object-oriented system development. The authors suggest to capture interaction requirements by means of EETs, and to compose individual EETs to obtain

the complete interaction behavior of the system components under consideration. Scenario elicitation, combination, and validation by means of MSCs is also mentioned in [SGW94] as a means for capturing, specifying, and designing protocol classes in ROOM models.

In [BK98] (and in Chapter 7 of this thesis) we use "interaction interfaces" as behavior specifications of components, connected via directed channels, to assign meaning to MSCs. The focus of that text is on the derivation of schematic assumption/commitment (cf. [MC81, Pan90, Bro95]) specifications from a given interaction interface; this paves the way for the semantic integration of state-based with interaction-oriented description techniques. [Bro98] and its extension [Bro99b] present transformation schemes from MSCs to defining equations for components. This not only defines a semantics for basic MSCs (by composition of the functions resulting from the defining equations for all components) but also complements the generic assumption/commitment specifications of [BK98].

The constructive transformation of MSCs into state-based automaton specifications for individual components is the topic of [KM93, KMST96, KSTM98], [LMR98], [Fei99], [HK99], and [BGK99, KGSB99]. The approaches differ mainly in the amount of guidance (in the form of design-knowledge about the state-oriented behavior of the components) the developer can contribute to the transformation procedures, the syntactic and semantic scope of the MSCs used, the necessity and potential for optimization of (intermediate) results, and the complexity of the transformation. In Chapter 7 we extend the work in [BGK99, KGSB99], and compare all of these approaches in detail.

Previously Published Material

The material covered in this thesis is based, in part, on our contributions in [BGH+97, BHKS97a, BHKS97b, BGH+98, BK98, BGK99, GKS99a, GKS99b, KGSB99, Krü99b, RBK99a, RBK99b, GKS00].

CHAPTER 2

MSC Notations – Introduction and Comparison

To obtain an intuition of the properties we can express with MSC specifications we discuss and compare several MSC dialects in this chapter. Our introduction to the different notations is example-oriented to convey their corresponding visual appearance. We identify potential for improvements of the existing MSC notations with respect to our goal of a seamless integration of MSCs into the development process.

Contents

2.1. Introduction **24**
2.2. MSC-96 . . . . **25**
2.3. Other MSC Dialects **50**
2.4. Comparison and Prospective Enhancements **73**
2.5. Related Work **77**
2.6. Summary **78**

2.1. Introduction

Several graphical description techniques for component interaction have emerged over the past decade. In this chapter we discuss several of these dialects to get an impression of the existing syntaxes, their application domains, as well as their semantic foundations.

We start our discussion with *MSC-96*, whose origin is the specification of interaction scenarios in the telecommunications domain. Our interest in this notation is triggered by its rich syntax and the corresponding formal semantics definition, which cover not only finite interaction scenarios, but also infinite behavior; moreover, MSC-96 provides means for structuring MSC specifications, going beyond what most other notations offer. Therefore, we give a thorough introduction to MSC-96's syntax, and let it serve as the "reference notation" during the discussion of the other MSC dialects. We also briefly mention the extensions introduced by *MSC 2000*, the successor to MSC-96.

The specification of (tele)communication protocols is one of the application domains for MSCs with the longest tradition. However, the increasing interest in interaction scenarios and use cases in object-oriented analysis and design have spawned several dedicated MSC dialects. These dialects typically cover finite interaction patterns only, but integrate special syntax for method calls and control flow. We have selected *Object Message Sequence Charts* and the UML's *Sequence Diagrams* as representatives of this class of MSC notations.

The remaining MSC dialects we cover in this chapter (*Extended Event Traces*, *Interworkings*, *Hybrid Sequence Charts*, and *Life Sequence Charts*) are not only syntactic variations of what MSC-96, Object Message Sequence Charts, and Sequence Diagrams provide. As opposed to MSC-96 message parameters have a formal semantics within Extended Event Traces; this allows integrating formal constraints on the data values of messages into interaction specifications. Besides being predecessors to MSC-96, Interworkings contribute a mentionable notion of completeness to interaction specifications. Life Sequence Charts stress the distinction between partial and complete interaction specifications further and provide syntax and semantics for expressing this distinction. Life Sequence Charts and Hybrid Sequence Charts are the only MSC dialects in our selection that assign a formal semantics to individual component's states as part of interaction patterns. Hybrid Sequence Charts also contribute syntax and semantics for preemption specifications to the MSC notation.

In the following sections we introduce all of these notations in more detail. The style of presentation is example-oriented, i.e. for most notational elements we give a corresponding example in the respective graphical notation. This way, we not only convey the notational features of the different MSC dialects, but also give an impression of their "look-and-feel". By intention we do not judge intensively on the design choices made or syntactic and semantic constraints imposed by the authors of the respective description technique. The purpose of this chapter is to expose the syntactic and semantic "features" of several MSC dialects, and to identify the potential for improvement we address with our own MSC

notation in Chapter 4. For a brief summary and a comparison of the MSC dialects we refer the reader to Section 2.4.

As the running example for this chapter we consider the interaction behavior in a much simplified central locking system (CLS) for car doors (cf. [KGSB99]). We assume that this system consists of three major components: a controller, a lock motor for the left door lock, and a lock motor for the right door lock. The purpose of the CLS is to activate the left and right motors according to a signal from the car user. If the controller receives a "lock" signal from the user then it initiates the locking of the car by sending both motors a "down" signal. Similarly, if the controller receives an "unlock" signal from the user then it initiates the unlocking of the car by sending both motors an "up" signal. The MSCs occurring in Sections 2.2 and 2.3 depict these interactions in several variations. For the explanation of some syntactic elements we modify the CLS slightly to include another component for logging traces of the actions performed by one of the motors.

Because Chapter 4 contains a thorough formal semantics definition of the MSC dialect used in the remainder of this thesis, we explain the semantics of the MSC dialects in this chapter only informally. Conversely, this chapter also serves as a source for examples for the much more concise definitions in Chapter 4.

Readers who are familiar with the graphical syntax of one or all of the mentioned MSC dialects, or are more interested in the formal semantics definitions, are invited to skip Sections 2.2 and 2.3. Instead, they might want to consult Section 2.4, which contains a brief comparison of the dialects, before continuing with Chapters 3 and 4.

The remainder of this chapter has the following structure. In Section 2.2 we treat MSC-96 in detail. This establishes the foundation for the discussion of the other MSC variants in Section 2.3. As mentioned above, Section 2.4 contains a comparison of the syntactic and semantic "features" of the MSC dialects discussed in Sections 2.2 and 2.3. We mention related work in Section 2.5, and give a summary of this chapter in Section 2.6.

2.2. MSC-96

Message Sequence Chart is the name of a description technique for component interaction recommended by ITU [IT96, IT98]. The version of this recommendation we cover here is called MSC-96; it introduces both a graphical and a textual syntax for MSCs. For the purposes of this section we concentrate on the graphical syntax.

The intended use of MSCs is to provide an "overview specification of the communication behavior for real time systems, in particular telecommunication switching systems" (cf. [IT96]). In fact, MSCs have their roots in the development of telecommunication systems using the "Specification and Description Language" (SDL, cf. [EHS98]). We refer the reader to [Ren99] for a detailed overview of the history of MSC-96.

The basic assumption underlying the use of MSC-96 is that the system under development consists of a set of components, communicating by means of asynchronous message passing. There is no notion of a global clock; the components (or *instances*, as [IT96] and [IT98] call them) operate time-asynchronously.

Compared to other MSC notations (see also Section 2.3), MSC-96 is a rather baroque language; it provides constructs for specifications in the range from simple interaction patterns to complete component behavior. We introduce these constructs along the following structure. In Section 2.2.1 we discuss the basic notational elements of MSC-96: instance axes, message arrows, environment frame, actions, and conditions. These constructs suffice for the specification of simple interaction patterns. MSC-96 also enables the capturing of alternative, repeated, and parallel interaction patterns. We deal with these constructs for structuring and composing MSC specifications, as well as with reference expressions (a substitution mechanism), gates (a means for splitting information among several MSCs), and High Level MSCs (a hierarchic notation for MSC composition) in Section 2.2.2. In Section 2.2.3 we describe miscellaneous notational elements for expressing instance creation and deletion, timing constraints, instance decomposition, and incomplete messages.

We refer the reader to [IT96, IT98, Ren99] for a detailed presentation of MSC-96's syntax and semantics, including all syntactic options we avoid here for reasons of brevity.

2.2.1. Basic MSC Notation

We start with the most basic MSC constituents: instance axes and message arrows. These two ingredients reappear in almost all graphical notations for component interaction. MSC-96 complements them by symbols for specifying component conditions and component actions. In the following paragraphs we discuss all of these concepts, as well as the ones MSC-96 provides for defining the ordering of events within MSCs.

Instances and Messages

Figure 2.1 shows an elementary MSC in graphical form. It displays a sequence of interactions among three instances that are represented by vertical axes. An axis starts and ends with a non-filled and a filled rectangle, respectively. The non-filled rectangle is the "instance head symbol", the filled rectangle is the "instance end symbol". Labels at the top of each axis indicate the corresponding instance's name. Here, the instance names are *Control*, *LM*, and *RM*. Arrows, directed from the sending to the receiving instance, denote communication. The label on an arrow denotes the message exchanged by the two instances. In our example, the message labels are *ldn*, *rdn*, *lmr*, and *rmr*. The frame around the instances is part of the MSC; it represents the environment. The name of the MSC, given after the boldface keyword **msc**, serves as an identifier for referencing the entire MSC. In this case, the name of the MSC is *locking*.

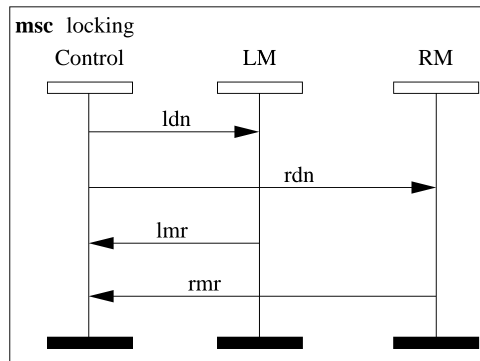


Figure 2.1.: Simple MSC

[IT98] defines the semantics of MSCs formally in a process-algebraic setting. Intuitively, the meaning of an MSC is the sequence of messages obtained by reading the MSC from top to bottom. Because – according to the documents defining the standard [IT96, IT98] – there is no global clock, there may exist an individual time scale for each instance depicted in an MSC. Moreover, communication happens asynchronously; there may be a delay between the sending and the receipt of a message. Therefore, for any message the authors of the semantics document [IT98] distinguish between two events: the sending of the message, and its receipt. If the name of a message is m, we denote the corresponding send and receive event by $s.m$, and $r.m$, respectively. As a consequence of the absence of a global clock the events on different instance axes are not ordered per se. The ordering is established by several rules in the MSC-96 semantics; we mention these rules informally along with the introduction of the respective syntax elements.

The following four restrictions, imposed by the MSC-96 standard, induce a partial order on the events occurring within an MSC:

1. every send event precedes its corresponding receive event;
2. on any location on an instance axis at most one send event may occur;
3. at most one receive event may be at the same location as a send event on the same axis; the receive event precedes the send event;
4. the events on a single axis are totally ordered according to their occurrence from top to bottom.

The semantics of an MSC, according to [IT98], is the set of all sequences of events that correspond to the messages depicted in the MSC and obey these rules. MSC-96 offers two constructs to relax the ordering imposed by these restrictions: coregions and generalized orderings. We will discuss these advanced concepts later in this section.

MSC *locking* of Figure 2.1 induces the partial event order shown in Figure 2.2 (a). An arrow from one event to another denotes the precedence of the event at the tail of the arrow over the event at the arrow's head. The events occurring on any particular instance axis of the MSC are totally ordered. Events on different instance axes can be unrelated, as *r.ldn* and *r.rdn* show in this example.

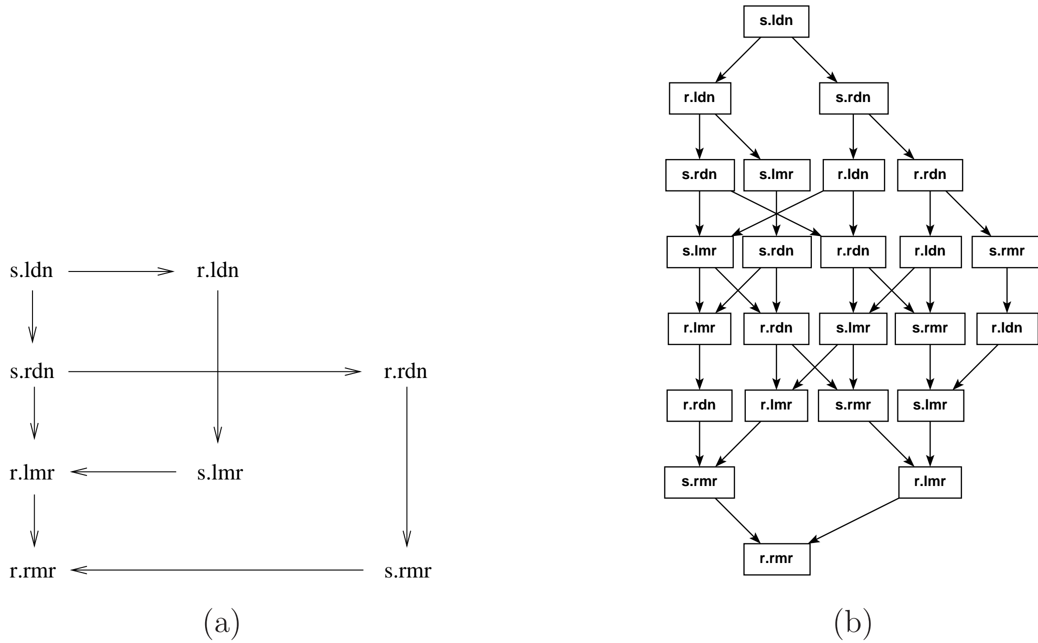


Figure 2.2.: Partial event order and MSC semantics

Obtaining the semantics of the MSC in Figure 2.1 proceeds as follows. We collect all possible sequences of the eight send and receive events, such that in each sequence the events obey the partial order from Figure 2.2 (a). Figure 2.2 (b) shows all of these sequences graphically. Here, the labeled nodes represent the events occurring in the MSC. An arrow from one node to another denotes that the event corresponding to the source node occurs before the event corresponding to the destination node. Each sequence of events obtained by following the graph from its top to its bottom, and collecting the events along the way, yields one element of the semantics of the MSC. In total there are 19 different paths through this graph.

This large number of different interpretations for such a simple MSC seems, at first sight, irritating. It stems from the many possible interleavings of send and receive events according to both the absence of a global clock, and the asynchronous message passing, as required by the semantics document [IT98].

Environment Frame

The environment frame of an MSC serves as a representative for the origin or destination of arrows whose source or destination, respectively, is outside the scope of the MSC. This allows us, for instance, to leave the concrete sender or receiver of a message unspecified. To denote message exchange with the environment we position the corresponding arrow's head or tail at the frame. As an example, consider Figure 2.3. Here, instance *Control* exchanges the messages *ldn*, *rdn*, *lmr*, and *rmr* with the environment.

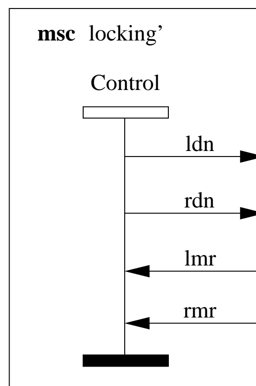


Figure 2.3.: Instance *Control* exchanges messages with the environment

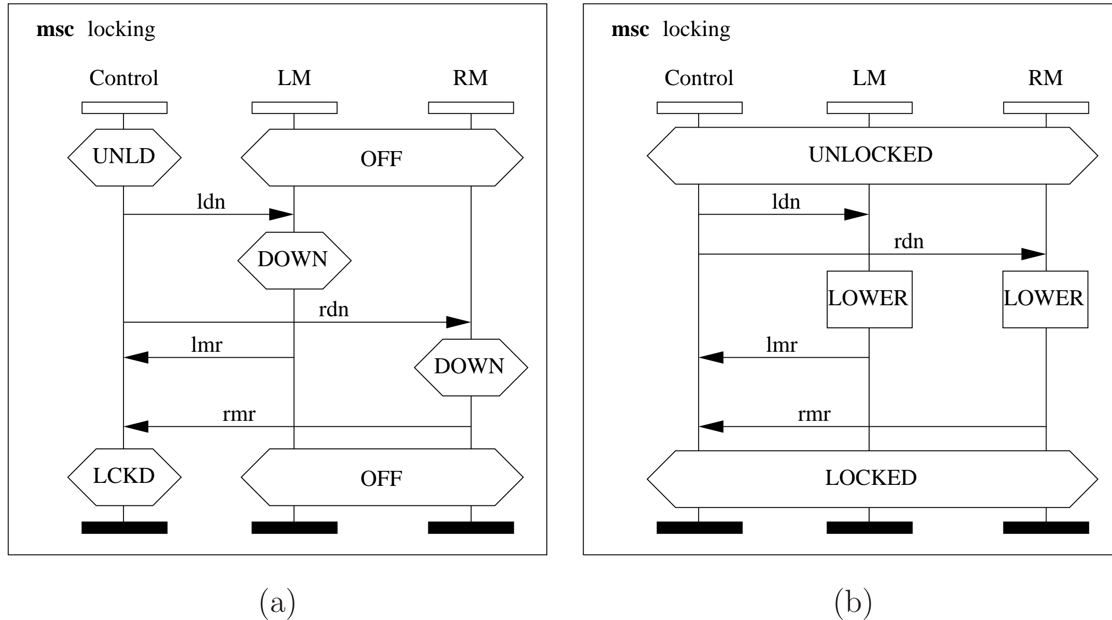
Events on the environment frame are unordered. In the example of Figure 2.3 the events *r.ldn*, *r.rdn*, *s.lmr*, and *s.rmr* can occur in an arbitrary order, whereas the other events must occur in the following sequence: *s.ldn*, *s.rdn*, *r.lmr*, and *r.rmr*.

Besides modeling communication with (possibly) anonymous environment instances, the environment frame also allows breaking up a single MSC into two, such that the source and the destination of arrows can end up in different MSCs. We will come back to this purpose of the environment frame later in this section, in connection with MSC-96's gate concept.

Conditions

MSC-96 offers condition symbols as a means for indicating that one or more instances fulfill a certain condition before or after they participate in an interaction sequence. Graphically, a condition is a labeled angular box placed on the instance axes of the components fulfilling the condition. As an example for an MSC with conditions, consider MSC *locking* from Figure 2.4 (a).

MSC-96 distinguishes three kinds of conditions, according to how many instances of an MSC they cover. *Local conditions* cover exactly one instance axis, *nonlocal conditions* cover more than one instance, but not necessarily all of them, and *global conditions* cover



(a) (b)

Figure 2.4.: MSCs with conditions and actions

all instances. The intuition behind nonlocal and global conditions is that the covered instances all fulfill the stated condition. The MSC of Figure 2.4 (a) has two nonlocal conditions labeled *OFF*. Figure 2.4 (b) shows two global conditions labeled *UNLOCKED* and *LOCKED*.

Semantically, conditions serve only one purpose in MSC-96: they are the basis for composing MSCs within High Level MSCs (see Section 2.2.2); besides that, conditions do not contribute to the meaning of an MSC. Although MSC-96 does not assign much meaning to conditions they have many useful applications. For instance, the developer may use conditions to mark phases of a communication protocol within an MSC; used in this way, conditions can enhance the readability of an MSC specification. Another application is expressing information on the control and data state of an instance by means of appropriately labeled conditions. We will exploit this possibility extensively in Chapter 7, where we discuss the transition from interaction-based to state-based description techniques.

Actions

To specify that an instance performs some local activity (such as state changes through assignments), MSC-96 provides the concept of actions. Their graphical representation is a labeled rectangle attached to the instance performing the action. Semantically, an action represents a local event of its corresponding instance. Action events contribute to the total event ordering along an instance axis, and, hence, indirectly to the partial ordering of all events occurring in an MSC. As a simple example for the specification of actions, consider the MSC of Figure 2.4 (b); here, both *LM* and *RM* perform the local action *LOWER* after

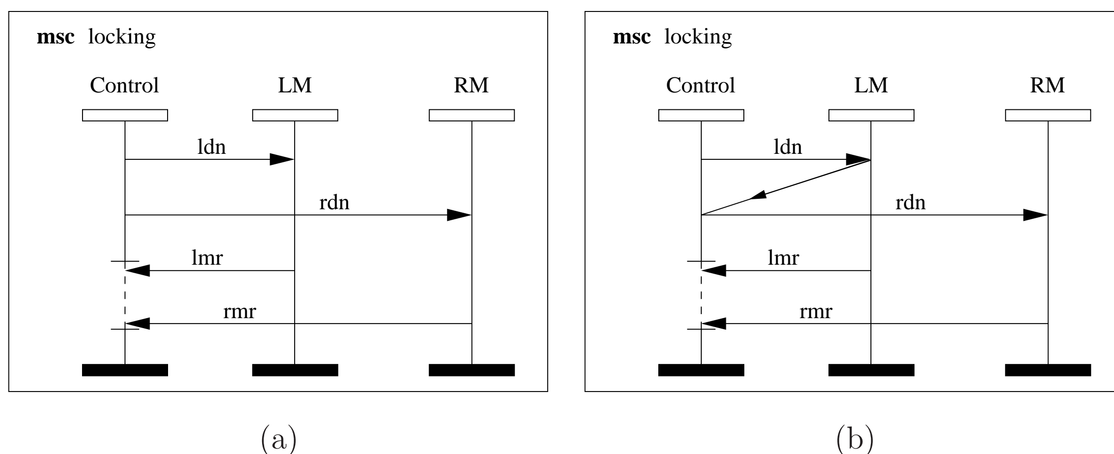
receiving the message *ldn* and *rdn*, respectively. The two action symbols represent two different local events of the instances *LM* and *RM*.

Event Ordering Mechanisms

The ordering of events induced by an MSC is defined by the rules we have given above. There are situations, however, where these rules are either too strong, or too weak to express the desired interaction sequences succinctly with MSCs. Consider, for instance, the MSC of Figure 2.1. Its semantics states that the messages *lmr* and *rmr* arrive at instance *Control* exactly in the following order: *lmr* precedes *rmr*. Hence, this MSC alone cannot describe situations in which *lmr* arrives at *Control* after *rmr*. MSC-96 provides *coregions* for weakening, and *general orderings* for strengthening the standard event ordering.

Coregions

A coregion, whose graphical representation is a vertical dashed line delimited by short horizontal lines, is part of an instance axis. All events located on a coregion are unordered.



(a) (b)

Figure 2.5.: MSCs with coregions and general orderings

As an example, consider the MSC of Figure 2.5 (a). Here, the two events *r.lmr* and *r.rmr* are located on *Control*'s coregion. Hence, these two events are unordered.

General Orderings

A general ordering establishes a precedence between two events in an MSC that would be unordered otherwise. It is represented graphically by a (possibly bent) line between the two events under consideration. Attached to the line is an arrow head that points from the event occurring first to the one occurring second. To distinguish general ordering symbols from message arrows, the former's arrow head's position must not be at the end of the line connecting the two events.

As an example, consider the MSCs from Figures 2.5 (a) and (b). In Figure 2.5 (a) the events *r.ldn* and *s.rdn* are unordered, whereas the general ordering of Figure 2.5 (b) states that event *r.ldn* must precede event *s.rdn*.

The general ordering of Figure 2.5 (b) relates two events on different instance axes. We may use general orderings also for ordering events within coregions. To increase the readability of such specifications MSC-96 allows an alternative way of drawing instance axes. Here, the axis is in fact a column, as wide as the instance head and end symbols. The incoming and outgoing arrows end and start, respectively, at the border of the column. This allows us to draw the generalized ordering symbols within the column.

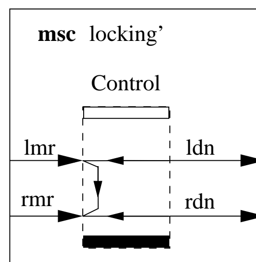


Figure 2.6.: MSC with general ordering within a coregion

Figure 2.6 shows an example of a general ordering of events on an instance. The axis of instance *Control* appears in column form; it is covered by a coregion.

Ordering arrows may connect orderable events (such as send, receive, and action events; cf. [IT98, Ren99]) only. Semantically, we can think of ordering arrows as being "regular" message arrows, labeled with distinct, anonymous messages.

2.2.2. MSC Composition and Structuring

The syntactic and semantic elements of MSC-96 introduced so far already suffice to model simple interaction sequences, and to give hints under which conditions these may occur. Often, however, we are interested not only in depicting one interaction sequence, but in alternatives or repetitions within interactions. We could, of course, draw individual MSCs for each possible combination of message exchanges, and accompany this with an explanation of how these different pictures relate. For typical systems this would lead to very large numbers of MSCs very quickly.

As a remedy, MSC-96 provides various concepts for composing and structuring MSC specifications:

- *Inline expressions* support the specification of alternative, repeated, and parallel interaction patterns within a single MSC.

- *References* allow reuse of interaction patterns among MSCs; the referencing MSC "imports" the interaction pattern of the referenced MSC.
- *High Level MSCs* depict alternative, repeated, and parallel interaction patterns by relating MSC references; this enables specification of "roadmaps" through sets of MSCs.
- *Gates* facilitate the decomposition of MSCs.

Inline Expressions

MSC-96 provides the notion of inline expressions to represent alternative, optional, parallel, and repeated parts within an MSC. The graphical syntax of an inline expression is a rectangle whose upper left corner indicates the expression's type. Every inline expression covers at least one instance, and makes a statement about the events within the rectangle.

Alternative Inline Expression Figure 2.7 (a) depicts two alternative message exchanges within an **alt**-inline expression box. The dashed line separates the two alternatives. In general, the box may hold any positive, finite number of alternatives, with a dashed line between any two consecutive alternatives. An alternative box specifies the choice among several paths through the MSC; each alternative yields a separate path. In the example of Figure 2.7 (a) we specify the occurrence of events *r.lmr* and *r.rmr* in arbitrary order.

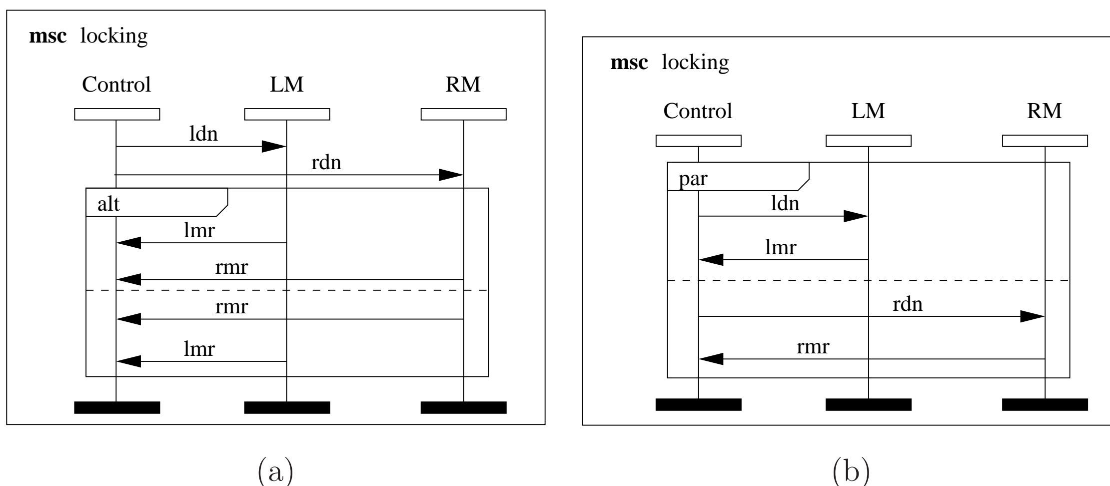


Figure 2.7.: MSCs with alternative and parallel inline expressions

MSC-96 offers a notational shortcut, called *optional region*, for alternative boxes with two alternatives, where one of the alternatives is empty; an empty alternative does not contain any events. The graphical syntax for an optional region is an inline expression box labeled with the keyword **opt**.

Parallel Inline Expression To express concurrency among a set of interaction sequences MSC-96 offers the parallel inline expression operator. Its graphical syntax is similar to the one of the **alt**-inline expression, except that the parallel operator's label is the keyword **par**. In the regions, which are separated by dashed lines, we now depict the interaction sequences occurring mutually independently. Note that the **par**-inline expression specifies the absence of event ordering only across, not within the operand regions.

Consider the example in Figure 2.7 (b). It specifies the independence of the send and receive events in the set $\{s.ldn, r.ldn, s.lmr, r.lmr\}$ from those in the set $\{s.rdn, r.rdn, s.rmr, r.rmr\}$. Yet, the events within each of the two sets are partially ordered, according to the rules given in Section 2.2.1.

Loop Inline Expression MSC-96 allows specification of repeated interaction patterns by means of the **loop**-inline expression. Depending on its format the **loop**-label determines the number of possible repetitions:

- **loop**$< l, u >$: let $l, u \in \mathbb{N}$. If $l \leq u$, then the interaction sequence may occur at least l and at most u times. If $u < l$, then the semantics of the **loop**-expression is equivalent to that of the empty MSC. If $l \in \mathbb{N}$, and $u =$ **inf** (short for infinity) then the number of occurrences of the interaction sequence is bounded only from below by l.
- **loop**$< l >$: if we have $l \in \mathbb{N}$, this is a notational shortcut for **loop**$< l, l >$, which specifies that the interaction sequence occurs exactly l times. If $l =$ **inf**, then the interaction sequence occurs infinitely often.
- **loop**: this is a notational shortcut for **loop**$< 1,$ **inf** $>$; the interaction sequence may occur an arbitrary number of times.

As an example, consider the MSC in Figure 2.8. Here we have specified that the interaction sequence must occur at least once and at most five times.

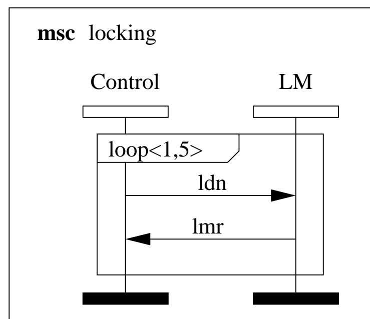


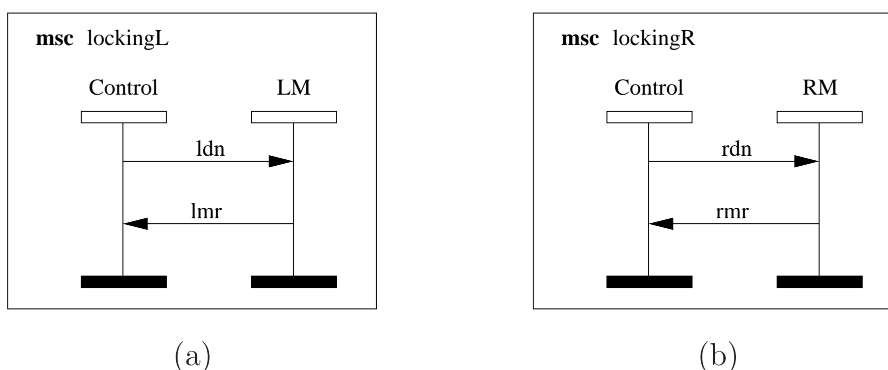
Figure 2.8.: MSC with repetition

MSC Documents and Reference Expressions

Typical specifications of realistic size consist of more than one MSC, even given the presence of the inline expressions we have introduced above. The collection of MSCs that a specification consists of forms an MSC document. Figure 2.9 shows an example of the graphical representation of an MSC document named *LU*. It consists of the keyword **mscdocument**, followed by the document's name, both enclosed within a frame. The idea is that all MSCs belonging to a specification appear in the same document. The developer must establish the containment relationship between an MSC and its document; MSC-96 does not provide graphical syntax for this purpose. In the textual syntax, however, this relationship is explicit (cf. [IT96, IT98]). The names of all MSCs within the same MSC document must be unique.

Figure 2.9.: MSC document frame

References The interaction depicted in one MSC may occur in the same or a similar way also in other MSCs. To facilitate reuse of MSCs within the same MSC document MSC-96 provides a referencing mechanism. As an example consider the MSCs *lockingL*, *lockingR*, and *locking* from Figures 2.10 and 2.11. We assume that they all belong to the same MSC document. MSC *locking* references the MSCs *lockingL* and *lockingR*. The graphical representation of an MSC reference is a box with rounded corners, labeled with the referenced MSC. To obtain the semantics of an MSC with references we have to substitute the events of the referenced MSC, in the order specified there, for the reference box in the MSC under consideration. Note that the placement of the reference symbols may establish an ordering of the events on an instance's axis; see instance *Control* in Figure 2.11 for an example.



(a) (b)

Figure 2.10.: Communication between *Control*, *LM*, and *RM*

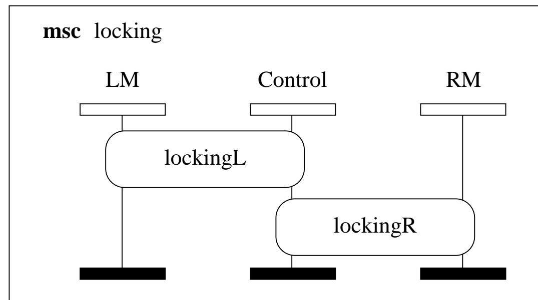


Figure 2.11.: MSC with references to *lockingL* and *lockingR*

An MSC containing references must fulfill three conditions:

1. references must be acyclic, i.e. no MSC may refer to itself, be it directly or indirectly; this ensures termination in the substitution process.
2. the reference box must overlap all instance axes that occur in the referenced MSC; it may also overlap instance axes that do not occur in the referenced MSC.
3. if the MSC contains at least two references, then all instance axes shared by at least two of the referenced MSCs must be present in the referencing MSC; this ensures that we can always assign an order to the events on axes that occur in referenced MSCs.

Reference Expressions Besides simply giving the name of the referenced MSC we may label reference boxes with expressions denoting alternatives, repetition, and parallelism, similar to the inline expressions we have discussed above. Examples of such expressions are (A **alt** B **alt** C) and (**loop** $<1,$ **inf** $>$ (A **par** B)), where A, B, and C are MSC names.

Furthermore, we can perform substitutions on the referenced MSC to change the names of instances and messages. This serves the purpose of adapting the referenced MSC to the context of the referencing MSC.

MSC *locking* of Figure 2.12 shows an example of references with substitutions. Its semantics is equivalent to the one of MSC *locking* in Figure 2.11.

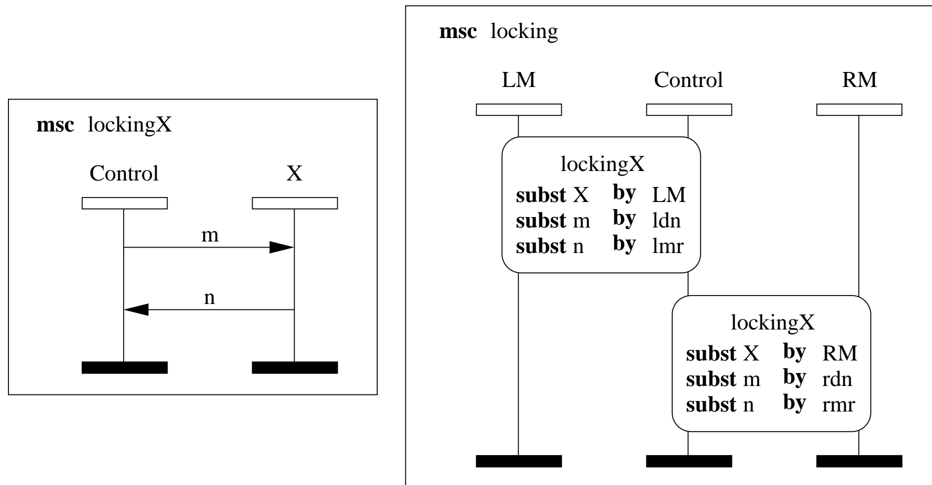


Figure 2.12.: MSC with references and substitutions

High Level MSCs (HMSCs)

The syntactic constructs we have introduced so far allow us to specify both simple and complex sequences of interactions. By means of the referencing mechanism we can decompose large interaction patterns into manageable parts. This is a first step towards increasing the comprehensibility of large MSC specifications. It does not suffice, however, for conveying the "big picture", i.e. the way all the MSCs that form a specification relate or compose; we still have to find and follow all references within an MSC document to obtain the sequences of interactions occurring in the system under specification. In addition, the instance axes appearing in all MSCs add to the complexity of the pictures we draw. If we wish to represent, say, three successive phases of a communications protocol as *connect*, *transmit*, and *disconnect*, we do not want to reveal right from the beginning in the development process into what components the system decomposes, and what the exact interactions among these components are. Instead, we would like to say something like "an execution of the system consists of an infinite sequence of steps; each step consists of three consecutive phases: connect, transmit, and disconnect". This high level specification of the protocol references neither components, nor messages. However, none of the syntactic elements we have studied up to now allows us to specify interaction sequences on this high level of abstraction.

MSC-96 introduces *High Level MSCs* (HMSCs) as a notational alternative to the *plain MSCs* we have discussed so far, to address these problems. An HMSC is, in essence, a directed graph whose nodes reference (H)MSCs; the graph describes a "roadmap" through the MSCs referenced in the nodes. An edge from a node labeled with MSC X to a node labeled with MSC Y in the graph indicates that part of the interaction behavior of the system consists of a sequencing of the interactions specified in X and those specified in Y. The edges determine how we must "paste together" the MSCs referenced in the nodes to

obtain the interaction sequences of the system.

More precisely, each node of an HMSC is any one of the following

- a start node,
- an end node,
- a reference node,
- a parallel node,
- a connection node,
- a condition node.

We will introduce each of these node classes informally, in turn. We also briefly relate HMSCs and plain MSCs so that we can translate the former into the latter.

Start, End, and Reference Nodes Each HMSC has exactly one start node, whose graphic representation is a downward outlined triangle. The start node indicates the beginning of any interaction sequence we can derive from the HMSC; it has no incoming edges. An end node, whose graphic representation is the horizontal mirror image of the start symbol, terminates paths through the HMSC; it has no outgoing edges. Interaction sequences obtained by following any path through the HMSC end when we reach an end node. Reference nodes are similar to the MSC reference symbols we have already discussed. Their labels may be MSC reference expressions as before; they also have the same graphic representation.

Consider the HMSC *l2u* from Figure 2.13. It consists of a start node, two MSC reference nodes, and the end node. The arrows indicate valid paths through the graph. In this case there is exactly one path through the HMSC. It begins at the start node, passes through the MSC references for *locking* and *unlocking* – in this order – and stops at the end node. Intuitively, we obtain the semantics of this HMSC by pasting the interactions of MSC *unlocking*, in their specified order, under those of MSC *locking*, and by determining the resulting MSC's semantics according to the event ordering rules introduced above.

MSC-96 calls the composition form used in MSC *l2u* "weak sequential composition". Assume given an arbitrary HMSC with an arrow from a reference node labeled A to a reference node labeled B. In the composite HMSC there is a semantic difference between events on common instances of A and B, and events on instances appearing in only one of the two referenced MSCs. On common instances A's events precede B's events. Events on instances of A that differ from B's instances are independent of events in B. Similarly, events on instances of B that are not also instances of A are independent of events in B.

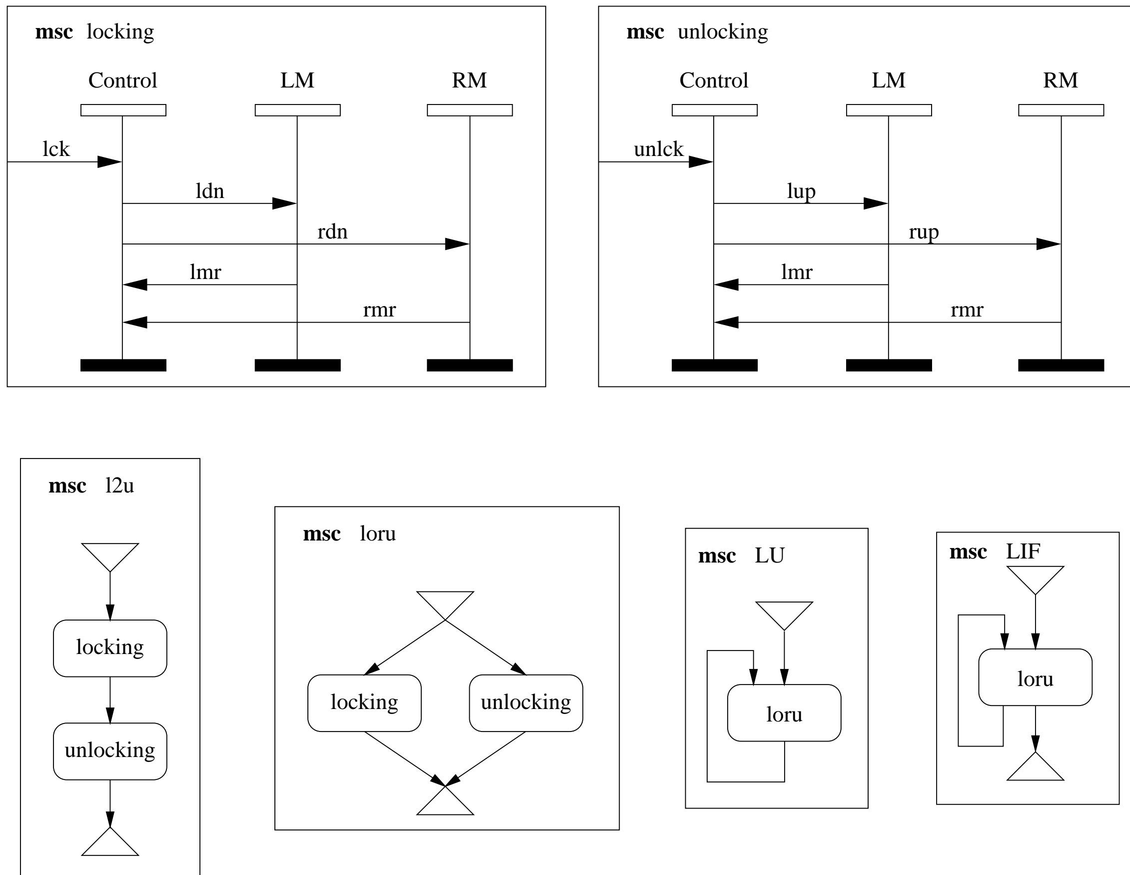


Figure 2.13.: Plain and HMSCs

The start node, and all reference, parallel, condition, and connection nodes of an HMSC can have an arbitrary number of outgoing edges. In fact, all nodes but the end node must have at least one outgoing edge. A node with more than one outgoing edge indicates existence of an alternative path through the HMSC. As an example consider MSC *loru* of Figure 2.13. Here the start node has two outgoing edges; the first leads to a reference to MSC *locking*, the second leads to a reference to MSC *unlocking*. Intuitively, the semantics of this HMSC is the set of interaction sequences obtained by following any one of the two possible paths through the graph. Note the similarity of this construct and the **alt**-inline or -reference expression.

MSC-96 allows cycles in HMSC graphs. This corresponds to an infinite or unbounded repetition of the interactions determined by the nodes along the path forming the cycle. Figure 2.13 shows an example of an infinite repetition in HMSC *LU*. This HMSC has a start but no end node. Intuitively, its semantics consists of an infinite sequencing of the interactions represented by HMSC *loru*. Note the similarity of this construct and the **loop**<**inf**>-inline expression. An unbounded repetition appears in HMSC *LIF* of Figure 2.13. This corresponds to the **loop**<1, **inf**>-inline expression.

Parallel Nodes Besides sequential composition, alternatives, and repetition, HMSCs also allow us to specify independence of the events of entire MSCs. For that purpose MSC-96 introduces the parallel node. Its graphic representation is a box. A parallel node may contain any number of HMSCs. Intuitively, the events occurring in the HMSCs within a parallel node are mutually independent.

Consider the HMSC *lpu* from Figure 2.14. It contains a parallel node, which, in turn, contains two HMSCs; each of these consists of only one reference to MSC *lockingL* and MSC *lockingR*, respectively. HMSC *lpu* specifies that the events defined in *lockingL* and *lockingR* are unrelated. Note the similarity of this construct and the **par**-inline and -reference expression.

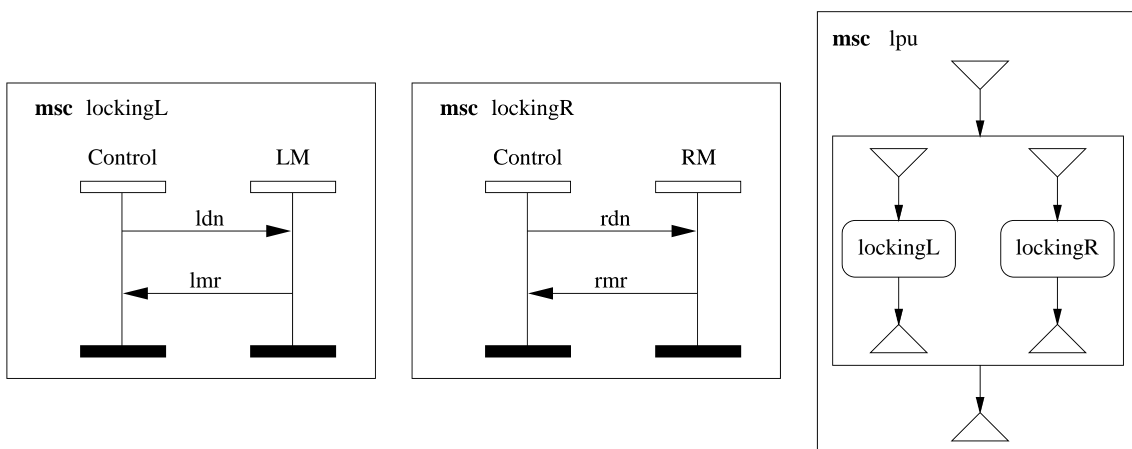


Figure 2.14.: HMSC with parallel node

Connection Nodes Edges in HMSCs may only connect nodes. To facilitate readability of HMSCs, MSC-96 introduces connection nodes, whose sole purpose is to form the starting or ending point of edges within the graph. This helps to reduce the number of incoming and outgoing edges, in particular, of reference nodes. The graphic representation of a connection node is a non-filled circle.

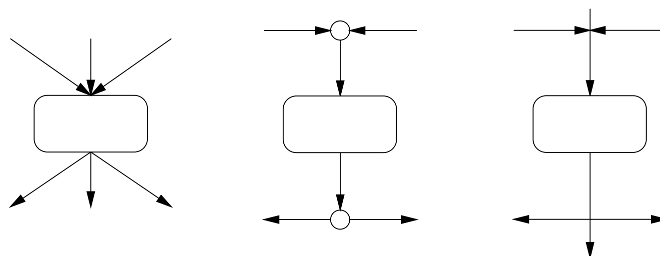


Figure 2.15.: HMSC fragments with connection nodes

Consider the HMSC fragments of Figure 2.15 as an example. The reference node to the left has three incoming and three outgoing edges, whereas the reference node in the middle has only one incoming and one outgoing edge. The two connection nodes serve as join and fork points, respectively.

To reduce the number of symbols needed in an HMSC further, MSC-96 allows us to omit the connection nodes, and to connect the lines directly, if no confusion can arise. An example appears in Figure 2.15, to the right.

Condition Nodes A condition node, graphically represented exactly as the condition symbol in simple MSCs, restricts the MSCs that may precede and succeed it in an HMSC. [IT96] defines a plethora of corresponding requirements. The semantics definition [IT98], however, assigns no meaning whatsoever to condition nodes.

A much simplified (and thus nonstandard) version of the restrictions stated in [IT96], applicable to the combination of a condition node and a reference node whose label is an MSC name, has two constituents:

- if there is an arrow from a condition node labeled C to a reference node labeled X, then X must start with a global condition labeled C;
- if there is an arrow from a reference node labeled X to a condition node labeled C, then X must end with a global condition labeled C.

We refer the reader to the literature mentioned above for the full-fledged set of restriction rules. Figure 2.16 shows an example of an HMSC with two condition nodes and one reference node; this HMSC obeys the requirements posed before.

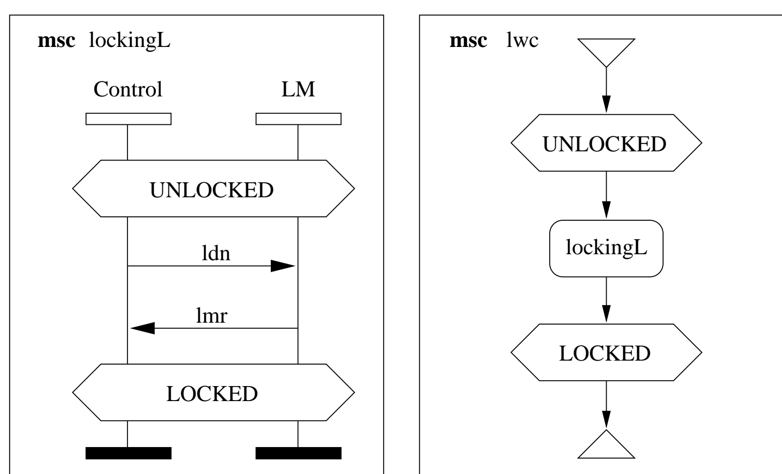


Figure 2.16.: HMSC with condition nodes

Mapping HMSCs to MSCs As we have mentioned in the previous paragraphs, the constructs provided by HMSCs correspond directly to similar constructs in simple MSCs. Therefore, we obtain the semantics of an HMSC simply by converting it into a corresponding plain MSC. In Chapter 4 we present such a conversion in detail.

Gates

References and HMSCs provide means for decomposing MSC specifications into smaller parts. Each part contains an interaction pattern we can study and understand on its own. This works fine as long as the sources and destinations of all message and ordering arrows are within the same MSC, as was the case in the examples we have introduced in Figures 2.10 through 2.12.

The situation changes, however, if the need for splitting arrows between MSCs arises. Consider, for instance, MSC G in Figure 2.17. If we want to decompose G into two parts, where one contains only instances W and X, and the other contains only instances Y and Z, we need a way of describing how the messages n and q continue outside the sub-MSC containing the corresponding send event.

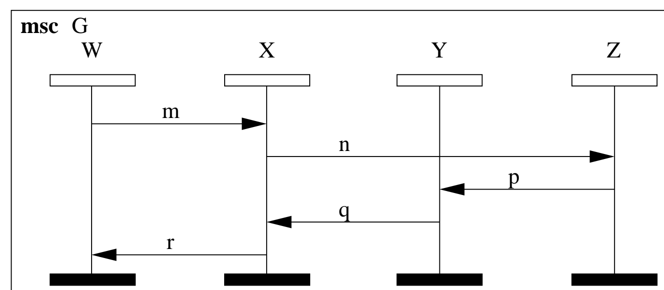


Figure 2.17.: How to decompose this MSC horizontally?

As a solution to this problem MSC-96 offers the concept of *gates*. Gates come in two forms: *message gates* and *order gates*. Intuitively, message gates allow us to specify “entry” and “exit” connection points for messages exchanged by instances within an MSC and their environment. Order gates serve a similar purpose: they allow us to include the events of a referenced MSC into the general ordering of its environment. In the following paragraphs we discuss message and order gates in more detail. We start out with message gates in combination with the most basic form of MSC reference expressions: MSC names. After that we also consider more complex reference expressions such as alternatives and loops. Finally, we show how message and order gates relate.

Message Gates Attaching a text label to the point of entry or exit, respectively, of a message arrow at the environment frame specifies a message gate. Figure 2.18 shows the definition of two gates: g and h. We call g a *message-out-gate*, because of its correspondence with an outgoing message. Analogously we call h a *message-in-gate*.

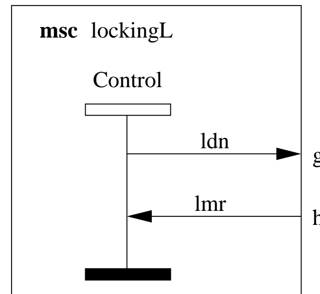


Figure 2.18.: MSC with definitions for message-out-gate g, and message-in-gate h

Gates and References The names of gates defined within an MSC reoccur in a corresponding MSC reference. To indicate how a message attached to a message-out-gate continues outside the referenced MSC, we specify the name of the corresponding gate at the reference symbol (this constitutes an *actual gate definition*), and start a message arrow from this actual gate to the desired destination; the message arrow must have the same label as in the referenced MSC. Similarly, to indicate how a message attached to a message-in-gate continues outside the referenced MSC, we specify the name of the corresponding gate at the reference symbol; we use this actual gate as the destination of a message arrow starting at the desired origin within the referencing MSC. The MSCs of Figure 2.19 show examples for both situations.

To yield a valid MSC reference the name of the actual gate and the name used in the gate's definition must coincide. In addition, the labels of the corresponding message arrows within the referencing and the referenced MSC must be identical. In particular, the gate and the message attached to it in the referencing MSC must both be present and attached to each other in the referenced MSC.

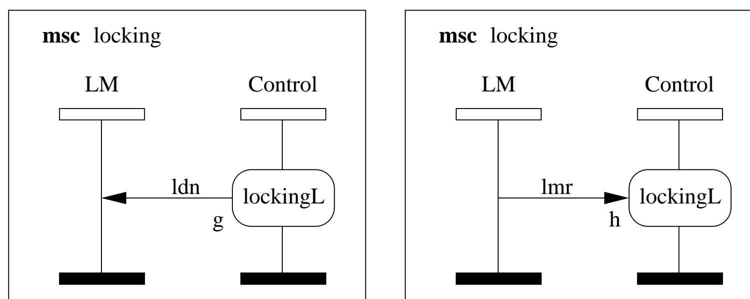


Figure 2.19.: MSC references with actual gate g and h, respectively

In both MSCs of Figure 2.19 we have made use of only one of the gates defined in Figure 2.18. This leaves open the question what happens to, say, MSC *lockingL*'s gate h and its corresponding message in Figure 2.19, left. MSC-96 defines that gates of a referenced MSC without connection to arrows in the referencing MSC propagate to the referencing MSC; thereby, all unconnected gates of referenced MSCs become gates of the referencing MSC.

As a result, the MSC of Figure 2.19, left, has an invisible message-out-gate *h*.

A message arrow that either originates from the environment frame, or has the environment frame as its destination without an explicit gate definition implicitly defines an anonymous message-in- or message-out-gate, respectively. Anonymous gates always propagate to the referencing MSC.

Message arrows whose one end connects to a gate may, with their other end, connect to any one of the following:

- an instance within the MSC; this is the situation we have illustrated in the examples of Figures 2.18 and 2.19.
- another gate of
  - a referenced MSC; Figure 2.20 shows an example of this kind of connection. This is the solution we need to decompose MSC *G* of Figure 2.17 horizontally.
  - the environment frame; an example of this also appears in Figure 2.20, where message *logDwn* connects to the environment via gate *f*. As mentioned above, this yields an anonymous gate definition.

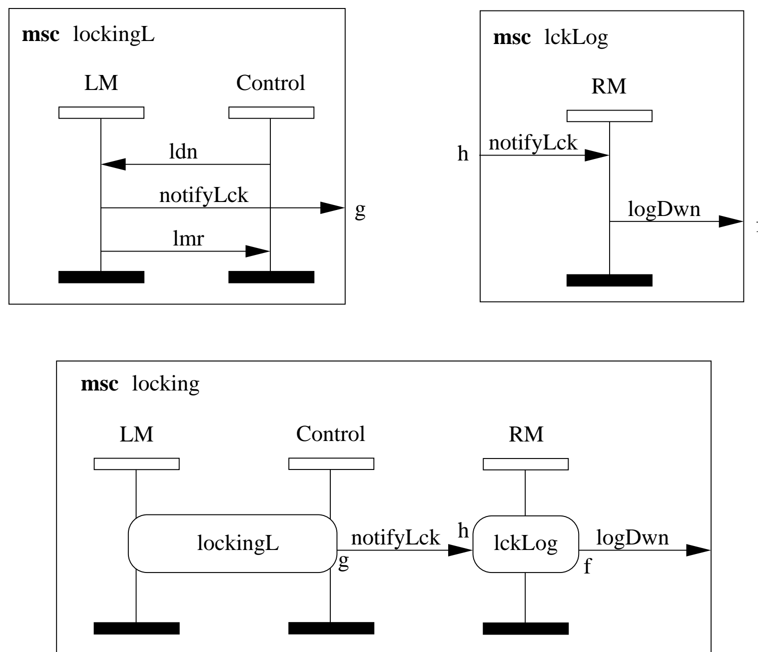


Figure 2.20.: Connecting references via gates

Gates and Reference Expressions As we have introduced above, reference expressions may be far more complex than simple MSC names. Examples are alternative- or parallel-reference expressions, such as (A **alt** B **alt** C), or (A **par** B) for given MSCs A, B, and C. MSC-96 extends the rules for valid MSC references in the presence of multiple MSCs within the same reference expressions in the following way: if the head or tail of a message arrow for message m connects to an actual gate of the reference expression, then at least one of the referenced MSCs must define the corresponding message-in- or message-out-gate, respectively, for message m. The set of gates of an MSC reference expression results from performing set union on the sets of gates of the referenced MSCs.

The combination of gates and non-basic reference expressions easily leads to complex and difficult to comprehend interpretations for MSCs. As an example consider the MSC fragment of Figure 2.21, left. Here, message m connects to gate g of reference expression (A **alt** B). The rules defined above state that A or B, but not necessarily both, must define message-in-gate g and the corresponding message m. If, say, B does not define this gate, then this MSC fragment specifies that m may or may not arrive at its destination. Similarly, the MSC fragment in Figure 2.21, right, specifies that message n is sent infinitely often, and arrives exactly once.

Figure 2.21.: Gates with **alt**- and **loop**-reference expressions

Gates and Inline Expressions Mostly for technical reasons and as a notational shorthand MSC-96 allows gates not only to attach to references, but also to inline expressions. However, because reference expressions also allow us to express alternatives, repetition, and parallelism, we can transform directly between an inline expression whose constituents are simple MSC references, and an MSC reference expression corresponding to the type of the inline expression. Therefore, there is no significant difference between the use of gates together with either reference or inline expressions; we refer the reader to [IT96, Ren99] for the details.

Order Gates Message gates allow us to specify how messages within and outside of a referenced MSC relate. Order gates serve a similar purpose. They allow us to specify how events – such as message send and receive, and action events – within and outside of a referenced MSC relate. Order gates provide a mechanism to integrate the event ordering within a referenced MSC into the event ordering of its environment. MSC-96's treatment

of order gates is entirely analogous to that of message gates. The only difference is that order gates relate ordering arrows, instead of message arrows. Everything else we have said about message gates transfers directly to order gates.

2.2.3. Miscellanea

For most of the syntactic elements we have introduced so far MSC-96 offers several presentation alternatives. Message arrows, for instance, may be bent to indicate message overtaking. The authors of [IT96, IT98, Ren99] discuss all of these alternative syntactic forms in detail.

In the remainder of this section we round up our treatment of MSC-96 by discussing the specification of lost and found messages, instance creation and stop, timers, and instance decomposition.

Lost and Found Messages

MSC-96 allows us to specify incomplete messages, i.e. messages for which a send event but no corresponding receive event, or a receive event but no corresponding send event occurs. A lost message never arrives at its destination and no instance ever sends a found message. The graphical representation for a lost message is a message arrow starting at an instance axis or at a message gate (a lost message has a corresponding send event), and ending at a filled black circle. The graphical representation for a found message is a message arrow ending at an instance axis or at a message gate (a found message has a corresponding receive event), and starting at a non-filled black circle. To indicate the intended receiver of a lost message or the expected sender of a found message, we may attach that instance's label to the circle of the lost or found message symbol, respectively.

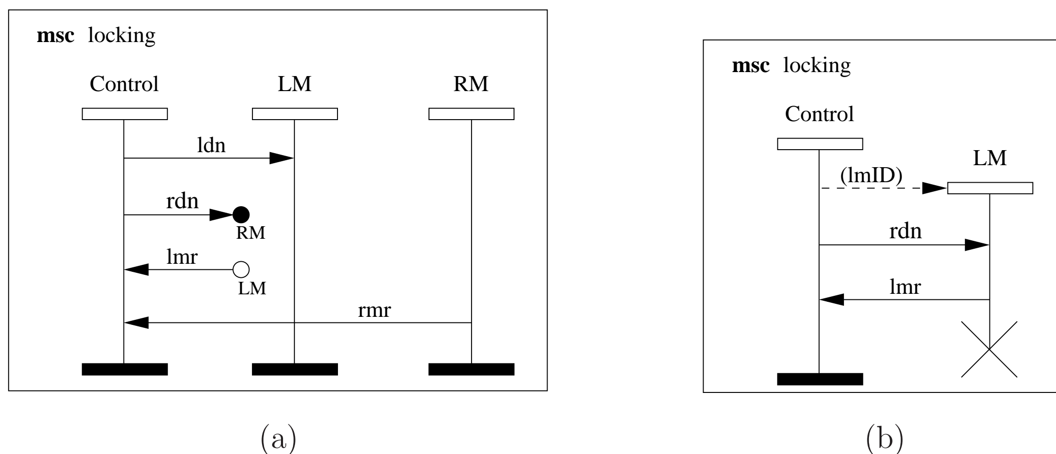


Figure 2.22.: Lost and found messages (a), and instance creation and stop (b)

As an example consider the MSC of Figure 2.22 (a). Here, *rdn* is a lost message whose intended receiver is *RM*, while *lmr* is a found message whose expected sender is *LM*.

Instance creation and stop

So far we have assumed that the instances depicted in MSCs existed at least long enough to partake in the specified interactions. We did not make assumptions about what brought an instance into existence, how long that existence lasted, and what caused the destruction of an instance. Although MSC-96 does not directly address all of these issues with its graphical syntax, it provides a means for specifying instance creation and stop. An instance can send and receive messages only after its creation and at most until its existence stops. Creation and deletion of any instance may occur at most once. The graphical representation of instance creation is a dashed message arrow, called "createline", from the creating instance to the head symbol of the created instance. The createline may but need not have a formal parameter list within parentheses. The idea is that the freshly created instance may use the values of these parameters for its own initialization. The instance head symbol of the freshly created instance need not be at the top of the MSC; rather, its typical vertical location is about the same as the vertical location of the createline's send event. To indicate termination of an instance graphically, we use a cross instead of an instance end symbol at the end of an instance axis.

In the example of Figure 2.22 (b) instance *Control* creates instance *LM*, and transmits the parameter *lmID* to it. After two interactions with its creator, *LM* stops. Note the asymmetry between instance creation and instance stop: creation of *LM* is under *Control*'s control, while *LM* itself controls its termination.

Timers

Timing constraints form an important part of specifications for technical systems. MSC-96 provides no quantitative notion of time, i.e. we can neither reference nor relate the exact times at which events occur. The only way of introducing time into an MSC specification is through using *timer events*. A timer is a clock that, once started, runs until a certain amount of time has elapsed; it then signals this fact to the instance which started the clock. Correspondingly, MSC-96 defines three timer events: set, reset, and time-out. Intuitively, "set" corresponds to initializing and starting a timer, "reset" sets the timer back to its initial value and restarts it, and "time-out" indicates the timer's expiration. The graphical representation of a timer set event is an hourglass symbol; it is connected by a line to the axis of the instance issuing the set event. The location at which the line ends at the instance axis indicates the relative order of the set event with respect to other events on the same axis. We can attach a label to the hourglass symbol to indicate the timer's name. This is useful if we use multiple timers within the same MSC document. Together with the timer's name we can informally indicate its duration, i.e. the initial value of the timer, in the form

of a text label in parenthesis. Because MSC-96 does not have any quantitative notion of time, such durations find no correspondence in the semantic treatment of timers; they have only informal documentational value. A timer reset event has a cross as its representation; again, a connecting line between the instance causing the reset and the cross indicates when the reset occurs. The only event caused by the timer itself is the timeout event. Its graphical representation is an hourglass symbol from which an unlabeled solid arrow emerges. The arrow's head connects to the instance axis on which the timeout event occurs. The position of the arrowhead determines when the timeout event occurs with respect to all other events on the same instance axis. Reset and timeout events may also carry the corresponding timer's name. All timer events with respect to a certain timer occur on a single instance axis; put another way, at most one instance relates to any timer in an MSC document. We refer the reader to [IT96, Ren99] for the syntactic details of timer specifications.

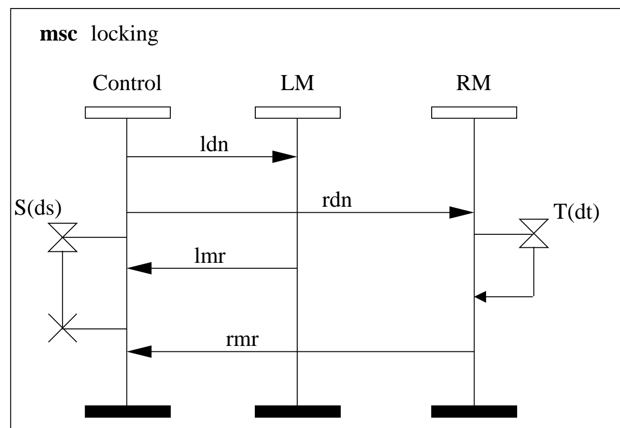


Figure 2.23.: MSC with timer events

The MSC of Figure 2.23 shows two timers and all possible timer events. Instance *RM* sets timer T with an initial value of dt. The timeout of T occurs before *RM* sends message *rmr* to instance *Control*. Instance *Control* sets timer S with an initial value of ds. Then it resets timer S after it has received message *lmr*.

Instance Decomposition

Besides references and gates MSC-96 provides another construct for introducing abstractions into plain MSCs: instance decomposition. A decomposed instance represents a set of instances that exchange messages among one another and with the environment. In the MSC containing the decomposed instance only the interactions with the environment appear; the instances it subsumes and their interactions remain hidden. This form of abstraction mainly serves to indicate structural containment relations. The idea is that the decomposed instance "contains" the instances into which it decomposes, be it logically or physically.

The graphical representation of a decomposed instance *X* is an instance axis whose head contains the keyword **decomposed**. Within the same MSC document we must also specify an MSC named *X* showing the instances and interactions subsumed by *X*. Figure 2.24 shows an example of a decomposed instance and its associated MSC. Here instance *CLS* decomposes into the three instances *Control*, *LM*, and *RM*. Messages *lck* and *rdy* are the only messages from MSC *CLS* that also appear in MSC *lckUC*.

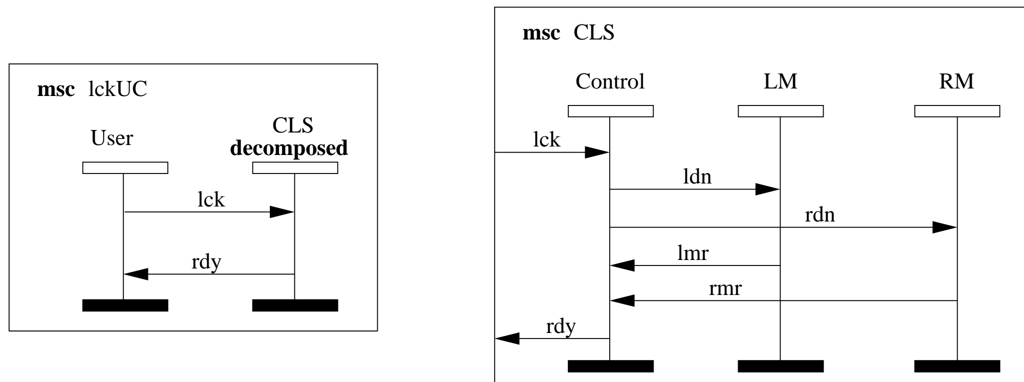


Figure 2.24.: MSC with decomposed instance *CLS*

The messages sent and received by a decomposed instance must reappear in its associated MSC as messages sent to and received by the environment, respectively; the source or destination of such messages may be any of the instances subsumed by the decomposed instance.

If the head of the decomposed instance indicates its corresponding MSC's name, after the keywords **decomposed as**, the instance's and the MSC's name may differ. MSC-96 poses two restrictions on any decomposed instance *X* and its corresponding MSC *Y*:

1. the event ordering of *Y* must respect the event ordering on *X*'s axis, i.e. if we consider only the events in MSC *Y* corresponding to events on *X*'s axis, the ordering of the former and the latter must coincide;

2. neither inline expressions nor reference symbols may occur on *X*'s axis.

2.3. Other MSC Dialects

Triggered by the increasing significance of interaction and collaboration specification several MSC-like notations have emerged over the past few years. In the following sections we give a coarse overview of some of these notations, and indicate their major application areas. The set of notations we cover comprises Object Message Sequence Charts, the UML's Sequence Diagrams, Extended Event Traces, Interworkings, Hybrid Sequence Charts, Live Sequence Charts, and MSC 2000.

Readers who prefer a succinct comparison of these MSC dialects to the example-oriented syntax presentation we give here, are referred to Section 2.4. Understanding of the material covered in subsequent chapters does not crucially depend on detailed knowledge about these syntaxes.

2.3.1. OMSCs

The authors of [BMR+96] introduced Object Message Sequence Charts (OMSCs) to describe interaction patterns in object-oriented software architectures. The basis of OMSCs are MSCs (see Section 2.2). Because of their intended application, however, the syntactic elements of OMSCs differ significantly from those proposed in the MSC-96 standard. Moreover, OMSCs do not provide any formal semantics; the topic of [BMR+96] is their use for explanatory purposes, neither their semantic foundation, nor their integration into the development process. Because the method call, and often also the yielding of control between caller and callee, is of major concern in object-oriented designs the developers of the OMSC notation offer specific syntactic support for these modeling tasks.

OMSCs provide the following modeling elements:

- labeled axes, representing part of the existence of an object (this corresponds to instance axes in MSCs);
- labeled arrows, indicating message exchange, method calls and returns (this corresponds to the message arrows within MSCs, although MSC-96 does not offer synchronous and return arrows);
- object activities, denoting phases where an object is active, i.e. where it executes the body of a method, function, or procedure (there is no corresponding concept in MSC-96);
- object creation arrows, denoting the point from which on an object exists (this corresponds to the createline of MSCs);
- object deletion symbols, denoting the end of an object's existence (this corresponds to instance stop in MSCs);

- process boundary symbols, denoting what set of objects belongs to what process of the system under design (there is no corresponding concept in MSC-96);

Elements from MSC-96 absent from OMSCs are the environment frame, conditions and actions, inline and reference expressions, coregions, general orderings, gates, lost and found messages, timers, and hierarchy. In particular, the lack of repetition and referencing constructs restricts the use of OMSCs to representing interaction scenarios of rather limited size. The major difference between MSC-96 and OMSCs is that OMSCs introduce the notion of control flow into the specification of interaction descriptions. This has gained them significant popularity especially among software engineers who use scenarios to describe sequences of method calls with corresponding returns. Because OMSCs support depicting focus of control, method calls, and returns, the authors of the UML [Rat97, RJB99] selected them as the basis for Sequence Diagrams (see Section 2.3.2). The lack of a formal semantics for OMSCs hinders their application in rigorous modeling environments.

In the following paragraphs we briefly introduce the graphical syntax of OMSCs.

Object Axis The graphical representation of (part of) an object's existence is a solid vertical line, which we call the object's axis. The object's name appears in a box to which the object axis connects from below. We call this box together with the label it contains the "object symbol". The object symbols of objects existing already before the depicted interaction scenario starts align at the top of the diagram. Figure 2.25 shows an axis for each of the three objects *Control*, *LM*, and *RM*.

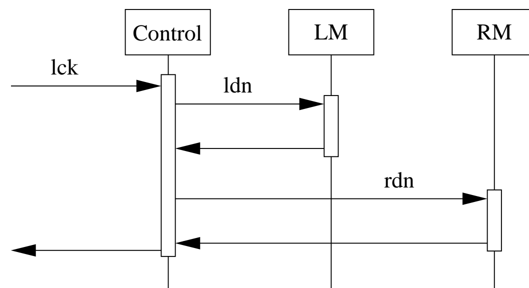


Figure 2.25.: OMSC for the "locking" use case

Arrows Arrows in OMSCs represent interaction among objects. The OMSC notation offers three kinds of arrows:

1. message arrows,
2. return arrows,
3. combined message and return arrows.

Message arrows are solid lines with full arrowheads, directed from the sender to the receiver of the corresponding message. The name of the message appears as a label above the arrow. Often, messages in OMSCs denote method calls; they can, however, also express asynchronous message exchange as in MSCs (see below). Return arrows, which have a much smaller arrowhead than message arrows, and are unlabeled, indicate the return corresponding to a method call (see below). Combined message and return arrows, which have arrowheads at both ends, represent both the call upon a method and the corresponding return; the large arrowhead indicates the direction of the call, the small arrowhead indicates the direction of the return. Figure 2.25 shows three messages representing method calls: *lck*, *ldn*, and *rdn*. An example of a combined arrow representing message *ldn* occurs in Figure 2.26.

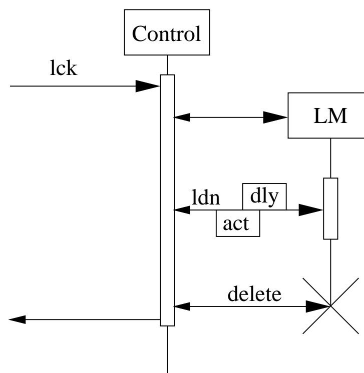


Figure 2.26.: OMSC with message/return parameters, and object creation/deletion

Method calls can have parameters; their labels appear as a comma-separated list in a rectangular box. The parameter box attaches from above to the message arrow. Similarly, returns may carry return values; their labels appear as a comma-separated list in a rectangular box attaching from below to the return arrow. Figure 2.26 shows the example of message parameter `dly` and return parameter `act`. To indicate transfer of object references [BMR+96] suggests to use italics for the corresponding parameter or return value label. To indicate the transfer of responsibility[1] for an object parameter or return value from the source to the destination of a message or return, respectively, [BMR+96] suggests to use boldface for the corresponding parameter or return value label.

Object Creation and Deletion An unlabeled message arrow whose arrowhead ends at another object symbol denotes object creation; the object from which the message arrow originates creates the object at whose symbol the arrow ends. The graphical representation of an object's deletion is a large cross at the end of the object's lifeline. To specify that another object causes the object's deletion we draw a message arrow from the former to

[1] [BMR+96] does not elaborate on the meaning of the term "responsibility"; one possible interpretation is that an object has responsibility for another if the former causes deletion of the latter.

the center of the cross. Otherwise the object on whose lifeline the deletion symbol appears controls its deletion itself. Figure 2.26 shows creation and deletion of object *LM* by object *Control*.

Object Activities The basic unit of interaction in most object-oriented systems is the method call. Typically, when an object calls upon a method, the receiver starts some computation, issues method calls itself, and processes the corresponding results. After these activities have finished the callee returns the result of the computation to the caller. To visualize phases of object activity, and to link the messages sent and received by an active object to fulfill a certain task or to handle a method call, the OMSC notation provides *object activities*; their graphical representations are vertical boxes on the object's axis. The activity's extent denotes how long the corresponding object is active. Consider again the OMSC of Figure 2.25. It shows three object activities. The longest belongs to *Control*. It shows that to process method *lck* object *Control* performs two method calls. Each of the called objects becomes active upon receipt of the corresponding method call, indicated also by means of activities.

Activities to Express Control in Sequential and Concurrent Systems In sequential systems at most one object can proceed, i.e. is active, at any point of time. The active object "has control" of the execution. If one object calls upon a method of itself or of another object, the caller typically yields control to the callee, so that the callee can process the request. The caller blocks until it receives control back from the callee. In such systems activities express the focus of control over time. Method calls start activities, and returns end them. The authors of [BMR+96] include the blocking phase, i.e. the time during which the caller waits for its callee, into the caller's activity. Put another way, they consider an object active even if it waits for the result of one of its own method calls.

In concurrent systems multiple threads of control may exist. Here, each activity can cover the entire axis of its corresponding object. The use of activities helps determine the amount of concurrency needed in the implementation of an interaction sequence. Figure 2.25, for instance, displays an interaction sequence where sequential method calls suffice as the basis for implementation: no object is active after performing the return corresponding to a method call. Otherwise, i.e. if there exists no binding between an object's activity, incoming method calls, and returns, this suggests using a multi-threaded object implementation.

Recursion If an object calls upon one of its own methods, we stack another object activation symbol slightly to the right of the one from which the method call originates. Figure 2.27 shows this via the example of *Control*'s *log* message.

Process Boundaries Thick angled lines can divide an OMSC into vertical regions. Each such region then corresponds to a separate process of the system under development. This

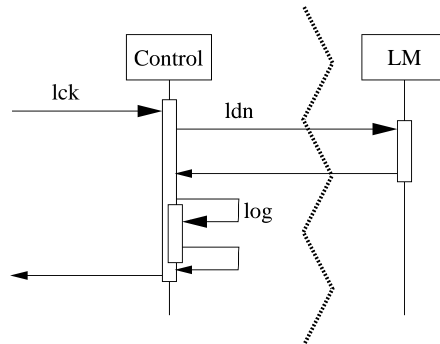


Figure 2.27.: OMSC with recursion and process boundary indicator

helps to introduce structure into the otherwise flat OMSCs. Moreover, the authors of [BMR+96] consider messages across process boundaries as asynchronous (with the exception of remote procedure calls). Other arrows denote method calls with the concrete type of interaction (synchronous, asynchronous, or either of the two) left implicit. Figure 2.27 indicates that objects *Control* and *LM* reside in different processes of the system.

2.3.2. Sequence and Collaboration Diagrams

Among the set of notations offered by the Unified Modeling Language (UML, cf. [Rat97, RJB99]) there are two whose major focus is component interaction: Sequence Diagrams (SDs), and Collaboration Diagrams (CoDs). While SDs stress the sequencing of messages in an interaction pattern CoDs stress the paths along which objects exchange messages.

Roughly, a CoD consists of a set of object symbols (rectangles, labeled with the object's name), and a set of arrows connecting communicating objects. These arrows represent communication links; the definition of these links may appear, for instance, in the form of association specifications within corresponding class diagrams. The messages exchanged by the objects label the communication links along which the messages occur; each message within a CoD carries a unique sequence number that determines its order within the interaction pattern; Figure 2.28 shows a simple example.

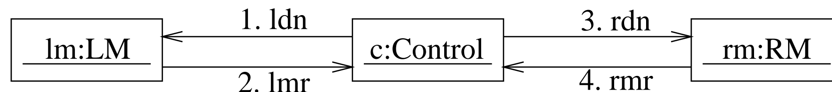


Figure 2.28.: CoD for the "locking" use case

Because the CoDs' graphical appearance aims more at clearly conveying structure instead of interaction sequences, we focus on SDs in the following. We refer the reader to [RJB99] for the details about CoDs.

The direct syntactic ancestors of the UML's SDs are the OMSCs we have described in Section 2.3.1. In fact, the SD's syntax is – up to a few minor changes and the omission of the process boundary symbol – a superset of the OMSC syntax; the major additions contributed by the UML are means for indicating alternatives, repetition, timing constraints, and state symbols within SDs.

SDs provide the following modeling elements:

- labeled axes, representing part of the existence of an object (this corresponds to instance axes in MSCs);
- labeled arrows, indicating message exchange, method calls, and returns (this corresponds to the message arrows within MSCs, although MSC-96 does not offer synchronous and return arrows);
- object activities, denoting phases where an object is active, i.e. where it executes the body of a method, function, or procedure (there is no corresponding concept in MSC-96);
- object creation arrows, denoting the point from which on an object exists (this corresponds to the createline of MSCs);
- object deletion symbols, denoting the end of an object's existence (this corresponds to object deletion in MSCs);
- guarded messages and conditional lifelines, expressing alternatives among interaction sequences (this corresponds to the **alt**-inline expression of MSCs);
- repetition boxes, denoting repeatable parts within an interaction sequence (this corresponds to the MSCs' **loop**-inline expression);
- timing constraints, specifying bounds on the duration of interaction sequences (this roughly corresponds to the timers of MSCs);
- state symbols, indicating state information about individual components during an interaction sequence (this corresponds to local conditions in MSCs, if we interpret the condition's label as state information);

Elements from MSC-96 absent from SDs are the environment frame, SD names, reference expressions, actions, coregions, general orderings, gates, lost and found messages, and hierarchy. In particular, the lack of referencing constructs restricts the use of SDs to representing interaction scenarios of rather limited size.

In the following paragraphs we briefly introduce the graphical syntax of SDs to the extent it differs from the one we have already discussed in the context of OMSCs in Section 2.3.1. We also omit some of the syntactic presentation options (alternative syntactic forms) contained in [Rat97] and [RJB99].

Object Axis, Lifeline The graphical representation of (part of) an object's existence is a dashed vertical line, which the authors of the UML call the object's lifeline. The object's name and class, separated by a colon, appear in a box to which the lifeline connects from below. Again, we call this box together with the label it contains the object symbol. The object symbols of objects already existing before the depicted interaction scenario starts align at the top of the diagram. Figure 2.29 shows an axis for each of the three objects *c* (of class *Control*), *lm* (of class *LM*), and *rm* (of class *RM*). The authors of [RJB99] use the term "classifier role" instead of "object symbol" to allow using lifelines as representations of roles rather than concrete objects. The idea is as follows: the interaction pattern corresponding to a lifeline is a placeholder for all objects that can "play this role", i.e. objects of the appropriate class. In particular, the same object may occur in different roles in different SDs; objects of the same class may occur in different roles in the same SD. We refer the reader to [RJB99] for a description of the technical details of this way of interpreting lifelines. For our purposes a simpler intuition suffices: a lifeline represents part of a concrete object's existence.

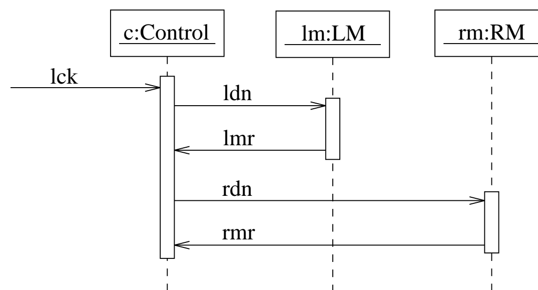


Figure 2.29.: SD for the "locking" use case

Arrows Arrows in SDs represent interaction among objects. The SD notation offers four kinds of arrows:

1. asynchronous message arrows,
2. synchronous message (method call) arrows,
3. "neutral" arrows (no indication of underlying communication paradigm),
4. return arrows.

Solid lines with half stick arrowheads, directed from the sender to the receiver of the corresponding message, represent asynchronous message exchange. The name of the message appears as a label above the arrow. Often, messages in SDs denote method calls; their representation is a solid line with a filled solid arrowhead. Return arrows, which consist of a dashed line and a stick arrowhead, indicate the return corresponding to a method call.

Figure 2.29 shows three messages representing method calls: *lck*, *ldn*, and *rdn*. Arrows with downward slope indicate that the transmission of the corresponding message takes time.

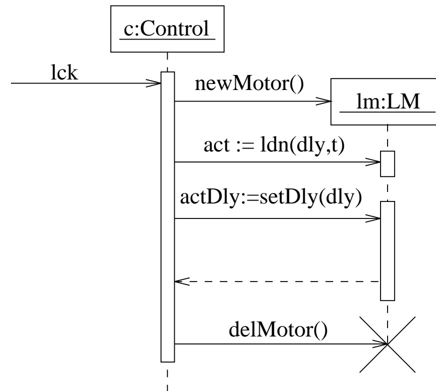


Figure 2.30.: SD with message/return parameters, and object creation/deletion

Method calls can have parameters; their labels appear as a comma-separated list within parenthesis after the method name on the corresponding arrow. Figure 2.30 shows two examples: method *ldn* has two parameters (`dly` and `t`), method *setDly* has one parameter (`dly`). Similarly, method calls may return values; the names of the attributes intended as the destination for these return values appear as a comma-separated list before the method name on the corresponding method call arrow; an assignment symbol ("`:=`") separates the attribute list and the method name. Figure 2.30 shows the example of the method call upon *ldn*; this call returns a value, which ends up in attribute `act`. At the end of an operation or method an explicit return arrow (consider the return of method *setDly* in Figure 2.30) from the callee to the caller may occur. If an activation (see below) indicates the duration of a method, the return is implicit and the corresponding arrow is optional in the SD (consider the implicit return of method *ldn* in Figure 2.30).

Object Creation and Deletion A message arrow whose arrowhead ends at another object symbol denotes object creation; the object from which the message arrow originates creates the object at whose symbol the arrow ends. The create-message's label indicates the method called at initialization of the newly created object. The graphical representation of an object's deletion is a large cross at the end of the object's lifeline with the same syntax as we have introduced for OMSCs in 2.3.1 (consider Figure 2.30 for an example).

Activations and Recursion The UML calls the OMSCs' "object activities" (cf. Section 2.3.1) "activations". Both terms describe the same underlying concept; their syntactic representation, a "tall thin rectangle" of an object's lifeline, is also equivalent. An activation indicates the focus of control within the system under consideration; it represents the duration of an action and the control relationship between the activation and its callers. The

UML's authors suggest to let activations start at the tip of an arrow representing a method call; the activations' ends should coincide with the corresponding return. If each object has its own flow of control, independent of other objects, they suggest to use activations to indicate operation durations. The activation of a recursive method is shown slightly to the right of its calling activation (see Figure 2.27 in Section 2.3.1). A shaded region within an activation indicates actual computation as opposed to having focus of control in a waiting or idling state. Figure 2.30 shows activations on the lifelines of objects *c* and *lm*.

Alternatives The UML's SDs can express alternatives within interaction patterns by means of two syntactic constructs: guarded messages and conditional lifelines. To indicate that an object sends exactly one of a set of possible messages we position the tails of all corresponding alternative arrows at the same point on the object's lifeline, and add a *guard* to the label of every arrow. The guard of a message is a text label within square brackets, indicating the occurrence condition for this message. Typically, the guard is a boolean expression over attributes of the sending object; the UML allows also other explanatory text. To ensure that exactly one message from the set of alternatives is enabled, all guards must be mutually exclusive; when, during execution of the system, the sender's control reaches the alternative message arrows, exactly one guard must yield true.

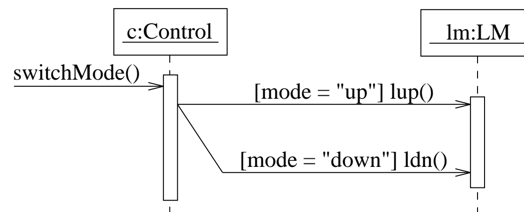


Figure 2.31.: SD with guarded message arrows

Figure 2.31 displays two guarded messages, representing the alternative between invoking method *lup* or *ldn*; the selection between the two alternatives depends on whether attribute `mode` has value `up` or `down`, respectively.

There is an obvious problem brought along by the use of guarded messages for expressing alternatives. By inspection of an object's lifeline alone we cannot determine whether the messages arriving at it belong to an alternative or are totally separate messages. We must, at least, determine whether their labels contain guards. But even if we find guards attached to the messages, we must still determine that these are messages from the same set of alternatives. The order of the arrowheads at an object's lifeline can easily become counterintuitive; does the order of two incoming arrows on the same lifeline denote sequentially arriving or alternative messages?

As a partial remedy, and to allow the developer to express different reactions of an object to alternative incoming arrows, the UML offers the concept of conditional lifelines. A lifeline may split into multiple ones to denote that the corresponding object can participate in the

depicted interaction sequences in alternative ways. Each lifeline represents one of a set of possible interaction scenarios in which the object can partake. Split lifelines must merge again within the SD.

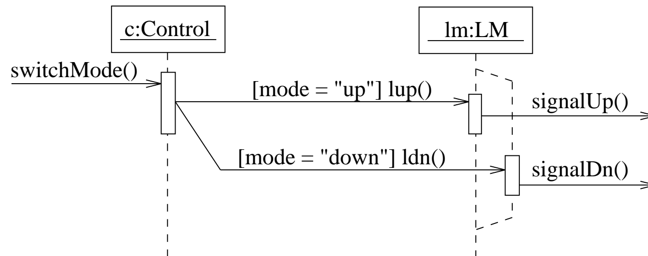


Figure 2.32.: SD with Guarded Message Arrows and Conditional Lifelines

As an example, consider the SD of Figure 2.32. Here, *lm*'s lifeline splits to represent different reactions according to the incoming method call. If *lm* receives message *lup* it sends message *signalUp*; if it receives message *ldn* it sends message *signalDn*.

Repetition For the specification of repeated interactions among a set of objects the UML offers only little syntactic support. We may enclose a repeatable sequence of messages, and add an iteration marker to indicate the repetition (cf. [Rat97, RJB99]). The UML does not specify *how* to enclose the sequence. The UML also leaves open the concrete syntax for the iteration marker.

Timing Marks The UML provides *timing marks* and *timing constraints* for the specification of the duration of message transmission or of interaction sequences as a whole. A timing mark is a letter put closely to the tail or head of a message arrow. The letter represents the time at which the corresponding send or receive event occurs. For an asynchronous message we only need to mark its sending time explicitly by a text label; implicitly, the same label plus an appended prime symbol (" ' ") represents the corresponding time of receipt (cf. [Rat97]). If `a` is the label at the tail of the asynchronous message arrow, then `a'` indicates the time of receipt for this message. A timing constraint can have one of the following two forms:

1. a predicate over expressions involving timing marks in curly braces in the left or right margin of the SD,

2. a vertical line whose extent covers all messages whose transmission must occur within the time specified by the constraint; two short horizontal lines bound the vertical line at its beginning and its end, respectively. The vertical line's text label indicates the allowed duration informally.

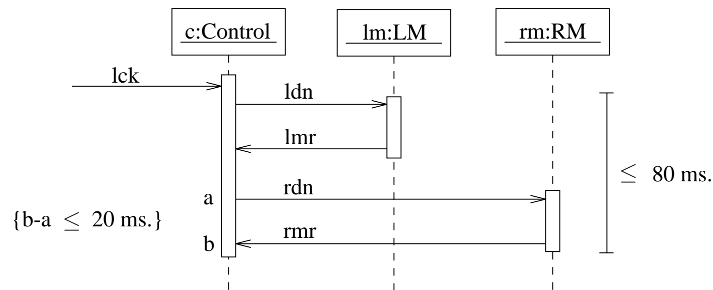


Figure 2.33.: SD with Timing Marks and Timing Constraints

Figure 2.33 shows examples for timing marks and timing constraints. `a` represents the sending time of message *rdn*, whereas `b` represents the time of receipt for message *rmr*. The two timing constraints in curly braces specify two conditions at the transmission times of the depicted messages. The sum of transmission times for messages *rdn* and *rmr* may not exceed twenty milliseconds. The message sequence from *ldn* through *rmr* must be complete within 80 milliseconds.

Note, however, that as for the other syntactic constructs we have discussed so far, the UML neither defines a formal mathematical semantics for timing marks, nor does it specify consistency criteria for the timing constraints within SDs and other system views, such as statecharts. Therefore, timing marks and timing constraints within SDs have annotational character only.

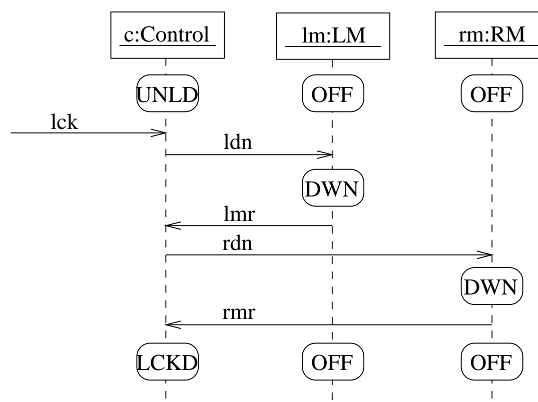


Figure 2.34.: SD with State Symbols

State Symbols The UML allows integration of state information for individual objects into SDs via the use of state symbols on the corresponding objects' lifelines. Syntactically state symbols in SDs equal those in statecharts (cf. [RJB99]); the graphical representation of a state is a rectangle with rounded corners, labeled with the name of the state. To indicate that an object is in a certain state before or after (part of) an interaction sequence,

we place an appropriately labeled state symbol on this object's lifeline; the position of the state symbol on the lifeline indicates when the object has assumed the corresponding state.

Figure 2.34 shows examples of state symbols on each of the depicted lifelines. Consider, for instance, the lifeline of object *lm*. Initially, *lm* is in state *OFF*. After having received message *ldn*, *lm* switches into its *DWN* state; once it has sent message *lmr* it changes back to state *OFF*.

2.3.3. EETs

Extended Event Traces (EETs, for short), are a much simplified variant of MSCs. Their origin is the specification of control oriented and embedded systems (see [SHB96]), and the specification of interaction architectures (see [BHKS97a]). EETs have evolved slightly differently in these two application domains; we concentrate on the variant presented in [BHKS97a] in the following.

EETs provide the following modeling elements:

- labels, indicating the name of an EET for referencing it in other EETs (this corresponds to MSC labels);
- labeled axes, representing part of the existence of a component (this corresponds to instance axes in MSCs);
- labeled arrows, indicating synchronous message exchange (synchronous message exchange is not a modeling element of MSC-96);
- choice boxes for referencing EETs and for the specification of alternatives (this corresponds to **alt**-inline and -reference expressions in plain MSCs, and to references and alternatives in HMSCs);
- repetition indicators for the specification of options and loops (this corresponds to the **opt**-inline expression, and the **loop**-inline expressions in plain MSCs and loops in HMSCs, respectively);
- an interleaving operator for the specification of independent interactions (this corresponds to coregions and **par**-inline expressions in simple MSCs, and parallel nodes in HMSCs);
- predicates for formulating properties over sequences of interactions (predicates are not a modeling element of MSC-96).

Elements from MSC-96 absent from EETs are asynchronous messages, the environment frame, conditions and actions, reference expressions, general orderings, gates, lost and

found messages, instance creation and stop, and timers. Furthermore, loops in EETs allow specification of finite repetitions only, whereas **loop**-inline expressions and loops in HMSCs also allow representation of infinite repetition.

In the following paragraphs we briefly introduce the graphical syntax and informal semantics of EETs. For a more elaborate discussion of these issues, and for the formal semantics of EETs we refer the reader to [SHB96, BHKS97a].

Component Axes and Arrows Similar to an MSC, an EET represents components by labeled axes. The label is the name of the component, the axis is a solid vertical line. A horizontal arrow denotes message exchange; its label is the name of the message together with a list of formal parameters within parentheses, if needed. As an example consider the EET of Figure 2.35, left. It shows two component axes, labeled *LM*, and *C*. These components exchange two messages: *ldn* and *lmr*. Message exchange in EETs occurs synchronously. Hence, each arrow denotes a single event. This allows us to determine the semantics of an EET without interleaving, alternatives, and repetition (see below) simply by reading the EET from top to bottom, recording the messages we encounter along the way. This yields for every EET, even for the more complex ones we discuss below, a set of message sequences as its semantics. For the EET in Figure 2.35, left, this set equals {<*ldn*, *lmr*>}[2]. Similarly, the semantics of the EET in Figure 2.35, right, equals {<*rdn*, *rmr*>}.

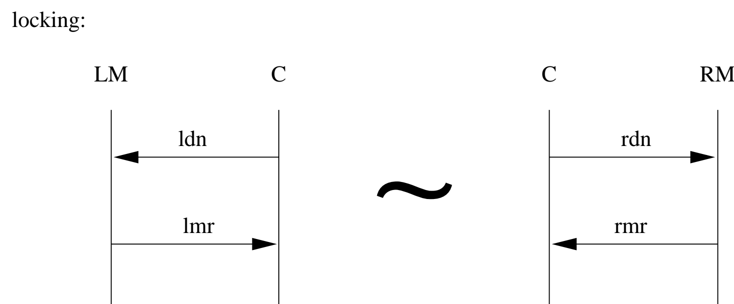


Figure 2.35.: EET *locking* with message exchange and interleaving

EET Names An EET may have a name for referencing it in other EETs. The name, followed by a colon, appears at the top left of an EET. The name of the EET in Figure 2.35 is *locking*. If an EET has no name, i.e. the EET is "anonymous", we assume that it has a globally unique implicit name. The two sub-EETs of Figure 2.35 left and right of the tilde symbol, respectively, are anonymous.

[2]We use comma-separated lists of messages within angular brackets to represent message sequences. For reasons of brevity we identify messages with their names in this section. In [BHKS97a] a message consists of sender, receiver, message name, and parameter list.

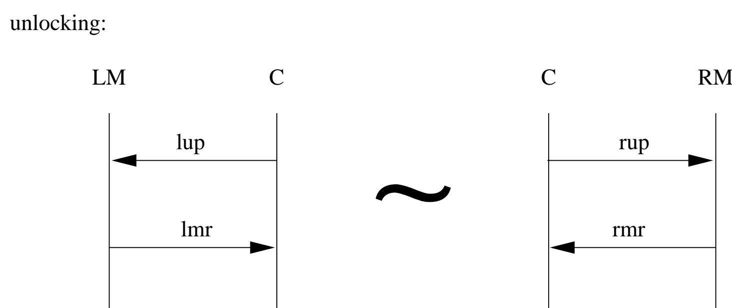


Figure 2.36.: EET *unlocking*

Interleaving To express simultaneous occurrence of messages, EETs offer the interleaving operator; this operator's graphic representation is the tilde symbol. As an example consider Figures 2.35 and 2.36. In each of these figures the interleaving operator connects two sub-EETs. The semantics of the interleaving of two EETs is the set of all possible interleavings of the message sequences represented by the operand EETs. The semantics of EET *unlocking*, for instance, is

$$\{ \; <lup, lmr, rup, rmr>, <lup, rup, lmr, rmr>, \\ <lup, rup, rmr, lmr>, <rup, lup, lmr, rmr>, \\ <rup, lup, rmr, lmr>, <rup, rmr, lup, lmr> \; \}$$

Each element of this set contains every message occurring in Figure 2.36 in the order given by the sub-EET from which it originates. The difference between two distinct elements from the set is the order of messages that originate from different operand EETs. MSC-96 assigns two events to each message, whereas EETs treat messages as atomic; besides this difference the interleaving operator achieves the same effect as the **par**-inline expression in a plain MSC, or a parallel node in an HMSC.

Environment Unlike MSCs, EETs provide no notion of environment frame; instead, we must model the environment explicitly as another system component. In Figures 2.37 through 2.39 we use component *ENV* to represent the environment.

EET Reference EETs have a referencing mechanism; the graphical symbol for an EET reference is a box that covers all component axes. The label of the box denotes the referenced EET. Figures 2.37 and 2.38 show examples of EET references. EETs cannot have cyclic references. Semantically, an EET reference corresponds to the substitution of the referenced EET for the reference symbol. Hence, we obtain the semantics of EET *U* by prefixing message *lck* to each element of EET *unlocking*'s semantics. The EET referencing mechanism subsumes the referencing mechanisms of plain MSCs and HMSCs.

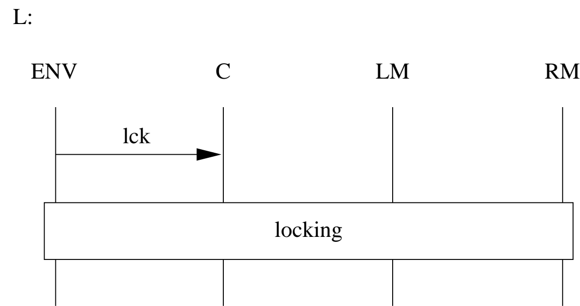


Figure 2.37.: EET L with reference to EET *locking*

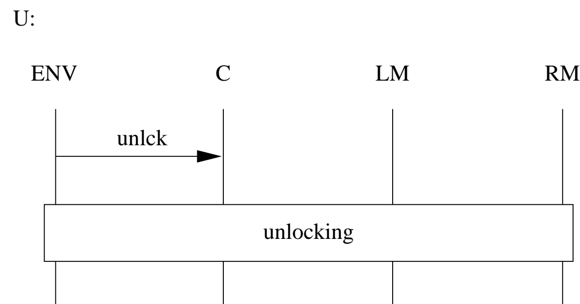


Figure 2.38.: EET U with reference to EET *unlocking*

Alternatives To specify the alternative among a set of message sequences the EET syntax allows us to label a reference box not only with a single EET name, but also with a set thereof. Consider EET *LorU* of Figure 2.39 as an example. Here the label of the reference box is the set $\{L, U\}$. This means that wherever this reference box appears, we have the choice of substituting either EET L, or EET U. This becomes particularly interesting in combination with repetition.

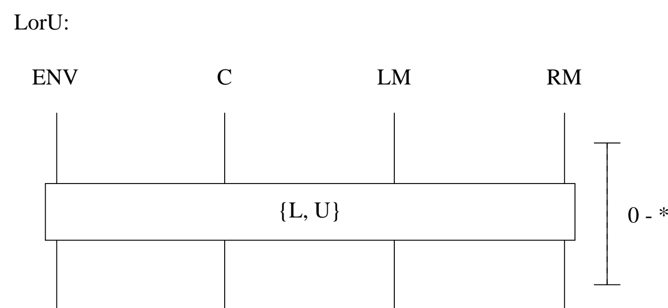


Figure 2.39.: EET *LorU* with alternative and repetition

Repetition The repetition operator provided by the EET syntax resembles the **loop**-inline expression from MSC-96. Its graphic representation, called "repetition indicator", is a vertical line bounded from above and below by two short horizontal lines. The length

of the repetition indicator determines the scope of the repetition. We call the syntactic elements of an EET covered by a repetition indicator the indicator's operand. The indicator's label determines the possible number of repetitions. The label may have one of two syntactic forms. The first is "$l - u$", where $l, u \in \mathbb{N}$. Then the semantics of the repetition indicator is the set of message sequences obtained by repeating the operand at least l and at most u times. Hence, the label "0 – 1" corresponds to the **opt**-inline expression of MSC-96. The second syntactic form for the label is "l – *', where $l \in \mathbb{N}$. Then the semantics of the repetition indicator is the set of message sequences obtained by repeating the operand any arbitrary finite number of times, but at least l times.

Consider Figure 2.39 as an example. Here the operand of the repetition indicator is the alternative box. The label indicates that the operand may occur any finite, nonnegative number of times. Because the operand is an alternative, in each repetition a different choice is possible.

Property Specification The major benefit of EETs is their simple, yet precisely defined semantics [SHB96, BHKS97a]; the meaning of each EET is a set of finite message sequences. Because of the assumption of synchronous message exchange, and due to the limited set of modeling elements, we can derive this set in an intuitive, direct fashion from the graphical representation. Moreover, in contrast to MSC-96 (cf. [IT96, IT98]), the EET semantics given in [BHKS97a] integrates the formal parameters of messages into the semantic treatment. This allows us to use predicates to formulate properties of interaction sequences; this is particularly interesting in cases where we cannot represent these properties by means of the EETs' graphical syntax. Examples of such properties are relations between parameter values, and the number of occurrences of certain messages with respect to others. We have exploited this potential in [BHKS97a] for the precise specification of interaction architectures, such as the "Observer" and "Pipes and Filters" patterns (cf. [GHJV95, BMR+96]).

An example of a property best formulated as a predicate for EET *LorU* is "between any two *lck* messages from the environment an *unlck* message occurs".

2.3.4. Interworkings

The origin of Interworkings is, similar to that of MSCs, the specification of interactions in telecommunication systems [MR96, MR94]. In fact, Interworkings are one of the direct predecessors of MSC-96 (cf. [Ren99]). In contrast to messages in MSCs, interactions occur synchronously within Interworkings; they do not model any delay between a message output and the corresponding input. Interworkings have a formal semantics. The authors of [MR96] give it in a process-algebraic setting, and use the semantics to define composition operators for Interworkings, such as sequencing and merge (see below).

Interworkings provide the following modeling elements:

- labels, indicating the name of an Interworking for referencing it in accompanying text (this corresponds to MSC labels, although MSC-96 allows references to occur within MSCs, not only within accompanying text);
- labeled axes, representing part of the existence of an entity (this corresponds to instance axes in MSCs);
- labeled arrows, indicating synchronous message exchange (synchronous message exchange is not a modeling element of MSC-96);
- a sequencing operator, yielding the sequential composition of two argument Interworkings (this corresponds to weak sequential composition in MSC-96);
- a merge operator for the specification of the parallel composition of two argument Interworkings. The resulting Interworking identifies instances that have the same labels, and messages that have the same sender, receiver, and label in the argument Interworkings (there is no corresponding construct in MSC-96);
- a formal refinement relation, relating an abstract and a more concrete Interworking (this corresponds to instance decomposition in MSC-96, although the definition of the latter is less formal).

Elements from MSC-96 absent from Interworkings are asynchronous messages, the environment frame[3], conditions and actions, reference expressions, general orderings, gates, lost and found messages, instance creation and stop, and timers. In particular, there is no means for expressing alternatives and repetition, and no referencing mechanism.

In the following paragraphs we briefly introduce the graphical syntax and informal semantics of Interworkings. For a more elaborate discussion of these issues, and for the formal semantics of Interworkings we refer the reader to [MR96].

Interworking Label, Entity Axis, and Message Arrow An Interworking may have a name; if the name exists it occurs at the top left of the diagram. The representation of (part of) an entity's existence is a solid vertical line, labeled with the entity's name. Horizontal arrows in Interworkings, directed from the sender of a message to its receiver, denote communication between entities. Message transmission occurs instantaneously, i.e. Interworkings model no delay between the sending of a message and its receipt. Time runs from top to bottom in Interworkings along each entity axis; the messages sent and received by a single entity form a total order. There exists, however, neither a global clock, nor a quantitative time scale along the entity axes. Hence, an Interworking defines a partial order on the messages it contains.

[3]The environment frame has no semantics in Interworkings; it serves to separate diagrams within the same picture.

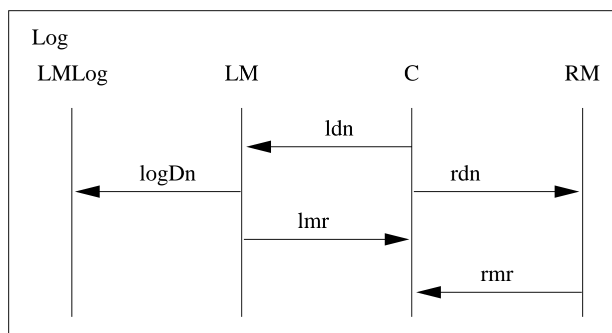


Figure 2.40.: Interworking *Log* with ordered and unrelated synchronous messages

As an example, consider Interworking *Log* in Figure 2.40. Because the position of the arrowheads and -tails on *LM*'s entity axis induce a total order on the corresponding messages, we obtain that *ldn* precedes *logDn*, which, in turn, precedes *lmr*. Similarly we obtain from the total ordering of messages along *C*'s axis that message *ldn* precedes *rdn*; *rdn* precedes *lmr*, which, in turn, precedes *rmr*. Because of the absence of a global clock we cannot assign an ordering to the messages *logDn* and *rdn*. Thus, *Log* represents the following set of interaction sequences:

$$\{ \quad <ldn, logDn, rdn, lmr, rmr>, <ldn, rdn, logDn, lmr, rmr> \quad \}$$

The developer may use multiple Interworkings to describe the interaction among the system's entities. The information contained in an Interworking is complete with respect to the entities referenced by the Interworking. This means that the interaction sequence between any two entities A and B in an Interworking proceeds as depicted; no other interaction sequence between A and B can occur interleaved to the depicted one. This rule does not restrict interactions between an entity occurring in an Interworking I, and any other entity absent from I.

Two operators allow the developer to compose Interworkings vertically and horizontally. These operators define how the interaction sequences among entities depicted in different Interworkings relate. We discuss both operators in turn, below.

Sequencing The sequencing operator yields the vertical composition of two Interworkings. The composite's set of instance axes is the union of the sets of instance axes of the operands. To obtain the composite's interaction sequences we copy the messages from the first operand, in the order they appear there, to the composite. Then, below the last message that stems from the first operand, we copy the messages from the second operand, in order.

As an example, consider the Interworkings *LDN* and *LMR* from Figure 2.41. Their sequencing occurs as Interworking *L* in Figure 2.42.

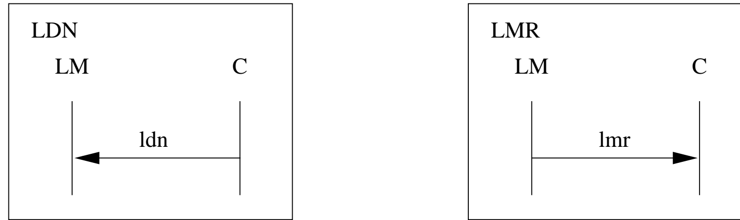


Figure 2.41.: Interworkings *LDN* and *LMR*

Sequencing does not necessarily yield sequential composition of the operand Interworkings. If the operands' sets of entities are disjoint, for instance, the messages from the two Interworkings are independent as we have discussed already, above. Hence, sequencing yields sequential composition of certain parts of the operand's interaction sequences at most if the operands have at least one entity in common that participates in interactions in both Interworkings.

Merge Intuitively, the merge of two Interworkings corresponds to their horizontal composition. For operand Interworkings having no entities in common this corresponds to parallel composition; the semantics of the composite is the set of all interleavings of the interaction sequences represented by the operands. For operand Interworkings that do have entities in common the interactions occurring between the common entities in one operand must appear in the exact same form in the other operand; no other interactions among such entities may occur in either operand. Thus, the merge identifies common entities and their interactions in the operand Interworkings. Put another way this means that the operand Interworkings synchronize on the interactions of common entities. Operand Interworkings violating this restriction are inconsistent. Their merge results in a deadlock. Messages whose sender or receiver occurs in one operand, but not in the other, get interleaved in the result of the merge.

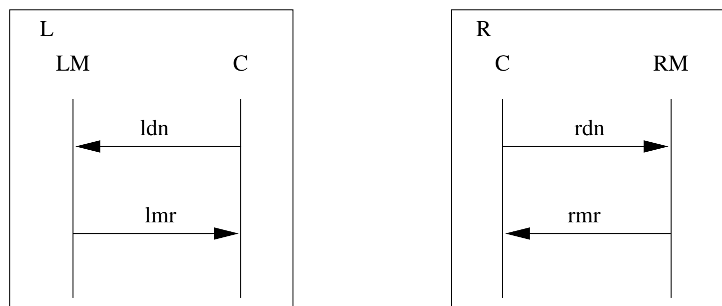


Figure 2.42.: Interworkings *L* and *R*

According to this definition the Interworkings *LDN* and *LMR* from Figure 2.41 are inconsistent. The Interworkings *L* and *R* from Figure 2.42, on the other hand, are consistent. The

merge of these two operands has the following set of interaction sequences as its semantics:

$$\begin{aligned}\{\quad & <ldn, lmr, rdn, rmr>, <ldn, rdn, lmr, rmr>,\\ & <ldn, rdn, rmr, lmr>, <rdn, ldn, rmr, lmr>,\\ & <rdn, ldn, lmr, rmr>, <rdn, rmr, ldn, lmr> \quad \}\end{aligned}$$

Refinement The authors of [MR96] provide a formal definition of Interworking refinement, which loosely corresponds to the notion of instance decomposition in MSC-96. Intuitively, Interworking A refines Interworking B if

- the set of entities from B is a subset of A's entity set, and
- for every entity x occurring in B there exists a set of entities in A whose elements together assume x's interaction responsibilities.

More precisely, every entity occurring in A must map to exactly one entity in B. Multiple entities from A, say e_0 through e_{n-1}, may map to the same entity, say e, of B. In such a situation we say that e_0 through e_{n-1} constitute e's refinement in A. Every message from B must reappear in A in the same ordering with respect to the other messages from B. If e sends or receives a message in B then any one of e_0 through e_{n-1} must send or receive this message in A. Entities e_0 through e_{n-1} may exchange arbitrary additional messages in the refinement.

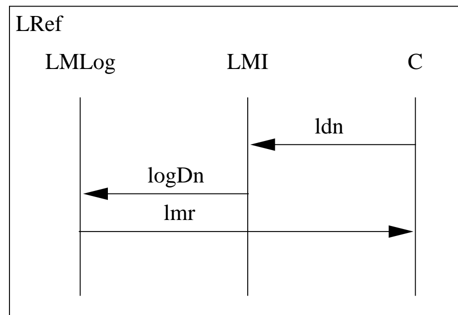


Figure 2.43.: Interworking *LRef*, a refinement of Interworking L

According to these rules, Interworking *LRef* from Figure 2.43 refines Interworking L from Figure 2.42. Entities *LMLog* and *LMI* constitute *LM*'s refinement. The refining entities exchange the additional *logDn* message; *LMI* now acts as the receiver of message *ldn*, whereas *LMLog* acts as the sender of message *lmr*.

2.3.5. HySCs

In [GKS99a] we have used the syntax of MSC-96 for the specification of complete interaction behavior in hybrid systems. The corresponding semantics differs significantly from the one of MSC-96; the semantics of Hybrid Sequence Charts (HySCs) bases on a shared variable communication model for clock-synchronously operating components.

We mention this MSC dialect for two reasons. First, it assigns a formal semantics to condition symbols (as predicates over discrete and continuous component variables), and provides access to a quantitative notion of time in MSCs (via appropriate differential equations). Second, it makes the concept of *preemption* syntactically and semantically accessible to MSC specifications.

Preemption has many practical applications in the context of reactive systems. Consider a communication protocol where any participant may terminate an ongoing communication at any time; a typical example is a telephone call, which any party can terminate by hanging up the phone. Capturing such protocols by means of MSCs without an appropriate syntactic and semantic preemption concept is a laborious task. It requires adding a plethora of interaction scenarios, each of which represents one of the possibly many situations in which the preemption may occur.

In Chapter 4 we discuss the syntactic and semantic treatment of preemption in more detail.

2.3.6. LSCs

The authors of [DH99] have introduced Live Sequence Charts (LSCs) as an extension of the ITU's MSC-96 standard for plain (basic) MSCs; [DH99] does not address HMSCs. The major addition here is integrating the specification of liveness properties into the MSC notation. To that end the authors relate LSC specifications to system runs. A system run, in their approach, is an infinite sequence of snapshots, where a snapshot consists of the set of current events (being either synchronous or asynchronous sends or receives between components or between a component and the environment), and an assignment of values to all variables of the system.

The authors associate liveness with four syntactic elements of MSCs:

- plain MSCs as a whole,
- locations (segments on an individual instance axis),
- messages,
- conditions.

We can specify the occurrence of each of these syntactic elements as either mandatory or provisional (but not both) in LSCs. In the following paragraphs we briefly address each of these items in turn.

Plain MSCs The authors of [DH99] associate with each plain MSC a mode: universal (mandatory) or existential (provisional). Any system run must satisfy a universal LSC, whereas an existential LSC requires only at least one such run to exist. This distinction, which also occurs in [Kle98, KGSB99, Krü99b], separates the traditional interpretation of MSCs as scenarios from complete behavior specification. Syntactically the frame around an LSC determines whether the LSC is a universal (solid-line frame) or an existential (dashed-line frame) one.

Location A dashed line segment on an instance axis denotes that, during a run of the system, the instance under consideration *need not* move beyond this line segment. A solid line segment indicates that during a system run the instance *must* move beyond this line segment. Intuitively, the distinction between these two cases allows the developer to specify local progress requirements in the global context of an interaction sequence.

Message A dashed arrow denotes that the corresponding message, if sent, may or may not arrive at its destination. Solid arrows indicate that a sent message must arrive. In addition to asynchronous message exchanges (indicated by open ended arrowheads) known from MSC-96, the authors allow synchronous message exchange (indicated by solid arrowheads).

Condition The authors of [DH99] associate state predicates with conditions occurring in LSCs. If, during a run of the system, execution reaches a provisional condition evaluating to false, then the LSC allows arbitrary behavior from this point onward. If, during a run of the system, execution reaches a mandatory condition evaluating to false, the modeled system halts. If the condition holds in either case then execution simply progresses beyond the condition. Provisional conditions appear syntactically as MSC-96 condition symbols with dashed outlines; the graphical representation of mandatory conditions are MSC-96 condition symbols with regular outlines.

Operational LSC Interpretation Given the interpretation of the above syntactic elements the authors define the semantics of an LSC operationally as follows. The LSC has a corresponding transition system (automaton) with three states labeled *active*, *terminated*, and *aborted*. Given a run of the system the automaton remains in state *active* while the run evolves as described by the LSC, according to the above-mentioned progress properties. Once the run reaches a mandatory condition evaluating to false, the automaton switches to and remains in state *aborted*, allowing only stuttering steps from this point in time on. If either the LSC describes a prefix of the run, or a provisional condition within the LSC does not hold, the automaton switches to and remains in state *terminated*, allowing arbitrary behavior from this point in time on.

The automaton accepts a run as conformant to the LSC if and only if either it reaches state *terminated*, or it stays in state *active* forever after all progress conditions have been met.

The authors hint at, but do not elaborate on, the representation of repetition constructs; similarly, they only briefly mention "forbidden" scenarios, i.e. scenarios that must not occur in a system run. They also introduce the concept of "subcharts", which are LSCs occurring within other LSCs. The idea is that subcharts, in which a provisional condition does not hold, terminate, and execution continues in the "parent" LSC outside the subchart; one application, therefore, of subcharts together with provisional conditions is to specify preemption in LSCs. However, the precise semantics of these subcharts does not appear in [DH99].

2.3.7. MSC 2000

MSC 2000 (cf. [IT99]) is ITU's follow-up on MSC-96. The new recommendation differs from MSC-96 mainly in the following areas:

- *control flow:* MSC 2000 offers syntax for specifying control flow, comparable to what OMSCs (cf. Section 2.3.1) and the UML's SDs (cf. Section 2.3.2) provide under the name "activities" and "activations", respectively. Furthermore, MSC 2000 allows indicating whether an arrow denotes a method call or the corresponding return.
- *better integration of conditions:* conditions in MSC 2000 have a meaning beyond denoting labels for "pasting together" MSCs; they now can express requirements at component and system states, and guide the selection of alternatives.
- *quantitative notion of time:* the duration of timers can now be specified formally. In addition, MSC 2000 enables the specification of timing constraints on the occurrences of events. The syntax and intended meaning is similar to what the UML's SDs provide (cf. Section 2.3.2).
- *data specification:* MSC 2000 assigns meaning to data specifications in MSCs; examples are messages with data parameters, and local instance variables.

We refer the reader to [IT99] for further details. So far, there exists no formal semantics comparable to the one of MSC-96 (cf. [IT98]) for MSC 2000.

2.4. Comparison and Prospective Enhancements

In Sections 2.2 and 2.3 we have studied the syntax and informal semantics of several MSC dialects. These dialects differ mainly in the underlying communication models (asynchronous vs. synchronous, or combinations thereof), the scope of the notation (simple, finite scenarios only vs. complete system behavior), and the degree of semantic foundation.

Tables 2.1 and 2.2 summarize the similarities and differences of the various dialects. Entries of the form "+" and "-" indicate the presence or absence, respectively, of a certain feature. "(+)" and "(-)" mean that the syntactic or semantic support for the feature is limited or very limited, respectively.

Table 2.1.: Comparison of MSC dialects (part I)

	MSC Dialect							
Feature	MSC-96	MSC 2000	OMSCs	SDs	EETs	IWs	HySCs	LSCs
Communication								
asynchronous	+	+	+	+	-	-	-	+
synchronous	-	-	+	+	+	+	-	+
shared variables	-	-	-	-	-	-	+	-
Composition								
bounded finite repetition	+	+	-	(-)[a]	+	-	-	(+)[b]
unbounded finite repetition	-	-	-	(-)[a]	+	-	+	(+)[b]
infinite repetition	+	+	-	(-)[a]	-	-	+	(+)[b]
non-guarded alternatives	+	+	-	-	+	-	+	-
guarded alternatives	-	+	-	+	-	-	-	+
parallel composition	+	+	-	(+)[c]	+	+	(+)[d]	(+)[d]
overlapping operands	-	-	-		-	+	-	-
Structuring								
gates	+	+	-	-	-	-	-	-
referencing	+	+	-	-	+	-	+	+
HMSCs	+	+	-	(-)[e]	-	-	+	-

[a] The SDs' syntax for general repetition (involving more than one repeated message) is not specified formally in [Rat97, RJB99].
[b] [DH99] omits the corresponding formal definitions.
[c] Only for individual messages.
[d] Only via coregions or for MSCs with disjoint instance sets.
[e] The UML's activity diagrams (cf. [RJB99]) are conceptionally very similar to HMSCs.

Table 2.2.: Comparison of MSC dialects (part II)

	MSC Dialect							
Feature	MSC-96	MSC 2000	OMSCs	SDs	EETs	IWs	HySCs	LSCs
Miscellanea								
control flow: method call/return	-	+	+	+	-	-	-	-
instance creation and stop	+	+	+	+	-	-	-	(+)[f]
actions	+	+	-	-	-	-	-	-
process boundaries	-	-	+	(-)[g]	-	-	-	-
preemption concept	-	-	-	-	-	-	+	(+)[h]
Semantics								
formal semantics	+	-	-	-	+	+	+	+
integration of component states	-	(+)[i]	-	(+)[j]	-	-	+	+
distinction between partial and complete behavior	-	-	-	-	-	-	-	+
formal refinement notions	-	-	-	-	-	(+)[k]	-	-
quantitative notion of time	-	(+)[i]	-	(+)[j]	-	-	+	-

[f] [DH99] omits the corresponding formal definitions.
[g] The notion of "swimlanes" (cf. [Rat97, RJB99]) could serve as a process boundary indicator.
[h] There is no explicit syntactic support for preemption in LSCs; however, there are combinations of mandatory LSCs with provisional sub-charts that, together, can model preemption.
[i] MSC 2000 has no formal semantics yet; therefore, condition symbols and timing constraints have annotational value only.
[j] SDs have no formal semantics; therefore, state markers and timing constraints have annotational value only.
[k] For structural refinement only.

Comparison

Clearly, MSC-96 and its "update" MSC 2000 provide the most elaborate syntax. MSC-96 offers notation for the specification of both finite and infinite behavior; it also allows description of alternatives and concurrency. The referencing concept enables structured

presentations of interaction sequences. High-Level MSCs hide the details – with respect to the components and messages involved – of a composite interaction protocol. MSC-96 has a precise, formal semantics; unfortunately, this semantics is purely event-oriented, and does not take states of individual components into account. As a consequence the specification of the precise conditions under which an interaction sequence is optional or inevitable is, in general, impossible with MSC-96. Moreover, as we will describe in Chapter 4, the use of "weak sequential composition" (cf. Section 2.2.2) as a fundamental composition form for MSCs can lead to unintuitive MSC specifications. Message parameters are not considered in the formal semantics of MSC-96; this makes the formulation of data-oriented behavior aspects, as we have discussed it in the context of EETs (cf. Section 2.3.3), impossible. MSC 2000 addresses these deficits; however, there is – as of now – no formal semantics definition for MSC 2000.

OMSCs and Interworkings support specification of finite interaction scenarios without alternatives and repetition only. They are not intended as description techniques for complete component behavior. Still, the features of these notations complement those of MSC-96. OMSCs introduce syntax for indicating method calls and returns, as well as process boundaries. Interworkings base on synchronous message exchange, in contrast to the asynchronous communication model underlying MSC-96. The notion of structural refinement is formally defined for Interworkings, whereas MSC-96's instance decomposition has no formal semantics. Moreover, the semantics of an Interworking is closed in the following sense: it excludes any further occurrence of communication between the depicted components via messages that appear explicitly in the Interworking. This is a step towards considering MSCs as specifications of complete interaction behavior.

SDs improve on OMSCs by adding syntax for alternatives and – to some extent – also for repetition. Timing constraints are also modeling elements of SDs. Because they lack an explicit referencing mechanism SDs are restricted to small-scale specifications. Yet, the state symbols provided by SDs as part of object lifelines indicate a tighter coupling between interaction- and state-oriented description techniques than what is offered by MSC-96, OMSCs, Interworkings, and EETs.

EETs provide synchronous communication, alternatives, repetition, parallel composition, and referencing as modeling elements; this already allows quite elaborate interaction specifications. In addition, EETs have a formal semantics, which enables their integration into formal development environments (cf. [BHS99, HMS+98, HSS96, HSSS96]). Yet, their interplay with state-oriented description techniques has been left open in [SHB96, BHKS97a].

HySCs target the specification of interaction in hybrid systems; they use the syntax, but not the semantic model of MSC-96. HySCs make heavy use of condition symbols. The latter capture the continuous aspects of system behavior, whereas interactions indicate discrete events. This increased semantic integration of condition symbols is one of the major contributions of HySCs; the other is the addition of syntax and semantics for preemption specifications in MSCs.

LSCs use a subset of MSC-96 as the basis for interaction specifications. They stress the distinction between optional and mandatory behavior to allow specification of progress or fairness aspects in MSCs. Because the developer must watch this distinction for every modeling element (messages, axes, and MSCs as a whole), LSC specifications are rather complicated to construct, to communicate and to handle methodically. Still, the work of [DH99] assigns a formal meaning to condition symbols, and thus improves on MSC-96.

Discussion

In Chapter 1 we have pointed out the importance of several features of an adequate MSC notation. One such feature is the existence of a formal semantics, integrating the notions of interaction and state; this facilitates the transition from interaction-oriented scenario specifications to state-oriented, complete component specifications. Tables 2.1 and 2.2 indicate only HySCs and LSCs as MSC notations with a corresponding formal semantics. On a very detailed level, LSCs support both mandatory and optional MSC specifications, whereas HySCs do not make this distinction.

Beyond instance decomposition in Interworkings and in MSC-96/MSC 2000 none of the MSC dialects studied here offers refinement notions for some or all of the modeling concepts of MSCs. As we have argued in the introduction (cf. Section 1.2.3) this deficit induces an early shift from MSCs to other description techniques that support formal refinement – if formal refinement is intended to be an integral part of a systematic development process at all.

Interestingly, except HySCs none of the MSC dialects offers dedicated syntax and semantics for the specification of preemption (or, similarly, exceptions and exception handlers); this is quite surprising given the history of MSCs as a means for illustrating telecommunication protocols. In such protocols exceptional cases are quite typical. Popular state-oriented notations for the specification of reactive system behavior, such as statecharts and ROOM-Charts (cf. Section 3.3.2), explicitly support preemption specifications.

Another important modeling aspect is also neglected by most MSC dialects. The composition operators of MSC-96/MSC 2000, SDs, EETs, HySCs, and LSCs treat their operands as entirely separate MSCs. Sometimes, however, different MSCs are used to represent separate views on the same service or execution segment. The lack of proper notation for making the overlap in the interactions (depicted in different MSCs) explicit introduces ambiguity into MSC specifications. Interworkings do provide an operator for overlapping scenarios; this operator, however, imposes a very strict requirement on its operands: the common interactions must match exactly to yield consistent specifications.

Unfortunately, none of the MSC dialects studied above combines the following important features:

- adequate expressiveness for both partial and complete interaction behavior in reactive systems,

- existence of a formal semantics, integrating the notions of interaction and state to enable the smooth transition between interaction- and state-oriented description techniques we seek,
- existence of refinement notions for the modeling aspects of structure *and* behavior addressed by MSCs,
- syntactic and semantic support for
  - preemption,
  - overlapping scenarios,
  - progress/fairness constraints.

This is our prime motivation for introducing another MSC dialect in Chapter 4. Its moderate syntactic and semantic extensions (to what is known from MSC-96) allow integrating interaction- and state-oriented specifications, and provide support for modeling preemption, overlapping interaction patterns, as well as progress constraints. Despite these extensions, which cover the entire list of features mentioned above, the new notation is syntactically close to MSC-96; we discuss the differences, in detail, in Chapter 4. Based on the notation of Chapter 4 we address MSC refinement in Chapter 5, discuss the distinction between partial and complete behavior specifications in Chapter 6, and provide transformation procedures for obtaining complete behavior specifications from MSCs in Chapter 7.

2.5. Related Work

Clearly, in the preceding sections we have covered only a small selection of the many different MSC-like notations for component interaction. Here, we give a brief list of further references to other MSC dialects.

The author of [Ren99] discusses the syntax and process-algebraic semantics definition for MSC-96, in detail. This reference also contains an overview of the history of MSC-96, including a brief survey of similar notations that have influenced or are related with MSC-96.

[Fac95] provides a formal definition of the syntax and semantics of Time Sequence Diagrams (TSDs), an earlier notation recommended by the International Standardization Organization (ISO, cf. [ISO87]), specifically in the context of ISO/OSI service specifications. TSDs typically depict the interactions between one particular layer of the ISO/OSI protocol and the users of this layer.

The MSC variant discussed in [AHP96, Hol95, Hol96] allows “plugging in” several semantic models for the communication between the depicted components. One example of such a

pluggable communication model is given by first-in-first-out channels as the component's connection. For each communication model the authors derive constraints at the consistency of MSCs; consistent MSCs admit component implementations that exhibit the depicted interaction behavior, based on the selected communication model.

The author of [Ber97] introduces a notation for interaction scenarios consisting of a graphical part (with syntax for alternatives, concurrency, and repetition), and a textual part. The textual part serves to comment the context of the graphical specifications; it also describes loop bounds and guards for alternatives. We also refer the reader to [Ber97] for a discussion of similar notations from object-oriented analysis and design. [KMST96, KSTM98] also contains a syntax for scenario specification; similar to the one of [Ber97] its roots are in the interaction diagram notation of [JCJO92].

2.6. Summary

In this chapter we have studied and compared the following graphical notations for component interactions:

- MSC-96,
- Object Message Sequence Charts,
- Sequence Diagrams,
- Extended Event Traces,
- Interworkings,
- Hybrid Sequence Charts,
- Life Sequence Charts, and
- MSC 2000.

We have thoroughly dealt with the rich syntax of MSC-96, and have thus established a basis for discussing the other MSC dialects. This discussion has highlighted potential for improvement in the notations under consideration. We address this potential in Chapter 4 where we introduce an MSC notation that fulfills the requirements identified here.

All MSC dialects we have studied in this chapter enable specification of structural *and* behavioral aspects of interaction patterns. An MSC depicts several components, interacting to achieve a certain goal. In this sense MSCs illustrate the coordination aspect of system behavior. This complements the typically complete specifications of individual component behavior, as state-oriented specification techniques provide them. In Chapter 3 we give an

impression of several popular automaton models for state-oriented behavior specifications. This highlights the difference between the two classes of description techniques. Moreover, we get an impression of the models we target when transiting from partial to complete behavior specifications with MSCs.

CHAPTER 3

State-Based Description Techniques for Component Behavior

Automaton models have been studied extensively in the literature as a means for specifying state-oriented behavior of reactive systems. In this chapter we take a closer look at the role of automata in the development process and contrast it with the one of MSCs. Furthermore, we give an overview of several automaton models. This overview complements the comparison of MSC dialects of Chapter 2, and helps underline the modeling aspects on which automaton specifications focus. This chapter also prepares our treatment of the transition between MSCs and automata in Chapter 7.

Contents

3.1. Introduction **82**
3.2. Automata in the Development Process **85**
3.3. Overview of Automaton Models **88**
3.4. Related Work **103**
3.5. Summary **103**

3.1. Introduction

In the preceding chapter we have studied several graphical description techniques for component interactions. Despite the possibility for specifying component states offered by MSC-96/MSC 2000, SDs, and LSCs, the major focus of MSCs is on component collaboration. This provides a global view on a sequence of steps performed by a set of components to establish a certain goal.

In Chapter 2, for instance, we have used MSCs to illustrate interaction scenarios within a simplified Central Locking System (CLS) for car doors. Recall that the CLS consists essentially of three components. Two of these (*LM* and *RM*) represent the lock motors of the left and right door lock, respectively. The third component (*Control*) coordinates the reactions of the CLS to incoming requests from the car user. Upon receipt of message *lck* component *Control* initiates the closing of the locks by sending messages *ldn* and *rdn* to *LM* and *RM*, respectively. *LM* and *RM* indicate fulfillment of the locking request by returning the messages *lmr* and *rmr*, respectively, to *Control*. The unlocking scenario proceeds similarly; it only differs in the messages sent, and in the result obtained (open locks instead of closed ones). Figures 3.1 (a) and (b) depict these two scenarios in the form of MSCs.

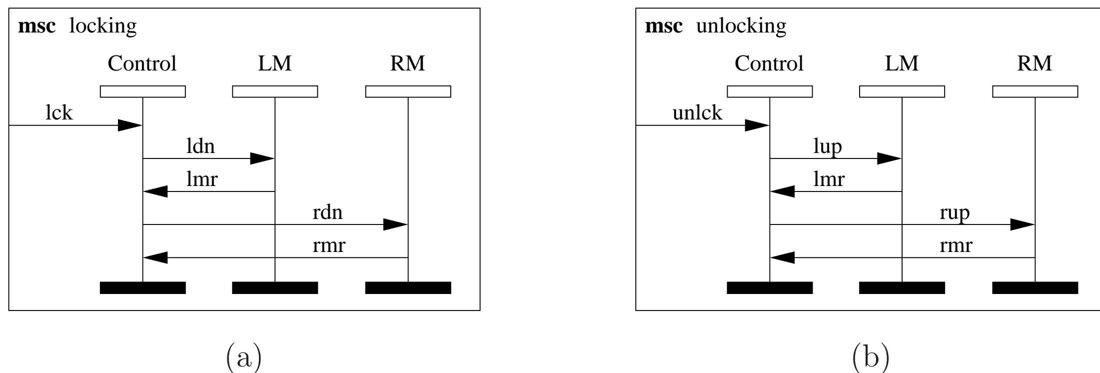


Figure 3.1.: Scenarios of the CLS

The two MSCs give no information on *how* the three components establish their interaction behavior. Instead, we obtain an overview of the sequence of steps resulting in the car being either locked or unlocked. In this chapter we discuss the contrast between overview specifications and detailed component behavior; moreover, we take a closer look at dedicated description techniques for the latter.

The specification and design of detailed component behavior is the domain of state-oriented modeling techniques. *Automata*, also called *State Machines* (SMs) or *State Transition Systems* (STSs), are the favorite graphical description technique for state-oriented component specifications; they stress state change and the input/output relationship that defines an individual component's behavior.

The authors of [RJB99] summarize the methodical difference between state-oriented and interaction-oriented description techniques as follows: "A state machine is a localized view of an object, a view that separates it from the rest of the world and examines its behavior in isolation. It is a reductionist view of a system. This is a good way to specify behavior precisely, but often it is not a good way to understand the overall operation of a system." (cf. [RJB99], p. 68). To illustrate this difference further we return to the CLS example, this time from a state-oriented perspective.

Figure 3.2 shows automaton specifications for the detailed behavior of *Control*, *LM*, and *RM*. In this figure we use labeled boxes to represent components[1]. Arrows between the boxes denote directed communication paths. In the following paragraphs we leave open how the communication is established; it suffices to know that only connected components communicate.

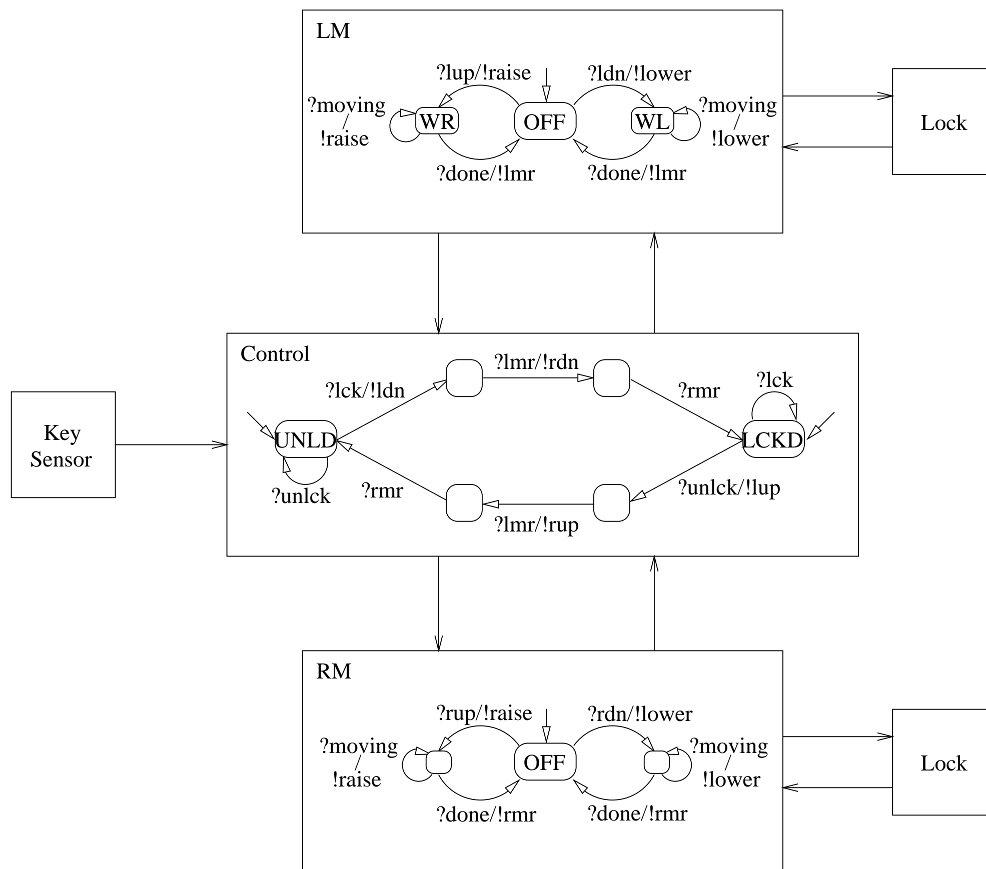


Figure 3.2.: Detailed behavior specification of the CLS

The boxes for *LM*, *RM*, and *Control* contain the respective component's automata. For a more detailed discussion of automaton syntax and semantics we refer the reader to Section

[1]The behavior of the components *Key Sensor* and *Lock* is irrelevant for this introduction.

3.3 and Chapter 7. Here, we aim only at conveying a general intuition about automata as a state-oriented modeling technique.

Each automaton consists of a set of states (indicated by rectangles with rounded corners) and transitions (indicated by arrows). A state represents a certain condition in the behavior of the component it is associated with. Transitions indicate the input messages (prefixed by "?") which cause a state change of the component, and (optionally) the output the component produces (prefixed by "!") when changing from the source to the target state of the transition.

If, for instance, component *LM* receives message *ldn* in state *OFF* it outputs message *lower* (which we assume to be directed at the component *Lock*), and switches to the state labeled *WL*. It remains in this state until it receives message *done*. Upon receipt of this message *LM* outputs message *lmr* and switches back to state *OFF*. While in state *WL*, the component replies every incoming message *moving* by sending message *lower*. Intuitively, component *LM* "waits" for the lock to close before it signals success back to *Control*.

This automaton gives a much more detailed specification of *LM*'s behavior than the corresponding MSCs of Figure 3.1 do. In particular, it describes what happens in *LM* between the receipt of a request and the corresponding reply for *both* use cases (locking and unlocking). Similarly, the automata for *Control* and *RM* represent the individual component's behavior in detail. *Control*'s automaton, for instance, specifies what happens if the car user tries to unlock a car whose door locks are already open: *Control* remains in the unlocked state (*UNLD*) without initiating further interactions until it receives a *lck* message.

Note, however, that the gain of complete information about the behavior of individual components comes at the price of losing the overview of the "big-picture", i.e. the coordination necessary to establish a certain goal. From the automata for *Control*, *LM*, and *RM* alone it is rather complicated to derive the interplay of the three components for the locking and the unlocking scenario.

	MSCs	Automata
scope	projection of overall system behavior onto particular services/use cases	projection of overall system behavior onto individual components
completeness of the depicted behavior	typically partial	typically complete

Table 3.1.: The role of MSCs versus that of automata

Table 3.1 gives a rough summary of typical roles played by MSCs and automata during the development process. Clearly, the borders between these roles are not tightly fixed. Each of the MSCs in Figure 3.1 represents only one of the scenarios of the CLS. However, we have studied several MSC dialects with operators for combining partial scenarios to yield complete behavior specifications in Chapter 2. In Chapter 7 of this thesis we study

the relationship between MSCs and automata in much more detail. In particular, we investigate the question of how to transit from MSCs (representing scenarios) to automata (representing complete component behavior).

To prepare this discussion we take a closer look at automata in this chapter. In Section 3.2 we discuss the methodical role of automata in the development process. Section 3.3 contains an overview of several automaton models for the specification of components in distributed, reactive systems. This overview includes Mealy machines, statecharts, ROOMCharts, ω-automata, and I/O-automata. This also provides a first impression of typical state-oriented modeling concepts, such as liveness and fairness. We mention related work in Section 3.4 and summarize this chapter in Section 3.5.

3.2. Automata in the Development Process

The roots of using automata for behavior specification lie in the huge body of theoretical work on the relationship between automata and formal language theory (cf. [HU90, Tho90], and the references therein). Today, automata often serve as a means for detailed behavior specification, slightly more abstract than programs in concrete programming languages.

As we will see in more detail in Section 3.3, typical automaton models base on two simple notions for representing component behavior: states and state transitions. The states classify certain conditions in the behavior of the component under consideration, such as the portion of an input already consumed. State transitions indicate possible reactions of the automaton with respect to the current state and the input of the automaton.

The notions of state and state transitions are also inherent in all state-oriented programming languages, such as Modula-2, C, C++, or Java. Here, the state space of a system is spanned by all variables defined by the system components. States relate these variables to concrete values. Transitions correspond to changing variable assignments.

By means of this direct correspondence, automata help capture the essence of state-oriented system behavior without having to commit to the syntax of any particular programming language. The transformation of an automaton specification into a corresponding implementation in any state-oriented programming language is entirely schematic. Figure 3.3 shows one of several possible ways for implementing *LM*'s automaton in Java. Here the state in which *LM* resides is captured by variable *currentState*. Transitions are represented by corresponding `if`-clauses in the `case`-statement for the respective state. By means of the functions `readInput()` and `writeOutput()` we capture the reading of input messages and the writing of output messages, respectively (without going into technical details here).

```
while(true) {
  ...
  switch(currentState) {
    case OFF:
      if(readInput() == "ldn") {
        writeOutput("lower");
        currentState = WL;
      } else if (readInput() == "lup") {
        writeOutput("raise");
        currentState = WR;
      } else {
        /* idle */
      }
      break;
    case WL:
      if(readInput() == "moving") {
        writeOutput("lower");
      } else if(readInput() == "done") {
        writeOutput("lmr");
        currentState = OFF;
      } else {
        /* idle */
      }
      break;
    case WR:
      ...
  }
  ...
}
```

Figure 3.3.: Fragment of one possible implementation of *LM*'s automaton (cf. Figure 3.2)

In fact, the rich syntax provided by popular automaton models, such as statecharts and ROOMCharts (cf. Section 3.3), turns automaton models into "graphical programming languages". Clearly, however, the role played by automata in the development process is not limited to the graphical representation of state-oriented implementations. For automaton models a large body of work on their use for property specification, on refinement calculi and verification support exists; examples include [Har87, CD94, Bro97, Rum96, Kle98, Sch98, Mül98, DW98]. This explains, in part, the popularity enjoyed by state transition systems: their methodical "handling" is, by now, quite well understood[2], and the basic idea of specifying a component's states, state changes, and outputs in relation to triggering input seems simple and intuitive.

Yet, state-oriented behavior specifications as provided by automata always reveal to some extent *how* the behavior is established; the partitioning of a component's or system's state space, and the selection of transitions between these states is usually connected with *design* decisions. During *specification*, on the other hand, we try to avoid introducing such decisions into the description of the system under consideration.

This, again, illustrates the complementary roles of MSCs and automata in the development process. MSCs help capture interaction requirements, typically without revealing or fixing the details of how the interaction is established. Automata put more focus on how the result is obtained.

This observation suggests using MSCs as the "front-end" for requirements capture and specification, and automata as the corresponding "back-end" for more detailed design. This, in turn, motivates establishing a link between the requirements captured by MSCs and their implementation in the form of automata for individual components.

In the remainder of this thesis we address these challenges. In Chapter 4 we prepare the semantic link between MSCs and automata by introducing a semantic framework for capturing both interaction and state within MSC specifications. In Chapters 5 and 6 we deal with MSCs as a tool for capturing, specifying, and manipulating system requirements by considering notions of MSC refinement, and by studying the properties that MSCs allow us to express. This stresses the role of MSCs as the "front-end" for requirements capture and specification. In Chapter 7 we establish the mentioned link between MSCs and individual component specifications in general, and between MSCs and automaton specifications in particular.

To give an impression of the modeling elements provided by popular automaton models, especially in the context of reactive systems, we take a closer look at several such models in the following section. This also complements our discussion of MSC dialects in Chapter 2, and prepares the formal definition of the automaton model we use in Chapter 7. More specifically, the automaton model of Chapter 7 combines features of the Mealy-, ω-, and "spelling"-automaton models we review here.

[2] This is not to say that this knowledge has found its way already into industrial practice and into most of the tools that claim to support state-based system specification.

3.3. Overview of Automaton Models

For the specification of individual component behavior, stressing state change and input/output relationships, automaton models and transition systems have been studied and used extensively for the past three decades. Among the most well-known examples of such models are

- Moore and Mealy automata,
- statecharts and ROOMCharts,
- ω-Automata, and
- I/O-Automata.

Whereas compiler construction, and more specifically, the specification of lexical analyzers, is the "original" application area for the automaton models of Moore and Mealy, the specification of behavior in concurrent and distributed systems is the domain of the other models. In industrial practice automata used to occur mostly in technical contexts, such as in circuit design; however, the increasing popularity of object-oriented modeling techniques and notations, such as the UML and others (cf. [Rat97, RJB99, Boo94]), has triggered the use of automaton models also in the development of business systems.

In the following paragraphs we briefly discuss each of the automaton models mentioned above, together with related ones, in turn. Similar to the approach we have taken in Chapter 2 we will present the syntax and the informal meaning of the respective model, instead of giving a precise, mathematical semantics. Chapter 7 contains a formal semantics definition of the automaton model we employ there. The syntax of this model is based on Mealy automata; its semantics, however, allows specification of infinite behavior, similar to ω- and I/O-automata. Readers who are familiar with automata as a description technique are invited to skip the remainder of this chapter, and to use it as reference material when studying Chapter 7.

3.3.1. Moore/Mealy-Automata

Moore and Mealy automata (see [HU90]) have emerged in the context of formal language acceptors. Given a formal language L, i.e. a set of words over a given set of symbols, and a particular word w over the alphabet, a language acceptor determines whether w belongs to L. Deterministic and nondeterministic finite automata (see [HU90]) accept a word if and only if, after processing it, they reach a final state. We can interpret the fact whether or not an automaton has reached a final state after having processed the input as an "output" of the automaton ("accepted" if the automaton has eventually reached a final state, and "rejected" else). Moore and Mealy automata can produce more elaborate output. In a

Moore automaton each state has an associated output symbol produced by the automaton whenever it reaches the state while processing the input. In a Mealy automaton each transition has an associated output symbol produced by the automaton whenever it takes the transition while processing the input.

Moore Automata Along the lines of [HU90], we define a Moore automaton A as a sixtuple $A = (S, I, O, \delta, \lambda, s_0)$, where the elements of the tuple denote

S	:	a finite set of states,
I	:	a finite set of input symbols,
O	:	a finite set of output symbols,
$\delta \subseteq (S \times I) \times S$	:	a (nondeterministic) transition relation,
$\lambda : S \to O$	:	a labeling of states with corresponding outputs,
$s_0 \in S$	:	the start state,

respectively. [HU90] defines deterministic Moore automata, whereas we allow nondeterministic transitions, too.

Processing of an input word $a_1 a_2 \dots a_n$ (with $n \geq 0$ and $a_i \in I$ for $1 \leq i \leq n$) starts with the automaton residing in the start state s_0. In each step the automaton nondeterministically selects a transition from its transition relation δ according to the current input symbol and its current state; when performing the transition the automaton advances to both the target state of the transition, and the next input symbol from the input word. Whenever the automaton reaches a state, which includes the start state, it outputs the symbol associated with the state through λ. Thus, an output of A with respect to the input word $a_1 a_2 \dots a_n$ is $\lambda(s_0)\lambda(s_1)\dots\lambda(s_n)$ with $s_i \in \delta(s_{i-1}, a_i)$ for $1 \leq i \leq n$.

Graphically we denote states and transitions by labeled rounded rectangles and labeled arrows between state symbols, respectively. A state symbol for a Moore automaton has two labels: the element from S represented by the state symbol, and an element from O, the output associated with this state. A transition label is an element from set I; the label indicates the input symbol "read" by the automaton when moving from the origin to the target state of the transition. For better readability we prefix elements from sets I and O with "?" and "!", respectively. An unlabeled arrow having a target but no source indicates the target as the start state.

As an example consider the Moore automaton from Figure 3.4. It has three states: *OFF* (the start state), *DWN*, and *UP*, as well as six transitions. The outputs of states *OFF*, *DWN*, and *UP* are *init*, *lmr_d*, and *lmr_u*, respectively. For this automaton we have $I = \{ldn, lup\}$, and $O = \{init, lmr_d, lmr_u\}$.

For the input word "*ldn ldn lup*" the automaton from Figure 3.4 produces the output word "*init lmr_d lmr_d lmr_u*".

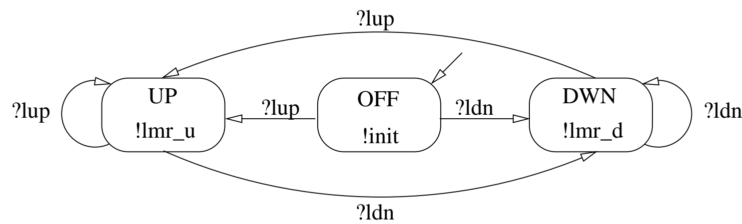


Figure 3.4.: Moore automaton

Mealy Automata As mentioned above, Mealy automata differ from Moore automata in that the former associate the output with transitions, whereas the latter associate the output with states. Consequently, the only difference between Moore and Mealy automata is the labeling function λ. We define a Mealy automaton A as a sixtuple $A = (S, I, O, \delta, \lambda, s_0)$, where with the exception of λ all elements of the tuple have the same meaning as for Moore automata. For Mealy automata we define $\lambda : S \times I \times S \to O$ as the function that establishes the association between a transition (represented by the source state, an input symbol, and the target state) and its output symbol.

Processing of an input word $a_1 a_2 \ldots a_n$ (with $n \geq 0$ and $a_i \in I$ for $1 \leq i \leq n$) for the most part proceeds as we have described for Moore automata. The difference is that a Mealy automaton, whenever it performs a transition, outputs the symbol associated with this transition through λ. Thus, the output of A with respect to the input word $a_1 a_2 \ldots a_n$ is $\lambda(s_0, a_1, s_1)\lambda(s_1, a_2, s_2) \ldots \lambda(s_{n-1}, a_n, s_n)$ with $s_i \in \delta(s_{i-1}, a_i)$ for $1 \leq i \leq n$.

Graphically we denote states and transitions as before by labeled rounded rectangles and labeled arrows between state symbols, respectively. A state symbol now has a single label: the element from S represented by the state symbol. A transition label is an element from $I \times O$; the label indicates the input symbol "read" by the automaton, and the output symbol produced by the automaton when moving from the origin to the target state of the transition. For better readability we separate the two constituents of a transition label by means of a slash ("/").

As an example consider the Mealy automaton from Figure 3.5. It has only one state (the start state *OFF*), and two transitions. The outputs of the transitions labeled with the input symbols *lup*, and *ldn* are *lmr_u*, and *lmr_d*, respectively. For this automaton we have $I = \{ldn, lup\}$, and $O = \{lmr_d, lmr_u\}$.

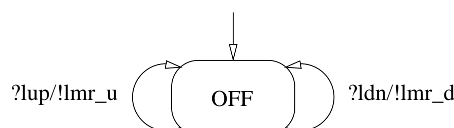


Figure 3.5.: Mealy automaton

For the input word "*ldn ldn lup*" the automaton from Figure 3.5 produces the output word "*lmr_d lmr_d lmr_u*".

Moore and Mealy automata are equivalent with respect to the class of languages they accept (see [HU90] for a proof). However, as the two examples from Figures 3.4 and 3.5 indicate, Mealy automata can become considerably smaller with respect to the number of states and transitions than any equivalent Moore automaton. A Moore automaton has at least as many states as there are different output symbols; this also affects the number of transitions needed for the automaton. A Mealy automaton, on the other hand, has at least as many transitions as there are different output symbols.

We can use both Moore and Mealy automata to describe a component's reaction to any possible input it can receive. Any input word must, however, have finite length. Therefore, a specification using either Moore or Mealy automata is inherently limited to the description of transformational components; the automaton produces the specified finite output for a given finite input and then "stops". This is particularly inadequate for reactive systems, where the basic assumption is that the component under specification does *never* stop. Yet, we can easily work around the restriction to finite objects, as the work on ω-automata (see Section 3.3.3) and the automaton model we introduce in Chapter 7 show.

More significant is that the number of states and transitions in Moore and Mealy automata specifications for realistic examples becomes large very quickly. This reduces the readability of automata specifications considerably, despite their intuitive graphical notation. Moreover, composition of automata was not in the center of concern at the time when Moore and Mealy automata were invented. The ability to compose specifications is, however, essential for systems of nontrivial size.

The mentioned deficits motivated several attempts at devising more "powerful" automaton models – with respect to expressiveness and graphical conciseness. The most prominent representatives of these are statecharts and ω-automata, which we consider in the two following sections.

3.3.2. Statecharts, ROOMCharts

David Harel introduced statecharts (see [Har87, HP98]) in 1987 as a visual formalism for the specification of reactive systems. In a sense, statecharts are a combination of Moore and Mealy automata with a number of syntactic and semantic additions. In particular, statecharts go beyond Moore and Mealy automata by providing the notions of hierarchic and concurrent states. The motivation for these additions was to

- reduce the number of state and transition symbols to keep specifications concise,
- add concepts relevant for the specification of systems of nontrivial size, such as composition of individual automata, and
- provide specification mechanisms of particular importance for reactive systems, such as preemption for the specification of interrupts, as well as timing constraints.

In the following paragraphs we describe the syntactic and semantic modifications with respect to pure Moore and Mealy automata, introduced by statecharts to address these issues.

Extended Transition and State Labels The basic syntactic form of transition labels in statecharts is "*trigger*[*condition*]/*action*", where *trigger*, *action* (optional), and *condition* (optional) represent an input event, i.e. an input symbol in the terminology of Moore and Mealy automata, an action performed by the automaton when taking the transition, and a boolean expression guarding the transition, respectively. Actions are not restricted to output events; in particular, they may alter the data state of the component under development through assignments to the component's variables. [HP98] describes the detailed transition label syntax, as well as a plethora of special actions admitted in transition labels; we can interpret most of these actions, however, as assignments to local component variables. If the guard condition exists, the transition is enabled during an execution of the model only if the condition evaluates to true.

The use of assignments to (local) variables in combination with guarded transitions increases the expressiveness of statecharts with respect to that of Moore and Mealy automata. As an example, consider the statechart of Figure 3.6, which models a simple counter component.

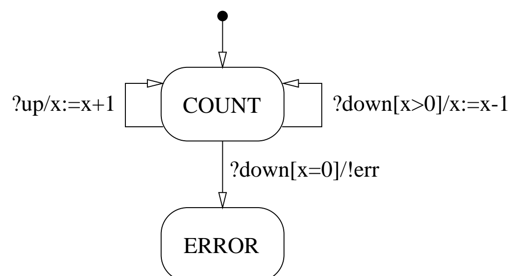


Figure 3.6.: Statechart model for a simple counter

We assume that the counter has a local variable x, which can take on values from the set $\mathbb{N}$ of natural numbers. The counter accepts two kinds of input messages: *up* and *down*. Upon receipt of message *up* in state *COUNT* the counter increases the value of x by one, according to the transition labeled "?up/x:=x+1". Upon receipt of message *down* there are two possibilities. If the value of x is 0, precisely the transition to state *ERROR* is enabled and taken. Otherwise the transition labeled "?down[x $>$ 0]/x:=x-1" is enabled and taken, decreasing the value of x by one. Note that we carry over our convention from the preceding automaton models, and label input symbols by "?" and output symbols by "!" (this is not part of the statecharts syntax).

There is no practicable Moore or Mealy automaton that displays the same input/output behavior as the statechart of Figure 3.6. The problem is that the value of x determines

how many *down* messages may occur before a transition to the *ERROR* state must happen. Thus, in pure Moore and Mealy automata we would have to encode every possible value of x in a corresponding control state of the automaton; this would induce explicit transitions between these states as well. Depending on the size of the domain of x this quickly becomes infeasible.

Actions can also label states in statecharts. A state's *exit* action gets executed when, during execution of the statechart, a transition leaving this state occurs. A state's *entry* action gets executed when, during execution of the statechart, a transition entering this state occurs. This abbreviates labeling *all* exiting and entering transitions of a state with the respective actions. Besides entry and exit actions statecharts introduce *general static reactions* as state-labeling actions. General static reactions execute without resulting in a state change. For further details on actions in statecharts we refer the reader to [HP98].

Hierarchical States ("OR-states") Any statechart state can hierarchically refine into another statechart; we call the decomposed state the *parent* of the *sub-states* of which the refining *sub-chart* consists. This hierarchy concept allows extracting common properties of states, such as common outgoing transitions to the same target state, and representing these graphically only once with the parent state.

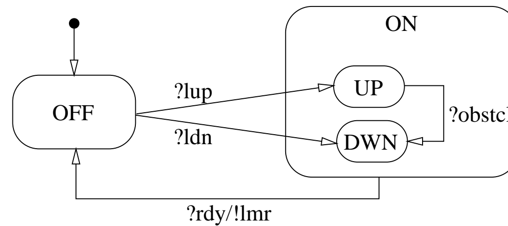


Figure 3.7.: Hierarchic state decomposition

Figure 3.7 shows the hierarchic decomposition of state *ON*. Its substates are *UP* and *DWN*. Upon receipt of message *rdy* in both of these a transition to state *OFF* occurs.

If, during model execution, the statechart enters a parent state, it enters precisely one of its sub-states. For this reason [HP98] terms hierarchic states also "OR-states". If a transition ends at a decomposed state, then the sub-chart must have either a designated initial state, or a history connector (cf. [HP98]) that indicates which of the sub-states is the actual target of the transition.

[HP98] allows *inter-level* transitions, i.e. transitions whose origin state is on a different level of the state hierarchy than the transition's destination state. This is a means for modeling preemption in statecharts; the behavior of a sub-state gets interrupted upon occurrence of a certain signal, and execution continues on a different hierarchic level.

Orthogonal States ("AND-states") Orthogonal states are the statecharts' mechanism for the implicit representation of product automata.

One context where the need for product automata arises is the independent description of several of a component's properties. As an example, consider the two properties represented by the two automata in Figure 3.8 (a) and (b). If we want to combine the corresponding statecharts into a single one, we need to form the product automaton depicted in Figure 3.8 (c). Here, each state represents the combination of one state of the automaton from Figure 3.8 (a), and one state of the automaton from Figure 3.8 (b).

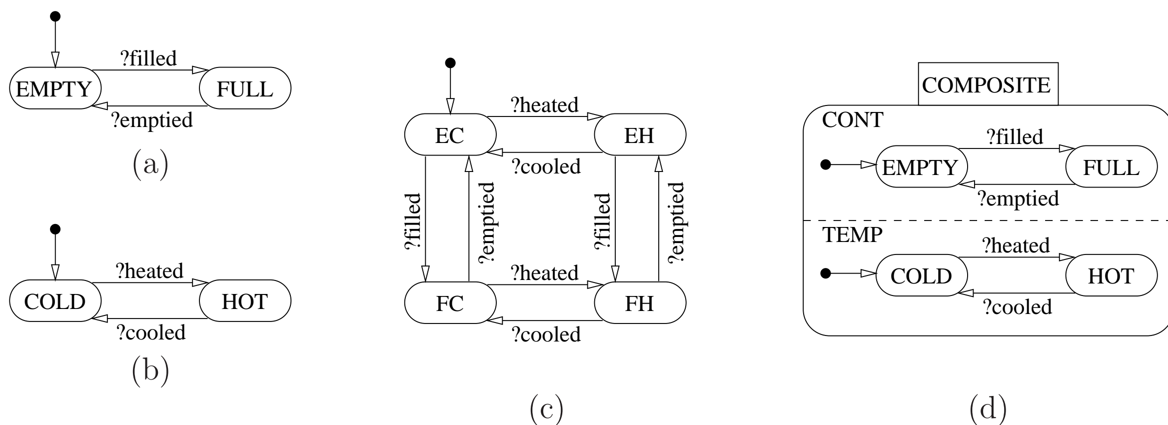


Figure 3.8.: Product automata

In general the number of states of the resulting automaton equals the product of the number of states of the two operand automata; the number of transitions increases correspondingly.

The graphical representation of the product of two automata in statecharts is a state symbol with two compartments separated by a dashed line; each compartment contains one of the two operand automata. The state name appears in a box attached from above to the top of the state symbol (cf. Figure 3.8 (d)).

The statecharts within the compartments of an orthogonal state are the state's sub-states. The state itself is the parent of its sub-states. While, during execution, the model is in an orthogonal state, each of this state's sub-states is in precisely one of its states. For this reason, [HP98] calls orthogonal states also "AND-states".

The second context where product automata are important is the specification of concurrent behavior within a component. The sub-states of an orthogonal state operate independently of one another in the following sense. All sub-states with an enabled transition independently change state upon occurrence of the triggering event. Consider the AND-state of Figure 3.9 (a). Each of its orthogonal states has transitions whose enabledness depends on the occurrence of message *up*. Assume that the sub-chart to the top (labeled "PARITY") is in its *ODD* state, and that the sub-chart to the bottom (labeled "UPCNT") is in its *CNT* state during execution of the model. If the input event corresponding to mes-

sage *up* occurs, then both “PARITY” and “UPCNT” change state: “PARITY” transits to state “EVEN” and “UPCNT” performs its self-loop, thereby increasing the value of x.

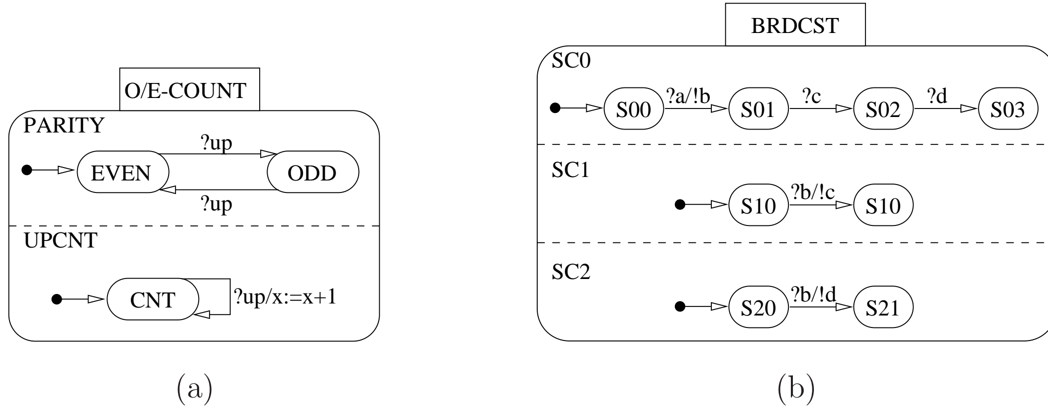


Figure 3.9.: Concurrency and broadcast in statecharts

An output event generated by one of the orthogonal sub-states gets broadcast to all other sub-components as well; the “recipients” of such a generated event may react immediately to it. Because all sub-states of an orthogonal state can react to the same input event simultaneously, [HP98] calls this form of “communication” between orthogonal sub-states “broadcasting”. An example of broadcasting appears in Figure 3.9 (b). The three depicted AND-states *SC0*, *SC1*, and *SC2* communicate via the events b, c, and d. Upon receipt of a, *SC0* produces event b. Due to broadcasting both *SC1* and *SC1* sense b, change state, and generate their respective outputs.

Events, Conditions, Actions The statechart language presented in [HP98] has an elaborate syntax for the specification of events, conditions, and actions. It allows us, for instance, to

- test the presence or the absence of triggering events,
- logically compose triggers by means of connectives such as `and` and `or`,
- test whether other sub-states of an orthogonal state are in a certain state configuration,
- manipulate conditions, i.e. setting their values to true or false,
- test conditions,
- test for the occurrence of timeout events,
- schedule transitions within time intervals.

Furthermore, it allows composing actions by means of sequencing, choice, and repetition. Together, these possibilities turn the statecharts language into a mixture of a state-based, graphical description technique, and a regular textual programming language.

Operational Statecharts Semantics The rich set of syntactic features provided by the statechart language comes at a price. Ever since their inception have statecharts been subject to a considerable amount of criticism largely due to possible ambiguities in the interpretation of the semantics of AND-states, general static reactions, and inter-level transitions. [vdB94] gives an excellent overview of these problems, and describes corresponding solutions. These solution strategies have lead to multiple variants of the original statechart language.

Here, we describe the statechart semantics informally along the lines of [HP98][3]. Intuitively, execution of a statechart model consists of an infinite sequence of steps. During each step, triggered by the presence or absence of events (including input signals and variable changes), the system transits from one state configuration to the next, thereby producing outputs and executing actions. The state configuration at the beginning of the step and the trigger events determine the set of enabled transitions, including the implicit transitions associated with static reactions. An enabled transition fires, which includes execution of the origin state's exit actions, the actions on the transition label, and the target state's entry actions[4]. Execution of an action can produce further events: output events, state entry and exit events, and so forth. Due to the broadcasting communication in an AND-state, all orthogonal sub-states can sense these events immediately; this may cause the enabledness of additional transitions. After all transitions that were enabled at the beginning of the step, as well as those that became enabled during the step, have fired, this step ends and the next one begins.

This simplified description disguises many of the intricacies that become apparent at closer inspection. It is easy to write down ambiguous statechart specifications. To give just one example (cf. [Sch98, Mar92]), consider the AND-state of Figure 3.10. Here, if a step starts with the orthogonal substates *S0* and *S1* in state *S00* and *S10*, respectively, then the absence of input event a, which we denote by the action label $\neg a$, causes *S0* to transit to state *S01*, thereby generating output event b. This enables the transition to state *S11* in *S1*, whose firing results in the *presence* of event a. This requires a to be both absent (otherwise *S0* could not have taken its transition) and present (as the result of *S1* taking its transition) during the same system step.

Whether or not the statechart of Figure 3.10 is, indeed, ambiguous depends on the concrete statechart semantics definition (cf. [vdB94]). Some statechart variants, most notably ROOM's ROOMCharts [SGW94], avoid most of these ambiguities by not making use of AND-states, and by providing other communication mechanisms than instantaneous broadcast.

[3] [HP98] discusses multiple execution variants, differing in the time points at which the system can react to *external* events. We avoid this distinction here for reasons of brevity.

[4] An inter-level transition causes execution of entry/exit actions on all levels of hierarchy it crosses.

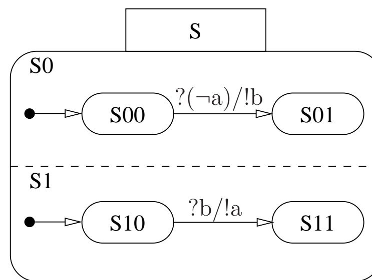


Figure 3.10.: Ambiguous statechart?

ROOMCharts ROOM (short for Real-Time Object-Oriented Modeling, cf. [SGW94]) is a development methodology for object-oriented, reactive systems; its constituents are a precise specification language together with an execution model, as well as the development guidelines contained in [SGW94]. In ROOM a system specification consists of a set of actors, communicating by means of asynchronous message passing over channels. Each actor is of a certain class (the actor class) that determines the actor's interface, and the actor's behavior. Actors receive and send messages along ports. By means of its associated protocol class a port syntactically defines the sets of messages an actor can send and receive via this port. Thus, an actor's ports fix the actor's syntactic interface. ROOM uses ROOMCharts to specify the behavior of actor classes.

ROOMCharts are, in a sense, a lean version of statecharts. From statecharts they have inherited the basic idea of modeling behavior by means of hierarchic state automata. However, the syntactic "features" of ROOMCharts are much simpler than the ones of their predecessor. The following list highlights some of the differences between statecharts and ROOMCharts:

- transition labels have the form "*trigger*[*condition*]/*action*", where *trigger* denotes an input message whose occurrence triggers firing of the transition only if the boolean guard *condition* evaluates to *true*. The input message specification of the trigger contains, in particular, the port along which the actor expects the message. A compound trigger consists of a set of input events any of which can trigger the transition. Nothing but arriving messages can trigger transitions in ROOMCharts; in particular, unlike in statecharts, there exists no means for making a trigger depend on another actor's current state. Upon firing of the transition the *action*, which can represent any executable piece of program that can alter actor variables and send messages along the actors ports, executes.

- each ROOMChart has an explicit initial transition that can have an associated action, but without an input trigger; its use, besides taking the ROOMChart into its starting control state, is the initialization of the actor's data variables.

- hierarchical states (OR-states) exist; however, unlike in statecharts, in ROOM there are no direct inter-level transitions. A ROOM state can have *transition points* whose

purpose is to connect transition segments across hierarchy levels. Transitions that start at a sub-state of an OR-state and continue beyond the parent's boundary end at a transition exit point. This, in a sense, defines an interface concept for states, which allows substitution of states with the same interface within ROOMCharts.

- group transitions, i.e. transitions that start at a transition exit point of an OR-state and have no destination within the OR-state denote "high-level interrupts"; they represent individual transitions with identical labels, starting at all sub-states of the OR state and leaving it via the same transition exit point; group transitions of an OR-state take precedence over all transitions within the OR-state.

- entry and exit actions exist; to obtain a similar effect as the statecharts' general static reactions in ROOM, the developer must specify an explicit internal self-loop.

- ROOMCharts have no AND-states. This is the most significant difference between ROOMCharts and statecharts. The rationale behind their omission is, according to [SGW94], to avoid unintentional coupling through implicit communication among orthogonal states. In ROOM different actors – each behaving according to a sequential ROOMChart –, communicating via explicit message exchange, model concurrent system aspects. There is no concurrency within a single actor. To translate an AND-state from a statechart specification into ROOM typically involves turning each of the orthogonal sub-states into a separate ROOM actor, and converting all (implicit) broadcasting communication between the orthogonal states into explicit message exchange along each actor's ports. The omission of AND-states avoids most, if not all, of the problems with statecharts we have mentioned above. In particular, it simplifies the underlying execution model considerably.

- The *run-to-completion* execution model bases on the queuing of messages on the channels that connect actors. All actors execute independently of one another. An actor that receives a message processes it by performing an enabled transition, if such a transition exists in the actor's corresponding ROOMChart. Every transition runs to completion; a queue holds all messages that arrive at a port of the actor while the transition fires.

Statecharts and ROOMCharts extend the expressiveness of Moore and Mealy machines both on the syntactic and semantic level. Hierarchical state decomposition allows state-based modeling on multiple levels of abstraction. The incorporation of data state, and of actions with the potential to modify the state, into the automaton models allows specification of systems with complex data state by means of automata with a finite set of "control" states. Because of their target application domain of reactive systems, the execution models for both statecharts and ROOMCharts impose no restriction on the length of the input. The instantaneous broadcasting among orthogonal states requires very careful use of AND-states in statechart models. ROOMCharts avoid these problems by means of precise actor interfaces, the omission of AND-states, and a simpler execution model.

3.3.3. ω-Automata, I/O-Automata, "Spelling" Automata

The major motivation behind the introduction of statecharts and their descendants was, as we have pointed out above, to reduce the visual complexity of automaton models to enable their practical application in an engineering context.

Researchers have studied other automaton models that lend themselves for the specification of reactive systems more from a scientific perspective. This has lead to a large body of work on the expressiveness of automaton models, on the theoretical and practical decidability of properties of the languages accepted by these automata, and – on the more pragmatic side – on the definition of refinement techniques to enable a methodical application of automaton models in the development process for reactive systems. Here, we give a brief overview of three models that we take as representatives of these research directions: ω-automata, I/O-Automata, and "spelling" automata.

ω-Automata The automaton models of Moore and Mealy we have described in Section 3.3.1 operate on finite input sequences only. Automaton models with finite state sets that deal with infinite input and output sequences – which we call ω-automata in the sequel – have emerged, and were studied in the context of decision problems in mathematical logic (cf. [Tho90]) in the 1960s. Because of their ability to model relevant properties of component behavior in reactive systems precisely – which they share with temporal logics (cf. [Tho90, Eme90]) – and because effective procedures for deciding whether an automaton fulfills a certain property have been invented in the 1980s (cf. [Eme90]), ω-automata have experienced a revival over the past decade. Here, we give only the basic ideas behind ω-automata to allow a comparison with the other models introduced in this section. For a detailed exposition of the members of this automaton class, and of their properties, we refer the reader to [Tho90].

The most famous approaches for defining ω-automata are those of Büchi, Muller, and Rabin. Each of them uses models with finite state and input symbol sets, as well as some form of transition relation. Their major difference is in how they define acceptance of an infinite input sequence. Recall that Moore and Mealy machines have no explicit sets of "accepting" states; the idea behind these automata is that the output sequence resulting from a given input sequence indicates whether or not the automaton has accepted the input. The ω-automata of Büchi, Muller, and Rabin have no output alphabet; their acceptance conditions involve the infinite occurrence of states or state sets in executions of an automaton.

As an example, consider the definition of a nondeterministic Büchi automaton A as a quintuple $A = (S, I, \delta, s_0, F)$, where the elements of the tuple denote

S	:	a finite set of states,
I	:	a finite set of input symbols,
$\delta \subseteq S \times I \times S$	:	a (nondeterministic) transition relation,
$s_0 \in S$	:	the start state,
$F \subseteq S$	:	an acceptance set,

respectively.

A run of automaton A on an infinite input sequence $\alpha = \alpha_0\alpha_1\alpha_2\ldots$, with $\alpha_i \in I$ for $i \geq 0$, is an infinite sequence $\sigma = \sigma_0\sigma_1\sigma_2\ldots$ of states, such that $\sigma_0 = s_0$ and $(\sigma_j, \alpha_j, \sigma_{j+1}) \in \delta$ for $j \geq 0$.

Automaton A accepts α if and only if there is an $f \in F$, and a run σ of A on α such that f occurs infinitely often in σ. This acceptance condition allows us to "force" progress onto an execution of the automaton: input sequences, whose runs avoid all states from F for infinitely many steps do not belong to the language accepted by the automaton.

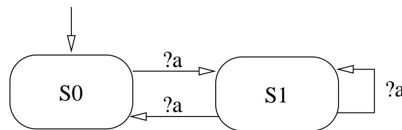


Figure 3.11.: Büchi automaton

Figure 3.11 shows an example Büchi automaton, accepting infinite sequences of a's. For the graphical representation of states and transitions (whose trigger is an input symbol) we have adopted the same conventions as in the previous sections. If we take $F = \{S0\}$ in this example, we force each execution of the automaton to leave state *S1* eventually whenever it occurs in the execution.

Muller and Rabin automata are deterministic and have slightly different acceptance conditions than Büchi automata (cf. [Tho90]).

The most interesting aspect about these automaton classes is, however, the proximity of the languages over infinite input sequences they accept, and temporal logics for which effective verification procedures exist. This has lead to a line of research that suggests for reactive systems to model their behavior, and to specify their properties by means of (variations of) ω-automata (cf. [Tho90, Eme90, MP95]); the question of whether a system has a certain property then becomes a question of determining language inclusion with respect to the corresponding automata.

I/O Automata Input/Output automata (I/O automata, for short) have been introduced by Lynch and Tuttle [LT87, LT89, Lyn96] as a model for the behavior of components in distributed systems, specifically targeting the verification of algorithms for such systems. The emphasis in the theoretical work on this automaton model is on refinement and compositionality [Lyn96, LV95, LV96].

In contrast to the state sets of ω-automata, the ones of I/O automata are not necessarily finite. An I/O automaton can perform three kinds of actions that label transitions: input, output, and local (internal) actions. More precisely, along the lines of [Mül98][5], we define

[5] [Mül98], as well as [LT87, LT89, Lyn96], uses a slightly different syntax; here, we render the definition in a style similar to the one of the preceding sections.

an I/O automaton A as an eight-tuple $A = (S, I, O, L, \delta, S_0, F_w, F_s)$, where the elements of the tuple denote

S	:	a (possibly infinite) set of states,
I	:	a set of input actions,
O	:	a set of output actions,
L	:	a set of local (internal) actions,
$\delta \subseteq S \times (I \cup O \cup L) \times S$	:	a (nondeterministic) transition relation,
$S_0 \subseteq S$	:	a set of start states,
$F_w \subseteq \mathcal{P}(L)$	:	a set of weak fairness constraints,
$F_s \subseteq \mathcal{P}(L)$	:	a set of strong fairness constraints,

respectively. I, O, and L must be pairwise disjoint; together, they form A's action signature $I \cup O \cup L$. δ must be *input enabled*: $\forall s \in S, a \in I : \exists t \in S : (s, a, t) \in \delta$; i.e. the automaton must be able to perform any input action in any state.

An execution of an I/O automaton A is a finite or infinite sequence[6] $s_0 a_1 s_1 a_2 s_2 \ldots$ that alternates between states and actions of A, beginning in a state (a finite sequence must also end in a state), such that $s_0 \in S_0$, and for all $j \in \mathbb{N}$: $(s_j, a_{j+1}, s_{j+1}) \in \delta$, with $s_j, s_{j+1} \in S$, and $a_{j+1} \in (I \cup O \cup L)$.

Sets F_w and F_s play the role of the acceptance sets of Büchi automata. For each set $f \in F_w$ an infinite execution of the I/O automaton must contain either an infinite number of occurrences of actions from f, or an infinite number of states in which no element of f is enabled. For each set $f' \in F_s$ an infinite execution of the I/O automaton must contain either an infinite number of occurrences of actions from f', or an at most finite number of states in which an element of f' is enabled. Without going into further details about the notions of weak and strong fairness induced by these two sets, we note that while the acceptance sets of Büchi automata contain states, I/O automata assign progress to transitions (via actions).

Input actions correspond to reading an input symbol in the terms of a regular Moore or Mealy automaton. A similar correspondence holds for output actions. Local actions result in a state change without reading input or writing output; they model local computation (or idling) steps of the automaton. Except in very simple cases there is no appealing graphical notation for I/O automata; because of the input enabledness requirement each state would need outgoing transitions for all input actions. Therefore, [Lyn96] uses a pre-/postcondition specification style for actions. As an example, consider the I/O automaton from Figure 3.12. The automaton operates on two variables: x and *signal*. For simplicity we assume that x and *signal* are of type $\mathbb{N}$ and $\mathbb{B}$, respectively. Action *up* takes an integer input and increments the value of variable x corresponding to the parameter value. Action *get* sets variable *signal* to true, thus enabling output of the current value of x through output action *out*. This "two step" approach to emitting x's value is necessary, because a transition can either perform an input or an output action, but not both.

[6] In the remainder of this section we disregard finite executions.

input	up(y)	**input**	get	**output**	out(x)
post:	x := x+y	**post:**	signal := true	**pre:**	signal = true
				post:	signal := false

Figure 3.12.: I/O automaton for a simple counter

The use of action signatures (together with the enabledness condition) helps giving clear definitions of action hiding (a form of encapsulation), as well as of parallel composition [Lyn96]. I/O automata are compositional in the sense that properties of parts carry over to their parallel composition; this can ease reasoning about larger specifications considerably. Moreover, the existence of refinement techniques [LT87, LT89] provides a basis for step-wise top-down system development. [Mül98] has extended these techniques by abstraction rules allowing to integrate I/O automata into interactive and fully automatic verification methodologies.

“Spelling” Automata [Rum96] introduces the class of “spelling” automata, which are, with respect to their expressiveness, equivalent to I/O automata. The target domain for the application of spelling automata is the state-based behavior specification of objects in distributed object-oriented systems. The major difference between I/O automata and spelling automata is that the former stress the concept of actions, whereas the latter place messages and message sequences in the center of concern.

[Rum96] defines a spelling automaton A as a quintuple $A = (S, I, O, \delta, S_0)$, where the elements of the tuple denote[7]

S	:	a (possibly infinite) nonempty set of states,
I	:	a nonempty set of input symbols,
O	:	a nonempty set of output symbols,
$\delta \subseteq S \times I \times S \times O^\omega$	:	a (nondeterministic) transition relation,
$S_0 \subseteq S \times O^\omega$	:	a set of initial states and initial output sequences,

respectively.

Processing of an input sequence $a_1a_2a_3\dots$ (with $a_i \in I$) starts with the automaton residing in a start state s_0 for some $(s_0, o_0) \in S_0$; before entering this state, the automaton produces the corresponding initial output o_0. In each step the automaton determines a transition from its transition relation δ according to the current input symbol and its current state; when performing this transition the automaton advances to both the target state of the transition, and the next input symbol from the input word; furthermore, it emits the output sequence associated with the transition through δ. Thus, the output of A with respect to the input word $a_1a_2a_3\dots$ is $o_1o_2o_3\dots$ where $(s_i, a_{i+1}, s_{i+1}, o_{i+1}) \in \delta$ for $i \geq 0$.

[7] By X^ω we denote the set of finite and infinite sequences over X; in Section 7 we give a precise definition of this notation.

Allowing transitions to have an output label avoids the I/O automata's need for extra intermediate states to separate input from output actions. Spelling automata translate the notion of input enabledness to chaotic behavior; if, in the current state, there is no transition that can process the current input symbol, the automaton can display arbitrary behavior from this point on. [Rum96] interprets this form of chaos as underspecification, and provides an extensive treatment of a refinement calculus for spelling automata that is largely based on the idea of removing underspecification in the sense just mentioned.

3.4. Related Work

The overview of automaton models we have given in Section 3.3 is, of course, incomplete. There exists a plethora of other state-based approaches, such as the process diagrams from SDL [EHS98], timed port automata [GR95], and variations of spelling automata [GKRB96, Bro97, Kle98]. Petri nets [Rei82] can express the state-based behavior of a single component, as well as the communication between several components in one diagram. Some synchronous languages, which have a similar communication model as statecharts, also describe component behavior as state-transition relations [BG88, Mar92]. In [Leu95] and [Kle98] automata serve as the basis for defining the semantics for MSCs and scenarios, respectively.

As we have mentioned earlier, the role of automata is by no means fixed to being only a slight abstraction from executable code. [CD94], [Rum96], [WK96], [Kle98], and [Sch98] contain examples of refinement notions, and even of refinement calculi for statecharts and related models. Their application makes automata accessible to the systematic design of the detailed behavior of individual components.

3.5. Summary

In the preceding sections we have treated two major topics. First, we have discussed the role of automata in the development process to contrast it with the role of MSCs. Second, we have given an overview of several relevant automaton models for reactive systems. The discussion of these two topics together has underlined the methodical difference between MSCs and automata as description techniques for system behavior.

The major way of using automaton models in this context today is the specification of complete component behavior; the automaton for a component covers its complete input/output relationship, as well as the component's complete state change relation. MSCs, in contrast, typically depict scenarios, i.e. certain parts of the behavior of a set of components, often without aiming at overall completeness.

Because automata usually reveal to a certain degree how a particular behavior is achieved, their typical position in the development process is closer to design and implementation

than to requirements capture. In this sense we see MSCs as the "front-end" of the analysis and specification task, and view automata as the corresponding "back-end" for design and implementation. This calls for establishing a link between these two description techniques.

The overview of automaton models we have given includes Moore and Mealy automata, statecharts and ROOMCharts, as well as ω- and I/O-automata; it prepares our definition of the automaton model we employ in Chapter 7, which combines features of Mealy- and ω-automata. Moore and Mealy automata are the predecessors of pragmatic automaton models like statecharts and ROOMCharts. The latter two have emerged as attempts at reducing the sizes of automaton specifications in practical engineering contexts. The work on ω-automata, I/O automata, and spelling automata centers around the notions of effective verification, compositionality, and refinement. ω-automata and their relatives allow, in particular, adding progress and fairness constraints to automaton specifications.

Using automata typically means focusing on the input/output behavior of a single component over time. An important step of the development process is, however, the composition of individual components such that they cooperate to achieve an overall goal. MSCs allow us to specify the interaction sequences into which the individual components must integrate to allow the system to achieve this goal. In the following chapters we will work out an approach for the combination of the two behavioral views we have studied so far: the interaction-oriented, global system view of MSCs and the state-oriented, "birds-eye-view" of automaton models.

CHAPTER 4

YAMS – Yet Another MSC Semantics

This chapter contains the syntax and semantics definition of the MSC notation we have developed to solve the deficits identified in Chapter 2. Our first step towards this goal is to define a precise model of the system class we target. Then, we base the semantics definition on this model. We discuss the expressiveness of the notation we introduce, and present several extensions going beyond the MSC-96 standard.

Contents

4.1. Introduction **106**
4.2. System Model and Mathematical Preliminaries **108**
4.3. Abstract Textual Syntax **112**
4.4. Denotational MSC-Semantics **115**
4.5. Discussion of the Semantics **131**
4.6. HMSCs **139**
4.7. Example: the Abracadabra-Protocol **146**
4.8. Related Work **152**
4.9. Summary **152**

4.1. Introduction

In this chapter we introduce the syntax and semantics of the MSC variant we employ in the remainder of this thesis. The semantic framework we establish here forms the basis for the investigation of the methodical aspects of MSC usage in the subsequent chapters.

Our motivation for the definition of "yet another" MSC syntax and semantics differing from already existing approaches is twofold.

First, the discussion of MSC variants in Sections 2.2 and 2.3, and the comparison in Section 2.4 have revealed several deficits in the already existing MSC variants. Recall, for instance, that MSC-96 provides no notion of component state, no preemption mechanism, no general concept of liveness conditions, and no semantic integration of message parameters. LSCs, which do improve on MSC-96 by integrating state information and means for the specification of liveness properties with MSCs, introduce other problems. The multitude of ways for specifying liveness, each with its own graphical syntax, can make MSC specifications extremely hard to read, especially if different liveness requirements occur in combination within the same LSC. As an example, in LSCs there is no clear distinction between the specification of preemption and provisional conditions on instance axes. This can result in ambiguous LSC specifications. Moreover, LSCs do not support High-Level MSCs (HMSCs), and provide no semantic integration of message parameters.

Second, we want to establish a semantic framework that supports both reasoning *about* the semantics, and reasoning *with* the semantics. Reasoning about the semantics includes, for instance, establishing properties of the mapping from syntax to semantics, and properties of the elements of the notation, such as symmetry and associativity of composition operators. Reasoning with the semantics includes, for instance, the definition of effective and manageable refinement notions, as well as the derivation of individual component specifications from MSCs. None of the MSC variants addresses the notion of MSC refinement beyond "instance decomposition"; we attribute this to a large extent to the respective semantic models, which – with a few exceptions – do not lend themselves to the definition of practicable notions of refinement.

With the MSC variant we introduce here we aim at providing a specification mechanism of adequate expressiveness that avoids the problems mentioned above. We use a graphical syntax almost identical to the one of MSC-96; we also introduce an abstract textual syntax that directly mimics the graphical one but is easier to manipulate and reason about in the semantics definition.

With a few exceptions the syntax and semantics we define directly correspond to the one of MSC-96. The major difference between our semantics and the one of MSC-96 is that we model neither "weak sequencing", nor "delayed choice". Weak sequencing is a form of MSC composition in MSC-96 that results in the parallel composition of its operands if the instance sets of the operands on which send or receive events occur are disjoint. Our sequential composition and interleaving operators have different semantics. The MSC-96

semantics avoids resolution of an alternative construct until it is inevitable, i.e. until the point where the two alternatives differ. Thus, the MSC-96 view on the interpretation of an MSC is analogous to the execution model underlying tree logics like CTL and CTL* (cf. [Tho90, Eme90]). In our approach each alternative is fixed right from the beginning of an execution; the set of all alternative behaviors is the semantics of the MSC. This is analogous to the execution model underlying linear temporal logics (cf. [Eme90]).

Our MSC variant has several "features" going beyond MSC-96. Major additions are guarded MSCs, guarded alternatives and loops, unbounded finite repetition, a "join" operator, the concept of preemption, and the "trigger composition" operator. Guarded MSCs assign meaning to condition symbols: the interactions guarded by a condition occur only if the condition evaluates to true. We use this concept also to guard the selection of alternatives and the termination of loops. As a simple means for specifying terminating loops with unknown loop bounds we add unbounded finite repetition to the set of loop specifiers known from MSC-96. The "join" operator allows to compose MSCs that represent non-orthogonal views on an interaction sequence; it identifies identical messages of its operands. The preemption mechanism allows specification of exceptional cases in MSC specifications. The trigger composition operator enables specifying that the occurrence of an interaction sequence always causes occurrence of another. This is an important way of defining liveness properties in our approach.

We select the mathematical model of streams as the domain for the MSC semantics; this eases the definition of refinement notions in Chapter 5, as well as the derivation of individual component specifications from MSCs in Chapter 7.

The current chapter has the following structure. In Section 4.2 we introduce the system model underlying our semantics definition; this model supports the specification of a large class of reactive systems. Section 4.3 contains the textual syntax we use in the semantics definitions of Section 4.4. There, we concentrate on the core notational elements, like message exchange, sequential, parallel and alternative composition, guarded MSCs, loops, references, preemption, as well as join and trigger composition. We investigate properties of the semantics definition, such as its well-definedness, and its relationship to the MSC-96 semantics and to temporal logics in Section 4.5. We describe our semantic treatment of HMSCs in Section 4.6 as an example for an extension of the core syntax and semantics of Sections 4.3 and 4.4 (further syntactic and semantic extensions, such as the integration of parameters and parametric MSCs, actions, timers and gates, appear in Appendix A). In Section 4.7 we give an example MSC specification to demonstrate, in particular, the use of unbounded repetition, join and trigger composition, and preemption. Section 4.9 contains a summary of what we have achieved.

4.2. System Model and Mathematical Preliminaries

The system class we aim at is that of open, distributed reactive systems with static structure. Here, we define a simple, yet precise mathematical model for the description of such systems. Along the way we introduce the notation and concepts we need to describe the model.

4.2.1. Notational Conventions

We start with a few notational conventions. By $\mathbb{B}$ and $\mathbb{N}$ we denote the set of booleans (the constants are true and false) and natural numbers (including 0), respectively. We define $\mathbb{N}_\infty \stackrel{\text{def}}{=} \mathbb{N} \cup \{\infty\}$ for the set of naturals together with their supremum (∞). We use the usual extensions of (binary) operations from $\mathbb{N}$ to $\mathbb{N}_\infty$; examples are $x \leq \infty$ for all $x \in \mathbb{N}_\infty$, $\max(x, \infty) = \max(\infty, x) = \infty$ and $\min(x, \infty) = \min(\infty, x) = x$ for all $x \in \mathbb{N}_\infty$. To denote function application we often use an infix dot ("$.$") instead of parentheses to increase readability of our formulae. For $Q \in \{\forall, \exists\}$ and predicates r and p we write $\langle Qx : r.x : p.x \rangle$ to denote the respective quantification over all $p.x$ for which x satisfies the quantification range $r.x$. If the range is understood from the context, we omit it from the quantifying formula. As another form of reduced notation we integrate simple ranges into the specification of the quantified variable; as an example, we sometimes write $\langle \forall x \in \mathbb{N} :: \ldots \rangle$ instead of $\langle \forall x : x \in \mathbb{N} : \ldots \rangle$. $\mathcal{P}(X)$ denotes the powerset of any set X. Given sets $Y_1, Y_2, \ldots$ we define for tuples $y = (y_1, y_2, \ldots) \in Y_1 \times Y_2 \times \ldots$ the projection onto the i-th element of the tuple as $\pi_i.y \stackrel{\text{def}}{=} y_i$ for $i \geq 1$. For the closed interval between $m \in \mathbb{N}_\infty$ and $n \in \mathbb{N}_\infty$ we write $[m, n]$; if $m > n$ then $[m, n] \stackrel{\text{def}}{=} \emptyset$.

The mathematical model serving below as the basis for our notion of system behavior is that of *streams*. Streams and predicates or functions on streams are an extremely powerful specification mechanism for distributed, interactive systems (cf. [Bro99a, Möl99, BS00, Ste97, Rum96]). It serves particularly well for property-oriented component specifications, as well as for the definition of refinement notions and for the verification of corresponding refinement relationships between specifications (cf. [BDD+92, Rum96, Kle98, Sch98]). Here, we give a concise overview of the major concepts and notations with respect to streams to the extent required for this thesis; for a thorough introduction to the topic, we refer the reader to [Bro99a, Möl99, Ste97].

A stream is a finite or infinite sequence of messages. By X^* and X^∞ we denote the set of finite and infinite sequences over set X, respectively. $X^\omega \stackrel{\text{def}}{=} X^* \cup X^\infty$ denotes the set of streams over set X. Note that we may identify X^* and X^∞ with $\bigcup_{i \in \mathbb{N}}([0, i] \to X)$ and $\mathbb{N} \to X$, respectively. This allows us, for $x \in X^\omega$ and $n \in \mathbb{N}$, to use function application to write $x.n$ for the n-th element of stream x. By $|x|$ we denote the length of stream x. It is equal to some natural number if $x \in X^*$; for $x \in X^\infty$, $|x|$ yields ∞. Furthermore, for $x \in X^\omega$ and $n \in \mathbb{N}$, with $|x| \geq n$, we define $x \downarrow n$ to be the prefix of x with length n. By $x \uparrow n$

we denote the stream obtained from x by removing the first n elements. $x\uparrow\infty$ yields the empty stream. We write the concatenation of two streams $x, x' \in X^\omega$ as $x \frown x'$. If $|x| = \infty$ then $x \frown x'$ equals x. By $<x_1, x_2, \ldots, x_n>$ we denote the finite stream consisting, in this order, of the elements x_1 through x_n with $x_i \in X$ for $1 \leq i \leq n$. For $x = <x_1, x_2, \ldots, x_n>$ and $y \in X$ we define $y \in x \stackrel{\text{def}}{=} \langle \exists i : 1 \leq i \leq n : x_i = y \rangle$. As a shorthand, we define $x_1 \frown x \stackrel{\text{def}}{=} <x_1> \frown x$ for $x_1 \in X$ and $x \in X^\omega$. We denote the empty stream by $<>$. For an element $x_1 \in X$ and $n \in \mathbb{N}_\infty$ we define x_1^n to be the stream over X that consists of n consecutive copies of x_1, i.e. $\langle \forall t : 0 \leq t < n : x_1^n.t = x_1 \rangle \ \wedge\ |x_1^n| = n$ holds. Given a subset $Y \subseteq X$ of the base set X, the filter operation $Y Ⓒ x$ yields the stream obtained from $x \in X^\omega$ by removing all elements not contained in Y. The restriction function $x|_{[m,n]} \stackrel{\text{def}}{=} (x\downarrow n)\uparrow m$ yields the part of stream x that starts at position m and ends at position n. Table 4.1 serves as a quick reference for these sets and operators.

Notation	Informal Meaning
X^*	set of finite sequences over set X
X^∞	set of infinite sequences over set X
X^ω	set of streams over set X
$x.n$	nth element of stream x
$\|x\|$	length of stream x
$x\downarrow n$	prefix of length n of stream x
$x\uparrow n$	stream obtained from x by removing the first n elements
$x \frown x'$	concatenation of streams x and x'
$<x_1, x_2, \ldots, x_n>$	finite stream consisting, in this order, of the elements x_1 through x_n
$y \in x$	y appears as an element in the finite stream x
$<>$	the empty stream
x_1^n	the stream consisting of n consecutive copies of element x_1
$Y Ⓒ x$	stream obtained from x by dropping all elements not contained in Y
$x\|_{[m,n]}$	stream obtained from x by considering only elements m through n

Table 4.1.: “Quick reference” for the stream notation used here

We lift the operators introduced above to finite and infinite tuples and sets of streams by interpreting them in a pointwise and elementwise fashion, respectively. Given, for instance, the stream tuple $x : [1, m] \to X^\omega$, with $x = (x_1, \ldots, x_m)$ for $m \in \mathbb{N}$, we denote by $x.n$ the tuple $(x_1.n, \ldots, x_m.n)$, if $n \in \mathbb{N}$.

Below, we use streams to model the behavior of components – including both the communication between components, and the states assumed by these components – over time. To stress this intuition we introduce the name *timed streams* for infinite streams (time does not halt) whose elements at position $t \in \mathbb{N}$ represent the messages transmitted or the states assumed at time t. Based on this intuition we identify tuples over timed streams with streams over tuples, and call both *timed stream tuples*. For instance, we identify $(X \times Y)^\infty$

with $X^\infty \times Y^\infty$ for sets X and Y. Moreover, for finite index sets X, and arbitrary sets Y we identify elements of the domains $X \to Y^\infty$ and $(X \to Y)^\infty$. This is a technical convention that gives us a convenient way of converting streams of functions into functions, whose ranges are streams, and vice versa. If, as an example, we have $z \in (X \to Y)^\infty$, and $x \in X$, then we allow ourselves to write $z.x$ to obtain z's projection onto x. Similarly, if we have $z \in (X \to Y^\infty)$ and $t \in \mathbb{N}$, then we consider $z.x.t$ and $z.t.x$ as synonyms.

4.2.2. System Structure

Structurally, a system consists of a set P of components, objects, or processes[1], and a set C of directed channels. Channels connect components that communicate with one another; they also connect components with the environment. With every $p \in P$ we associate a unique set of states, i.e. a component state space, S_p. We define the state space of the system as $S \stackrel{\text{def}}{=} \Pi_{p \in P} S_p$. For simplicity we represent messages by the set M of message identifiers.

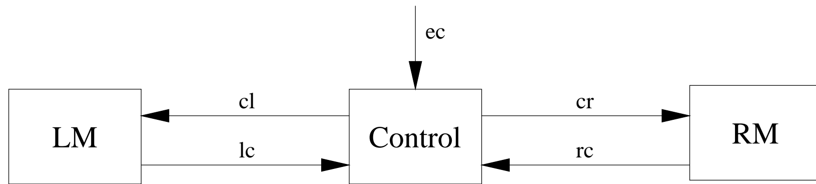


Figure 4.1.: Simple SSD that defines the sets P and C

We use system structure diagrams (SSDs) to describe the sets P and C in graphical notation. Alternatively, we could use ROOM's actor- and binding-diagrams for this purpose. The UML does not have an explicit channel concept; however, we could map the associations described by class and object diagrams onto "logical" channels. The SSD of Figure 4.1, for instance, defines $P = \{LM, Control, RM\}$ and $C = \{cl, lc, cr, rc, ec\}$.

Each channel $ch \in C$ is directed from its source to its destination component. To distinguish multiple channels with identical source and destination components we associate a name (an element from the set CN of channel names) with every channel. Thus, we treat each element $ch \in C$ as a triple $ch = (cn, cs, cd) \in CN \times P \times P$, where cn, cs, and cd denote the channel name, the source component, and the destination component of channel ch, respectively. We use the functions $chn : C \to CN$, $src : C \to P$, and $dst : C \to P$ to project a channel on its name, source, and destination component, respectively; hence, we have $ch = (cn, cs, cd) \Rightarrow chn.ch = cn \wedge src.ch = cs \wedge dst.ch = cd$. We assume that channel names are unique within C, and, where no confusion can arise, identify a channel with its name.

Table 4.2 summarizes these structural elements.

[1]In the remainder of this thesis, we use the terms components, objects, and processes interchangeably.

Entity	Meaning
P	set of system components
C	set of directed channels
S_p	state of component $p \in P$
S	system state ($S \stackrel{\mathsf{def}}{=} \Pi_{p \in P} S_p$)
M	set of message identifiers

Table 4.2.: Structural elements of the system model

To assign structure to the set S_p of states of component $p \in P$ we will explicitly name the local variables of p together with their types in concrete examples. Then, we can interpret a state of p as an assignment of each of p's variables to a value of the corresponding type. Similarly, we can further structure the set of messages M, say, to allow messages with parameters (see Section A.3). To keep our presentation concise, we will not employ a graphical syntax such as the UML's class diagrams or ROOM's actor class specifications.

Above, we have not distinguished a component's control and data state; we defer this distinction until Chapter 7. For the time being we consider the control state as one part of the components' overall state space.

The systems we consider here are fixed in the sense that neither set P, nor set C changes over time. This does not exclude, however, dealing with the dynamic creation or deletion of system components or channels. We refer to Appendix A for a treatment of component creation and deletion in our semantic framework.

4.2.3. System Behavior

Now we turn to the dynamic aspects of the system model. In our model we take into account both the interaction- and state-oriented part of system behavior; this prepares the integration of both aspects in the following chapters.

We assume that the system components communicate between one another and with the environment by exchanging messages over channels. We assume further that a discrete global clock drives the system. We model this clock by the set $\mathbb{N}$ of natural numbers. Intuitively, at time $t \in \mathbb{N}$ every component determines its output based on the messages it has received until time $t - 1$, and on its current state. It then writes the output to the corresponding output channels and changes state. The delay of at least one time unit models the processing time between an input and the output it triggers; more precisely, the delay establishes a strict causality between an output and its triggering input (cf. [Bro99a, BK98]).

Formally, with every channel $c \in C$ we associate the histories obtained from collecting all messages sent along c in the order of their occurrence. Our basic assumption here is that

communication happens asynchronously: the sender of a message does not have to wait for the latter's receipt by the destination component. This allows us to model channel histories by means of streams.

We define $\tilde{C} \stackrel{\text{def}}{=} C \to M^*$ as a channel valuation that assigns a sequence of messages to each channel; we obtain the timed stream tuple $\tilde{C}^\infty$ as an infinite valuation of all channels. This models that at each point in time a component can send multiple messages on a single channel.

With timed streams over message sequences we have a model for the communication among components over time. Similarly we can define a succession of system states over time as an element of set S^∞.

With these preliminaries in place, we can now define the semantics of a system with channel set C, state space S, and message set M as an element of $\mathcal{P}((\tilde{C} \times S)^\infty)$. Any element (φ_1, φ_2) of a system's semantics consists of a valuation of the system's channels ($\varphi_1 \in \tilde{C}^\infty$) and a description of the system state over time ($\varphi_2 \in S^\infty$). The existence of more than one element in the semantics of a system indicates nondeterminism.

Table 4.3 summarizes the semantic entities for modeling system behavior.

Entity	Meaning
$\tilde{C}$	channel valuation at a particular time point ($\tilde{C} \stackrel{\text{def}}{=} C \to M^*$)
$\tilde{C}^\infty$	overall channel history
S^∞	state history
$(\tilde{C} \times S)^\infty$	combined channel and state history
$\mathcal{P}((\tilde{C} \times S)^\infty)$	semantics domain for system behaviors

Table 4.3.: Behavioral elements of the system model

In summary, our system model consists of two parts: the system's static structure and its behavior. The set of channels, the set of component states, and the set of messages determine the system's structure. The channel valuations, i.e. the occurrences of messages on channels over time, and the sequences of states determine the system's behavior.

In the following sections we will use the system model we have defined here as the basis for defining the semantics of MSCs.

4.3. Abstract Textual Syntax

To simplify the semantics definition of our MSC variant we first introduce its abstract textual syntax. We describe the translation from the graphical to the textual syntax in

Section 4.5.5. In the syntax definition we use an extended Backus-Naur Form (BNF) (cf. [Wir86, ASU88]), where production rules of the grammar have the following form:

$$\begin{aligned}\langle \mathrm{N} \rangle ::= \quad & alt_1^{\langle \mathrm{N} \rangle} \\ \| \quad & alt_2^{\langle \mathrm{N} \rangle} \\ \ldots & \\ \| \quad & alt_n^{\langle \mathrm{N} \rangle}\end{aligned}$$

We denote nonterminals of the grammar by enclosing them in angular brackets. To the right of the assignment symbol "::=" we give the alternative productions $alt_1^{\langle \mathrm{N} \rangle}$ through $alt_n^{\langle \mathrm{N} \rangle}$ of the nonterminal $\langle \mathrm{N} \rangle$, such that the symbol $\|$ separates any two alternatives. To indicate repetition of a nonterminal $\langle \mathrm{N} \rangle$ we write $\{\langle \mathrm{N} \rangle\}^*$ and $\{\langle \mathrm{N} \rangle\}^+$ for any finite and any positive number of successive occurrences of $\langle \mathrm{N} \rangle$, respectively. If a terminal symbol $\underline{\mathrm{t}}$ separates any two of these occurrences we write $\{\langle \mathrm{N} \rangle\}^*_{\underline{\mathrm{t}}}$ and $\{\langle \mathrm{N} \rangle\}^+_{\underline{\mathrm{t}}}$, respectively.

For notational convenience in the subsequent sections and chapters, we identify any nonterminal $\langle \mathrm{N} \rangle$ with the language defined by its corresponding grammar rule.

MSC Documents

We use the syntactic category $\langle \mathrm{MSCDOC} \rangle$ to represent MSC documents, i.e. sequences of MSC definitions.

$$\langle \mathrm{MSCDOC} \rangle ::= \{\langle \mathrm{MSCDEF} \rangle\}^*$$

MSC Definitions

An MSC definition, represented by the syntactic category $\langle \mathrm{MSCDEF} \rangle$, associates an interaction description (as defined by $\langle \mathrm{MSC} \rangle$) to an MSC name (as defined by $\langle \mathrm{MSCNAME} \rangle$, which represents a text string). The MSC name becomes important in combination with references; there it acts as a placeholder for the interaction description to which the name relates:

$$\langle \mathrm{MSCDEF} \rangle ::= \mathbf{msc}\ \langle \mathrm{MSCNAME} \rangle = \langle \mathrm{MSC} \rangle$$

MSC Terms

The syntactic category $\langle \mathrm{MSC} \rangle$ represents an interaction description. It captures the syntactic correspondences between the graphical notation's arrows, conditions, inline expressions, and references. In addition, it provides the syntax for expressing arbitrary interactions and

preemption, as well as join and trigger composition.

$$
\begin{aligned}
\langle\text{MSC}\rangle ::= \; & \textbf{empty} \\
\| \; & \textbf{any} \\
\| \; & \langle\text{MSG}\rangle \\
\| \; & \langle\text{GMSC}\rangle \\
\| \; & \langle\text{MSC}\rangle \; ; \langle\text{MSC}\rangle \\
\| \; & \langle\text{GMSC}\rangle \mid \langle\text{GMSC}\rangle \\
\| \; & \langle\text{MSC}\rangle \sim \langle\text{MSC}\rangle \\
\| \; & \langle\text{MSC}\rangle\uparrow_{\langle\text{LSPEC}\rangle} \\
\| \; & \langle\text{MSC}\rangle \otimes \langle\text{MSC}\rangle \\
\| \; & \langle\text{MSC}\rangle \mapsto \langle\text{MSC}\rangle \\
\| \; & \rightarrow\langle\text{MSCNAME}\rangle \\
\| \; & \langle\text{MSC}\rangle\overset{\langle\text{MSG}\rangle}{\rightarrow}\langle\text{MSC}\rangle \\
\| \; & \langle\text{MSC}\rangle\Uparrow_{\langle\text{MSG}\rangle}
\end{aligned}
$$

For simplicity, we call elements of ⟨MSC⟩ "MSC terms" or "MSCs" for short. The intuitive interpretations of the syntactic elements in ⟨MSC⟩ is as follows: **empty** and **any** represent the absence of, and any form of interaction, respectively. ⟨MSG⟩ denotes a message specification; it consists of a channel name (defined by ⟨CHNAME⟩, a text string) and a message header (represented by ⟨MSGH⟩).

$$\langle\text{MSG}\rangle ::= \langle\text{CHNAME}\rangle\triangleright\langle\text{MSGH}\rangle$$

A message header consists of a message name (defined by ⟨MSGNAME⟩, a text string) or of a message name followed by a comma-separated list of formal message parameter names within parentheses:

$$
\begin{aligned}
\langle\text{MSGH}\rangle ::= \; & \langle\text{MSGNAME}\rangle \\
\| \; & \langle\text{MSGNAME}\rangle(\{\langle\text{FPNAME}\rangle\}^{+}_{,})
\end{aligned}
$$

We discuss the treatment of message parameters in Appendix A. For the most part of this thesis, however, we will make use of messages without parameters only.

⟨GMSC⟩ represents guarded MSCs; they consist of a guard specification (defined by the nonterminal ⟨GUARD⟩, a text string) and the guarded interaction description.

$$\langle\text{GMSC}\rangle ::= \quad \langle\text{GUARD}\rangle{:}\langle\text{MSC}\rangle$$

Operators ; , | , and $\sim$ denote the sequencing of, the alternative between, and the interleaving of the operand interaction descriptions, respectively. ↑ indicates a repetition; the

loop specification (defined by ⟨LSPEC⟩) determines the number of repetitions either in the sense of MSC-96's **loop**-inline expression, or in the form of a guard condition.

$$
\begin{array}{rl}
\langle \text{LSPEC} \rangle ::= & <\langle \text{NATI} \rangle> \\
\| & <\langle \text{NATI} \rangle,\langle \text{NATI} \rangle> \\
\| & <*> \\
\| & <\langle \text{GUARD} \rangle>
\end{array}
$$

⟨NATI⟩ represents either a natural number (defined by ⟨NAT⟩, a number string), or the infinity symbol ∞.

$$
\begin{array}{rl}
\langle \text{NATI} \rangle ::= & \langle \text{NAT} \rangle \\
\| & \infty
\end{array}
$$

$\otimes$ and $\mapsto$ represent the join and trigger composition of two interaction descriptions. The join of two MSCs corresponds to the interleaving of the interaction sequences they represent with the exception that common messages on common channels synchronize. The trigger composition of two MSCs expresses the property that whenever the interactions specified by the first have occurred the interactions specified by the second are inevitable.

$\rightarrow$ and $\stackrel{\langle \text{MSG} \rangle}{\rightarrow}$ represent MSC referencing and preemption, respectively. A reference to an MSC X is semantically equivalent to the interaction sequence represented by X. Preempting one MSC by another means that upon occurrence of a certain message the interaction sequence specified by the first MSC stops immediately and is continued by the interaction sequence specified by the second MSC. Preemption is of particular importance for the specification of exception and interrupt mechanisms. To allow specification of "self-preemption", i.e. the restarting of an interaction upon occurrence of a message, we use the $\Uparrow_{\langle \text{MSG} \rangle}$ notation.

We omit the obvious definitions of the syntactic categories ⟨MSCNAME⟩, ⟨CHNAME⟩, ⟨MSGNAME⟩, ⟨GUARD⟩, and ⟨NAT⟩; we assume all of them to define text strings. Furthermore, we use parentheses freely to group terms in ⟨MSC⟩, and let the repetition operators (loops and preemptive loops) bind stronger than the other operators.

4.4. Denotational MSC-Semantics

In this section we introduce the formal, denotational semantics for the MSC dialect whose textual syntax we have given in Section 4.3.

Intuitively, we associate with a given MSC a set of channel and state valuations, i.e. a set of system behaviors according to the system model we have introduced in Section 4.2. Put another way, we interpret an MSC as a constraint at the possible behaviors of the

system under consideration. More precisely, with every $\alpha \in \langle\text{MSC}\rangle$ and every $u \in \mathbb{N}_\infty$ we associate a set $[\![\alpha]\!]_u \in \mathcal{P}((\tilde{C} \times S)^\infty \times \mathbb{N}_\infty)$; any element of $[\![\alpha]\!]_u$ is a pair of the form $(\varphi, t) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty$. The first constituent, φ, of such a pair describes an infinite system behavior. u and the pair's second constituent, t, describe the time interval within which α constrains the system's behavior. Intuitively, u corresponds to the "starting time" of the behavior represented by the MSC; t indicates the time point when this behavior has finished. Hence, outside the time interval specified by u and t the MSC α makes no statement whatsoever about the interactions and state changes happening in the system (cf. Figure 4.2).

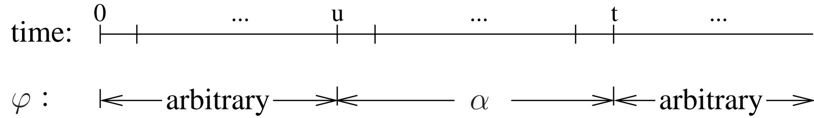


Figure 4.2.: MSC α constrains system behavior φ only over time interval $[u, t]$

Our motivation for using u as a parameter of the semantics, and for returning t as a "result" parameter in the semantics definition is twofold:

1. we view an individual MSC as a representation of (part of) a system execution. Without further information – MSCs do not contain explicit absolute timing information – we cannot say in advance at what time during the system execution the behavior modeled by an MSC starts; this explains parameter u. Similarly, because of the lack of explicit timing information (say, between two message occurrences), we cannot say in advance at what time the behavior modeled by an MSC is over; this explains parameter t.

2. the use of the lower time-bound u as a parameter eases the semantics definition; sequential composition and preemption (see below) demonstrate this benefit.

Definition 1 (Behavior "Beyond Infinity") To model that we cannot observe (or constrain) system behavior "beyond infinity" we define that for all $\varphi \in (\tilde{C} \times S)^\infty$, $\alpha \in \langle\text{MSC}\rangle$, and $t \in \mathbb{N}_\infty$ the following two predicates hold:

$$\begin{aligned} &(\varphi, t) \in [\![\alpha]\!]_\infty \\ &(\varphi \uparrow \infty, t) \in [\![\alpha]\!]_\infty \end{aligned} \tag{4.1}$$

□

The behavior modeled by an MSC is independent of the time point at which the behavior starts, i.e. for all $\varphi \in (\tilde{C} \times S)^\infty$, $u, n \in \mathbb{N}$, $t \in \mathbb{N}_\infty$, and $\alpha \in \langle\text{MSC}\rangle$ we have:

$$(\varphi, t + n) \in [\![\alpha]\!]_{u+n} \equiv (\varphi \uparrow n, t) \in [\![\alpha]\!]_u \tag{4.2}$$

This means that our model is independent of absolute time (see Appendix B.1 for the proof).

We assume given a relation $MSCR \subseteq \langle \text{MSCNAME} \rangle \times \langle \text{MSC} \rangle$, which associates MSC names with their interaction descriptions. We expect $MSCR$ to be the result of parsing all of a given MSC document's MSC definitions. For every MSC definition **msc** $X = \alpha$ in the MSC document we assume the existence of an entry (X, α) in $MSCR$. For simplicity we require the MSC term associated with an MSC name via $MSCR$ to be unique, i.e.:

$$\langle \forall X \in \langle \text{MSCNAME} \rangle, \alpha, \beta \in \langle \text{MSC} \rangle :: (X, \alpha) \in MSCR \wedge (X, \beta) \in MSCR \quad \Rightarrow \quad \alpha \doteq \beta \rangle$$

where we denote the syntactic equivalence of MSCs α and β by $\alpha \doteq \beta$.

In some of the semantic definitions we introduce here and in Appendix A we need to determine the messages and channels contained in an MSC. To that end, we define function $msgs : \langle \text{MSC} \rangle \rightarrow \mathcal{P}(\langle \text{MSG} \rangle)$ inductively as follows:

$$\begin{array}{lcl}
msgs.\mathbf{empty} & \stackrel{\mathsf{def}}{=} & \emptyset \\
msgs.\mathbf{any} & \stackrel{\mathsf{def}}{=} & \{ch \triangleright m : ch \in C \ \wedge \ m \in M\} \\
msgs.ch \triangleright m & \stackrel{\mathsf{def}}{=} & \{ch \triangleright m\} \\
msgs.(\alpha \dagger \beta) & \stackrel{\mathsf{def}}{=} & msgs.\alpha \cup msgs.\beta, \ \text{for } \dagger \in \{\, ; \, , \, | \, , \sim , \otimes , \ \mapsto \} \\
msgs.(p_K : \alpha) & \stackrel{\mathsf{def}}{=} & msgs.\alpha \\
msgs.\alpha\uparrow_{<..>} & \stackrel{\mathsf{def}}{=} & msgs.\alpha \\
msgs.(\rightarrow X) & \stackrel{\mathsf{def}}{=} & \begin{cases} msgs.\alpha & \text{if } (X, \alpha) \in MSCR \\ \{ch \triangleright m : ch \in C \ \wedge \ m \in M\} & \text{if } \neg \langle \exists \alpha :: (X, \alpha) \in MSCR \rangle \end{cases} \\
msgs.(\alpha \stackrel{ch \triangleright m}{\rightarrow} \beta) & \stackrel{\mathsf{def}}{=} & msgs.\alpha \cup msgs.\beta \cup \{ch \triangleright m\} \\
msgs.\alpha\Uparrow_{ch \triangleright m} & \stackrel{\mathsf{def}}{=} & msgs.\alpha \cup \{ch \triangleright m\}
\end{array}$$

The semantics mapping $[\![.]\!]_u$ induces an equivalence relation on MSC terms in the usual way:

Definition 2 (Semantic Equivalence of MSCs) We define the semantic equivalence of two MSCs $\alpha, \beta \in \langle \text{MSC} \rangle$ with respect to time $u \in \mathbb{N}_\infty$, written $\alpha \ \equiv_u \ \beta$, by

$$\alpha \ \equiv_u \ \beta \quad \stackrel{\mathsf{def}}{=} \quad [\![\alpha]\!]_u = [\![\beta]\!]_u \qquad \Box$$

In the following paragraphs we define the MSC semantics by means of structural induction over the grammar from Section 4.3. Along the way we also state several properties of the operators we introduce. For formal proofs of these properties we refer the reader to Appendix B. Where appropriate we show the graphical notation corresponding to an element of the textual syntax.

Empty MSC For any time $u \in \mathbb{N}_\infty$ **empty** describes arbitrary system behavior that starts and ends at time u. Formally, we define the semantics of **empty** as follows:

$$[\![\mathbf{empty}]\!]_u \stackrel{\mathsf{def}}{=} \{(\varphi, u) : \varphi \in (\tilde{C} \times S)^\infty\}$$

empty is the neutral element with respect to sequential composition, interleaving, and the join of MSCs (see below for the corresponding definitions), i.e. for all $\alpha \in \langle\text{MSC}\rangle$ each of the following equivalences holds:

$$\begin{array}{rclrcl} \textbf{empty} \,;\, \alpha & \equiv_u & \alpha \qquad & \alpha \,;\, \textbf{empty} & \equiv_u & \alpha \\ \textbf{empty} \sim \alpha & \equiv_u & \alpha \qquad & \alpha \sim \textbf{empty} & \equiv_u & \alpha \\ \textbf{empty} \otimes \alpha & \equiv_u & \alpha \qquad & \alpha \otimes \textbf{empty} & \equiv_u & \alpha \end{array}$$

Arbitrary Interactions MSC **any** describes completely arbitrary system behavior; there is neither a constraint on the allowed interactions and state changes, nor a bound on the time until the system displays arbitrary behavior:

$$[\![\textbf{any}]\!]_u \stackrel{\text{def}}{=} \{(\varphi, t) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty : t \geq u\}$$

any subsumes all possible behavior, i.e. for all $\alpha \in \langle\text{MSC}\rangle$ we have:

$$[\![\alpha]\!]_u \subseteq [\![\textbf{any}]\!]_u$$

any has no direct graphical representation; we use it to resolve unbound MSC references (see below).

Single Message An MSC that represents the occurrence of message m on channel ch (cf. Figure 4.3 (a)) constrains the system behavior until the minimum time such that this occurrence has happened:

$$[\![ch \triangleright m]\!]_u \stackrel{\text{def}}{=} \{(\varphi, t) \in (\tilde{C} \times S)^\infty \times \mathbb{N} : t = \min\{v : v > u \wedge m \in \pi_1(\varphi).v.ch\}\}$$

Because we disallow pairs (φ, ∞) in $[\![ch \triangleright m]\!]_u$ we require the message to occur eventually (within finite time). This corresponds with the typical intuition we associate with MSCs: the depicted messages do occur within finite time.

We add the channel identifier explicitly to the label of a message arrow in the graphical representation; this is useful in situations where a component has more than one communication path to another component. The channel names used in message specifications, and the channel names appearing in an SSD (such as the one shown in Figure 4.1) must be consistent, i.e. a message can occur only on a channel between two components if such a channel exists in the corresponding SSD. If we employed ROOM's actor classes and bindings, or the UML's class and object diagrams to specify the communication links between components, we would have to map ports and bindings, or associations, respectively, to 'virtual' channel names.

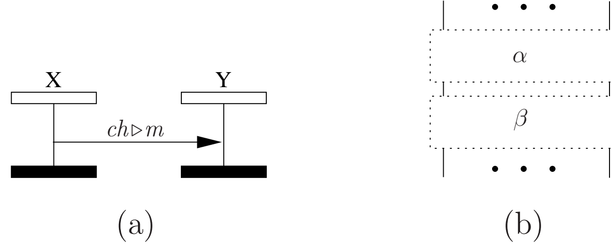


Figure 4.3.: Single message exchange and sequential MSC composition

Sequential Composition The semantics of the semicolon operator is sequential composition: given two MSCs α and β the MSC $\alpha\,;\,\beta$ denotes that we can separate each system behavior in a prefix and a suffix such that α describes the prefix and β describes the suffix (cf. Figure 4.3 (b)):

$$[\![\alpha\,;\,\beta]\!]_u \stackrel{\text{def}}{=} \{(\varphi, t) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty : \langle \exists t' \in \mathbb{N}_\infty :: (\varphi, t') \in [\![\alpha]\!]_u \wedge (\varphi, t) \in [\![\beta]\!]_{t'} \rangle\}$$

This definition differs from what MSC-96 calls "weak sequential composition" or weak sequencing for short. In MSC-96 the weak sequencing of independent interaction sequences results in their interleaving. Our definition keeps the intuition behind the term "sequencing" and forbids the interleaving of the interactions of the operand MSCs. For unordered interactions we introduce a separate interleaving concept, below. In Section 4.5.3 we discuss our reason for avoiding weak sequencing in detail.

Sequential composition is associative, and distributes over the alternative operator, i.e. for all $\alpha, \beta, \gamma \in \langle \text{MSC} \rangle$ we have:

$$(\alpha\,;\,\beta)\,;\,\gamma \equiv_u \alpha\,;\,(\beta\,;\,\gamma)$$

and

$$\begin{aligned} \alpha\,;\,(\beta \mid \gamma) &\equiv_u (\alpha\,;\,\beta) \mid (\alpha\,;\,\gamma) \\ (\beta \mid \gamma)\,;\,\alpha &\equiv_u (\beta\,;\,\alpha) \mid (\gamma\,;\,\alpha) \end{aligned}$$

Guarded MSC Let $K \subseteq P$ be a set of instance identifiers. By p_K we denote a predicate over the state spaces of the instances in K. Let $[\![p_K]\!] \in \mathcal{P}(S)$ denote the set of states in which p_K holds. Then we define the semantics of the guarded MSC $p_K : \alpha$ as the set of behaviors whose state projection fulfills p_K at time u, and whose interactions proceed as described by MSC α:

$$[\![p_K : \alpha]\!]_u \stackrel{\text{def}}{=} \{(\varphi, t) \in [\![\alpha]\!]_u : \pi_2(\varphi).u \in [\![p_K]\!]\}$$

We require p_K to hold only at instant u. This allows arbitrary state changes from time u on. In particular, at no other point within the time interval covered by α can we assume that p_K still holds.

We can conjoin multiple guards of an MSC into a single one, i.e. for all $p, q \in \langle\text{GUARD}\rangle$ and $\alpha \in \langle\text{MSC}\rangle$ we have:

$$p : (q : \alpha) \ \equiv_u \ (p \ \wedge \ q) : \alpha$$

Moreover, the boolean constants true and false hold in all and none of the system's executions, respectively, i.e. for all $\alpha \in \langle\text{MSC}\rangle$ we have:

$$\begin{aligned} [\![\text{true} : \alpha]\!]_u &= [\![\alpha]\!]_u \\ [\![\text{false} : \alpha]\!]_u &= \emptyset \end{aligned}$$

As Figure 4.4 (a) shows, where $K = \{X_1, \ldots, X_n\}$ for some $n \in \mathbb{N}$, we use the condition symbol from MSC-96 to represent guards in MSCs. Put another way, we have assigned a meaning to conditions by treating them as guards. In the semantics for MSC-96 the meaning of conditions is void (cf. [IT98]), besides the composition constraints they impose on references in HMSCs (cf. [IT96]).

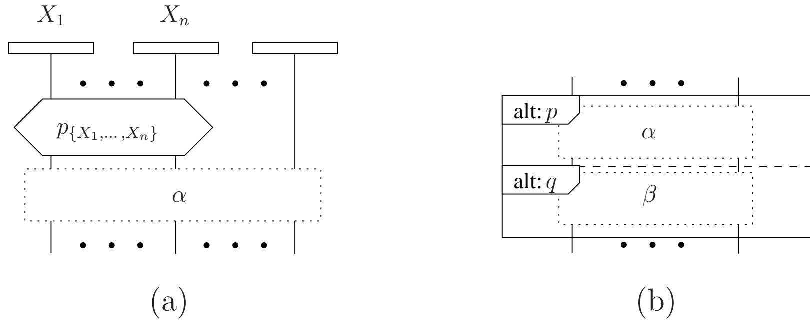


Figure 4.4.: Guarded MSC and alternative

Alternative An alternative denotes the union of the semantics of its two operand MSCs. The operands must be guarded MSCs; the disjunction of their guards must yield true. Thus, for $\alpha = p : \alpha'$ and $\beta = q : \beta'$ for $\alpha', \beta' \in \langle\text{MSC}\rangle$ and $p, q \in \langle\text{GUARD}\rangle$ with $p \vee q \equiv \text{true}$ we define:

$$[\![\alpha \mid \beta]\!]_u \stackrel{\text{def}}{=} [\![\alpha]\!]_u \cup [\![\beta]\!]_u$$

For guards p and q with $p \wedge q \equiv \text{true}$ the alternative expresses a nondeterministic choice.

The alternative operator is symmetric and associative, i.e. for all $\alpha, \beta, \gamma \in \langle\text{MSC}\rangle$ we have:

$$\begin{aligned} \alpha \mid \beta &\equiv_u \beta \mid \alpha \\ \alpha \mid (\beta \mid \gamma) &\equiv_u (\alpha \mid \beta) \mid \gamma \end{aligned}$$

Our treatment of the alternative operator differs from the approach taken in MSC-96. There, choices are resolved at the latest possible moment, i.e. at the moment where the alternatives differ ([IT98] terms this "delayed choice"). This corresponds to regarding MSCs as representations of execution trees in the sense of branching time temporal logics, such as CTL and CTL* (cf. [Tho90, Eme90]). Each element in $[\![.]\!]_u$, on the other hand, represents a single system execution, with all choices fixed. This corresponds to regarding MSCs as representations of execution sequences in the sense of linear time temporal logics, such as PLTL (cf. [Eme90]). In Section 4.5.4 we study the relationship between temporal logics and our semantics definition further.

Our motivation for choosing the simpler alternative construct is that it leads to easier equivalence and refinement notions. While the definition of MSC equivalence and refinement in MSC-96 requires an appeal to bisimulation, we can use set equality and set inclusion instead (cf. Chapter 5).

MSC-96 provides no means for guiding the choice among alternatives; this corresponds to setting both p and q to true in our definition. In the graphical representation of alternatives (cf. Figure 4.4 (b)) we add the guarding predicates after a colon to the keyword **alt** in the respective compartment of the alternative box.

Interleaving With interleaving we capture the idea of causally unrelated interactions. The result of interleaving two operand MSCs α and β is a set of behaviors whose elements contain the interactions of α and β in any order (cf. Figure 4.5 (a)).

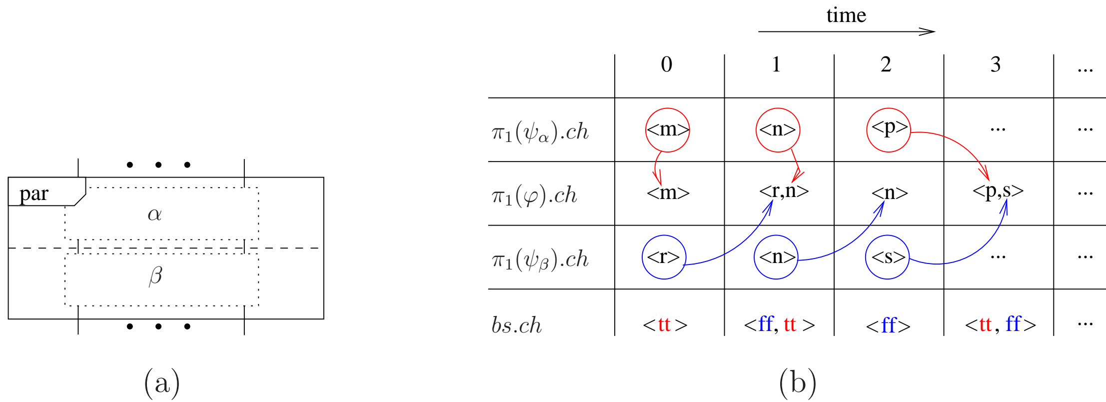


Figure 4.5.: MSC interleaving[2]

The basic idea behind the semantics definition for $\alpha \sim \beta$ is as follows. Let (φ, t) be an element of $[\![\alpha \sim \beta]\!]_u$. If we remove α's contribution from φ, then the one of β must still be present in what is left; the same holds with the roles of α and β interchanged. In other words, we can reconstruct the behaviors of α and β independently from the behaviors of their interleaving.

[2] "tt" and "ff" abbreviate true and false, respectively.

As an example, assume given the two MSC definitions

$$\alpha \;=\; ch \triangleright m \;;\; ch \triangleright n \;;\; ch \triangleright p$$

and

$$\beta \;=\; ch \triangleright r \;;\; ch \triangleright n \;;\; ch \triangleright s$$

Figure 4.5 (b) illustrates the relationship between the semantics of α and β, and the semantics of the interleaving $\alpha \sim \beta$. In this figure, time increases to the right. Columns indicate time points. Rows indicate the valuations of channel *ch* in exemplary elements of the semantics of α (first row), β (third row), and $\alpha \sim \beta$ (second row). The arrows indicate where the contribution of an operand ends up in this particular interleaving.

In the semantics definition for interleaving, we use an oracle $bs \in (C \rightarrow \mathbb{B}^*)^\infty$ that tells us for every time $t \in \mathbb{N}$ which of the elements of a channel valuation $cv \in C \rightarrow M^*$ are contributed by α, and which by β. In terms of the example from Figure 4.5 (b), where $bs.ch$ occupies the bottom row, the oracle encodes the information expressed by the arrows. Thus, the purpose of bs is to help us filter any behavior in the semantics of an interleaving $\alpha \sim \beta$ independently for the contributions of α and β. To that end, we specify the two helper functions

$$\mathit{filter} : (\tilde{C}^\infty \times (C \rightarrow \mathbb{B}^*)^\infty \times \mathbb{B}) \rightarrow \tilde{C}^\infty$$

and

$$h : (M^* \times \mathbb{B}^* \times \mathbb{B}) \rightarrow M^*$$

Intuitively, function h filters the contents of one particular channel at one particular point in time. *filter* filters entire interaction histories. For all $\psi \in \tilde{C}^\infty$, $bs \in (C \rightarrow \mathbb{B}^*)^\infty$, $bs' \in \mathbb{B}^*$, $b, bv \in \mathbb{B}$, $x \in M$, and $xs \in M^*$ we define:

$$\langle \forall t \in \mathbb{N}, ch \in C :: (\mathit{filter}.\psi.bs.bv).t.ch \;=\; h.(\psi.t.ch).(bs.t.ch).bv \rangle$$

and

$$\begin{aligned} h.\langle\rangle.\langle\rangle.bv &= \langle\rangle \\ h.(x \frown xs).(b \frown bs').bv &= \begin{cases} x \frown h.xs.bs'.bv & \text{if } bv = b \\ h.xs.bs'.bv & \text{else} \end{cases} \end{aligned}$$

In the definition of h we have assumed the validity of $|xs| = |bs'|$, i.e. the length of the stream to be filtered and the length of the corresponding oracle must coincide. The function application $h.xs.bs'.bv$ yields the projection of the finite stream of messages xs onto those elements $xs.i$ for which $bs'.i$ equals bv $(0 \leq i < |xs|)$. Our intuition for the parameter

bv is to let it indicate which operand's contribution we currently consider; for α we use $bv = \text{true}$, for β we use $bv = \text{false}$.

The function *filter* extends h over all time points and channels. Thus, *filter* allows us to project an entire execution ψ onto the respective operand's contribution.

With these preliminaries in place, we define the semantics of the interleaving operator as follows:

$$
\begin{aligned}
[\![\alpha \sim \beta]\!]_u \stackrel{\mathsf{def}}{=} \{ & (\varphi, t) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty : \\
& \langle \exists bs, t_1, t_2, \psi_1, \psi_2 : t_1, t_2 \in \mathbb{N}_\infty \wedge \psi_1, \psi_2 \in (\tilde{C} \times S)^\infty : \\
& \quad bs \in (C \to \mathbb{B}^*)^\infty \\
& \wedge \ \langle \forall t' \in \mathbb{N}, ch \in C :: |bs.t'.ch| = |(\pi_1(\varphi) \uparrow u).t'.ch| \rangle \\
& \wedge \ ((\mathit{filter}.(\pi_1(\varphi) \uparrow u).bs.\,\text{true}, (\pi_2(\varphi) \uparrow u)) \frown \psi_1, t_1) \in [\![\alpha]\!]_0 \\
& \wedge \ ((\mathit{filter}.(\pi_1(\varphi) \uparrow u).bs.\,\text{false}, (\pi_2(\varphi) \uparrow u)) \frown \psi_2, t_2) \in [\![\beta]\!]_0 \\
& \wedge \ t = u + \max(t_1, t_2) \rangle \}
\end{aligned}
$$

The first two conjuncts in this definition ensure the oracle's type-correctness; for every time $t \in \mathbb{N}$ bs fixes the origin (α or β) for precisely the messages appearing in $\pi_1(\varphi)$ at time t. Together, the third and fourth conjunct require that we can extract the behaviors represented by α and β independently of each other from φ. This captures our intuitive understanding of the interleaving operator.

The interleaving operator is symmetric and associative, i.e. for all $\alpha, \beta, \gamma \in \langle \text{MSC} \rangle$ we have:

$$
\begin{aligned}
\alpha \sim \beta \quad &\equiv_u \quad \beta \sim \alpha \\
\alpha \sim (\beta \sim \gamma) \quad &\equiv_u \quad (\alpha \sim \beta) \sim \gamma
\end{aligned}
$$

Join The join $\alpha \otimes \beta$ of two operand MSCs α and β is similar to their interleaving with the exception that the join identifies common messages, i.e. messages on the same channels with identical labels in both operands. MSC-96 does not offer an operator with a similar semantics. In our graphical representation (cf. Figure 4.6 (a)) we use the same representation as for interleaving, with the exception that instead of the keyword **par** we use the keyword **join**.

As an example, consider again the two MSC definitions

$$\alpha = ch \triangleright m \,;\, ch \triangleright n \,;\, ch \triangleright p$$

and

$$\beta = ch \triangleright r \,;\, ch \triangleright n \,;\, ch \triangleright s$$

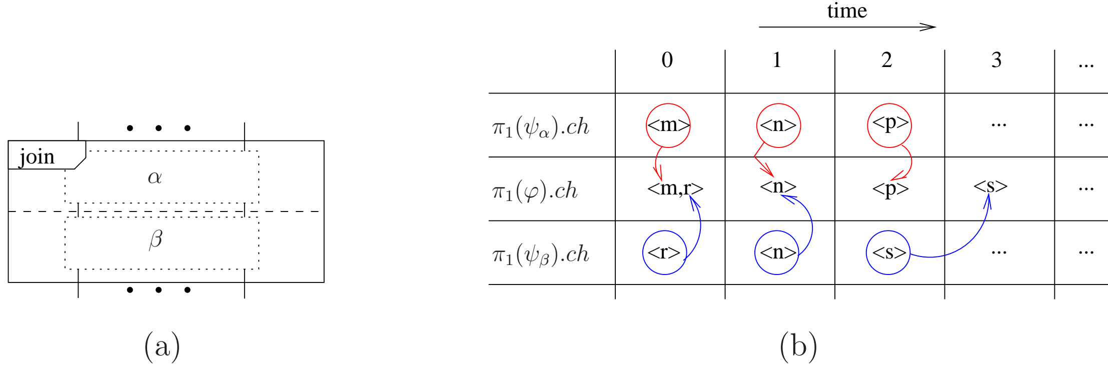


Figure 4.6.: MSC join

α and β share the occurrence of message n on channel ch. Figure 4.6 (b) illustrates the relationship between the semantics of α and β, and the semantics of the join $\alpha \otimes \beta$ in the style we used already to explain the intuition behind interleaving (see above). Here, the top, bottom, and middle rows contain exemplary elements of the semantics of α, β, and $\alpha \otimes \beta$, respectively. The arrows indicate where the contribution of an operand ends up in this particular join. Message n occurs only once in Figure 4.6 (b), whereas it occurs twice in Figure 4.5 (b).

The idea behind our semantics definition for the join operator is as follows. Let (φ, t) be an element of $[\![\alpha \otimes \beta]\!]_u$. We require that φ is an execution of both α and β, i.e. both α and β contribute the behavior they represent to their join. There may not be any redundancy in the semantics of the join with respect to messages shared by α and β. In other words, if we drop one such message anywhere in $\varphi|_{[u,t]}$, then the remaining execution is neither an execution of α, nor of β. This captures that α and β contribute only one copy per occurrence of a common message. This leads to the following definition of the join semantics:

$$
\begin{aligned}
[\![\alpha \otimes \beta]\!]_u \overset{\text{def}}{=} \{(\varphi, t) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty :\\
&\langle \exists t_1, t_2 :: (\varphi, t_1) \in [\![\alpha]\!]_u \wedge (\varphi, t_2) \in [\![\beta]\!]_u \wedge t = \max(t_1, t_2)\rangle\\
&\wedge \langle \forall X \in (msgs.\alpha \cap msgs.\beta)^*, \psi \in (\tilde{C} \times S)^\infty, ch \in C, t' \in [u,t] \cap \mathbb{N} ::\\
&\quad ((X \neq \langle\rangle) \wedge (\pi_1(\psi).t'.ch = \pi_1(\varphi).t'.ch \setminus X))\\
&\quad \Rightarrow \langle \forall t'' \in \mathbb{N} :: (\psi, t'') \notin [\![\alpha]\!]_u \wedge (\psi, t'') \notin [\![\beta]\!]_u \rangle\rangle\}
\end{aligned}
$$

In this definition we use the notation $m \setminus n$ as a shorthand for $\{x \notin n\}©m$, i.e. for the stream obtained from m by dropping all elements that do also appear in n.

The second outer conjunct of this definition ensures that we *cannot* reconstruct the behaviors of α and β independently from the behaviors of their join, if the two MSCs have messages in common. This distinguishes the join clearly from the interleaving of α and

β, if $msgs.\alpha \cap msgs.\beta \neq \emptyset$ holds. In this state of affairs, we call α and β *non-orthogonal* (or *overlapping*); if $msgs.\alpha \cap msgs.\beta = \emptyset$ holds, then we call α and β *orthogonal* (or *non-overlapping*).

The definition of the join operator is quite restrictive. As an example, consider the two MSCs

$$\alpha = c \triangleright m \,;\, c \triangleright n$$

and

$$\beta = c \triangleright n \,;\, c \triangleright m$$

which define two different orderings of the messages $c \triangleright m$ and $c \triangleright n$. What is the join of α and β in this case?

It is easy to see that if $(\varphi, t) \in [\![\alpha \otimes \beta]\!]_u$ holds, then $\pi_1(\varphi).c$ must contain at least two copies of either m or n within the time interval $[u, t]$. Otherwise, $\pi_1(\varphi).c$ would not model either α or β, and the first outer conjunct of the join semantics would be false. However, if we drop only one of the duplicates of either m or n from $\pi_1(\varphi).c$, then the resulting channel assignment still models either α or β on a time interval starting at u. This violates the second outer conjunct of the join semantics. This shows that for α and β as given above we have $[\![\alpha \otimes \beta]\!]_u = \emptyset$. This corresponds with our intuitive understanding of identifying occurrences of common messages of the operands of the join operator.

Guided by this example we now define under what circumstances we consider two MSCs consistent with respect to the join operator.

Definition 3 (Consistency with respect to join) Let $\alpha, \beta \in \langle \text{MSC} \rangle$ be MSCs. If $[\![\alpha \otimes \beta]\!]_0 \neq \emptyset$ holds, then we call α and β *consistent* (with respect to join); otherwise, we call α and β *inconsistent* (with respect to join). □

In Chapter 7 we will describe a constructive procedure for determining whether or not two MSCs are consistent.

The join operator allows us to represent different aspects of a communication in different MSCs. We could, for instance, represent the interactions between a customer and an ATM for a withdrawal from the customer's bank account in one MSC (showing only the axes representing the customer and the ATM), and the interactions between the ATM and the component representing the bank account in another.

Thus, the importance of join is not so much in the construction of individual MSCs but in the composition of MSCs that display the same instances in different roles.

The join operator is symmetric and associative, i.e. for all $\alpha, \beta, \gamma \in \langle \text{MSC} \rangle$ we have:

$$\begin{aligned} \alpha \otimes \beta &\equiv_u \beta \otimes \alpha \\ \alpha \otimes (\beta \otimes \gamma) &\equiv_u (\alpha \otimes \beta) \otimes \gamma \end{aligned}$$

Loops We introduce four classes of loop constructs:

- *guarded loops*, where the validity of a guard determines further execution of the loop body;
- *bounded loops*, where a lower and upper bound from set $\mathbb{N}$ statically determine a finite number of repetitions;
- *unbounded loops*, that represent any finite number of repetitions;
- *infinite loops*, where the loop body occurs an infinite number of times.

MSC-96 offers only bounded and infinite loops, and combinations thereof. Because, in contrast to MSC-96, we have access to component states in our semantic framework we can give a simple definition for guarded loops. The existence of unbounded loops eases the definition of liveness properties as we shall discuss in Chapter 6. In the following paragraphs we introduce all forms of interactions and their combinations as allowed by MSC-96.

We start by defining the semantics of guarded loops, i.e. loops of the form $\alpha\uparrow_{<p>}$ where $p \in \langle\text{GUARD}\rangle$ denotes a guard. The semantics $[\![\alpha\uparrow_{<p>}]\!]_u$ of the guarded loop is the greatest fixpoint (with respect to set inclusion) of the following equation:

$$[\![\alpha\uparrow_{<p>}]\!]_u = [\![(p : (\alpha \,;\, \alpha\uparrow_{<p>})) \mid ((\neg p) : \mathbf{empty})]\!]_u \tag{4.3}$$

The fixpoint exists because of the monotonicity of its defining equation (with respect to set inclusion); in Appendix B.1.3 we turn Equation (4.3) into a corresponding set transformer, which makes the required monotonicity explicit.

The semantics of a loop whose guard is false equals the semantics of **empty**, i.e. for all $\alpha \in \langle\text{MSC}\rangle$ we have:

$$\alpha\uparrow_{<\text{false}>} \;\equiv_u\; \mathbf{empty}$$

On the basis of guarded repetition we can easily define the semantics of α's infinite repetition as follows:

$$[\![\alpha\uparrow_{<\infty,\infty>}]\!]_u \;\stackrel{\mathsf{def}}{=}\; [\![\alpha\uparrow_{<\text{true}>}]\!]_u$$

For the description of the combinations of bounded and infinite loops allowed by MSC-96 we first introduce a syntactic abbreviation for the finite repetition of a loop body α. Intuitively, for any $i \in \mathbb{N}$ by α^i we denote the i-fold finite sequential composition of copies of α. More precisely, we define

$$\begin{array}{lcll} \alpha^0 & \stackrel{\mathsf{def}}{=} & \mathbf{empty} & \\ \alpha^{i+1} & \stackrel{\mathsf{def}}{=} & \alpha \,;\, \alpha^i & \text{for } i \in \mathbb{N} \wedge i \geq 0 \end{array}$$

In repetitions of the form $\alpha\uparrow_{<m,n>}$ the bounds $m, n \in \mathbb{N}_\infty$ determine how often the loop body α occurs. We present the definition by case distinction on the values of the lower bound m and the upper bound n. Note the close correspondence between these definitions and the loop constructs of MSC-96 (see Section 2.2):

$$[\![\alpha\uparrow_{<m,n>}]\!]_u \stackrel{\text{def}}{=} \begin{cases} \bigcup_{0\leq i\leq n}[\![\alpha^i]\!]_u & \text{if } m = 0 \wedge n \in \mathbb{N} \\ \bigcup_{i\in\mathbb{N}}[\![\alpha^i]\!]_u \cup [\![\alpha\uparrow_{<\infty,\infty>}]\!]_u & \text{if } m = 0 \wedge n = \infty \\ [\![\alpha^m \,;\, \alpha\uparrow_{<0,n-m>}]\!]_u & \text{if } m, n \in \mathbb{N} \wedge 0 < m \leq n \\ [\![\alpha^m \,;\, \alpha\uparrow_{<0,\infty>}]\!]_u & \text{if } m \in \mathbb{N} \wedge 0 < m \wedge n = \infty \\ [\![\mathbf{empty}]\!]_u & \text{if } (m, n \in \mathbb{N} \wedge n < m) \vee (n \in \mathbb{N} \wedge m = \infty) \end{cases}$$

Furthermore, we carry over two syntactic abbreviations from MSC-96. We define a shortcut for the n-fold repetition of α, where $n \in \mathbb{N}_\infty$, by

$$[\![\alpha\uparrow_{<n>}]\!]_u \stackrel{\text{def}}{=} [\![\alpha\uparrow_{<n,n>}]\!]_u$$

The semantics of the shortcut for any positive or infinite number of repetitions of α is as follows:

$$[\![\alpha\uparrow\;]\!]_u \stackrel{\text{def}}{=} [\![\alpha\uparrow_{<1,\infty>}]\!]_u$$

Unbounded finite repetition has the following semantics:

$$[\![\alpha\uparrow_{<*>}]\!]_u \stackrel{\text{def}}{=} \bigcup_{n\in\mathbb{N}}[\![\alpha\uparrow_{<0,n>}]\!]_u$$

MSC-96 has no repetition operator for expressing unbounded finite repetition. This operator is, however, one way of specifying liveness properties with MSCs (cf. Section 4.5).

From the definitions of the various loop constructs we can easily deduce for any $\alpha \in \langle\text{MSC}\rangle$, and $m, n \in \mathbb{N}_\infty$ the two equivalences

$$\alpha \,;\, \alpha\uparrow_{<m,n>} \;\equiv_u\; \alpha\uparrow_{<m+1,n+1>}$$

and

$$\alpha\uparrow_{<m,n>} \,;\, \alpha \;\equiv_u\; \alpha\uparrow_{<m+1,n+1>}$$

Graphically, we extend the syntax of MSC-96 for representing loops by allowing all forms of loop specifiers we have introduced above, i.e. in the loop definition of Figure 4.7 (a) we allow $<L> \in \{<p>, <m,n>, <*>\}$, where p is a guard and $m, n \in \mathbb{N}_\infty$.

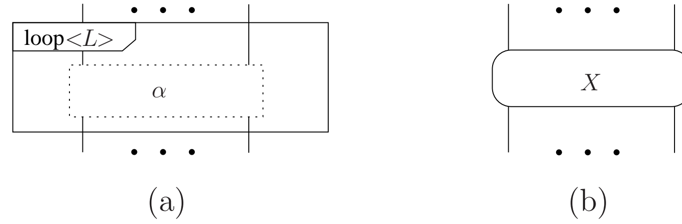


Figure 4.7.: MSC loops and reference

References If an MSC named X exists in the given MSC document, i.e. there exists a pair $(X, \alpha) \in \mathit{MSCR}$ for some $\alpha \in \langle\text{MSC}\rangle$, then the semantics of a reference to X equals the semantics of α. Otherwise, i.e. if no adequate MSC definition exists, we associate the meaning of **any** with the reference (cf. Figure 4.7 (b)):

$$[\![\to X]\!]_u \stackrel{\text{def}}{=} \begin{cases} [\![\alpha]\!]_u & \text{if } (X, \alpha) \in \mathit{MSCR} \\ [\![\textbf{any}]\!]_u & \text{else} \end{cases}$$

To identify **any** with an unbound reference has the advantage that we can understand the binding of references as a form of refinement (cf. Section 5.2). The simple reference resolution scheme we use here allows acyclic references only. We can easily construct more complex reference mechanisms. We could, for instance, allow recursive MSC definitions, i.e. MSC definitions referring either directly or indirectly to themselves. Semantically this corresponds to defining and solving fixpoint equations over the set of given MSC definitions. However, within this document we deal with acyclic MSC references only and stick with the simple referencing scheme given above.

This simplifies the semantics definition, because we do not need to add further fixpoint equations. The price we pay for this simplification is restricted expressiveness; without recursive references the specification of an MSC α with the following property is impossible with the constructs introduced so far: for all $n \in \mathbb{N}$, α contains n occurrences of message x and n occurrences of message y such that all occurrences of x precede all occurrences of y. In favor of a simpler semantics we are willing to pay this price for the moment. In Chapter 6 we discuss how to define such properties through combinations of graphical and textual specifications.

Preemption Preemption is an important concept that occurs in various disguises in almost all kinds of systems, most often in the form of interrupt or exception handling. Consider, for instance, that we want to model a telephony call protocol. A typical event within every scenario is that, say, the caller suddenly hangs up the phone. We would expect that this ends any call scenario immediately and maybe results in tearing town the rest of the communication link in a certain order. However, as we have explained in Chapter 2, none of the existing MSC dialects provides adequate syntax and semantics for specifying such behavior. We solve this deficit by defining the concept of preemptive reference along the lines of the telephony scenario just outlined. We allow a message m (such as "putting

a phone on hook") to preempt a given MSC (such as "initiate phone call") and to cause continuation by a possibly different MSC (such as "tear down communication link").

Thus, the semantics of MSC $\alpha \overset{ch\triangleright m}{\rightarrow} \beta$ is equivalent to the one of α as long as message $ch\triangleright m$ has not occurred. From the moment in time at which $ch\triangleright m$ occurs, the MSC immediately switches its semantics to the one given by β:

$$
\begin{aligned}
[\![\alpha \overset{ch\triangleright m}{\rightarrow} \beta]\!]_u \overset{\text{def}}{=} \; & \{(\varphi, t) \in [\![\alpha]\!]_u : \langle \forall v \in \mathbb{N} : u \leq v \leq t : m \notin \pi_1(\varphi).v.ch \rangle\} \\
& \cup \{(\varphi, t) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty : \\
& \quad \langle \exists v : v \in \mathbb{N} : \\
& \quad\quad v = \min\{t' : t' > u \;\wedge\; m \in \pi_1(\varphi).t'.ch\} \\
& \quad \wedge (\varphi, v-1) \in [\![\alpha]\!]_u^{v-1} \\
& \quad \wedge (\varphi, t) \in [\![\beta]\!]_v \rangle\}
\end{aligned}
$$

Here we use the set $[\![\alpha]\!]_u^v \subseteq (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty$ for any $\alpha \in \langle \text{MSC} \rangle$, which is similar to $[\![\alpha]\!]_u$ except that each element of $[\![\alpha]\!]_u^v$ constrains the system behavior until time $v \in \mathbb{N}_\infty$:

$$
\begin{aligned}
[\![\alpha]\!]_u^v \overset{\text{def}}{=} \{ & (\varphi, v) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty : \\
& \langle \exists (\psi, t) : (\psi, t) \in [\![\alpha]\!]_u : \varphi|_{[u,v]} = \psi|_{[u,v]} \;\wedge\; v \leq t \rangle\}
\end{aligned}
$$

Graphically we use the same notation as for interleaving, with the exception that we label the upper box with the preempting message, followed by the keyword **preempts** (cf. Figure 4.8 (a)). The box at the top holds the MSC part that the message can preempt; the box at the bottom holds the MSC part describing the continuation once preemption has occurred. Often the content of the MSC part at the top will be more "elaborate" than the one at the bottom. Depending on the context, the handling of an exception, for instance, is often less involved than the code or protocol that "throws" the exception. In Section 4.6 we introduce a more convenient notation than what Figure 4.8 (a) suggests for such cases, in the context of HMSCs.

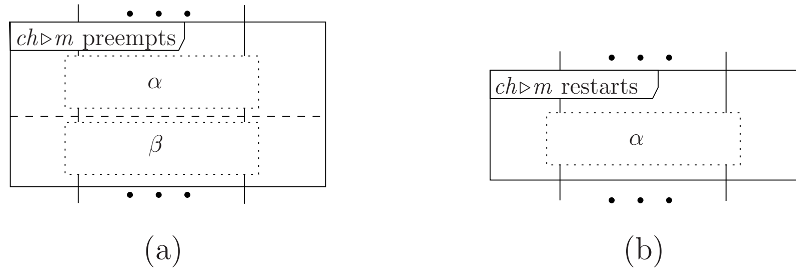


Figure 4.8.: MSC preemption and preemptive loop

The way we have defined the semantics of preemption was inspired by the notion of "catastrophe", introduced by C.A.R. Hoare in [Hoa85]. Note, however, that our preemption operator is more permissive with respect to the occurrence of the preemptive message than

catastrophe is with respect to the catastrophic interrupt event. In particular, in a preemption specification $\alpha \stackrel{ch \triangleright m}{\rightarrow} \beta$ we do *not* exclude the syntactic occurrence of $ch \triangleright m$ in α. In anticipation of the different MSC interpretations we discuss in Chapter 6, we mention that the closed world semantics (cf. Section 6.2.3) of the preemption operator comes close to the semantics of catastrophe.

Preemptive Loop The definition of MSC preemption above does not capture the restarting of an interaction in case of the occurrence of a certain message. To handle this case we define the notion of preemptive loop. The intuitive semantics of the preemptive loop of α for a given message $ch \triangleright m$ is that whenever $ch \triangleright m$ occurs α gets interrupted and then the interaction sequence proceeds as specified by α, again with the possibility for preemption. More precisely, we define $[\![\alpha \Uparrow_{ch \triangleright m}]\!]_u$ to equal the greatest fixpoint (with respect to set inclusion) of the following equation:

$$[\![\alpha \Uparrow_{ch \triangleright m}]\!]_u = [\![\alpha \stackrel{ch \triangleright m}{\rightarrow} (\alpha \Uparrow_{ch \triangleright m})]\!]_u \tag{4.4}$$

The fixpoint exists due to the monotonicity of its defining equation (with respect to set inclusion). Appendix B.1.3 contains the details for justifying this claim; there, we turn Equation (4.3) into a corresponding set transformer, which makes the required monotonicity explicit.

Graphically we use the same notation as for loops, with the exception that we label the box with the preempting message, followed by the keyword **restarts** (cf. Figure 4.8 (b)). For this situation we also introduce more convenient syntax in Section 4.6.

The preemptive loop operator we have defined above is a variation of the "restart"-operator introduced in [Hoa85]. As was the case with regular preemption, we do not exclude the syntactic occurrence of the preemptive message within the body of the preemptive loop.

Trigger Composition By means of the trigger composition operator we can express a temporal relationship between two MSCs α and β; whenever an interaction sequence corresponding to α has occurred in the system we specify, then the occurrence of an interaction sequence corresponding to β is inevitable:

$$\begin{aligned} [\![\alpha \mapsto \beta]\!]_u \stackrel{\mathsf{def}}{=} \{ & (\varphi, t) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty : \\ & \langle \forall t', t'' : \infty > t'' \geq t' \geq u : \\ & \quad (\varphi, t'') \in [\![\alpha]\!]_{t'} \Rightarrow \langle \exists t''' : \infty > t''' > t'' : (\varphi, t) \in [\![\beta]\!]_{t'''} \rangle \rangle \} \end{aligned}$$

The relevance of this operator is that it gives us a handle at defining certain liveness properties (cf. Sections 4.5 and 6.3) for MSC specifications. In fact, the liveness properties that we can specify using the trigger composition are a form of fairness constraints (cf. [Fra86]) at the executions of the system we model. In $\alpha \mapsto \beta$ (read: "α triggers β"

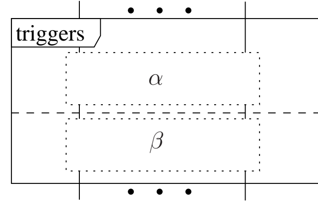


Figure 4.9.: Trigger composition

or "α leads to β") we call α and β the trigger (or the triggering MSC) and the triggered MSC, respectively.

Trigger composition is "transitive", i.e. for all $u, t \in \mathbb{N}_\infty$, $\varphi \in (\tilde{C} \times S)^\infty$, and $\alpha, \beta, \gamma \in \langle \text{MSC} \rangle$ we have:

$$\langle \exists t_1, t_2 : t_2 \geq t_1 : (\varphi, t_1) \in [\![\alpha \mapsto \beta]\!]_u \wedge (\varphi, t) \in [\![\beta \mapsto \gamma]\!]_{t_2} \rangle \Rightarrow (\varphi, t) \in [\![\alpha \mapsto \gamma]\!]_u$$

Graphically we use the same notation as for alternatives, with the exception that we label the box with the keyword **triggers** (cf. Figure 4.9). The top compartment holds the trigger, the bottom compartment holds the triggered MSC.

4.5. Discussion of the Semantics

In Section 4.4 we have defined the semantics for the MSC dialect we employ in this thesis. However, before we go about and use this semantic framework in the following chapters, we discuss some of its properties in this section.

First, we consider the well-definedness of the semantics function $[\![.]\!]_u$ (cf. Section 4.5.1), and give a characterization of the elements in $[\![\alpha]\!]_u$ for a given $\alpha \in \langle \text{MSC} \rangle$ (cf. Section 4.5.2).

Second, we study the relationship between the sequential composition and the interleaving operator in our semantics (cf. Section 4.5.3). This allows us to relate our approach to the standard semantics [IT98].

Third, to get a first impression on the expressiveness of our MSC dialect, we briefly relate temporal logic formulae with certain classes of MSC terms in Section 4.5.4.

Together, the topics we discuss in this section serve as "sanity checks" for the semantics foundation we have established in Section 4.4

4.5.1. Well-Definedness

Section 4.4 contains two recursive equations: (4.3), and (4.4). This induces the question whether the semantics we have presented is well-defined. Because all other definitions are

non-recursive, and involve basic set operations only, answering this question amounts to establishing that each of the recursive equations has a (unique) greatest fixpoint.

Proposition 1 (Well-Definedness of the Semantics) *The semantics of Section 4.4 is well defined. In particular, each of the recursive Equations (4.3) and (4.4) has a unique greatest fixpoint.* □

Proof See Appendix B.1.3. ■

4.5.2. General Observations

In Section 4.4 we have stated informally that for an MSC α the semantics function $[\![.]\!]_u$ yields the set of pairs $(\varphi, t) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty$ where φ is a system behavior and $[u, t]$ is the time interval in which α constrains φ. To support this intuition we observe that

- no MSC can "reset" time, i.e. $[\![.]\!]_u$ constrains the system behavior only from time u onward, and
- the semantics $[\![\alpha]\!]_u$ contains indeed *all* system behaviors that exhibit the interaction sequence specified by α, from time u on.

The following two propositions make these observations more precise.

Proposition 2 *For all $\alpha \in \langle MSC \rangle$, $t, u \in \mathbb{N}_\infty$, and $\varphi \in (\tilde{C} \times S)^\infty$ the following implication holds:*

$$(\varphi, t) \in [\![\alpha]\!]_u \Rightarrow t \geq u$$

□

Proof By induction on the structure of α. ■

Proposition 3 $[\![\alpha]\!]_u$ *is the equivalence class of all system behaviors that "comply to" α from time u on, i.e. for all $\alpha \in \langle MSC \rangle$, $t, u \in \mathbb{N}_\infty$, and $\varphi \in (\tilde{C} \times S)^\infty$ the following equivalence holds:*

$$(\varphi, t) \in [\![\alpha]\!]_u \equiv \langle \forall \psi : \psi \in (\tilde{C} \times S)^\infty \ \wedge \ \psi|_{[u,t]} = \varphi|_{[u,t]} : (\psi, t) \in [\![\alpha]\!]_u \rangle$$

□

Proof The direction "⇐" is trivial. The direction "⇒" follows by induction on the structure of α. ■

The consequence of Proposition 3's validity is that our MSC semantics is extremely loose. The elements of $[\![\alpha]\!]_u$ all represent at least a behavior depicted explicitly in MSC α, but do *not* exclude arbitrary other behavior. This looseness is helpful in the definition of refinement notions (cf. Chapter 5), as well as in relating MSCs with other forms of system specifications. In Chapter 6 we establish several such relationships; one of them (called "exact" MSC interpretation) removes all looseness, and fixes the MSC semantics to what is explicitly specified in the MSC.

4.5.3. The Relationship between Sequential Composition and Interleaving

In Section 4.4 we have noted the difference between our definition of sequential composition and the weak sequencing operator used by the authors of [IT98] for what they call the "vertical composition" of MSCs. Here, we make this statement more precise by studying the relationship between sequential composition and interleaving in our semantics.

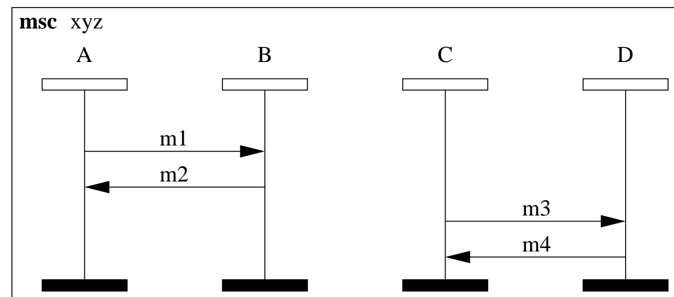


Figure 4.10.: Composition of MSCs with disjoint instance sets

The weak sequencing of two MSCs can result in their interleaving, if the two MSCs do not share instances. As an example, consider MSC *xyz* of Figure 4.10; we can think of this MSC as being the vertical composition of the MSC that contains only the interactions of instances *A* and *B*, and the MSC that contains only the interactions of instances *C* and *D*. Despite the vertical positioning of the two message "clusters" $\{m1, m2\}$ and $\{m3, m4\}$, the weak-sequencing used by the authors of [IT98] yields the interleaving of these two message clusters as the semantics of MSC *xyz*.

Another way to interpret *xyz* is that it is the horizontal composition of the two sub-MSCs we have identified before. In this case, too, the semantics of [IT98] assigns the interleaving of the individual interaction sequences as the meaning of the composite MSC.

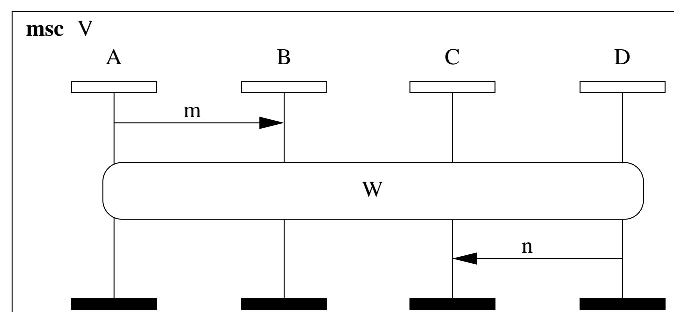


Figure 4.11.: The ordering between m and n depends on the content of MSC W

Defining vertical composition in this way can result in rather counter-intuitive MSC specifications. As an example, consider the MSC in Figure 4.11. Under the weak sequencing

of [IT96, IT98] it depends on the content of MSC W whether or not there is an order between messages m and n. If we use MSC W from Figure 4.12 to resolve the reference, then it depends on the alternative selected during execution, whether or not n can precede or must succeed m. Such specifications are difficult to communicate in practice because their semantics "contradicts" their graphical representation.

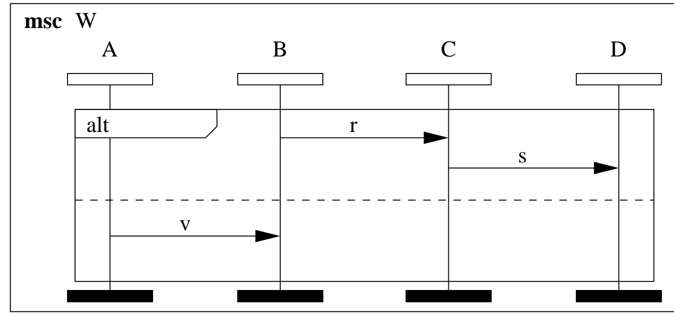


Figure 4.12.: Depending on the selected alternative n can or cannot precede m in the MSC from Figure 4.11

Therefore, we follow a different approach in our semantics definition. We require explicit use of the adequate composition operators to yield the desired interpretation. The only cases where sequential composition and interleaving coincide in our semantics are those, where at least one of the operands is **empty**. The following proposition makes this statement more precise.

Proposition 4 *For all $\alpha, \beta \in \langle MSC \rangle$ the following equivalence holds:*

$$(\alpha \sim \beta \equiv_u \alpha ; \beta) \quad \equiv \quad (\alpha \equiv_u \textbf{empty}) \vee (\beta \equiv_u \textbf{empty}) \qquad \square$$

PROOF See Appendix B.1.4. ■

This result shows the difference between our sequential composition and the weak sequencing of [IT98]. Our composition operator is "strong" in the sense that all interactions contained in its first operand precede any interactions of the second operand.

This difference in interpretation has a methodical implication. We require that all messages within an MSC be vertically ordered. Lack of ordering be only established through the interleaving operator (see also the discussion in Section 4.5.5).

Even if we follow this convention, MSCs like the one in Figure 4.10 are difficult to comprehend from a practical point of view. There is no visible communication between instances B and C that could establish an ordering between messages $m2$ and $m3$. This also suggests to avoid MSCs like xyz, and to make ordering and the lack of it explicit through messages, and application of the interleaving operator, respectively.

4.5.4. MSCs versus Temporal Logic

Given the language of MSC terms and its semantics a natural question arises: what kinds of system properties can we specify using the language?

To get a first impression on the expressiveness of our MSC dialect, we relate it with propositional linear temporal logic (PLTL, cf. [Eme90]). We do so on a rather informal level to convey the basic idea; in Chapter 6 we give a precise characterization of the properties we can express by means of MSC terms.

[Eme90] gives an informal characterization of PLTL formulae over a set of atomic propositions $\{p, q, \dots\}$, based on infinite sequences of states as a system's execution model, as summarized in Table 4.4.

Formula	Read as	Informal Meaning
$\mathbf{F}\ p$	"sometime p"	there is a state in the execution of the system where p holds
$\mathbf{G}\ p$	"always p"	p holds in all states of the system's execution
$\mathbf{X}\ p$	"nexttime p"	p holds in the next system state
$p\ \mathbf{U}\ q$	"p until q"	q holds eventually, and until that happens, p holds

Table 4.4.: PLTL formulae and their intuitive interpretation

As a more elaborate example we mention that, under this interpretation, the formula $\mathbf{GF}\ p$ characterizes system executions in which p holds infinitely often. We can, on the basis of a given system model, view each PLTL formula as the equivalence class of system executions satisfying the formula.

In our semantic framework we use infinite sequences of channel and state valuations as the model of a single system execution. In Section 4.5.2 we have identified $[\![\alpha]\!]_u$ as the equivalence class of system executions complying to α from time u on. Based on this intuition we identify MSC terms that mimic the PLTL operators **F** and **GF** as follows:

MSC Term	Read as	Informal Meaning
α	"sometime α"	from the next time point onward the system displays the behavior as specified by α
any $\mapsto \alpha$	"infinitely often α"	the system displays a behavior as specified by α infinitely often

Table 4.5.: Rephrasing of temporal logic formulae in terms of MSCs

To see the correspondence between **any** $\mapsto \alpha$ and the PLTL-operator **GF**, we simply have to write out the term's semantics definition from Section 4.4. We derive:

$$(\varphi, t) \in [\![\mathbf{any} \mapsto \alpha]\!]_u$$

$\equiv$ (* definition of . $\mapsto$. *)

$$\langle \forall t', t'' : \infty > t'' \geq t' \geq u : (\varphi, t'') \in [\![\mathbf{any}]\!]_{t'} \Rightarrow \langle \exists t''' : \infty > t''' > t'' : (\varphi, t) \in [\![\alpha]\!]_{t'''} \rangle \rangle$$

$\equiv$ (* definition of $[\![\mathbf{any}]\!]$, predicate calculus *)

$$\langle \forall t' : \infty > t' \geq u : \langle \exists t''' : \infty > t''' > t' : (\varphi, t) \in [\![\alpha]\!]_{t'''} \rangle \rangle$$

This shows that if $(\varphi, t) \in [\![\mathbf{any} \mapsto \alpha]\!]_u$ holds, then the interaction sequence specified by α occurs infinitely often in φ from time u on.

The correspondence between the PLTL-operator **F** and an MSC α as such seems less intuitive at first sight. Yet, the semantics of α includes all behaviors with an arbitrary finite delay before the behavior explicitly represented by α starts. Thus, if α is not a guarded MSC[3], we have $[\![\alpha]\!]_u = [\![\mathbf{fany} ; \alpha]\!]_u$; here we use the new MSC term **fany** (defined by $[\![\mathbf{fany}]\!]_u \stackrel{\mathsf{def}}{=} \{(\varphi, t) \in [\![\mathbf{any}]\!]_u : t \in \mathbb{N}\}$) to express arbitrary finite behavior.

The intuition behind the PLTL-formula $\mathbf{G}p$ is that every state fulfills p during an infinite system execution. In other words, there is no state in which p does *not* hold. Our MSC semantics explicitly *includes* arbitrary other behavior besides the one depicted in the MSC under consideration. Similarly, an "until" formula $p \; \mathbf{U} \; q$ excludes occurrence of states in which p does not hold until q eventually holds; we cannot express this exclusion, in general, with the MSC semantics as such. To mimic the PLTL-operators **G** and **U**, we need a more restrictive MSC interpretation. We discuss this "exact" MSC interpretation in more detail in Chapter 6; it excludes other behavior than the explicitly depicted one. By means of this interpretation we are able to use MSC terms like $\alpha\uparrow_{<\infty>}$ and $\alpha\uparrow_{<*>} ; \beta$ to rewrite the PLTL-operators **G** and **U**, respectively.

Temporal logics such as UNITY [CM88] and TLA [Lam94] provide a "leadsto" operator instead of PLTL's "until". For instance, $p \mapsto q$ in UNITY means that whenever p holds during system execution then q will also hold within finite time. The trigger composition $\alpha \mapsto \beta$ directly mimics this interpretation.

The correspondence between temporal logic formulae and MSC terms suggests to construct MSC specifications that follow the specification patterns contained in Table 4.5 and the accompanying text above. It also gives a first hint at a way of integrating MSCs into the validation task: given a system specification *Sys* and a property *Prop* whose truth in *Sys* we want to validate, we can construct an MSC α that *has* property *Prop* according to the schemata above, and check whether α's behaviors are abstractions of the behaviors of *Sys*.

4.5.5. Adequacy of the Syntax and its Semantics

In Section 4.4 we have defined the semantics for our MSC dialect based on the textual syntax of MSC terms. This raises the question whether the textual syntax is "adequate", i.e. whether it has the same expressiveness as the graphical syntax. The second question related to the adequacy of our MSC dialect is a semantic one: does the model we have

[3]Guarded MSCs inject safety-properties into MSC specifications, cf. Section 6.3.

chosen indeed capture asynchronous message exchange? We tackle both of these questions, in turn, in this section.

Relating the Graphical and the Textual Representation

The semantics of our MSC dialect bases on the textual MSC syntax from Section 4.3, not directly on the graphical notation. In Section 4.4 we have motivated the relationship between the graphical and the textual syntax by means of generic examples (cf. Figures 4.3 through 4.9).

We consider the proximity of the constructs provided by the textual syntax and those of the graphical notation convincing enough to claim that for every MSC in graphical notation there is a corresponding MSC term and vice versa. The remaining issue is how to find, say, the MSC term corresponding to an MSC in graphical form. The discussion of the relationship between sequential composition and interleaving in Section 4.5.3 shows the existence of MSCs whose graphical representation can be misleading. Recall the MSC of Figure 4.10, where messages *m3* and *m4* are located below messages *m1* and *m2*. Yet, the "horizontal" and "vertical" composition operators of MSC-96 implicitly yield the interleaving of the two independent message sequences as the semantics of the MSC.

To simplify the derivation of an MSC term from a given MSC in graphical form, we require the graphical representation to contain all forms of composition for "sub-MSCs" *explicitly*, by means of adequate inline expressions. As an example, we require a rewriting of the MSC from Figure 4.10 to yield the form displayed in Figure 4.13.

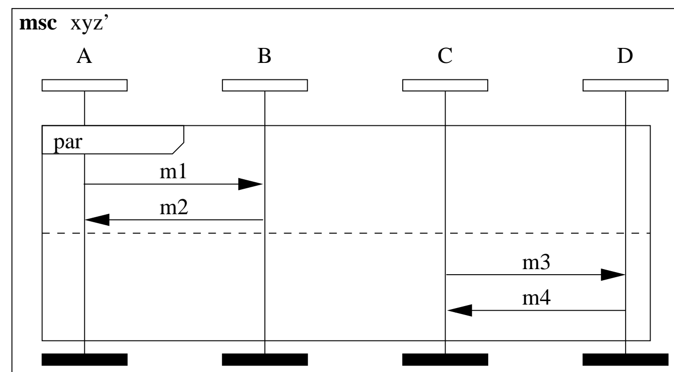


Figure 4.13.: Unambiguous version of Figure 4.10

The derivation of the MSC term corresponding to the MSC in graphical form is then just a matter of selecting the appropriate composition operators in ⟨MSC⟩, according to Figures 4.3 through 4.9.

With this technical convention we have established a one-to-one correspondence between the graphical and the textual syntax. Therefore, in the remainder of this thesis, we use

the phrases "semantics of the MSC" and "semantics of the MSC term corresponding to the MSC" interchangeably.

Capturing Asynchronous Message Exchange

MSC-96 models asynchronous message exchange by introducing two distinct events for every message m: the send and the receive event; furthermore, MSC-96 requires a message's send event to precede the corresponding receive event. No other constraints exist on the duration between sending and receipt of a message. In particular, there is no notion of a global clock (cf. [IT96, IT98]).

In contrast, the system model we have introduced in Section 4.2 does not associate two events to a single message; instead, we associate a specific time point with respect to a global clock at which the message occurs on its corresponding channel.

This raises the question whether the system model we use is adequate for modeling asynchronous message exchange in the sense of MSC-96. Indeed, associating a single time point with the occurrence of a message, as it happens in our semantics definition, seems to suggest that we have modeled a synchronous communication mechanism.

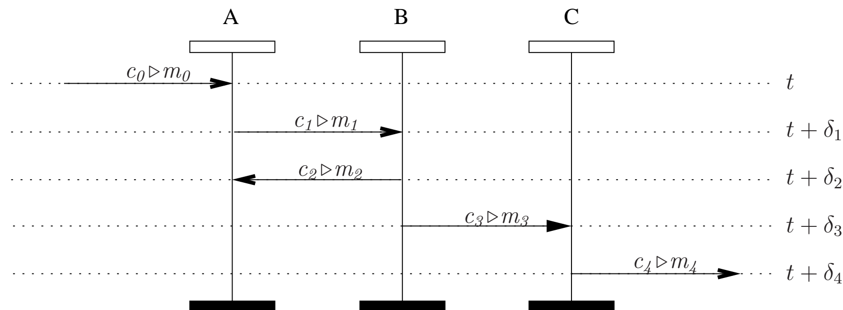


Figure 4.14.: Rationale for the adequacy of the MSC semantics

To answer this question we observe that the order in which the messages arrive at or emanate from a single axis induces an order on the times at which these messages may occur on their respective channels. As an example, consider message $c_2 \triangleright m_2$ in the MSC fragment of Figure 4.14. To the right of this figure we have indicated the time points at which the messages occur. $c_2 \triangleright m_2$ can occur only after at least δ_2 time units, calculated from the occurrence of message $c_0 \triangleright m_0$ on, have elapsed (see the definition of sequential composition in Section 4.4). The total ordering of time points for the occurrences of messages in Figure 4.14 is $1 \leq \delta_1 < \delta_2 < \delta_3 < \delta_4$. Remember that in our system model the "reaction" of a component to an incoming message must have a delay of at least one time unit. The idea is that the channels between components act as unbounded message buffers. The time point at which a message occurs on its channel indicates the moment at which the originator sent this message. If we identify the time at which a component

reacts to an input message with the actual receipt of this message, then we have in fact established a distinction between the sending of a message and its receipt in the sense of MSC-96. The required delay of at least one time unit between the occurrence of a message on its channel and the receiving component's reaction to it models the precedence of send over receive "event".

However, the system model we have chosen is not bound to one specific style of communication. From an abstract point of view the channel valuations in an MSC α's semantics reveal the message orderings specified by α, whether the underlying communication mechanism is asynchronous or not. The distinction between these two mechanisms becomes of importance only when we discuss the transition from MSC specifications, which describe "global" behavior, to the specification of individual component behavior (see Chapter 7).

4.6. HMSCs

As we have mentioned in Section 2.2.2, the constructs provided by HMSCs – namely references, alternatives, finite and infinite repetition, and interleaving – correspond directly to similar constructs in plain MSCs. Here, we exploit this similarity to base the semantics definition for HMSCs on the one of plain MSCs. Our major task is, therefore, to transform the topological ordering of MSC references within an HMSC into a linear MSC term. The standard semantics in [IT98] achieves this transformation by introducing recursive process-algebraic equations. The approach we take here is purely syntactic and constructive; thereby we avoid changing the semantics definition of Section 4.4.

We assume given an HMSC whose reference nodes refer to plain MSCs only. If we encounter an HMSC violating this rule, we perform the mapping for the HMSCs it references first. This simplification is possible, because references in both plain MSCs and HMSCs must be acyclic (cf. [IT96, IT98], as well as Section 4.4). We require the HMSC to have at least one reference node. Furthermore, we assume that the HMSCs have no parallel nodes. We can replace any parallel node by a reference to a plain MSC that hosts the translated parallel HMSCs in the operands of a **par**-inline expression. Hence, the only nodes contributing message exchange to the semantics of an HMSC are its reference nodes. The first step towards the definition of the mapping is to remove all connection and condition nodes from an HMSC.

The remaining graph already resembles a finite state automaton known from formal language theory; the only major difference is that HMSC graphs have their labels on the nodes, not on the edges as usual.

The idea behind our mapping is to use a modification of the well-known transformation from finite-state automata to regular expressions. However, standard regular expressions and their corresponding finite state automata do not capture the infinite repetition of interaction sequences that HMSCs can represent. Therefore, we adapt the transformation

from automata to regular expressions presented in [HU90] to obtain a term in ⟨MSC⟩ as the textual representation for the HMSC; the semantics of this term is, by definition, the one of the HMSC.

The mapping itself consists of performing the following two steps in order:

1. transform the HMSC graph into a finite state automaton;
2. transform the automaton into an MSC term.

In the following paragraphs we explain each of the mentioned transformations in turn; along the way we give examples for the individual transformation steps.

Transforming an HMSC Graph into an Automaton To keep the first phase of our mapping from HMSCs to plain MSCs as close as possible to the well-known transformation from finite state automata to regular expressions (cf. [HU90]), we turn the given HMSC graph into an "equivalent" finite state automaton (FSA). The difference between the HMSC graph and its corresponding FSA is that the latter has the reference labels on its edges, whereas the former has them on the nodes. Furthermore, an FSA represents the connections between HMSC reference and end nodes by means of terminal nodes.

More precisely, we use a sixtuple

$$\mathcal{A} = (\Sigma, L, \delta, s_0, s_t, F)$$

as the FSA representation of HMSC graphs. Σ and L denote the automaton's set of states and the set of transition labels, respectively. Every label is an element from ⟨MSC⟩. $\delta : \Sigma \times L \times \Sigma$ denotes the transition relation; we have $(x, y, z) \in \delta$ if and only if there exists an edge labeled with y from state x to state z in the FSA. $s_0 \in \Sigma$ denotes the automaton's initial state; $s_t \in \Sigma$ is a special "trap" state that becomes relevant later in the translation process. $F \subseteq \Sigma$ denotes the set of terminal states.

As the running example for illustrating the transformations in this section we use the HMSC of Figure 4.15 (a).

To obtain the FSA $\mathcal{A} = (\Sigma, L, \delta, s_0, s_t, F)$ with $\Sigma = \{s_0, s_1, ..., s_n, s_t\}$ from an HMSC A with n reference nodes, we perform the following steps:

- with every reference node we associate exactly one element of the set $\{s_1, ..., s_n\}$, such that no two reference nodes map to the same element from this set. Hence, every automaton state in the set $\{s_1, ..., s_n\}$ represents precisely one of the reference nodes of the HMSC graph. Figure 4.15 (b) shows one possible result of this transformation on the HMSC;

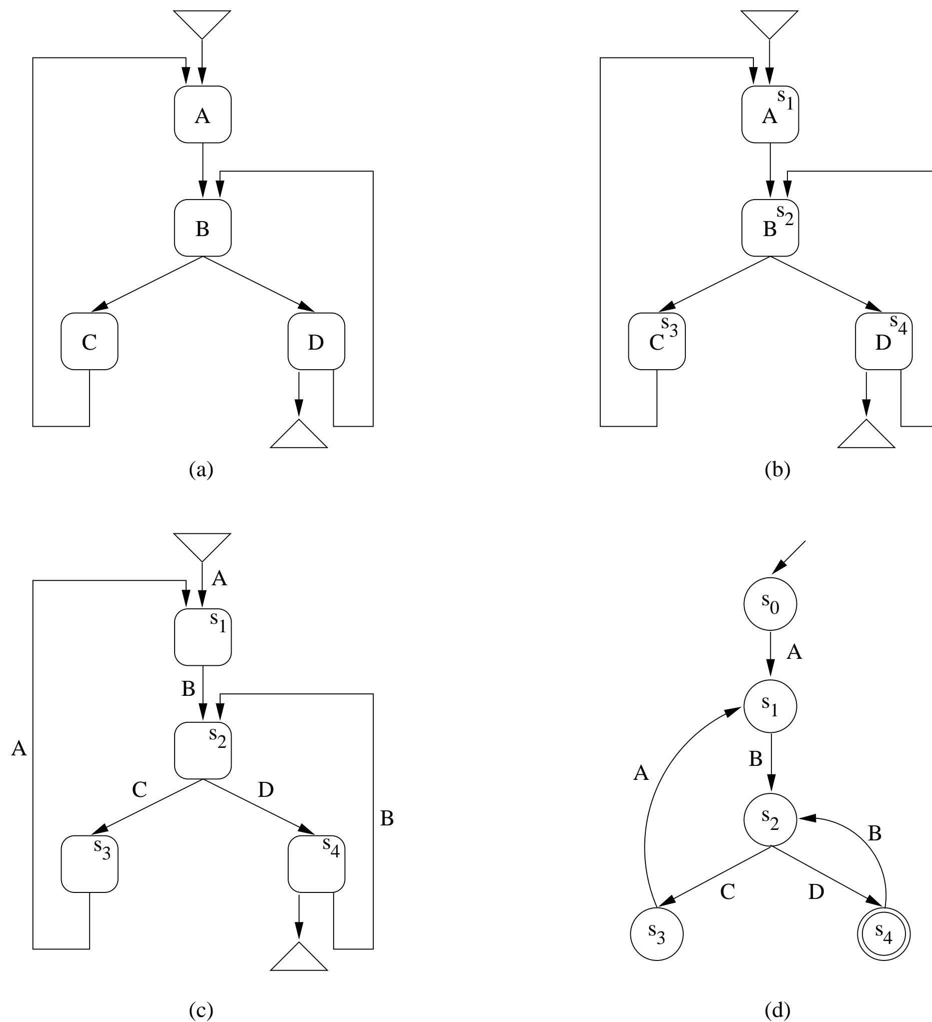


Figure 4.15.: Transformation from an HMSC to its corresponding FSA

- we modify the HMSC graph by moving the label of every reference node to all its incoming edges. Figure 4.15 (c) shows the result of this transformation on the example HMSC;
- the automaton's initial state s_0 represents the HMSC's start node;
- if a reference node connects to the graph's end node, the state associated with this reference node becomes a member of F,
- we obtain $\mathcal{A}$'s transition relation by considering every edge in the modified HMSC graph. For all pairs $(i, j) \in \{0, ..., n\} \times \{0, ..., n\}$, and for all $y \in L$ we add the triple (s_i, y, s_j) to δ if and only if there exists an edge labeled with y from the reference (or start) node associated with s_i to the reference node associated with s_j.

For the example HMSC from 4.15 (a) successive application of these steps yields the graphic representation of the corresponding FSA in Figure 4.15 (d).

The formal automaton representation of an HMSC graph according to the steps introduced here is the basis for the transformation from HMSCs to MSC terms.

Transforming an Automaton into an MSC Term The next step of our translation from HMSCs to plain MSCs is to obtain a term in ⟨MSC⟩ from the HMSC's FSA representation. We do so along the lines of well-known procedures from formal language theory for regular expressions (cf. [HU90]). The only difference between these and the procedure we describe here is that we have to decide whether a cycle in the automaton corresponds to an infinite or a possibly finite repetition in the MSC term.

Intuitively, the procedure we apply consists of a sequence of steps. Each step transforms a given automaton into a "smaller" one; the target automaton is smaller in the sense that it has either less states, or less transitions than its origin. When we eliminate a state in transiting between the source and the target automaton, we determine the terms in ⟨MSC⟩ corresponding to "short-circuiting" the incoming, loop, and outgoing transitions (in all possible combinations) with respect to this particular state. We then label the short-circuted transitions with the corresponding MSC term. Thus, we build the MSC term represented by the original automaton in an iterative fashion by eliminating states and short-circuiting the eliminated states' incoming, loop, and outgoing transitions. The procedure terminates, once there remain exactly two states and one transition, whose label is the overall MSC term, between these two states.

Before we describe the state elimination formally, we observe that, when eliminating a state from an automaton, we assume the situation depicted in Figure 4.16. The state is either final or non-final. It has precisely one loop transition whose label we denote with l. l may equal **empty**. Furthermore, we assume that, besides the self loop, the state has $n \in \mathbb{N}$ incoming and $m \in \mathbb{N}$ outgoing transitions, whose labels are i_1 through i_n, and o_1 through o_m, respectively. If a state has multiple self-loops labeled, say, by l_1 through l_k, $k \in \mathbb{N}$, we can replace them by a single one whose label is the MSC term $l_1 \mid \ldots \mid l_k$. More generally we replace multiple transitions between any two automaton states by a single one, whose label represents the alternatives in the form of an MSC term before we perform any state elimination step.

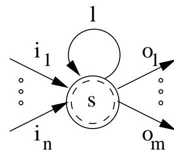


Figure 4.16.: General state with n incoming and m outgoing transitions besides the self-loop

The crucial point in state elimination is, as we have mentioned above, to determine whether a self-loop represents an infinite or a possibly finite repetition. Intuitively, final states of the automaton, as well as states with outgoing transitions other than the self-loop, yield possibly finite repetition, whereas all other states yield infinite repetition of the label on their self-loop.

Formally, we define each step of the state elimination procedure as a transformation between the two automata $\mathcal{A} = (\Sigma, L, \delta, s_0, s_t, F)$ and $\mathcal{A}' = (\Sigma', L', \delta', s_0, s_t, F')$. The result of one step, i.e. $\mathcal{A}'$, becomes the starting point for the next state elimination step. For any given state $s \in \Sigma$ we denote by $P.s = \{(s_p, i, s) \in \delta : s_p \in \Sigma \setminus \{s\}\}$ the set of its incoming transitions, and by $Q.s = \{(s, o, s_q) \in \delta : s_q \in \Sigma \setminus \{s\}\}$ the set of its outgoing transitions. Neither $P.s$, nor $Q.s$ contains a self-loop of s. We denote the self-loop of s by $(s, l, s) \in \delta$.

Let $s \in \Sigma \setminus \{s_0, s_t\}$ be the state we want to eliminate. We never eliminate the initial or the "trap" state. The labels of incoming transitions of the trap state s_t indicate possible suffixes of the MSC term we seek. The procedure terminates if $\Sigma = \{s_0, s_t\}$. The label of the only remaining transition between s_0 and s_t is the resulting MSC term. Below we will show that the elimination will, indeed, terminate in the mentioned configuration.

In the following, we assume that $\Sigma \setminus \{s_0, s_t\} \neq \emptyset$. We set $\Sigma' = \Sigma \setminus \{s\}$, and distinguish four cases, depending on whether s is a final state or not, and whether it has outgoing transitions or not.

1.) If $s \notin F \wedge Q.s = \emptyset$, i.e., if s neither is a final state nor has outgoing transitions other than its self-loop labeled l, we have found a state whose self-loop describes an infinite repetition of the corresponding label. We set $F' = F$ and eliminate s by redirecting all incoming transitions to the trap state, and by appending the MSC term representing the infinite repetition of the self-loop's label to each such transition. Formally, we define

$$\begin{aligned} \delta' &= (\delta \setminus \{(x, y, z) : x = s \vee z = s\}) \\ &\cup \{(s_p, i \,;\, l{\uparrow}_{<\infty>}, s_t) : (s_p, i, s) \in P.s \;\wedge\; (s, l, s) \in \delta\} \end{aligned}$$

2.) If $s \notin F \wedge Q.s \neq \emptyset$, i.e., if s is not a final state and has outgoing transitions other than its self-loop labeled l, we have found a state whose self-loop describes a possibly finite repetition of the corresponding label. Intuitively, we could perform a finite number of self-loops after having entered the state through any incoming transition, and then leave the state along any of its other outgoing transitions. Alternatively, we could perform the self-loop infinitely often. We eliminate this state by establishing all possible short-circuitings between any one incoming and any one outgoing transition. The label of such a short-circuited transition consists of the sequencing of the label of the corresponding incoming transition, the finite or infinite repetition of the self-loop's label, and the label of

the corresponding outgoing transition. Formally, we define $F' = F$ and

$$\begin{aligned}\delta' &= (\delta \setminus \{(x,y,z) : x = s \vee z = s\}) \\ &\cup \{(s_p, i \,;\, l{\uparrow}_{\lhd 0,\infty\rhd} \,;\, o, s_q) : (s_p, i, s) \in P.s \wedge (s, o, s_q) \in Q.s \wedge (s, l, s) \in \delta\}\end{aligned}$$

3.) and 4.) Now we assume that $s \in F$ holds. In this case we set $F' = F \setminus \{s\}$. s is a state whose self-loop describes a possibly finite repetition of the corresponding label. The repetition may be finite because we can stop interpretation of the automaton at any of its final states. The first step towards elimination of this state is to redirect all incoming transitions to the trap state, and by appending the MSC term that represents the possibly finite repetition of the self-loop's label to each such transition. Formally, we define

$$\begin{aligned}\delta'' &= (\delta \setminus \{(x,y,z) : x = s \vee z = s\}) \\ &\cup \{(s_p, i \,;\, l{\uparrow}_{\lhd 0,\infty\rhd}, s_t) : (s_p, i, s) \in P.s \wedge (s, l, s) \in \delta\}\end{aligned}$$

If $Q.s = \emptyset$ we are done and set $\delta' = \delta''$, which concludes case **3.)**. Otherwise, i.e., if s has outgoing transitions other than its self-loop labeled l (case **4.)**), we add all possible short-circuitings between any one incoming and any one outgoing transition to the transition relation of $\mathcal{A}'$. The label of such a short-circuited transition consists of the sequencing of the label of the corresponding incoming transition, the finite or infinite repetition of the self-loop's label, and the label of the corresponding outgoing transition. Formally, we define

$$\begin{aligned}\delta' &= \delta'' \\ &\cup \{(s_p, i \,;\, l{\uparrow}_{\lhd 0,\infty\rhd} \,;\, o, s_q) : (s_p, i, s) \in P.s \wedge (s, o, s_q) \in Q.s \wedge (s, l, s) \in \delta\}\end{aligned}$$

To obtain L' in any of the four cases mentioned above we simply union the labels of the freshly added transitions with the existing ones from L.

Above we claimed that successive application of state elimination terminates and results in an automaton with precisely two states (s_0 and s_t), and one transition between these states. In every elimination step precisely one state vanishes; this ensures termination of the elimination process. Furthermore, no elimination step removes incoming transitions from s_t[4]. In cases 1.) and 3.) we add incoming transitions to s_t. In case 2.)(and similarly in case 4.)) we short-circuit the incoming and outgoing transitions of the state under consideration. This leaves the number of incoming transitions of s_t intact. It cannot remain zero, however, because successive application of rules 2.) and 4.), with no incoming transitions at s_t, can happen at most until there exist precisely four states in the automaton: s_0, s, t, and s_t. For case 2.) or 4.) to apply, s must have an outgoing transition to t. After application of either rule 2.) or 4.) s vanishes, and t can have at most a self-loop. In particular, it cannot have other outgoing transitions. Hence, neither rule 2.), nor rule 4.) applies when

[4] up to replacing multiple incoming transitions by a single one that represents all alternatives

we eliminate t; this leads to addition of an incoming transition to s_t by either rule 1.) or 3.). This shows that in the final configuration there is precisely one transition between s_0 and s_t.

As an example, consider again the FSA from Figure 4.15 (d). Figures 4.17 (a) through (d) show the result of eliminating the states s_1, s_3, s_2, and s_4, in this order. For better readability we write X instead of $\rightarrow X$ in this example. The resulting MSC term is $A \,;\, B \,;\, (C \,;\, A \,;\, B) \uparrow_{\lhd 0,\infty\rhd} \,;\, D \,;\, (B \,;\, (C \,;\, A \,;\, B) \uparrow_{\lhd 0,\infty\rhd} \,;\, D) \uparrow_{\lhd 0,\infty\rhd}$. By application of the simplification rules for loops (cf. Section 4.4), we obtain the equivalent term $A \,;\, (B \,;\, (C \,;\, A \,;\, B)\uparrow_{\lhd 0,\infty\rhd} \,;\, D)\uparrow_{\lhd 1,\infty\rhd}$

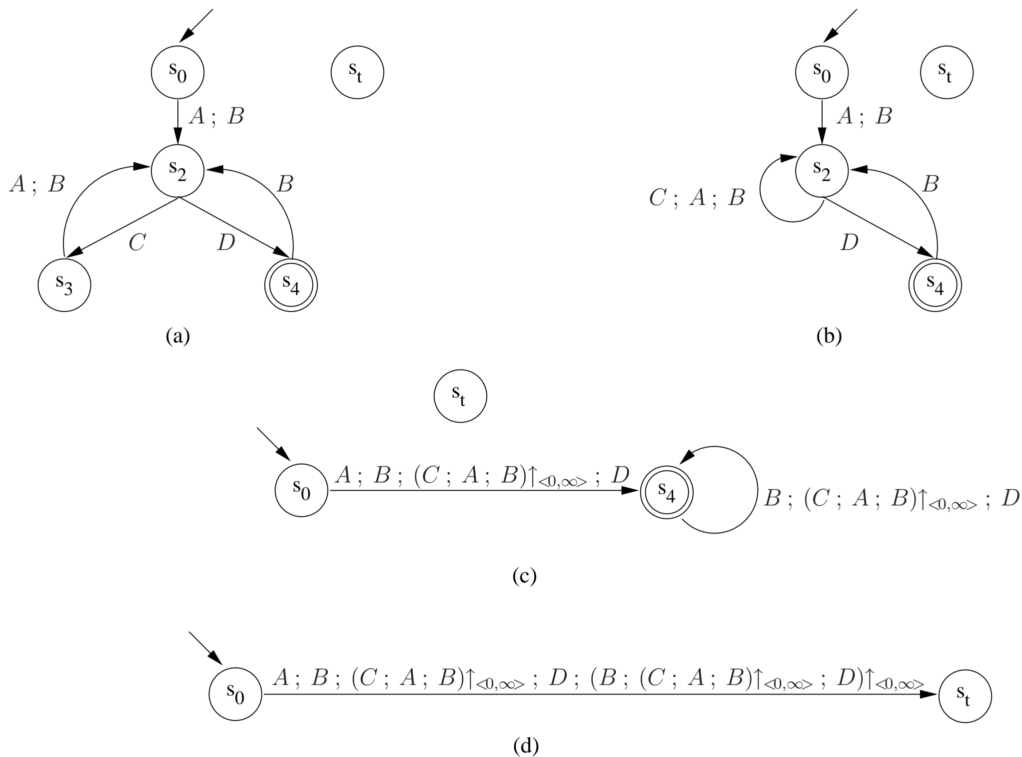


Figure 4.17.: Example application of state elimination

Extensions for Preemption In Sections 4.3 and 4.4 we have introduced the “core” syntax and semantics for preemption. We allow preemption specifications in HMSCs as well. Figures 4.18 (a) and (b) show the corresponding graphical notation: we use a dashed arrow whose label is the preempting message, directed from the MSC reference being preempted to the MSC reference representing the behavior in case the preemptive message occurs. The integration of this notation into the transformation procedure above is straightforward. We have to label automaton transitions with the information whether they correspond to a regular or a preemption arrow. For the elimination of a state with exiting or entering preemptive arrows we have to consider the situations depicted in Figures 4.18 (a) and (b).

For the preemption in Figure 4.18 (a) we use the MSC term $(\to A) \overset{ch \triangleright m}{\to} (\to B)$ (instead of sequential composition), and for the preemptive loop in Figure 4.18 (b) we use the term $(\to A)\Uparrow_{ch \triangleright m}$ (instead of the loop construct). For regular transitions the transformation remains unchanged.

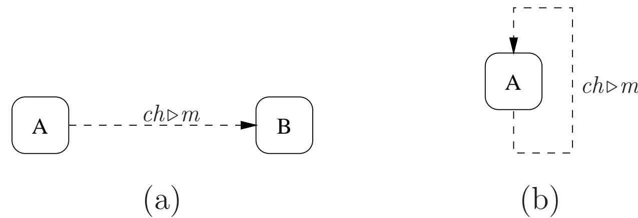


Figure 4.18.: Notation for preemption and preemptive loop in HMSCs

HMSC semantics The elimination procedure we have introduced above leaves freedom in the order in which the nodes of the FSA get eliminated. Each node ordering may yield a syntactically different MSC term. However, these different MSC terms are semantically equivalent. The proof of this, which we do not carry out here, proceeds along the exact same lines as the proof in [HU90] that asserts the equivalence of finite automata and corresponding regular expressions.

In the remainder of this thesis we identify HMSCs with the MSC terms produced by the transformation above, without mentioning the transformation explicitly.

4.7. Example: the Abracadabra-Protocol

In the preceding sections we have introduced and discussed the syntax and semantics of our MSC dialect. Because of the direct correspondence between MSC-96 and our MSC dialect almost all of the examples we have given in Section 2.2 are also examples for our syntax and semantics.

Here, we illustrate the new concepts, such as unbounded repetition, join and trigger composition, and preemption, provided by our approach. To that end, we use MSCs to specify a slightly modified version of the Abracadabra-protocol (cf. [Bro87, BK98]).

We start with an informal description of this protocol in Section 4.7.1. From this we extract the major "use cases" of the protocol, and integrate them into an HMSC that captures all possible protocol executions (cf. Section 4.7.2). Sections 4.7.3 and 4.7.4 contain the plain MSCs for the use cases we have identified. In Section 4.7.5 we show how to employ join and trigger composition to add a progress (cf. [CM88]) or liveness (cf. [Lam94]) property to the protocol specification. We modify the protocol slightly in Section 4.7.6 to demonstrate the use of preemption.

4.7.1. Informal "Requirements Specification"

We consider a system that consists of two components X and Y, and the channels xy (directed from X to Y) and yx (directed from Y to X). Figure 4.19 shows this structure in the form of an SSD.

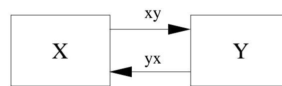


Figure 4.19.: SSD for the ABRACADABRA-protocol

The symmetric ABRACADABRA-protocol (cf. [BK98]) describes a scheme that allows any of the two components to

1. establish a connection to the other component,
2. send data messages once a connection exists,
3. tear down an existing connection it has initiated.

If both components try to establish a connection simultaneously, the system is in conflict. Both components tear down their "attempted" connections to resolve the conflict.

We describe the symmetric protocol in more detail from X's viewpoint. To establish a connection, X sends message *sreq* ("sending requested") to Y. If Y returns message *sack* the connection is established. In this case, X sends a finite number of data messages to Y, such that Y replies each data message individually by sending message *dack* to X. Upon receipt of a *dack* message, X can either send another data message, or close the connection. To close the connection, X sends message *ereq* ("end requested") to Y, and receives message *eack* ("end acknowledged") as a reply.

If, after sending message *sreq* to Y, X receives message *sreq* instead of *sack* from Y, both components resolve this conflict by sending message *ereq*, and – upon receipt of their partner component's *ereq* message – message *eack*.

After a successful transmission, as well as after conflict resolution, the protocol starts over again (ad infinitum).

4.7.2. "Roadmap" for the Major Use Cases

As a first step towards an MSC specification of the ABRACADABRA-protocol we identify four major use cases of the system under specification from the informal description above:

Use Case	Description
(SX)	X initiates a successful transmission,
(SY)	Y initiates a successful transmission,
(C)	conflict; X and Y try to establish a connection simultaneously,
(CR)	X and Y perform conflict resolution.

In Sections 4.7.3 and 4.7.4 we will develop plain MSCs for each of these use cases. Here, we use the informal protocol description to integrate the use cases into a roadmap for an overall system specification.

We can understand the ABRACADABRA-protocol as consisting of an infinite sequence of steps, where during any particular step the system evolves as either of the use cases *SX*, *SY*, or *C* (followed by *CR*) describes. The HMSC *A* from Figure 4.20 captures this understanding.

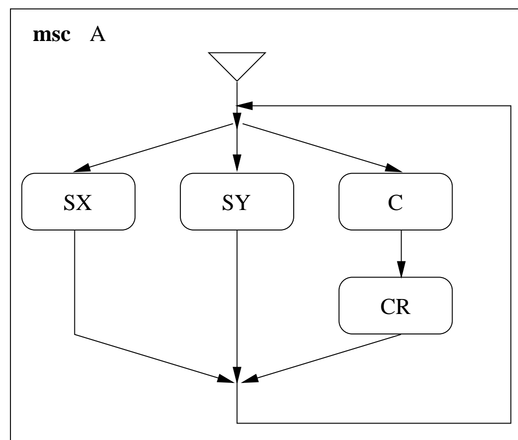


Figure 4.20.: HMSC for the ABRACADABRA-protocol

An MSC definition corresponding to the graphical representation in Figure 4.20 is the following:

$$\textbf{msc}\ A = (\rightarrow SX \mid\ \rightarrow SY \mid (\rightarrow C\ ;\ \rightarrow CR))\uparrow_{<\infty>}$$

MSC *A* describes all possible paths through the use cases of the ABRACADABRA-protocol we have identified above. Our remaining task is to provide MSC definitions for the references occurring in MSC *A*.

4.7.3. Successful Communication

A successful transmission, initiated by X consists of

1. establishing the connection through the message sequence $xy \triangleright sreq$; $yx \triangleright sack$,
2. transmitting a finite sequence of data messages (with corresponding acknowledgments), i.e. a repetition of the form: $(xy \triangleright d \;;\; yx \triangleright dack)\uparrow_{<*>}$,
3. tearing town the connection through the message sequence $xy \triangleright ereq$; $yx \triangleright eack$.

The MSC from Figure 4.21 (a) displays this part of the protocol in graphical form; Figure 4.21 (b) contains the symmetric case where Y is the initiator.

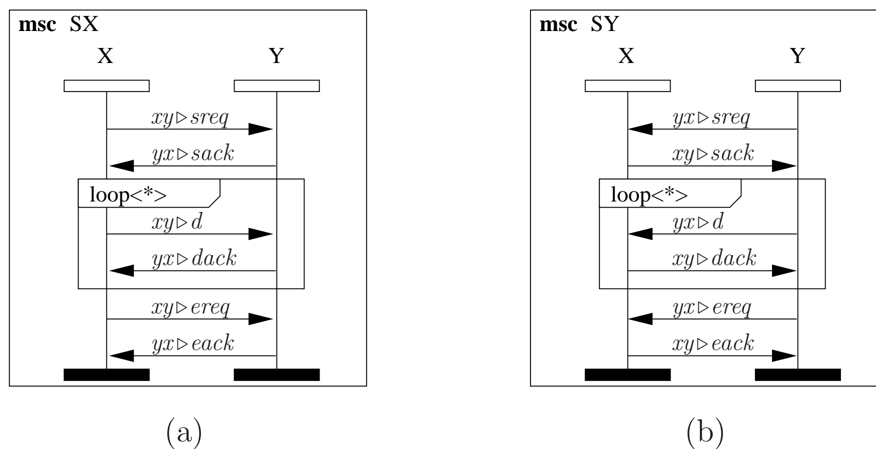


Figure 4.21.: MSCs for successful transmission

Because MSC-96 lacks a loop operator for unbounded finite repetition, we could not have formulated these two use cases adequately in MSC-96.

4.7.4. Conflict and Conflict Resolution

Conflict occurs if both X and Y try to establish a connection simultaneously. The MSC in Figure 4.22 (a) captures this case by means of causally unrelated messages, i.e. through the use of interleaving.

Conflict resolution is, according to the informal specification, a matter of mutual exchange of messages *ereq* and *eack* by X and Y. Again, there need not be a specific order between the *ereq* and *eack* messages with different origins (cf. Figure 4.22 (b)).

All MSCs together, i.e. A, SX, SY, C, and CR, capture the requirements stated informally at the beginning of this example.

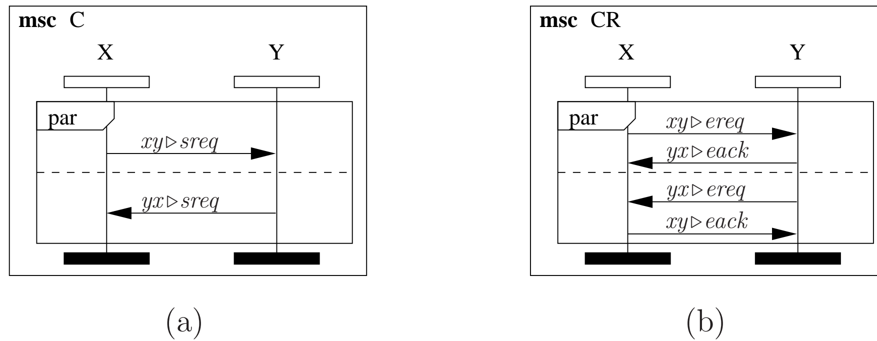


Figure 4.22.: MSCs for conflicts and their resolution

4.7.5. Adding Progress/Liveness

The informal requirements specification leaves open whether a send request by either component will, eventually, result in an established connection. The use of nondeterministic choice in the definition of MSC A allows an infinite sequence of steps consisting only of conflict and conflict resolution.

We can cast the progress/liveness property (cf. [CM88, Lam94]) that a message $xy \triangleright sreq$ must lead to subsequent data exchange, i.e. occurrence of at least one $xy \triangleright d$ message, in terms of a trigger composition of the form $xy \triangleright sreq \mapsto xy \triangleright d$. To express the validity of this property in all executions of the Abracadabra-protocol, we can join the reference to MSC A to the MSC expressing the trigger composition (cf. Figure 4.23).

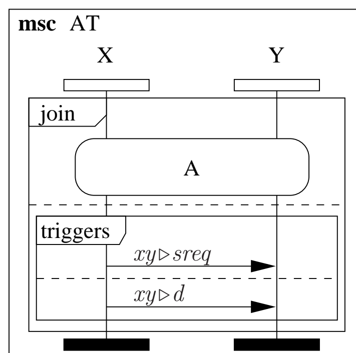


Figure 4.23.: Abracadabra with progress property

Through the join operator we "bind" the messages occurring in the trigger composition to those in A. The semantics of the joint MSC is the subset of A's semantics where every $xy \triangleright sreq$ message is followed by an $xy \triangleright d$ message eventually. Put another way, no element of the semantics of the joint MSC has only conflicts or successful transmissions initiated by Y, if X issues message $xy \triangleright sreq$ at least once.

The combination of trigger composition and join helps to *express* the desired property; it does not, however, provide an immediate strategy for implementing it. In Section 5.3 we

will – in the context of MSC refinement – investigate transformations of specifications that allow us to obtain MSCs *implementing* a given progress property.

4.7.6. Adding Preemption

To demonstrate an application of preemption we extend the informal protocol specification given above as follows: we modify the system structure from Figure 4.19 by connecting component X to the "environment" (represented by component ENV) through channel ex (cf. Figure 4.24).

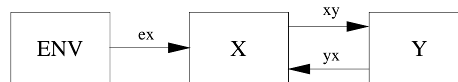
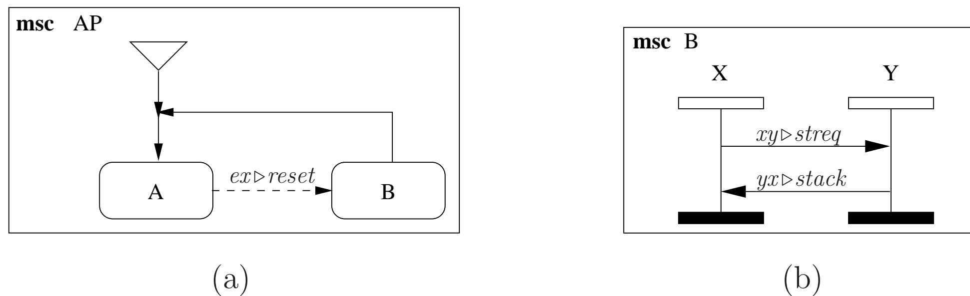


Figure 4.24.: SSD for the ABRACADABRA-protocol with preemption

By sending message *reset* along channel ex the environment can force X to stop any communication it may currently be involved in, and to restart the whole protocol afresh. Upon receipt of message *reset*, component X sends message *streq* ("stop requested") to Y; Y replies by sending message *stack* ("stop acknowledged") to X.

The HMSC AP from Figure 4.25 (a) models this behavior by means of a preemption arrow labeled with the preemptive message $ex \triangleright reset$. MSC B (cf. Figure 4.25 (b)) shows the handling of the preemption. Recall that MSC A (cf. Figure 4.20) captures the overall behavior of the ABRACADABRA-protocol (without preemption).



(a) (b)

Figure 4.25.: Preemption and its handling

The textual MSC definition corresponding to the graphical representation in Figure 4.25 (a) is **msc** $AP = ((\to A) \stackrel{ex \triangleright reset}{\to} (\to B))\uparrow_{<\infty>}$.

Without the preemption construct we would have to rewrite the MSCs A, SX, SY, C, and CR completely to accommodate the external reset request. The resulting MSCs would lose their intuitive appeal almost entirely, because the one exceptional case would dominate the whole specification.

None of the MSCs we have introduced so far *generates* the message $ex \triangleright reset$. Nevertheless, $[\![\rightarrow AP]\!]_u$ contains all behaviors that, whenever $ex \triangleright reset$ occurs, preempt the Abracadabra-protocol, perform some preemption handling (according to MSC B), and then "restart" the whole protocol.

4.8. Related Work

The semantics we have discussed in this chapter extends earlier work on EETs (cf. [SHB96, BHKS97a, BHKS97b]) and HySCs (cf. [GKS99a, GKS99b]); it was also influenced by the semantics for TSDs of [Fac95].

[Bro98, Bro99b] and [Kle98] use stream processing functions (directly or indirectly) as the basis for defining MSC semantics. Our approach employs sets of streams instead. This simplifies the treatment of repetition, preemption, and trigger composition; it also permits basing MSC refinement directly on the notion of set inclusion, as we will see in Chapter 5.

In Chapter 2 we have already given an overview of other MSC dialects and their semantics bases. The ones most closely related to the notation introduced in this chapter are MSC-96/MSC 2000 (cf. [IT96, IT98, IT99]), and LSCs (cf. [DH99]).

In Section 4.5.4 we have established the relationship between our MSC notation and formulae in PLTL and similar logics (such as UNITY and TLA) on an informal level. Obviously, there is also a similarity with the μ-calculus (for a brief introduction see [Eme90]). In the μ-calculus fixpoint formulae express eventuality and invariance properties. However, as [Eme90] points out, the expressiveness of the μ-calculus subsumes the one of tree-logics such as CTL and CTL*. To simplify the notions of equivalence and refinement we have deliberately chosen a semantic framework more closely related with linear temporal logics.

4.9. Summary

In this chapter we have defined the formal syntax and semantics of the MSC dialect we use in this thesis. We have based our treatment on a precise system model that captures communication among, and state changes of components over time; this model is adequate for the specification of all kinds of reactive systems.

To support the inductive definition of our MSC dialect's denotational semantics we have introduced an abstract textual syntax for MSC terms. Syntax and semantics capture the following notational elements: empty MSC (no interaction), arbitrary interactions, message exchange, sequential composition, guarded MSCs, alternatives (choice), interleaving (parallel composition), join, loops, references, preemption, and trigger composition. Our treatment goes beyond the approaches of MSC-96 and most of the other MSC variants

we have described in Chapter 2 in that it assigns a meaning to condition symbols, allows guiding choices and loop termination through guarding predicates, introduces the concept of preemption, and allows specification of liveness properties in an intuitive fashion through MSCs.

We have established a number of properties of the semantics: it is well-defined, distinguishes strictly between sequential composition and interleaving (in contrast to MSC-96), and, despite minor differences, models the same interaction sequences as MSC-96. We have also sketched the relationship between temporal logic formulae and certain terms in our MSC dialect. This relationship motivates the use of "specification patterns" in constructing MSCs to yield systems that have the required temporal properties.

Moreover, we have described how to extend our MSC dialect to capture High-Level MSCs (HMSCs). Together with the syntactic and semantic extensions of Appendix A (instance start and stop, timers, message parameters and parametric MSCs, actions, and gates) our MSC dialect covers a large subset of MSC-96, and – as mentioned above – even goes beyond MSC-96's expressiveness.

By means of the Abracadabra-protocol we have demonstrated the use of unbounded repetition, trigger and join composition, and preemption in an MSC specification.

This extensive treatment of MSC syntax and semantics prepares the ground for the investigation of the methodical usage of MSCs in the subsequent chapters of this thesis. It allows us to study refinement notions for MSCs, and guides the transition from "global" interaction descriptions to individual component specifications.

CHAPTER 5

MSC Refinement

Based on the semantic framework we have established in the preceding chapter we introduce four notions of MSC refinement in this chapter: *binding references*, *property refinement*, *message* or *interface refinement*, and *structural refinement*. These refinement notions form the basis for turning coarse-grained MSC specifications into fine-grained ones in a systematic way.

Contents

5.1. Introduction **156**
5.2. Binding References **160**
5.3. Property Refinement **161**
5.4. Message Refinement **167**
5.5. Structural Refinement **176**
5.6. Related Work **184**
5.7. Summary **184**

5.1. Introduction

The fundamental aim of system development is to turn an idea of what the system shall do into a product implementing the idea. A systematic development process provides means for establishing that the resulting product meets the requirements corresponding to the before-mentioned idea. One way to achieve this is to start from a first rough sketch of the idea, and to apply to it a number of successive transformations that preserve the idea, adding more and more detail along the way; "preserving the idea" here means maintaining the invariant that the result of each transformation still implements the idea

[Dij68, Wir71, DDH72] have, for the design of (sequential) algorithms, contributed the notions of "stepwise program construction" and "stepwise program refinement" as a methodical approach to achieving a certain goal in the program under development. For instance, [Dij68] notes in a discussion of relating a more detailed and a more abstract program version: "The successive versions appear as successive levels of elaboration. It is apparently essential for each level to make a clear separation between 'what it does' and 'how it works'. The description of 'what it does', the definition of its nett effect, requires introduction of the adequate concepts ... ".

Extending the notion of program refinement to more general system descriptions, we are interested in development steps that relate system descriptions on different levels of detail, such that the more detailed version still captures the "idea" (the "what it does") behind the more abstract description. In this state of affairs we say that the detailed description *refines* the abstract one; in the words of [DW98]: "A refinement is a relationship between two descriptions of the same thing at two levels of detail, wherein one – the *realization* – *conforms* to the other – the *abstraction*. A refinement is accompanied by a mapping that justifies this claim and shows how the abstraction is met by the realization.".

Adopting the view that the definition of a refinement notion requires exhibiting a mapping that makes the relation between a detailed and an abstract system description explicit, we define – based on the semantic framework we have established in Chapter 4 – four refinement relations for MSCs in this chapter: *binding references*, *property refinement*, *message* or *interface refinement*, and *structural refinement*. Together, these refinement relations allow us to change the level of detail of all aspects of an MSC specification. The refinement rules we present along with the definitions of the refinement relations capture the methodical steps from more abstract MSCs to more detailed ones.

In the following paragraphs we briefly describe each of these refinement notions, in turn.

Binding of References The binding of a reference amounts to adding a fresh pair (X, α) to the binding relation *MSCR*. This has the effect of changing the semantics of all references $\rightarrow X$ within the MSC document under consideration from $[\![\mathbf{any}]\!]_u$ to $[\![\alpha]\!]_u$. As an example for refinement through the binding of references recall how we developed the MSC specification of the ABRACADABRA-protocol in Section 4.7. We started out with a single

HMSC A with references to the plain MSCs SX, SY, C, and CR (cf. Figure 4.20 in Section 4.7.2). Without knowledge about the contents of these MSCs A's semantics equals the one of **any**; an MSC document containing only the definition of A is just too unspecific. In Sections 4.7.3 and 4.7.4 we have, step by step, "filled in the holes" by adding the definitions for the plain MSCs. Each such addition has bound one of the references in A and has thus made the behavior represented by A more specific.

Property Refinement Property refinement addresses the reduction of the possible behaviors of the overall system. In our semantic framework this amounts to restricting the set of system behaviors in the equivalence class $[\![\alpha]\!]_u$, if α is the MSC we want to refine. Examples of property refinements are removing alternatives, removing interleaving, and strengthening guards. If β' is the refined version of MSC β (with respect to property refinement), then β' describes an "implementation strategy" for the interaction properties specified by β. In fact, we have already seen an application of property refinement in our treatment of the ABRACADABRA-protocol. In Section 4.7.5 we have removed all behaviors from the protocol specification (given through HMSC A) where a send request of component X did not result in a successful transmission initiated by X within finite time. We achieved this removal by joining a progress property to the original HMSC A.

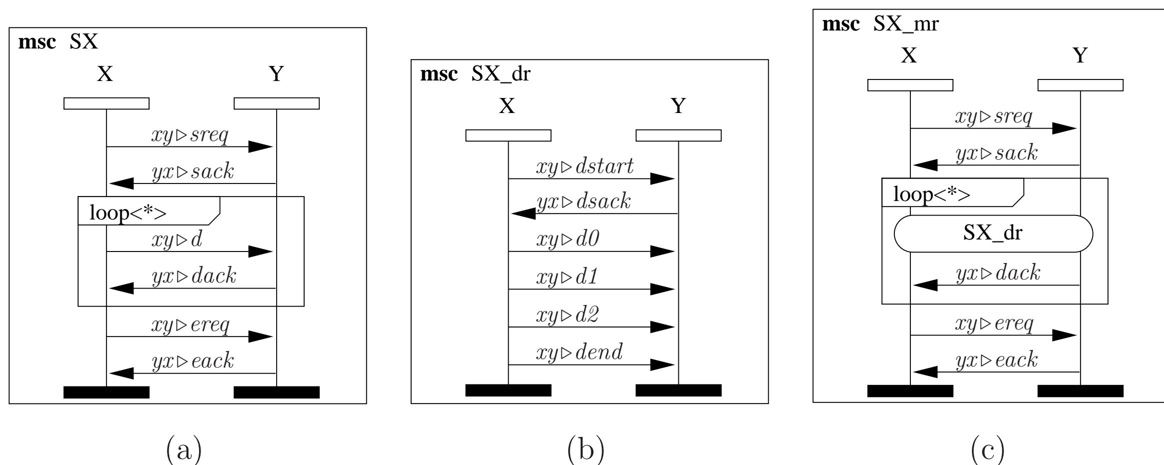


Figure 5.1.: Message refinement

Message Refinement Message refinement denotes the substitution of a whole interaction sequence or protocol for a single message. We can capture a simple version thereof by replacing the message, say $ch \triangleright m$, by a reference to an MSC containing the protocol to be substituted for $ch \triangleright m$. We could, for instance, replace message $xy \triangleright d$ in MSC SX (cf. Figure 5.1 (a)) with the protocol given in Figure 5.1 (b), and obtain MSC SX_mr (cf. Figure 5.1 (c)) as a message refinement of MSC SX. This refinement notion is, however, very liberal with respect to the protocol allowed as a replacement for the original message; therefore, we study also restricted versions of message refinement in this chapter.

Structural Refinement Structural refinement denotes replacing a single component appearing in an MSC α, say p, with a set of (other) components, say $\{p_0, ..., p_n\}$ for some $n \in \mathbb{N}$. Intuitively, this results in decomposing component p into its subcomponents, and redirecting all messages arriving at or emanating from p to corresponding subcomponents. In the ABRACADABRA-protocol we could, for instance, replace the component Y with the two components *Y1* and *Y2*, and obtain MSC *SX_sr* (cf. Figure 5.2) as a structural refinement of MSC *SX* (cf. Figure 5.1 (a)).

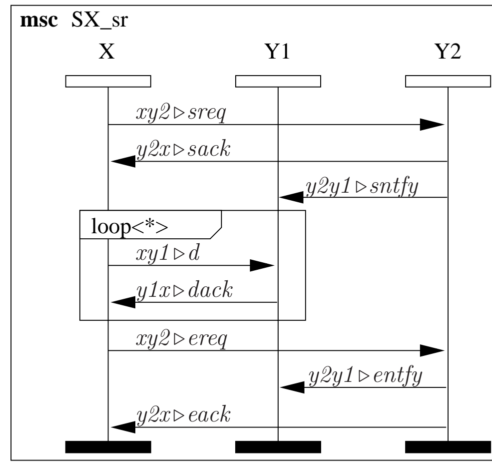


Figure 5.2.: Structural refinement

Refinement and the System Model To give a precise characterization for each of these refinement notions we observe that some refinement steps result in changes to the system model we have introduced in Section 4.2. Structural refinement is an example: here, as we will see below, we want to exclude the reappearance of the refined component p in the refining MSC. We can ensure this by excluding p from the set P of available components. This, however, induces a fresh semantic function $[\![.]\!]_u$, under which we must interpret the refining MSC. Recall that we have introduced $[\![.]\!]_u$ in Section 4.4 on the basis of the fixed sets P (the set of component identifiers), C (the set of channels), M (the set of messages), and S (the set of states). Moreover, we have used the relation *MSCR* to represent the mapping from MSC names to their corresponding MSC terms (as the result of parsing an MSC document) for the resolution of references. Refinement steps that change any of these sets and relations, therefore, induce another semantics mapping $[\![.]\!]_u$. To make clear on what sets the MSC semantics bases we write $[\![.]\!]_u^{(P,C,M,S,MSCR)}$ instead of $[\![.]\!]_u$; we use the latter only if P, C, M, S, and *MSCR* are clear from the context.

We capture the transformation of the system model that underlies a refinement step from a tuple $B = (P, C, M, S, MSCR)$ to $B' = (P', C', M', S', MSCR')$ by relations $\rightsquigarrow$, such that $B \rightsquigarrow B'$ holds, if B and B' are the semantic bases for the abstract and the refined system, respectively. This allows us to take such system model transformations as a parameter to

the definition of the corresponding refinement notion for MSCs. More precisely, for every refinement step on an MSC we capture the underlying system model transformation by defining a specific $\rightsquigarrow$ relation. While this may, at first, seem to be yet another notational complication, it serves a very practical purpose. Each relation $\rightsquigarrow$ we introduce reflects changes in other system views that we better take into consideration when interpreting MSCs. Thus, we can understand each relation $\rightsquigarrow$ as a way of describing consistency conditions for MSC refinements with respect to other system views.

Throughout this chapter, we label the relation $\rightsquigarrow$ with b, p, m, and s to make explicit that it expresses the binding of references, property, message, and structural refinement, respectively. We use the same kind of labeling also for the refinement relations themselves, which we will denote by the symbol $\leq$.

MSC Refinement vs. Component Refinement The refinement notions we introduce in this chapter relate MSCs on different levels of abstraction. Each MSC describes the interplay of a set of components. An interesting question is whether and how refinements on MSCs correspond to refinements of the individual components or processes appearing in the MSCs. Typically, a refinement step on an MSC affects multiple components. The removal of a message by performing a property refinement, for instance, weakens the "assumptions" made by the recipient at its environment, while it strengthens the "commitment" of the sender. We defer this discussion until Chapter 7, where we investigate the relationship between MSCs and individual component specifications in detail.

The remainder of this chapter has the following structure. In Section 5.2 we treat the binding of references. Then, in Section 5.3, we discuss property refinement. Besides giving its definition, which we base on set inclusion, in Section 5.3.1 we also present a set of concrete refinement rules for turning a given MSC into a more detailed one. The compositionality of property refinement is the topic of Section 5.3.2. We introduce the definitions of message and structural refinement in Sections 5.4 and 5.5, respectively. Both of these follow the intuitive and pragmatic explanations we have given above. For message refinement this intuition fails in the presence of interleaving in the MSCs under consideration. We discuss solutions to this problem in Section 5.4.2. Section 5.7 contains a summary.

5.2. Binding References

In Section 4.4 we have introduced the semantics of an MSC reference $\rightarrow X$ as the semantics of the MSC α it refers to, if there is a corresponding pair $(X, \alpha) \in \mathit{MSCR}$. Otherwise, i.e. if such a pair does not exist in MSCR, the semantics of $\rightarrow X$ equals the one of **any**.

The methodical motivation for defining MSC referencing in this way is to use references as placeholders for more specific interaction sequences to be added later in the development process. As soon as we introduce a pair (X, α) into MSCR for a previously unbound reference $\rightarrow X$ we have established a refinement step for all MSC terms that include such a reference.

Adding a fresh MSC to an MSC document, which induces adding a new binding relation between MSC names and MSC terms, constitutes a change to the base of the semantics definition. Therefore, we capture the addition of a fresh pair $(X, \beta) \in \langle\text{MSCNAME}\rangle \times \langle\text{MSC}\rangle$ to an MSC document for the semantics bases $B = (P, C, M, S, \mathit{MSCR})$ and $B' = (P', C', M', S', \mathit{MSCR}')$ through the relation $\overset{X,\beta}{\underset{b}{\rightsquigarrow}}$ with

$$B \overset{X,\beta}{\underset{b}{\rightsquigarrow}} B' \overset{\mathsf{def}}{=} P = P' \wedge M = M' \wedge C = C' \wedge S = S' \\ \wedge\ \mathit{MSCR}' = \mathit{MSCR} \uplus \{(X, \beta)\}$$

In this definition we use the dyadic set operator $. \uplus .$, which yields the disjoint union of its operands.

Our goal now is to define a refinement relation $\leq_b$ such that $\alpha' \leq_b \alpha$ holds for two MSCs α' and α if the semantics base underlying the interpretation of α' binds at least as many MSC references as the semantics base underlying the interpretation of α, and – with respect to their corresponding semantics bases – the behaviors of α' are a subset of the behaviors of α. To this end, we proceed in three steps. First, we introduce the relation $\leq_b^{\mathit{Sys}}$; it regards the semantics bases B and B', as well as the MSC name X and the MSC term β as given. We define $\leq_b^{\mathit{Sys}}$ such that it implies the validity of $B \overset{X,\beta}{\underset{b}{\rightsquigarrow}} B'$. Intuitively, $\leq_b^{\mathit{Sys}}$ captures a single refinement step between the two MSCs under consideration with respect to a corresponding transformation of the underlying system model. Second, we abstract from the concrete values for B, B', X, and β, and introduce the refinement relation $\leq_b^{\exists}$ such that $\alpha' \leq_b^{\exists} \alpha$ holds if there exist adequate semantics bases, MSC names and terms to establish the validity of the relation $\alpha' \leq_b^{\mathit{Sys}} \alpha$. Third, because we want $\leq_b$ to be transitive and reflexive, we define $\leq_b$ as the transitive, reflexive closure of $\leq_b^{\exists}$. Transitive refinement relations allow us to perform successive refinement steps – where the MSC resulting from one refinement is the starting point for the next –, and to establish a relation between the original MSC and the overall result of the sequence of refinement steps. These considerations lead us to the following definition for the binding of references as a refinement notion for MSCs:

Definition 4 (Binding References) For all $\alpha, \alpha' \in \langle\text{MSC}\rangle$, and the semantics bases $B = (P, C, M, S, MSCR)$, and $B' = (P', C', M', S', MSCR')$ we define

$$\begin{aligned} \alpha' \leq_b^{Sys} \alpha &\stackrel{\text{def}}{=} B \underset{b}{\overset{X,\beta}{\leadsto}} B' \wedge [\![\alpha']\!]_0^{B'} \subseteq [\![\alpha]\!]_0^B \\ \alpha' \leq_b^{\exists} \alpha &\stackrel{\text{def}}{=} \langle \exists B, B', X, \beta :: \alpha' \leq_b^{Sys} \alpha \rangle \\ \leq_b &\stackrel{\text{def}}{=} (\leq_b^{\exists})^* \end{aligned}$$

where $(\leq_b^{\exists})^*$ denotes the reflexive, transitive closure of $\leq_b^{\exists}$. We say "α' refines α (with respect to the binding of references)" or "α' is a binding refinement of α", if $\alpha' \leq_b \alpha$ holds. □

The refinement through the binding of references allows us to start the development process with very abstract MSCs, such as the MSC A in Figure 4.20 (cf. Section 4.7.2), and to add MSCs that resolve the unbound references, incrementally. This corresponds to using MSCs methodically along the lines of the classical top-down stepwise refinement notions of [Dij68, Wir71, DDH72].

5.3. Property Refinement

Recall from Proposition 3 (cf. Section 4.5.2) that $[\![\alpha]\!]_u$ is the equivalence class of all system behaviors exhibiting the behavior specified by α from time u on. This observation induces a "natural" way of comparing two MSCs α and β: simply compare the sets of system behaviors represented by the two MSCs.

As a simple example, consider the two MSCs $\alpha \mid \beta$ and α. By the definition of operator $. \mid .$ we know that $[\![\alpha \mid \beta]\!]_u = [\![\alpha]\!]_u \cup [\![\beta]\!]_u$ holds. This shows the validity of $[\![\alpha]\!]_u \subseteq [\![\alpha \mid \beta]\!]_u$. Hence, α alone represents at most the behaviors of the alternative construct $\alpha \mid \beta$. Put another way, removing one alternative from an alternative construct has the potential of reducing the size of the equivalence class of an MSC.

Restricting the set of behaviors represented by an MSC corresponds to turning a less restrictive specification into a more specific one: the MSC $\alpha \mid \beta$ "allows" the behaviors of both α and β, whereas α does, in general, not capture β's contribution. Therefore, we call the restriction of the set of behaviors represented by an MSC also "removing underspecification" or *property refinement*. The following definition makes this informal description precise:

Definition 5 (Property Refinement) For $\alpha, \alpha' \in \langle\text{MSC}\rangle$, and all semantics bases $B = (P, C, M, S, MSCR)$ we define

$$\alpha' \leq_p \alpha \equiv [\![\alpha']\!]_0^B \subseteq [\![\alpha]\!]_0^B$$

and say "α' refines α" (with respect to property refinement) or "α' is a property refinement of α", if $\alpha' \leq_p \alpha$ holds. □

Property refinement inherits reflexivity, transitivity, and antisymmetry directly from set inclusion. The independence of the semantics mapping $[\![.]\!]_u$ of the concrete value of u (cf. Equation (4.2)) allows us to pick 0 as the time point for the comparison of the equivalence classes in the definition above. The same system model underlies the evaluation of MSCs α and α' in this definition. This captures that besides restricting the set of executions a property refinement has no further consequences on a specification.

Continuing the example we have started above, we recognize α as a property refinement of $\alpha \mid \beta$. In the remainder of this section we study ways of refining given MSCs besides the removal of alternatives (cf. Section 5.3.1), and discuss how "compositional" property refinement is, i.e. under what circumstances the refinement of part of an MSC results in a refinement of the whole MSC (cf. Section 5.3.2).

5.3.1. Refinement Rules

Now that we have the notion of property refinement available, we can investigate transformations on MSCs leading to refined versions. Each such transformation constitutes a systematic development step that makes the MSC specification more precise.

In Appendix B.2 we formally justify the validity of the refinement relationships we describe here.

Every MSC refines any

any is the least specific MSC term available, i.e. for all $\alpha \in \langle\text{MSC}\rangle$ we have

$$\alpha \ \leq_p \ \textbf{any}$$

Strengthening Guards

The guard p of a guarded MSC $p : \alpha$ determines which elements of $[\![\alpha]\!]_u$ end up in $[\![p : \alpha]\!]_u$. In Section 4.4 we have seen two extreme examples of guarded MSCs: $[\![\text{true} : \alpha]\!]_u = [\![\alpha]\!]_u$ and $[\![\text{false} : \alpha]\!]_u = \emptyset$. The stronger the guard p, the more restricted is the set $[\![p : \alpha]\!]_u$. More precisely, we have for all $\alpha \in \langle\text{MSC}\rangle$ and $p, q \in \langle\text{GUARD}\rangle$:

$$(p \Rightarrow q) \ \ \Rightarrow \ \ p : \alpha \ \leq_p \ q : \alpha$$

Removing Alternatives

In the motivation for the definition of property refinement above we have already shown that removing one alternative from an alternative construct yields a property refinement.

More precisely, due to the symmetric definition of $[\![\alpha \mid \beta]\!]_u$ with respect to α and β, each of the following two refinement relations holds for all $\alpha, \beta \in \langle \text{MSC} \rangle$:

$$\begin{aligned} \alpha &\leq_p \alpha \mid \beta \\ \beta &\leq_p \alpha \mid \beta \end{aligned}$$

Joining Properties

Elements $(\varphi, t) \in [\![\alpha \otimes \beta]\!]_u$ have corresponding elements $(\varphi, t') \in [\![\alpha]\!]_u$ and $(\varphi, t'') \in [\![\beta]\!]_u$, i.e. each of the following two implications holds:

$$\begin{aligned} (\varphi, t) \in [\![\alpha \otimes \beta]\!]_u &\Rightarrow \langle \exists t' \in [u, t] :: (\varphi, t') \in [\![\alpha]\!]_u \rangle \\ (\varphi, t) \in [\![\alpha \otimes \beta]\!]_u &\Rightarrow \langle \exists t' \in [u, t] :: (\varphi, t') \in [\![\beta]\!]_u \rangle \end{aligned}$$

This follows directly from the join operator's definition, which constrains the behavior of α such that it must also be a behavior of β (and vice versa). In general, we cannot expect the validity of

$$(\varphi, t) \in [\![\alpha \otimes \beta]\!]_u \quad \Rightarrow \quad (\varphi, t) \in [\![\alpha]\!]_u \wedge (\varphi, t) \in [\![\beta]\!]_u$$

as the MSCs $\alpha = \textbf{empty}$ and $\beta = ch \triangleright m$ show. Here, we have $(\varphi, t) \in [\![\alpha]\!]_u \equiv t = u$ and $(\varphi, t) \in [\![\beta]\!]_u \Rightarrow t > u$. Thus, we only obtain a "pseudo-refinement" for the join operator. However, if we abstract from the time-component of the semantics domain, we observe that the set of behaviors of the join is a subset of the behaviors of each operand.

We have already seen an application of this "refinement step" in Section 4.7.5, where we have restricted the behaviors of HMSC A to those fulfilling a certain progress property.

Sequential Composition Refines Interleaving

The purpose of the interleaving operator is to model the absence of causality between the contributions of the operand MSCs to an overall specification. Fixing an arbitrary order between the behaviors modeled by the operand MSCs yields a refinement step. More specifically we obtain that for all $\alpha, \beta \in \langle \text{MSC} \rangle$ each of the following two refinement relations holds:

$$\begin{aligned} \alpha ; \beta &\leq_p \alpha \sim \beta \\ \beta ; \alpha &\leq_p \alpha \sim \beta \end{aligned}$$

These two refinement relations identify sequential composition as an implementation strategy for interleaving.

Narrowing Loop Bounds

Besides guarded loops – whose refinement we have discussed above, in connection with the one of guarded MSCs – we have introduced bounded, unbounded, and infinite loops in Section 4.4.

Narrowing the bounds of a bounded loop yields a property refinement, i.e. for all $\alpha \in \langle \text{MSC} \rangle$, and $m, m', n, n' \in \mathbb{N}_\infty$ we have

$$[m', n'] \subseteq [m, n] \quad \Rightarrow \quad \alpha\uparrow_{<m', n'>} \;\leq_p\; \alpha\uparrow_{<m,n>}$$

Similarly, all finite bounded repetitions of an MSC α refine α's unbounded repetition, i.e. for all $m \in \mathbb{N}$ we have

$$\alpha\uparrow_{<0,m>} \;\leq_p\; \alpha\uparrow_{<*>}$$

Removing Preemption

In a preemption construct, such as $\alpha \overset{ch \triangleright m}{\rightarrow} \beta$, the interruption of α and the immediate continuation of the corresponding execution as specified by β depends on the occurrence of the message $ch \triangleright m$ during the time interval covered by α. If $ch \triangleright m$ does not occur during this time interval the preemption does not happen, and $[\![\alpha \overset{ch \triangleright m}{\rightarrow} \beta]\!]_u$ equals $[\![\alpha]\!]_u$. More formally:

$$(\alpha \;\equiv_u\; \alpha \overset{ch \triangleright m}{\rightarrow} \beta) \quad \Leftarrow \quad \langle \forall t, t', \varphi : (\varphi, t) \in [\![\alpha]\!]_u \;\wedge\; u \leq t' \leq t : m \notin \pi_1(\varphi).t'.ch \rangle$$

Therefore, to yield a refinement relation like $\alpha \;\leq_p\; \alpha \overset{ch \triangleright m}{\rightarrow} \beta$ we have to find criteria for excluding the occurrence of $ch \triangleright m$ during executions that "comply to" α.

However, a simple criterion like $ch \triangleright m \notin msgs.\alpha$ does not suffice. Remember that $[\![\alpha]\!]_u$ represents *all* behaviors evolving as α "requires". The messages occurring syntactically in α appear in the specified order in $[\![\alpha]\!]_u$. $[\![\alpha]\!]_u$ makes no statement whatsoever about messages absent from α. As we will discuss in more detail in Chapter 6, we need this very loose MSC interpretation to relate MSCs with other system views.

Therefore, under the given circumstances, we can only formulate a very strong "context condition" that explicitly excludes the occurrences of certain messages. To that end, we add another MSC term, the "message closure" operator $\lceil . \rceil^{\cdot}$, and define its semantics for all $\alpha \in \langle \text{MSC} \rangle$ and $N \subseteq M$ by

$$[\![\lceil \alpha \rceil^N]\!]_u \;\overset{\text{def}}{=}\; \{(\varphi, t) \in [\![\alpha]\!]_u : \langle \forall t', ch, m : u \leq t' \leq t : m \in \pi_1(\varphi).t'.ch \Rightarrow ch \triangleright m \in N \rangle\}$$

Thus, $[\![\lceil\alpha\rceil^N]\!]_u$ contains only pairs (φ, t) such that messages outside N do not occur in φ within the time interval $[u, t]$.

Equipped with this operator we can easily prove the validity of the following refinement relation, where $\overline{M}$ denotes the set of all channel and message pairs, i.e. $\overline{M} \stackrel{\mathsf{def}}{=} \{ch \triangleright m : ch \in C \wedge m \in M\}$:

$$\lceil\alpha\rceil^{\overline{M}\setminus\{ch\triangleright m\}} \ \leq_p \ \alpha \stackrel{ch\triangleright m}{\rightarrow} \beta$$

Similarly, we can show that the following refinement relation holds:

$$\lceil\alpha\rceil^{\overline{M}\setminus\{ch\triangleright m\}} \ \leq_p \ \alpha {\Uparrow}_{ch\triangleright m}$$

The message closure operator induces a very strong assumption at the context in which the behavior represented by an MSC may occur. Checking that the context satisfies this assumption is, in general, possible “at runtime” only, which is unsatisfactory for most practical purposes. We can be sure of the absence of a particular message during a certain time interval only if we have complete information about the behavior of all parts of the system. Later, in Chapter 7, we will study a much more restricted MSC interpretation under which $(ch\triangleright m \notin msgs.\alpha) \ \Rightarrow \ (\alpha \ \equiv_u \ \lceil\alpha\rceil^{\overline{M}\setminus\{ch\triangleright m\}})$ holds. This is a context condition whose validity we can check syntactically.

LHS Weakening And RHS Strengthening of Trigger Composition

Trigger composition inherits anti-monotonicity in its first argument from the use of the implication operator in the semantics definition in Section 4.4. As a consequence we have to handle the refinement of trigger composition with care: we cannot simply refine each of the trigger composition's operands to yield a refinement of the trigger composition as a whole.

In analogy to the handling of predicates and the implication operator, we call development steps leading from an MSC α to an MSC α' such that $\alpha' \ \leq_p \ \alpha$ holds “strengthening” steps. Similarly, we call development steps leading from α' to α such that $\alpha' \ \leq_p \ \alpha$ holds “weakening” steps.

Using this vocabulary, we can state the refinement properties of the trigger composition as follows: weakening the left-hand-side and strengthening the right-hand-side of a trigger composition both result in a property refinement. More precisely, we have for all $\alpha, \alpha', \beta, \beta' \in \langle \mathrm{MSC} \rangle$:

$$((\alpha' \ \leq_p \ \alpha) \wedge (\beta' \ \leq_p \ \beta)) \ \Rightarrow \ ((\alpha \mapsto \beta') \ \leq_p \ (\alpha' \mapsto \beta))$$

5.3.2. Compositionality

A refinement notion is particularly useful if it is compositional, i.e. if the refinement of one of a composite's constituents yields a refinement of the composite. A compositional refinement notion allows us to refine parts of an MSC (specification) individually, without knowledge about the rest of the MSC (specification).

Fortunately, property refinement is – almost – compositional with respect to our MSC dialect. The only exception is trigger composition, whose anti-monotonicity in its first argument destroys general compositionality. The following proposition makes this informal claim more precise.

Proposition 5 (Compositionality of Property Refinement) *Property refinement is compositional for MSCs that do not contain trigger composition. In particular, for all* $\alpha, \beta, \gamma \in \langle MSC \rangle$, $p \in \langle GUARD \rangle$, $m, n \in \mathbb{N}_\infty$, $<\!L\!> \in \{<\!*\!>, <\!m{,}n\!>, <\!m\!>\}$, $\dagger \in \{;\, , |, \sim, \otimes\}$, $X, Y \in \langle MSCNAME \rangle$*, with* $(X, \alpha'), (Y, \alpha) \in MSCR$*, and* $\alpha' \ \leq_p \ \alpha \ \wedge \ \beta' \ \leq_p \ \beta$*, each of the following refinement relations holds:*

$$p : \alpha' \ \leq_p \ p : \alpha \tag{5.1}$$

$$\alpha' \dagger \gamma \ \leq_p \ \alpha \dagger \gamma \tag{5.2}$$

$$\gamma \dagger \alpha' \ \leq_p \ \gamma \dagger \alpha \tag{5.3}$$

$$\alpha' \uparrow_{<\!L\!>} \ \leq_p \ \alpha \uparrow_{<\!L\!>} \tag{5.4}$$

$$\rightarrow X \ \leq_p \rightarrow Y \tag{5.5}$$

$$\alpha' \stackrel{ch \triangleright m}{\rightarrow} \beta' \ \leq_p \ \alpha \stackrel{ch \triangleright m}{\rightarrow} \beta \tag{5.6}$$

$$\alpha' \Uparrow_{ch \triangleright m} \ \leq_p \ \alpha \Uparrow_{ch \triangleright m} \tag{5.7}$$

$$\alpha \ \mapsto \ \beta' \ \leq_p \ \alpha' \ \mapsto \ \beta \tag{5.8}$$

□

Proof The validities of relations (5.1) through (5.7) follow directly from the use of only set inclusion tests, set union and conjunction in the semantics definitions of the corresponding operators (cf. Section 4.4). We discussed the validity of (5.8) in the previous section. ■

This proposition has a methodical implication. We can perform arbitrary property refinements on any part of an MSC α, as long as this part does not contain a trigger composition. This suggests to keep MSCs for the specification of fairness properties separate from the "other" MSCs. Then, we can perform property refinement in a compositional way on the non-trigger part of the specification, and add the fairness conditions later. We will pick up this issue again in Chapter 6, where we discuss the relationship between individual MSCs and other parts of a system specification.

5.4. Message Refinement

With message refinement we want to capture two kinds of modifications to a specification (cf. [Bro93, Bro99a]):

1. changing the number and names of channels connecting components;
2. changing the representation or granularity of messages occurring on the channels.

Message refinement allows us to start with rough sketches of the actual behavior when we construct a specification. The messages we begin with may indicate the nett effect of a more elaborate interaction, while a refinement reveals the actual message exchange within the system's implementation.

As an example, consider a system consisting of two components: *Customer* and *Vendor*. There are two channels that connect the components: *cv* (directed from *Customer* to *Vendor*), and *vc* (directed from *Vendor* to *Customer*). We can capture a purchase performed by the customer using the MSC of Figure 5.3 (a): the customer sends a "purchase" order along *cv*, and the *Vendor* returns the "delivery" on *vc*. Clearly, placing an order typically is not an atomic action: it usually consists of selecting the item to be purchased, and paying for it. MSC *prchsr* in Figure 5.3 (b) shows the sequence of messages *selectItem* and *pay* instead of the more abstract message *purchase* in Figure 5.3 (a). Despite this difference, both MSCs represent the same nett effect: the customer places an order, and the vendor delivers. Therefore, we would like to consider the MSC from Figure 5.3 (b) a refinement of the one from Figure 5.3 (a).

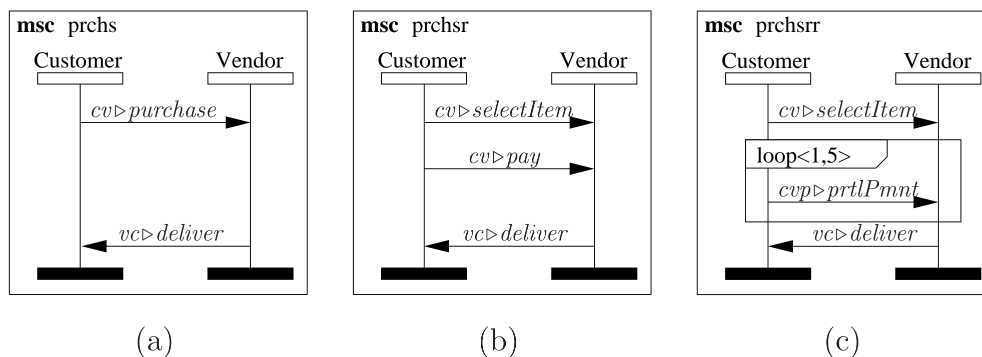
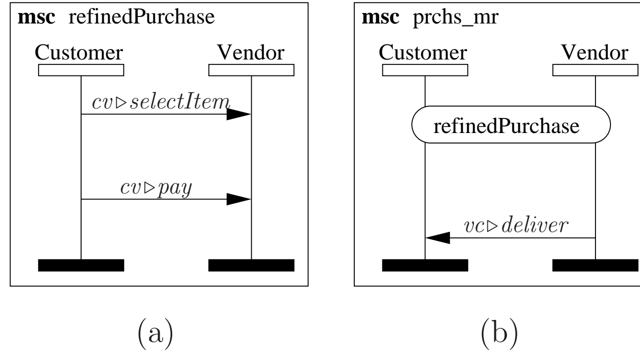


Figure 5.3.: Three MSCs with messages on different levels of detail

Similarly, we want to view the MSC of Figure 5.3 (c) as a refinement of the one of Figure 5.3 (b); Figure 5.3 (c) adds the information that the payment happens in form of up to five installments along a fresh channel *cvp*. Again, the overall ordering of the "actions" performed has not changed; the only difference is the level of detail revealed by the respective MSCs about the interaction between *Customer* and *Vendor*.

To get a first idea for a precise definition of message refinement, we observe that using the MSC referencing mechanism we can nicely capture the effect of replacing a single message with an entire protocol. Together, the MSCs in Figure 5.4 (a) and (b) represent the same interaction sequence as the MSC from Figure 5.3 (b). From *prchs* we can derive an MSC that represents the same behavior as MSC *prchsr* simply by substituting the reference to *refinedPurchase* for the occurrence of message $cv \triangleright purchase$.



(a) (b)

Figure 5.4.: Capturing message refinement through referencing

Generalizing this example, we consider MSC α' a message refinement of MSC α with respect to message $ch \triangleright m \in msgs.\alpha$ if there is a pair $(X, \beta) \in \langle\text{MSCNAME}\rangle \times \langle\text{MSC}\rangle$ such that if we substitute $\rightarrow X$ for all occurrences of $ch \triangleright m$ in α we obtain an MSC subsuming the behavior of α'.

As was the case with the binding of references, the addition of a pair (X, β) results in a change of the semantic model. Because we also allow the set of channels and messages to change in a message refinement step, we present the formal definition of message refinement in two stages. First, we introduce relation $\overset{ch \triangleright m, X, \beta}{\underset{m}{\rightsquigarrow}}$; it describes the changes to the semantic model underlying the refinement of message $ch \triangleright m$ into the protocol represented by the pair (X, β). Second, we describe the substitution process formally and use it to relate the MSCs on different levels of detail.

Before we can formulate relation $\overset{ch \triangleright m, X, \beta}{\underset{m}{\rightsquigarrow}}$, a few considerations about the protocol allowed as a replacement for message $ch \triangleright m$ are in order; clearly, we want the protocol to maintain the intuitive meaning we associate with messages. We don't want to replace a message occurrence with empty behavior; this would turn the mandatory occurrence of the abstract message into an optional protocol in the refined version. We consider this to be as counter-intuitive as the replacement of a single message with an infinite protocol. If the abstract message occurs on its channel at all, this happens within finite time. Therefore, we require the replacing protocol to exhibit neither empty nor infinite behavior.

Thus, for two given tuples $B = (P, C, M, S, MSCR)$ and $B' = (P', C', M', S', MSCR')$, a message $ch \triangleright m \in \overline{M} \overset{\text{def}}{=} \{ch' \triangleright m' : ch' \in C \wedge m' \in M\}$, a pair $(X, \beta) \in \langle\text{MSCNAME}\rangle \times$

$\langle\text{MSC}\rangle$ we define

$$B \overset{ch\triangleright m, X, \beta}{\underset{m}{\leadsto}} B' \overset{\mathsf{def}}{=} \quad M \subseteq M' \tag{5.9}$$
$$\wedge\ C \subseteq C' \tag{5.10}$$
$$\wedge\ MSCR' = MSCR \uplus \{(X, \beta)\} \tag{5.11}$$
$$\wedge\ ch\triangleright m \notin msgs.\beta \tag{5.12}$$
$$\wedge\ \neg(\textbf{empty} \ \leq_p\ \beta) \tag{5.13}$$
$$\wedge\ \beta\ \leq_p\ \textbf{fany} \tag{5.14}$$

This definition captures the following properties:

1. the sets of messages and channels can increase in the refined system (conjuncts (5.9) and (5.10)),
2. the pair (X, β) is the only addition to the binding relation (conjunct (5.11)),
3. the refinement is proper, i.e. the abstract message does not reappear in the refining protocol (conjunct (5.12)),
4. β cannot exhibit empty behavior (conjunct (5.13)),
5. β represents finite behavior (conjunct (5.14)).

The second step is to formalize the substitution process. With one exception the definition of the substitution of reference $\rightarrow X$ for message $ch\triangleright m$ is straightforward. Only preemption requires special treatment. If we decide to refine MSC α with respect to message $ch\triangleright m$, and α contains a preemption specification like $\gamma_0 \overset{ch\triangleright m}{\rightarrow} \gamma_1$ or $\gamma_0 {\Uparrow}_{ch\triangleright m}$ for some $\gamma_0, \gamma_1 \in \langle\text{MSC}\rangle$, then it is not entirely obvious what result the substitution should produce. The syntax for preemption, as we have introduced it, allows a single preemptive message only. This suggests to designate one of the messages that occur in the refining protocol as the new preemptive message. Moreover, because the occurrence of the abstract preemptive message on its channel is an atomic “action”, and preemption only happens when the message has indeed occurred, the new preemptive message should occur at the end of the refining protocol. Based on these two conventions we can refine preemptive messages in the same way as the other messages.

These considerations lead to the following definition for the substitution operator

$$.[.,./.] : \langle\text{MSC}\rangle \times \langle\text{MSCNAME}\rangle \times \langle\text{MSG}\rangle \times \langle\text{MSG}\rangle \rightarrow \langle\text{MSC}\rangle$$

which, for given MSC α, MSC name X, and messages $\tilde{ch}\triangleright\tilde{m}$ and $ch\triangleright m$ substitutes reference $\rightarrow X$ for all occurrences of $ch\triangleright m$ in α except for those occurrences where $ch\triangleright m$ acts as

a preemptive message. In these cases, $\tilde{ch} \triangleright \tilde{m}$ replaces $ch \triangleright m$. We define the substitution operator for $\dagger \in \{\, ;\, ,\, |\, ,\, \sim\, ,\, \otimes\, ,\, \mapsto \}$, $ch' \triangleright m' \in \langle \text{MSG} \rangle$, and $\alpha, \beta \in \langle \text{MSC} \rangle$ as follows:

$$
\begin{aligned}
\mathbf{empty}[X, \tilde{ch} \triangleright \tilde{m}/ch \triangleright m] &\stackrel{\mathsf{def}}{=} \mathbf{empty} \\
(ch' \triangleright m')[X, \tilde{ch} \triangleright \tilde{m}/ch \triangleright m] &\stackrel{\mathsf{def}}{=} \begin{cases} \rightarrow X & \text{if } ch' \triangleright m' = ch \triangleright m \\ ch' \triangleright m' & \text{else} \end{cases} \\
(\alpha \dagger \beta)[X, \tilde{ch} \triangleright \tilde{m}/ch \triangleright m] &\stackrel{\mathsf{def}}{=} (\alpha[X, \tilde{ch} \triangleright \tilde{m}/ch \triangleright m]) \dagger (\beta[X, \tilde{ch} \triangleright \tilde{m}/ch \triangleright m]) \\
(\rightarrow Y)[X, \tilde{ch} \triangleright \tilde{m}/ch \triangleright m] &\stackrel{\mathsf{def}}{=} \rightarrow Y \\
(\alpha \stackrel{ch' \triangleright m'}{\rightarrow} \beta)[X, \tilde{ch} \triangleright \tilde{m}/ch \triangleright m] &\stackrel{\mathsf{def}}{=} \begin{cases} (\alpha[X, \tilde{ch} \triangleright \tilde{m}/ch \triangleright m]) \stackrel{\tilde{ch} \triangleright \tilde{m}}{\rightarrow} (\beta[X, \tilde{ch} \triangleright \tilde{m}/ch \triangleright m]) \\ \quad \text{if } ch' \triangleright m' = ch \triangleright m \\ (\alpha[X, \tilde{ch} \triangleright \tilde{m}/ch \triangleright m]) \stackrel{ch' \triangleright m'}{\rightarrow} (\beta[X, \tilde{ch} \triangleright \tilde{m}/ch \triangleright m]) \\ \quad \text{else} \end{cases} \\
(\alpha \Uparrow_{ch' \triangleright m'})[X, \tilde{ch} \triangleright \tilde{m}/ch \triangleright m] &\stackrel{\mathsf{def}}{=} \begin{cases} (\alpha[X, \tilde{ch} \triangleright \tilde{m}/ch \triangleright m]) \Uparrow_{\tilde{ch} \triangleright \tilde{m}} & \text{if } ch' \triangleright m' = ch \triangleright m \\ (\alpha[X, \tilde{ch} \triangleright \tilde{m}/ch \triangleright m]) \Uparrow_{ch' \triangleright m'} & \text{else} \end{cases}
\end{aligned}
$$

To determine the set of messages allowed as the final messages of a refining protocol we introduce the function

$$
lastmsgs : \langle \text{MSC} \rangle \rightarrow \mathcal{P}(\langle \text{MSG} \rangle)
$$

For $\dagger \in \{\, ;\, ,\, \mapsto \}$, $\ddagger \in \{\, |\, ,\, \sim\, ,\, \otimes \}$, $\alpha, \beta \in \langle \text{MSC} \rangle$, $ch \triangleright m \in \langle \text{MSG} \rangle$, and a given semantics base $(P, C, M, S, MSCR)$, we define, with $\overline{M} \stackrel{\mathsf{def}}{=} \{ch \triangleright m : ch \in C \wedge m \in M\}$:

$$
\begin{aligned}
lastmsgs.\mathbf{empty} &\stackrel{\mathsf{def}}{=} \emptyset \\
lastmsgs.\mathbf{any} &\stackrel{\mathsf{def}}{=} \overline{M} \\
lastmsgs.ch \triangleright m &\stackrel{\mathsf{def}}{=} \{ch \triangleright m\} \\
lastmsgs.(\alpha \dagger \beta) &\stackrel{\mathsf{def}}{=} lastmsgs.\beta \\
lastmsgs.(\alpha \ddagger \beta) &\stackrel{\mathsf{def}}{=} lastmsgs.\alpha \cup lastmsgs.\beta \\
lastmsgs.(\alpha \stackrel{ch \triangleright m}{\rightarrow} \beta) &\stackrel{\mathsf{def}}{=} lastmsgs.\alpha \cup lastmsgs.\beta \\
lastmsgs.(\alpha \Uparrow_{ch \triangleright m}) &\stackrel{\mathsf{def}}{=} lastmsgs.\alpha \\
lastmsgs.(\rightarrow X) &\stackrel{\mathsf{def}}{=} \begin{cases} lastmsgs.\alpha & \text{if } (X, \alpha) \in MSCR \\ \overline{M} & \text{else} \end{cases}
\end{aligned}
$$

With these preliminaries in place, we define message refinement on MSCs as follows:

Definition 6 (Message Refinement) For all $\alpha, \alpha' \in \langle \mathrm{MSC} \rangle$, and semantics bases $B = (P, C, M, S, MSCR)$, and $B' = (P', C', M', S', MSCR')$ we define

$$\begin{aligned}
\alpha' \leq_m^{Sys} \alpha &\stackrel{\mathsf{def}}{=} B \overset{ch \triangleright m, X, \beta}{\underset{m}{\rightsquigarrow}} B' \\
&\quad \wedge \langle \exists \tilde{ch} \triangleright \tilde{m} : \{\tilde{ch} \triangleright \tilde{m}\} = \mathit{lastmsgs}.\beta : [\![\alpha']\!]_0^{B'} \subseteq [\![\alpha[X, \tilde{ch} \triangleright \tilde{m} / ch \triangleright m]]\!]_0^{B'} \rangle \\
\alpha' \leq_m^{\exists} \alpha &\stackrel{\mathsf{def}}{=} \langle \exists B, B', ch \triangleright m, X, \beta :: \alpha' \leq_m^{Sys} \alpha \rangle \\
\leq_m &\stackrel{\mathsf{def}}{=} (\leq_m^{\exists})^*
\end{aligned}$$

where $(\leq_m^{\exists})^*$ denotes the reflexive, transitive closure of $\leq_m^{\exists}$. We say "α' refines α (with respect to message refinement)" or "α' is a message refinement of α", if $\alpha' \leq_m \alpha$ holds. □

According to this definition, $\alpha' \leq_m^{Sys} \alpha$ holds for a given MSC term β, and a message $ch \triangleright m$ in $msgs.\alpha$ if the semantics of α' is a subset of the one of the MSC term obtained by replacing all occurrences of $ch \triangleright m$ in α with a reference to β. Thus, $\alpha' \leq_m^{Sys} \alpha$ captures a single message refinement step with respect to a corresponding transformation of the underlying system model. Relation $\leq_m^{\exists}$ abstracts from concrete semantics bases B and B', messages $ch \triangleright m$, and MSC definitions (X, β). It holds if we can find an appropriate MSC term β, and a message $ch \triangleright m$ such that $\alpha' \leq_m^{Sys} \alpha$ is true. $\leq_m^{\exists}$ is not transitive: if we have $\alpha'' \leq_m^{\exists} \alpha'$ and $\alpha' \leq_m^{\exists} \alpha$, then we cannot, in general, exhibit a *single* MSC β and a *single* message $ch \triangleright m$ to derive $\alpha'' \leq_m^{\exists} \alpha$. Therefore, we have forced transitivity (and reflexivity) on $\leq_m$ by taking the reflexive, transitive closure of $\leq_m^{\exists}$ as the definition for $\leq_m$.

As an example, consider the MSCs *prchs* and *prchsr* from Figures 5.3 (a) and (b), respectively. We have

$$\mathit{prchsr} \leq_m \mathit{prchs}$$

because we obtain MSC *prchsr_mr* (cf. Figure 5.4(b)) by replacing the occurrence of message $cv \triangleright \mathit{purchase}$ in MSC *prchs* with a reference to MSC *refinedPurchase*; by expanding the reference we immediately obtain the validity of $[\![\mathit{prchsr}]\!]_0 = [\![\mathit{prchs_mr}]\!]_0$.

5.4.1. Refinement Rule

The "operational" definition we have given for message refinement immediately suggests how to carry out a concrete refinement of an MSC α:

1. Select a message $ch \triangleright m \in msgs.\alpha$. $ch \triangleright m$ is the message we want to replace with an entire protocol.

2. Define the refining protocol $\beta \in \langle \mathrm{MSC} \rangle$ such that β fulfills the requirements posed by the definitions of the relations $\overset{\cdots}{\underset{m}{\rightsquigarrow}}$ and $\leq_m$:

(a) $ch \triangleright m$ must not occur in β ($ch \triangleright m \notin msgs.\beta$),

(b) β must not display empty behavior ($\neg(\mathbf{empty} \ \leq_p \ \beta)$),

(c) β must not display infinite behavior ($\beta \ \leq_p \ \mathbf{fany}$),

(d) β may introduce fresh messages and fresh channels,

(e) $lastmsgs.\beta$ must be a singleton, i.e. $lastmsgs.\beta = \{\tilde{ch} \triangleright \tilde{m}\}$ for some $\tilde{ch} \triangleright \tilde{m} \in \langle \text{MSG} \rangle$; this means that β is semantically equivalent to an MSC of the form $\gamma \, ; \, \tilde{ch} \triangleright \tilde{m}$ for some $\gamma \in \langle \text{MSC} \rangle$ with $\gamma \ \leq_p \ \mathbf{fany}$.

3. Perform a global substitution (i.e. a substitution in all MSCs of the MSC document under consideration) of β for $ch \triangleright m$, mimicking the definition of the substitution operator $[.,./.]$, to yield the refined MSC α'.

The substitution must be global, because we want the refinement to be consistent across all MSCs of the MSC document; this includes, in particular, all MSCs referred to by α.

Clearly, after performing these steps, we obtain the validity of the relation $\alpha' \leq_m \alpha$; we can easily exhibit tuples B and B', and an MSC definition $\mathbf{msc}\ X = \beta$ such that

$$\alpha' \leq_m^{Sys} \alpha$$

holds.

5.4.2. Problems With Message Refinement

Our definition of message refinement is a very pragmatic one: simply syntactically replace the message under consideration with an entire protocol, everywhere the message occurs in the MSC document. This intuitive refinement notion is particularly well-suited for the incorporation into tools performing the substitution automatically.

Unfortunately, however, the intuition comes at a price: message refinement does not preserve the equivalence relation $\equiv_u$ we have defined in Section 4.4, i.e. $\equiv_u$ is not a congruence with respect to message refinement. This means that if we have $\alpha, \alpha', \beta, \beta' \in \langle \text{MSC} \rangle$ such that we obtain α' from α and β' from β by means of the exact same message refinement, and we also have $\alpha \ \equiv_u \ \beta$ we cannot, in general, conclude $\alpha' \ \equiv_u \ \beta'$.

To illustrate this we adapt an example from [vG96]. Consider the two MSC terms

$$\alpha \ = \ (a \triangleright m_a \, ; \, b \triangleright m_b) \ \sim \ c \triangleright m_c$$

and

$$\beta \ = \ (a \triangleright m_a \, ; \, (b \triangleright m_b \ \sim \ c \triangleright m_c)) \mid ((a \triangleright m_a \ \sim \ c \triangleright m_c) \, ; \, b \triangleright m_b)$$

where a, b, c are channels and m_a, m_b, m_c the corresponding messages. It is easy to see that

$$\alpha \equiv_u \beta$$

holds. If we refine $c \triangleright m_c$ through the sequence $(c_1 \triangleright m_{c_1} ; c_2 \triangleright m_{c_2})$ in both MSC terms, we obtain:

$$\alpha' = (a \triangleright m_a ; b \triangleright m_b) \sim (c_1 \triangleright m_{c_1} ; c_2 \triangleright m_{c_2})$$

and

$$\beta' = (a \triangleright m_a ; (b \triangleright m_b \sim (c_1 \triangleright m_{c_1} ; c_2 \triangleright m_{c_2}))) \mid ((a \triangleright m_a \sim (c_1 \triangleright m_{c_1} ; c_2 \triangleright m_{c_2})) ; b \triangleright m_b)$$

Observe that

$$(c_1 \triangleright m_{c_1} ; a \triangleright m_a ; b \triangleright m_b ; c_2 \triangleright m_{c_2}) \leq_p \alpha'$$

holds, i.e. one possible behavior for α' is a message sequence starting with m_{c_1} and ending with m_{c_2}. β', on the other hand, cannot exhibit such a sequence. Here, any behavior either starts with m_a or ends with m_b. Obviously, we thus have

$$\alpha' \not\equiv_u \beta'$$

The reason for this discrepancy is that the equivalence relation $\equiv_u$ directly bases on the equality of the sets of (partial) system executions represented by the operands. The semantic model we have chosen for these executions treats message occurrences on channels as "atomic" entities. In the transition from α to α' this atomicity breaks, thus allowing the refinement of $c \triangleright m_c$ to split across an entire execution. In β' the prefix $a \triangleright m_a$ and the suffix $b \triangleright m_b$ limit the possible extent of $c_1 \triangleright m_{c_1}$ and $c_2 \triangleright m_{c_2}$ "to the left" and "to the right", respectively.

Viewed in this light, the discrepancy is not quite as surprising as it might at first have seemed. Replacing an atomic "action" with a sequence of actions usually offers more potential for interference.

In the context of "action refinement" the problem of the non-preservation of equivalences was studied extensively from the mid 1980s to the mid 1990s; see [AH93] and [vG96] for an overview and an extensive list of references. [vG96] compares numerous semantic models for concurrent systems and corresponding equivalence notions with respect to the preservation of equivalence under action refinement. The author shows that

1. no interleaving model has a corresponding equivalence notion that preserves action refinement, and

2. there are partial order models (also known as event structures) with corresponding equivalences that are preserved under action refinement; examples are
   - pomset (cf. [Pra86, vG96]) trace equivalence (for linear time partial orders), and
   - history preserving bisimulation (for branching time partial orders).

In fact, [vG96] shows for branching time partial orders that less fine-grained equivalences than history preserving bisimulation do *not* preserve equivalences under action refinement.

The approach followed by [vG96] to obtain a sufficiently fine-grained equivalence notion is to allow actions to have a duration, and to consider as system states (or configurations) the sets of already started, but not yet finished actions (together with their causal predecessors). The work of [AH93] goes in a similar direction: here, the authors split an action into two parts, the action's beginning, and its end. This allows them to obtain an equivalence preserving notion of action refinement for a subset of the process algebra CCS.

This leaves us with the question of how to deal with the problem in our semantic framework. We identify the following three possibilities:

1. we keep the current semantics basis, but strengthen our equivalence notion, or
2. we switch to a pomset or branching time partial order semantics with an equivalence-preserving message refinement notion, or
3. we introduce a concept for avoiding interferences such as the ones exhibited through the refined MSC α in the example above.

In the following paragraphs we discuss each of these possibilities in turn.

Strengthening the Equivalence Notion

As we have discussed above, the equivalence notion $\equiv_u$ we have introduced in Section 4.4 is too weak to distinguish the MSCs α and β in view of the potential interference introduced by the message refinement of $c \triangleright m_c$ into the sequence $(c_1 \triangleright m_{c_1} \; ; \; c_2 \triangleright m_{c_2})$.

A standard construction (cf. [Mil80, AH93]) to yield a stronger equivalence notion is to consider two MSCs equivalent if and only if they represent the same sets of executions, and all of their possible message refinements are equivalent as well. More precisely, we define the equivalence relation $\equiv_u^{mr} \subseteq \langle \text{MSC} \rangle \times \langle \text{MSC} \rangle$ as follows, for all $\gamma, \delta \in \langle \text{MSC} \rangle$:

$$\gamma \equiv_u^{mr} \delta \quad \stackrel{\text{def}}{=} \quad (\gamma \equiv_u \delta) \wedge \langle \forall \gamma', \delta' : \gamma' \leq_m \gamma \wedge \delta' \leq_m \delta : \gamma' \equiv_u \delta' \rangle$$

Clearly, we have $\alpha \equiv_u \beta$ but $\alpha \not\equiv_u^{mr} \beta$ in the example above.

By construction of $\equiv_u^{mr}$ we obtain that message refinement preserves $\equiv_u^{mr}$. However, because its definition involves a closure over all possible refinements of the operand MSCs,

we have no general, practical way of determining whether $\gamma \equiv_u^{mr} \delta$ holds for MSC terms γ and δ.

Switching the Semantics Basis

As [vG96] and [AH93] show, there are semantic domains with corresponding equivalence notions that are preserved under action, or – in our setting – message refinement.

Therefore, we could switch from our stream-based model to one of the appropriate partial order models. It is, for instance, not a difficult exercise to define a pomset semantics for MSC terms and to use pomset trace equivalence (cf. [vG96]) as the equivalence notion on these terms. We could even keep our stream-based model "for all other purposes" and resort to pomset trace equivalence only when it comes to defining MSC equivalence; the authors of [AH93] use this technique for the definition of their equivalence notion: they introduce a more fine-grained semantic model, define equivalence on it, and then map this equivalence to the original, coarser model.

Again, however, this additional effort would not yield a practicable procedure for determining whether two given MSCs are equivalent or not. In all but trivial cases we would have to resolve this question by semantic, not by syntactic considerations.

Avoiding Interference

The two MSCs α' and β' from our example above are not equivalent under $\equiv_u$ because refining $c \triangleright m_c$ into $(c_1 \triangleright m_{c_1} \; ; \; c_2 \triangleright m_{c_2})$ breaks the "atomicity" of the single message $c \triangleright m_c$; this results in the additional possibility for interference exhibited by α', as compared to α. Thus, another way of coping with the equivalence preservation problem is to avoid the additional interference by explicit syntactic and semantic means.

We could, for instance, introduce a new MSC term **protect** with $[\![\mathbf{protect}(\gamma)]\!]_u \stackrel{\mathsf{def}}{=} [\![\gamma]\!]_u$, redefine the semantics of the interleaving operator such that for all messages $ch \triangleright m$

$$ch \triangleright m \sim \mathbf{protect}(\gamma) \equiv_u (ch \triangleright m \; ; \mathbf{protect}(\gamma)) \mid (\mathbf{protect}(\gamma) \; ; \; ch \triangleright m)$$

holds, and modify the definition of $\leq_m$ by defining the substitution operator on **protect** terms as follows:

$$\mathbf{protect}(\gamma)[X, \tilde{ch} \triangleright \tilde{m} / ch \triangleright m] \stackrel{\mathsf{def}}{=} \mathbf{protect}(\gamma[X, \tilde{ch} \triangleright \tilde{m} / ch \triangleright m])$$

Then, if we refined message $c \triangleright m_c$ into $\mathbf{protect}(c_1 \triangleright m_{c_1} \; ; \; c_2 \triangleright m_{c_2})$ we could no longer distinguish the MSCs $\hat{\alpha}$ and $\hat{\beta}$, which we obtain from α and β as follows:

$$\hat{\alpha} = (a \triangleright m_a \; ; \; b \triangleright m_b) \sim \mathbf{protect}(c_1 \triangleright m_{c_1} \; ; \; c_2 \triangleright m_{c_2})$$

and

$$\begin{aligned} \hat{\beta} = \; & (a \triangleright m_a \; ; \; (b \triangleright m_b \sim \mathbf{protect}(c_1 \triangleright m_{c_1} \; ; \; c_2 \triangleright m_{c_2}))) \\ & \mid ((a \triangleright m_a \sim \mathbf{protect}(c_1 \triangleright m_{c_1} \; ; \; c_2 \triangleright m_{c_2})) \; ; \; b \triangleright m_b) \end{aligned}$$

protect keeps the "atomicity" of single messages, even for protocols refining them, intact.

We could then prove that the refinement of messages by protected protocols preserves the equivalence relation $\equiv_u$.

This approach makes the methodical challenges behind message refinement explicit. In the presence of any form of parallel composition we cannot, in general, hope to preserve equivalences if we break an individual message into an entire protocol. A very practical example is the removal of locking mechanisms around a database transaction: loss of coherence may be the undesirable result. Using **protect** in the refinement of a message corresponds to *introducing* a "locking mechanism" around the refining protocol to maintain the causality between the message to be refined and its context in the transition from the abstract to the more detailed MSC.

Conclusion

Each of the three approaches for defining an adequate equivalence relation that is invariant under message refinement for MSCs has both advantages and disadvantages. The first, based on the definition of $\equiv_u^{mr}$, induces no changes to the semantic model, but does not offer much insight on how we can determine the equivalence of two MSCs practically (or even syntactically). Switching to a semantics model equipped with an appropriate equivalence notion (such as pomset equivalence or history preserving bisimulation) has the advantage that determining the semantic equivalence of two MSCs is easier than with $\equiv_u^{mr}$. The disadvantage is that, still, there is no general syntactic way of determining MSC equivalence for practical purposes. Using the **protect**-construct to avoid additional interference through message refinement has the advantage of making the methodical difference between atomic messages and non-atomic protocols explicit; using it, the developer can decide whether or not a refinement should preserve equivalences. Its disadvantage is the addition of "yet another" operator to the MSC language.

The choice between these three alternatives depends on the goals we follow in our usage of MSCs. In a pragmatic application of MSCs we favor the third approach, because it allows the developer to make an "informed choice" about how to refine messages that may appear within operands of a parallel composition.

For the remainder of this thesis it shall suffice that we now know about the difficulties of message refinement; we will stick with the original semantics definition, and with the "weak" equivalence relation $\equiv_u$, as we have defined it in Section 4.4.

5.5. Structural Refinement

A typical step in the systematic development of distributed systems is to decompose a single component into several sub-components. This step reveals (part of) the internal structure of the component under consideration; therefore, we call this form of adding more detail

to a specification *structural refinement*. Before performing a structural refinement we view the component under consideration as a “black box”, and do not care about how this box internally handles the messages it receives. Afterwards the component has become a “glass box”, i.e. we have additional knowledge about its internals that we can refer to in our specifications. Other common names for this form of refinement are “glass box refinement” (cf. [Bro93, Bro99a]) and “object refinement” (cf. [DW98]).

MSCs have both a structural and a behavioral dimension; they display (logical) component distribution, as well as the communication of the depicted components over time. Therefore, in this section, we study a refinement notion for MSCs that reflects changes in the system structure.

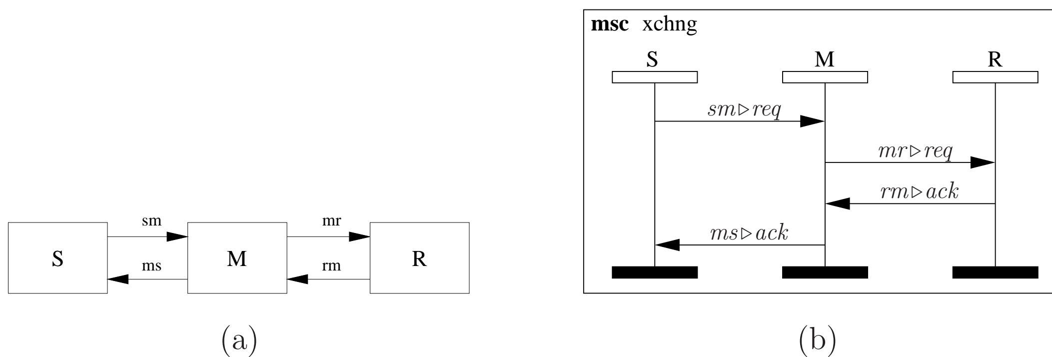


Figure 5.5.: System structure diagram and MSC before structural refinement

As an example, consider the structure of a simple communication system, consisting of a sender (*S*), a receiver (*R*), and a transmission medium (*M*) as depicted in Figure 5.5 (a). Figure 5.5 (b) displays a simple two-way communication between *S* and *R* via *M*. A possible structural refinement of M is to decouple the two communication paths from *S* to *R* and from *R* to *S* internally. Figure 5.6 shows a corresponding system structure diagram. Here we have redirected incoming and outgoing channels of *M* such that they now end and start at either *M1* or *M2*. We use the dashed box labeled *M* in Figure 5.6 to indicate that *M1* and *M2* are sub-components of *M*.

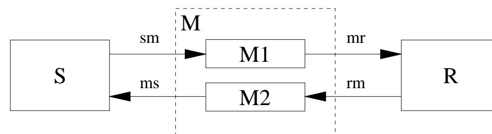


Figure 5.6.: System structure diagram after structural refinement

To obtain an MSC that reflects the structural refinement from Figure 5.5 (a) to Figure 5.6 we have to perform two modifications to the MSC of Figure 5.5 (b). First, we have to replace the axis for *M* with axes for *M1* and *M2*. Second, we have to “redirect” each message arriving at or emanating from *M* to either *M1* or *M2*. Figure 5.7 shows a possible result of these modifications.

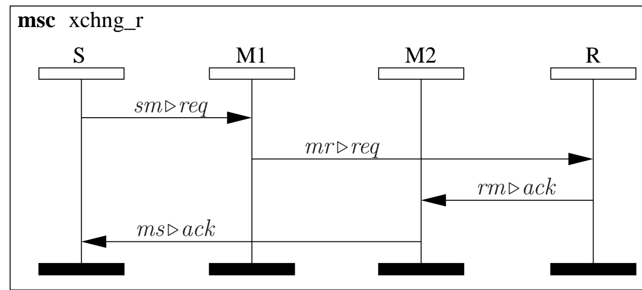


Figure 5.7.: MSC after structural refinement

The behavior represented by MSC *xchng_r* differs from the one of *xchng* only in the sources and destinations of certain messages; the ordering of the messages is identical.

So far, the decomposition we have shown in Figures 5.6 and 5.7 is simple in the sense that the components *M1* and *M2* cannot communicate; there is no channel between *M1* and *M2*. Often, however, we want the refining components to coordinate their behavior explicitly over "local" channels, i.e. channels connecting only refining components. Figure 5.8 shows an example decomposition of *M* into *M1* and *M2* where the local channel *mm* connects *M1* and *M2*.

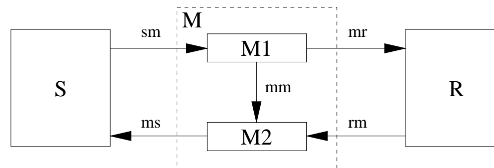


Figure 5.8.: System structure diagram with added local channel

MSC *xchng_r2* in Figure 5.9 displays the same behavior as MSC *xchng_r*, up to the message *notify* sent by *M1* to *M2* along channel *mm*.

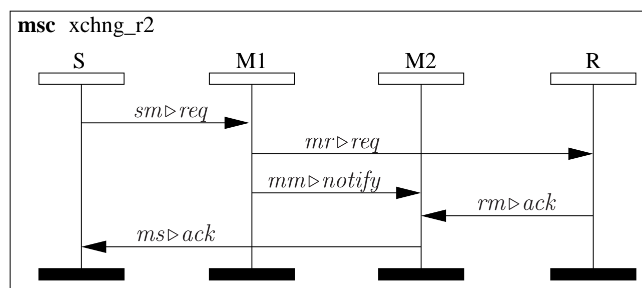


Figure 5.9.: MSC with message on local channel *mm*

This example already hints at a possible treatment of structural refinement in our setting: if we find an abstraction from communication on local channels, as well as a mapping

from the sub-components' channels to the channels of the abstract component such that the behavior represented by the MSC resulting from this mapping "equals" the one of the more abstract MSC, we have "witnessed" a structural MSC refinement.

Before we can turn this intuition into a precise definition we have to discuss another open question: can the decomposed component reappear in the decomposition? This question addresses a situation that is quite common in practical designs: a component decomposes into several others, but still participates in the interplay between its constituents.

As an example for a design where this question is important we note the "Mediator" pattern (cf. [GHJV95]), which describes a strategy for encapsulating and coordinating complex interactions among a set of (sub-)components in object-oriented designs. The parent object (the mediator) interacts with the environment and delegates incoming requests to its sub-components; the sub-components interact not among one another, but only via the mediator. The mediator thus coordinates the interactions among its constituents to achieve a desired result.

This example suggests to answer "yes" to the question we have raised above. However, allowing reappearance of the parent component induces a much more complicated definition for structural refinement; we cannot, in general, determine syntactically, whether an interaction between the parent component and its environment occurring in the refined MSC was already there in the non-refined version. Therefore, we require the composition to be strong in the sense that the parent component cannot reappear in the refined MSC. This restriction does not result in loss of generality, however. If we want to model structural refinement of a mediator object, for instance, we simply have to provide an additional sub-component in the refined MSC that takes over the coordination role of the "abstract" mediator.

We introduce structural MSC refinement in two steps. First, we make precise what a refinement on the system structure, or, more precisely, on the system model, is; to that end, we define the relation $\overset{p,\tilde{P}}{\underset{s}{\leadsto}}$ that captures the changes on the sets of components and channels, reflecting the decomposition of component p into the elements of set $\tilde{P}$. Second, based on relation $\overset{p,\tilde{P}}{\underset{s}{\leadsto}}$, we translate the semantics of an MSC in the structurally refined system model to one in the non-refined system model to enable the comparison of the two semantics.

We start with the definition of relation $\overset{p,\tilde{P}}{\underset{s}{\leadsto}}$. As we have discussed in detail, above, we want the set of components of the refined system to differ from the one of the abstract system such that the former includes the components in $\tilde{P}$, but does not contain component p. We also require the refined system to contain verbatim copies of all channels of the abstract system that neither start nor end at p; the abstract system's other channels appear in the refined system with either their source or destination component changed to one of $\tilde{P}$'s elements. Furthermore, we allow fresh "local" channels that connect only components from $\tilde{P}$ in the refined system.

Thus, for two given tuples $B = (P, C, M, S, \mathit{MSCR})$ and $B' = (P', C', M', S', \mathit{MSCR}')$, a component $p \in P$ of the abstract system, and a set of components $\tilde{P} \subseteq P'$ in the refined system, we define[1]

$$B \overset{p,\tilde{P}}{\underset{s}{\rightsquigarrow}} B' \tag{5.15}$$

$$\overset{\mathsf{def}}{=} \quad P' = (P \setminus \{p\}) \uplus \tilde{P} \tag{5.16}$$

$$\wedge \; \langle \forall (chn, src, dst) \in C' :: src \notin \tilde{P} \wedge dst \in \tilde{P} \Rightarrow (chn, src, p) \in C \rangle \tag{5.17}$$

$$\wedge \; \langle \forall (chn, src, dst) \in C' :: src \in \tilde{P} \wedge dst \notin \tilde{P} \Rightarrow (chn, p, dst) \in C \rangle \tag{5.18}$$

$$\wedge \; \langle \forall (chn, src, dst) \in C' :: src \notin \tilde{P} \wedge dst \notin \tilde{P} \Rightarrow (chn, src, dst) \in C \rangle \tag{5.19}$$

$$\wedge \; \langle \forall (chn, src, dst) \in C :: \langle \exists src', dst' :: (chn, src', dst') \in C' \rangle \rangle \tag{5.20}$$

The definition of $\overset{p,\tilde{P}}{\underset{s}{\rightsquigarrow}}$ captures the following properties:

1. the decomposition is proper, i.e. the decomposed component does not reappear in the decomposition, and none of the refining components appears in the abstract system (conjunct (5.16)),
2. every channel that ends at a refining component, and that does not also start at a refining component, maps to a corresponding channel with the same source, ending at the abstract component (conjunct (5.17)),
3. every channel that starts at a refining component, and that does not also end at a refining component, maps to a corresponding channel with the same destination, starting at the abstract component (conjunct (5.18)),
4. every channel that neither starts nor ends at either a refining or the abstract component appears unchanged in both the abstract and the refined system (conjunct (5.19)),
5. up to the possible change of source and destination components all channels of the abstract system reappear – with the same names – in the refined system (conjunct (5.20)).

As an example, consider the SSDs of Figure 5.5 (a) and 5.8. Clearly, we have

$$(P, C, M, S, \mathit{MSCR}) \overset{M,\{M1,M2\}}{\underset{s}{\rightsquigarrow}} (P', C', M', S', \mathit{MSCR}')$$

where Figure 5.5 (a) induces tuple $(P, C, M, S, \mathit{MSCR})$, and Figure 5.8 induces tuple $(P', C', M', S', \mathit{MSCR}')$.

[1] To simplify the formulae later in this section we implicitly assume $S' = S$, although modifying the set of components induces a modification of the overall state space.

The next step is to base structural MSC refinement on the definition of $\overset{p,\tilde{P}}{\underset{s}{\rightsquigarrow}}$. This allows us to relate MSCs whose semantics use different system models on different levels of structural abstraction.

To this end, we mimic the procedure we have carried out in the introductory example. We assume given two tuples $B = (P, C, M, S, MSCR)$ and $B' = (P', C', M', S', MSCR')$, a component $p \in P$, and a set of components $\tilde{P} \subseteq P'$, such that

$$B \overset{p,\tilde{P}}{\underset{s}{\rightsquigarrow}} B'$$

holds. To establish whether an MSC α', interpreted with respect to B', is a structural refinement of the MSC α, interpreted with respect to B, we proceed in three steps. First, we remove all local communication between the components in $\tilde{P}$ from the semantics of α'. Second, in executions according to α' we redirect all other communication – appearing on channels starting or ending at components in $\tilde{P}$ – to component p. Finally, we compare the resulting set of executions with the one represented by α.

1.) Removing "local" communication of the refining components

For a set $Y \subseteq C'$ we define the restriction of $[\![\alpha']\!]_u^{B'}$ to the set of channels in Y, written $[\![\alpha']\!]_u^{B'}|_Y$, by

$$\begin{aligned}
[\![\alpha']\!]_u^{B'}|_Y \overset{\text{def}}{=} \{(\varphi, t) \in (\tilde{Y} \times S)^\infty \times \mathbb{N}_\infty : \\
\langle \exists \psi : (\psi, t) \in [\![\alpha]\!]_u^{B'} : \\
\langle \forall t', ch : t' \in [u, t] \wedge ch \in Y : \quad & \pi_1(\psi).t'.ch = \pi_1(\varphi).t'.ch \\
& \wedge \pi_2(\psi).t' = \pi_2(\varphi).t' \rangle\rangle
\end{aligned}$$

We define the set of channels that are in C', but whose source or destination component is outside $\tilde{P}$ by

$$C'_{\tilde{P}} \overset{\text{def}}{=} \{(chn, src, dst) \in C' : src \notin \tilde{P} \vee dst \notin \tilde{P}\}$$

With $C'_{\tilde{P}}$ defined like this, $[\![\alpha']\!]_u^{B'}|_{C'_{\tilde{P}}}$ contains all behaviors represented by α, restricted to channels whose source or destination component is outside $\tilde{P}$.

2.) Redirecting communication involving the refining components

To capture the redirection of channel sources and destinations, we define function

$$relabel^{p,C}_{\tilde{P},C'} : C'_{\tilde{P}} \rightarrow C$$

by

$$relabel^{p,C}_{\tilde{P},C'}.(chn, src, dst) = \begin{cases} (chn, p, dst) & \text{if } src \in \tilde{P} \\ (chn, src, p) & \text{if } dst \in \tilde{P} \\ (chn, src, dst) & \text{if } src \notin \tilde{P} \wedge dst \notin \tilde{P} \end{cases}$$

We lift $relabel^{p,C}_{\tilde{P},C'}$ to sets of system executions as follows: for all $Z \subseteq \mathcal{P}((\tilde{C}' \times S)^{\infty} \times \mathbb{N}_{\infty})$ we define

$$\overline{relabel}^{p,C}_{\tilde{P},C'} : \mathcal{P}((\tilde{C}' \times S)^{\infty} \times \mathbb{N}_{\infty}) \rightarrow \mathcal{P}((\tilde{C} \times S)^{\infty} \times \mathbb{N}_{\infty})$$

by

$$\begin{aligned} \overline{relabel}^{p,C}_{\tilde{P},C'}.Z \overset{\text{def}}{=} & \{(\varphi, t) \in (\tilde{C} \times S)^{\infty} \times \mathbb{N}_{\infty} : \\ & \langle \exists \psi : (\psi, t) \in Z : \\ & \quad \langle \forall t', ch' : t' \in [0, t] \wedge ch' \in C' : \\ & \quad\quad \pi_1(\psi).t'.ch' = \pi_1(\varphi).t'.(relabel^{p,C}_{\tilde{P},C'}.ch') \wedge \pi_2(\psi).t' = \pi_2(\varphi).t' \rangle \rangle \end{aligned}$$

3.) Comparing the resulting set of executions with $[\![\alpha]\!]_0$

Intuitively, the aim of the preceding two steps was to "undo" the effects of a potential refinement to allow us to compare the set of executions represented by the refined MSC with the one of the abstract MSC. If the set of executions $\overline{relabel}^{p,C}_{\tilde{P},C'}.([\![\alpha']\!]^{B'}_0|_{C'_{\tilde{P}}})$ is a subset of the set of executions $[\![\alpha]\!]^B_0$ then we call α' a structural refinement of α. More precisely, we define:

Definition 7 (Structural Refinement) For MSCs $\alpha, \alpha' \in \langle \text{MSC} \rangle$, and the semantics bases $B' = (P', C', M', S', MSCR')$ and $B = (P, C, M, S, MSCR)$, component $p \in P$, and set $\tilde{P} \subseteq P'$, we define

$$\begin{aligned} \alpha' \leq^{Sys}_s \alpha & \overset{\text{def}}{=} B \overset{p,\tilde{P}}{\underset{s}{\rightsquigarrow}} B' \wedge \overline{relabel}^{p,C}_{\tilde{P},C'}.([\![\alpha']\!]^{B'}_0|_{C'_{\tilde{P}}}) \subseteq [\![\alpha]\!]^B_0 \\ \alpha' \leq^{\exists}_s \alpha & \overset{\text{def}}{=} \langle \exists B, B', p, \tilde{P} :: \alpha' \leq^{Sys}_s \alpha \rangle \\ \leq_s & \overset{\text{def}}{=} (\leq^{\exists}_s)^* \end{aligned}$$

□

$\alpha' \leq^{Sys}_s \alpha$ holds for a given component p, and a set of refining components $\tilde{P}$, if the semantics obtained from α' by "undoing" the effects of structural refinement is a subset of α's semantics. Thus, $\alpha' \leq^{Sys}_s \alpha$ captures a single structural refinement step with respect to a corresponding transformation of the underlying system model. Relation $\leq^{\exists}_s$ abstracts from concrete semantics bases B and B', components p, and component sets $\tilde{P}$. $\leq^{\exists}_s$ is not transitive: if we have $\alpha'' \leq^{\exists}_s \alpha'$ and $\alpha' \leq^{\exists}_s \alpha$, then we cannot, in general, exhibit a *single* component p and a *single* component set $\tilde{P}$ to derive $\alpha'' \leq^{\exists}_s \alpha$. Therefore, we use $\leq_s$ to obtain the reflexive, transitive closure of $\leq^{\exists}_m$.

5.5.1. Refinement Rule

Similar to $\leq_m$, we have defined $\leq_s$ in a very pragmatic, operational fashion. This allows us to formulate the following steps leading to a concrete structural refinement of an MSC α:

1. Select a component p occurring within α as the source or destination of a message; p is the component we want to replace with a set $\tilde{P}$ of fresh components.
2. Transform α into a fresh MSC α' such that α', p, and $\tilde{P}$ fulfill the requirements posed by the definition of relation $\overset{p;\tilde{P}}{\underset{s}{\rightsquigarrow}}$:
   (a) p does not reappear in α',
   (b) none of the refining components appears in α,
   (c) messages arriving at p in α arrive at an element of $\tilde{P}$ in α',
   (d) messages originating from p in α originate from an element of $\tilde{P}$ in α',
   (e) all messages that neither start nor end at p in α are copied verbatim to α',
   (f) up to the redirection of channels and additional messages on local channels connecting only components in $\tilde{P}$, the ordering and labeling of message arrows in the MSCs α' and α coincides.
3. Perform the structural refinement consistently in all MSCs within the MSC document.

The refinement must be "global" to allow consistent composition of MSCs within the document.

Clearly, after performing these steps, we obtain the validity of the relation $\alpha' \leq_s \alpha$; we can easily exhibit semantics bases B' and B such that $\alpha' \leq_s^{Sys} \alpha$ holds.

5.5.2. Relationship With MSC-96's Instance Decomposition

Recall from Section 2.2.3 that MSC-96 provides the "decomposed instance" construct, which is similar to our notion of structural refinement.

The major difference between decomposed instances and our approach is that neither inline expressions nor references may appear on the abstract component in MSC-96. Because we base our refinement notion on a "global" syntactic and semantic refinement of both the system structure and all MSCs within the same MSC document, we do not have to pose such restrictions.

5.6. Related Work

The work on MSC refinement in the literature concentrates mostly on structural refinement. Interworkings, for instance, have a formal notion for structural refinement (cf. [MR96]), which is similar to ours. MSC-96 contributes the notion of instance decomposition as an informal version thereof (cf. [IT96]).

The notion of property refinement we have introduced for MSCs is founded on set inclusion with respect to sets of entire system executions. Similar refinement notions for the behavior of individual components appear, for instance, in [Bro99a, Rum96, Sch98, Kle98]. In Chapter 7 we discuss the distinction between property refinement of MSCs and of individual components in more detail.

As we have explained above, the difficulties we encountered with message refinement are instances of similar problems in the context of action refinement. For a discussion of these problems, as well as for solution strategies, we refer the reader to [vG96, AH93] and the references contained therein. Our notion of message refinement was inspired by [Bro93, Bro99a], as well as by the notion of action refinement in [DW98].

5.7. Summary

In this chapter we have defined and discussed four refinement notions for MSCs: the binding of references, property refinement, message refinement, and structural refinement.

The binding of references accommodates the incremental development of MSC documents. Each new MSC definition of the form **msc** $X = \alpha$ makes the semantics of the other MSCs in the document that refer to X more precise.

A property refinement restricts the set of behaviors of an MSC; this corresponds to the removal of underspecification. We have presented several concrete transformations on MSCs resulting in refined versions with respect to property refinement. Examples are the removal of alternatives and the use of the join operator to add properties to an MSC specification. Moreover, we have discussed the compositionality of property refinement, which is valid with the exception of the trigger composition operator.

By our notion of message refinement we have formalized the substitution of an entire protocol for a single message. We have given an intuitive definition for this refinement concept by means of the referencing mechanism of our MSC dialect. The drawback of this intuitive definition is that – within our semantic model – message refinement does not preserve the equivalence relation $\equiv_u$. We have discussed this problem and possible solutions for it, in detail.

Structural refinement introduces the replacement of a single component with a set thereof (the set of the component's constituents) into our methodical framework. We have given

a very pragmatic definition for structural refinement, and have related it to the notion of "decomposed instance" in MSC-96.

The refinement notions, as well as the refinement rules we have introduced in this section enable the systematic use of MSCs in the development process. Each development step that bases on one of the refinement rules we have presented adds to the detail of a specification, while maintaining the "nett effect" or the "idea" behind the specification.

An interesting question is whether the rules for property refinement allow us to perform an efficient *syntactic* check whether one MSC is a refinement of another. If that was the case we could use MSCs effectively in the validation process. We consider this an important direction for future work.

CHAPTER 6

MSCs for Property-Oriented System Specifications

In this chapter we investigate four interpretations of MSCs with respect to a given system specification: the *existential*, *universal*, *exact*, and *negated* interpretation. Each of these is the basis for a different kind of methodic MSC usage in the development process, which ranges from using MSCs as scenarios (based on the existential interpretation) to MSCs as a complete specification technique (based on the exact interpretation). Moreover, we determine the classes of properties our MSC dialect allows us to express according to the classical distinction between *safety* and *liveness* properties.

Contents

6.1. Introduction **188**
6.2. MSC Interpretations **193**
6.3. Property Specification with MSCs: Safety and Liveness . . . **202**
6.4. Related Work **211**
6.5. Summary **212**

6.1. Introduction

In the preceding two chapters we have considered MSCs more or less in isolation; the semantics chapter (cf. Chapter 4) was entirely devoted to individual MSCs, whereas the chapter on MSC refinement (cf. Chapter 5) took corresponding refinements of the system structure into account, and has thus broadened our view beyond MSCs as such.

Now we turn our attention to a question that is at the heart of a methodical application of MSCs within the overall development process:

What properties does an MSC express?

By answering this question we clarify what constraints an MSC imposes on the system under development.

Because MSCs address multiple aspects of a system specification in the range from structure to behavior (cf. Chapter 1), the question we face has multiple facets:

- how complete is the information contained in an MSC, i.e.
  - can there be other components in the system besides the ones depicted in the MSC?
  - may a component involve in other interactions with components within or outside the scope of the MSC?
- what triggers the occurrence of the interaction sequence depicted in an MSC?
- must or may the depicted interaction sequences occur in all or any executions of the system under development?
- how to transit from possible to mandatory behavior?

Clearly, this list is neither exhaustive, nor orthogonal. However, these questions illustrate the spectrum of possible MSC interpretations within the development process. The answers we can give strongly depend on the intention we follow in using MSCs as a property-oriented specification technique.

If we consider *analysis* (capturing of functional and nonfunctional requirements), *specification* (constructing a model that fulfills the requirements captured during analysis), *design* (tailoring the model resulting from specification to fit a specific target architecture), and *implementation* (coding the design in a set of target programming languages to obtain an executable product system) as the phases of an iterative development process as we did in Chapter 1, we quickly see the relevance of the questions we have phrased above within and across all development phases.

A popular example of specifying system properties by means of MSCs is to illustrate *scenarios*, i.e. partial system executions capturing a segment of one of possibly many execution alternatives of the system under consideration. Across all development phases we can use scenarios to illustrate key interaction patterns that we want (or do *not* want), or observe the system components to exhibit. During analysis, scenarios often serve as coarse sketches of the principle idea behind the collaboration of a set of components. During specification and design we might use scenarios at a level of greater detail as a precise documentation of certain relevant interaction patterns. During and after implementation more or less detailed scenarios often serve the purpose of knowledge-management (by means of re-documenting interaction patterns within the "running" system), tracing (as illustrations of simulation or real system runs), and even reengineering (by feeding back traces into the analysis phase).

In each of these uses of scenarios we treat the corresponding MSCs – on the respective level of detail – as a very liberal statement about the system under consideration; each of the depicted components might also involve in arbitrary other interactions, as long as the ones depicted in the MSCs at least have the potential for occurring.

However, if we were to focus on MSCs as a visualization of scenarios only, we would exploit only little of the methodical potential of MSCs within the development process.

We might also be interested, for instance, in illustrating inevitable interaction patterns, in contrast to scenarios, which may or may not occur. As an example, we might want to specify the following property: in a communication protocol data eventually gets transmitted between two parties. We cannot specify this by means of scenarios alone; only if we adopt the view that a set of scenarios describes a certain system behavior *completely*, i.e. leaves no options for arbitrary other behavior, can we describe such eventuality properties by means of MSCs.

We can even go a step further, and require a given (set of) MSC(s) to describe the system's behavior *exactly*, from the beginning of an execution on. As a result, we consider MSCs as a description technique for complete system or component behavior, as an alternative for complete automaton specifications in the sense we have described in Chapter 3. This idea is particularly appealing if we use MSCs to collect interaction requirements, say, as a set of scenarios during analysis, and want to determine how the individual components must operate to establish the interactions contained in the MSCs, as part of the construction of early prototypes during specification and design, or even to prepare the implementation phase.

As an example, recall the Abracadabra-protocol we have specified by means of MSCs in Section 4.7. Figure 6.1 (a) shows another specification for part of the protocol, this time in a state-oriented fashion by means of a Mealy automaton variant (cf. Section 3.3.1) for one of the participating components. In Chapter 7 we give a formal semantics for Figure 6.1 (a) in terms of our system model; here, we appeal to the reader's intuitive understanding to observe that the automaton displays those behaviors we have identified

for the Abracadabra-protocol – from the viewpoint of component X – such that X is the only initiator of a successful communication.

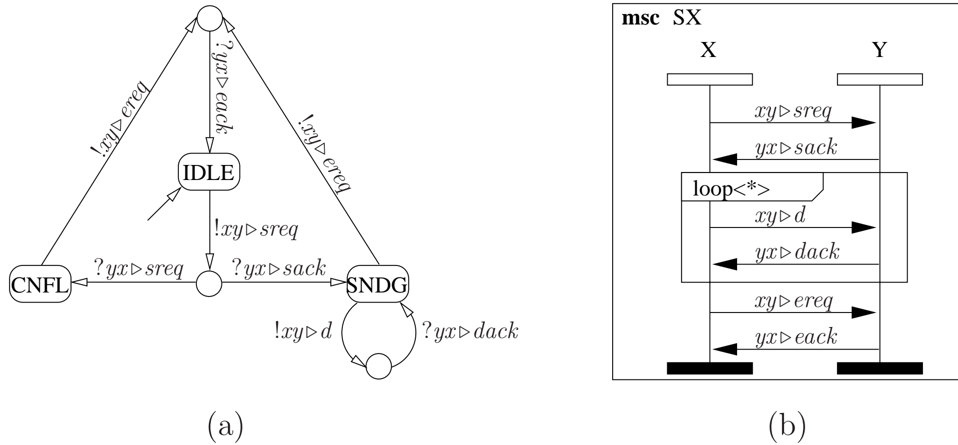


Figure 6.1.: Automaton and MSC for part of Abracadabra

MSC *SX* of Figure 6.1 (b), which we have copied verbatim from Section 4.7.3, shows only a subset of the automaton's behaviors: it displays a successful transmission, initiated by X. Conflict and conflict resolution are missing in *SX*. Having both the automaton and the MSC as part of an overall system specification raises the question whether the system, as specified through the automaton, *can* or *must* display the behavior represented by the MSC. Put another way, we can ask whether the MSC represents optional (possible) or mandatory (necessary) system behavior. In view of the MSC A from Section 4.7.2 (cf. Figure 4.20), which also captures all behaviors of the protocol, we could even require the system behaviors to match the ones of A exactly.

Motivated by these considerations we make precise and discuss the following MSC interpretations in this chapter:

Existential MSC Interpretation: As we have already mentioned above, the most prominent usage of MSCs to date in the development process is to consider them as "scenarios", i.e. as a representation of partial system behavior that may occur during the system's execution. There may also be executions of the system differing completely from the MSC under consideration. An MSC under existential interpretation expresses only that the system may not prohibit occurrence of the behavior represented by the MSC in *all* executions.

As an example, we can regard MSC *SX* from Figure 6.1 (b) as a scenario with respect to the behaviors represented by the automaton in Figure 6.1 (a), and obtain the existence of system executions where X initiates a successful transmission. We can also interpret each of the "use cases" C and CR (cf. Figure 4.22) of the Abracadabra-protocol existentially with respect to the automaton for the complete protocol specification.

Universal MSC Interpretation: Alternatively, we can require the behaviors represented by an MSC to occur in *all* executions of the system. Put another way, there may not be any system executions in which the behavior represented by the MSC is absent. In this sense, universal interpretation is "stronger" than existential interpretation. Still, universal interpretation allows arbitrary other behavior before, during, and after the interaction sequence specified via the MSC occurs.

We could, for instance, interpret MSC SX universally with respect to the automaton, and obtain that in every execution of the system X must initiate a successful transmission. Interpreting MSC A from Figure 4.20, which captures all use cases of the ABRACADABRA-protocol, universally with respect to another given system specification, yields only those executions that – besides other interactions in which the components may partake – fulfill the ABRACADABRA-protocol.

Exact MSC Interpretation: Strengthening the meaning of universal interpretation further to explicitly prohibit other behaviors than the ones specified through the MSC under consideration, we obtain the notion of "exact" interpretation. This MSC interpretation holds only for system executions displaying precisely the behavior specified by the MSC under consideration with nothing else in between.

If, as an example, we apply the exact interpretation to the MSC A from Figure 4.20 (with the behavior of all references occurring in A bound to the MSCs defined in Sections 4.7.3 and 4.7.4), we obtain system executions containing nothing else than communication between the components X and Y according to the ABRACADABRA-protocol.

We use the exact interpretation as the basis for the translation of MSCs into individual (and complete) component behavior in Chapter 7.

Negation: Unwanted Behaviors: So far, we have considered MSCs as a description of what may or what should happen only. Clearly, we can use MSCs also to describe what should *not* happen. This leads to the interpretation of MSCs as unwanted behaviors or "negative scenarios".

As an example, consider the MSCs AC and C in Figure 6.2 (a) and (b), respectively. Together, they represent an infinite repetition of conflicts between X and Y. If we interpret the infinite repetition of MSC C as a negative scenario with respect to the automaton from Figure 6.1 (a), we obtain only those system executions with a finite number of conflicts.

The extreme choices of treating MSCs as either exemplary execution segments (scenarios) or as complete behavior specifications, span the range of possible interpretations of entire MSCs with respect to the system under development. Making precise which of the possible choices we adopt for any particular MSC fixes – on a rather large scale – this MSC's role as an artifact within the development process, and thus fixes a property we want the system to have.

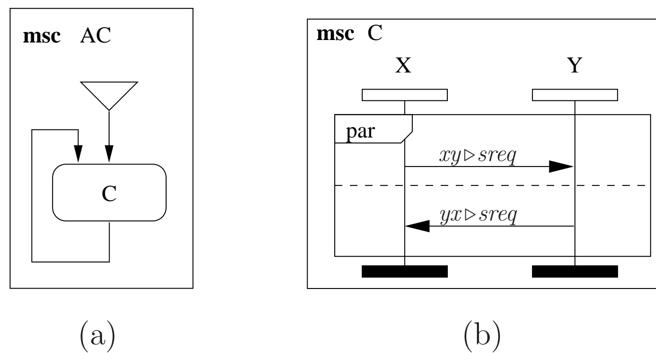


Figure 6.2.: Negative scenario: infinite repetition of conflicts

Safety and Liveness Properties Interestingly, we face similar choices of interpretation, if we approach the properties addressed by MSCs on a much smaller scale. We can ask, for instance, whether the messages contained in an MSC must occur within finite time or can have an infinite delay. An answer to this question induces responsibilities for all components occurring in the MSC. More generally, we are interested in what constraints each individual arrow in an MSC imposes on the system as a whole, as well as on each individual component – according to the semantics definition of Chapter 4.

The author of [Lam77] has introduced the distinction between *safety* and *liveness* properties, which we use to discuss this aspect of the interpretation of MSCs. A safety property states that "nothing bad" ever happens during system execution, while a liveness property states that "something (good)" does eventually happen. Having these terms available, we can ask whether MSCs express safety or liveness properties with respect to the system under consideration: must the messages depicted within an MSC occur within finite time? Can they occur out of order? Can other than the depicted messages occur?

These questions complement the larger-scale interpretation of MSCs in the sense that the smaller-scale interpretations make precise the properties an MSC specifies for any concrete execution, whereas the larger-scale interpretations determine whether a system must or may (not) have executions fulfilling these properties.

As a result of this chapter we will see that the MSC dialect of Chapter 4 specifies essentially liveness properties. This corresponds with the typical intuition responsible for the popularity of MSCs: an MSC describes a sequence of interactions, where all the depicted messages do indeed appear in the specified order. As a methodical consequence we obtain that our MSC dialect nicely complements specification techniques focusing on safety properties.

The remainder of this chapter has the following structure. In Section 6.2 we discuss the existential, universal, exact, and negated MSC interpretations. For each of these we give a formal definition based on our system model, and discuss methodical implications of using them. In particular, we discuss the relationship between each interpretation and system refinement. Our analysis of the safety and liveness properties represented by an

MSC appears in Section 6.3. In Section 6.4 we mention related work; this includes a brief comparison of our approach with the ones of [Kle98] and [DH99], which also distinguish between existential and universal MSC interpretations. Section 6.5 contains a summary of this chapter.

6.2. MSC Interpretations

In this section we explore several MSC interpretations, and discuss their methodical implications. Recall from the presentation of our system model in Section 4.2 that we consider subsets $Spec \subseteq (\tilde{C} \times S)^\infty$ as system specifications. $Spec$ contains all possible executions of the system under consideration. Recall further that, given a time $u \in \mathbb{N}_\infty$, we have defined the semantics of an MSC α to be the set of all pairs of the form (φ, t), where $\varphi \in (\tilde{C} \times S)^\infty$ and $t \in \mathbb{N}_\infty$, such that α represents φ during the time interval $[u, t]$. Thus, relating an MSC α with a given system specification $Spec$ amounts to comparing the sets of executions represented by α with those in $Spec$.

For each of the MSC interpretations introduced below we define a relation $\vdash: \mathcal{P}((\tilde{C} \times S)^\infty) \times \langle \mathrm{MSC} \rangle$; this relation makes precise under what circumstances a system specification $Spec$ fulfills an MSC according to the respective interpretation. Observe that we do not say how we have obtained $Spec$ in the first place. As we have sketched in the introduction, $Spec$ may, for instance, be the result of a complete system specification through an automaton; it may also result from any other form of behavior specification. Moreover, in this section we focus on *defining* the various MSC interpretations to make their intuitive meanings precise and accessible to methodical application. *Checking* whether a system specification actually fulfills an MSC is a matter of validation, and is beyond the scope of this thesis.

One of the distinguishing characteristics of the interpretations we study is their stability under property refinement of a given system specification. We carry over the definition we gave in Section 5.3 for property refinement of MSCs to the refinement of system specifications as follows:

Definition 8 (Specification Property Refinement) For all sets $Spec_0, Spec_1 \subseteq (\tilde{C} \times S)^\infty$ we define the relation $\leq_p : \mathcal{P}((\tilde{C} \times S)^\infty) \times \mathcal{P}((\tilde{C} \times S)^\infty)$ by

$$Spec_0 \;\leq_p\; Spec_1 \;\;\equiv\;\; Spec_0 \subseteq Spec_1$$

and say "$Spec_0$ refines $Spec_1$ (with respect to property refinement)" or "$Spec_0$ is a property refinement of $Spec_1$, if $Spec_0 \;\leq_p\; Spec_1$ holds. □

Specification property refinement inherits reflexivity, transitivity, and antisymmetry from set inclusion.

With these preliminaries in place, we can now present the definitions of the existential (cf. Section 6.2.1), universal (cf. Section 6.2.2), exact (cf. Section 6.2.3), and negated (cf. Section 6.2.4) MSC interpretations, in turn.

6.2.1. Existential MSC Interpretation

MSCs and similar description techniques have gained significant popularity over the past decade as a means for illustrating *scenarios* of component interaction in distributed systems.

As an example, consider the specification of a telecommunication switching system, which typically covers several thousand services (such as starting up, processing calls, billing, etc.). Each of these services may result in several thousand lines of executable code. By means of scenarios we can illustrate key patterns of the communication behavior of the components constituting the switch, without having to present all the nitty gritty details of what all else happens before, during, and after the time interval covered by the scenario.

In this section we give a formal definition for the term "scenario", and discuss its methodical usage.

Typical informal definitions for the term in the literature are "a sequence of actions that illustrates a behavior" (cf. [RJB99]), and "a particular trace of action occurrences starting from a known initial state" (cf. [DW98]). Some authors view scenarios as instances of use cases, and informally define the term "use case" as "the specification of sequences of actions, including variant sequences and error sequences, that a system, subsystem, or class can perform by interacting with outside actors" (cf. [RJB99]), or as "a sequence of actions a system performs that yields an observable result of value to a particular actor" (cf. [Kru99a]).

In our view, a scenario is a *partial*, possible behavior of the system under specification. A scenario describes the system's behavior from a certain point in time on, and under certain circumstances that have enabled occurrence of the scenario. Formally, we define the notions of existential interpretation and scenarios as follows:

Definition 9 (Existential Interpretation, Scenario) For all $\mathit{Spec} \subseteq (\tilde{C} \times S)^\infty$, and $\alpha \in \langle \text{MSC} \rangle$ we define

$$\mathit{Spec} \vdash_\exists \alpha \stackrel{\text{def}}{=} \langle \exists \psi \in \mathit{Spec}, u \in \mathbb{N}, (\varphi, t) \in [\![\alpha]\!]_u :: \psi|_{[u,t]} = \varphi|_{[u,t]} \rangle$$

and say that "*Spec* fulfills α under existential interpretation" or "α represents a scenario of the system specification *Spec*, if $\mathit{Spec} \vdash_\exists \alpha$ holds. □

According to this definition, α represents a scenario with respect to *Spec*, if α represents at least one of the possible system behaviors according to *Spec* from a certain point in time onward.

Our definition is liberal in the sense that we allow the use of alternatives, repetition, and referencing in the MSCs we interpret as scenarios. Most of the traditional approaches, which view scenarios as instances of use cases, require scenarios to represent a single, finite sub-sequence of the system behavior. This restricted view quickly leads to a proliferation

of scenarios that capture, say, different numbers of repetitions of a loop body, but are identical otherwise. Therefore, we have chosen our liberal scenario definition and, in a sense, identify the notions of scenario and use case in our treatment.

Because scenarios describe only potential behavior, and do not constrain *all* behaviors of a system specification, the existential interpretation is not monotonic with respect to specification property refinement, as the following proposition demonstrates:

Proposition 6 ($\leq_p$ does not preserve $\vdash_\exists$) *There are* $Spec_0, Spec_1 \subseteq (\tilde{C} \times S)^\infty$ *and* $\alpha \in \langle MSC \rangle$ *with*

$$Spec_1 \vdash_\exists \alpha \wedge Spec_0 \leq_p Spec_1 \wedge Spec_0 \not\vdash_\exists \alpha$$

□

Proof Consider $Spec_0 \stackrel{\mathsf{def}}{=} \{(\{c \mapsto m_c\}, s)^\infty\}$ and $Spec_1 \stackrel{\mathsf{def}}{=} Spec_0 \cup \{(\{b \mapsto m_b\}, s)^\infty\}$ for some $s \subseteq S$, channels c, b, and the corresponding messages m_c, and m_b. Clearly, we have $Spec_0 \leq_p Spec_1$ and $Spec_1 \vdash_\exists b \triangleright m_b$, but $Spec_0 \not\vdash_\exists b \triangleright m_b$. ■

The methodical consequence of this proposition is that, during systematic development based on specification property refinement, the developer must revalidate all scenarios against the specification after each refinement step. Only development steps *coarsening* a specification, i.e. leading from a specification $Spec_0$ to another specification $Spec_1$ with $Spec_0 \leq_p Spec_1$, preserve an existential interpretation.

This sensitivity to specification property refinement of the existential interpretation is not a problem if the purpose of MSC usage is to illustrate or document the system at a certain stage of the development process. On the contrary, we can view the documentation of complex interaction patterns by means of scenarios in already existing systems as a means of knowledge management, and as an important step in the reengineering process; this step prepares the transition from one system version to the next.

Moreover, if we start our development process by collecting interaction requirements in the form of scenarios, then we can try to *construct* a system fulfilling these scenarios. This drives the methodical character of scenarios from a mere means of documentation or illustration towards becoming a specification technique for complete system behavior, and constructively avoids the problem of the existential interpretation's sensitivity to specification property refinement.

In the remainder of this chapter, as well as in Chapter 7 we discuss the step from scenarios to complete behavior specifications in detail. One of the prerequisites for this step is a sufficiently expressive MSC notation, such as the one from Chapter 4, allowing us to compose both orthogonal scenarios (by means of sequential and alternative composition, interleaving, repetition, preemption, or trigger composition), and overlapping scenarios (by means of join). The other important prerequisite is the existence of the refinement notions we have established in Chapter 5; they allow us to tune the level of detail we want the constructed specification to have already while working on the MSCs.

6.2.2. Universal MSC Interpretation

Above, we have identified the documentation and illustration of key interaction patterns as the predominant application area for scenarios. If the system specification is not (any more) subject to change then every existentially interpreted MSC remains a scenario of this specification.

If we place MSCs in the center of a systematic development process where a scenario, which we have identified as such, will remain a scenario of the system across the entire development process, we have to come up with a stronger MSC interpretation.

The universal MSC interpretation we define in this section designates the interaction patterns represented by an MSC as *mandatory* behavior, as opposed to the *optional* interpretation underlying scenarios. Formally, we define

Definition 10 (Universal Interpretation) For all $Spec \subseteq (\tilde{C} \times S)^\infty$, and $\alpha \in \langle \text{MSC} \rangle$ we define

$$Spec \ \vdash_\forall \ \alpha \ \stackrel{\text{def}}{=} \ \langle \forall \psi \in Spec :: \langle \exists u \in \mathbb{N}, (\varphi, t) \in [\![\alpha]\!]_u :: \psi|_{[u,t]} = \varphi|_{[u,t]} \rangle \rangle$$

and say that "*Spec* fulfills α under universal interpretation", if $Spec \ \vdash_\forall \ \alpha$ holds. □

According to this definition, *every* behavior of a specification that fulfills an MSC under universal interpretation must, from a certain (finite) point in time onward, equal one of the behaviors represented by the MSC. Because we have defined specification property refinement based on set inclusion, we immediately see that the universal MSC interpretation is stable under property refinement:

Proposition 7 (Stability of the Universal Interpretation) *For all system specifications $Spec_0, Spec_1 \subseteq (\tilde{C} \times S)^\infty$ and MSCs $\alpha \in \langle MSC \rangle$ we have*

$$Spec_1 \ \vdash_\forall \ \alpha \ \wedge \ Spec_0 \leq_p Spec_1 \ \Rightarrow \ Spec_0 \ \vdash_\forall \ \alpha$$
□

PROOF For all $Spec_0, Spec_1 \subseteq (\tilde{C} \times S)^\infty$ and $\alpha \in \langle \text{MSC} \rangle$ we observe

$Spec_1 \ \vdash_\forall \ \alpha \ \wedge \ Spec_0 \leq_p Spec_1$

$\equiv$ (∗ definitions of $\vdash_\forall$ and $\leq_p$ ∗)

$\langle \forall \psi \in Spec_1 :: \langle \exists u \in \mathbb{N}, (\varphi, t) \in [\![\alpha]\!]_u :: \psi|_{[u,t]} = \varphi|_{[u,t]} \rangle \rangle \ \wedge \ Spec_0 \subseteq Spec_1$

$\Rightarrow$ (∗ predicate calculus ∗)

$\langle \forall \psi \in Spec_0 :: \langle \exists u \in \mathbb{N}, (\varphi, t) \in [\![\alpha]\!]_u :: \psi|_{[u,t]} = \varphi|_{[u,t]} \rangle \rangle$

$\equiv$ (∗ definition of $\vdash_\forall$ ∗)

$Spec_0 \ \vdash_\forall \ \alpha$

The universal interpretation lets us use MSCs to express eventuality properties: one of the behaviors represented by the MSC does eventually occur in every execution of the system. In Section 6.3 we examine the properties we can express with MSCs in more detail.

Recall that $[\![\alpha]\!]_u$ is the equivalence class of *all* behaviors "complying" to α over some time interval. Therefore, if $Spec \vdash_\forall \alpha$ holds, then arbitrary other behavior may occur before, during, and after the time interval covered by α in each of *Spec*'s elements. In the next section, we strengthen the definition of universal interpretation further to rule out this "background noise".

6.2.3. Exact MSC Interpretation

The MSC interpretations we have introduced so far target the combination of MSCs with (other) given system specifications. For every MSC α and $u \in \mathbb{N}_\infty$ the semantics $[\![\alpha]\!]_u$ contains *all* system behaviors exhibiting the interaction sequence specified by α from time u on (cf. Section 4.5.2). This ensures that in both the existential and the universal interpretation the MSC under consideration requires neither more nor less than that the specified behavior occurs; in particular, it does not prohibit other behavior before, during, and after the time interval covered by the MSC, as long as the other behavior does not "contradict" the one represented by the MSC. This flexibility is necessary because we do not want an MSC to constrain the specification it complements more than necessary. It allows us to relate MSCs with arbitrary other system specifications of which we know nothing more than the sets of components, channels, and messages they refer to.

If, instead, we use MSCs for a complete system specification, i.e. if we want to *construct* a specification from a set of MSCs, such that the specification captures precisely the interaction patterns described by the MSCs – nothing more or less – we must use a different MSC interpretation. Intuitively, for a given $\alpha \in \langle \text{MSC} \rangle$ we start with the semantics $[\![\alpha]\!]_u$ and eliminate from it every execution where anything else than what is specified explicitly in α occurs. Formally, we define

Definition 11 (Closed World Semantics) For all $\alpha \in \langle \text{MSC} \rangle$, $u \in \mathbb{N}_\infty$, and $(\varphi, t) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty$ we define the semantics function $[\![.]\!]_{.,CW} : \langle \text{MSC} \rangle \times \mathbb{N}_\infty \to \mathcal{P}((\tilde{C} \times S)^\infty \times \mathbb{N}_\infty)$ by

$$\begin{aligned}(\varphi, t) \in [\![\alpha]\!]_{u,CW} \equiv \quad & (\varphi, t) \in [\![\alpha]\!]_u \\ & \wedge \langle \forall t' \in [u, t], \psi \in (\tilde{C} \times S)^\infty : \psi \neq \varphi : \\ & \quad \pi_1(\psi).t' \subseteq \pi_1(\varphi).t' \Rightarrow (\psi, t) \notin [\![\alpha]\!]_u \rangle\end{aligned}$$

and call $[\![\alpha]\!]_{u,CW}$ the "closed world semantics" of α with respect to u. By $\equiv_u^{CW}$ and $\leq_p^{CW}$ we denote the equivalence and MSC property refinement relations we derive from $\equiv_u$ and $\leq_p$, respectively, by substituting $[\![\alpha]\!]_{u,CW}$ for $[\![\alpha]\!]_u$ in the original definitions (cf. Sections 4.4 and 5.3). □

The elements of the closed world semantics of an MSC α must explicitly display the interaction behavior represented by α (according to the first conjunct of Definition 11); moreover, they do not display any other interaction behavior during the time interval covered by α (according to the second conjunct of Definition 11). The following proposition illustrates this by stating that a message that does not occur explicitly in an MSC term α does not occur at any time in the closed world semantics of α:

Proposition 8 *For all $ch \triangleright m \in \langle MSG \rangle$, and $\alpha \in \langle MSC \rangle$ we have*

$$ch \triangleright m \notin msgs.\alpha \;\Rightarrow\; \langle \forall (\varphi, t) \in [\![\alpha]\!]_{u,CW}, t' \in [u, t] :: m \notin \pi_1(\varphi).t'.ch \rangle \qquad \square$$

PROOF See Appendix B.3. ■

By means of the following corollary we can round up our treatment of property refinement for the preemption construct we have started in Section 5.3. There, we could not formulate a syntactic criterion for removing a preemption construct from an MSC term; all we could do was to give a very strong semantic criterion that prevented the preemptive message from occurring. The refinement rule we gave read as follows:

$$\lceil \alpha \rceil^{\overline{M} \setminus \{ch \triangleright m\}} \;\leq_p\; \alpha \stackrel{ch \triangleright m}{\rightarrow} \beta$$

where $\overline{M} \stackrel{\mathsf{def}}{=} \{ch \triangleright m : ch \in C \wedge m \in M\}$. Recall that the semantics of the message closure operator $\lceil . \rceil^{\cdot}$ ensures that in the refined MSC $\lceil \alpha \rceil^{\overline{M} \setminus \{ch \triangleright m\}}$ the message $ch \triangleright m$ does not occur during the time interval covered by α.

Under the closed world semantics, however, we can determine syntactically whether or not a message can occur in the behavior represented by an MSC. Therefore, we have the following corollary of Proposition 8:

Corollary 1 (Syntactic Criterion for Preemption Refinement) *For all messages $ch \triangleright m \in \langle MSG \rangle$, and MSCs $\alpha \in \langle MSC \rangle$ we have*

$$(ch \triangleright m \notin msgs.\alpha) \;\Rightarrow\; (\alpha \equiv_u^{CW} \lceil \alpha \rceil^{\overline{M} \setminus \{ch \triangleright m\}})$$

and

$$(ch \triangleright m \notin msgs.\alpha) \;\Rightarrow\; (\alpha \leq_p^{CW} \alpha \stackrel{ch \triangleright m}{\rightarrow} \beta) \qquad \square$$

Still, we have to handle the refinement of preemption with care. Consider the MSC term

$$(\alpha \stackrel{ch \triangleright m}{\rightarrow} \beta) \;\sim\; ch \triangleright m$$

and let $ch \triangleright m \notin msgs.\alpha$ hold. Clearly, we have $\alpha \leq_p^{CW} (\alpha \stackrel{ch \triangleright m}{\rightarrow} \beta)$, but

$$(\alpha \;\sim\; ch \triangleright m) \leq_p^{CW} ((\alpha \stackrel{ch \triangleright m}{\rightarrow} \beta) \;\sim\; ch \triangleright m)$$

does *not* hold, i.e. this (syntactic) form of preemption refinement destroys the compositionality of property refinement (cf. Section 5.3.2). We mention two ways for dealing with this problem. The first is to check the validity of $ch \triangleright m \notin msgs.\gamma$ for all MSC terms γ in the MSC documents in which α or a reference to α appears. The second is to protect the preemption construct by means of the **protect** operator we have introduced in Section 5.4 in connection with message refinement. This prohibits the occurrence of $ch \triangleright m$ during the time period covered by α in the semantics of $\textbf{protect}(\alpha \stackrel{ch \triangleright m}{\rightarrow} \beta) \sim ch \triangleright m$; then, the suggested refinement is valid and compositional.

Now we use the closed world semantics to define the exact MSC interpretation.

Definition 12 (Exact Interpretation) For all $Spec \subseteq (\tilde{C} \times S)^{\infty}$, and $\alpha \in \langle \text{MSC} \rangle$ we define

$$Spec \vdash_X \alpha \stackrel{\text{def}}{=} \psi \in Spec \equiv \langle \exists t \in \mathbb{N}_{\infty} :: \quad (\psi, t) \in [\![\alpha]\!]_{0,CW} \\ \wedge \langle \forall t' > t, ch \in C :: \pi_1(\psi).t'.ch = \texttt{<>} \rangle \rangle$$

and say that "*Spec* fulfills α under exact interpretation", if $Spec \vdash_X \alpha$ holds. □

This definition characterizes the exact interpretation as the strongest universal interpretation. Moreover, we have

$$\vdash_X \; \subseteq \; \vdash_\forall \; \subseteq \; \vdash_\exists$$

which we derive easily by inspection of the three interpretations' definitions. If an MSC α represents finite behavior, i.e. we have $(\psi, t) \in [\![\alpha]\!]_{0,CW} \Rightarrow t \in \mathbb{N}$ for all $\psi \in (\tilde{C} \times S)^{\infty}$, then the exact interpretation requires "silence" on all channels after the interactions as specified in α have occurred. We could relax the exact interpretation by dropping the second conjunct of the definition, above. However, the stronger version we have chosen here matches best our intention of having an interpretation that leaves *no* room for other behavior than the one explicitly contained in the MSC under consideration.

Under the closed world semantics and the exact interpretation an MSC gives us complete information about what happens among the depicted components. If, for instance, we consider the closed world semantics of the MSC *SX* (cf. Figure 6.1 (b)) we find that it contains only executions where after the sending of message *sreq* by X there is no other communication between X and Y until the arrival of message *sack* from Y at X. In other words, $yx \triangleright sack$ is the *immediate response* of Y to X's $xy \triangleright sreq$ message.

As we have mentioned above, the exact interpretation is of particular interest in the derivation of complete system behavior from a set of MSCs; it allows us to clearly separate the "absolutely necessary" behavior from possible "background noise". Therefore, this interpretation forms the basis for our transformation from MSCs to individual component specifications in Chapter 7.

6.2.4. Negation: Unwanted Behaviors

The existential, universal, and exact MSC interpretations allow us to express behaviors we want the system under consideration to have. The exact interpretation even goes beyond both the existential and the universal interpretation in that it not only requires what *can* happen (the behavior explicitly represented by the MSC); it also makes precise what *must not* happen (everything else).

By putting our focus on the second aspect of the exact interpretation, i.e. on specifying what must not happen, we obtain another possibility for interpreting an MSC with respect to a system specification: MSCs as unwanted behaviors or as "anti-scenarios".

Definition 13 (Negative Interpretation, Anti-Scenarios) For all $Spec \subseteq (\tilde{C} \times S)^{\infty}$ and $\alpha \in \langle \text{MSC} \rangle$ we define

$$Spec \vdash_{\neg} \alpha \stackrel{\mathsf{def}}{=} \langle \forall \psi \in Spec, u \in \mathbb{N}, t \in \mathbb{N}_{\infty} :: (\psi, t) \notin [\![\alpha]\!]_u \rangle$$

and say that "*Spec* forbids α", or that "α represents an anti-scenario of *Spec*", if $Spec \vdash_{\neg} \alpha$ holds. □

If α is an anti-scenario with respect to *Spec*, then no behavior represented by α occurs in any element of *Spec*. Anti-scenarios are, in a sense, the duals to MSCs under universal interpretation. While the latter specify what must eventually happen in all executions, the former specify what must not happen in any execution.

Clearly, our definition for anti-scenarios, which we have obtained by "negating" the existential interpretation, is just one of several alternatives. We could also have negated the right-hand-side of the universal interpretation, and would have obtained anti-scenarios as a specification of behavior absent from at least one execution of the system under consideration. Our selection reflects the intuition behind the typical interpretation of anti-scenarios as counterexamples or error cases. A good example of this way of using MSCs is the counterexample output of the model checker SPIN [Hol97]. As the result of trying to check a temporal logic formula that does *not* hold in the system (i.e. the specification) under consideration SPIN presents a specific message trace – in the form of an MSC – leading to a violation of the formula with respect to the specification.

After having detected an MSC as an error case of a specification we can conserve the MSC as a (negative) test case for subsequent versions of the specification.

In the introduction to this chapter we have already given an example for an anti-scenario that we might want to interpret with respect to specifications of the ABRACADABRA-protocol. If *Spec* denotes such a specification, then $Spec \vdash_{\neg} (\rightarrow AC)$ holds if *Spec* contains no executions with an infinite number of simultaneous send requests by X and Y.

Because for an MSC term α the semantics function $[\![\alpha]\!]_u$ allows arbitrary communication besides the one represented explicitly by α, we must resort to the closed world semantics

to specify certain anti-scenarios. As an example, consider the MSC *ETX* of Figure 6.3, where, after a simultaneous send request by X and Y component X starts the transmission of data by sending message d to Y.

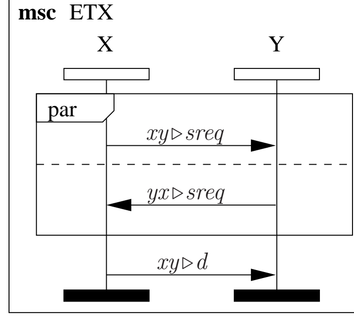


Figure 6.3.: Error case: missing conflict resolution

For any $Spec \subseteq (\tilde{C} \times S)^{\infty}$ the relation $Spec \vdash_{\neg} (\rightarrow ETX)$ denotes that no $xy \triangleright d$ occurrence may follow a simultaneous send request in any execution in *Spec*, even if between the collision and the transmission of data there is a collision resolution and a subsequent send request by X for which Y returns an acknowledgment.

What we really might want to express with MSC *ETX* as an anti-scenario is that $xy \triangleright d$ must not *immediately* follow a collision between X and Y. Recall from Section 6.2.3 that the exact MSC interpretation allows us to express such immediate responses. Therefore, if we interpret *ETX* as an anti-scenario under the closed world semantics with respect to *Spec*, then *ETX* excludes only those executions where $xy \triangleright d$ immediately follows a collision between X and Y.

To take this aspect of negation into account, we introduce a closed world version of our definition of the negative interpretation:

Definition 14 (Closed World Anti-Scenarios) For all specifications $Spec \subseteq (\tilde{C} \times S)^{\infty}$ and MSCs $\alpha \in \langle \text{MSC} \rangle$ we define

$$Spec \vdash_{\neg}^{CW} \alpha \overset{\text{def}}{=} \langle \forall \psi \in Spec, u \in \mathbb{N}, t \in \mathbb{N}_{\infty} :: (\psi, t) \notin [\![\alpha]\!]_{u,CW} \rangle$$

and say that "α represents an anti-scenario of *Spec* under the closed world semantics", if $Spec \vdash_{\neg}^{CW} \alpha$ holds. □

Because of their definitions by means of universal quantification over the elements of *Spec*, we immediately observe the monotonicity of both negative interpretations with respect to specification property refinement.

As an aside we note that if *Spec* is given by means of an MSC, say α, we can *check* whether an MSC β is a closed world anti-scenario with respect to α. This is the case if and only if α and β are inconsistent with respect to join. In Section 7.4.4 we introduce a constructive approach for deciding this question on MSCs.

6.3. Property Specification with MSCs: Safety and Liveness

The MSC interpretations we have introduced in the preceding sections give us a handle at clarifying the "large-scale" properties specified by an MSC, such as whether or not the interactions within the MSC may or must (not) occur as part of any or all system executions. We can use any of these interpretations to relate MSCs with arbitrary other forms of system specifications.

Here, we turn our focus back on MSCs as such, and study the properties we can express with MSCs according to the "traditional" classification into safety and liveness properties as suggested by Lamport (cf. [Lam77, Lam99]). The main result of this section is as follows: our MSC dialect allows us to specify – in essence – liveness properties. Therefore, our MSCs nicely complement specification techniques whose prime target is safety. By means of slight changes to the semantics definition, however, we can also use MSCs to express safety properties.

We start our investigation of safety and liveness properties by giving formal definitions for these terms in Section 6.3.1. In Section 6.3.2 we determine the class of properties our MSC dialect allows us to express. We do so by determining, for each MSC operator, what kinds of properties it *preserves* in a composition or, alternatively, what kinds of properties it *defines*. In particular, we will find that messages, infinite loops, and trigger composition define liveness properties; almost all MSC operators – with guarded MSCs and guarded loops as the only exceptions – preserve liveness properties.

6.3.1. Safety and Liveness

To prepare our discussion of the properties we can express directly with MSCs, we now give formal definitions of the terms "property", "safety", and "liveness". We do so by adapting the well-accepted definitions of [AS85] to the context of our system model.

Definition 15 (System Property) A (system) property[1] q is a set of system executions with respect to the system model of Section 4.2, i.e. $q \subseteq (\tilde{C} \times S)^{\infty}$. □

Thus, the notion of system specification we have introduced earlier in this chapter coincides with the notion of system property we have given here. Determining whether a system specification $Spec \subseteq (\tilde{C} \times S)^{\infty}$ fulfills a property $q \subseteq (\tilde{C} \times S)^{\infty}$ is a matter of checking whether $Spec \subseteq q$ holds.

According to [Lam77, AS85], a safety property includes all system executions where "nothing bad" ever happens. Put another way, if a given system execution does *not* fulfill a safety

[1] In Chapter 7 we differentiate between *system* properties and *component* properties. For the remainder of this section we refer to system properties when using the term property alone.

property, we can find a point in time where "something bad" has happened during this execution; whatever happens afterwards cannot "undo" the "bad thing" (the execution has reached a "point of no return"). Formally, we define

Definition 16 (Safety Property) $q \subseteq (\tilde{C} \times S)^\infty$ is a safety property, if and only if

$$\langle \forall \psi \in (\tilde{C} \times S)^\infty : \psi \notin q : \langle \exists t \in \mathbb{N} :: \langle \forall \varphi \in (\tilde{C} \times S)^\infty :: (\psi \downarrow t) \frown \varphi \notin q \rangle \rangle \rangle$$

holds. □

By inspection of this definition we immediately obtain that both $\emptyset$ and $(\tilde{C} \times S)^\infty$ are safety properties. We call these two properties the *trivial safety properties*.

[Rem92], [Möl99], and [Bro95, Bro99a] give an alternative characterization, which identifies safety properties as the prefix-closed subsets of the set of all system executions. We repeat it here, because it facilitates our proofs for safety properties in Appendix B.3.2.

Definition 17 (Prefix Closure) Let $q \subseteq (\tilde{C} \times S)^\infty$ be a property, and $\psi \in (\tilde{C} \times S)^\infty$ any system execution. We define the prefix-closure operator $prefc : \mathcal{P}((\tilde{C} \times S)^\infty) \rightarrow \mathcal{P}((\tilde{C} \times S)^\infty)$ by

$$\psi \in prefc.q \ \equiv \ \langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in q :: \psi \downarrow t = \varphi \downarrow t \rangle \rangle$$ □

Proposition 9 (Alternative Characterization of Safety) *Let $q \subseteq (\tilde{C} \times S)^\infty$ be a property. q is a safety property if and only if*

$$q = prefc.q$$

holds. □

Proof See Appendix B.3. ■

A liveness property includes all system executions in which "something (good)" happens eventually (cf. [Lam77, AS85]); put another way, there is *no* "point of no return" during any system execution fulfilling the liveness property.

Definition 18 (Liveness Property) $q \subseteq (\tilde{C} \times S)^\infty$ is a liveness property, if and only if

$$\langle \forall \psi \in (\tilde{C} \times S)^\infty, t \in \mathbb{N} :: \langle \exists \varphi \in (\tilde{C} \times S)^\infty :: (\psi \downarrow t) \frown \varphi \in q \rangle \rangle$$

holds. □

From this definition we immediately infer that $(\tilde{C} \times S)^\infty$ is a liveness property, whereas $\emptyset$ is not. We call $(\tilde{C} \times S)^\infty$ the *trivial liveness property*.

Again, there is another characterization of liveness (cf. [Rem92, Möl99, Bro95, Bro99a]), which facilitates our proofs in Appendix B.3.2; it identifies liveness properties as dense sets with respect to $(\tilde{C} \times S)^\infty$:

Proposition 10 (Alternative Characterization of Liveness) *Let $q \subseteq (\tilde{C} \times S)^\infty$ be a property. q is a liveness property if and only if*

$$prefc.q = (\tilde{C} \times S)^\infty$$

holds. □

Proof See Appendix B.3. ■

Some properties are neither safety nor liveness properties according to the definitions above. This is true for properties that simultaneously require "nothing bad" and "something (good)" to happen in all system executions. We will see examples of such properties shortly. The following proposition (cf. [AS85]) shows that we can decompose every property into a safety part and a liveness part:

Proposition 11 (Property Decomposition) *Let $q \subseteq (\tilde{C} \times S)^\infty$ be a property. Then there is a safety property $q_s \subseteq (\tilde{C} \times S)^\infty$ and a liveness property $q_l \subseteq (\tilde{C} \times S)^\infty$, such that*

$$q = q_s \cap q_l$$

holds. □

Proof See Appendix B.3. ■

We call properties that directly fall into the two categories of Definitions 16 and 18 *pure* safety and liveness properties, respectively, to distinguish them from arbitrary other properties.

Equipped with these characterizations of safety and liveness we will now investigate the properties MSCs allow us to express.

6.3.2. MSC Properties

In the following paragraphs we will identify the classes of properties expressible by the operators of our MSC dialect. We derive our results via the following steps. First, we define how to associate a property with an individual MSC. Second, we determine the MSC composition operators defining or preserving safety and liveness properties, respectively. This shows how safety and liveness properties propagate through a composite MSC. As a result we obtain that MSCs without guards, guarded loops, and applications of the join operator describe pure liveness properties. Thus, such MSCs nicely complement other, safety-oriented, specification techniques. Third, we dicuss how to alter the semantics definition to yield safety specifications with MSCs.

Assigning Properties to MSCs

Recall from Section 4.4 that we have defined the semantics $[\![\alpha]\!]_u$ relative to an arbitrary starting time $u \in \mathbb{N}_\infty$. The elements of $[\![\alpha]\!]_u$ are pairs of the form (φ, t), where $\varphi \in (\tilde{C} \times S)^\infty$ and $t \in \mathbb{N}_\infty$ holds. To associate a property with an MSC α, we fix the starting time to 0, and project each element of $[\![\alpha]\!]_0$ onto its first position. More precisely, we define

Definition 19 (MSC Properties) Let $\alpha \in \langle \text{MSC} \rangle$ be an MSC term. We define the property functions $[\![.]\!] : \langle \text{MSC} \rangle \to \mathcal{P}((\tilde{C} \times S)^\infty)$ and $[\![.]\!]_{CW} : \langle \text{MSC} \rangle \to \mathcal{P}((\tilde{C} \times S)^\infty)$ by

$$[\![\alpha]\!] \stackrel{\mathsf{def}}{=} \{\varphi \in (\tilde{C} \times S)^\infty : \langle \exists t \in \mathbb{N}_\infty :: (\varphi, t) \in [\![\alpha]\!]_0 \rangle\}$$

and

$$\begin{aligned} [\![\alpha]\!]_{CW} \stackrel{\mathsf{def}}{=} \{\varphi \in (\tilde{C} \times S)^\infty : & \\ \langle \exists t \in \mathbb{N}_\infty :: \quad & (\varphi, t) \in [\![\alpha]\!]_{0,CW} \\ & \wedge \langle \forall t' > t :: \pi_1(\varphi).t' = \text{<>} \rangle \rangle \} \end{aligned}$$

respectively. We call $[\![\alpha]\!]$ "the property associated with" or "defined by the MSC α", and call $[\![\alpha]\!]_{CW}$ "the property associated with" or "defined by α under the closed world semantics". □

Thus, the property we associate with an MSC α contains all infinite executions "complying" to α from time 0 on. This definition highlights once again the difference between the "regular" and the closed world semantics; if the MSC covers only a finite time interval of the system's behavior, the property under the "regular" semantics allows any behavior afterwards, whereas the property under the closed world semantics requires "silence" instead.

As an example, consider the MSC property $[\![\to A]\!]$, where A is the MSC for the Abracadabra-protocol (cf. Figure 4.20 in Section 4.7.2). $[\![\to A]\!]$ contains all infinite executions "complying" to the Abracadabra-protocol from time 0 on; still, X and Y may perform arbitrary other message exchanges. $[\![\to A]\!]_{CW}$ contains only behaviors where X and Y execute the Abracadabra-protocol and do nothing else.

Our definition of MSC *properties* allows us to use MSCs as *specifications*, as well. We can, for instance, formulate that we interpret a successful transmission initiated by X existentially with respect to $[\![\to A]\!]$ as follows:

$$[\![\to A]\!] \vdash_\exists (\to SX)$$

The next step we take to investigate the properties we can express with our MSC dialect is to observe how safety and liveness properties propagate through MSC composition, and which MSC operators *define* safety and liveness operators, respectively. We limit this investigation (and the accompanying propositions) to MSC properties on the basis of $[\![.]\!]$,

and deal with properties under the closed world semantics separately at the end of this section.

Safety Preservers

Most of the operators we have available in our MSC dialect yield safety properties, if their operands are also safety properties. More precisely, we have

Proposition 12 (Safety Preserving Operators) *Let $\alpha, \beta \in \langle MSC \rangle$, $p \in \langle GUARD \rangle$, $\dagger \in \{\ ;\ ,\ |\ ,\ \sim\ ,\ \otimes\ \}$, $<L> \in \{<*>, <m>, <m,n>\}$ with $m, n \in \mathbb{N}$, $ch \triangleright m \in \langle MSG \rangle$, and $X \in \langle MSCNAME \rangle$ with $(X, \alpha) \in MSCR$. Furthermore, let $[\![\alpha]\!]$ and $[\![\beta]\!]$ be safety properties. Then each of the following is a safety property:*

$$[\![p : \alpha]\!]$$
$$[\![\alpha \dagger \beta]\!]$$
$$[\![\alpha \uparrow_{<L>}]\!]$$
$$[\![\alpha \overset{ch \triangleright m}{\rightarrow} \beta]\!]$$
$$[\![\alpha \Uparrow_{ch \triangleright m}]\!]$$
$$[\![\rightarrow X]\!] \qquad \square$$

Proof See Appendix B.3. ■

Message occurrence, infinite loops, and trigger composition are absent from this list. As we will see below, message occurrence and trigger composition both *generate* liveness properties, whereas infinite loops produce hybrid properties, in general.

Liveness Preservers

All operators but those for guarded MSCs, guarded loops, and join preserve the liveness of their operands. More precisely, we have

Proposition 13 (Liveness Preserving Operators) *Let $\alpha, \beta \in \langle MSC \rangle$, $m, n \in \mathbb{N}_\infty$, $<L> \in \{<*>, <m>, <m,n>\}$, $ch \triangleright m \in \langle MSG \rangle$, and $X \in \langle MSCNAME \rangle$ with $(X, \alpha) \in MSCR$. Furthermore, let $[\![\alpha]\!]$ and $[\![\beta]\!]$ be liveness properties. Then each of the following is a liveness property:*

$$[\![\alpha\ ;\ \beta]\!]$$
$$[\![\alpha \mid \beta]\!]$$
$$[\![\alpha\ \sim\ \beta]\!]$$
$$[\![\alpha \uparrow_{<L>}]\!]$$
$$[\![\alpha \overset{ch \triangleright m}{\rightarrow} \beta]\!]$$
$$[\![\rightarrow X]\!] \qquad \square$$

Proof See Appendix B.3. ■

For $[\![\alpha\Uparrow_{ch\triangleright m}]\!]$ we also conjecture that it yields a liveness property, if $[\![\alpha]\!]$ does. Our rationale for this conjecture is that elements φ of $[\![\alpha\Uparrow_{ch\triangleright m}]\!]$ have one of two possible shapes. The first possibility is that there is a time interval $[u, t]$ with $0 \leq u < t$ such that a contribution represented by α occurs during this time interval, but $ch\triangleright m$ does not. Such elements of $[\![\alpha\Uparrow_{ch\triangleright m}]\!]$ essentially express the property "eventually α". The second possible property expressed by elements of $[\![\alpha\Uparrow_{ch\triangleright m}]\!]$ is "$ch\triangleright m$ occurs infinitely often". "Eventually α", and "infinitely often $ch\triangleright m$" are both classical liveness properties.

Hybrids

empty and **any** both yield $(\tilde{C} \times S)^\infty$ as their associated properties, and are, therefore, both safety and liveness properties.

Guarded MSCs yield neither safety nor liveness properties, if their operand is a liveness property. However, we can determine the property represented by a guarded MSC as the intersection of a nontrivial safety and a nontrivial liveness property. In the property $[\![p : \alpha]\!]$ of an MSC term $p : \alpha$ the safety part captures that the guard p must hold precisely at time 0, whereas the liveness part captures that the execution complies to α from time 0 onward. By a similar line of reasoning we also identify guarded loops as combinations of safety and liveness properties.

The join of two MSCs α and β has also both a safety and a liveness part, if $[\![\alpha]\!]$ and $[\![\beta]\!]$ are liveness properties. Recall from the join operator's definition that $(\psi, t) \in [\![\alpha \otimes \beta]\!]_u$ holds for $\psi \in (\tilde{C} \times S)^\infty$, $u \in \mathbb{N}$, and $t \in \mathbb{N}_\infty$ precisely if there exist t' and t'' such that both $(\psi, t') \in [\![\alpha]\!]_u$ and $(\psi, t'') \in [\![\beta]\!]_u$ hold, and there are no redundant messages from the set $msgs.\alpha \cap msgs.\beta$ in $\psi|_{[u,t]}$. This latter requirement is the safety part of the property expressed by the join operator.

Proposition 14 (Hybrid Properties) *Let $\alpha, \beta \in \langle MSC\rangle$ be such that both $[\![\alpha]\!]$ and $[\![\beta]\!]$ are nontrivial liveness properties, and $p \in \langle GUARD\rangle$. Then, in general, each of the following is neither a pure safety nor a pure liveness property:*

$$[\![p : \alpha]\!]$$
$$[\![\alpha\uparrow_{<p>}]\!]$$
$$[\![\alpha \otimes \beta]\!] \qquad \square$$

Proof See Appendix B.3. ■

Liveness Generators

Two operators *generate* liveness properties, whatever property class their operands belong to: message occurrence and trigger composition.

Recall from Section 4.4 that we have defined the semantics of a single message $ch\triangleright m \in \langle \text{MSG}\rangle$ by

$$[\![ch\triangleright m]\!]_u \stackrel{\text{def}}{=} \{(\varphi, t) \in (\tilde{C} \times S)^\infty \times \mathbb{N} : t = \min\{v : v > u \wedge m \in \pi_1(\varphi).v.ch\}\}$$

As a consequence, message occurrence generates a liveness property: we can always extend any finite execution where the message has not yet occurred by an infinite execution where the message does occur at some finite time.

A similar line of thought shows that trigger composition generates liveness properties. We have

Proposition 15 (Liveness Generators) *Let $\alpha, \beta \in \langle MSC \rangle$, and $ch \triangleright m \in \langle MSG \rangle$. Furthermore, let $[\![\alpha]\!]$ and $[\![\beta]\!]$ be arbitrary nontrivial properties. Then each of the following is a liveness property:*

$$[\![ch \triangleright m]\!]$$
$$[\![\alpha \ \mapsto \ \beta]\!]$$
□

Proof See Appendix B.3. ■

Liveness Specifications with MSCs

Because message occurrence, as we have defined it in Section 4.4, defines a liveness property, and all of our composition operators up to those for guarded MSCs, guarded loops, and join preserve liveness properties, we obtain the following corollary:

Corollary 2 *MSC terms containing neither guarded MSCs, nor guarded loops, nor applications of the join operator express pure liveness properties.* □

This corollary has a methodical consequence. Our MSC dialect nicely complements other specification techniques whose prime focus is safety. As a simple example, consider a system consisting of the components X and Y, connected by means of the channels xy (from X to Y) and yx (from Y to X). Let us assume that X and Y communicate by exchanging the messages $xy \triangleright g$ and $yx \triangleright h$. The following is a safety property for this system: at any time $t \in \mathbb{N}$, Y may have sent at most as many h messages as it has received g messages from X until before time t. We can formalize this property as the set $q \subseteq (\tilde{C} \times S)^\infty$ of executions, where we define q as follows:

$$\psi \in q \stackrel{\mathsf{def}}{=} \langle \forall t \in \mathbb{N} :: cnt.(yx, h, \psi \downarrow (t+1)) \leq cnt.(xy, g, \psi \downarrow t) \rangle$$

Here, the function $cnt : C \times M \times (\tilde{C} \times S)^* \to \mathbb{N}$ counts the number of occurrences of a message on a certain channel in a given execution segment:

$$cnt.(ch, m, \varphi) \stackrel{\mathsf{def}}{=} \begin{cases} 0 & \text{if } |\varphi| = 0 \\ 1 + cnt.(ch, m, \varphi \uparrow 1) & \text{if } |\varphi| > 0 \wedge m \in \pi_1(\varphi).0.ch \\ cnt.(ch, m, \varphi \uparrow 1) & \text{if } |\varphi| > 0 \wedge m \notin \pi_1(\varphi).0.ch \end{cases}$$

Clearly, q is a safety property:

$\psi \in prefc.q$

$\equiv$ (* definition of $prefc$ *)

$\langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in q :: \psi \downarrow t = \varphi \downarrow t \rangle \rangle$

$\equiv$ (* definition of q, predicate calculus *)

$$\langle \forall t \in \mathbb{N} :: \\ \langle \exists \varphi \in (\tilde{C} \times S)^\infty :: \\ \langle \forall t' \in \mathbb{N} :: cnt.(yx, h, \psi \downarrow (t'+1)) \leq cnt.(xy, g, \psi \downarrow t') \rangle \wedge \psi \downarrow t = \varphi \downarrow t \rangle \rangle$$

$\Rightarrow$ (* select $t \geq t' + 1$ *)

$\langle \forall t \in \mathbb{N} :: cnt.(yx, h, \psi \downarrow (t+1)) \leq cnt.(xy, g, \psi \downarrow t) \rangle$

$\equiv$ (* definition of q *)

$\psi \in q$

We can complement q by an MSC specifying a liveness property. Consider MSC L in Figure 6.4. It captures the following requirements:

- there is an infinite number of $xy \triangleright g$ messages, as well as an infinite number of $yx \triangleright h$ messages,
- a $yx \triangleright h$ message eventually follows every $xy \triangleright g$ message.

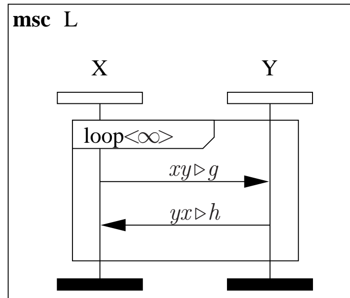


Figure 6.4.: Liveness specification

Together, i.e. by performing the intersection $q \cap [\![\rightarrow L]\!]$, these properties yield a specification restricting the system's behavior in three ways:

1. Y must not produce unsolicited reactions,
2. Y must eventually answer every incoming $xy \triangleright g$ message by a $yx \triangleright h$ message,
3. X issues an infinite number of $xy \triangleright g$ messages.

A typical source for safety specifications are automata models like the ones we have mentioned in Chapter 3. Likewise, the "transition relation" in TLA models the safety part of an overall TLA specification (cf. [Lam99]). We can complement all of these by stating liveness properties by means of MSCs.

Safety Specifications with MSCs

As we have seen above, most of the MSC operators also propagate safety properties. The exceptions are infinite loops and trigger composition. Therefore, if we alter the definition of message occurrence such that its new version yields a safety property instead of a liveness property, and employ neither infinite loops nor trigger composition, we obtain a specification mechanism for safety properties.

We can turn message occurrence into a safety property generator by placing an upper bound on the time until when the message under consideration must have occurred. This places a "bounded response" requirement on the originator of the message. If we require immediate response from the message sender, i.e. if we set the upper bound on the response time to 0, and denote the sending of a message now by $ch \triangleright_s m$, we obtain the following semantics definition:

$$[\![ch \triangleright_s m]\!]_u \overset{\mathsf{def}}{=} \{(\varphi, u+1) \in (\tilde{C} \times S)^\infty \times \mathbb{N} : m \in \pi_1(\varphi).u.ch\}$$

Clearly, $[\![ch \triangleright_s m]\!]$ is a safety property. We can, by looking at what occurs on the channels at time 0, determine immediately whether or not a given execution belongs to $[\![ch \triangleright_s m]\!]$. The extension of $[\![ch \triangleright_s m]\!]_u$ to other upper bounds on the sender's response time is an easy exercise.

Safety and Liveness under the Closed World Semantics

Properties under the closed world semantics are neither pure safety nor pure liveness properties. Instead, we can separate such an MSC property into a safety and a liveness part, similar to what we have done with guarded MSCs under the "regular" semantics, above.

An example for this is message occurrence under the closed world semantics. We can easily show that with

$$\begin{aligned} q_S \overset{\mathsf{def}}{=} \{\varphi \in (\tilde{C} \times S)^\infty : & \\ & \langle \forall ch' : ch' \neq ch : \pi_1(\varphi).ch' = \,<>^\infty \rangle \\ & \wedge (\pi_1(\varphi).ch \neq \,<>^\infty \Rightarrow \langle \exists t \in \mathbb{N} :: \pi_1(\varphi).ch = \,<>^t \frown <m> \frown <>^\infty \rangle\} \end{aligned}$$

and

$$q_L \overset{\mathsf{def}}{=} \{\varphi \in (\tilde{C} \times S)^\infty : \pi_1(\varphi).ch \neq \,<>^\infty\}$$

we have

$$[\![ch \triangleright m]\!]_{CW} = q_S \cap q_L$$

Clearly, q_S is a safety property, and q_L is a liveness property.

Such specific safety and liveness properties do not propagate through an MSC specification under the closed world semantics. The MSC $ch \triangleright m\,;\, ch \triangleright m$, for instance, does not represent the safety property q_S.

However, without proof we note the two parts of the major safety property of an MSC α under the closed world semantics:

- at most the messages specified in α occur in any element of $[\![\alpha]\!]_{CW}$, and
- the order of the messages occurring in any element of $[\![\alpha]\!]_{CW}$ is as specified in α.

The corresponding liveness property of α is that the specified messages do indeed occur (there are no infinite delays).

In Chapter 7 we present this decomposition in more detail; there, we derive these properties – from the viewpoint of individual component specifications – schematically from MSCs. In particular, we will present a transformation scheme that produces from a given MSC the corresponding safety part in the form of an automaton model.

6.4. Related Work

The distinction between the existential and universal interpretation of behavior specifications in general, and of MSC specifications in particular appears also in [Kle98], [KGSB99, Krü99b], and [DH99].

The author of [Kle98] represents system specifications by means of labeled state transition systems, and uses the latter also as a semantics basis for scenario specifications. The instability of properties under property refinement is the defining characteristic of existential system properties in this approach; similarly, stability under property refinement defines a property as being a universal one. Therefore, the notions of existential and universal MSC interpretation we have introduced in this chapter correspond directly with the notions of existential and universal properties of [Kle98]. Our work in this chapter, which is an extension of earlier contributions in [KGSB99, Krü99b], adds the notions of exact interpretation, and negation to the set of interpretations known from [Kle98]. The exact interpretation reduces the semantics of a specification to only the explicitly depicted behavior. The negated interpretation allows us to specify “anti-scenarios”.

As we have discussed already in Section 2.3.6 the authors of [DH99] allow the developer to distinguish between optional and mandatory behavior for all modeling elements within an LSC specification: entire LSCs, individual messages, and locations on component axes. As a result, an LSC specification composes segments of optional and mandatory behavior within the same LSC. The authors of [DH99] use this flexibility to encode alternatives,

and repetition. In our approach we use the dedicated MSC-96 syntax for this purpose, and distinguish between existential (optional) and universal (mandatory) behavior only on the level of entire specifications. This supports the methodical transition of a specification from being a collection of scenarios only to a complete behavior description of the system (components) under consideration.

[Fac95] defines the notions of “scenaric” and “complete” interpretation in the context of Time Sequence Diagrams (TSD). The scenaric interpretation describes a single occurrence of the behavior captured by a TSD. The complete interpretation captures, in addition, an arbitrary number of repetitions of this behavior. Thus, the scenaric TSD interpretation is similar to (but not identical with) our existential MSC interpretation. The complete TSD interpretation most closely corresponds to a combination of our universal MSC interpretation and a corresponding repetition construct.

[DH99], [Fac95] and [NGH93] also discuss the separation of interaction specifications into their safety and liveness part.

In [DH99] safety and liveness specifications are achieved by means of a combination of optional and mandatory modeling elements. [Fac95] proves that typical TSDs specify combinations of safety and liveness properties; he also differentiates between the safety and liveness constraints posed by a TSD at the component providing a service, and at the component using a service. We pick up this discussion again in Chapter 7, where we describe the transition from MSCs to specifications for individual components. In the context of testing the authors of [NGH93] identify MSCs as a specification technique for guarantee properties, i.e. properties fulfilled at least once during any system execution. This, together with our observation that our MSC dialect specifies – mainly – liveness properties, corresponds to our notion of universal MSC interpretation.

6.5. Summary

In this chapter we have discussed two major topics. First, we have defined four MSC interpretations – the existential, universal, exact, and negated MSC interpretation – with respect to other system specifications. Second, we have investigated the classes of properties we can express with our MSC dialect. Together, these two topics help highlight the usage of MSCs as a property oriented description technique in the development process.

The existential interpretation corresponds to the traditional view of MSCs as scenarios. The behavior represented by the MSC may, but need not necessarily happen during an execution of the system under development. The existential interpretation is not monotonic with respect to specification property refinement. This interpretation, therefore, typically underlies the usage of MSCs as a documentation mechanism for relatively short segments of system behavior.

The universal interpretation leaves little room for behavior other than the one represented by the MSC under consideration; at some point in time every execution must exhibit the MSC's behavior. Therefore, the universal MSC interpretation allows us to use MSCs to express eventuality properties.

The exact interpretation leaves *no* room for behavior other than the one represented explicitly by the MSC under consideration. An MSC under exact interpretation fixes the system behavior entirely; put another way, the MSC captures the complete information we have about the system under development.

The negated interpretation treats MSCs as a specification technique for invalid execution segments. One way of using this interpretation is to collect sets of MSCs as "counterexamples" or error cases for the system under development. These counterexamples can later serve as negative test-cases during validation.

To complement our treatment of *how* to relate MSCs with (other) system specifications, we have also studied *what* kinds of properties we can express with MSCs as such. As a result we have obtained that MSCs without guards and join yield liveness properties. This suggests the use of MSCs as an addition to safety-oriented specification techniques, such as the automata we have introduced in Chapter 3. By means of a slightly modified message semantics we have also shown how to use MSCs for safety specifications.

CHAPTER 7

From MSCs to Component Specifications

In this chapter, we discuss the transition from an overall collaboration specification, as given by a set of MSCs, to the specification of individual components. To avoid inconsistencies between the information captured by the MSCs, and the one contained in the individual component specifications, we present two constructive transformation schemes from MSCs to components. The first has relational component specifications in the assumption/commitment format as its result, the other produces finite state machines. The results contained in this chapter help close the gap between global system specifications, and local component specifications.

Contents

7.1. Introduction **216**
7.2. Relational Component Specifications **221**
7.3. From MSCs to A/C-Specifications **230**
7.4. From MSCs to Automaton Specifications **245**
7.5. Related Work **285**
7.6. Summary **292**

7.1. Introduction

The preceding two chapters on the methodical usage of MSCs have – mainly – addressed the following questions:

- How to capture the interaction requirements of distributed components by means of MSCs?
- How to increase the level of detail of MSC specifications?
- How to express relevant system properties with MSCs?

Each of these questions is or can be of relevance within each of the four development phases we consider: analysis, specification, design, and implementation. In Chapter 6 we have already mentioned that MSCs can serve as a description technique for scenarios, as well as for complete interaction behavior specifications. Clearly, we can apply any of the MSC interpretations from Section 6.2 within any of the development phases referred to above.

For a seamless integration of MSCs into the overall development process we are, however, not only interested in using MSCs *within* each development phase, but also in using them *across* all phases. In particular, if we use MSCs during the analysis phase to capture the system's interaction requirements, but want to perform, say, development steps on an automaton specification of the relevant components later, then we need a way to transit from the MSCs to a state-oriented form of specification. As we now briefly discuss, such a transition between models requires care.

Often, the artifacts (MSCs in our case) or models produced during one phase of the development process get thrown away or remain as mere (and sometimes obsolete) documentation when the transition to the next development phase happens. Whereas the model in one phase may capture all system requirements correctly, the model in the next phase may display arbitrary behavior – unless we prove the correctness of the new model formally. In the words of [SGW94] (p. 8): "One of the major trouble spots in traditional systems development is the presence of discontinuities that occur within the development process. These discontinuities are caused by the lack of formal relationships between different notations... These discontinuities also make it difficult to trace the linkages between system requirements and the implementation that is supposed to satisfy them. Maintaining this linkage is important to ensure not only that all the requirements are met, but also that (as the system evolves) the effects of any change can be determined precisely in terms of its effect on the original requirements."

In the chapter on MSC refinement (cf. Chapter 5) we have already reduced some of the potential discontinuities with respect to using MSCs within the development process: not only have we provided refinement notions for every aspect of behavior and structure addressed by an MSC specification, but also have we discussed the consequences of these

refinements on the system model as a whole. This allows us to reflect refinement steps we perform on an MSC also, say, on a corresponding system structure diagram. Thus, we can increase the specification's level of detail directly while working with the MSCs, without having to resort to other description techniques.

In Chapter 6 we have laid a second foundation for the transition from MSCs interpreted as scenarios to complete system specifications, by introducing the exact MSC interpretation. The latter fixes the behavior of all system components to what is explicitly depicted in the MSCs. This interpretation makes precise what is allowed in a collaboration between the system components, and what is not. In this sense the MSCs capture the entire system behavior completely.

It is, however, one of the decisive steps in the development process to transit from collaboration specifications, as MSCs provide them, to the specification of individual components. The ultimate aim of system development is the specification, design and implementation of individual components that provide the required functionality within the environment they are supposed to operate in. One of the major benefits of MSCs is their focus on the interaction behavior of the components we are particularly interested in, embedded within their respective environment. This makes MSCs particularly useful for the specification of the interface the depicted components must provide in order to operate correctly in the context of the overall system.

As an example, recall the specification of the central locking system (CLS) we have used in Chapter 2 to introduce various MSC dialects. The variant of the CLS we consider here consists of four components: a key sensor (*KS*), a left and a right lock motor (*LM* and *RM*), and the controller (*Control*). The controller receives message *kc*▷*lck* or *kc*▷*unlck* from the key sensor when the operator locks or unlocks the car, respectively. Upon receipt of either message the controller initiates the locking and unlocking by issuing appropriate messages (*cl*▷*down*/*cr*▷*down* or *cl*▷*up*/*cr*▷*up*) to both motors. Each of the motors acknowledges the controller's request by sending a reply message (*lc*▷*rdy* and *rc*▷*rdy*) to the controller.

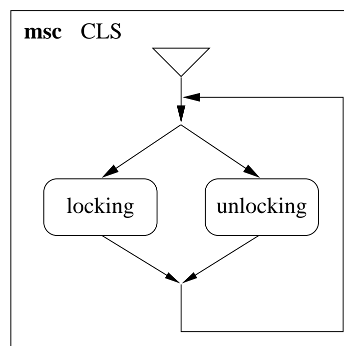


Figure 7.1.: HMSC for the CLS

We can capture these informal requirements by means of three MSCs as shown in Figures 7.1 and 7.2. The HMSC of Figure 7.1 specifies that every system execution is an infinite

sequence of steps, where each step consists of the locking or the unlocking of the car. The "locking" use case appears in Figure 7.2 (a): the MSC shows how the controller interacts with its environment to close the lock. Similarly, the MSC in Figure 7.2 (b) for the "unlocking" use case shows how the controller interacts with its environment to open the lock.

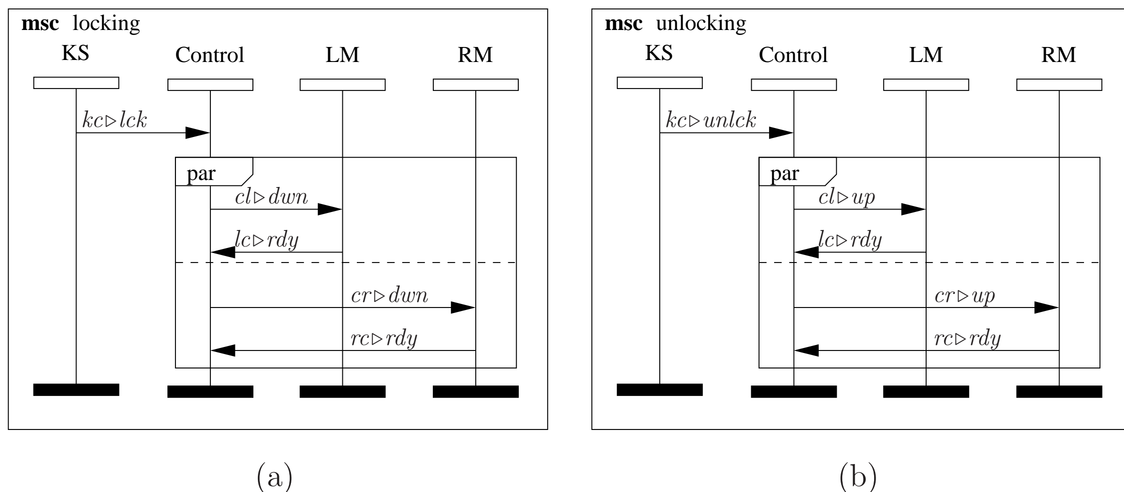


Figure 7.2.: Specification of the "locking" and "unlocking" use case

The MSCs of Figure 7.2 represent a "global" view on the collaboration of the four components to establish the desired effect for the respective use case. The HMSC *CLS* of Figure 7.1 specifies how the locking and unlocking use cases compose to yield the overall system behavior. The exact interpretation of *CLS* (together with *locking* and *unlocking*) fixes the behavior of *KS*, *Control*, *LM*, and *RM* to what explicitly occurs in the MSCs.

Typically, when developing a system like the CLS in this example, we do not have to specify or implement all of the components on our own; we can (or must) assume some of them to be given. In the CLS our task might be, for instance, to specify and implement the controller on the basis of an existing key sensor and existing lock motors. Clearly, we want an implementation of the controller to exhibit the behavior as we have specified it by means of the MSCs – under the assumption that the key sensor and the lock motors also behave as expected.

Therefore, after capturing the "global" interaction requirements of the system part we are interested in, our next task is to come up with individual specifications for the relevant components, such that the individual specifications together fulfill the captured requirements. An obvious way of introducing a discontinuity into the development process at this point is to start developing the individual component specifications from scratch. Because there is no formal relation between the individual component specifications and the interaction requirements specification we can only hope for the former to comply to the latter. The typical way of gaining back confidence in our work is to perform more or less extensive tests on the resulting components, and to check whether the test sequences match

the requirements specification. MSCs can support the testing process nicely, because we can derive both test-drivers and result checks from them rather schematically. However, if testing is our *only* way of establishing the correctness of an implementation we are in quite bad shape: "Program testing can be used to show the presence of bugs, but never to show their absence!" (cf. [DDH72]) – unless the test is exhaustive.

In our approach we avoid the mentioned discontinuity by *constructing* individual component specifications *directly* from the given MSCs. This ensures that the components we thus obtain exhibit precisely the interaction behavior we have captured before by means of the MSCs. The construction process establishes the formal link between the two representations of the relevant component behavior.

We present and discuss two derivation schemes for the construction of individual component specifications from MSCs. The one yields relational component specifications in the assumption/commitment format, the other produces finite state machines. In the following paragraphs, we briefly discuss these two forms of behavior specification, in turn.

Assumption/Commitment Specifications One way to specify the behavior of a particular component is to relate each stream of inputs received by the component on its input channels with sets of possible output streams produced by the component on its output channels; the sets of output streams model the component's reaction to the corresponding input. Such relational component specifications have proven to be a very powerful tool for the development of distributed systems [BDD+92, Bro99a, BS00]. They capture the "black-box"-behavior of the component, in the sense that they only refer to the externally observable component behavior without directly revealing *how* the component establishes its reaction to the presented input.

The format in which an MSC depicts the behavior of the components appearing in it already strongly hints at the use of input/output relations for component specifications: we can view every axis for the component under consideration as a certain part of the relation we want to find. In fact, the way we have defined the semantics of MSCs in Chapter 4 follows precisely this idea.

The *assumption/commitment* format [MC81, Pan90, Bro95] allows us to structure the component specification into two parts: the expectations the component has at the environment (the assumption), and the behavior the component displays (the commitment), provided the environment fulfills the assumption. Intuitively, in the context of MSCs the assumption requires the environment of the component under consideration to provide the expected input messages in the order as given by the MSCs under the exact interpretation. The commitment then ensures that the component partakes as intended in the collaboration and produces its output messages in the required order.

Based on this intuition, we derive assumption/commitment specifications from MSCs in a fully schematic way. This derivation is completely general: it allows us to use the full MSC dialect we have introduced in Chapter 4.

Automaton Specifications As we have mentioned above, the assumption/commitment specifications we derive from MSCs give us a black-box view on the component we are interested in. In later phases of the development process, however, we want to model not only the externally visible component behavior, but also the way how the component achieves its results.

In Chapter 3 we have already introduced various forms of finite state machines as popular models for detailed, state-oriented behavior specifications for individual components.

To support the direct transition from MSCs to individual component specifications in state-oriented form we introduce a fully automatic procedure for deriving finite state machines from MSCs. The procedure is constructive; it takes MSCs as input, and syntactically transforms them into an automaton for the component under consideration. The resulting automaton's set of behaviors is a subset of the semantics of the MSCs we have started with. In the presentation of this procedure we restrict ourselves to a subset of our MSC dialect.

The automaton we obtain as a result of the procedure gives us a "glass-box" view on the component; we can consider this glass-box view as a "jump-start" model for further elaboration, as well as for the construction of early prototypes from MSC specifications.

Closing the Methodical Gap Together, these two transformations close the methodical gap between the specification of component collaboration and the specification of individual component behavior. Closing this gap is a methodical necessity for a seamless integration of MSCs into the overall software development process for distributed systems.

The remainder of this chapter has the following structure. In Section 7.2 we introduce the notation and mathematical concepts we need to describe the black-box and glass-box behavior of individual components. In Section 7.3, we define the assumption/commitment format for relational component specifications and show how to derive assumption/commitment specifications for individual components from MSC specifications. This allows us to extract the black-box behavior of components schematically from an MSC specification. In Section 7.4 we present our transformation procedure from MSCs to finite state machines, which provides us with glass-box behavior specifications for the components we are interested in. Section 7.6 contains our summary.

7.2. Relational Component Specifications

Recall from Section 4.2 that the systems we consider consist of components whose interaction proceeds via directed channels. Because of the channels' directedness we can classify the channels to which a component connects as either *input* channels, or *output* channels, depending on whether the component is the channel's destination or source, respectively. A component controls only its output channels, whereas other system components, which constitute the *environment* of the component under consideration, control the input channels.

Graphically, we depict a component together with its input and output channels by means of system structure diagrams, as we have done before for the system as a whole. Figure 7.3 shows an example where the input and output channels' labels are i_1 through i_n and o_1 through o_m, respectively.

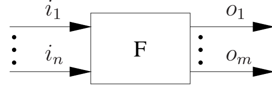


Figure 7.3.: SSD for component F

For the specification of the black-box behavior of individual components we use relations over input channel valuations and output channel valuations as described in [BS00, Bro99a, BK98]. In this section we introduce the notation and mathematical concepts we need within this framework. In our presentation we closely follow [Bro99a, BK98] but restrict ourselves to the notation and concepts of interest for the remainder of this chapter.

7.2.1. Basic Definitions

The separation of the component's channels into input and output channels gives rise to the following definition for the notion of syntactic component interface:

Definition 20 (Syntactic Component Interface) Let C be a set of channels, $F \in P$ a system component, and $I_F \subseteq C$, $O_F \subseteq C$ be defined as follows[1]:

$$I_F \stackrel{\text{def}}{=} \{ch \in C : dst.ch = F\}$$
$$O_F \stackrel{\text{def}}{=} \{ch \in C : src.ch = F\}$$

Then the pair

$$(I_F, O_F) \in \mathcal{P}(C) \times \mathcal{P}(C)$$

is the *syntactic interface* of component F. □

[1] Recall from Chapter 4 that for channels $ch \in C$ the function applications $src.ch$ and $dst.ch$ yield the source and destination component, respectively, of ch, i.e. with $ch = (chn, s, d)$ we have $src.ch = s$ and $dst.ch = d$.

If the component F to which a syntactic interface (I_F, O_F) belongs is clear from the context, we omit the indices from I_F and O_F.

The definition above includes components F where $I_F \cap O_F \neq \emptyset$ holds, i.e. where the sets of input and output channels are not disjoint. Channels in $I_F \cap O_F$ constitute feedback-loops of F. In the following sections we will mainly consider components without feedback-loops, and call the corresponding syntactic interfaces "directed".

Definition 21 (Directed Syntactic Interface) Let $(I_F, O_F) \in \mathcal{P}(C) \times \mathcal{P}(C)$ be the syntactic interface of component $F \in P$. We call (I_F, O_F) and F *directed*, if $I_F \cap O_F = \emptyset$ holds. □

As before, we represent the "content" of a channel over time by means of timed streams, and call the assignment of a timed stream to a channel an infinite valuation or history.

Definition 22 (Valuation, History) Let $X \subseteq C$ be a set of channels, and let M be a set of messages. By

$$\tilde{X} \stackrel{\mathsf{def}}{=} (X \to M^*)$$

we denote the set of *valuations* of the channels in X. By

$$\vec{X} \stackrel{\mathsf{def}}{=} \tilde{X}^\infty$$

we denote the set of *infinite valuations* or *histories* of the channels in X. For a component $F \in P$ with the syntactic interface (I, O) we call $\vec{I}$ and $\vec{O}$ the set of F's *input* and *output histories*, respectively. □

Now we have everything in place to define the notion of black-box component behavior, which we also call the semantic component interface:

Definition 23 (Black-Box Component Behavior, Semantic Interface) Let $F \in P$ be a system component, and let (I, O) be its syntactic interface. We call a relation (here expressed as a family of predicates)[2]

$$F : \vec{I} \to (\vec{O} \to \mathbb{B})$$

the *black-box behavior specification* or *semantic interface* of component F. □

Thus, the semantic interface relates input histories with output histories of the component under consideration. Therefore, we call black-box behavior specifications also *relational component specifications* or I/O behaviors.

The absence of any explicit notion of state or any other explicit information about the inner workings of the component justifies the attribute "black-box" in the definition above.

[2]We identify the name of the relation with the name of its corresponding component if no confusion can arise.

7.2.2. Causality, Realizability, and Nondeterminism

In their full generality, relational component specifications allow us to model quite "strange" components. We can, for instance, easily specify a component predicting its input. Consider the sets $I = \{i\}$, $O = \{o\}$, and $M = \{1, 2\}$ of input and output channels, and messages, respectively. If we define the relation $F : \vec{I} \to (\vec{O} \to \mathbb{B})$ for all $x \in \vec{I}$ and $y \in \vec{O}$ by

$$(F.x).y \;\equiv\; \langle \forall t \in \mathbb{N} :: (x.(t+1)).i = (y.t).o \rangle$$

then the component F produces as its output at any time $t \in \mathbb{N}$ the sequence of values it will receive as input one time unit later.

For modeling reactive components we are, however, interested in specifications without the ability or necessity of predicting the future, because – ultimately – we must implement them as part of a real system.

Causal component specifications, i.e. specifications where any output follows its triggering input, have pleasant properties with respect to the composition of components as we will see later in this section.

Definition 24 (Time Guardedness, Causality) Let $F : \vec{I} \to (\vec{O} \to \mathbb{B})$ be a relational component specification. We call F *time guarded* or *causal*, if

$$\langle \forall t \in \mathbb{N}, x, y \in \vec{I} :: x{\downarrow}t = y{\downarrow}t \Rightarrow (F.x){\downarrow}(t+1) = (F.y){\downarrow}(t+1) \rangle$$

holds. □

The output of a time guarded component at time $t \in \mathbb{N}$ depends only on the input history the component has received strictly before time t. The asymmetry between input and output models the causal dependence of the component's reaction upon the input having triggered the reaction. Another interpretation of the delay between an input and the possible reaction to it is as follows: no component is infinitely fast; therefore, every component needs some time to calculate its output.

Functions $f : \vec{I} \to \vec{O}$ are special cases of relations over $\vec{I}$ and $\vec{O}$. A partial function assigns at most one output stream in $\vec{O}$ to every input stream in $\vec{I}$. Total functions $f : \vec{I} \to \vec{O}$ describe deterministic components.

To characterize specifications for which we can find implementations we introduce the notion of realizability:

Definition 25 (Realizability) Let $F : \vec{I} \to (\vec{O} \to \mathbb{B})$ be a relational component specification. We call F *realizable* if there exists a total time guarded function $f : \vec{I} \to \vec{O}$ such that

$$\langle \forall x \in \vec{I} :: (F.x).(f.x) \rangle$$

holds. □

For a realizable specification we can find a deterministic implementation strategy (a total function) associating with every input history precisely one output history from the set of possible output histories offered by the specification. For some of the possible output histories such a function need not exist. As an example, consider a component that nondeterministically behaves as either the time guarded identity function, which outputs every input with a delay of one time unit, or as the oracle we have specified above. This component is realizable because we can identify the time guarded identity as one implementation strategy for the component.

If *every* possible output history of the component is the result of some deterministic implementation strategy, then we call the specification *fully realizable*. More precisely, we define:

Definition 26 (Full Realizability) Let $F : \vec{I} \to (\vec{O} \to \mathbb{B})$ be a relational component specification. Let the set $RZ.F$ be defined as follows:

$$RZ.F \stackrel{\mathsf{def}}{=} \{f : \vec{I} \to \vec{O} : f \text{ is time guarded} \wedge \langle \forall x \in \vec{I} :: (F.x)(f.x) \rangle\}$$

$RZ.F$ is the set of time guarded implementation strategies for F. We call F *fully realizable* if

$$\langle \forall x \in \vec{I} :: F.x = \{f.x : f \in RZ.f\} \rangle$$

holds. □

For every possible behavior of a fully realizable specification we can find a deterministic implementation.

7.2.3. Composition

To integrate individual components into a larger system we need a notion of composition. Here, we use a very simple composition operator that identifies common channels of its operands. Because channels have a direction, this corresponds to “connecting” the output channels of the one operand with corresponding input channels of the other (cf. Figure 7.4). Syntactically, we denote the composition operator by the symbol “⊗”.

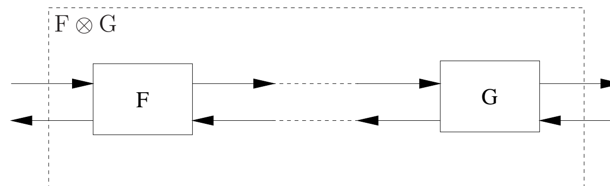


Figure 7.4.: Effect of composing F and G

We will define the composition operator $\otimes$ such that the common channels of the operands are absent from the syntactic interface of the composition. This corresponds to "hiding" the channels by which the components are connected. This form of composition is particularly useful for modeling a strict hierarchic system structure.

We refer the reader to [Bro99a] and [SRS99] for the definition of several other composition operators with and without channel hiding.

Definition 27 (Composition) Let $F : \vec{I}_F \to (\vec{O}_F \to \mathbb{B})$ and $G : \vec{I}_G \to (\vec{O}_G \to \mathbb{B})$ be relational component specifications with $I_F \cup O_F \cup I_G \cup O_G \subseteq C$ and $O_F \cap O_G = \emptyset$. We denote the composition of F and G by $F \otimes G$, and define its syntactic interface (I, O) as follows:

$$I \stackrel{\mathsf{def}}{=} (I_F \cup I_G) \setminus (O_F \cup O_G)$$
$$O \stackrel{\mathsf{def}}{=} (O_F \cup O_G) \setminus (I_F \cup I_G)$$

The semantic interface $F \otimes G : \vec{I} \to (\vec{O} \to \mathbb{B})$ has the following definition, where x and y range over $\vec{I}$ and $\vec{C}$, respectively:

$$(F \otimes G).x.(y|_O) \equiv y|_I = x \wedge F.(y|_{I_F}).(y|_{O_F}) \wedge G.(y|_{I_G}).(y|_{O_G})$$ □

We note the following facts, whose justification the reader can find in [Bro99a] and [SRS99]:

1. if both F and G are realizable, then so is $F \otimes G$,
2. if both F and G are fully realizable, then so is $F \otimes G$,
3. if both F and G are time guarded and fully realizable, then so is $F \otimes G$.

Furthermore, we observe the symmetry of $\otimes$ by inspection of this composition operator's definition. Under the assumption that every channel has at most one source and at most one destination component, $\otimes$ is also associative (the proof of this is straightforward).

A special case of this general form of composition is *sequential* composition:

Definition 28 (Sequential Composition) Let $F : \vec{I}_F \to (\vec{O}_F \to \mathbb{B})$ and $G : \vec{I}_G \to (\vec{O}_G \to \mathbb{B})$ be directed relational component specifications with $O_F = I_G$ and $I_F \cap O_G = \emptyset$. In this state of affairs we introduce $F;G$ as a synonym for $F \otimes G$:

$$F;G \stackrel{\mathsf{def}}{=} F \otimes G$$

and call $F;G$ the *sequential composition* of F and G. □

7.2.4. Component Refinement

[Bro99a] introduces three notions of refinement for component specifications:

- property refinement,
- glass box refinement,
- interaction refinement.

Property Refinement allows us to reduce the set of I/O behaviors represented by a component specification. *Glass box refinement* addresses the decomposition of a component into subcomponents. *Interaction refinement* allows us to change the number and types of channels of a component, and thus to change the representation of communication histories.

We have derived our notions for MSC refinement (cf. Chapter 5) from these three refinement notions for individual components. Therefore, there is a direct correspondence between

- property refinement of MSCs and of components,
- structural refinement of MSCs and glass box refinement of components, and
- message refinement of MSCs and interaction refinement of components.

To make this correspondence explicit we repeat the formal definitions of the component refinement notions from [Bro99a]. In the remainder of this thesis, however, we will concentrate mostly on property refinement.

Definition 29 (Property Refinement) Let $F, G : \vec{I} \to (\vec{O} \to \mathbb{B})$ be relational component specifications. We call G a *property refinement* of F, and write $G \leq_p F$, if

$$\langle \forall x \in \vec{I}, y \in \vec{O} :: G.x.y \Rightarrow F.x.y \rangle$$

holds. □

An alternative definition of $G \leq_p F$ is

$$G \leq_p F \stackrel{\mathsf{def}}{=} G \subseteq F$$

which highlights the similarity of property refinement for components with our definition for property refinement for MSCs.

Definition 30 (Glass Box Refinement) Let $F : \vec{I} \rightarrow (\vec{O} \rightarrow \mathbb{B})$ be a relational component specification. If there exist relational component specifications $F_i : \vec{I_i} \rightarrow (\vec{O_i} \rightarrow \mathbb{B})$ for $i \in [1, n]$ for some $n \in \mathbb{N}$, $n \geq 1$, such that

$$F_1 \otimes \ldots \otimes F_n \leq_p F$$

holds, then we call $F_1 \otimes \ldots \otimes F_n$ a glass box refinement of F. □

Note how closely our definition of structural refinement of MSCs follows the idea behind this definition of glass box refinement; both of them allow the introduction of "fresh" local channels for the communication between the subcomponents.

[Bro99a] also treats the refinement of a component specification into state transition systems as a special case of glass box refinement. We deal with this special case of property refinement later in this chapter, in detail.

Interaction refinement allows us to change the number and names of a component's input and output channels, as well as the granularity of the messages on these channels. Here, we consider only one special case of this very general refinement notion. For a much more detailed treatment of interaction refinement we refer the reader to [Bro99a].

Definition 31 (Interaction Refinement) Let $F : \vec{I_F} \rightarrow (\vec{O_F} \rightarrow \mathbb{B})$, $G : \vec{I_G} \rightarrow (\vec{O_G} \rightarrow \mathbb{B})$, $A_1 : \vec{I_G} \rightarrow (\vec{I_F} \rightarrow \mathbb{B})$, and $A_2 : \vec{O_G} \rightarrow (\vec{O_F} \rightarrow \mathbb{B})$ be directed relational component specifications with $I_G \cap O_F = \emptyset$ and $I_F \cap O_G = \emptyset$. We call G an *interaction refinement* of F, and write $G \leq_m F$, if

$$G; A_2 \leq_p A_1; F$$

holds. □

Intuitively, A_1 and A_2 in this definition are components whose purpose is to translate the input and output streams of G into those for F; this is a form of abstraction. Figure 7.5 illustrates this situation.

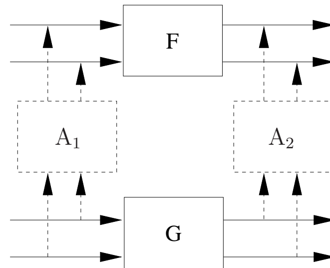


Figure 7.5.: Interaction refinement

Because A_1 and A_2 are arbitrary directed components we can express all kinds of changes to messages and channels by means of interaction refinement; in particular, via appropriately

chosen relations A_1 and A_2 we can express the implementation of a single message by means of an entire protocol. This explains the proximity between this notion of interaction refinement and our definition of message refinement for MSCs.

We will come back to the relationship between MSC refinement and component refinement in Section 7.3.4.

7.2.5. Component Properties, Safety and Liveness of Components

In Section 6.3.1 we have, on the basis of our system model from Section 4.2, introduced the notions of system properties in general, and MSC properties in particular. Here, we introduce similar concepts for individual components, and also adapt the notions of safety and liveness we have defined in Section 6.3.1, accordingly.

A system property $q \subseteq (\tilde{C} \times S)^\infty$ is a set of system executions that "have" or "exhibit" the property. Components appear only rather indirectly in this formalization of system properties; channels – not their sources and destinations – are in the center of concern here. This gives us a global view on the communication (and state changes) within the system. The direction of the channels (from their source to their destination component), as well as the individual responsibilities for establishing a certain message order are of little relevance in this view.

For an individual component, however, the directedness of its channels is of high significance. With feedback channels as the only exception, a component has no direct control of the histories it receives on its input channels. These are the channels controlled by the component's environment.

The component has, however, full control over its output channels. The only way for the environment to influence the component's output histories is to provide the adequate input histories; what the component produces as its reaction to this input is entirely up to the component.

An adequate notion of component properties must, therefore, take the directedness of the component's channels, as well as the component's responsibility for establishing the desired output into account. This responsibility clearly differentiates, for instance, the definitions of component safety and liveness from their "global" counterparts, as we will see, below. A component can establish only the safety and liveness of its output, but not the one of its input.

In the following definitions, we follow [Bro99a, Bro95].

Definition 32 (Component Property) We call $\vec{I} \rightarrow (\vec{O} \rightarrow \mathbb{B})$ the *set of component properties* for the syntactic interface (I, O). If $F : \vec{I} \rightarrow (\vec{O} \rightarrow \mathbb{B})$ is a relational component specification, then we call F also a *component property.* $\square$

This defines the notion of component property directly on the basis of the (directed) relation between the component's inputs and outputs.

In Section 6.3.1 we have classified system properties into safety and liveness properties. We have defined system safety properties to be those properties that hold on all finite prefixes of all system executions; our definition of liveness properties identifies them as those properties that hold eventually in all infinite system executions. These definitions refer to the system as a whole. A system safety property, therefore, makes a statement about the joint safety of all components of the system; similarly, a system liveness property makes a statement about the entire collaboration of all components.

As we have argued above, each component can control only its output directly. Therefore, it makes sense to define the notions of component safety and liveness with respect to the *output* of the component for any given input.

The following two definitions for component safety and liveness are reformulations of the alternative characterizations for system safety ($prefc.q = q$ for some $q \subseteq (\tilde{C} \times S)^{\infty}$) and system liveness ($prefc.q = (\tilde{C} \times S)^{\infty}$) from Section 6.3.1. In these reformulations we introduce an asymmetry between input and output by fixing an arbitrary input history and determining whether the output of the component is safe or live, respectively.

Definition 33 (Component Safety Property) Let $F : \vec{I} \to (\vec{O} \to \mathbb{B})$ be a relational component specification. We call F a *component safety specification* or *component safety property*, if

$$\langle \forall x \in \vec{I}, y \in \vec{O} :: \langle \forall t \in \mathbb{N} :: \langle \exists z \in \vec{O} :: z{\downarrow}t = y{\downarrow}t \ \wedge \ (F.x).z \rangle \equiv (F.x).y \rangle \rangle$$

holds. □

Note the form of the left-hand-side operand of the equivalence in this definition; its result is the prefix closure of the outputs of component F with respect to the fixed input history x.

Time guardedness is a simple example for a component safety property.

Definition 34 (Component Liveness Property) Let $F : \vec{I} \to (\vec{O} \to \mathbb{B})$ be a relational component specification. We call F a *component liveness specification* or *component liveness property*, if

$$\langle \forall x \in \vec{I}, y \in \vec{O}, t \in \mathbb{N} :: \langle \exists z \in \vec{O} :: z{\downarrow}t = y{\downarrow}t \ \wedge \ (F.x).z \rangle \rangle$$

holds. □

Given these two definitions, as well as those for system safety and liveness, we face an important methodical question (cf. [Bro95, BK98]): given a system safety or liveness specification, how can we decompose it into safety and liveness specifications for individual components?

We answer this question in the following section, where we treat the decomposition of MSC specifications into relational component specifications in the assumption/commitment format.

7.3. From MSCs to A/C-Specifications

In the preceding section we have described the framework we use to model individual components. Our aim for this section is to derive the specification of an individual component from a given MSC.

Our point of departure here is the observation that an MSC describes the behavior of each component occurring in it within a certain context or environment. This context consists, on the structural side, of all the other components partaking in the depicted interaction, as well as, on the behavioral side, of the contribution of these other components to the interaction. Clearly, we want every individual component's specification to be such that if the environment exhibits the behavior required by the MSC, then the component fulfills its part of the collaboration.

As an example, recall the specification of the CLS from Section 7.1 by means of the MSCs *CLS*, *locking*, and *unlocking* from Figures 7.1, 7.2 (a) and 7.2 (b). Let us assume that our task is to develop an individual component specification for the component *Control*. This means that we have to come up with a relational component specification

$$Control : \vec{I}_{Control} \rightarrow (\vec{O}_{Control} \rightarrow \mathbb{B})$$

where $\vec{I}_{Control} = \{kc, lc, rc\}$ and $\vec{O}_{Control} = \{cl, cr\}$, such that it associates appropriate output histories with the input histories received by *Control*. In particular, if message $kc \triangleright lck$ occurs we want *Control* to react by sending messages $cl \triangleright down$ and $cr \triangleright down$ and to "wait for" the replies $lc \triangleright rdy$ and $rc \triangleright rdy$.

Without further information, beyond the given MSCs, about *Control* and its environment, we cannot, in general, say how *Control* must or should react if the environment displays *other* behavior than the explicitly depicted one. What if, for instance, the component *LM* would reply message $cl \triangleright down$ by $lc \triangleright error$? The mentioned MSCs give no indication how *Control* should handle this situation.

One way of dealing with this problem is to fix the behavior of a component only for those input histories where the environment exhibits the expected behavior, and to leave the component's behavior unspecified otherwise. Put another way we commit the component to a certain behavior only provided the environment fulfills certain assumptions.

Component specifications that distinguish, and make explicit the assumptions made by the component at its environment, and the commitment that the component promises to fulfill

are called assumption/commitment (A/C) or rely/guarantee specifications in the literature (cf., among others, [MC81, Pan90, Bro95], and the references contained therein).

Guided by the intuition that an MSC indeed depicts the component we are interested in together and in collaboration with the relevant part of its environment, we study the A/C specification style in more detail in the remainder of this section. In Section 7.3.1 we review the A/C format for relational component specifications along the lines of [Bro95]. In accordance with the system model we have introduced in Section 4.2 we capture the collaboration of a component with its environment by means of predicates over I/O histories, and call these predicates "interaction interfaces" in Section 7.3.2. In Section 7.3.3 we present a decomposition scheme that allows us to turn any interaction interface into a relational component specification in the A/C style. Because MSC properties under the closed world semantics (cf. Section 6.3.2) are special cases of interaction interfaces, the decomposition is applicable to MSC specifications as well. On this basis we take another look at MSC refinement in Section 7.3.4, and determine the implications of an MSC property refinement on the individual component specifications we can derive from the respective MSCs.

7.3.1. A/C Specifications

The A/C specification style has been extensively studied in the literature, mainly as a tool for achieving modular specifications with the potential for simplified verification (see, among others, [XS98, Sha98] for an overview and further references). The modularity of A/C specifications stems from the clear separation of the specification into the responsibilities of the component under consideration, and of its environment. This, by itself, is already helpful from a methodical point of view as [Bro95] points out, because it supports the definition of clear semantic component interfaces. Whether the separation into assumptions and commitments has indeed beneficial consequences on the task of verification is debatable (cf. [Lam98, Sha98]) and certainly depends on how strongly the assumptions and commitments are formulated.

Here, we concentrate on the first aspect responsible for the popularity of A/C specifications: their potential for the definition of clear semantic component interfaces.

Before we describe in detail how to derive an A/C specification directly from a given MSC specification, we first make the notions of assumptions and commitments for relational component specifications precise. [Bro95] contains a detailed derivation of several A/C specification styles for a functional setting. Here we pick the most general one of these for our purposes.

Definition 35 (A/C Style for Relational Component Specifications)
Let $F : \vec{I} \to (\vec{O} \to \mathbb{B})$ be a relational component specification. If there exist relations $E_S, E_L, F_S, F_L : \vec{I} \to (\vec{O} \to \mathbb{B})$ such that for all $x \in \vec{I}$ and $y \in \vec{O}$

$$\begin{aligned} F.x.y \equiv \quad & (E_S.x.y \ \Rightarrow \ F_S.x.y) \\ & \wedge (E_S.x.y \wedge E_L.x.y \ \Rightarrow \ F_L.x.y) \end{aligned} \tag{7.1}$$

holds, and E_S and F_S are component safety specifications, and E_L and F_L are component liveness specifications, then we call the right-hand-side of Equation (7.1) a *relational A/C specification* for component F. Furthermore, we call E_S the environment safety assumption, E_L the environment liveness assumption, F_S the component safety commitment, and F_L the component liveness commitment. □

Thus, an A/C specification consists of two parts, represented by the two (outer) conjuncts of Equation (7.1). The first part deals with the safety properties ensured by the component, provided the environment fulfills certain safety requirements. The second part deals with the liveness properties ensured by the component, provided the environment fulfills its safety and liveness requirements.

Figure 7.6 shows the structural view behind the decomposition of responsibilities between component F and its environment E. F controls the channels in O, whereas E controls the channels in I. In this sense, F and E are each other's "duals". Because of this duality, the requirement imposed by Definition 35 on E_S and E_L to be safety and liveness properties, respectively, is sloppy. Recall that the definition of component properties depends on the directedness of input and output channels. A more precise formulation would have been: there exist $\hat{E}_S, \hat{E}_L : \vec{O} \rightarrow (\hat{I} \rightarrow \mathbb{B})$ such that $\hat{E}_S$ and $\hat{E}_L$ are component safety and liveness specifications, respectively, and we have $\langle \forall x \in \vec{I}, y \in \vec{O} :: E_S.x.y \equiv \hat{E}_S.y.x \rangle$ and $\langle \forall x \in \vec{I}, y \in \vec{O} :: E_L.x.y \equiv \hat{E}_L.y.x \rangle$. In view of this quick remedy, we stick with the signatures for E_S and E_L as they occur in Definition 35.

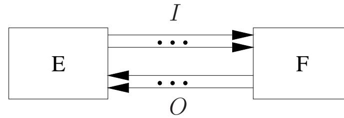


Figure 7.6.: Structural view behind A/C specifications

By closer inspection of Definition 35 we observe that $F.x.y$ holds trivially if either $E_S.x.y$ is false, or $E_S.x.y$ holds, but $E_L.x.y$ does not. If $E_S.x.y$ is false, then there is a time $t \in \mathbb{N}$ such that F cannot react properly to the input $x \downarrow t$ it has received until time t (recall that E_S is a "dual" safety property). If $E_L.x.y$ is false, then from some time $t \in \mathbb{N}$ on the environment does never produce an input when F expects it to occur.

Therefore, a component F specified in the A/C format is *reactive* in the following sense: for every input history $x \in \vec{I}$ there exists at least one output history $y \in \vec{O}$, such that $F.x.y$ holds.

As [Bro95] explains in detail, each of the relations E_S, E_L, F_S, and F_L must reference both the component's input and its output histories to support specifications of full generality. Intuitively, we need both the input and output histories of a component to "reconstruct" the component's internal state from a black-box behavior. Later in this section we will encounter an environment safety assumption we can only formulate if we refer to both the input and the output histories of the component.

As a simple example, we give an alternative specification for component LM of the CLS in the A/C format. Given the message set $M = \{dwn, up, lck, unlck, rdy\}$, the component set $P = \{Control, LM, RM, KS\}$, and the channel sets $C = \{kc, cl, lc, cr, rc\}$, $I = \{cl\}$, and $O = \{lc\}$ we define the relations $E_S, E_L, LM_S, LM_L : \vec{I} \to (\vec{O} \to \mathbb{B})$ as follows:

$$
\begin{aligned}
E_S.x.y &\equiv \langle \forall t \in \mathbb{N} :: x.t.cl \in \{\langle up \rangle, \langle dwn \rangle, \langle\rangle\} \rangle \\
LM_S.x.y &\equiv \langle \forall t \in \mathbb{N} : y.t.lc \neq \langle\rangle : \\
&\qquad\qquad y.t.lc = \langle rdy \rangle \\
&\qquad \wedge \langle \exists t' < t : x.t'.cl \in \{\langle up \rangle, \langle dwn \rangle\} : \langle \forall t'' : t' < t'' < t : y.t.lc = \langle\rangle \rangle \rangle \rangle \\
E_L.x.y &\equiv \text{true} \\
LM_L.x.y &\equiv \langle \forall t \in \mathbb{N} :: x.t.cl \neq \langle\rangle \Rightarrow \langle \exists t' \in \mathbb{N} : t' > t : y.t'.lc \neq \langle\rangle \rangle \rangle
\end{aligned}
$$

Relation E_S constrains the input histories of component LM to messages of the form $cl \triangleright up$ and $cl \triangleright dwn$, if a message occurs on channel cl at all. LM_S specifies that if LM produces an output it is the message $lc \triangleright rdy$; moreover, LM produces neither unsolicited, nor redundant replies. E_L imposes only the trivial liveness constraint on the environment; it may or may not send messages. Relation LM_L specifies that if there is a request on LM's input channel then LM will send a reply eventually.

By means of these four predicates we define the relational component specification in A/C form for $LM : \vec{I} \to (\vec{O} \to \mathbb{B})$ as follows:

$$
\begin{aligned}
LM.x.y \equiv \quad & (E_S.x.y \Rightarrow LM_S.x.y) \\
\wedge\, & (E_S.x.y \wedge E_L.x.y \Rightarrow LM_L.x.y)
\end{aligned}
$$

for all $x \in \vec{I}$ and $y \in \vec{O}$.

The relations E_S and E_L in this specification are rather weak. For instance, E_S allows multiple occurrences of *up* or *dwn* messages in sequence, without a corresponding reply from LM in between. E_L poses no constraint at all at the liveness of the environment.

If we take a closer look at the MSCs *CLS*, *locking*, and *unlocking* from Figures 7.1, 7.2 (a), and 7.2 (b) (under the exact interpretation), we notice additional safety and liveness requirements in the MSC specification. MSC *CLS*, for instance, specifies an infinite number of occurrences of either use case (locking or unlocking). This induces an infinite number of either $cl \triangleright up$ or $cl \triangleright dwn$ messages as well. Furthermore, the MSCs *locking* and *unlocking* require the occurrence of an $lc \triangleright rdy$ message between any two messages from the set $\{cl \triangleright up, cl \triangleright dwn\}$. Intuitively, this means that the controller waits until it has received a reply from the motor before it issues another request. We can enrich the A/C specification above by means of the following two relations E'_S and E'_L to capture these additional properties:

$$\begin{aligned} E'_S.x.y \;\equiv\; & \; E_S.x.y \\ & \wedge \langle \forall t, t' \in \mathbb{N} : t' > t \wedge x.t.cl \in \{\texttt{<}up\texttt{>}, \texttt{<}dwn\texttt{>}\} \wedge x.t'.cl \in \{\texttt{<}up\texttt{>}, \texttt{<}dwn\texttt{>}\} : \\ & \quad \langle \exists t'' : t < t'' < t' : y.t''.lc = \texttt{<}rdy\texttt{>} \rangle \rangle \\ E'_L.x.y \;\equiv\; & \#(\{up, dwn\} \copyright x.lc) = \infty \end{aligned}$$

In the definition of E'_S we must refer to both the input and the output histories of *LM*. Otherwise we could not capture the condition under which the environment must fulfill the assumption. The assumption makes a statement about the "state" of *LM* in which it can handle requests from the environment: *LM* must have replied any pending prior request.

This example already shows that even for "simple" components like *LM* an A/C specification can be quite elaborate. By means of the construction we present in the remainder of this section, however, we can extract A/C specifications from MSC specifications fully schematically.

7.3.2. MSCs and Interaction Interfaces

Now that we have fixed the format of the component specifications we want to end up with, we put our focus back on the overall collaboration specifications we assume given as the starting point of our derivation scheme. Of course, we use MSCs to represent these overall collaboration specifications. To this end, we first define the notion of *interaction interface*, which captures the black-box behavior of all components partaking in a collaboration. Second, we show how to obtain an interaction interface from an MSC specification; this is not a big step, because we have defined the semantic basis for MSCs essentially as a special form of interaction interfaces.

An important first step in the derivation of an individual component specification from an overall collaboration specification is to fix both the component, and the subset of its channels we are interested in. This defines, in particular, what part of the system we consider as the component's environment.

If, for instance, we set out to derive an A/C specification for component *LM* in our CLS example, then we have several (nontrivial) options for defining the environment for this component:

- *Control*
- *Control*, *RM*
- *Control*, *KS*
- *Control*, *RM*, *KS*.

If we add other components, besides *Control* (to which *LM* connects directly), to the environment of *LM*, then we indirectly strengthen the requirements at *LM* through the interplay between components without direct connection with *LM*.

In view of this freedom of choice we define interaction interfaces as a projection of all channel histories on a certain subset thereof. Clearly, this subset should include all relevant input and output channels of the component under consideration.

Definition 36 (Interaction Interface) Let C denote the set of directed channels in the system as before. Let $I \subseteq C$, $O \subseteq C$ be sets of channels. We call any predicate

$$R : \overrightarrow{(I \cup O)} \rightarrow \mathbb{B}$$

an *interaction interface specification* with respect to the syntactic interface (I, O). By $\overrightarrow{(I \cup O)}$ we denote the set of infinite valuations (or histories) over the channel set $I \cup O$. If (I, O) is directed, we call R directed as well. □

An interaction interface identifies all channel valuations where the components controlling the channels in I and O all display the required behavior. In other words, an interaction interface is the projection of the interaction part of the system model from Section 4.2 onto a subset of the channel set C. Writing interaction interfaces as characteristic predicates instead of as sets is triggered by the technical convenience of predicates in the formulation of component specifications.

For the derivation of individual component specifications we are, of course, not interested in arbitrary interaction interfaces; instead, we select the sets I and O such that they are subsets of the sets of input and output channels of the component under consideration. Introducing this directedness of the channels right from the beginning would, however, destroy the symmetry with respect to the component and its environment unnecessarily early.

If R is a directed interaction interface over (I, O), and $x \in \vec{I}$, $y \in \vec{O}$ are valuations of the channels in I and O, respectively, we write $x \oplus y$ for the valuation of the channels in $I \cup O$, whose projections on I and O yield x and y, respectively. More formally, we define

$$ch \in I \quad \Rightarrow \quad (x \oplus y).ch \stackrel{\mathsf{def}}{=} x.ch$$

and

$$ch \in O \quad \Rightarrow \quad (x \oplus y).ch \stackrel{\mathsf{def}}{=} y.ch$$

In Section 7.2 we have introduced three special kinds of relational component specifications: time guarded, realizable, and fully realizable ones. Because interaction interfaces are, in essence, specifications of the joint behavior of multiple components, we can lift these qualities to interaction interfaces as well. We demonstrate this by means of time guardedness.

Definition 37 (Time Guarded Interaction Interface) Let $R : \overrightarrow{(I \cup O)} \rightarrow \mathbb{B}$ be a directed interaction interface. We call R *time guarded*, if both

$$\langle \forall x, x' \in \vec{I}, t \in \mathbb{N} : x\downarrow t = x'\downarrow t : \{y\downarrow(t+1) : R.(x \oplus y)\} = \{y\downarrow(t+1) : R.(x' \oplus y)\}\rangle$$

and

$$\langle \forall y, y' \in \vec{O}, t \in \mathbb{N} : y\downarrow t = y'\downarrow t : \{x\downarrow(t+1) : R.(x \oplus y)\} = \{x\downarrow(t+1) : R.(x \oplus y')\}\rangle$$

hold. □

Intuitively, an interaction interface over the syntactic interface (I, O) is time guarded, if all components controlling the channels in I are time guarded, and all components controlling the channels in O are time guarded.

Time guardedness for interaction interfaces is a safety property of the entire subsystem defined by the syntactic interface (I, O).

The way we have defined the semantics of MSCs and the notion of interaction interfaces allows us an easy transition from the one to the other. As the basis for this transition we use the exact MSC interpretation we have introduced in Section 6.2.3. It captures precisely the interaction sequences we want the component under consideration to display – nothing more, and nothing less. More precisely, we use the extension of the exact interpretation to MSC properties from Section 6.3.2.

Definition 38 (Derived Interaction Interface) Let $\alpha \in \langle \text{MSC} \rangle$ be an MSC term. Let $I \subseteq C$ and $O \subseteq C$ be sets of channels. Then we call the interaction interface

$$R_\alpha : \overrightarrow{(I \cup O)} \rightarrow \mathbb{B}$$

which we define for $x \in \overrightarrow{(I \cup O)}$ by

$$R_\alpha.x \equiv \langle \exists \varphi \in [\![\alpha]\!]_{CW} :: \pi_1(\varphi)|_{I \cup O} = x \rangle$$

the *interaction interface derived from* α with respect to (I, O). □

Thus, the interaction interface with respect to (I, O) derived from an MSC α is the projection of the channel valuations under the exact MSC interpretation for α onto $I \cup O$.

The interaction interfaces we derive from MSCs are time guarded, as the following proposition shows.

Proposition 16 (Time Guardedness of MSCs) *Let* $\alpha, \beta \in \langle MSC \rangle$ *be MSC terms, and* $I, O \subseteq C$ *sets of channels such that* (I, O) *is directed, and* $R_\alpha : \overrightarrow{(I \cup O)} \rightarrow \mathbb{B}$ *and* $R_\beta : \overrightarrow{(I \cup O)} \rightarrow \mathbb{B}$ *are time guarded. Furthermore, let* $p \in \langle GUARD \rangle$, $ch \triangleright m \in \langle MSG \rangle$, *and* $<\!L\!> \in \{<\!p\!>, <\!n\!>, <\!m,n\!>\}$ *hold for* $m, n \in \mathbb{N}_\infty$. *Then each of the following interaction interfaces (of the same signature as* R_α *and* R_β*) is also time guarded:* $R_{\textbf{empty}}$, $R_{\textbf{any}}$, $R_{ch \triangleright m}$, $R_{\alpha\,;\,\beta}$, $R_{p:\alpha}$, $R_{\alpha\,|\,\beta}$, $R_{\alpha\,\sim\,\beta}$, $R_{\alpha \otimes \beta}$, $R_{\alpha \uparrow_{<L>}}$, $R_{\alpha \overset{ch \triangleright m}{\rightarrow} \beta}$, $R_{\alpha \Uparrow_{ch \triangleright m}}$, $R_{\alpha \mapsto \beta}$. □

Proof See Appendix B.4 for the details. The basic intuition behind this result is that time guardedness is a safety property, and all MSC operators yield properties we can decompose into a safety part (implying time guardedness) and a liveness part. Finally, $[\![\alpha \mapsto \beta]\!]_{CW} = [\![\textbf{empty}]\!]_{CW}$, and therefore, $R_{\alpha \mapsto \beta}$ is trivially time guarded. ■

This proposition identifies the MSC dialect we have introduced in Chapter 4 as a graphical description technique for time guarded interaction interfaces.

Now we have everything in place to describe the transition from an MSC to an individual component specification in the A/C format.

7.3.3. From MSCs to A/C Specifications

In this section we describe the transformation of an MSC specification into an individual component specification in the A/C format. This transformation consists of several steps:

1. Fix the MSC α we want to transform.
2. Fix the component F and the syntactic interface (I, O) for which we want to obtain an A/C specification.
3. Derive environment and component safety and liveness relations from the interaction interface R_α with respect to (I, O).

Throughout this section we assume given a directed component F. In terms of the MSC α this means the absence of message arrows that both start and end at F's axis in α.

Let $R : \overrightarrow{(I \cup O)} \to \mathbb{B}$ be any directed and time guarded interaction interface, such that $I \subseteq \{ch \in C : dst.ch = F\}$ and $O \subseteq \{ch \in C : src.ch = F\}$ holds. Our goal is to construct relations E_S, E_L, F_S, and F_L, and to define

$$F : \vec{I} \to (\vec{O} \to \mathbb{B})$$

by a time guarded relational A/C specification

$$\begin{aligned} F.x.y \equiv \quad & (E_S.x.y \;\Rightarrow\; F_S.x.y) \\ & \wedge (E_S.x.y \wedge E_L.x.y \;\Rightarrow\; F_L.x.y) \end{aligned}$$

We start with the definition of E_S and F_S, i.e. with the environment safety assumption and F's safety commitment. Intuitively, E_S specifies the following safety property: the inputs F receives on its input channels at any time $t \in \mathbb{N}$ fulfill the interaction interface R. In other words, at every time point $t \in \mathbb{N}$ the environment produces only input histories

F can handle according to R's definition. We use the relation $E_S^t.x.y : \vec{I} \rightarrow (\vec{O} \rightarrow \mathbb{B})$ to capture this intuition until time $t \in \mathbb{N}$:

$$E_S^t.x.y \equiv \langle \exists x' \in \vec{I}, y' \in \vec{O} :: x \downarrow t = x' \downarrow t \wedge y \downarrow t = y' \downarrow t \wedge R.(x' \oplus y') \rangle$$

For any $t \in \mathbb{N}$, E_S^t identifies those finite prefixes of length t of x permitted as the prefix of an input of component F according to the interaction interface R. More specifically, E_S^t identifies those pairs of input/output channel valuations where both the environment and F proceed as specified by R. From the view of component F, however, the real constraint imposed by E_S^t is on x. Based on the definition of the relations E_S^t, we construct E_S as their limit with respect to t:

$$E_S.x.y \equiv \langle \forall t \in \mathbb{N} :: E_S^t.x.y \rangle$$

Clearly, E_S is a safety relation.

Next, we turn our attention to the safety commitment of F. If the environment provides the correct input on F's input channels until time $t \in \mathbb{N}$, then – by the requirement of time guardedness – we commit F to producing the correct output until time $t + 1$:

$$F_S^t.x.y \equiv \langle \exists x' \in \vec{I}, y' \in \vec{O} :: x \downarrow t = x' \downarrow t \wedge y \downarrow (t+1) = y' \downarrow (t+1) \wedge R.(x' \oplus y') \rangle$$

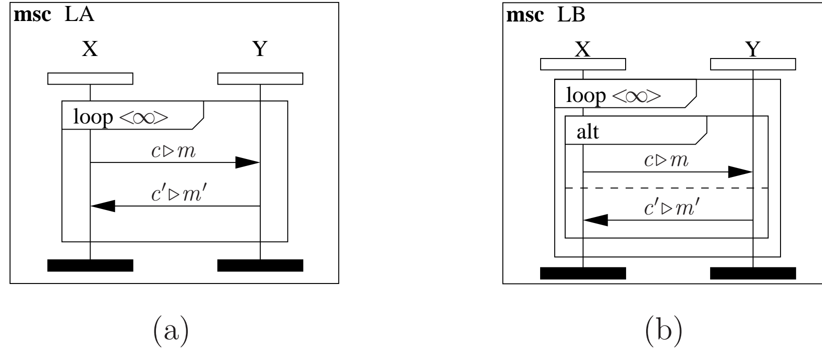
F_S^t identifies those prefixes of pairs of input/output valuations, where the input of F is correct at least until time t and the output of F is correct at least until time $t + 1$. Again, we take F_S as the limit of the relations F_S^t with respect to t:

$$F_S.x.y \equiv \langle \forall t \in \mathbb{N} :: F_S^t.x.y \rangle$$

The decomposition of R into environment and component safety properties is purely schematic. E_S captures that the environment never makes a wrong "move". F_S specifies that F never produces the wrong output (under the assumption of the environment working as expected), and, moreover, that F is time guarded. In the composite system neither F nor its environment may fail to fulfill a safety property at any time. If R is a derived interaction interface with respect to an MSC α, then the environment safety assumption requires the environment to send only messages for which there is a corresponding arrow ending at F in α. Similarly, F sends only messages for which there is a corresponding arrow starting at F in α. Moreover, the ordering of the messages exchanged by F and its environment is as depicted in α.

Our remaining task is to capture the liveness requirements of the environment and of component F. In contrast to the decomposition of R's safety part the decomposition of R's liveness part into clearly separated responsibilities for F and its environment is, in general, impossible.

As an example, consider the MSCs LA and LB from Figure 7.7 (a) and (b).



(a) (b)

Figure 7.7.: Decomposable and non-decomposable liveness specifications

MSC LA implies the following liveness property: valuations of c and c' according to R_{LA} contain an infinite number of $c \triangleright m$ and $c' \triangleright m'$ messages. This is a property we can clearly decompose into responsibilities for X and Y; each component must produce its output infinitely often.

MSC LB only requires infinitely many occurrences of either $c \triangleright m$ *or* $c' \triangleright m'$. As a consequence, in executions where $c \triangleright m$ occurs infinitely often Y has no liveness responsibility at all. However, without further knowledge about X's complete output history, Y cannot determine by itself at any finite time whether or not X will contribute to the fulfillment of the liveness property.

This leaves us with two options. We can either add further information to the specification of one of the components such that the liveness properties of both components are fixed uniquely. One way to achieve this in the example above is to add the property that message $c \triangleright m$ may occur only finitely often. This fixes Y's liveness property: $c' \triangleright m'$ must occur infinitely often according to MSC LB.

Or, we can start with the strongest possible liveness constraints for both F and its environment; these constraints require all components to work individually towards fulfillment of the liveness property, independent of whether the contribution of the other components already suffices or not. Then, we can weaken either F's or the environment's liveness properties once new information, say, through refinement steps, becomes available.

To investigate the most general case (without further knowledge about the components) we follow the second option here. According to Section 6.3.2 we can decompose any system property (and thus, any interaction interface) into a canonic safety and liveness part. For an interaction interface R this decomposition yields the safety part

$$S_R.x.y \equiv \langle \forall t \in \mathbb{N} :: \langle \exists x' \in \vec{I}, y' \in \vec{O} :: x \downarrow t = x' \downarrow t \wedge y \downarrow t = y' \downarrow t \wedge R.(x' \oplus y') \rangle \rangle$$

The right-hand-side of S_R's definition is an expansion of the canonic safety property *prefc*.$\{z \in \overrightarrow{(I \cup O)} : R.z\}$. This allows us to obtain the canonic liveness part of R by means of the following definition:

$$L_R.x.y \equiv \neg(S_R.x.y) \vee R.(x \oplus y)$$

L_R is the liveness property that F and its environment must jointly fulfill. We define F's environment liveness assumption E_L and F's component liveness commitment F_L as the weakest relations (with respect to set inclusion) $E'_L, F'_L : \vec{I} \to (\vec{O} \to \mathbb{B})$ whose conjunction nontrivially implies L_R. More precisely, we require E'_L and F'_L to fulfill each of the following three conditions:

$$E'_L.x.y \wedge F'_L.x.y \Rightarrow L_R.x.y \tag{7.2}$$

$$\langle \forall x \in \vec{I} : \langle \exists y \in \vec{O} :: L_R.x.y \rangle : \langle \exists y \in \vec{O} :: F'_L.x.y \rangle \rangle \tag{7.3}$$

$$\langle \forall y \in \vec{O} : \langle \exists x \in \vec{I} :: L_R.x.y \rangle : \langle \exists x \in \vec{I} :: E'_L.x.y \rangle \rangle \tag{7.4}$$

According to Condition (7.2) E'_L and F'_L must together fulfill the system liveness property L_R. Condition (7.3) restricts F'_L such that it must produce at least one output history for every input history allowing fulfillment of L_R. In other words, F must produce an output history if there is a chance to fulfill L_R jointly with E. Condition (7.4) defines a similar restriction for E.

For $R \not\equiv$ false we can substitute L_R for both F'_L and E'_L to convince ourselves of the existence of a solution for these three conditions. Thus, F_L and E_L are well-defined as the weakest solutions for F'_L and E'_L.

These considerations show that there is freedom in assigning liveness properties to the individual components of a system. Resolving this freedom by fixing one particular assignment according to the conditions (7.2) through (7.4) is a design step.

This completes our decomposition of R into an A/C specification for component F. We have found relations E_S, E_L, F_S, and F_L such that we can define $F : \vec{I} \to (\vec{O} \to \mathbb{B})$ by a time guarded relational A/C specification

$$\begin{aligned} F.x.y \equiv \quad & (E_S.x.y \;\Rightarrow\; F_S.x.y) \\ & \wedge (E_S.x.y \wedge E_L.x.y \;\Rightarrow\; F_L.x.y) \end{aligned}$$

By definition, we have established the validity of

$$E_S.x.y \wedge F_S.x.y \wedge E_L.x.y \wedge F_L.x.y \Rightarrow R.(x \oplus y)$$

for all $x \in \vec{I}$ and $y \in \vec{O}$. Hence, the relational component specification obtained for F forces F to operate as expected in environments complying to R.

If we have derived the interaction interface R from an MSC α, then we can interpret the component safety and liveness commitments F_S and F_L as follows. If F produces an output message $ch \triangleright m$ at all, then there is a corresponding arrow whose source is F in α. If there were no such arrow, then $ch \triangleright m$ could not occur in executions that comply to α under the exact interpretation. As a consequence $R.x.y$ would be false for all output histories y of F in which $ch \triangleright m$ occurs at any finite time. F_S, whose validity depends on R's validity

on any finite prefix of inputs and outputs of F, would then also yield false. By a similar line of reasoning, we could show that F_S can only hold if F produces its output messages in the order specified by α. The requirement for time guardedness, which we have coded into the definition of F_S, is already given for interaction interfaces derived from MSCs (cf. Proposition 16).

The liveness commitment F_L of F asserts an at most finite delay before F sends a message on whose occurrence the overall liveness of the system depends. Otherwise F_L would violate the conditions (7.3) and (7.4).

Because of the duality between F and its environment we immediately obtain similar interpretations for the assumptions E_S and E_L.

According to these considerations the component F derived schematically from an MSC α is bound to producing the specified output if and only if the components constituting F's environment do not deviate from what α specifies for them.

Thus, the A/C specification format is helpful in determining the formal responsibilities defined by an MSC for the components appearing in it as the sources and destinations of messages. We exploit this benefit of the A/C format again in the next section, where we apply it to explain the consequences of property refinement steps on MSCs with respect to individual components.

7.3.4. MSC Refinement Revisited

In Chapter 5 we have studied four refinement notions for MSCs: binding references, property, message, and structural refinement. Given the decomposition of an MSC into individual component specifications according to Section 7.3.3 we can now investigate in more detail the effect of an MSC refinement on the components themselves. We demonstrate these effects by considering MSC property refinement as the representative refinement notion.

In Section 6.2 we have introduced the notion of specification property refinement by means of set inclusion. This induces a "natural" form of property refinement on interaction interfaces, because interaction interfaces are projections of system specifications. Thus, for given interaction interfaces $R_1, R_2 : \overrightarrow{(I \cup O)} \to \mathbb{B}$ we call R_1 a property refinement of R_2, if for all $x \in \vec{I}$ and $y \in \vec{O}$

$$R_1.(x \oplus y) \Rightarrow R_2.(x \oplus y)$$

holds.

Because property refinement of MSCs is a special case of specification property refinement (cf. Sections 5.3 and 6.2), and because MSC properties under the closed world semantics are subsets of regular MSC properties, we can conclude from $[\![\alpha]\!]_0 \subseteq [\![\beta]\!]_0$ that also $[\![\alpha]\!]_{CW} \subseteq$

$[\![\beta]\!]_{CW}$ holds. Hence, property refinement of MSCs indeed yields a property refinement of the derived interaction interface.

The separation of channels into the sets I and O in the definition of an interaction interface allows us to distinguish three forms of property refinements, depending on whether the refinement has an effect on I only, on O only, or on both:

1. I-stable refinements,
2. O-stable refinements,
3. joint refinements.

An I-stable refinement leaves the set of input histories, for which the component produces an output history, unchanged. It may only reduce the set of possible output histories for the same input history of the component. This corresponds to reducing the output-nondeterminism of the component.

Definition 39 (*I*-stable Refinement) Let $R_1, R_2 : \overrightarrow{(I \cup O)} \to \mathbb{B}$ be interaction interfaces with respect to the syntactic interface (I, O). We call R_1 an *I-stable refinement* of R_2 if for all $x \in \vec{I}$ and $y \in \vec{O}$ both

$$R_1.(x \oplus y) \Rightarrow R_2.(x \oplus y)$$

and

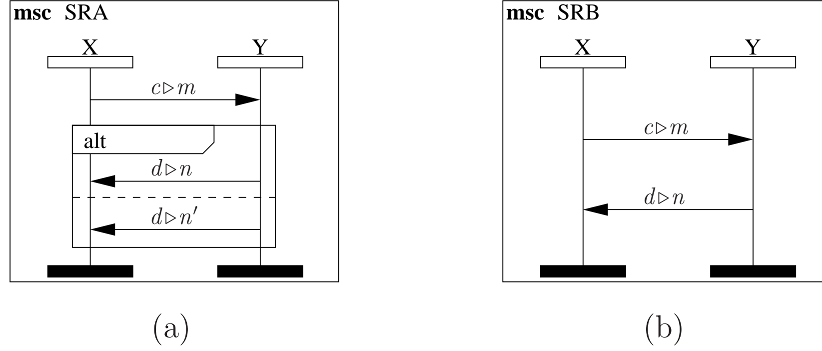
$$\{x \in \vec{I} : \langle \exists y \in \vec{O} :: R_1.(x \oplus y) \rangle\} = \{x \in \vec{I} : \langle \exists y \in \vec{O} :: R_2.(x \oplus y) \rangle\}$$

hold. □

Consider the MSCs *SRA* and *SRB* from Figures 7.8 (a) and (b). If we derive interaction interfaces from *SRA* and *SRB* with respect to $I = \{c\}$ and $O = \{d\}$, i.e. from the viewpoint of component Y, then R_{SRB} is an I-stable refinement of R_{SRA}.

In the step from *SRA* to *SRB* we have limited the choices of Y for producing a reaction to message $c \triangleright m$ from X. We have not, however, modified the set of inputs to which Y must react.

An O-stable refinement leaves the set of output histories the component can produce unchanged. It may only reduce the set of input histories for which the component produces the same output history. This corresponds to reducing the input-nondeterminism of the component.



(a) (b)

Figure 7.8.: Stable refinements

Definition 40 (*O*-stable Refinement) Let $R_1, R_2 : \overrightarrow{(I \cup O)} \to \mathbb{B}$ be interaction interfaces with respect to the syntactic interface (I, O). We call R_1 an *O-stable refinement* of R_2 if for all $x \in \vec{I}$ and $y \in \vec{O}$ both

$$R_1.(x \oplus y) \Rightarrow R_2.(x \oplus y)$$

and

$$\{y \in \vec{O} : \langle \exists x \in \vec{I} :: R_1.(x \oplus y) \rangle\} = \{y \in \vec{O} : \langle \exists x \in \vec{I} :: R_2.(x \oplus y) \rangle\}$$

hold. □

Consider again the MSCs *SRA* and *SRB* from the example above. If we now derive interaction interfaces from *SRA* and *SRB* with respect to $I = \{d\}$ and $O = \{c\}$, i.e. from the viewpoint of component X, then R_{SRB} is now an O-stable refinement of R_{SRA}.

From a methodical point of view the use of O-stable refinements – which are, in fact, I-stable refinements of the environment – means *weakening* the environment assumptions E_S or E_L, whereas an I-stable refinement means *strengthening* the component's commitments F_S or F_L, as we can easily verify by considering the A/C specifications derived from a corresponding interaction interface.

As the examples above suggest, refinement steps (on MSCs) leading to I-stable or O-stable refinements of the corresponding interaction interface remove alternatives from the MSC; all arrows in the removed alternative point into the same direction: either away from or towards the component under consideration, but not both.

A joint refinement is neither I-stable, nor O-stable. It influences both the sets of input histories a component must react to, and the sets of output histories the component is committed to produce.

Definition 41 (Joint Refinement) Let $R_1, R_2 : \overrightarrow{(I \cup O)} \to \mathbb{B}$ be interaction interfaces with respect to the syntactic interface (I, O). We call R_1 a *joint refinement* of R_2 if for all $x \in \vec{I}$ and $y \in \vec{O}$

$$R_1.(x \oplus y) \Rightarrow R_2.(x \oplus y)$$

holds, and R_1 is neither an I-stable, nor an O-stable refinement of R_2. □

Consider the MSCs *JRA* and *JRB* from Figures 7.9 (a) and (b). Let (I, O) be defined as $(\{c\}, \{d\})$. Then R_{JRB} is a joint refinement of R_{JRA}.

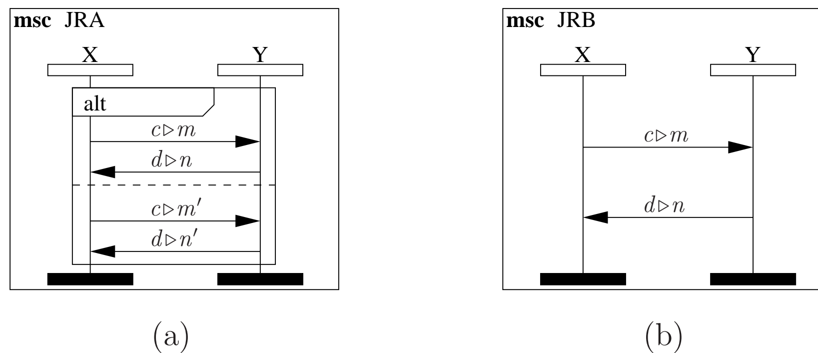


Figure 7.9.: Joint refinement

A joint refinement results from removing alternatives in MSCs where the arrows within the alternative point both towards and away from the component under consideration.

We conclude that property refinement of MSCs in general has consequences on all components involved in an interaction, even in the case of I- and O-stable refinements.

7.3.5. Discussion

In Section 7.3.3 we have shown how to transform an MSC specification schematically into an individual component specification for any of the components occurring in the MSC.

This approach has several benefits:

- it is purely schematic up to the possible weakening of the liveness relations E_L and F_L.
- the A/C format clearly identifies and separates the environment's from the component's responsibilities; an application of this separation is our discussion of the consequences of MSC property refinement on individual components in Section 7.3.4.
- the derivation is based on interaction interfaces, instead of on the syntactic form of MSCs. Therefore, it is independent of the concrete MSC syntax we employ; in particular, it covers the entire MSC dialect we have introduced in Chapter 4.

However, because the approach *is* so schematic, it also has the drawback of providing only implicit specifications of assumptions and commitments. This need not be problematic

if a concrete component specification is not subject to further manual manipulation or "paper-and-pencil" proofs. It is not particularly helpful, however, if we have to produce executable specifications or code for the component under consideration. We cannot, for instance, derive sensible finite state machine specifications schematically from interaction interfaces or A/C specifications.

This observation motivates the approach we pursue in the next section. There, we construct finite state machines directly from MSCs in their syntactic form.

7.4. From MSCs to Automaton Specifications

The A/C specifications we have associated with MSCs in the preceding section are a powerful tool for studying the relationship between MSCs and individual component specifications. As an application of this we have investigated in much more detail than before the implications of property refinement steps, which we carry out on an entire MSC, with respect to the individual components appearing in the MSC.

By intention, an A/C specification yields a black-box view: it only considers the relation between input and output histories of the component under consideration. This view is particularly useful during the requirements capture process, because it focuses on *what* the component must achieve, abstracting away from the details of *how* the component establishes its result.

At certain stages during the development process we are, however, specifically interested in *how* a component achieves the *what*. This is certainly true when it comes to implementing the component in a state-based programming language, such as C, C++, or Java. Each of these languages is state-oriented in the sense that programs in these languages operate on a certain control and data state-space, often partitioned among the processes or objects representing the component under consideration at runtime.

Besides the "final" implementation there are further reasons why we are interested in glass-box specifications of the component under consideration. Even during requirements capture we might want to produce early prototypes of the component to demonstrate the feasibility of our development approach. If we aim at employing automatic verification techniques, such as model checking, then we definitely have to come up with a state-based model for the component. Furthermore, we can also derive a state-based model for the component's environment and use this model as a test-driver for the component itself.

Accepting the importance of state-oriented component specifications as an integral part of the development process, the following question arises: can we derive such specifications equally schematically from MSCs as we did with A/C specifications?

The approach we pursue in the remainder of this chapter allows us to answer this question positively. As a consequence, we establish a seamless transformation from MSCs – as a

means for requirements specification – to (prototypic) state-oriented component implementations.

A first idea for establishing this result might be to derive an A/C specification from a set of given MSCs, and to convert this A/C specification into a corresponding finite state machine. This second step is, however, not feasible in general. Automata with finite state and transition sets are hard to construct from A/C specifications. The problem is that determining the component's "current" control state from (a finite prefix of) its interaction behavior is, in general, impossible.

Therefore, we follow a different approach in this section. Instead of trying to perform transformations on the semantic domain to yield an automaton specification, we operate purely syntactically. We present a translation procedure that takes as main input the given MSCs in their syntactic form. Step by step the procedure syntactically turns each MSC into a segment of the finite state automaton we want to construct. By construction the resulting automaton has a finite number of states. Moreover, the syntactic transformations we suggest have another advantage: they are independent of the underlying semantic model of the resulting automata. This means we could, for instance, apply the same transformation scheme if we interpreted both the MSCs and the resulting automata in a time and message synchronous way – as compared to the message asynchronous semantics we have selected in Chapter 4.

To this end, in Section 7.4.1 we fix a specific automaton syntax that is general enough for modeling reactive components within our system model. In Chapter 3 we have already discussed several variants of automaton models. The one we use in this section is a mixture of a Mealy machine and an ω-automaton. We avoid the more elaborate concepts (including hierarchic states and preemption) we have reviewed in Chapter 3 in connection with statecharts and ROOMCharts in favor of a simpler semantic treatment. However, we can map most of these concepts to the simpler ones we introduce here, and thus do not experience a significant loss of generality. In Section 7.4.2 we present the transformation procedure in detail. In particular, we restrict our attention to the subset of the MSC dialect from Chapter 4 we can translate directly into the automaton model of Section 7.4.1. We describe extensions to this subset, and how to translate them into corresponding automata in Section 7.4.4. In Section 7.5 we compare our approach with related ones from the literature. Section 7.6 contains a summary.

7.4.1. Automaton Syntax and Semantics

Our first step towards the automatic derivation of finite state machines from MSCs is to fix the syntax and semantics of the automata we want to end up with. As we have explained in Chapter 3 there is a plethora of automaton models for reactive systems in the literature, each with its own strengths and weaknesses. Instead of limiting the discussion of this section to one specific automaton model from Chapter 3, say statecharts or ROOMCharts,

we have chosen a mixture of Mealy machines and ω-automata. The syntax of our model is a subset of what most other automaton models provide; this makes our results applicable to a wide range of state-oriented specification techniques. Moreover, the restricted syntax helps us to focus on the key aspects of the translation procedure we present in Section 7.4.2. We discuss extensions of our automaton model together with corresponding extensions for the MSCs we accept as input for the translation procedure in Section 7.4.4.

Recall from Chapter 3 that the general form of a transition in a Mealy machine is as depicted in Figure 7.10. The transition starts in a state s_k, and ends at a target state s_l; the transition's label is of the form i/o, where i denotes the input read by the machine, and o denotes the output written by the machine when it takes the transition.

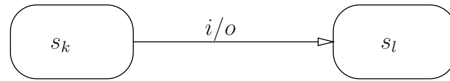


Figure 7.10.: General form of transition in Mealy machine

In our automaton model we also capture spontaneous input/output transitions, i.e. transitions where either i or o equals ϵ, indicating that the machine reads no input or writes no output, respectively, when taking the transition.

According to the system model we have introduced in Section 4.2, each component can have multiple input and output channels. To indicate the channel on which a certain message occurs, we use the same syntax as in our MSCs: $c \triangleright m$ indicates the occurrence of message m on channel c.

Graphically, we borrow the representation for (initial) states and transitions from Mealy machines. Instead of the symbol ϵ we also use "-" to indicate the absence of input or output on a transition label; instead of "-/-" we simply write "-" or "ϵ". To stress the separation between the role of input and output messages on transition labels further, we use the symbols "?" and "!" as prefixes of input and output messages, respectively. This allows us to use "$?c \triangleright m$" and "$!c \triangleright m$" as shorthands for "$c \triangleright m/$-" and "-$/c \triangleright m$", respectively.

Definition 42 (Automaton Syntax) Let S and S_0 be finite sets of states with $S_0 \subseteq S$, and let $\hat{I}_\epsilon, \hat{O}_\epsilon \subseteq \langle \text{MSG} \rangle \uplus \{\epsilon\}$ be sets of input and output message specifications, respectively, such that $\hat{I}_\epsilon \cap \hat{O}_\epsilon = \{\epsilon\}$ holds; furthermore, let

$$\delta : \hat{I}_\epsilon \times \hat{O}_\epsilon \times S \rightarrow \mathcal{P}(S)$$

be a state transition function. Then we call the quintuple

$$A = (S, \hat{I}_\epsilon, \hat{O}_\epsilon, S_0, \delta)$$

a *(nondeterministic) syntactic automaton specification*[3], or *automaton* for short. □

[3]The automaton specification is syntactic in the following sense: it contains no mapping from the state transition function to the system model. Later in this section we associate safety and liveness properties with the automaton syntax introduced here; this association constitutes the semantics definition for automata.

As a simple example, consider the automaton from Figure 7.11, which we could use to represent the state-oriented behavior of the component *Control* in the CLS example from the introduction to this chapter (cf. Figures 7.2 (a) and (b)). For this automaton we obtain the state set $S = \{UNLD, LCKD, s1, s2, s3, s4\}$, the set of initial states $S_0 = \{UNLD, LCKD\}$, the set of input message specifications $\hat{I}_\epsilon = \{\epsilon, kc \triangleright lck, kc \triangleright unlck, lc \triangleright rdy, rc \triangleright rdy\}$, and the set of output message specifications $\hat{O}_\epsilon = \{\epsilon, cl \triangleright up, cl \triangleright dwn, cr \triangleright up, cr \triangleright dwn\}$; the state transition function δ is as the figure specifies.

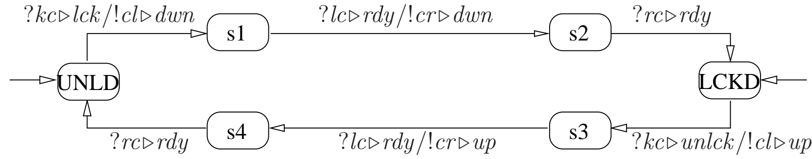


Figure 7.11.: Automaton specification

The transformation procedure we suggest below is purely syntactic; in particular, it is independent of the semantics we associate with either MSCs or automata – as long as we use the same semantic model for interpreting both. Nevertheless, in the following paragraphs we introduce an automaton semantics. Our motivation for this is two-fold. First, we demonstrate that the automata indeed fit into our semantic model. Second, we can relate the result of the translation procedure constructively with the semantics of the MSCs we have started out with.

Clearly, there is a plethora of possibilities for defining a semantics for the automata whose syntax we have just introduced. The one we chose here is particularly simple. Intuitively, a component operating according to this semantics in each step has the opportunity of reading a single input message, writing a single output message, and changing state; if there is neither an input the component can read, nor an output the component can write, then the component remains in its current state.

We separate introduction of the automaton semantics into the automaton's safety and liveness properties. We fix the safety properties by unfolding the transition relation "over time", and use arbitrary liveness properties to constrain the liveness of the component corresponding to the automaton.

Definition 43 (Automaton Safety Property) Let C and S be the sets of system channels and states, respectively. By F we denote the system component whose behavior the automaton defines (F is an element of the set P of system components). Let $A = (S_A, \hat{I}_\epsilon, \hat{O}_\epsilon, S_0, \delta)$ be an automaton specification such that $S_A \subseteq S|_F$ holds. $S|_F$ denotes the restriction of the overall system state space S with respect to F. Let, furthermore, $I, O \subseteq C$ hold with $I = \{c \in C : \langle \exists m \in M :: c \triangleright m \in \hat{I}_\epsilon \rangle\}$ and $O = \{c \in C : \langle \exists m \in M :: c \triangleright m \in \hat{O}_\epsilon \rangle\}$.

Then we define the safety property $AS.A$ of automaton A as follows:

$$
\begin{aligned}
AS.A \stackrel{\text{def}}{=} \{(\psi,\sigma) \in (\tilde{C} \times S)^\infty :\\
&\sigma|_F.0 \in S_0\\
&\wedge \langle \forall t \in \mathbb{N} : t \geq 1 :\\
&\quad (\sigma|_F.t = \sigma|_F.(t-1) \wedge \langle \forall c \in I, c' \in O :: \psi.(t-1).c = \text{<>} \wedge \psi.t.c' = \text{<>}\rangle)\\
&\quad \vee \langle \exists (i,o) \in \hat{I}_\epsilon \times \hat{O}_\epsilon ::\\
&\qquad \sigma|_F.t \in \delta(i,o,\sigma|_F.(t-1))\\
&\qquad \wedge (i \neq \epsilon \Rightarrow \langle \exists c,m : i = c \triangleright m : m \in \psi.(t-1).c\rangle)\\
&\qquad \wedge (i = \epsilon \Rightarrow \langle \forall c \in I :: \psi.(t-1).c = \text{<>}\rangle)\\
&\qquad \wedge (o \neq \epsilon \Rightarrow \langle \exists c,m : o = c \triangleright m : m \in \psi.t.c\rangle)\\
&\qquad \wedge (o = \epsilon \Rightarrow \langle \forall c \in O :: \psi.t.c = \text{<>}\rangle)\rangle\rangle\}
\end{aligned}
$$

□

In this definition, we denote by $\sigma|_F$ the projection of the overall stream of system states onto F's contribution. Thus, every execution in which the component corresponding to the automaton A partakes starts in an initial state of the automaton ($\sigma|_F.0 \in S_0$). At any time $t \geq 1$ the component either

- has remained in its previous state (i.e. $\sigma|_F.t = \sigma|_F.(t-1)$), if no input message occurs on any input channel at time $(t-1)$, and no output message occurs on any output channel at time t, or

- it has taken a transition from state $\sigma|_F.(t-1)$ to $\sigma|_F.t$, labeled i/o in δ (i.e. $\sigma|_F.t \in \delta(i,o,\sigma|_F.(t-1))$), and the input at time $t-1$ equals i, and the output at time t equals o.

Compared to, say, statecharts this is a very simple, operational model for component behavior. At any time the component consumes at most one input message, changes state, and produces at most one output message. As a result, the specification of certain properties, such as the lack of causality between messages, requires more elaborate automaton specifications. We will come back to this point in Section 7.4.4.

Now we turn to the liveness properties of our automaton specifications; as the "role-model" for liveness specifications with automata we use the ones provided by ω-automata. Recall from Chapter 3 that the liveness properties we can associate with an ω-automaton are defined via sets of states, whose elements must occur infinitely often during any execution of the system. Here, we take a more liberal approach and provide the opportunity to assign any system liveness property $AL.A \subseteq (\tilde{C} \times S)^\infty$ as the liveness property of the automaton A.

An example for a liveness property for component *Control*, whose safety property is given through the automaton specification from Figure 7.11 is

$$AL.A_{Control} = \{\varphi \in (\tilde{C} \times S)^{\infty} : \#(\{UNLD\} \copyright \pi_2(\varphi)) = \infty\}$$

which expresses that state *UNLD* must occur infinitely often in any execution. Another example for a liveness property is the MSC specification

$$kc \triangleright lck \ \mapsto \ (cl \triangleright dwn \ \sim \ cr \triangleright dwn)$$

which requires every $kc \triangleright lck$ message to be eventually followed by corresponding "down" messages to both motors of the CLS. The first example is formulated over the internal state of component *Control*, whereas the second one only refers to the externally visible behavior of *Control*. We obtain the overall automaton semantics as the conjunction of the automaton safety property and an automaton liveness property:

Definition 44 (Automaton Semantics) Let A be an automaton specification, and let $AL.A \subseteq (\tilde{C} \times S)^{\infty}$ be a system liveness property. Then we define the semantics of A by

$$[\![A]\!] = \{\varphi \in (\tilde{C} \times S)^{\infty} : \varphi \in AS.A \wedge \varphi \in AL.A\}$$

□

Without proof we note that standard (nondeterministic) finite state machines (in the sense of [HU90]), as well as Mealy machines, and ω-automata are special cases of our automaton model.

In summary, we have defined a simple automaton model for reactive components in the preceding paragraphs. We have separated the semantics definition into the safety and the liveness part of the automaton. The safety part is, in essence, defined by an expansion of the state transition function over time. In the following section we use this automaton model as the target for our transformation procedure from MSCs to finite state machines.

7.4.2. Translation Scheme

Our next aim is to construct an automaton specification for one of the components appearing in a given set of MSCs. To this end, in this section, we present a transformation procedure that takes as input a set of MSCs, and, through a sequence of syntactic manipulations, turns them into an automaton in the syntax as we have introduced it above.

Clearly, we want the semantics of the automaton we thus obtain to be "compatible" with the one of the given MSCs, in the sense that the automaton "implements" the interaction behavior specified by the MSCs for the component under consideration. This notion of compatibility strongly depends on the semantics of the MSCs and on the semantics of the target automaton model. We will show later in this section that, indeed, if we associate the semantics of the preceding section with the automaton obtained as the output of the

transformation scheme, then the automaton's behavior is a subset of the behavior we associate with the set of MSCs according to the closed-world semantics.

The basic idea behind our approach is as follows: we view MSCs as "unfoldings" of (partial) paths through the automaton corresponding to the component under consideration. Our task is, then, to "reconstruct" the complete automaton from sets of such partial paths as given by the set of MSCs we start out with.

Consider, as an example, the MSCs *slocking* and *sunlocking* from Figures 7.12 (a) and (b). We can view each of them as one specific path through the automaton for component *Control* from Figure 7.13. MSC *slocking* "takes" the automaton from state *UNLD* to state *LCKD*, whereas MSC *sunlocking* describes the transition path from state *LCKD* back to state *UNLD*.

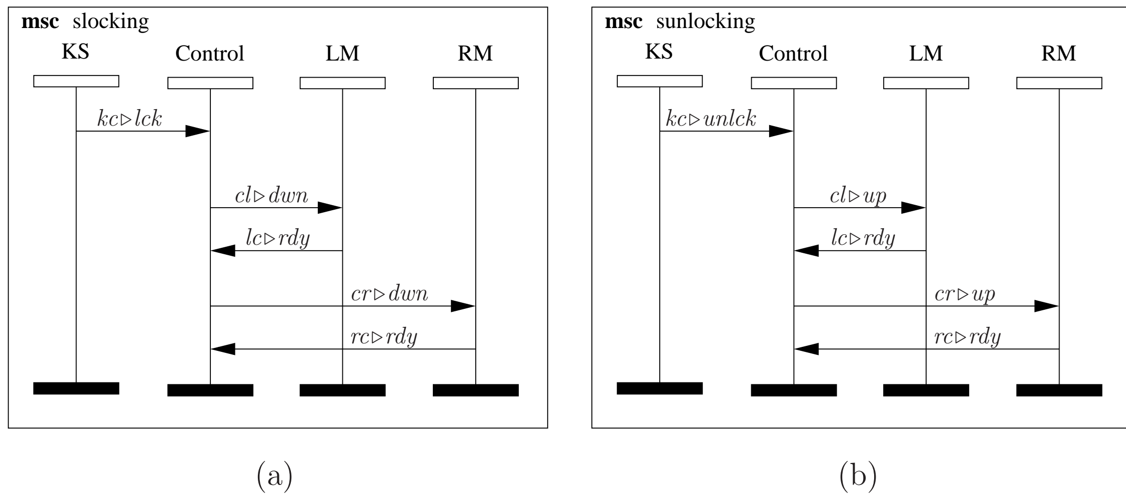


Figure 7.12.: Specification of the (simplified) "locking" and "unlocking" use case

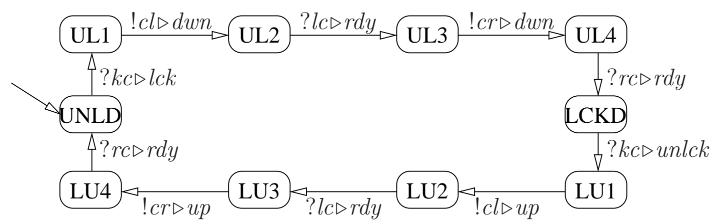


Figure 7.13.: Automaton for component *Control*

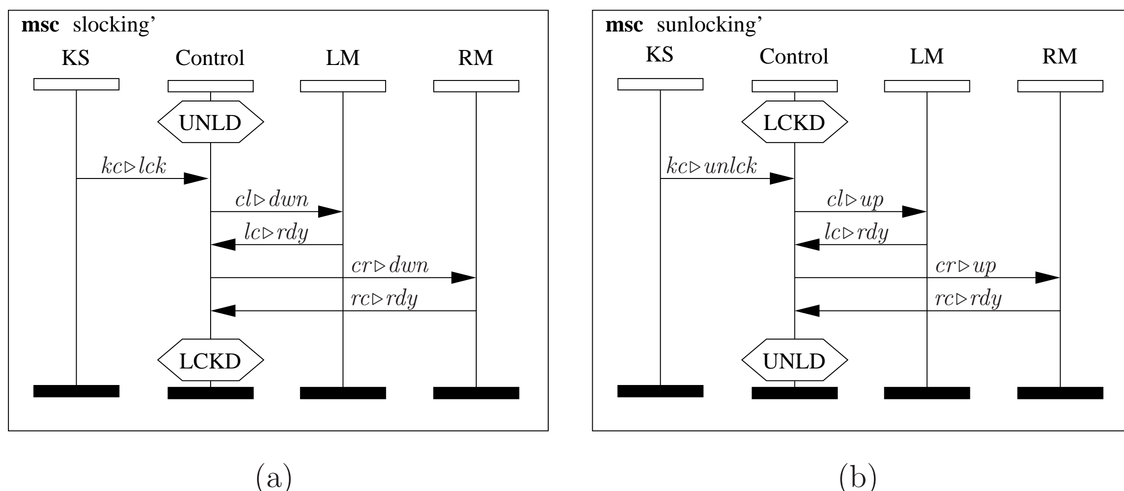
Thus, the main problem we have to solve, if we have given a set of MSCs, and want to construct an automaton such as the one from Figure 7.13, is to find an appropriate set of states, as well as an appropriate state transition function that establishes the link between the different execution paths represented by the MSCs.

The example MSCs *slocking* and *sunlocking* already show that finding an appropriate state set and transition function is, in general, nontrivial: neither of the MSCs indicates what

state the component *Control* assumes after the respective interaction pattern has occurred. We could, for instance, use the same automaton state as the start and end state of both use cases. This would lead to a slightly different automaton specification than that of Figure 7.13; it would allow component *Control* to initiate a locking process even if the CLS is already locked.

One of the challenges we face, therefore, is to find not only arbitrary state sets and state transition functions, but also ones yielding manageable and sensible automata. The solution we follow here is to use guards within MSCs as a means for guiding the construction process. Each guard in the MSC will define one control state of the resulting automaton.

Consider, as an example, the MSCs *slocking'* and *sunlocking'* from Figures 7.14 (a) and (b). We use the guards labeled *UNLD* and *LCKD* to express in what control state the component *Control* resides after partaking in the respective use case.



(a) (b)

Figure 7.14.: Specification of the "locking" and "unlocking" use case with guards

Below, we will see that taking guards as state labels into account can increase the quality of the resulting automaton (with respect to a lower number of states and transitions) considerably. In fact, the automaton from Figure 7.13 is what our transformation procedure yields if applied to the MSCs of Figure 7.14.

In the remainder of this section we present our transformation procedure from MSCs to automaton specifications in detail; the structure of our presentation is as follows. First, we define the subset of the MSC syntax for which we give the transformation explicitly. This syntactic subset suffices to express guarded message exchange, alternatives, and repetition. The transformation of more elaborate syntactic constructs appears in Section 7.4.4. Second, we give a roadmap for the transformation procedure to convey a first idea of the four steps we perform to obtain the automaton from a set of MSCs. Third, we discuss a few preliminaries we have to establish before the transformation can succeed. Fourth, we present each of the phases of the transformation procedure, in detail. We illustrate our

approach by transforming the MSCs *slocking'* and *sunlocking'* into an automaton for the component *Control* along the way. Another example appears in Section 7.4.3, where we revisit the Abracadabra-protocol.

Restricted MSC Syntax

The syntax for which we define the transformation of MSCs into automata is a small subset of the one we have introduced in Chapter 4. The reason for this restriction is twofold. First, we want to focus on the key aspects of the transformation, not on syntactic technicalities. Second, although the automaton model of Section 7.4.1 provides no direct syntactic support for expressing the advanced "features" of our MSC dialect, such as preemption, interleaving, join, and guarded repetition, we will see in Section 7.4.4 that for most of these constructs simple "macro-expansions", based on the restricted MSC syntax, exist.

An MSC we accept as input for the transformation procedure may contain – besides component axes in the graphical representation – possibly empty sequences of messages and guards only. For simplicity, we allow guards to refer only to the control states of the components. We assume further that each such guard is represented by a unique name. We use these names to refer to the components' control states within MSCs.

As an example, consider the CLS again. The overall state space S is the cross product of the individual component state spaces S_{KS}, $S_{Control}$, S_{LM}, and S_{RM}:

$$S = S_{KS} \times S_{Control} \times S_{LM} \times S_{RM}$$

We treat control states as part of each component's overall state space, and write

$$S_F = S_F^{PC} \times S_F^{D}$$

to denote the separation of component F's state space into the control state space S_F^{PC} and the data state space S_F^{D}, respectively.

In our example, let us assume that the control state space of component *Control* includes the two states *LCKD* and *UNLD*, i.e. $\{LCKD, UNLD\} \subseteq S_{Control}^{PC}$ holds. Then the guards occurring in the MSCs *slocking'* and *sunlocking'* of Figure 7.14 (a) and (b) express that component *Control* is in the respective control state before and after the depicted interaction pattern occurs.

We use the interpretation of guards as control states also to express alternatives and repetition in our restricted MSC syntax:

- MSCs starting with the same guard-label express alternative interaction patterns,
- an MSC starting *and* ending with the same guard-label expresses the repetition of the interaction pattern between the two guards.

Thus, the two MSC terms in restricted syntax:

$$\textbf{msc } X = (g_1 : c \triangleright m) \; ; \; (g_1 : \textbf{empty})$$
$$\textbf{msc } Y = (g_1 : \textbf{empty}) \; ; \; (g_2 : \textbf{empty})$$

together represent the same "black-box" behavior as the MSC term

$$\textbf{msc } Z = (c \triangleright m) \uparrow_{<*>}$$

in our full MSC dialect with inline expressions.

Under this interpretation, the MSCs *slocking'* and *sunlocking'*, for instance, represent that component *Control* switches between the two control states *UNLD* and *LCKD*, triggered by the receipt of the corresponding messages from component *KS*.

In this example, local guards represent control states of component *Control*. For MSCs containing non-local guards we assume the existence of a surjective mapping from non-local guards to local guards for each of the components covered by the non-local guards. This reduces these components' dependency on the states of their environment.

Roadmap

The roadmap we follow with our transformation procedure is as depicted in Figure 7.15. It consists of four successive phases:

1. **Projection:**
   After having selected the component for which we want to construct an automaton specification we project each of the given MSCs onto this component.

2. **Normalization:**
   We determine the transition-path segments defined by the projected MSCs, according to the guards appearing in the MSCs; if necessary, we add appropriate guards at the beginning and at the end of the projected MSCs.

3. **Transformation into an Automaton:**
   We turn every message arrow that appears in an MSC into a transition of the automaton; if necessary, we add intermediate states to link the transitions, such that they correspond to a sequence of messages within a normalized MSC.

4. **Optimization:**
   Because the resulting automata are, in essence, standard nondeterministic finite state machines, we can apply any of the well-known optimization algorithms from automata theory, say, to minimize the number of states or transitions.

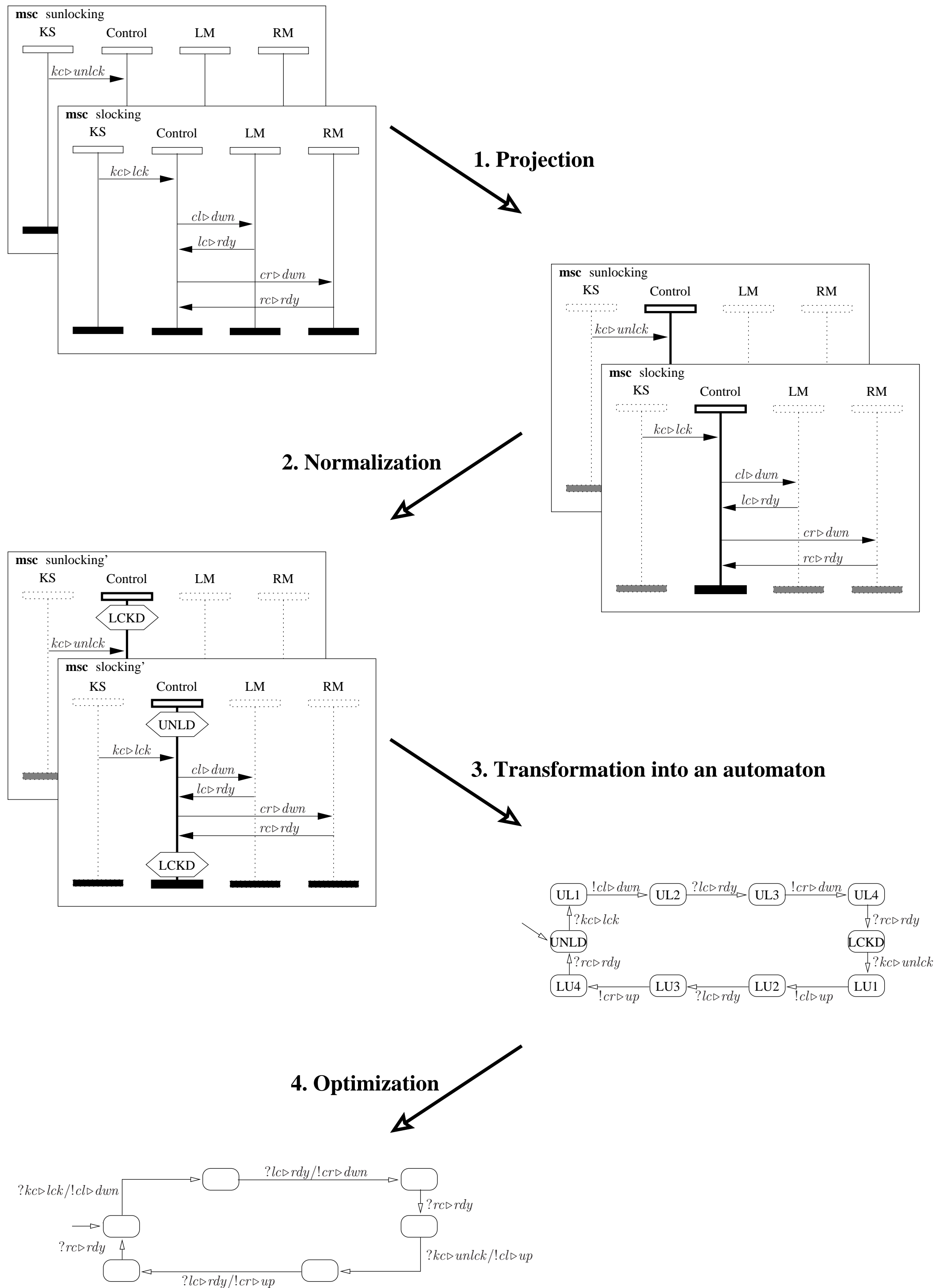


Figure 7.15.: Roadmap for the transformation procedure

Next, we will discuss each of these phases in detail.

Preliminaries

To prepare and simplify our presentation of the transformation procedure, we first mention a few prerequisites, as well as some technical assumptions we make.

Before we can apply the procedure, we have to determine the component for which we want to construct an automaton. In the following we call this component F. We assume given a set of MSCs describing F's interaction behavior. From this set of MSCs we construct an automaton for F.

Our automaton model requires the specification of a set $S_0 \subseteq S_F$ of initial states. For the purposes of the transformation procedure, we assume S_0 to be a singleton: $S_0 = \{s_0\}$; the initial state s_0 be given before we apply the procedure. If the MSCs are already annotated with guards, then it often makes sense to pick one of these as the initial state s_0. However, in the general case, we cannot determine s_0 canonically from the given MSCs. Therefore, we leave it to the designer to decide which of the component's states should "be" s_0.

At several points during the transformation we need to introduce fresh guards (or control states) into the MSCs (or automata) produced as intermediate results. Such fresh guards (or control states) occur nowhere else in the specification. Therefore, we assume we can always select a fresh element from S_F^{PC} whenever we need it; we do so only finitely many times during the transformation, which ensures that S_F^{PC} remains finite. Moreover, we assume that S_F^{PC} contains all state labels corresponding to guard labels within the given MSCs.

With these preliminaries in place, we now turn to the description of the four transformation phases *projection*, *normalization*, *transformation into an automaton*, and *optimization*.

Projection

The first step of the transformation procedure is to project all MSCs onto the component we are interested in (component F). This means removing all other instance axes, as well as message arrows that neither start nor end at the instance axis of component F.

The result of this is a fresh set of partial MSCs; each of these MSCs has precisely one component axis whose label is F. The MSCs are partial in the following sense: we can determine the source and destination of a message only through the labels of the message arrow; the axes of components other than F are absent.

If we perform the projection of the MSCs *slocking'* and *sunlocking'* onto component *Control*, we obtain (partial) MSCs as depicted in Figures 7.16 (a) and (b).

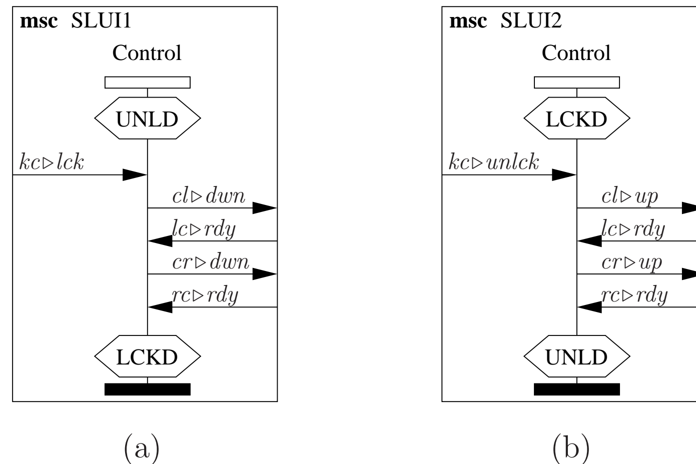


Figure 7.16.: Projected and Normalized MSCs for component *Control*

Normalization

As we have mentioned above, we adopt the view that every one of the projected MSCs describes one particular state/transition path within the automaton we construct. This path leads from a "start" state, i.e. the control state assumed by the component before it partakes in the interaction pattern, to an "end" state, i.e. the control state assumed by the component after the interaction has occurred.

Our next step, therefore, is to assign start and end states to each of the (partial) MSCs resulting from the projection according to the preceding transformation phase.

We call (partial) MSCs that both start and end with a guard, and that have no other guards in between *MSCs in normal form*; *SLUI1* and *SLUI2* are examples of such MSCs.

MSCs in normal form are particularly helpful in our transformation procedure, because they make explicit the start and end states of the transition path we aim at.

Clearly, the MSCs resulting from the projection need not be in normal form right away. In general, there may be no guards at all; the MSCs may also contain an arbitrary number of guards between occurrences of message arrows. If we were to perform the projection of the MSC *slocking"* (cf. Figure 7.17) onto the component *RM*, say, then we would obtain a partial MSC having neither a start guard, nor an end guard; instead, it has a guard (labeled *INTER*) between the message receipt and the corresponding reply.

How do we normalize MSCs without guards at their beginning or at their end? Because the developer of the MSC has specified no constraint at the state of the component, we have a plethora of options. We mention four of them to illustrate the design decisions we face.

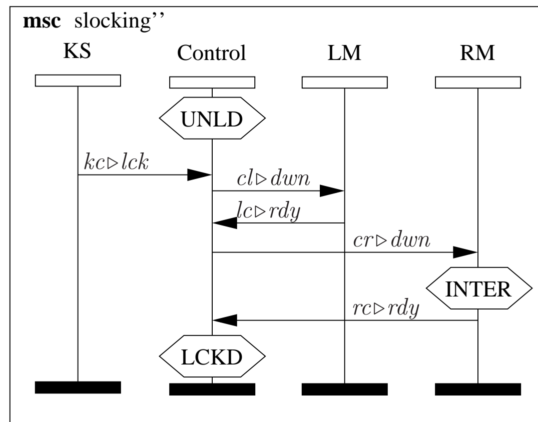


Figure 7.17.: Component *RM* without start and end guard

If the start guard is missing, we could either

1.a) add the given initial state instead, or

1.b) produce copies of the MSC such that for every control state of the component there is one copy with this state as the start guard.

According to the first alternative the interaction pattern corresponding to the MSC is "reachable", i.e. can occur, at least once. There is a path through the automaton under construction that starts in the initial state and that corresponds to the interaction pattern.

The second alternative results in a component specification where the interaction pattern under consideration may start in any control state; accordingly, the interaction pattern potentially preempts any other behavior displayed by the component.

If the end guard is missing, we could either

2.a) add the given initial state instead, or

2.b) add any control state without an outgoing transition, i.e. a trap state.

Here, the first alternative allows the component to continue to react according to some (other) given MSC. The second alternative treats the interaction patterns as "one-way" paths, from which there is no exit.

Clearly, there is a lot of other advantageous possibilities in concrete applications. Here, we select the two options 1.a) and 2.a), and add the given initial state, whenever a guard is missing. This is in line even with the weakest of our MSC interpretations, i.e. the existential interpretation, which requires that the scenario represented by an MSC should

at least have the potential for occurring; moreover, other scenarios can occur after the one under consideration has finished.

Now we assume that all MSCs have both a start and an end guard. The final step of the normalization phase is to split MSCs with more than two guards at any of the intermediate ones. We replace each such MSC by two new ones; the first of these is the original MSC up to (and including) the guard at which we split, the second one is the original MSC from this guard on. We repeat the splitting process until all MSCs have precisely two guards: their start and their end guard.

Figures 7.18 (b) and (c) show the result of applying the last step to the MSC fragment of Figure 7.18 (a).

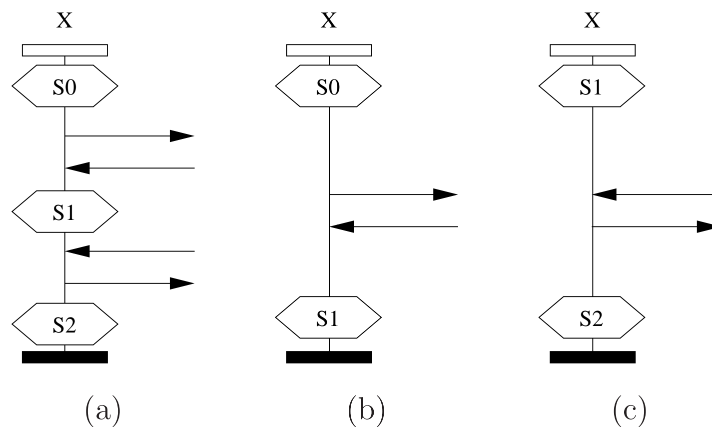


Figure 7.18.: MSCs with intermediate guard (a) and in normal form (b), (c)

Transformation into an Automaton

So far, we have extracted the relevant part of the MSCs for the component we are interested in (during the projection phase), and have fixed the start- and end-points of the transition paths we associate with each (partial) MSC (during the normalization phase).

Now we switch from the MSC syntax to the one of the target automata. In fact, we are already almost there. Our remaining task is to add intermediate states and transitions between any two control states that correspond to the start and end guards of the normalized MSCs.

To illustrate how close the MSCs in normal form are to a coarse-grained automaton description, and to establish a link between the semantics of the MSCs and the semantics of the resulting automaton, we introduce the notion of an *MSC automaton*. This is a finite state machine, whose states are the start and end guards of the normalized MSCs, and whose transition labels are MSC names. Intuitively, there is a transition labeled X from state s_0 to state s_1 in an MSC automaton, if and only if there is a normalized MSC named X in the MSC document, such that X starts with guard s_0 and ends with guard s_1.

Definition 45 (MSC Automaton) We consider an MSC document that contains only MSCs in the restricted syntax and in normal form for a component F in the set of all components P. Let C and S be the set of all channels, and component states, respectively, of the system. Let S_F be the set of control states of component F. S_F includes, in particular, all start and end guards appearing in the MSC document. $s_0 \in S_F$ is the initial state given as an input to the transformation procedure. Let $\Sigma \subseteq \langle \text{MSCNAME} \rangle$ be the set of names of the MSCs within the MSC document. We call a quadruple

$$A_F^{MSC} = (S_F, \Sigma, \delta, s_0)$$

an *MSC automaton* for F, if

$$\delta : S_F \times \Sigma \rightarrow \mathcal{P}(S_F)$$

is a state transition function, such that

$$s' \in \delta(s, X)$$

holds for $s', s \in S_F$ and $X \in \Sigma$ if and only if there is an MSC with name X, start guard s, and end guard s' in the MSC document. □

The semantics of an MSC automaton is the set of system executions where the component F displays the interaction behavior as defined by the MSCs in the document, and assumes

- the control state corresponding to the start guard *before* it partakes in the interaction pattern,
- the control state corresponding to the end guard *after* the interaction pattern is finished.

Between the time points fixed by the start and end guards the MSC labeling the transition in the MSC automaton describes F's behavior. Formally, we define:

Definition 46 (MSC Automaton Semantics) Let $A_F^{MSC} = (S_F, \Sigma, \delta, s_0)$ be an MSC automaton for component F. Let C and S be the sets of system channels and states, respectively. We define the semantics of A_F^{MSC}, which we denote by $[\![A_F^{MSC}]\!]$, as follows:

$$\begin{aligned}
[\![A_F^{MSC}]\!] \stackrel{\text{def}}{=} \{(\psi, \sigma) \in (\tilde{C} \times S)^\infty :\ & \\
& \sigma|_F.0 = s_0 \\
& \wedge \langle \exists T \in \mathbb{N}^\infty : T.0 = 0 : \\
& \quad \langle \forall t \in \mathbb{N} : t \geq 1 : \\
& \quad\quad \langle \exists X \in \Sigma :: \\
& \quad\quad\quad \sigma|_F.(T.t) \in \delta(\sigma|_F.(T.(t-1)), X) \\
& \quad\quad\quad \wedge ((\psi, \sigma), T.t) \in [\![\rightarrow X]\!]_{T.(t-1), CW} \rangle\rangle\rangle\}
\end{aligned}$$

□

The two innermost conjuncts in this definition capture the three informal requirements we have stated above:

- there is an MSC named X whose start guard is $\sigma|_F.(T.(t-1))$, and whose end guard is $\sigma|_F.(T.t)$,
- at the time points $\sigma|_F.(T.(t-1))$ and $\sigma|_F.(T.t)$ the component F assumes the control states $\sigma|_F.(T.(t-1))$ and $\sigma|_F.(T.t)$, respectively,
- between time $T.(t-1)$ and $T.t$ the system behavior is as the MSC X specifies.

The stream of time points T, which occurs in this definition, acts as an oracle predicting the "duration" of an MSC when its transition fires.

With $\{UNLD, LCKD\} \subseteq S_{Control}$ and $s_0 = UNLD$ as a description of the control states and the initial state, we obtain the MSC automaton from Figure 7.19 for our CLS example with respect to component *Control*.

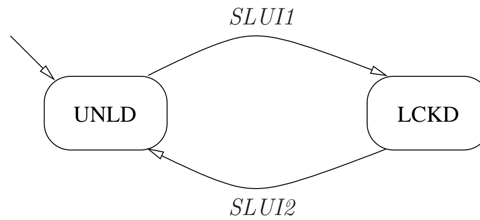


Figure 7.19.: MSC automaton for component *Control*

The safety part of this MSC automaton's semantics equals the one of the MSC term

$$(\rightarrow SLUI1\,;\,\rightarrow SLUI2)\uparrow_{<\infty>}$$

under the closed world semantics, starting from time 0.

Note the close relationship between MSC automata and HMSCs. The MSC automaton for a component F provides a "high-level" view on the state-oriented behavior of F.

To finalize the step from MSCs to finite state machines in the sense of Section 7.4.1 we only have to turn the transitions labeled with MSCs in the MSC automaton into state/transition-segments, such that every transition is labeled with an input message, an output message, or ϵ. Thus, we obtain the automaton corresponding to F in two steps.

First, we introduce fresh guards into the normalized MSCs, such that before and after each message arrow there is a guard; guards we add between the start and end guards of an MSC must be unique.

Second, we interpret each guard as a control state of the resulting automaton, and draw a transition arrow from state s to s' if and only if there is an MSC with a message arrow

between guards whose labels are s and s', respectively. We label the transition with an input or output message specification according to whether the message arrow in the MSC is an incoming or an outgoing one. If s and s' are the start and end guards of an MSC, and there is no message arrow between s and s' in that MSC we also introduce a transition, and label it by ϵ. More formally, to obtain the state transition function

$$\delta : \hat{I}_\epsilon \times \hat{O}_\epsilon \times S_F \to \mathcal{P}(S_F)$$

we require that

- $s' \in \delta(c \triangleright m, \epsilon, s)$ holds if and only if there is an MSC where an incoming arrow, labeled $c \triangleright m$, occurs directly after guard s and directly before guard s',
- $s' \in \delta(\epsilon, c \triangleright m, s)$ holds if and only if there is an MSC where an outgoing arrow, labeled $c \triangleright m$, occurs directly after guard s and directly before guard s',
- $s' \in \delta(\epsilon, \epsilon, s)$ holds if and only if there is an MSC where guard s directly precedes s' (with no message arrow in between).

According to its definition the set $\hat{I}_\epsilon$ contains exactly the messages labeling incoming arrows of the normalized MSCs, as well as ϵ; $\hat{O}_\epsilon$ contains – besides ϵ – exactly the messages labeling outgoing arrows of the normalized MSCs.

This construction yields a nondeterministic automaton specification

$$A_F = (S_F, \hat{I}_\epsilon, \hat{O}_\epsilon, \delta, \{s_0\})$$

where S_F is the set of control states, defined by the guards that occur explicitly in the MSCs, or were added between message arrows as intermediate states, $\hat{I}_\epsilon$ and $\hat{O}_\epsilon$ are the sets of input and output messages of F as defined by the (direction) of the arrows within the MSCs, δ is the transition function we have built in this transformation phase, and s_0 is the given, unique initial state.

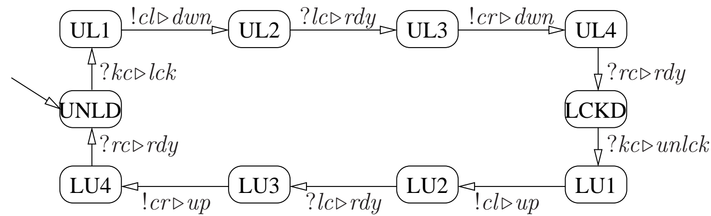


Figure 7.20.: Result of the transformation with respect to component *Control*

If we apply this construction to our CLS example for component *Control*, and label the intermediate states by *UL1* through *UL4*, and *LU1* through *LU4*, we obtain the automaton specification of Figure 7.20 as a result of applying the transformation of MSCs *SLUI1* and *SLUI2* into an automaton.

MSC Semantics vs. Automaton Semantics As we have noted before, the transformation procedure we develop here is purely syntactic; in particular, it is independent of the concrete semantics associated with MSCs and automata. Yet, because we have defined a semantics for our exemplary automaton model, we can relate the semantics of the original MSCs with the semantics of the resulting automata. We do so informally to convey the basic idea; intuitively, we sketch a trace back from the behavior of the automaton A_F to the original MSCs in restricted syntax. Without proof we note that the safety part of the semantics of A_F implies the safety part of the semantics of A_F^{MSC} by construction (up to the renaming of states). The only essential difference between A_F and A_F^{MSC} is that A_F makes precise which control states F assumes in the transition between the states representing the start and end guards of the given MSCs. A_F^{MSC} only restricts the time points at which the control states corresponding to start and end guards *must* occur, but makes no statement about the states occurring during any of the MSC automaton's transitions. Therefore, the step from A_F^{MSC} to A_F is a specification property refinement step in the sense of Section 6.2, and – if we consider only the I/O behavior of the component F corresponding to the automaton – a (component) property refinement step in the sense of Section 7.2.4. To see the relationship between the MSCs we have started out with and the resulting automaton's semantics recall the definition of A_F^{MSC}, which we gave earlier in this section (cf. Definition 46). A transition between two states of the MSC automaton captures precisely the behavior represented by one of the projected and normalized MSCs under the closed world semantics. The projection of the closed world semantics of each individual MSC in restricted syntax (cf. Section 7.4.1) equals the semantics of the MSC resulting from the projection phase by construction. Normalization does not change the sequences of interactions represented by the restricted MSCs. Thus, if we accept that the MSCs in restricted syntax are "connected" via guards representing control states (to express alternatives and repetition), then we see that up to the projection onto the component F, and up to the renaming of states, the semantics of A_F is a property refinement of the semantics of the original restricted MSCs under the closed world semantics.

We have deliberately chosen to use transition labels consisting of any one of the following only: ϵ, an input message specification, or an output message specification. As a result, the automaton specification we obtain is syntactically isomorphic to a standard finite state automaton (modulo fixing terminal states). In the final phase of the transformation procedure we exploit this "feature" by applying various optimization algorithms that exist for standard finite state machines to the automata we have obtained so far.

Optimization

In general, the automata we obtain by application of the transformation as we have described it up to this point, are nondeterministic and can have ϵ-transitions. As an example, consider the three MSCs *ND1*, *ND2*, and *ND3* from Figures 7.21 (a) through (c).

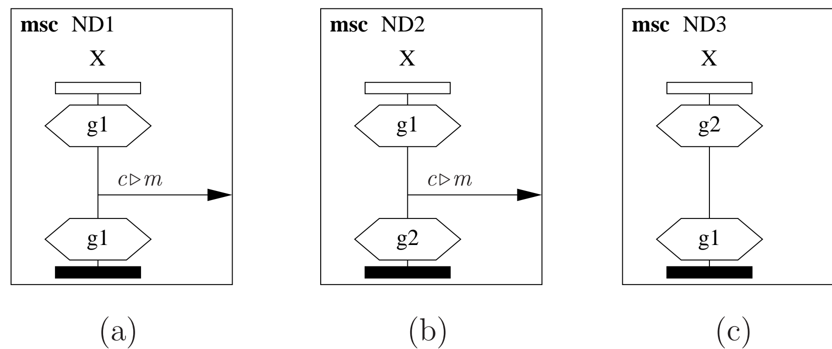


Figure 7.21.: MSCs yielding a nondeterministic automaton with an ϵ-transition

Figure 7.22 shows the automaton resulting from these MSCs, if we use *g1* as the initial state.

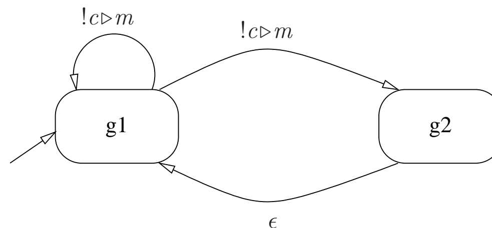


Figure 7.22.: Automaton derived from MSCs *ND1*, *ND2*, and *ND3*

Clearly, the automaton of Figure 7.22 bears potential for optimization with respect to the number of its states and transitions.

During the optimization phase we try to turn the result of the schematic application of the preceding phases into a more compact automaton specification. The reduction of nondeterminism is just one example for a standard algorithm known from automata theory that can bring us closer towards this goal. In addition, we might want to exploit "specialties" of the target automaton model. In the following paragraphs, we describe both of these forms of optimization, in turn.

Algorithms from Automata Theory

Automata theory provides various algorithms for optimizing standard finite state machines with respect to their number of states and transitions. Typical examples of such algorithms are the removal of ϵ-transitions, determinization, and minimization; the authors of [HU90] describe each of these in detail.

We can make these algorithms applicable to our automaton model as well. As we have noted earlier, the major difference between our model and the one of [HU90] for nondeterministic finite state machines is that the latter operate on finite sequences of inputs, whereas we have defined our automaton model on infinite channel valuations. [HU90]'s model includes

the following acceptance condition: after processing the finite input word, the automaton finally resides in a terminal state. Such acceptance conditions for finite input sequences correspond to liveness conditions for automata operating on infinite input (cf. Section 3.3.3).

Therefore, to carry over the standard optimization algorithms for finite state machines from [HU90], we have to designate a set of terminal states for the automaton we have constructed in the preceding transformation phases. For simplicity we let every automaton state be a terminal state, and apply the mentioned optimization algorithms starting with the nondeterministic finite state automaton

$$A^{NFS}_{inter} = (S_F, \hat{I}_\epsilon \cup \hat{O}_\epsilon, \delta_{inter}, s_0, S_F)$$

where we define $\delta_{inter} : S_F \times (\hat{I}_\epsilon \cup \hat{O}_\epsilon) \to \mathcal{P}(S_F)$ by

$$\begin{aligned} s' \in \delta_{inter}(s, x) \equiv \;& \langle \exists c \triangleright m \in \hat{I}_\epsilon : x = c \triangleright m : s' \in \delta(s, c \triangleright m, \epsilon) \rangle \\ & \vee \langle \exists c \triangleright m \in \hat{O}_\epsilon : x = c \triangleright m : s' \in \delta(s, \epsilon, c \triangleright m) \rangle \\ & \vee (x = \epsilon \wedge s' \in \delta(s, \epsilon, \epsilon)) \end{aligned}$$

By application of the algorithm for the elimination of ϵ-transitions, determinization, and minimization, in this order, with A^{NFS}_{inter} as the initial input, we obtain an optimal, deterministic automaton

$$A^{DFS}_{opt} = (S^{opt}_F, \hat{I}_\epsilon \cup \hat{O}_\epsilon, \delta^{DFS}_{opt}, s_0, S^{opt}_F)$$

which we turn back into an automaton within our model

$$A_{opt} = (S^{opt}_F, \hat{I}_\epsilon, \hat{O}_\epsilon, \delta_{opt}, s_0)$$

by defining $\delta_{opt} : S_F \times \hat{I}_\epsilon \times \hat{O}_\epsilon \to \mathcal{P}(S_F)$ by

$$\begin{aligned} s' \in \delta_{opt}(s, i, o) \equiv \;& \langle \exists c \triangleright m \in \hat{I}_\epsilon : i = c \triangleright m \wedge o = \epsilon : s' \in \delta^{DFS}_{opt}(s, c \triangleright m) \rangle \\ & \vee \langle \exists c \triangleright m \in \hat{O}_\epsilon : i = \epsilon \wedge o = c \triangleright m : s' \in \delta^{DFS}_{opt}(s, c \triangleright m) \rangle \end{aligned}$$

If, as an example, we start out with the MSCs from Figures 7.23 (a) through (e), and *g0* as the initial state, we obtain the automaton of Figure 7.24 (a) as the result of the transformation steps before optimization.

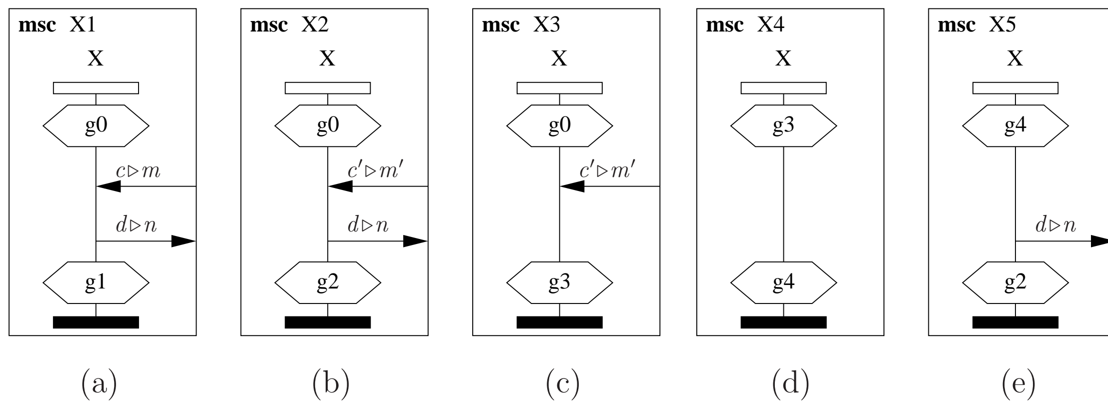


Figure 7.23.: Example MSCs amenable to optimization

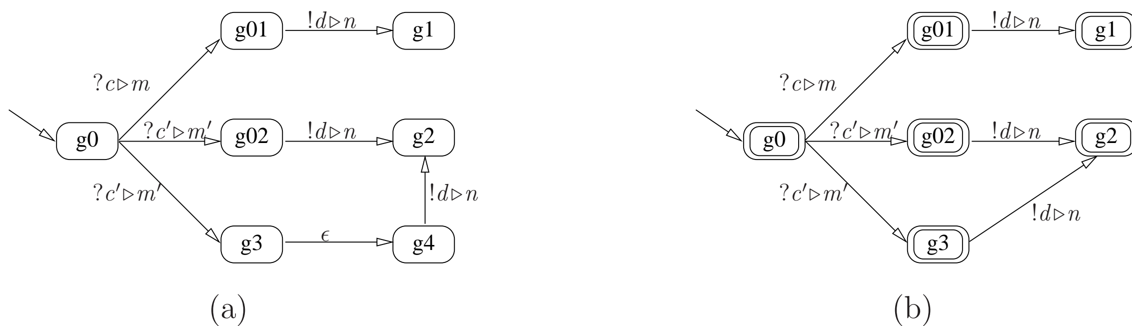


Figure 7.24.: Automaton before optimization (a) and after elimination of ϵ-transitions (b)

The elimination of ϵ-transitions yields the nondeterministic automaton of Figure 7.24 (b); Figure 7.25 (a) displays a deterministic equivalent. By means of the minimization with respect to the number of automaton states we obtain the automaton from Figure 7.25 (b).

Figure 7.25.: Automaton after determinization (a) and minimization (b)

In the course of optimization even states corresponding to guards within MSCs can "vanish", i.e. may become unreachable from the initial state. Hence, the state-part of the optimized automaton's semantics may not remain the same as the one of the MSC semantics. However, the I/O behavior of the automaton still implements the one specified by the MSCs under the closed world semantics.

Other Optimizations

The transition labels we have used so far are of the simple form "ϵ", "$?c \triangleright m$" or "$!c \triangleright m$" for some $c \triangleright m \in \langle \text{MSG} \rangle$. This simplicity was very helpful in view of the optimization algorithms we could directly apply to the resulting automata.

However, our automaton syntax also allows transition labels of the form "$?c \triangleright m/!d \triangleright n$"; this bears potential for further optimization of the automata produced by the transformation. We can shorten transition segments as depicted in Figure 7.26 (a) by dropping the intermediate state s_I from the set of states, and by connecting all transition paths from s_1 to each of the states s_1 through s_k, as Figure 7.26 (b) demonstrates. Clearly, this makes sense only if s_I is not the initial state of the automaton.

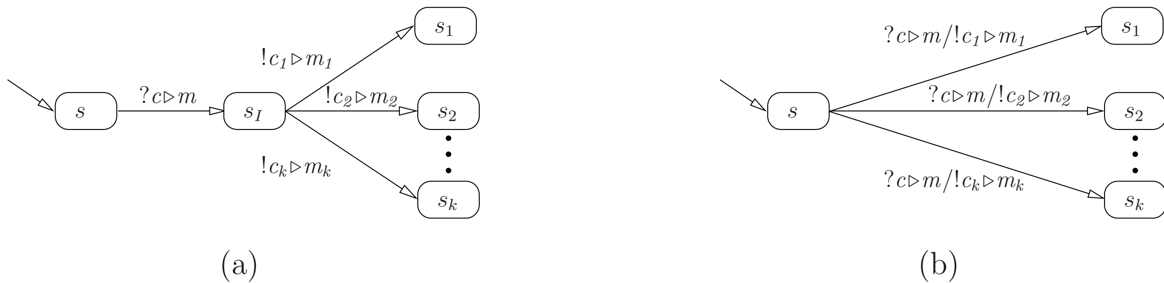


Figure 7.26.: Exploiting more elaborate transition labels

Other automaton models, such as statecharts, allow us to associate even longer chains of outputs with an input event. However, the causal relationship thus established between triggering input and triggered output may be rather artificial. Therefore, we consider this already a form of heuristic optimization. Heuristic optimizations require careful application to avoid introduction of unwanted side-effects, such as possibly unintended causality in this case.

7.4.3. Example: the Abracadabra-Protocol

In Section 4.7 we have studied the ABRACADABRA-protocol as an example for an MSC specification. Here, we show on a simplified version of this symmetric communication protocol how to derive an automaton specification for the individual components.

We use the MSCs from Figures 7.27 (a) through (d) to specify the simplified protocol from the view of component X; we have omitted the use case where X acts as the receiver in a successful communication initiated by Y. Moreover, we have fixed the orders of the messages in case of a conflict: here, X is always the first to send.

We apply the transformation procedure to obtain an automaton for X, and take *IDLE* as the initial state.

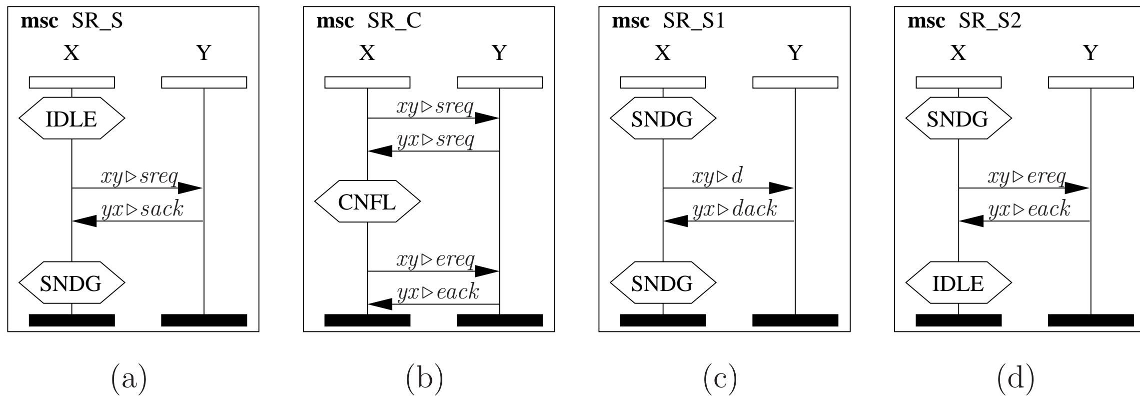


Figure 7.27.: MSCs for the simplified Abracadabra-protocol

Projection

Performing the projection onto X here simply amounts to omitting the axis for component Y; therefore, we do not draw separate MSCs for the projected use cases. In the following we refer to the MSCs of Figure 7.27 (a) through (d) as if they were projected onto X already.

Normalization

The only MSC that leaves room for normalization is the projection of *SR_C* onto X. According to the transformation procedure, we insert state *IDLE* as the start and end guard, and split the resulting MSC at guard *CNFL* to obtain the MSCs *SR_CN1* and *SR_CN2* from Figures 7.28 (a) and (b); these two MSCs are in normal form.

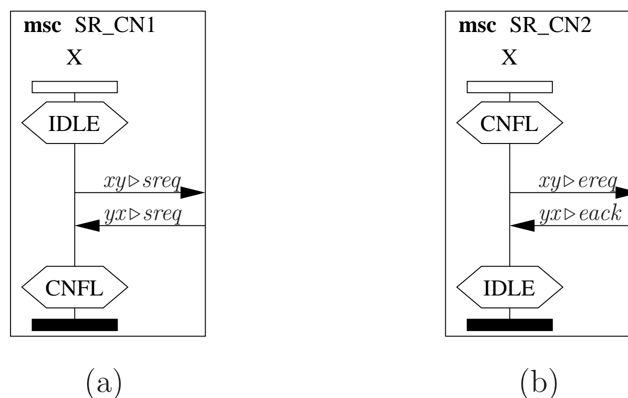


Figure 7.28.: Result of normalizing MSC *SR_C*

Transformation into an Automaton

The interpretation of the normalized MSCs as transition paths, and the insertion of intermediate states yields the automaton specification of Figure 7.29. Here, we show the intermediate states as unlabeled circles to distinguish them from the guards that occur in the MSCs.

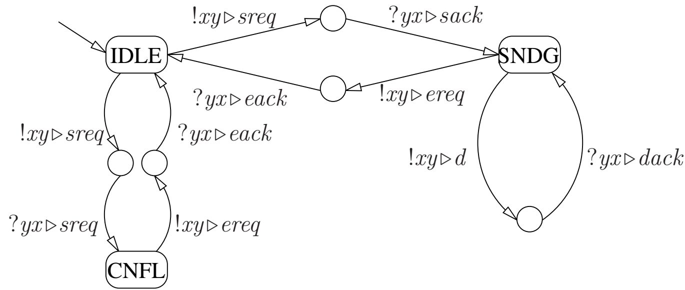


Figure 7.29.: Automaton for component X

Optimization

The automaton of Figure 7.29 has no ϵ-transition, but it is not deterministic: two transitions with the label "$!xy \triangleright sreq$" leave state *IDLE*. By application of the determinization algorithm from [HU90], we obtain the automaton from Figure 7.30 (a).

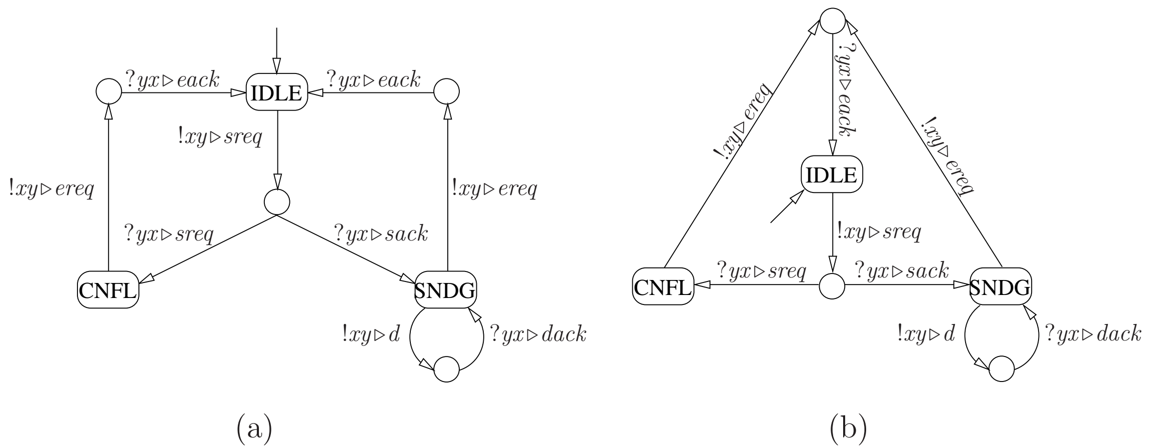


Figure 7.30.: Automaton after determinization (a) and minimization (b)

Minimization of this automaton identifies the two intermediate states whose outgoing transition is labeled with "$?yx \triangleright eack$" and ends at state *IDLE*. This yields the automaton from Figure 7.30 (b).

7.4.4. Extensions

The MSC syntax for which we have defined the transformation into automata is rather limited. However, by means of a few conventions we can apply it also to the more elaborate MSCs allowed by our MSC dialect. In the following paragraphs we show how to deal with alternatives, bounded and unbounded repetition, interleaving, join, preemption, and trigger composition.

Alternatives

Different MSCs starting with the same guard express the nondeterministic choice between alternatives. Therefore, we can translate every MSC term

$$\alpha \mid \beta$$

into the two terms

$$g : \alpha$$
$$g : \beta$$

for $\alpha, \beta \in \langle \text{MSC} \rangle$ and a (fresh) guard g to obtain the same nett effect in the restricted syntax.

Bounded Repetition

The most basic transformation for a bounded loop of the form $\alpha\uparrow_{<m,n>}$ for $m, n \in \mathbb{N}$ and $m \leq n$ is to produce n copies (named α_1 through α_n) of α, each with a unique set of guards. We assume that α is in normal form, and that α's start and end guards differ. As the start guard of α_1 we use the start guard of α, and as the end guard of α_n we use the end guard of α.

The next step is to derive the automata $A_{\alpha_1} = (S_{\alpha_1}, \hat{I}_\epsilon^{\alpha_1}, \hat{O}_\epsilon^{\alpha_1}, \delta_{\alpha_1}, s_0^{\alpha_1})$ through $A_{\alpha_n} = (S_{\alpha_n}, \hat{I}_\epsilon^{\alpha_n}, \hat{O}_\epsilon^{\alpha_n}, \delta_{\alpha_n}, s_0^{\alpha_n})$ corresponding to the n copies of α. Then, we combine these automata into a single result automaton by inserting ϵ-transitions "between" the automata A_{α_r} and $A_{\alpha_{r+1}}$ for all $r \in [1, n-1]$; this arranges the n automata into a chain corresponding to an n-fold sequential composition of the interaction pattern represented by α. Furthermore, because all repetitions beyond the lower bound m are optional, we add ϵ-transitions from the initial states of the automata A_{α_r}, for $r \geq m+1$, to the state of A_{α_n} that corresponds to the end guard of α_n. More precisely, we construct the resulting automaton as follows:

- the state set is the union of all state sets S_{α_1} through S_{α_n},
- the input symbol set is the union of all input symbol sets $\hat{I}_\epsilon^{\alpha_1}$ through $\hat{I}_\epsilon^{\alpha_n}$,
- the output symbol set is the union of all output symbol sets $\hat{O}_\epsilon^{\alpha_1}$ through $\hat{O}_\epsilon^{\alpha_n}$,

- the transition function δ is the union of the transition functions δ_{α_1} through δ_{α_n}, enriched by ϵ-transitions such that both

$$s_0^{\alpha_{i+1}} \in \delta(s_T^{\alpha_i}, \epsilon, \epsilon)$$

and

$$s_T^{\alpha_n} \in \delta(s_0^{\alpha_j}, \epsilon, \epsilon)$$

holds for all $i \in [1, n-1]$, and $j \in [m+1, n]$, where by $s_T^{\alpha_k}$, $k \in [1, n]$, we denote the control state corresponding to the end guard of MSC α_k,

- the initial state is the initial state of automaton A_{α_1}, which equals the initial guard of α.

As an example, consider the MSC *BLX* of Figure 7.31 (a). Figure 7.31 (b) shows the automaton with respect to component X that results from this expansion strategy.

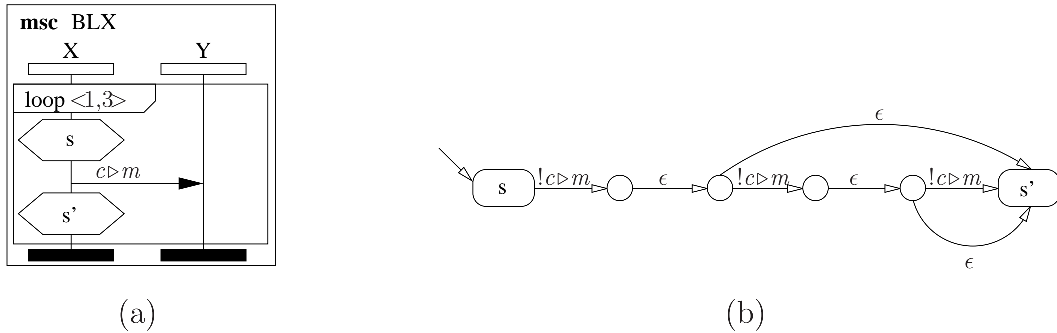


Figure 7.31.: Transformation of bounded repetition

Automaton models that provide access to the data state of the component under consideration would allow us a much more compact transformation. In statecharts and ROOM-Charts, for instance, we could simply introduce a counter variable for the number of repetitions of the interaction pattern; we would control the values of this counter within the range $[1, n]$ such that the pattern must occur at least m times, and at most n times.

Unbounded Repetition

In Section 7.4.2 we have already shown how to insert guards into copies of an MSC α, such that together these modified copies model the unbounded repetition of α when we perform the transformation to automata.

The motivation for this expansion is, again, the following understanding of MSCs with appropriately labeled guards: each such MSC represents a path through the automaton we construct; if such a path ends in the same state with which it started, then we can follow the same path over and over again. If there is another transition leaving the start state of

this path, and ending at another state, then we may also have an exit out of the repetition. This models unbounded loops.

Infinite Repetition

To model an infinite repetition, such as $\alpha\uparrow_{<\infty>}$ of an MSC α without guards, we simply have to use the same, unique guard g as both the start and as the end guard of the MSC:

$$(g : \alpha) \; ; \; (g : \textbf{empty})$$

If α does contain guards, we also have to ensure that none of these may occur as a start guard of any other MSC, i.e. there is no "exit" out of the infinite loop. Clearly, this only captures the safety part specified by an infinite loop. Below, we discuss how we can also integrate the corresponding liveness property.

Interleaving

The major purpose of interleaving in our MSC dialect is to express the lack of causality between message sequences. Our automaton model has no dedicated syntactic and semantic counterpart to the parallel inline expression of MSCs. However, we can always explicitly enumerate all possible interleavings of the operands of a parallel inline expression, and treat these interleavings as execution alternatives. Then, we can construct the component's automaton via the alternative MSCs.

Consider, for instance, the **par**-inline expression in the MSC from Figure 7.32. We can expand this inline expression into the MSCs of Figure 7.33 (with respect to component X).

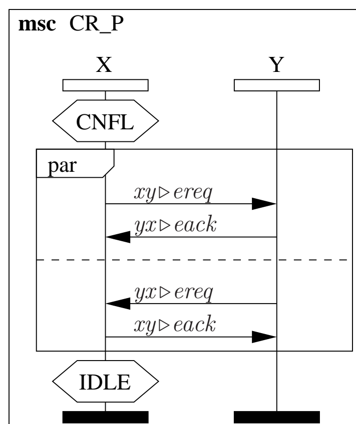


Figure 7.32.: MSC with **par**-inline expression

Figure 7.34 shows the nondeterministic automaton resulting from the transformation of these six MSCs with *CNFL* as the initial state.

The drawback of this syntactic expansion of the **par**-inline expression is the induced "combinatoric explosion" of the number of intermediate states for the nondeterministic automaton.

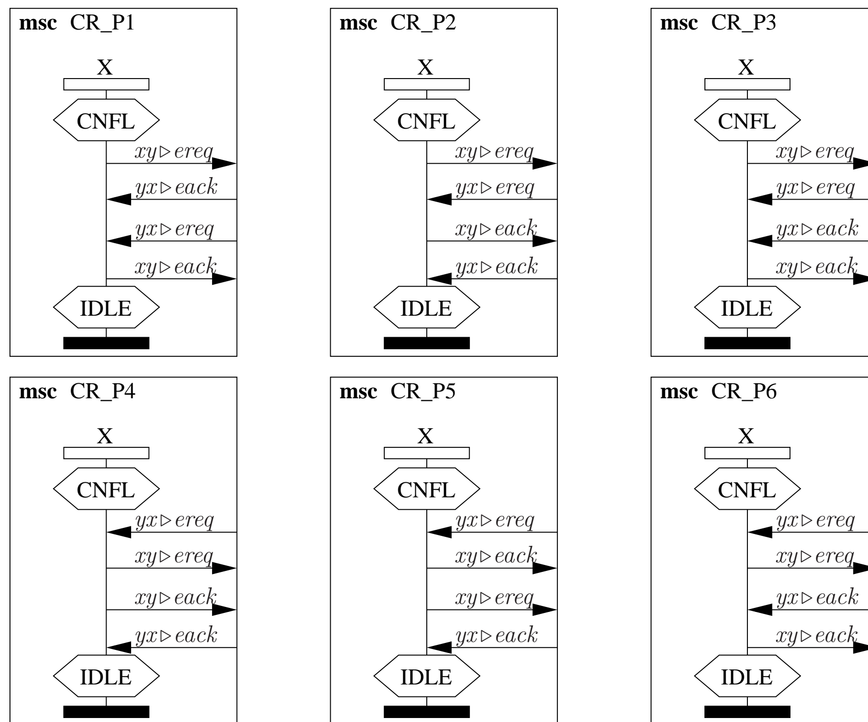


Figure 7.33.: Explicit interleavings of the messages from Figure 7.32

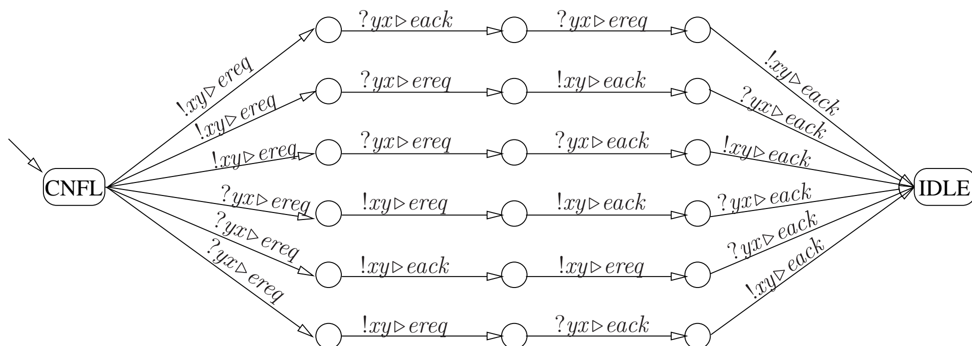


Figure 7.34.: Automaton obtained from the MSCs of Figure 7.33

More elaborate automaton models, such as statecharts and ROOMCharts, provide better syntactic support for modeling causal independence. Recall from Section 3.3.2 that statecharts offer the concept of AND-states, which we could exploit in our transformation procedure if the operands of the inline expressions do not share messages. We simply would have to transform each operand separately, and put the results of the transformations into separate compartments of an AND-state; Figure 7.35 illustrates the idea of this approach with respect to the example we have started above.

In ROOM a possible solution would also be to perform a separate transformation of the two operands, and to create two sub-actors of the actor corresponding to component X,

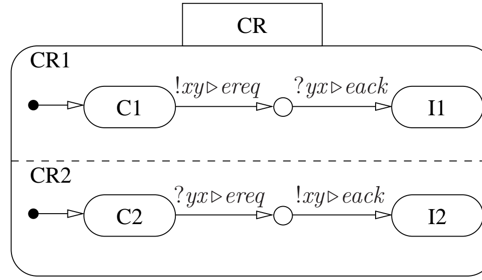


Figure 7.35.: Statechart (with AND-state) corresponding to the automaton from Figure 7.34

such that these sub-actors operate as separate "threads" to handle the parallel behavior.

Join

The semantics of the join of two MSCs α and β identifies the messages shared by α and β; messages outside the set $msgs.\alpha \cap msgs.\beta$ can occur interleaved. Again, our automaton model provides no direct support for "implementing" the join operator.

As a remedy, we can translate each operand of the join separately – as we have suggested above to implement the interleaving operator with statecharts –, and then combine the results by means of an operation on automata that captures the idea of the join operator.

To this end, we define the cross product of two automata $A = (S_A, \hat{I}^A_\epsilon, \hat{O}^A_\epsilon, \delta_A, s^A_0)$ and $B = (S_B, \hat{I}^B_\epsilon, \hat{O}^B_\epsilon, \delta_B, s^B_0)$, which we denote by $A \otimes B$, as

$$A \otimes B \stackrel{\text{def}}{=} (S_A \times S_B, \hat{I}^A_\epsilon \cup \hat{I}^B_\epsilon, \hat{O}^A_\epsilon \cup \hat{O}^B_\epsilon, \delta_{A\otimes B}, (s^A_0, s^B_0))$$

where we have

$$\begin{aligned}
& (s'_A, s'_B) \in \delta_{A\otimes B}((s_A, s_B), i, o) \\
\equiv\; & (i \neq \epsilon \equiv o = \epsilon) \\
& \wedge ((s'_A \in \delta_A(s_A, i, \epsilon) \wedge s'_B \in \delta_B(s_B, i, \epsilon)) \Leftarrow i \in \hat{I}^A_\epsilon \cap \hat{I}^B_\epsilon) \\
& \wedge ((s'_A \in \delta_A(s_A, \epsilon, o) \wedge s'_B \in \delta_B(s_B, \epsilon, o)) \Leftarrow o \in \hat{O}^A_\epsilon \cap \hat{O}^B_\epsilon) \\
& \wedge ((s'_A \in \delta_A(s_A, i, \epsilon) \wedge s'_B = s_B) \Leftarrow (i \in \hat{I}^A_\epsilon \wedge i \notin \hat{I}^B_\epsilon)) \\
& \wedge ((s'_A = s_A \wedge s'_B \in \delta_B(s_B, i, \epsilon)) \Leftarrow (i \notin \hat{I}^A_\epsilon \wedge i \in \hat{I}^B_\epsilon)) \\
& \wedge ((s'_A \in \delta_A(s_A, \epsilon, o) \wedge s'_B = s_B) \Leftarrow (o \in \hat{O}^A_\epsilon \wedge o \notin \hat{O}^B_\epsilon)) \\
& \wedge ((s'_A = s_A \wedge s'_B \in \delta_B(s_B, \epsilon, o)) \Leftarrow (o \notin \hat{O}^A_\epsilon \wedge o \in \hat{O}^B_\epsilon))
\end{aligned}$$

for states $(s_A, s_B), (s'_A, s'_B) \in S_A \times S_B$, and message specifications $i \in \hat{I}^A_\epsilon \cup \hat{I}^B_\epsilon$, $o \in \hat{O}^A_\epsilon \cup \hat{O}^B_\epsilon$.

Thus, by construction, $\delta_{A\otimes B}$ contains a transition from state (s_A, s_B) to (s'_A, s'_B) if and only if

- both A and B can take a transition from s_A to s'_A, and s_B to s'_B, respectively, labeled with the same input or output message, or
- A can take a transition from s_A to s'_A labeled with i/ϵ or ϵ/o, and B has no transition with the same label, or
- B can take a transition from s_B to s'_B labeled with i/ϵ or ϵ/o, and A has no transition with the same label.

In essence, $A \otimes B$ yields the product of the automata A and B such that transitions with the same label synchronize.

As an example, consider the two MSCs JA and JB from Figure 7.36. These MSCs describe two different views on a system in which X sends the message $xy \triangleright m1$ to Y. JA specifies what happens between X and Y only, whereas JB shows that the message $zx \triangleright r$ is an "indirect" result of the triggering message $xy \triangleright m1$.

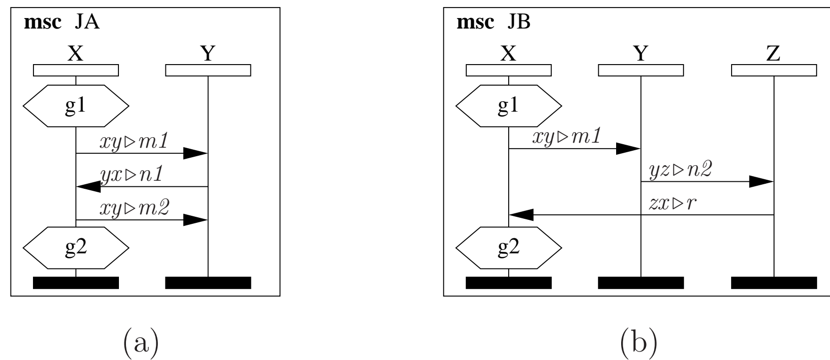


Figure 7.36.: Operand MSCs for the join operator

To derive an automaton for the join of these two MSCs with respect to the component X, we transform each of the MSCs into a separate automaton, as depicted in Figures 7.37 (a) and (b).

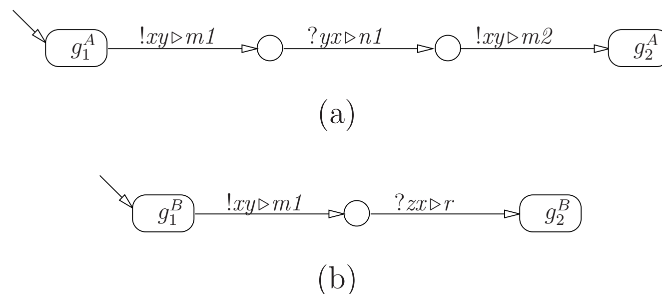


Figure 7.37.: Automata for X, derived from JA and JB

Figure 7.38 shows (the reachable part of) the product of these two automata schematically. It identifies the two occurrences of the message label "$!xy \triangleright m1$", and interleaves the rest of the messages.

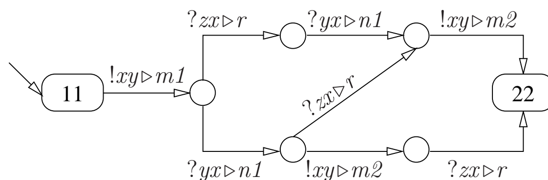


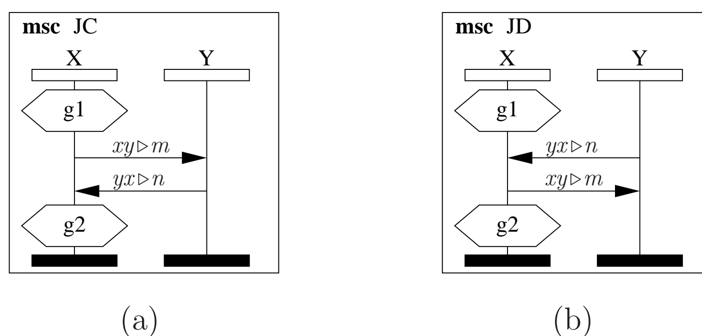
Figure 7.38.: Automaton for the join of MSCs *JA* and *JB*

Again, we could exploit the syntactic "features" of statecharts to reduce the sets of states and transitions of the resulting automaton.

In Section 4.4 we have noted the existence of MSCs α and β whose join has an empty semantics, i.e. MSCs for which $[\![\alpha \otimes \beta]\!] = \emptyset$ holds; in other words, α and β are inconsistent with respect to join.

The construction of the corresponding product automaton reflects this by yielding a transition function where the state representing the join of the two MSC's end guards is unreachable from the initial state.

Consider the two MSCs *JC* and *JD* of Figure 7.39. The join of *JC* and *JD* has an empty semantics, because there is no way to identify the occurrences of $xy \triangleright m$ and $yx \triangleright n$ that respects the ordering of these messages in *both* MSCs.



(a) (b)

Figure 7.39.: MSCs that are inconsistent wrt. join

The automata corresponding to the projection of *JC* and *JD* onto *X* with *g1* as the initial state appear in Figure 7.40.

(a) (b)

Figure 7.40.: Automata for *X* according to MSCs *JC* and *JD*

If we construct the product automaton $A_{JC} \otimes A_{JD}$, we find the set of states reachable from (g_1^C, g_1^D) to be empty. In particular, the state corresponding to the "joint" end guard (g_2^C, g_2^D) is unreachable from the initial state.

This observation leads to a constructive procedure for checking whether two normalized MSCs α and β are consistent with respect to join. We formulate the procedure based on a simpler notion of join consistency for individual components, as follows.

Definition 47 (Join Consistency of Components) Let α and β be normalized MSCs, and let $F \in P$ represent a component occurring in α and β. Let A_α and A_β be the automata corresponding to α and β according to the transformation procedure. For $\tau \in \{\alpha, \beta\}$ we denote by s_0^τ and s_T^τ the automaton state corresponding to τ's start and end guard, respectively. We call α and β *join-consistent with respect to* F if the state (s_T^α, s_T^β) is reachable from the initial state (s_0^α, s_0^β) in the transition function $\delta_{A_\alpha \otimes A_\beta}$ of the product automaton $A_\alpha \otimes A_\beta$. □

To check whether two MSCs α and β are consistent with respect to join, we have to check whether all components that occur within α and β are mutually join-consistent.

Proposition 17 (Join Consistency) *Let α and β be normalized MSCs. α and β are consistent with respect to join if and only if α and β are join-consistent with respect to every component appearing in α and β.* □

Proof See Appendix B.3. ■

This constructive criterion for join consistency allows us to determine whether a set of MSCs describes "sensible" interaction patterns.

Preemption

To translate a preemption specification like

$$\alpha \stackrel{ch \triangleright m}{\rightarrow} \beta$$

into an automaton specification, we observe the following property of the semantics of the preemption construct: whenever message $ch \triangleright m$ occurs during an execution corresponding to α, then from this time point on the execution evolves as described by β. In terms of the automaton we aim at, this means that every state corresponding to α's behavior should have an outgoing transition to the state where β's behavior starts.

This immediately suggests to transform α and β separately into the automata $A_\alpha = (S_\alpha, \hat{I}_\epsilon^\alpha, \hat{O}_\epsilon^\alpha, \delta_\alpha, s_0^\alpha)$ and $A_\beta = (S_\beta, \hat{I}_\epsilon^\beta, \hat{O}_\epsilon^\beta, \delta_\beta, s_0^\beta)$ with disjoint state sets S_α and S_β, and to integrate A_α and A_β such that from every state $s_\alpha \in S_\alpha$ there is precisely one outgoing transition labeled $ch \triangleright m$, which ends at s_0^β. More precisely, we define

$$A = (S_\alpha \cup S_\beta, \hat{I}_\epsilon^\alpha \cup \hat{I}_\epsilon^\beta, \hat{O}_\epsilon^\alpha \cup \hat{O}_\epsilon^\beta, \delta, s_0^\alpha)$$

such that

$$\begin{aligned} & s' \in \delta(s, i, o) \\ \equiv \quad & (s \in S_\alpha \wedge s' = s_0^\beta \wedge ((i = ch \triangleright m \wedge i \in \hat{I}_\epsilon^\alpha) \vee (o = ch \triangleright m \wedge o \in \hat{O}_\epsilon^\alpha))) \\ & \vee (s \in S_\alpha \wedge s' \in \delta_\alpha(s, i, o) \wedge (i \neq ch \triangleright m \wedge o \neq ch \triangleright m)) \\ & \vee (s \in S_\beta \wedge s' \in \delta_\beta(s, i, o)) \end{aligned}$$

holds. Clearly, this makes only sense if ch is either an incoming or an outgoing channel of the component under consideration. Otherwise the component cannot directly observe the occurrence of message $ch \triangleright m$ in its own I/O behavior. Hence, if we encounter a preemptive message $ch \triangleright m$ during the transformation process, and ch is not a channel of the component for which we construct the automaton, then we first have to enrich the MSCs to reflect the occurrence of message $ch \triangleright m$ on a channel to which the component has access. One way to achieve this is to perform a message refinement on $ch \triangleright m$ (cf. Section 5.4), such that the refining protocol indicates the "occurrence" of the (abstract) preemptive message on one of the channels of the component.

By means of a similar construction, we can also translate preemptive loops like $\alpha \Uparrow_{ch \triangleright m}$. Here, we produce an automaton A_α for α, and change δ_α such that there is precisely one outgoing transition labeled $ch \triangleright m$ from every state in S_α, and this transition must end at the initial state of A_α; otherwise δ_α remains as is.

Trigger Composition

The MSC term $\alpha \mapsto \beta$ expresses the following liveness property: whenever an interaction sequence corresponding to α has occurred in an execution, then an interaction sequence corresponding to β follows.

It makes only little sense to construct an automaton with respect to $\alpha \mapsto \beta$ directly, because there is a plethora of automata that trivially implement this property; it holds, in particular, for all automata with an empty transition set.

We are more interested in specifications of the form $\alpha \otimes (\beta \mapsto \gamma)$, and aim at constructing an automaton A_α from α whose semantics has the liveness property $\beta \mapsto \gamma$. As we have seen above (cf. Section 7.4.1), we can use $\beta \mapsto \gamma$ directly as the liveness property $AL.A_\alpha$ to obtain the semantics of the automaton A_α. The intersection of $AS.A_\alpha$ and $AL.A_\alpha$ implicitly identifies the messages shared by α and β.

HMSCs

So far, we have considered plain MSCs only. However, as we have demonstrated in Section 4.6 there is a constructive transformation from HMSCs to plain MSCs that we can apply first, before we transform the resulting plain MSCs into an automaton for the component under consideration.

HMSCs are helpful in our transformation procedure from a methodical point of view. Because their main purpose is to express alternatives and repetition of entire MSCs, we can use them to find guards representing control states in the plain MSCs referenced by the HMSC.

For instance, we could add guards to the MSCs A and B referenced by the HMSC from Figure 7.41 (a), such that A and B have the same start guard, as well as the same end guard.

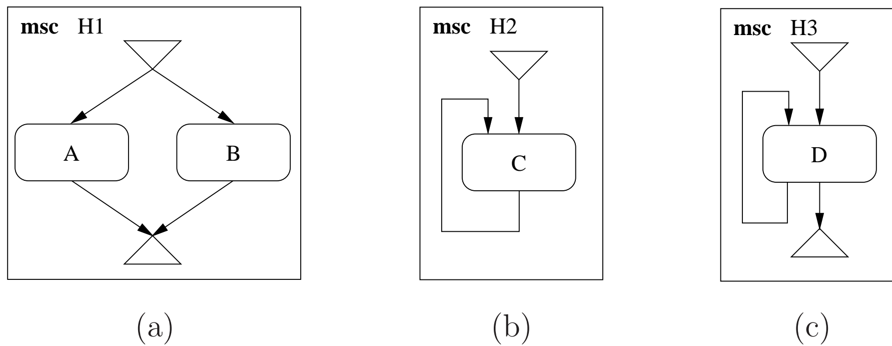


Figure 7.41.: HMSCs defining the labeling of plain MSCs A, B, C, and D with guards

The HMSC of Figure 7.41 (b) suggests an identical start and end guard for MSC C.

The HMSC of Figure 7.41 (c) induces the need for two copies of MSC D with the same start guard: one whose end guard equals the start guard (to represent the possible repetition), and another whose end guard differs from the start guard (to model exit from the repetition).

Thus, HMSCs define ("sanity checks" for) the placement of guards within the corresponding plain MSCs used as the input for our transformation procedure.

Data States

Our transformation procedure considers MSCs with guards representing control states only. We give a rough sketch of how to integrate data states into this framework.

An easy way to extend our work to guards that also refer to data states is to consider MSC guards as pairs (p_c, p_d) (cf. [Bro98, Bro99b]). In such a pair p_c is a control state as before, and p_d is a predicate on the data state of the component under consideration. Based on this structuring of guards we perform the transformation with respect to the control states as before.

The data state predicates p_d then act as "slices" of the control state they are associated with.

A thorough treatment of data states is an interesting area for future work (cf. Chapter 8).

7.4.5. Methodological Issues

By now, we have available a transformation procedure suitable for deriving automaton specifications from a significant subset of the MSC dialect we have introduced in Chapter 4.

We conclude this section with a few methodical considerations about the application of our transformation procedure within the development process.

The Role of Guards

The shape of the resulting automata can strongly depend on how many guards the developer of an MSC has specified. This is a consequence of our choice to introduce *fresh* intermediate states between message occurrences in the normalized MSCs.

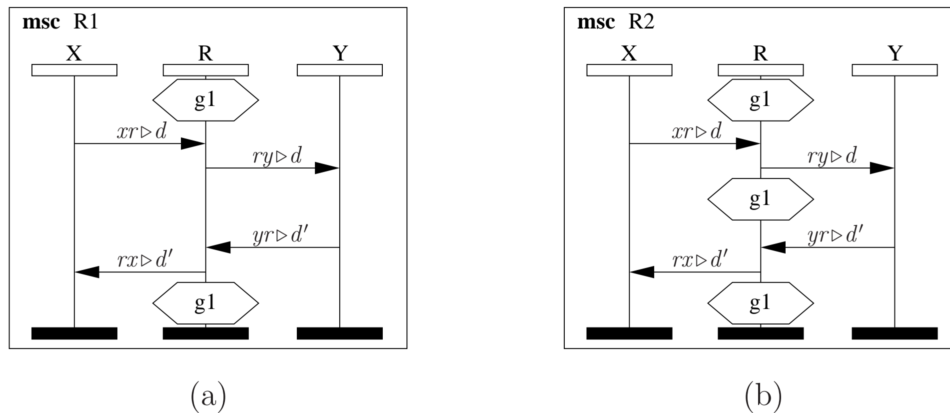


Figure 7.42.: MSCs with different numbers of occurrences of guard *g1*

For instance, the MSCs *R1* and *R2* from Figure 7.42 yield the two different automaton specifications from Figure 7.43 (a) and (b), respectively, according to our transformation procedure for component R with *g1* being the initial state. Both MSCs specify R as a "relay" component that forwards the messages it receives.

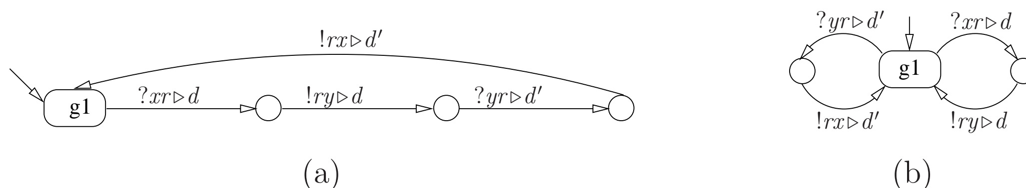


Figure 7.43.: Automata corresponding to the MSCs *R1* and *R2*

The information provided by the MSC *R1* does not clarify what states the component R assumes between forwarding the messages from X and Y. This results in an automaton where R must first forward a message from X before it can forward one from Y (cf. Figure

7.43 (a)). *R2*, on the other hand, yields an automaton that decouples the forwarding tasks to some extent (cf. Figure 7.43 (b)); R now can forward messages from X and Y independently.

This example shows that the placement of guards not only has an effect on the size of the resulting automata, but also on the overall behavior of the component in relation to its environment. Our choice of inserting fresh intermediate states (as compared to, say, arbitrary guards) follows "the principle of least surprise", but may make the component more dependent on its environment than necessary. Once we have detected such an unnecessary dependency in the resulting automaton, we can go back to the MSC specification, and relax it by adding further (alternative) MSCs.

The important role played by guards during the transformation process induces the question at what points during the development process we should add guards to MSCs, and at what level of detail we should do so.

We consider the focus of MSCs on message exchange, instead of on state change, one of the particular advantages of this description technique; this focus makes MSCs applicable already in very early development phases, where we have fixed only little about the detailed behavior of the individual components. Once we start adding guards with control information to the MSCs we reduce our freedom for implementing the component's behavior (adding guards is a property refinement step on MSCs, cf. Section 5.3). This suggests to add as few guards as possible to MSCs, and to do so as late as possible in the development process, close to the switch from the interaction-oriented to the state-oriented component view.

However, guards also help structure large MSC specifications, and thus help highlight the roles played by individual components in an interaction. We have already seen a simple example for this usage of guards in our treatment of the ABRACADABRA-protocol in Section 7.4.3. Here, the control states *IDLE*, *CNFL*, and *SNDG* indicate phases of the communication protocol. Structuring the specification around the start and end points of such phases can even be the origin for developing an MSC specification in the first place. Then it definitely makes sense to keep these start and end guards around within the MSCs.

In summary, we conclude that the MSCs we work with until just before the derivation of automata should contain guards only sparsely; these guards should, in particular, clarify the structuring of a component's behavior (in phases) over time. The fewer states we fix through guards before application of the transformation procedure, the more work we leave to the procedure itself. If the resulting automata display unwanted dependencies with respect to their environment, then we can take a step back, and relax the MSC specification by means of additional alternative MSCs.

Relaxing the MSC Interpretation

Our decision to use the exact MSC interpretation as the basis for the transformation procedure yields automata capable of processing exactly the messages contained in the given MSCs, in precisely the same order as specified there.

If, however, we assume that the MSCs fix only the interaction behavior displayed by the component and its environment with respect to the messages occurring explicitly within the MSCs, and allow arbitrary *other* message exchange in between, we have to change the shape of the resulting automata slightly.

One approach to relaxing the MSC interpretation in this way is to add idle-loops to every automaton state, such that these loops force the automaton to maintain its current state if it receives an input message absent from all of the given MSCs. It is easy to imagine multiple other sensible relaxations of the MSC interpretation we start out with. Which one we choose strongly depends on the target automaton model, as well as on the execution environment we have given. If the MSCs describe the behavior of both the component and its environment completely, the exact interpretation is the right choice; if we only have partial knowledge about the environment, we need a more "forgiving" MSC interpretation, or, alternatively, a less strict target automaton model.

From Automata to A/C Specifications

We have given the semantics of an automaton A as a subset $[\![A]\!] \subseteq (\tilde{C} \times S)^{\infty}$ of the set of all system executions. $[\![A]\!]$ contains all those infinite executions where both, the component associated with A, and its environment "play by the rules".

We can convert this automaton specification into a relational component specification along the same lines we followed in Section 7.3, where we derived A/C specifications from MSCs. We simply have to come up with a closed world version of the automaton semantics, and derive an interaction interface from it; this interaction interface is the input we need for the decomposition scheme of Section 7.3.3. The resulting A/C specification yields a reactive component specification that associates with every input history of the component at least one output history.

"Non-Local Choice"

The automata resulting from application of the transformation procedure may have states with outgoing transitions labeled with both input messages and output messages (cf. Figure 7.44).

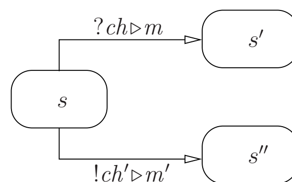


Figure 7.44.: State with outgoing input and output transitions

Thus, in such a state the automaton can either read an input message present on the corresponding input channel, or output a message and change state. As [LL95, Leu95] discuss

in detail under the label of “non-local choice”, this nondeterminism can lead to undesired behavior, if the automata in the environment and the automaton of the component under consideration take their nondeterministic choices independently.

[Leu95] and [HK99] describe several schemes for solving this problem by means of explicit coordination. In a tool environment supporting the construction of automata according to our procedure, we could highlight states that display this problem, and request the developer to add adequate communication among the affected components within the appropriate MSCs to establish explicit coordination.

Automata: Safety and Liveness

Our discussion of the transformation procedure has focused mainly on the safety part of the resulting automaton specifications. In fact, we can view the transformation procedure as a constructive derivation scheme for the safety part of the closed world semantics of an MSC. However, as we have shown in Chapter 6, the properties specified by MSCs under the closed world semantics have a liveness part as well. We can add this liveness part schematically to an automaton specification as follows.

Let $A = (S_A, \hat{I}_\epsilon, \hat{O}_\epsilon, \delta, s_0)$ be the automaton we obtain by application of the transformation procedure to a set of MSCs with respect to component F. Let, furthermore, $R : \overrightarrow{(I \cup O)} \to \mathbb{B}$ be the interaction interface derived from the set of MSCs (according to Section 7.3.2). Then we define the liveness property $AL.A$ by

$$AL.A \stackrel{\mathsf{def}}{=} \{\varphi \in (\tilde{C} \times S)^\infty : \langle \exists i \in \vec{I}, o \in \vec{O} :: \pi_1(\varphi)|_{I \cup O} = i \oplus o \wedge L_R.i.o \rangle\}$$

where L_R is the canonic liveness property of the derived interaction interface (cf. Section 7.3.2). This ensures that the semantics of automaton A implies the liveness properties specified by the MSCs. However, this rather implicit liveness specification provides only little help if we have to implement the liveness requirement for automaton A in an operational way.

From the semantics definition in Chapter 4, the discussion of liveness properties in Chapter 6, as well as from the derivation scheme for A/C specifications in Section 7.3.3 we know the major liveness requirement captured by MSCs: if the environment of the component F plays by the rules (i.e. fulfills the environment safety assumption and indeed supplies the specified messages within finite time) then F can delay its response at most for a finite amount of time.

We can turn this observation into schematic liveness requirements on the occurrences of control states in executions of the automaton corresponding to component F. For instance, we could require that whenever a state s with at least one outgoing transition whose label is an output message of F occurs in an execution, then eventually a direct successor state s' of s with respect to the transition function δ follows in this execution, and the transition taken by the component to reach s' is labeled with an output message of F. Operationally,

one way to implement this requirement is to allow only provably terminating program statements to implement such an output transition.

We could formulate similar requirements to capture fairness specifications, such as the trigger composition of two MSCs. If, however, the automata produced by our transformation procedure are only an intermediate step, the target being a temporal logic like TLA or UNITY (which provides means for *proving* liveness or fairness properties) then we have to encode the liveness requirements in a form supported by this logic. Instead of presenting such a special-purpose encoding here, we stick with the schema we have just outlined above.

Refinement of Automata

In Chapter 5 we have discussed the refinement of MSC specifications in detail. In line with our view of MSCs as an interaction-oriented specification technique we have paid special attention to the refinement of interaction protocols there. Now that we have established the transition to state-oriented specifications we can use the latter as "jump-start" models of detailed component behavior, and use refinement techniques whose focus is the addition, removal, or rearrangement of states instead of interactions.

The authors of [Rum96], [CD94], [Kle98], and [Sch98] present such refinement techniques for automaton models, in detail. Because we can carry over most of their work into the context of our semantic model, we have now available an entire portfolio of refinement techniques for specifications on various levels of detail.

From Automata Back to MSCs

If, after transforming a set of MSCs into an automaton, we perform modifications to this automaton as part of the further development of the component, the question arises how we can keep the set of MSCs we have started out with consistent with the modified automaton. This is of particular importance if we wish to use the MSCs as part of the documentation, or as the source for developing a test-suite for the implementation.

However, the step from an automaton back to a set of MSCs is not as canonic as the other way around; this step depends crucially on what start and end points we chose to "flatten" the transition paths through the automaton. Matching the MSCs we obtain from these transition paths with the MSCs that were the basis for the construction of the automaton is, in general, a nontrivial task, which is outside the scope of this thesis.

7.5. Related Work

The transformation of MSCs or, more generally, scenarios into specifications of individual components has recently become a very active area of research. The authors of [HK99] give a good overview of part of this topic's "historic" development in the literature, including notes on relationships with synthesis problems treated in the context of temporal logic.

In the following, we give an overview of the approaches in the literature we consider most related to our work. First, we discuss approaches related to A/C specifications and automaton models in general. Then, we focus on the derivation of automaton specifications from MSCs.

7.5.1. A/C Specifications, Automaton Models

The derivation of A/C specifications from MSCs was prepared in [BK98], where we used interaction interfaces as the starting point for the construction of component specifications. Our work in Section 7.3 extends this contribution by enabling the derivation of interaction interfaces directly from the MSC semantics of Chapter 4 – without any intermediate transformations or conversions. [Fac95] uses the A/C specification style directly for the semantics definition of TSDs. The authors of [KM99] use projections of a Petri-Net formalization of scenarios to obtain criteria for individual components that resemble A/C specifications.

The authors of [HU99] discuss an instance of the submodule construction problem (SCP, cf. [MB83]): given a specification of system M_0 and of one of its components M_1, derive a specification of a component M_2, such that the composition of M_1 and M_2 implements M_0. The authors use prefix-closed finite state machines to represent M_0, M_1, and M_2, and present an algorithm that yields an automaton for M_2 with the required property, if the automata for M_0 and M_1 are given, and if a solution to the SCP exists. If, in our setting, we consider M_0 as the specification of an interaction interface, and M_1 as a specification of the environment of M_2, then we can consider the construction of an A/C specification (or an automaton, if M_0 is given as an MSC) for M_2 also as an instance of the SCP. However, the approach we have presented in Section 7.3 is more general than the one of [HU99], because we also discuss the mapping of general liveness properties. We mention the work of [HU99] mainly for another reason. The criterion given by the authors as an indicator for automata for which no solution to the SCP exists basically coincides with our definition for the join-consistency of MSCs.

The authors of [LL95] present various automaton models, which can directly serve as the semantics base for MSCs. In particular, they discuss the relationship between message flow graphs (MFGs, an automaton variant) and temporal logic, in detail. This includes a treatment of various approaches for assigning liveness properties to MFGs. An extension of this work appears in [Leu95], where the author investigates the relationship between MSCs

and finite state machines thoroughly. In particular, he discusses the representation of both synchronous and asynchronous communication primitives, as well as the use of conditions in MSCs. Mapping MSCs to state machines for the individual components induces the problem of "non-local choice", which the author addresses by means of history variables.

[Rum96], and [Kle98] describe the syntax and semantics of more elaborate automaton models than the one we have used in this chapter. [Sch98] treats a subset of the statechart language syntactically and semantically. [Kle98] uses his model to give a formal, as well as a methodical basis for system development with scenarios; the derivation of automata from MSCs plays a less prominent role there. [Rum96], [Kle98], and [Sch98] also contain a plethora of refinement notions for automata, which we could easily adapt to our work to obtain a seamless transition from interaction requirements captured by MSCs to the specification and design of detailed component behavior on the basis of automaton specifications. The authors of [BP99] adapt the proof rules for safety and progress from UNITY [CM88] to stream-based black-box specifications for state machines; as a result they obtain a framework for reasoning about state machine behavior. The aim here is to take a state machine as given, and to capture its relevant properties by means of more abstract specifications referring to the machine's I/O behavior only. In a sense, this latter view corresponds to our notion of interaction interfaces. Therefore, there is potential for relating the work of [BP99] with the MSC specifications we use here to obtain an integration of the properties expressed by MSCs into the validation process.

7.5.2. Work on the Transformation of MSCs to Automaton Specifications

In the following paragraphs we compare our approach to the transformation of MSCs into automaton specifications with similar ones from the literature. We summarize the approaches of [Fei99], [LMR98], [KM93, KMST96, KSTM98], and [HK99]. After this, we highlight the "features" that distinguish these approaches, and discuss their relationships with our work.

Summaries

[Fei99]: "Generating FSMs from interworkings"

[Fei99] contains a transformation scheme enabling the derivation of finite state machines (in the sense of SDL) from Interworkings (cf. [MR94, MR96] and Section 2.3.4). The author bases the transformation on translating Interworkings into process-algebraic terms, and on generating a state-model by application of a series of term-rewriting rules. The transformation scheme proceeds in five steps. The first step consists of projecting the given Interworkings onto the process under consideration. The result of this step is one

process-algebraic term for every alternative behavior of this process. The second step of the transformation serves the purpose of extracting common prefixes and suffixes of the alternative process terms. The aim of this extraction is twofold: first, it serves to delay the choice between different alternative behaviors with common prefixes until their first deviation; second, it serves to reduce the number of generated states by representing common suffixes only once. The third step consists of labeling sub-process-terms in each of the terms obtained from step two to identify intermediate states of the finite state machine, such that there is an intermediate state between any two receive actions of the process. The idea here is to associate one control state with each receive action, and to "complete" an automaton transition before the next receive action occurs. The fourth step consists of unfolding alternatives with common suffixes to avoid SDL transitions using the "join" construct. The fifth step associates an automaton state with every label assigned to the sub-process-terms in step three. A transition consists of one receive action (if the sub-process-term starts with one) and all successive send actions that occur until the next receive action (or the end of the process term) occurs. The state associated with the next receive action (or the stop state of the SDL automaton) determines the target state of the transition.

[LMR98]: "Synthesizing ROOM Models from Message Sequence Chart Specifications"

The authors of [LMR98] describe an approach for synthesizing ROOM models from MSC specifications. They assume given one HMSC and a set of plain MSCs. The idea is that the HMSC describes the succession of the interactions represented by the plain MSCs, which display mutually exclusive scenarios. ROOM models consist of a structural and a behavioral description. The structural part fixes the components of the system, their syntactic interfaces, and their connections via channels. State machines describe the dynamic properties of the component they are associated with. To obtain a ROOM model from a given set of MSCs the authors first extract the structural information contained in the MSCs, i.e. the process identifiers, message signatures, and communication paths, and translate this information into corresponding ROOM actor classes, protocol classes, and bindings. For the derivation of state machines from the given MSCs the authors propose two strategies. Using the first, termed "maximum traceability", they obtain a high-level state machine from the HMSC by associating an automaton state with each plain MSC referenced in the HMSC. Transitions between the states thus obtained mimic the edges between MSC references in the HMSC. The next step is to expand each of the high-level states by deriving a state machine from the corresponding plain MSC. To this end, for each process referenced in the plain MSC, the authors split the sequence of actions of this process into transitions with intermediate states. Each transition consists of one receive action (or a substituting timeout action, if there is no receive action on the instance axis of the process) followed by the sequence of send actions until the next receive action or the end of the instance axis of the plain MSC under consideration occurs. The second strategy, termed "maximum progress", does not use the information contained in the HMSC to

obtain high-level state machines. Instead, the authors directly split the action sequences occurring on an instance axis (even across different plain MSCs) into transitions with intermediate states. Here, a transition consists of one receive action (or a substituting timeout action, see above) followed by the sequence of send actions until the next receive action in the MSC under consideration or in any MSC following sequentially according to the HMSC.

[KM93]: "Inferring State Machines From Trace Diagrams"

The authors of [KM93] present an incremental algorithm for inferring state machines (in the sense of OMT [RBP$^+$91]) from scenarios given as event traces. The algorithm takes as input a set of event traces and produces the minimal automaton, with respect to the number of states, for any of the components participating in the interaction. The traces are finite sequences of pairs of the form (a, e) where a and e are actions and events, respectively (in the sense of OMT), of the system under consideration. The meaning the authors of [KM93] associate with such a pair is that the object for which we construct the automaton sends event a to another object, and then expects to receive event e before it changes state. The traces contain no control-state labels; the authors discuss this as a possible extension of their work in [KM93]. The algorithm is based on backtracking, which results in a potentially exponential runtime complexity for generating the result automaton. It proceeds in a series of steps, reading the event trace resulting from the concatenation of all given scenario traces, from left to right. In each step it "consumes" one event pair (a_i, e_i), $i \in \mathbb{N}$ from the trace, and then moves on to the next event pair (a_{i+1}, e_{i+1}). Simply put, the consumption of the pair (a_i, e_i) proceeds as follows: if there is no state labeled a_i yet, then the algorithm adds such a state to the automaton. In any case, it adds a transition labeled e_{i-1} from the state labeled a_{i-1} to the one labeled a_i. By considering all action/event-pairs of the trace in order, the algorithm produces a minimal, deterministic automaton such that every one of the initially given scenarios reoccurs as a state/transition path in the automaton. Common subsequences of action/event-pairs of multiple scenarios end up as the same state/transition path in the automaton. Whenever, in a step, the algorithm tries to add a fresh state such that the resulting set of states is larger than necessary, i.e. there is a shorter state/transition path, which already "subsumes" the part of the trace consumed so far, then the algorithm backtracks. As the authors of [KM93] note, the merging of scenarios into state/transition paths as described above induces the problem that the resulting automaton may express more general behavior than the scenarios themselves. In [KMST96] the authors discuss extensions of their approach by adding support for scenarios with conditions and repetition; they also discuss consistency issues, induced on class diagrams by corresponding scenarios in form of an MSC variant. In [KSTM98] the authors present the application of the algorithm proposed in [KM93] within a tool for automated modeling of object-oriented software, called SCED, in detail. SCED allows the designer to create scenario diagrams, to infer state machines from these diagrams, and to simulate a system consisting of several state machines. During simulation the designer can add information by entering events corresponding to an incomplete or

missing state machine. The simulation runs completed in this way may then serve as the basis for adding information to the existing state machines using the incremental inference algorithm.

[HK99]: "Synthesizing Object Systems from LSC Specifications"

The work of [HK99] bases on the MSC dialect LSC [DH99], which we have discussed in Section 2.3.6. LSCs allow specification of existential sequence charts (roughly corresponding to MSCs under the existential interpretation in our approach) and universal sequence charts (roughly corresponding to MSCs under the universal interpretation in our approach). Moreover, for any location on any instance axis within an LSC we can determine whether it may or must be reached during an execution of the system. In particular, in an LSC specification, both existential and universal charts and locations can occur together. The authors of [HK99] present an algorithm for checking the consistency of an LSC specification (answering the question whether or not there is a system satisfying the specification), as well as schemata for synthesizing state machines from a consistent LSC specification. The algorithm for checking consistency proceeds by constructing a global, deterministic finite state automaton, whose states are associated with locations within LSCs that indicate when prefixes of an interaction have occurred. The transition function relates two such states if there is a message in the LSC such that extending the prefix represented by the source state yields the execution prefix represented by the target state; the transition function also respects the specification of liveness constraints on the locations in the LSCs. The automaton is global in the sense that it captures the behavior of all instances together. In the approach of [HK99] the occurrence of messages not fixed by any LSC is allowed at any time; besides this there is a correspondence between the automata obtained by the construction of [HK99] and the message flow graphs of [LL95, Leu95]. Whereas the latter consider both synchronous and asynchronous communication primitives, [HK99] treats only synchronous communication. After a few transformations on the global automaton, the algorithm of [HK99] decides whether all existential and universal charts are fulfilled by the automaton. The starting point for the synthesis of individual component specifications is also the global automaton constructed during consistency checking. The authors give several strategies for projecting the global automaton onto state machines for individual components, such that the synthesized state machines solve "non-local choice" problems by coordination (cf. also the remarks in [Leu95]).

Comparison

The following questions address the major differences between these approaches and our work:

- Can the developer guide the transformation by specifying automaton states already within the scenarios?

- What is the syntactic and semantic scope of the underlying scenarios, i.e. can they express, say, alternatives and repetition or safety and liveness properties?
- Can the scenarios overlap?
- Are the resulting automata deterministic or (possibly) nondeterministic?
- Are optimization schemes for the resulting automata necessary and available?
- How complex is the transformation?

We use these questions to structure our comparison between the transformation procedure we have introduced in Section 7.4, and the ones we have summarized above.

Guiding the Transformation by Guards Within MSCs

The major difference between our approach and those we have summarized above is as follows: none of the others directly takes control state information supplied by the developer inside the MSCs into account. As we have seen already, such state information can significantly reduce the number of states of the automaton produced. In their "maximum traceability" strategy, the authors of [LMR98] use HMSCs and the implicit states obtained from the MSCs referenced therein, as a partial remedy. In [KMST96] the authors also hint at a translation scheme that extends the one from [KM93] by taking conditions (which could encode control states) into account.

MSC Syntax and Semantics

Through the use of guards, and the "macro-expansions" we have introduced in Section 7.4.4, our approach allows the transformation of almost the full MSC dialect as presented in Chapter 4. In particular, we support sequential composition, alternatives, and repetition of MSCs. [LMR98] allows sequential composition, alternatives, and repetition via a given HMSC. [Fei99] allows neither sequential composition, nor repetition. The author treats each interworking as a separate alternative (modulo common prefixes). In [DH99] the authors hint at an encoding of alternatives and repetition in LSCs by means of properly chosen mandatory and optional locations within LSCs. On this basis the transformation of [HK99] would support, besides sequential composition, also alternatives and repetition of LSCs. [KM93] considers sequential composition and alternatives only; extensions for repetition appear in [KMST96].

MSC Interpretations

In our approach, we perform the transformation to automaton specifications based on the exact interpretation for MSCs. Thus, we do not mix interpretations when deriving automata. [HK99] allows both existential and universal charts as the source for their synthesis algorithm. This leads to a more complex consistency definition. [Fei99] uses an

interpretation matching our exact MSC interpretation; this approach considers the safety properties contained in the source Interworkings only. [LMR98] start out from mutually exclusive MSCs. If no HMSC is given, then the MSCs represent alternative executions, otherwise the HMSC fixes the relationship between the HMSCs as either of sequential composition, alternatives, or repetition. Here, too, the interpretation corresponds to our exact interpretation. The authors of [KM93] produce deterministic automata from possibly overlapping scenarios. This corresponds to our exact interpretation where the given scenarios are treated as alternatives.

Overlapping Scenarios

The MSCs that serve as the input of our transformation procedure can overlap; we simply have to treat the overlapping MSCs as the operands of a join construct. The author of [Fei99] considers all given interworkings as possibly overlapping finite scenarios, and always performs a join on them during the construction of the automaton specification. [LMR98] allows non-overlapping scenarios only. [KM93] allows overlapping scenarios, but the semantics of overlapping differs from ours. Whereas we identify all occurrences of identical messages in the arguments of a join (which leads to the notion of inconsistency with respect to join, if the arguments of a join specify different message orders for the identical messages), [KM93] treats MSCs whose projection on the identical messages differ in their message order as separate, alternative executions. Existential charts can always overlap in the approach of [HK99]. Universal charts with the same activation message (this is the external message triggering the occurrence of an interaction as depicted in the chart) can overlap only if their identical messages occur precisely in the same order; otherwise the LSC specification is inconsistent.

Nondeterministic versus Deterministic Automata, Optimization

The result automata of [KM93] are deterministic by construction; this is a tribute to the target automaton model from OMT. [Fei99], [HK99], as well as our procedure produce nondeterministic state machines as the result of their respective transformations. Similarly, the HMSC in the approach of [LMR98] could lead to nondeterministic behavior specifications. However, by augmenting the resulting ROOMCharts the authors let the developer resolve possible nondeterminism during execution time of the ROOM model.

In our approach, we treat optimization as a separate phase of the transformation; this allows us to work with state machines on various levels of efficiency with respect to their number of states and transitions. We place an intermediate state between any two input/output actions, and label the automaton's transitions with precisely one such action. This results in a nondeterministic automaton that we can then subject to well-known optimization algorithms from automata theory, such as determinization (for finding common prefixes in the automaton), and minimization (for finding common suffixes). We also suggest to identify maximal sequences of output actions with the input that triggers them, and to form a single transition out of such sequences. However, we consider this already a

heuristic optimization, because – depending on the underlying system model, or the intention of the developer – the outputs might just as well occur within disjoint time intervals. Moreover, after such a transition contraction the automata are no longer in a form permitting direct application of the above-mentioned optimization algorithms. The "maximum progress" algorithm of [LMR98] reduces the set of states and transitions by associating maximal sequences of output actions with a single input action whenever possible. [Fei99] directly associates maximal output sequences with their preceding input actions and thus produces contracted transitions directly (motivated by the SDL's requirement that at least one state be present between adjacent receive events). The "left contraction" and "right contraction" steps of [Fei99] roughly correspond to the determinization and minimization steps of our approach. [HK99] use the global, minimal, deterministic finite state machine that is consistent with the LSC specification as a starting point for projections onto the individual components; they mention three projection schemes that result in state machines of differing degrees of optimization, depending on how much information each machine must store to resolve non-local choices. The authors of [KM93] determine the optimal automaton (with respect to the number of states) that fulfills the given MSCs under the interpretation we have summarized above.

Complexity

A thorough treatment of the complexity of the various transformation approaches is beyond the scope of this thesis. The worst-case complexity of the algorithm in [KM93] is exponential in the length of the input MSCs due to potential backtracking. The authors of [HK99] claim that the run-time complexity of their transformation algorithm from the global system automaton A to the automata for individual components is polynomial in the size of A. They do not give a time estimate for the construction of A. [Fei99] also does not mention the complexity of automaton construction according to his approach. In our transformation procedure, the projection, normalization, and automaton construction phases are all linear in the length of the argument MSCs. The complexity of the optimization phase depends on which of the optimizations we actually carry out.

7.6. Summary

In this chapter, we have established the transition from interaction-oriented collaboration specifications for multiple components, as captured by MSCs, to specifications for individual components. We have addressed this transition from a more theoretical point of view in the context of relational component specifications in the A/C format, as well as from a more pragmatic, constructive point of view on the basis of a syntactic transformation procedure from MSCs to automaton specifications.

The A/C specifications we have derived starting from the black-box view on the component under consideration yield a separation of the overall responsibilities in a collaboration into

those of the component and the ones of its environment. Our major observation here is that we can derive the safety properties of the component fully schematically, whereas we have, in general, a spectrum of possibilities for splitting liveness responsibilities between the component and its environment. We have exploited the A/C format to study the effects of property refinement steps on entire MSCs with respect to the corresponding individual component specifications.

The link between MSCs and A/C specifications is completely general, there are no restrictions with respect to the syntax of the MSCs we start out with. Because of their generality A/C specifications serve particularly well for formal reasoning about component properties induced by MSC specifications.

In a more pragmatic approach, aiming at *implementing* the interaction patterns captured by MSCs, we have defined a transformation procedure from MSCs to finite state machines. The procedure operates entirely on the syntax of the MSCs and automata, and is thus independent of any concrete semantic model. To establish a close connection between the derived automata and their source MSCs we have, nevertheless, defined both a syntax and a semantics for the target automata.

The syntactic transformation consists of four phases:

1. projection of the given MSCs onto the component of interest,
2. normalization of the MSCs, i.e. adding missing start and end guards, and splitting MSCs with more than two guards at an intermediate guard,
3. transformation into an automaton by identifying the MSCs as transition paths, and by adding intermediate states accordingly,
4. optimization of the resulting automata.

After the presentation of this transformation procedure for a very restricted syntactic form of MSCs we have sketched the procedure's extension to almost the entire MSC dialect from Chapter 4.

The major limitation – and potential for future improvement – is that the transformation considers control states only, instead of a combination of control and data states.

The applications of the results of this chapter are manifold; we list a few of them explicitly, to give a coarse overview.

First, we can use either approach for the derivation of prototypes for the component of interest at all levels of detail within any development phase. In particular, the availability of a feature for *automatic* "code generation" from MSCs can significantly reduce the discontinuities arising from manual implementations of interaction requirements; because of its potential for tool-support it may even increase the acceptance of MSCs as a description technique in industrial contexts in the first place.

Second, we can use the automata resulting from the transformation procedure also as test-drivers for the component under consideration. Automatically generated test drivers and test sequences are extremely important in contexts where requirements keep changing: implementing tests manually after each modification of the requirements is typically infeasible. The transformation procedure establishes a traceability of the requirements captured by means of MSCs, and the properties implemented via the component automata; this way, the test drivers remain consistent with their corresponding components despite changing requirements.

Third, both the A/C format and the derived automata make MSCs amenable for formal verification tools such as theorem provers and model checkers. The work of this chapter enables using MSCs also as an intuitive, accessible language for expressing system and component properties; the existence of more accessible specification languages has the potential of increasing the acceptance of formal verification environments beyond the academic context.

In summary, the transition from MSCs to individual component specifications as we have discussed it in this chapter closes the methodical gap between global system specifications, and local component specifications.

CHAPTER 8

Summary and Outlook

In the preceding chapters we have studied the semantics of MSCs, corresponding refinement notions, the use of MSCs for property specifications, and the transformation from MSCs to behavior specifications for individual components. Here, we summarize the results we have obtained, and give directions for further work.

Contents

8.1. Summary **296**

8.2. Outlook **298**

8.1. Summary

In the introduction to this thesis we have set out to work towards the seamless integration of MSCs into the development process for reactive distributed systems. As important steps into this direction we have identified a thorough understanding of a sufficiently expressive MSC semantics and of the properties represented by MSCs, effective refinement notions, and transformation schemes from MSCs to individual component specifications. In the following paragraphs we recapitulate and relate the results we have obtained so far.

Based on the mathematical notion of streams we have defined a formal model for reactive distributed systems that combines structural system aspects (component distribution) with dynamic system aspects (component state and interaction over time). The selection of this system model was guided by the modeling scope of MSC specifications. MSCs represent system structure via separate component axes, as well as system behavior via message arrows and their ordering. The model we chose consists of components, directed channels between components, and infinite valuations of these channels. Component states, the other "ingredient" of our system model, were motivated by our intention of relating MSCs with individual component specifications. The integration of states into our model allowed us to give a semantics for conditions within MSCs. By means of conditions representing control states of the individual components we have established a smooth transition from the MSC semantics to the semantics of state-oriented component specifications.

Our semantic model allowed us to define all major constructs of the standard MSC-96 succinctly; we have deliberately avoided to incorporate MSC-96's delayed choice and weak sequential composition in favor of a simpler diagram interpretation. We have also gone beyond the standard syntax and semantics; guarded and unbounded finite repetition, the join and trigger composition operators for the specification of overlapping message sequences and fairness constraints, as well as the treatment of preemption, message parameters, and parametric MSCs are examples of such extensions. The selection of MSC-96 as the basis for our investigations was motivated by this notation's expressiveness: it allows specification of alternatives, finite and infinite repetition, and parallelism. In addition, MSC-96 provides the concept of High-Level MSCs as a structuring mechanism for MSC specifications; most other MSC dialects support only a subset of these operators and mechanisms in their syntax and semantics. Our aim, however, was to model reactive distributed systems (including infinite behavior), and to use MSCs not only as scenarios but also for the specification of complete component behavior; hence, we selected MSC-96 as the starting point for a correspondingly expressive MSC notation.

Systematic refinement of specifications and designs is an alternative to ad hoc development steps that leave open whether the result of a particular development activity still displays some or all of the properties of the model as it was before the step was carried out. Based on our system model we have defined and investigated several refinement notions for MSCs; these refinement notions address all system aspects captured by MSCs. By means of property refinement we can reduce the nondeterminism contained in an MSC

specification. This corresponds to fixing design choices in the system under development. Message refinement allows us to adjust the granularity of individual messages. Similarly, structural refinement addresses the hierarchical decomposition of components. In addition, we have discussed the binding of references as a systematic way of using MSCs in stepwise top-down system development. Given these four refinement notions we can express system properties at the appropriate level of detail by means of MSCs – without having to resort to other description techniques. This is particularly important for using MSCs systematically across multiple development phases. Adjusting the appropriate level of detail also significantly facilitates discussing specifications and designs among different groups of developers – each with a specific technical background and responsibility for a particular system part – as is typical in industrial software and system development today.

Our investigation of different MSC interpretations in the range from simple scenarios to complete system behavior has clarified the different roles MSCs can play across the entire development process. The bare MSC semantics is quite loose: it only requires the existence of a time interval in a corresponding system execution where the depicted messages occur in the specified order – possibly among arbitrary (other) messages. The existential, universal, exact, and negated MSC interpretations relate an MSC's semantics with (other forms of) system specifications. The existential interpretation, which corresponds to the classical role of MSCs as scenarios, requires that the behavior specified by the MSC under consideration can, but need not occur as part of the system's behavior. The universal interpretation requires the behavior represented by the MSC to occur eventually in every system execution. The exact MSC interpretation is even more restrictive: it allows only system behaviors corresponding precisely to what is given by the MSC. This interpretation is the basis for transiting from scenarios to complete component behavior. An "anti-scenario", i.e. an MSC under the negated interpretation, shows what must not happen in any system execution. The investigation of these various interpretations of MSCs is far from being purely academic. In fact, every tool with MSC support in use in industry bases on one of these interpretations. The lesson we learn from this multiplicity of MSC interpretations is that making the intention we follow with an MSC specification explicit is a prerequisite if the MSC specification is to be of any (formal) value during the development process.

We have also analyzed the properties expressed by MSCs along the classical notions of safety and liveness. As a result, we have obtained that our MSCs specify mainly liveness properties and thus nicely complement description techniques for safety specifications. This fits in with the intuition we relate with the arrows in MSCs: the depicted messages do occur in the specified order.

Our discussion of MSC interpretations and properties has prepared another central contribution of this thesis: the derivation of individual component specifications from collections of MSCs. Expressing system requirements by means of MSCs is of limited value if the captured requirements serve only as more or less accurate documentation during requirements capture. Yet, most development methods and their supporting CASE tools today provide systematic design transformations and code generation mechanisms at most for

other description techniques, such as class diagrams and automata. This forces the developer to leave the MSC notation quickly for carrying out the "real" component development steps. The translation schemes from MSCs to A/C and automaton specifications we have described in our work help bridge the gap between the interpretation of MSCs as inter-component scenarios and as complete behavior descriptions for individual components. The schematic A/C specifications we derive from MSCs provide a handle at using our full MSC dialect in connection with formal validation techniques. Still, the A/C specification address mainly the black-box component behavior, i.e. the relation between a component's input and output histories. We also have shown how to produce a state-oriented model for the glass-box behavior of individual components by the syntactic transformation from MSCs to automaton specifications. The constructed automata can serve as "jump-start" behavior specifications, ready for detailed individual component design. Although our transformation procedure is purely syntactic, we have also established a semantic relationship between the MSCs and the corresponding automata via the exact MSC interpretation. Thus, the transformation from MSCs to automata directly links and integrates our work on MSC semantics, MSC refinement, and MSC interpretations.

Together, these results increase the seamlessness of the MSC's integration into the overall development process for reactive distributed systems. Clearly, however, there is still much work left to do. In the next section we briefly discuss areas for further exploration.

8.2. Outlook

In Chapters 4 through 7 we have laid the foundation for future work of both theoretical and practical nature. This includes, in particular:

1. extensions to the syntax and semantics of MSCs,
2. integration of the suggested refinement techniques and transformation procedures into existing development processes,
3. integration of MSCs into validation technologies,
4. tool support,
5. extended case studies.

In the following paragraphs we briefly discuss each of these areas, in turn.

Syntactic and Semantic Extensions

The MSC dialect we have defined in Chapter 4 is quite expressive already. However, there are several modeling situations where we might welcome additional syntactic and semantic

concepts. Examples are the specification of synchronous messages, procedure calls and broadcasting communication. Typically, procedure calls combine data- and control-flow between components. One way of coping with both of these aspects is to add corresponding invariants to the system model [RGG99]. The need for broadcasting arises frequently in component-oriented development, be it as a requirement of the underlying system model (as is the case with statecharts [Har87, HK99] and (other) synchronous programming languages such as [BG88, Mar92]), or as a deliberate design choice. An example for the latter is the Observer design pattern (cf. [GHJV95]), where an object broadcasts changes of its internal state to all observers having registered with the object. Observers can subscribe and unsubscribe to the notification about the object's state change at any time. This poses a challenge at the graphical notation of MSCs: if each axis represents a single concrete object we can capture at most scenarios of such situations with MSCs. Otherwise we would have to know in advance what observers are registered with an object at any point in time. The UML, for instance addresses this problem by distinguishing sequence diagrams where the axes represent concrete objects from others where the axes represent object roles or classes (cf. [RJB99]); clearly, such an approach requires careful analysis and investigation.

Another direction for syntactic and semantic extensions is the incorporation of notation and concepts for the specification of mobile systems. The challenge here is to represent changing component configurations syntactically and semantically. Instance start and stop, as we discuss it in Appendix A, gives only a first idea of how we can capture such changes by means of invariants. However, the leeway we have for representing changes in structure in a two-dimensional description technique – with time being the one dimension and structure being the other one – is quite limited. Obviously, we can always resort to tool support and multiple dimensions. Whether this indeed leads to more readable specifications is rather questionable.

Indeed, adding "gimmicks" to graphical description techniques always comes at a price. Reducing the complexity of the graphical representation is typically accompanied by a significant increase in semantic complexity. Shifting expressiveness between different description techniques to prevent visual overload often increases the complexity of the induced context conditions. Finding the right balance between how intuitively comprehensible a (graphical) description technique and its artifacts are, and how expressive the description technique is, is a difficult task. The UML suggests the use of the Object Constraint Language (OCL) for the specification of context conditions for graphical models (cf. [RJB99]). The combination of MSC specifications with textual OCL constraints might be an interesting starting point for moderate increases in expressiveness without loosing the MSC's intuitive appearance.

MSCs and Development Processes

The integration of the methodical MSC usage into concrete development methods and processes is another promising area of future work. An interesting application of our work is to assign meaning to connector specifications as they appear, for instance, in

various forms in Catalysis [DW98], and ROOM [SGW94]. ROOM's "protocol classes", for instance, consist only of a static list of input and output messages that describe a certain aspect of a ROOM component's interface in the form of message names and signatures. Protocol classes do not constrain the dynamic aspects of the collaborating components. Using MSCs to specify both the static and the dynamic protocol aspects, and deriving the specifications of each component's assumptions and commitments directly from the MSCs results in a much more expressive form of protocol classes, and in a much better integration of MSCs into ROOM. Similarly, we could enhance the Interface Description Languages (IDLs) used in middleware technologies like CORBA, Java RMI, and DCOM by supplying (semi)automatically derived behavior descriptions in addition to the static method signatures in use today.

Catalysis makes use of pre- and postconditions to clarify the context in which an interaction specification is applicable. Combining these pre- and postconditions with the assumptions and commitments we derive schematically from MSCs also seems promising.

The MSCs' expressiveness significantly exceeds description techniques whose only target is scenario specification. In fact, the border between scenario and use case specifications by means of the MSC dialect we use is quite fuzzy. This suggests to investigate to what extent we can use our semantic model and the MSCs as such to give an adequate semantics to existing specification techniques for use case or activity specifications (as they appear, for instance, in the UML [RJB99]).

MSCs and Validation

An important application of our thorough investigation of the MSC semantics is their integration into verification technologies. The individual component specifications we obtain schematically from MSCs provide an obvious starting point for applying theorem provers (in the case of general A/C specifications) and model checkers (in the case of finite-state automaton models). This way we can check whether an MSC specification is consistent with respect to (additional) requirements at the individual components. A similarly direct application of our transformation procedures is the derivation of test drivers for the depicted components.

Another possibility for integrating MSCs into validation is to view them as abstractions of more detailed system behavior, and to exploit the coordination information they capture. Assume we start out with a network of components whose behavior is specified by a set of automata, and have given a property to verify against the component network. The idea is to come up with MSCs capturing the relevant interactions in the network with respect to the property under consideration, such that proving the property by "inspection" of the MSCs is easier than proving the property by considering each individual component separately.

Instead of using MSCs for a posteriori verification we could also extend the work on refinement and property preserving MSC transformations we have started here to obtain a

description technique supporting specification and design for validation. We can imagine, for instance, a combination of the techniques we have explored in this thesis with the interaction specifications of design patterns as they appear in [GHJV95] and [BMR+96]. Proving interaction properties once and for all for certain interaction patterns, and providing the corresponding designs in a library of "design components" or frameworks has the potential for increasing the quality of specifications making use of such a library.

Tool Support

To obtain the maximum benefit of the automatic and semi-automatic transformations from MSCs to individual component specifications we must integrate these transformations into corresponding development tools. The transformation from MSCs into automata is, however, particularly easy to implement, because it is purely syntactic, and bases on well-known algorithms from automata theory.

Similarly important is the tool support for the refinement notions and rules we have given in this thesis. Despite the increasing popularity of MSCs typical development tools of today offer only little methodical support for performing systematic development steps on MSCs.

Case Studies

The investigation of case studies of substantial size is also an important aspect of future work. This includes an assessment of the quality of the automata obtained from the transformation procedure. This might lead to more elaborate heuristics for automaton optimization, as well as to the incorporation of more elaborate automaton models than the one we have employed here. Producing automata with hierarchic states, as statecharts and ROOMCharts provide them, is one interesting starting point for obtaining more readable results from large MSC specifications.

Finally, we consider a thorough treatment of data states, in addition to pure control states as we use them in our simplified transformation procedure, a source for further improvement of our approach towards more succinct automaton specifications.

APPENDIX A

Syntactic And Semantic Extensions

In Chapter 4 we have introduced the “core” of the MSC syntax and semantics used in this thesis. We kept the core notation intentionally small in size to ease the semantics definition, as well as the investigation of its properties. Despite its size, the notation we have at our disposal already is quite powerful as its proximity to linear time temporal logics indicates (cf. Section 4.5.4).

Here, we present several extensions of the core syntax and semantics that increase the usability of our notation in various respects. Sections A.1 and A.2 contain sketches of how to deal with instance creation and deletion, and timers, respectively, in our framework. We discuss the treatment of messages with parameters, as well as the substitution of channel, message, and parameter names in Section A.3. Action specifications and gates are the topics of Sections A.4 and A.5, respectively.

A.1. Instance Start and Stop

The syntax of MSC terms does not provide means for specifying the starting or stopping of instances. Here, we discuss how we can integrate these two concepts into our framework even without changes to the syntax.

The basic idea of our treatment of instance start and stop is that no instance can participate in interactions before its creation, and after its deletion, respectively. Therefore, we can augment the system model of Section 4.2 with an invariant ensuring this property.

To this end, we structure the set of messages such that it equals the union of the three pairwise disjoint sets $\{start\}$, $\{stop\}$, and M', where M' is the message set we have introduced in Section 4.2:

$$M = \{start\} \cup \{stop\} \cup M'$$

Starting and stopping an instance corresponds to sending message *start* and *stop*, respectively, on one of the input channels of this instance. The invariant we define below characterizes system executions respecting the intuitive requirement stated above: no message can occur before a start message, and no message can occur after a stop message on any channel to which the instance under consideration is connected.

Given a system specification $Sys \subseteq (\tilde{C} \times S)^\infty$ we require the validity of the following invariant:

$$\begin{array}{l}\langle \forall s, p : s \in Sys \ \wedge \ p \in P : \varphi \in s \Rightarrow \langle \forall ch : ch \in C : \\ \quad \langle \forall t' :: stop \in \pi_1(\varphi).t'.ch \Rightarrow \langle \forall t'', ch' : t'' > t' \ \wedge \ ch' \in chans.p : \pi_1(\varphi).t''.ch' = \text{<>} \rangle\rangle \\ \quad \wedge \langle \forall t' :: start \in \pi_1(\varphi).t'.ch \Rightarrow \langle \forall t'', ch' : t'' < t' \ \wedge \ ch' \in chans.p : \pi_1(\varphi).t''.ch' = \text{<>} \rangle\rangle\rangle\rangle\end{array}$$

Here, we use function $chans : P \to \mathcal{P}(C)$, which associates with every instance $p \in P$ the set of its channels.

The start or stop message can come from any instance; we do not exclude the instance to be started or stopped from the set of possible message originators. Whether this makes sense depends on the system (model) under consideration. More restrictive requirements result in strengthening the above invariant. As an example, recall from Chapter 2 that in MSC-96 only "self-destruction" is allowed, whereas the UML's SDs allow any object with a corresponding association to destroy another object.

A.2. Timers

As we have described in Section 2.2, we can understand timers as a special kind of instances with one input and one output channel, whose interface consists of messages *set*, *reset*, and *timeout* only. *set* and *reset* are allowed on the incoming channel of the timer only, whereas *timeout* can occur only on the timer's outgoing channel.

The ordering of messages between the instance using the timer, and the timer itself then corresponds to the ordering of the respective timer events of MSC-96.

Remember that we use only a qualitative notion of time in the system model of Section 4.2. The "time" in timed streams serves only the purpose of expressing precedence or causality between messages or message sequences.

One way of adding "real-time" to MSC specifications is to associate a real-time clock (modeled, for instance, as a stream over the nonnegative real numbers) with each timer and to add invariants on the points in real-time where timeout events can occur on the outgoing channels of the timer.

A.3. Message Parameters and Parametric MSCs

So far, we have not assigned a meaning to messages with parameters. In MSC-96 the meaning of parameters is void (cf. [IT96, IT98]). Here, we sketch an approach to integrating parameters into our semantic framework. Such an integration allows us not only to formulate properties about control and message flow in the systems we study, but also to characterize quantitative data-flow. Our treatment of parameters in this section was inspired by, and extends similar approaches in [BHKS97a] and [Fac95].

Allowing parameters in message specifications is one way of parameterizing MSCs; another one is to enable the substitution of channel, message and parameter names in MSC terms, which allows reuse of the substituted terms in different contexts. This can reduce MSC specifications significantly.

Messages with Parameters

The assignment of meaning to message parameters requires three minor modifications to the semantics from Section 4.4. First, we have to establish a link between the formal parameter specifications occurring in the MSCs, and the concrete parameter values associated with messages during an execution. Second, we must alter the message semantics such that it allows all possible parameter value assignments. Third, we have to decide on how to handle constants, as well as message parameters occurring more than once within the same MSC term. In the following paragraphs we deal with each of these issues in turn.

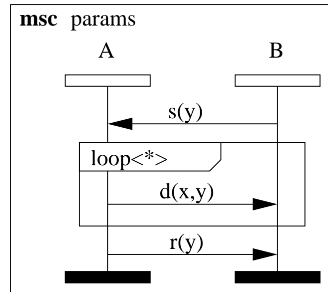


Figure A.1.: Messages with formal parameters

Figure A.1 shows messages with parameters; y occurs in the parameter lists of s, d, and r, whereas x occurs only in the parameter list of d.

We call the parameters occurring within the parameter list of a message in an MSC *formal* parameters; the values associated with the message during an execution of the system are the *actual* parameters of this message. To make this distinction precise we introduce the set $\tilde{M}$ of messages with formal parameters; we define: $\tilde{M} \subseteq MN \times FPL$. Here, MN denotes the set of message names (a text string), and $FPL \stackrel{\text{def}}{=} \{\emptyset\} \cup \bigcup_{i \in \mathbb{N}}([0, i] \to PN)$ models empty

and nonempty parameter lists; in this definition, PN denotes the set of formal parameter names. A parameter list is a mapping from index positions to formal parameter names. As an example, consider the message label *d(x,y)* in Figure A.1. We model such a label as the element $(d, \{0 \mapsto x, 1 \mapsto y\}) \in \tilde{M}$, with $d \in MN$ and $x, y \in PN$. For easy access to the names of message parameters, we define the selector function *pars* as follows:

$$m = (m_1, m_2) \in \tilde{M} \Rightarrow m.pars = \{\xi_1 : (\xi_0, \xi_1) \in m_2\}$$

Actual messages, i.e. messages carrying actual values instead of formal parameters, are elements of set $M \subseteq MN \times APL$, where $APL \stackrel{\text{def}}{=} \{\emptyset\} \cup \bigcup_{i \in \mathbb{N}}([0, i] \rightarrow PV)$ models empty and nonempty lists of parameter values. We assume that every parameter occurring within a parameter list has some specific type. We do not write out these types in MSCs; rather, we expect the specification of parameter types to be part of another document, such as a class diagram or interface specification. For simplicity we assume given a function *tdom* whose purpose is to map parameter names to the domains of their types. PV is the set of parameter values; we define it as the union over all domains of the types associated with the parameter names: $PV \stackrel{\text{def}}{=} \bigcup_{pn \in PN} tdom.pn$. For easy access to the constituents of an actual message, we define the two selector functions *mn* and *pval* as follows:

$$m = (m_1, m_2) \in M \Rightarrow m.mn = m_1 \;\wedge\; m.pval = m_2$$

Because of the direct correspondence between formal and actual parameter lists, we furthermore use *pval* as a function from formal parameter names to parameter values: $pval : PN \rightarrow PV$, instead of as a function from index sets to parameter values.

With these preliminaries in place we can define function $dom : \tilde{M} \rightarrow \mathcal{P}(M)$, which associates with a formal message the set of all actual messages, i.e. the set of all messages with the same name and arity where concrete values replace the formal parameters.

This allows us to modify the semantics definition for messages from Section 4.4 as follows:

$$\begin{aligned} [\![ch \triangleright m]\!]_u \stackrel{\text{def}}{=} \{&(\varphi, t) \in (\tilde{C} \times S)^\infty \times \mathbb{N} : \\ &t = \min\{v : v > u \wedge \langle \exists m' : m' \in dom.m : m' \in \pi_1(\varphi).v.ch \rangle\}\} \end{aligned}$$

Here, m is an element of set $\tilde{M}$, i.e. a message with a formal parameter list.

These slight modifications yield a new MSC semantics that integrates messages with parameters into our framework. We can use this integration, for instance, to constrain the data flow in the executions modeled by an MSC. To this end, we can use predicates $p : (\tilde{C} \times S) \times \mathbb{N}_\infty \times \mathbb{N}_\infty \rightarrow \mathbb{B}$ and define

$$[\![\alpha]\!]_u^p \stackrel{\text{def}}{=} \{(\varphi, t) \in [\![\alpha]\!]_u : p.\varphi.u.t\}$$

As an example, consider the MSC of Figure A.1; it corresponds to the MSC term

$$ch \triangleright s(y); (ch' \triangleright d(x, y))\uparrow_{<*>}; ch' \triangleright r(y),$$

which we abbreviate by α. We might want to constrain the data flow modeled by this term such that parameter y carries the same value in all occurrences of message d. Predicate

$$
\begin{aligned}
p_{equal}.\varphi.u.t \stackrel{\text{def}}{=} \; & \langle \exists v : v \in tdom.x : \\
& \langle \forall t', m : u \le t' \le t \;\wedge\; m \in \pi_1(\varphi).t'.ch : m.mn = d \Rightarrow (m.pval).y = v \rangle \rangle
\end{aligned}
$$

formalizes this requirement. Thus, $[\![\alpha]\!]_u^{p_{equal}}$ denotes the set of executions that correspond to α and fulfill p_{equal}.

This example already hints at an inconvenience introduced by the modified semantics for messages. As it stands, this semantics allows completely arbitrary value assignments to parameters in system executions. If a formal parameter occurs multiple times within an MSC, such as parameter y in Figure A.1, and we want to express that each occurrence carries the same value, we have to introduce explicit invariants such as p_{equal} in the example above. To fix this we could define that all occurrences of parameters within an MSC term must carry the same actual value during system execution. This, however, is clearly too restrictive, as the MSC of Figure A.1 suggests. Only message d references parameter x; this message, however, occurs within a loop construct. We identify two sensible interpretations in this state of affairs: during system execution an arbitrary finite number of messages d with the same (interpretation 1) or different (interpretation 2) values of x occur. The liberal treatment of parameter values makes it easy to yield the second interpretation, whereas it renders the first interpretation hard to represent. The restrictive parameter treatment eases yielding the first interpretation, whereas it makes representing the second interpretation (almost) impossible.

Mixing both interpretations seems also sensible. Consider MSC *params* as the specification of a database transaction. B initiates the transaction by sending message *s(y)* with the transaction number y to A. A then sends an arbitrary finite number of messages *d(x,y)* to B, where x represents a data value that may differ from one loop iteration to the next, whereas y still denotes the same transaction number supplied by B. Finally A closes transaction y by sending message *r(y)* to B.

To allow both interpretations we structure the set of parameter names *PN* into two disjoint sets: $PN \stackrel{\text{def}}{=} PN_{\text{Fixed}} \cup PN_{\text{Flexible}}$ with $PN_{\text{Fixed}} \cap PN_{\text{Flexible}} = \emptyset$. We interpret PN_{Flexible} as the set of parameter names whose concrete values can change during the part of the execution covered by the MSC under consideration; PN_{Fixed} denotes the set of parameter names whose values remain fixed during this execution segment.

Given this distinction, we introduce an MSC term $\exists_{Pars} : \alpha$ whose semantics is, intuitively, equal to the one of α with the exception that for the time period covered by α all formal parameters contained in set $Pars \subseteq PN_{\text{Fixed}}$ have the same value. The parameter names in PN_{Flexible} can change arbitrarily from one occurrence of the corresponding message to

the next. Formally, we define:

$$
\begin{array}{rcl}
[\![\exists_{Pars} : \alpha]\!]_u & \stackrel{\text{def}}{=} & \{(\varphi, t) \in [\![\alpha]\!]_u : \\
& & \langle \forall pn : pn \in Pars : \\
& & \ \langle \exists v : v \in tdom.pn : \\
& & \ \ \langle \forall t', ch, m : u \leq t' \leq t \ \wedge \ ch \triangleright m \in msgs.\alpha : \\
& & \ \ \ \langle \exists m' : m' \in dom.m : \\
& & \ \ \ \ m' \in \pi_1(\varphi).t'.ch \ \wedge \ pn \in m.pars \Rightarrow (m'.pval).pn = v\rangle\rangle\rangle\rangle\}
\end{array}
$$

Note the similarity between this operator and predicate p_{equal} in the example, above.

The set PN_{Fixed} can also hold the names of constants. We model constants by parameter names with a singular domain type holding precisely the constant's value. This allows us to mix constants freely with formal parameters in message headers.

Using $[\![\exists_{.} : .]\!]_u$ instead of simply $[\![.]\!]_u$ in all semantics definitions of Section 4.4 yields the desired integration of parameters into our semantic framework. The only additional information we require for an MSC with parameters is to what sort (PN_{Fixed} or PN_{Flexible}) each parameter belongs.

Clearly, we could extend this integration of parameters into our MSC syntax and semantics further. A semantic extension would be, for instance, to use the values of message parameters in guards of loop constructs. On the syntactic side we could add inline expressions that declare parameters as either flexible or fixed. Because, in this thesis, we do not work extensively with parameters, we refrain from a further elaboration of this topic.

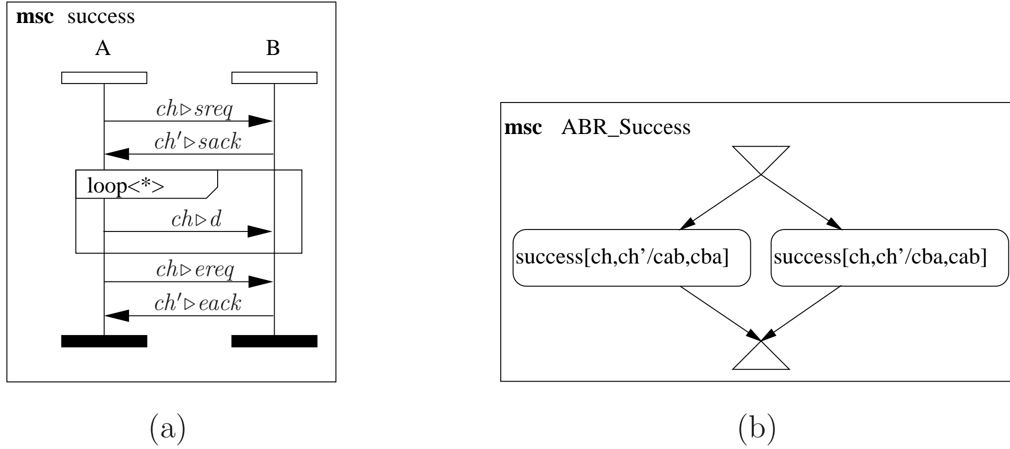
Parametric MSCs

Formal parameters are a means for abstracting from concrete parameter values, which means that we need to draw only a single MSC with formal parameters to capture all possible value assignments to the formal parameters as well.

Another way of reducing the number of MSCs needed for the specification of (part of) a system behavior is to use substitution, i.e. to take an existing MSC and, say, modify the name of one channel or message occurring in it, to yield a new MSC.

As an example, consider MSC *success* in Figure A.2 (a). Imagine that the depicted interaction sequence is part of a symmetric communication protocol between instances A and B. The protocol consists of three phases: establishing a connection (through messages *sreq* and *sack*), transmitting data (via message d), and tearing down the connection (through messages *ereq* and *eack*). Because the protocol is symmetric we would have to draw two MSCs similar to *success*, one where instance A initiates the transmission, the other where instance B is the initiator.

To increase the amount of reuse achievable within MSC specifications we allow the use of the following three syntactic substitution operators (cf. [IT96, IT98] or Section 2.2 for the corresponding substitution operators of MSC-96):



(a) (b)

Figure A.2.: Generic MSC and substitutions

$\alpha[ch'/ch]$	substitutes channel name ch' for ch in α
$\alpha[pn'/pn]$	substitutes formal parameter name pn' for pn in α
$\alpha[mn'/mn]$	substitutes message name mn' for mn in α

We omit the obvious definitions by induction on the structure of α for each of these substitution operators. Each of these definitions extends in the usual way to lists of simultaneous substitutions of the form $x'_0, x'_1, \ldots / x_0, x_1, \ldots$, where x'_0 is the substitute for x_0, x'_1 is the substitute for x_1, and so on.

Using the operator for channel substitution allows us to reuse MSC *success* in the specification of the symmetric communication protocol above (cf. Figure A.2 (b)).

A.4. Actions

As we have described in Section 2.2, MSC-96 allows specification of local actions on individual instance axes. Besides their presence in the ordering relation imposed by an MSC on the events it contains, however, the meaning of actions is void in [IT98].

Regarding actions as state transformers in the sense of [DS90] allows us to integrate them easily into our semantic framework. We introduce a new MSC term $\mathbf{act}(p,q)$. Intuitively, predicates p and q are pre- and postconditions describing the assumption under which the action executes, and the effect its execution has on the state space, respectively. For reasons of simplicity we consider terminating actions only. Then we define

$$\begin{aligned}[\![\mathbf{act}(p,q)]\!]_u \stackrel{\mathsf{def}}{=} \{&(\varphi,t) \in (\tilde{C} \times S)^\infty \times \mathbb{N} : \\ &\langle \exists t', t'' : u \le t' < t'' \le t : \pi_2(\varphi).t' \in [\![p]\!] \ \wedge \ \pi_2(\varphi).t'' \in [\![q]\!] \rangle \}\end{aligned}$$

A simple example of an action specification is $\mathbf{act}(\text{true}, a = 5)$, where we assume that "$a$" is a local variable of the instance on whose axis the action specification occurs. This action corresponds to setting the value of variable a to 5.

If we want to mimic MSC-96's action usage, we can constrain the predicates p and q such that they refer to local variables of a single instance only. Otherwise, we use more elaborate pre- and postconditions to describe joint actions of multiple components (cf. [DW98]).

A.5. Gates

Message gates in MSC-96 serve the purpose of specifying the continuation of message arrows outside the scope of an MSC, such that we can depict the sending of a message in one MSC, and the receipt of the same message in a different one (cf. Section 2.2). The notion of gates allows establishing a link between a send event and the corresponding receive event across MSCs.

Because we associate a particular channel, connecting sender and receiver, with every message, we do not need an extra gate concept in our approach to describe the continuation of messages. The channel *is* the link between sender and receiver.

Thus, the only remaining question is how to connect an MSC containing the sending of a message with another MSC containing the receipt of the same message. This is precisely the purpose of the join operator $\otimes$, whose semantics we have defined in Section 4.4: if, say, MSCs α and β each contain an occurrence of message $ch \triangleright m$, then $\alpha \otimes \beta$ identifies these two occurrences, just as the corresponding MSC-96 gates link the send event with the appropriate receive event.

Thus, we can model message gates in our framework without modifications to the semantics.

As we have argued in Section 2.2, we can understand general orderings as a special sort of message arrows, labeled with (anonymous) ordering messages. Similarly, we can consider *order gates* as special forms of message gates. Therefore, we can integrate order gates without changes to the semantics definition by partitioning the set of messages M into the set of ordering messages and the set of other, regular messages. An ordering arrow then gets an "implicit" label from the set of ordering messages.

If needed, we can filter out the ordering messages from each execution in the set $[\![\alpha]\!]_u$, and obtain those executions obeying the message order imposed by the ordering arrows, but without the corresponding ordering messages.

APPENDIX B

Proofs

In Chapters 4 through 7 we have omitted most of the proofs for better readability of the "main" text. In the following sections we make up for this omission. Section B.1 contains the proofs concerning properties of the semantics in general (such as independence of absolute time, and well-definedness), as well as of the semantics of MSC operators in particular (such as associativity, and distributivity). This covers the claims made in Chapter 4. In Section B.2 we prove the validity of the refinement rules stated in Chapter 5. We deal with safety and liveness properties in the context of MSC specifications in Section B.3; the proofs we give there correspond to the claims in Chapter 6. In Section B.4 we discharge the proof obligations on time guardedness and join consistency of MSCs from Chapter 7. Table B.1 summarizes the mapping between chapters in the main text and sections of the appendix for ease of reference.

Chapter in the main text	Part of the appendix containing the proofs
Chapter 4	Section B.1 (page 312)
Chapter 5	Section B.2 (page 324)
Chapter 6	Section B.3 (page 328)
Chapter 7	Section B.4 (page 355)

Table B.1.: Mapping between chapters and the corresponding section of the appendix

Where appropriate we employ the calculational proof style proposed in [DS90].

B.1. Properties of the MSC Semantics

In Chapter 4 we have stated several properties of the MSC semantics introduced there. In particular, we have

- claimed the independence of absolute time of the semantics function $[\![.]\!]_u$,
- listed symmetry, associativity, and distributivity of MSC operators,
- asserted the well-definedness of the semantics,
- discussed the difference between sequential composition and interleaving in our MSC notation.

This section contains the proofs corresponding to these topics. First, we deal with the semantic mapping's independence of absolute time. Second, we give a sequence of propositions, one for each of the operator properties stated in Section 4.4. Third, we deal with the well-definedness of the semantics mapping, and mention a proof principle for determining properties of operators defined via greatest fixpoints. Fourth, we prove that in the MSC semantics defined in Section 4.4 sequential composition and interleaving never coincide for non-empty MSCs.

B.1.1. Independence of Absolute Time

Proposition 18 *For all $\alpha \in \langle MSC \rangle$, $n, u \in \mathbb{N}$, $t \in \mathbb{N}_\infty$, and $(\varphi, t) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty$ the following equivalence holds:*

$$(\varphi, t + n) \in [\![\alpha]\!]_{u+n} \equiv (\varphi \uparrow n, t) \in [\![\alpha]\!]_u \qquad \square$$

Proof For $n \in \mathbb{N}$ the claim follows by induction on n. The base case $n = 0$ is trivial. For the inductive step we first observe without proof that for all $u, t \in \mathbb{N}$ we have $\varphi \uparrow (u + t) = (\varphi \uparrow u) \uparrow t$ and $\varphi \uparrow (u + t) = (\varphi \uparrow t) \uparrow u$. Now we assume that the claim holds for all n' with $0 \leq n' \leq n$ and derive

$$
\begin{aligned}
& (\varphi \uparrow (n + 1), t) \in [\![\alpha]\!]_u \\
\equiv \quad & (\ast \text{ above observation } \ast) \\
& ((\varphi \uparrow 1) \uparrow n, t) \in [\![\alpha]\!]_u \\
\equiv \quad & (\ast \text{ induction hypothesis } \ast) \\
& (\varphi \uparrow 1, t + n) \in [\![\alpha]\!]_{u+n} \\
\equiv \quad & (\ast \text{ induction hypothesis } \ast) \\
& (\varphi, t + n + 1) \in [\![\alpha]\!]_{u+n+1}
\end{aligned}
$$

This shows the validity of the claim for all $n \in \mathbb{N}$. ■

B.1.2. Properties of the MSC Operators

In the following paragraphs we prove the properties we have listed in Section 4.4 as part of the semantics definition. We structure our presentation along the introduction of the MSC operators in Section 4.4.

empty

Proposition 19 **empty** *is the neutral element with respect to sequential composition, interleaving, and the join of MSCs, i.e. for all* $\alpha \in \langle MSC \rangle$ *each of the following equivalences holds:*

$$\begin{array}{rclrcl} \textbf{empty} \,;\, \alpha & \equiv_u & \alpha & \alpha \,;\, \textbf{empty} & \equiv_u & \alpha \\ \textbf{empty} \sim \alpha & \equiv_u & \alpha & \alpha \sim \textbf{empty} & \equiv_u & \alpha \\ \textbf{empty} \otimes \alpha & \equiv_u & \alpha & \alpha \otimes \textbf{empty} & \equiv_u & \alpha \end{array}$$

□

Proof For all $\alpha \in \langle \text{MSC} \rangle$ we observe:

$$\begin{array}{cl} & [\![\textbf{empty} \,;\, \alpha]\!]_u \\ = & (* \text{ definition of } [\![\,;\,]\!]_u *) \\ & \{(\varphi, t) : \langle \exists t' :: (\varphi, t') \in [\![\textbf{empty}]\!]_u \;\wedge\; (\varphi, t) \in [\![\alpha]\!]_{t'} \rangle\} \\ = & (* \text{ definition of } [\![\textbf{empty}]\!]_u, \text{ predicate calculus } *) \\ & \{(\varphi, t) : (\varphi, u) \in [\![\textbf{empty}]\!]_u \;\wedge\; (\varphi, t) \in [\![\alpha]\!]_u\} \\ = & (* \text{ definition of } [\![\textbf{empty}]\!]_u *) \\ & \{(\varphi, t) : (\varphi, t) \in [\![\alpha]\!]_u\} \\ = & (* \text{ definition of } [\![.]\!]_u *) \\ & [\![\alpha]\!]_u \\ = & (* \text{ definition of } [\![.]\!]_u, \text{ definition of } [\![\textbf{empty}]\!]_u *) \\ & \{(\varphi, t) : (\varphi, t) \in [\![\alpha]\!]_u \;\wedge\; (\varphi, t) \in [\![\textbf{empty}]\!]_t\} \\ = & (* \text{ definition of } [\![.\,;\,.]\!]_u *) \\ & [\![\alpha \,;\, \textbf{empty}]\!]_u \end{array}$$

This shows the validity of the equivalences in the first row.

We derive the validity of **empty** $\sim \alpha \equiv_u \alpha$ directly from the definition of the interleaving operator; we only have to select a type correct oracle bs with $bs \in (C \to \{\text{true}\}^*)^\infty$, and to observe the validity of $(\psi, u) \in [\![\textbf{empty}]\!]_u$ for all $(\psi, u) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty$. Because the

interleaving operator is symmetric (as we will see below) we thus obtain the validity of the second row of equivalences.

With respect to the third row of equivalences we observe for all $\alpha \in \langle \text{MSC} \rangle$:

$$
\begin{aligned}
& (\varphi, t) \in [\![\mathbf{empty} \otimes \alpha]\!]_u \\
\equiv \quad & (* \text{ definition of } [\![\otimes]\!]_u,\ msgs.\mathbf{empty} = \emptyset *) \\
& \langle \exists t_1, t_2 \in \mathbb{N}_\infty :: (\varphi, t_1) \in [\![\mathbf{empty}]\!]_u \ \wedge \ (\varphi, t_2) \in [\![\alpha]\!]_u \ \wedge \ t = \max(t_1, t_2) \rangle \\
\equiv \quad & (* \ (\varphi, t) \in [\![\mathbf{empty}]\!]_u \ \equiv \ t = u *) \\
& \langle \exists t_2 \in \mathbb{N}_\infty :: (\varphi, t_2) \in [\![\alpha]\!]_u \ \wedge \ t = \max(u, t_2) \rangle \\
\equiv \quad & (* \ t_2 \geq u *) \\
& (\varphi, t) \in [\![\alpha]\!]_u
\end{aligned}
$$

The remaining equivalence follows by the symmetry of the join operator (see below). ■

any

Proposition 20 any *subsumes all MSCs, i.e. for all $\alpha \in \langle MSC \rangle$ the following set inclusion holds:*

$$[\![\alpha]\!]_u \subseteq [\![\mathbf{any}]\!]_u$$ □

PROOF We observe for all $\alpha \in \langle \text{MSC} \rangle$:

$$
\begin{aligned}
& [\![\alpha]\!]_u \subseteq [\![\mathbf{any}]\!]_u \\
\equiv \quad & (* \text{ definition of set inclusion } *) \\
& \langle \forall (\varphi, t) :: (\varphi, t) \in [\![\alpha]\!]_u \ \Rightarrow \ (\varphi, t) \in [\![\mathbf{any}]\!]_u \rangle \\
\equiv \quad & (* \ (\varphi, t) \in [\![\mathbf{any}]\!]_u \text{ for all } (\varphi, t) \text{ with } t \geq u,\ (\varphi, t) \in [\![\alpha]\!]_u \ \Rightarrow \ t \geq u *) \\
& \text{true}
\end{aligned}
$$

■

Sequential Composition

Proposition 21 (Associativity of ;) *Sequential composition is associative, i.e. for all $\alpha, \beta, \gamma \in \langle MSC \rangle$ we have:*

$$(\alpha \,;\, \beta) \,;\, \gamma \ \equiv_u \ \alpha \,;\, (\beta \,;\, \gamma)$$ □

PROOF For all $\alpha, \beta, \gamma \in \langle \text{MSC} \rangle$ we observe:

$$\begin{aligned}
& [\![(\alpha \,;\, \beta) \,;\, \gamma]\!]_u \\
= \quad & \text{(* definition of } [\![. \,;\, .]\!]_u \text{ *)} \\
& \{(\varphi, t) : \langle \exists t' :: (\varphi, t') \in [\![\alpha \,;\, \beta]\!]_u \;\wedge\; (\varphi, t) \in [\![\gamma]\!]_{t'} \rangle\} \\
= \quad & \text{(* definition of } [\![. \,;\, .]\!]_u \text{, predicate calculus *)} \\
& \{(\varphi, t) : \langle \exists t', t'' :: (\varphi, t'') \in [\![\alpha]\!]_u \;\wedge\; (\varphi, t') \in [\![\beta]\!]_{t''} \;\wedge\; (\varphi, t) \in [\![\gamma]\!]_{t'} \rangle\} \\
= \quad & \text{(* definition of } [\![. \,;\, .]\!]_u \text{ *)} \\
& \{(\varphi, t) : \langle \exists t'' :: (\varphi, t'') \in [\![\alpha]\!]_u \;\wedge\; (\varphi, t'') \in [\![\beta \,;\, \gamma]\!]_{t''} \rangle\} \\
= \quad & \text{(* definition of } [\![. \,;\, .]\!]_u \text{ *)} \\
& [\![\alpha \,;\, (\gamma \,;\, \beta)]\!]_u
\end{aligned}$$

■

Proposition 22 (; distributes over |) *Sequential composition distributes both from the left and from the right over* $|$*, i.e. for all* $\alpha, \beta, \gamma \in \langle MSC \rangle$ *the following two equivalences hold:*

$$\alpha \,;\, (\beta \mid \gamma) \;\equiv_u\; (\alpha \,;\, \beta) \mid (\alpha \,;\, \gamma)$$

$$(\beta \mid \gamma) \,;\, \alpha \;\equiv_u\; (\beta \,;\, \alpha) \mid (\gamma \,;\, \alpha)$$

□

PROOF We show the validity of the first equivalence. We observe for all $\alpha, \beta, \gamma \in \langle \text{MSC} \rangle$:

$$\begin{aligned}
& [\![\alpha \,;\, (\beta \mid \gamma)]\!]_u \\
= \quad & \text{(* definition of } [\![. \,;\, .]\!]_u \text{ *)} \\
& \{(\varphi, t) : \langle \exists t' :: (\varphi, t') \in [\![\alpha]\!]_u \;\wedge\; (\varphi, t) \in [\![\beta \mid \gamma]\!]_{t'} \rangle\} \\
= \quad & \text{(* definition of } [\![. \mid .]\!]_u \text{ *)} \\
& \{(\varphi, t) : \langle \exists t' :: (\varphi, t') \in [\![\alpha]\!]_u \;\wedge\; ((\varphi, t) \in [\![\beta]\!]_{t'} \;\vee\; (\varphi, t) \in [\![\gamma]\!]_{t'}) \rangle\} \\
= \quad & \text{(* predicate calculus, definitions of } [\![. \,;\, .]\!]_u \text{, and } [\![. \mid .]\!]_u \text{ *)} \\
& [\![(\alpha \,;\, \beta) \mid (\alpha \,;\, \gamma)]\!]_u
\end{aligned}$$

We omit the proof for the distributivity from the right, which proceeds along the same lines as the one we have carried out explicitly. ■

Guarded MSCs

Proposition 23 (Conjunction of guards) *For all guards $p_K, p'_{K'} \in \langle GUARD \rangle$, and MSCs $\alpha \in \langle MSC \rangle$ we have:*

$$p_K : (p'_{K'} : \alpha) \equiv_u (p_K \wedge p'_{K'}) : \alpha \qquad \square$$

PROOF We observe for any $p_K, p'_{K'} \in \langle \text{GUARD} \rangle$, and any $\alpha \in \langle \text{MSC} \rangle$:

$$\begin{aligned}
& [\![p_K : (p'_{K'} : \alpha)]\!]_u \\
= \quad & (* \text{ definition of } [\![. : .]\!]_u *) \\
& \{(\varphi, t) \in [\![p'_{K'} : \alpha]\!]_u : \pi_2(\varphi).u \in [\![p_K]\!]\} \\
= \quad & (* \text{ definition of } [\![. : .]\!]_u *) \\
& \{(\varphi, t) \in \{(\varphi', t') \in [\![\alpha]\!]_u : \pi_2(\varphi').u \in [\![p'_{K'}]\!]\} : \pi_2(\varphi).u \in [\![p_K]\!]\} \\
= \quad & (* \text{ set theory } *) \\
& \{(\varphi, t) \in [\![\alpha]\!]_u : \pi_2(\varphi).u \in [\![p'_{K'}]\!] \wedge \pi_2(\varphi).u \in [\![p_K]\!]\} \\
= \quad & (* \text{ definition of } [\![. : .]\!]_u *) \\
& [\![(p_K \wedge p'_{K'}) : \alpha]\!]_u
\end{aligned}$$

■

Proposition 24 (true holds always) *For all $\alpha \in \langle MSC \rangle$ and $u \in \mathbb{N}_\infty$ we have:*

$$[\![\text{true} : \alpha]\!]_u = [\![\alpha]\!]_u \qquad \square$$

PROOF We observe for any $u \in \mathbb{N}_\infty$, and any $\alpha \in \langle \text{MSC} \rangle$:

$$\begin{aligned}
& [\![\text{true} : \alpha]\!]_u \\
= \quad & (* \text{ definition of } [\![. : .]\!]_u *) \\
& \{(\varphi, t) \in [\![\alpha]\!]_u : \pi_2(\varphi).u \in [\![\text{true}]\!]\} \\
= \quad & (* [\![\text{true}]\!] = S *) \\
& [\![\alpha]\!]_u
\end{aligned}$$

■

Proposition 25 (false holds never) false *holds in no execution, i.e. for all $\alpha \in \langle MSC \rangle$ and $u \in \mathbb{N}_\infty$ the following holds:*

$$[\![\text{false} : \alpha]\!]_u = \emptyset \qquad \square$$

PROOF We observe for any $u \in \mathbb{N}_\infty$, and any $\alpha \in \langle \text{MSC} \rangle$:

$$\begin{aligned}
& [\![\text{false} : \alpha]\!]_u \\
= \quad & (* \text{ definition of } [\![. : .]\!]_u *) \\
& \{(\varphi, t) \in [\![\alpha]\!]_u : \pi_2(\varphi).u \in [\![\text{false}]\!]\} \\
= \quad & (* [\![\text{false}]\!] = \emptyset *) \\
& \emptyset
\end{aligned}$$

■

Alternatives

Proposition 26 (| is symmetric and associative) *The alternative operator | is symmetric and associative, i.e. for all $\alpha, \beta, \gamma \in \langle MSC \rangle$ the following two equivalences hold:*

$$\alpha \,|\, \beta \;\equiv_u\; \beta \,|\, \alpha$$

$$\alpha \,|\, (\beta \,|\, \gamma) \;\equiv_u\; (\alpha \,|\, \beta) \,|\, \gamma \qquad \square$$

Proof Both equivalences follow directly by the definition of $[\![. \,|\, .]\!]_u$, and the symmetry and associativity of set union. ∎

Interleaving

Proposition 27 (Symmetry of $\sim$) *The interleaving operator $\sim$ is symmetric, i.e. for all $\alpha, \beta \in \langle MSC \rangle$ the following equivalence holds:*

$$\alpha \sim \beta \;\equiv_u\; \beta \sim \alpha \qquad \square$$

Proof The validity of this equivalence is based on the symmetry (with respect to α and β) of the right-hand-side of the interleaving operator's definition; by inverting one of the oracles that exist for $\alpha \sim \beta$ in a pointwise manner we obtain an appropriate oracle for $\beta \sim \alpha$. ∎

Proposition 28 (Associativity of $\sim$) *The interleaving operator $\sim$ is associative; for all $\alpha, \beta, \gamma \in \langle MSC \rangle$ the following equivalence holds:*

$$\alpha \sim (\beta \sim \gamma) \;\equiv_u\; (\alpha \sim \beta) \sim \gamma \qquad \square$$

Proof Because the full proof is technically rather cumbersome (it involves a significant amount of time point transformations), we give a rough sketch of the proof outline only.

We prove the validity of $[\![\alpha \sim (\beta \sim \gamma)]\!]_u = [\![(\alpha \sim \beta) \sim \gamma]\!]_u$ by mutual set inclusion. The strategy we follow is driven by the use of an oracle in the interleaving operator's semantics definition; in the argument below, we relate the two oracles corresponding to the left-hand-side with those corresponding to the right-hand-side of the equation we aim at.

Let us assume that $(\varphi, t) \in [\![\alpha \sim (\beta \sim \gamma)]\!]_u$ holds. We must show that this implies $(\varphi, t) \in [\![(\alpha \sim \beta) \sim \gamma]\!]_u$. To see this, we observe that $(\varphi, t) \in [\![\alpha \sim (\beta \sim \gamma)]\!]_u$ implies the existence of two oracles bs_1 and bs_2, such that bs_1 controls the interleaving of α and $\beta \sim \gamma$, and bs_2 controls the interleaving of β and γ. Whenever $bs_1.ch.t.n$ is true, then α contributes the element $\pi_1(\varphi).ch.t.n$ to the interleaving. Otherwise, i.e. if $bs_1.ch.t.n$ is false, the contribution comes from $\beta \sim \gamma$. In this latter case, we know that there is a time

t' and a position n' such that if $bs_2.ch.t'.n'$ is true, then the element $\pi_1(\varphi).ch.t.n$ stems from β; otherwise it stems from γ.

Furthermore, we observe that for $(\varphi, t) \in [\![(\alpha \sim \beta) \sim \gamma]\!]_u$ to hold there must exist two oracles bs_3 and bs_4, such that bs_3 controls the interleaving of $\alpha \sim \beta$ and γ, and bs_4 controls the interleaving of α and β. The question now is whether we can construct the oracles bs_3 and bs_4, given the oracles bs_1 and bs_2. This is, indeed, possible. We must set $bs_3.ch.t.n$ to true if and only if $(bs_1.ch.t.n \vee bs_2.ch.t'.n')$ holds (where t, t', n, and n' have the same meaning as above). Otherwise we know that the element $\pi_1(\varphi).ch.t.n$ is contributed by γ, and $bs_3.ch.t.n$ must be false. We must set $bs_4.ch.t'.n'$ to true if and only if $bs_1.ch.t.n$ is true; this corresponds to the case where α contributes the element $\pi_1(\varphi).ch.t.n$.

By this construction we obtain the validity of $(\varphi, t) \in [\![\alpha \sim (\beta \sim \gamma)]\!]_u \Rightarrow (\varphi, t) \in [\![(\alpha \sim \beta) \sim \gamma]\!]_u$. The proof of the other direction, i.e. $(\varphi, t) \in (\varphi, t) \in [\![(\alpha \sim \beta) \sim \gamma]\!]_u \Rightarrow [\![\alpha \sim (\beta \sim \gamma)]\!]_u$, proceeds along the same lines. ■

Join

Proposition 29 (Symmetry of $\otimes$) *The join operator $\otimes$ is symmetric, i.e. for all $\alpha, \beta \in \langle MSC \rangle$ the following equivalence holds:*

$$\alpha \otimes \beta \equiv_u \beta \otimes \alpha \qquad \square$$

Proof This equivalence follows directly by the symmetry of the right-hand-side of the join operator's definition in α and β, and the symmetry of conjunction and the maximum function. ■

Proposition 30 (Associativity of $\otimes$) *The join operator $\otimes$ is associative, i.e. for all $\alpha, \beta, \gamma \in \langle MSC \rangle$ the following equivalence holds:*

$$\alpha \otimes (\beta \otimes \gamma) \equiv_u (\alpha \otimes \beta) \otimes \gamma \qquad \square$$

Proof Again, the proof is technically rather cumbersome, so we just sketch the basic idea for reasons of brevity.

Observe for arbitrary $\delta_1, \delta_2 \in \langle \mathrm{MSC} \rangle$ that, in essence, $[\![\delta_1 \otimes \delta_2]\!]_u$ is a certain subset of $[\![\delta_1]\!]_u \cap [\![\delta_2]\!]_u$; this is due to the first conjunct of the join operator's definition. Thus $[\![\alpha \otimes (\beta \otimes \gamma)]\!]_u$ and $[\![(\alpha \otimes \beta) \otimes \gamma]\!]_u$ are both subsets of $[\![\alpha]\!]_u \cap [\![\beta]\!]_u \cap [\![\gamma]\!]_u$. To see that $[\![\alpha \otimes (\beta \otimes \gamma)]\!]_u$ and $[\![(\alpha \otimes \beta) \otimes \gamma]\!]_u$ are indeed identical subsets of $[\![\alpha]\!]_u \cap [\![\beta]\!]_u \cap [\![\gamma]\!]_u$ we concentrate on what elements do *not* occur in these two subsets. If $(\varphi, t) \in [\![\alpha]\!]_u \cap [\![\beta]\!]_u \cap [\![\gamma]\!]_u$ holds, but we have also $(\varphi, t) \notin [\![\alpha \otimes (\beta \otimes \gamma)]\!]_u$, then $\pi_1(\varphi)$ contains a redundant message from $msgs.\alpha \cap msgs.(\beta \otimes \gamma)$, i.e. a message $ch \triangleright m$ appearing in α and either in β or in γ ($msgs.(\beta \otimes \gamma) = msgs.\beta \cup msgs.\gamma$). If $ch \triangleright m \in msgs.\beta$ holds, then $(\varphi, t) \notin [\![\alpha \otimes \beta]\!]_u$ is the

consequence, because $\pi_1(\varphi)$ is redundant with respect to the message $ch \triangleright m$, which occurs in both α and β. Thus, $(\varphi, t) \notin [\![(\alpha \otimes \beta) \otimes \gamma]\!]_u$ holds in this case. Assume, instead, that both $ch \triangleright m \notin msgs.\beta$ and $ch \triangleright m \in msgs.\gamma$ hold. Then, again, $\pi_1(\varphi)$ is redundant with respect to message $ch \triangleright m$, and the join of $\alpha \otimes \beta$ and γ. Thus, in this case, we also have $(\varphi, t) \notin [\![(\alpha \otimes \beta) \otimes \gamma]\!]_u$. By taking the contrapositive, we obtain the validity of

$$[\![(\alpha \otimes \beta) \otimes \gamma]\!]_u \subseteq [\![\alpha \otimes (\beta \otimes \gamma)]\!]_u$$

The other direction follows by a similar line of thought. ∎

Loops

Proposition 31 *For all $\alpha \in \langle MSC \rangle$ the following equivalence holds:*

$$\alpha\uparrow_{<\text{false}>} \;\equiv_u\; \textbf{empty} \qquad \square$$

Proof For all $\alpha \in \langle \text{MSC} \rangle$ we observe:

$$
\begin{aligned}
& [\![\alpha\uparrow_{<\text{false}>}]\!]_u \\
= \quad & (* \text{ defining functional (Proposition 34) } *) \\
& \langle \nu X :: \tau_{gl}.X \rangle \\
= \quad & (* \text{ definition of } \tau_{gl} \text{ with } p = \text{false}, [\![\text{false} : \alpha]\!]_u = \emptyset *) \\
& \langle \nu X :: [\![\text{true} : \textbf{empty}]\!]_u \rangle \\
= \quad & (* \text{ property of } \langle \nu :: \rangle, \text{ true} : \beta \;\equiv_u\; \beta \text{ for any } \beta *) \\
& [\![\textbf{empty}]\!]_u
\end{aligned}
$$

∎

Proposition 32 (Loop unfolding) *For any $\alpha \in \langle MSC \rangle$, and $m, n \in \mathbb{N}_\infty$ the following two equivalences hold:*

$$\alpha\,;\,\alpha\uparrow_{<m,n>} \;\equiv_u\; \alpha\uparrow_{<m+1,n+1>}$$

$$\alpha\uparrow_{<m,n>}\,;\,\alpha \;\equiv_u\; \alpha\uparrow_{<m+1,n+1>} \qquad \square$$

Proof Before we show the validity of the first equivalence for $m, n \in \mathbb{N}$, $n \geq m$, we observe that $\alpha\uparrow_{<m,n>} \;\equiv_u\; \alpha^m\,;\,\alpha\uparrow_{<0,n-m>}$ holds directly by definition of $\alpha\uparrow_{<m,n>}$. We derive

$$
\begin{aligned}
& \alpha\,;\,\alpha\uparrow_{<m,n>} \\
\equiv_u \; & (* \text{ definition of } \alpha\uparrow_{<m,n>} *) \\
& \alpha\,;\,(\alpha^m\,;\,\alpha\uparrow_{<0,n-m>}) \\
\equiv_u \; & (* \text{ associativity of } ; \text{ and definition of } \alpha^{m+1} *)
\end{aligned}
$$

$\alpha^{m+1} \; ; \; \alpha\!\uparrow_{\lhd 0,n-m \rhd}$
$\equiv_u$ (* definition of $\alpha\!\uparrow_{\lhd m+1,n+1 \rhd}$ *)
$\alpha\!\uparrow_{\lhd m+1,n+1 \rhd}$

For the second equivalence we observe:

$\alpha\!\uparrow_{\lhd m,n \rhd} \; ; \; \alpha$
$\equiv_u$ (* definition of $\alpha\!\uparrow_{\lhd m,n \rhd}$ *)
$(\alpha^m \; ; \; \alpha\!\uparrow_{\lhd 0,n-m \rhd}) \; ; \; \alpha$
$\equiv_u$ (* associativity of ; and observation, below *)
$\alpha^m \; ; \; \alpha\!\uparrow_{\lhd 1,n-m+1 \rhd}$
$\equiv_u$ (* definition of $\alpha\!\uparrow_{\lhd 1,n-m+1 \rhd}$, associativity of ; , definition of α^{m+1} *)
$\alpha^{m+1} \; ; \; \alpha\!\uparrow_{\lhd 0,n-m \rhd}$
$\equiv_u$ (* definition of $\alpha\!\uparrow_{\lhd m+1,n+1 \rhd}$ *)
$\alpha\!\uparrow_{\lhd m+1,n+1 \rhd}$

The second step of this derivation is valid due to the following observation:

$(\varphi, t) \in [\![\alpha\!\uparrow_{\lhd 0,n-m \rhd} \; ; \; \alpha]\!]_u$
$\equiv$ (* definition of $[\![. \; ; \; .]\!]_u$ *)
$\langle \exists t' \in \mathbb{N}_\infty :: (\varphi, t') \in [\![\alpha\!\uparrow_{\lhd 0,n-m \rhd}]\!]_u \wedge (\varphi, t) \in [\![\alpha]\!]_{t'} \rangle$
$\equiv$ (* definition of $[\![\alpha\!\uparrow_{\lhd 0,n-m \rhd}]\!]_u$ *)
$\langle \exists t' \in \mathbb{N}_\infty :: (\varphi, t') \in \bigcup_{0 \le i \le n-m} [\![\alpha^i]\!]_u \wedge (\varphi, t) \in [\![\alpha]\!]_{t'} \rangle$
$\equiv$ (* set theory, predicate calculus *)
$\langle \exists t' \in \mathbb{N}_\infty, i \in \mathbb{N} : 0 \le i \le n-m : (\varphi, t') \in [\![\alpha^i]\!]_u \wedge (\varphi, t) \in [\![\alpha]\!]_{t'} \rangle$
$\equiv$ (* definition of $[\![. \; ; \; .]\!]_u$ *)
$\langle \exists i \in \mathbb{N} : 0 \le i \le n-m : (\varphi, t) \in [\![\alpha^i \; ; \; \alpha]\!]_u \rangle$
$\equiv$ (* definition of α^{i+1} *)
$\langle \exists i \in \mathbb{N} : 0 \le i \le n-m : (\varphi, t) \in [\![\alpha^{i+1}]\!]_u \rangle$
$\equiv$ (* set theory, predicate calculus *)
$(\varphi, t) \in \bigcup_{0 \le i \le n-m} [\![\alpha^{i+1}]\!]_u$
$\equiv$ (* index shift *)
$(\varphi, t) \in \bigcup_{1 \le i \le n-m+1} [\![\alpha^i]\!]_u$
$\equiv$ (* definition of $[\![\alpha\!\uparrow_{\lhd 1,n-m+1 \rhd}]\!]_u$ *)
$(\varphi, t) \in [\![\alpha\!\uparrow_{\lhd 1,n-m+1 \rhd}]\!]_u$

■

Trigger Composition

Proposition 33 ("Transitivity" of trigger composition)
Trigger composition is "transitive", i.e. for all $u, t \in \mathbb{N}_\infty$, $\varphi \in (\tilde{C} \times S)^\infty$, *and* $\alpha, \beta, \gamma \in \langle MSC \rangle$ *we have:*

$$\langle \exists t_1, t_2 : t_2 \geq t_1 : (\varphi, t_1) \in [\![\alpha \mapsto \beta]\!]_u \wedge (\varphi, t) \in [\![\beta \mapsto \gamma]\!]_{t_2} \rangle \Rightarrow (\varphi, t) \in [\![\alpha \mapsto \gamma]\!]_u \qquad \square$$

Proof For all $u, t \in \mathbb{N}_\infty$, $\varphi \in (\tilde{C} \times S)^\infty$, and $\alpha, \beta, \gamma \in \langle \text{MSC} \rangle$ we observe:

$$\begin{aligned}
& \langle \exists t_1, t_2 : t_1 \geq t_2 : (\varphi, t_1) \in [\![\alpha \mapsto \beta]\!]_u \wedge (\varphi, t) \in [\![\beta \mapsto \gamma]\!]_{t_2} \rangle \\
\equiv \quad & (* \text{ definition of } [\![. \mapsto .]\!]_u \text{, twice } *) \\
& \langle \exists \quad t_1, t_2 : t_1 \geq t_2 : \\
& \qquad \langle \forall t', t'' : \infty > t'' \geq t' \geq u : \\
& \qquad\quad (\varphi, t'') \in [\![\alpha]\!]_{t'} \Rightarrow \langle \exists t''' : \infty > t''' > t'' : (\varphi, t) \in [\![\beta]\!]_{t'''} \rangle \rangle \\
& \qquad \wedge \langle \forall \hat{t}', \hat{t}'' : \infty > \hat{t}'' \geq \hat{t}' \geq t_2 : \\
& \qquad\quad (\varphi, \hat{t}'') \in [\![\beta]\!]_{\hat{t}'} \Rightarrow \langle \exists \hat{t}''' : \infty > \hat{t}''' > \hat{t}'' : (\varphi, t) \in [\![\gamma]\!]_{\hat{t}'''} \rangle \rangle \rangle \\
\equiv \quad & (* \text{ predicate calculus } *) \\
& \langle \exists t_1, t_2 : t_1 \geq t_2 : \langle \forall \quad t', t'', \hat{t}', \hat{t}'' : \infty > t'' \geq t' \geq u \wedge \infty > \hat{t}'' \geq \hat{t}' \geq t_2 : \\
& \qquad (\varphi, t'') \in [\![\alpha]\!]_{t'} \Rightarrow \langle \exists t''' : \infty > t''' > t'' : (\varphi, t) \in [\![\beta]\!]_{t'''} \rangle \\
& \qquad \wedge \quad (\varphi, \hat{t}'') \in [\![\beta]\!]_{\hat{t}'} \Rightarrow \langle \exists \hat{t}''' : \infty > \hat{t}''' > \hat{t}'' : (\varphi, t) \in [\![\gamma]\!]_{\hat{t}'''} \rangle \rangle \rangle \\
\Rightarrow \quad & (* \text{ predicate calculus: set } \hat{t}'' \text{ to } t_1 \text{; transitivity of } \Rightarrow; \\
& \qquad t_1 \geq t''' > t'' \text{ (see Proposition 2) } *) \\
& \langle \exists t_1, t_2 : t_1 \geq t_2 : \langle \forall t', t'' : \infty > t'' \geq t' \geq u \wedge \infty > t_1 \geq t_2 : \\
& \qquad (\varphi, t'') \in [\![\alpha]\!]_{t'} \Rightarrow \langle \exists \hat{t}''' : \infty > \hat{t}''' > t_1 > t'' : (\varphi, t) \in [\![\gamma]\!]_{\hat{t}'''} \rangle \rangle \rangle \\
\Rightarrow \quad & (* \text{ definition of } [\![. \mapsto .]\!]_u *) \\
& (\varphi, t) \in [\![\alpha \mapsto \gamma]\!]_u
\end{aligned}$$

■

B.1.3. Well-Definedness of the Semantics

Proposition 34 *The semantics of Section 4.4 is well defined. In particular, each of the recursive Equations (4.3) and (4.4) has a unique greatest fixpoint.* □

Proof To see that both (4.3), and (4.4) have unique greatest fixpoints (with respect to set inclusion), we rewrite the equations to make the underlying set transformers explicit.

For every $\alpha \in \langle \text{MSC} \rangle$, $p \in \langle \text{GUARD} \rangle$, and $u \in \mathbb{N}_\infty$ we define the set transformers

$$\tau_{gl}, \tau_{pl} : \mathcal{P}((\tilde{C} \times S)^\infty \times \mathbb{N}_\infty) \rightarrow \mathcal{P}((\tilde{C} \times S)^\infty \times \mathbb{N}_\infty)$$

by

$$\begin{aligned}\tau_{gl}.X = \quad & [\![(\neg p) : \mathbf{empty}]\!]_u \\ & \cup \{(\varphi, t) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty : \\ & \quad \langle \exists t', t'' \in \mathbb{N}_\infty : u \leq t' \leq t'' : \\ & \quad\quad (\varphi, t') \in [\![p : \alpha]\!]_u \ \wedge \ (\varphi \uparrow t', t'') \in X \ \wedge \ t = t' + t'' \rangle\}\end{aligned}$$

and

$$\begin{aligned}\tau_{pl}.X = \quad & \{(\varphi, t) \in [\![\alpha]\!]_u : \langle \forall v \in [u, t] :: m \notin \pi_1(\varphi).v.ch \rangle\} \\ & \cup \{(\varphi, t) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty : \\ & \quad \langle \exists v \in \mathbb{N}, t'' \in \mathbb{N}_\infty :: \\ & \quad\quad\quad v = \min\{t' : u < t' \leq t \ \wedge \ m \in \pi_1(\varphi).t'.ch\} \\ & \quad\quad \wedge (\varphi, v-1) \in [\![\alpha]\!]_u^{v-1} \\ & \quad\quad \wedge (\varphi \uparrow v, t'') \in X \\ & \quad\quad \wedge t = v + t'' \rangle\}\end{aligned}$$

Here, τ_{gl} and τ_{pl} are the set transformers for the semantics definition of guarded loops and preemptive loops, respectively. We have obtained τ_{gl} and τ_{pl} essentially by expanding the Equations (4.3), and (4.4). Thus, to see whether (4.3), and (4.4) have unique greatest fixpoints, we determine whether this holds for the equations $\tau_{gl}.X = X$ and $\tau_{pl}.X = X$.

In the definitions of τ_{gl}, and τ_{pl} the variable X occurs only positively on the respective right-hand-sides. This syntactic criterion suffices to deduce that both τ_{gl} and τ_{pl} are monotonic with respect to set inclusion[1]. By the theorem of Knaster-Tarski (cf. [Win93]), we can conclude that each of these functionals has a (unique) greatest fixpoint. ■

In the following sections we have to discharge several proof obligations with respect to the semantics of guarded and preemptive loops. One of the proof principles we employ for this purpose is as follows:

Proposition 35 (Proof principle for fixpoints) *Let $\tau : \mathcal{P}(\top) \rightarrow \mathcal{P}(\top)$ be a set transformer for a given set $\top$, and let $P : \mathcal{P}(\top) \rightarrow \mathbb{B}$ be a predicate, such that all of the following hold:*

(1) τ is monotonic

[1] Note that the monotonicity of the predicate transformers τ_{gl} and τ_{pl} is independent of the (lack of) monotonicity of the individual MSC operators with respect to their arguments (see also Section 5.3.2 and the proofs in Appendix B.2).

(2) P is admissible, i.e.

$$\langle \forall n \in \mathbb{N} :: P.X_n \rangle \Rightarrow P.(\bigcap_{n \in \mathbb{N}} X_n)$$

holds for all infinite descending chains $(X_i)_{i \in \mathbb{N}}$ *with* $X_{i+1} \subseteq X_i$ *for all* $i \in \mathbb{N}$

(3) $P.\top$

(4) $\langle \forall X \subseteq \top :: P.X \Rightarrow P.(\tau.X) \rangle$

Then, we can conclude the validity of

$$P.\langle \nu X :: \tau.X \rangle$$ □

Proof See, for instance, [CC92] for the dual claim for the least fixpoint $\langle \mu X :: \rangle$. Because of the duality of $\langle \nu X :: \rangle$ and $\langle \mu X :: \rangle$ we obtain our proof rule as a simple rewriting of the one in [CC92]. ■

B.1.4. Sequential Composition versus Interleaving

Proposition 36 *For all* $\alpha, \beta \in \langle MSC \rangle$ *the following equivalence holds:*

$$(\alpha \sim \beta \equiv_u \alpha \,;\, \beta) \quad \equiv \quad (\alpha \equiv_u \textbf{empty}) \vee (\beta \equiv_u \textbf{empty})$$ □

Proof We show

$$\alpha \sim \beta \equiv_u \alpha \,;\, \beta \Rightarrow (\alpha \equiv_u \textbf{empty}) \vee (\beta \equiv_u \textbf{empty})$$

The other direction follows trivially from the properties of **empty**, interleaving, and sequential composition. Assume that $\alpha \not\equiv_u$ **empty** $\wedge$ $\beta \not\equiv_u$ **empty** holds. For simplicity, we consider the special case $\alpha = ch \triangleright m$, and $\beta = ch \triangleright n$, for $m \neq n$, without loss of generality. Then, for any $u \in \mathbb{N}$, we can exhibit a behavior φ, and a time point $t = u + 1$ with $\pi_1(\varphi).t.ch = \langle m, n \rangle$. Clearly, $(\varphi, t) \in [\![\alpha \sim \beta]\!]_u$ holds, as any type-correct oracle $bs \in (C \to \mathbb{B}^*)^\infty$ with $bs.t.ch = \langle \text{true}, \text{false} \rangle$ shows. However, we can also show that $(\varphi, t) \notin [\![\alpha \,;\, \beta]\!]_u$ holds. Observe that

$$\begin{aligned} & (\varphi, t) \in [\![ch \triangleright m \,;\, ch \triangleright n]\!]_u \\ \equiv \quad & (* \text{ definition of } [\![. \,;\, .]\!]_u \, *) \\ & \langle \exists t' :: (\varphi, t') \in [\![ch \triangleright m]\!]_u \wedge (\varphi, t) \in [\![ch \triangleright n]\!]_{t'} \rangle \\ \Rightarrow \quad & (* \; t > t' > u, \text{ from the semantics of message occurrence } *) \\ & t \geq u + 2 \end{aligned}$$

holds. Thus, we have shown the validity of

$$(\alpha \not\equiv_u \textbf{empty}) \wedge (\beta \not\equiv_u \textbf{empty}) \Rightarrow (\alpha \sim \beta \not\equiv_u \alpha \,;\, \beta)$$

By taking the contrapositive, the original claim follows. ■

B.2. Property Refinement Rules

The notion of property refinement (" $\leq_p$ ") we have introduced in Section 5.3 is based on set inclusion; we consider an MSC α a property refinement of an MSC β if $[\![\alpha]\!]_u \subseteq [\![\beta]\!]_u$ holds. In the following paragraphs we recapitulate and prove the property refinement rules of Section 5.3.1.

any

Proposition 37 (Every MSC refines any) *For all MSCs $\alpha \in \langle MSC \rangle$ we have*

$$\alpha \ \leq_p \ \textbf{any}$$

□

Proof This follows directly from Proposition 20. ■

Guarded MSCs

Proposition 38 *For all MSCs $\alpha \in \langle MSC \rangle$ and guards $p, q \in \langle GUARD \rangle$ we have*

$$(p \Rightarrow q) \ \ \Rightarrow \ \ p : \alpha \ \leq_p \ q : \alpha$$

□

Proof We observe for all $\alpha \in \langle \text{MSC} \rangle$ and $p, q \in \langle \text{GUARD} \rangle$:

$p : \alpha \ \leq_p \ q : \alpha$

$\equiv$ (* definition of $\leq_p$ *)

$[\![p : \alpha]\!]_u \subseteq [\![q : \alpha]\!]_u$

$\equiv$ (* set theory *)

$\langle \forall (\varphi, t) \in (\tilde{C} \times S)^{\infty} \times \mathbb{N}_{\infty} :: (\varphi, t) \in [\![p : \alpha]\!]_u \ \Rightarrow \ (\varphi, t) \in [\![q : \alpha]\!]_u \rangle$

$\equiv$ (* definition of $[\![p : \alpha]\!]_u$ and $[\![q : \alpha]\!]_u$ *)

$\langle \forall (\varphi, t) \in [\![\alpha]\!]_u :: \pi_2.(\varphi).u \in [\![p]\!] \ \Rightarrow \ \pi_2.(\varphi).u \in [\![q]\!] \rangle$

$\Leftarrow$ (* $p \Rightarrow q \ \equiv \ [\![p]\!] \subseteq [\![q]\!]$ *)

$p \Rightarrow q$

■

Alternatives

Proposition 39 *For all MSCs $\alpha, \beta \in \langle MSC \rangle$ we have*

$$\begin{aligned} \alpha &\leq_p \alpha \mid \beta \\ \beta &\leq_p \alpha \mid \beta \end{aligned}$$
□

Proof For all MSCs $\alpha, \beta \in \langle \text{MSC} \rangle$ we observe:

$$\begin{aligned} & [\![\alpha]\!]_u \\ \subseteq \quad & (* \text{ set theory } *) \\ & [\![\alpha]\!]_u \cup [\![\beta]\!]_u \\ \supseteq \quad & (* \text{ set theory } *) \\ & [\![\beta]\!]_u \end{aligned}$$

Because of $[\![\alpha \mid \beta]\!]_u = [\![\alpha]\!]_u \cup [\![\beta]\!]_u$ the claim follows trivially by definition of $\leq_p$ and the above observation. ■

Sequential Composition Refines Interleaving

Proposition 40 *For all MSCs $\alpha, \beta \in \langle MSC \rangle$ the following two refinement relations hold:*

$$\begin{aligned} \alpha ; \beta &\leq_p \alpha \sim \beta \\ \beta ; \alpha &\leq_p \alpha \sim \beta \end{aligned}$$
□

Proof To prove the implication

$$(\varphi, t) \in [\![\alpha ; \beta]\!]_u \Rightarrow (\varphi, t) \in [\![\alpha \sim \beta]\!]_u$$

we observe that $(\varphi, t) \in [\![\alpha ; \beta]\!]_u$ implies the existence of a time $t' \in \mathbb{N}_\infty$ such that both $(\varphi, t') \in [\![\alpha]\!]_u$ and $(\varphi, t) \in [\![\beta]\!]_{t'}$ hold. By constructing a type-correct oracle $bs \in (C \to \mathbb{B}^*)^\infty$ with $(bs.ch)|_{[0,t'-u]} \in (\text{true}^*)^\omega$ and $(bs.ch)|_{[t'-u+1,\infty]} \in (\text{false}^*)^\omega$ for all channels $ch \in C$ we can easily show the validity of $(\varphi, t) \in [\![\alpha \sim \beta]\!]_u$. The proof of $\beta ; \alpha \leq_p \alpha \sim \beta$ proceeds along the same lines of thought. ■

Narrowing Loop Bounds

Proposition 41 *For all MSCs $\alpha \in \langle MSC \rangle$ and $m, m', n, n' \in \mathbb{N}_\infty$ the following refinement relation holds:*

$$[m', n'] \subseteq [m, n] \Rightarrow \alpha \uparrow_{<m',n'>} \leq_p \alpha \uparrow_{<m,n>}$$
□

PROOF Let $m, m', n, n' \in \mathbb{N}$. Then $[m', n'] \subseteq [m, n]$ implies $m \leq m' \leq n' \leq n$. Thus, $\alpha{\uparrow}_{<m',n'>}$ requires at least as many and also at most as many repetitions as $\alpha{\uparrow}_{<m,n>}$ does. Any element $(\varphi, t) \in [\![\alpha{\uparrow}_{<m',n'>}]\!]_u$ exhibits at least m and at most n repetitions of α. Thus, (φ, t) is also an element of $[\![\alpha{\uparrow}_{<m,n>}]\!]_u$. The line of reasoning for the cases where any one of m, m', n, and n' equals ∞ is similar. ■

Proposition 42 *For all MSCs $\alpha \in \langle MSC \rangle$ and $m \in \mathbb{N}$ the following refinement relation holds:*

$$\alpha{\uparrow}_{<0,m>} \;\leq_p\; \alpha{\uparrow}_{<*>}$$

□

PROOF We observe for all MSCs $\alpha \in \langle \text{MSC} \rangle$ and $m \in \mathbb{N}$:

$$\begin{array}{ll} & [\![\alpha{\uparrow}_{<0,m>}]\!]_u \\ = & (*\ \text{definition of } [\![\alpha{\uparrow}_{<0,m>}]\!]_u\ *) \\ & \bigcup_{0 \leq i \leq m} [\![\alpha^i]\!]_u \\ \subseteq & (*\ \text{set theory}\ *) \\ & \bigcup_{m \in \mathbb{N}} (\bigcup_{0 \leq i \leq m} [\![\alpha^i]\!]_u) \\ = & (*\ \text{definition of } [\![\alpha{\uparrow}_{<0,m>}]\!]_u\ *) \\ & \bigcup_{m \in \mathbb{N}} [\![\alpha{\uparrow}_{<0,m>}]\!]_u \\ = & (*\ \text{definition of } [\![\alpha{\uparrow}_{<*>}]\!]_u\ *) \\ & [\![\alpha{\uparrow}_{<*>}]\!]_u \end{array}$$

■

Removing Preemption

Proposition 43 *Let $\alpha \in \langle MSC \rangle$ and $ch \triangleright m \in \langle MSG \rangle$ be an MSC and a message specification, respectively, and let $\overline{M}$ be defined as follows:*

$$\overline{M} \overset{\mathsf{def}}{=} \{ch \triangleright m : ch \in C \wedge m \in M\}$$

Then the following two refinement relations hold:

$$\lceil \alpha \rceil^{\overline{M} \setminus \{ch \triangleright m\}} \;\leq_p\; \alpha \overset{ch \triangleright m}{\to} \beta$$

$$\lceil \alpha \rceil^{\overline{M} \setminus \{ch \triangleright m\}} \;\leq_p\; \alpha{\Uparrow}_{ch \triangleright m}$$

□

PROOF With respect to the first relation we observe for any $\alpha \in \langle \text{MSC} \rangle$ and $ch \triangleright m \in \langle \text{MSG} \rangle$:

$$
\begin{array}{ll}
 & (\varphi, t) \in [\![\lceil\alpha\rceil^{\overline{M}\setminus\{ch\triangleright m\}}]\!]_u \\
\equiv & \texttt{(*}\ \text{definition of } [\![\lceil\alpha\rceil^{\overline{M}\setminus\{ch\triangleright m\}}]\!]_u\ \texttt{*)} \\
 & (\varphi, t) \in [\![\alpha]\!]_u \\
 & \wedge\langle\forall t' \in \mathbb{N}, ch'\triangleright m' \in \langle\text{MSG}\rangle : u \leq t' \leq t : m' \in \pi_1(\varphi).t'.ch' \Rightarrow ch'\triangleright m' \neq ch\triangleright m\rangle \\
\equiv & \texttt{(*}\ \text{simplification of the second conjunct}\ \texttt{*)} \\
 & (\varphi, t) \in [\![\alpha]\!]_u \ \wedge\ \langle\forall v \in [u, t] :: m \notin \pi_1(\varphi).v.ch\rangle \\
\Rightarrow & \texttt{(*}\ \text{definition of } [\![\alpha \stackrel{ch\triangleright m}{\to} \beta]\!]_u\ \texttt{*)} \\
 & (\varphi, t) \in \alpha \stackrel{ch\triangleright m}{\to} \beta
\end{array}
$$

The second relation follows by substituting α for β in the last step of this proof, and by definition of $[\![\alpha{\Uparrow}_{ch\triangleright m}]\!]_u$. ■

LHS Weakening and RHS Strengthening of Trigger Composition

Proposition 44 *Weakening the left-hand-side and strengthening the right-hand-side of a trigger composition both result in a property refinement, i.e. we have for all $\alpha, \alpha', \beta, \beta' \in \langle MSC\rangle$:*

$$((\alpha' \ \leq_p\ \alpha) \wedge (\beta' \ \leq_p\ \beta)) \ \Rightarrow\ ((\alpha \mapsto \beta') \ \leq_p\ (\alpha' \mapsto \beta)) \qquad \square$$

Proof We observe for all $\alpha, \alpha', \beta, \beta' \in \langle\text{MSC}\rangle$ with $(\alpha' \ \leq_p\ \alpha) \wedge (\beta' \ \leq_p\ \beta)$:

$$
\begin{array}{ll}
 & \langle\forall t', t'' \in \mathbb{N} : u \leq t' \leq t'' < \infty : \\
 & \quad (\varphi, t'') \in [\![\alpha']\!]_{t'} \Rightarrow \langle\exists t''' \in \mathbb{N} : t'' < t''' < \infty : (\varphi, t) \in [\![\beta]\!]_{t'''}\rangle\rangle \\
\Leftarrow & \texttt{(*}\ \Rightarrow \text{ is antimonotonic in its first, and monotonic in its second argument}\ \texttt{*)} \\
 & \langle\forall t', t'' \in \mathbb{N} : u \leq t' \leq t'' < \infty : \\
 & \quad (\varphi, t'') \in [\![\alpha]\!]_{t'} \Rightarrow \langle\exists t''' \in \mathbb{N} : t'' < t''' < \infty : (\varphi, t) \in [\![\beta']\!]_{t'''}\rangle\rangle
\end{array}
$$

■

B.3. MSCs for Property-Oriented System Specifications

In Chapter 6 we have investigated several MSC interpretations. One of these, the exact MSC interpretation, forbids the occurrence of messages that do not appear syntactically in the MSC under consideration. Based on this result we obtain a syntactic criterion for preemption refinement; Section B.3.1 contains the corresponding proofs.

Besides MSC interpretations we have also discussed the notions of safety and liveness with respect to MSC specifications in Chapter 6. To this end, we have given several definitions for safety and liveness. We have also studied how safety and liveness properties propagate through MSC specifications. In Section B.3.2 we show the equivalence of the different characterizations for safety and liveness, respectively; moreover, we give the proofs corresponding to our observations regarding property propagation.

B.3.1. Exact MSC Interpretation

Proposition 45 (Minimality of the CW-Semantics) *Only messages specified syntactically in α, i.e. messages inside the set msgs.α can occur in elements of $[\![\alpha]\!]_u$ for any MSC $\alpha \in \langle MSC \rangle$; more precisely, the following implication holds for all messages $ch \triangleright m \in \langle MSG \rangle$:*

$$ch \triangleright m \notin msgs.\alpha \;\Rightarrow\; \langle \forall (\varphi, t) \in [\![\alpha]\!]_{u,CW}, t' \in [u,t] :: m \notin \pi_1(\varphi).t'.ch \rangle \qquad \square$$

Proof The basic idea behind this proof is to observe that the only MSC operator that explicitly requires the presence of a message in $\pi_1(\varphi).t.ch$ for $\varphi \in (\tilde{C} \times S)^\infty$, $t \in \mathbb{N}$, and $ch \in C$ is the message occurrence operator $ch \triangleright m$. All other MSC operators (with at least one argument) simply combine what their operands contribute. The validity of

$$(\varphi, u) \in [\![\mathbf{empty}]\!]_{u,CW} \quad \equiv \quad \langle \forall ch \in C :: \pi_1(\varphi).u.ch = \langle\rangle \rangle$$

is easy to see, because $\langle\rangle \sqsubseteq x$ holds for all $x \in M^*$. Similarly easily, we can show that we have

$$(\varphi, t) \in [\![\mathbf{any}]\!]_{u,CW} \quad \equiv \quad \langle \forall ch \in C, t' \in [u,t] :: \pi_1(\varphi).t'.ch = \langle\rangle \rangle$$

For message occurrence we observe that the least element $(\varphi, t) \in [\![ch \triangleright m]\!]_u$ (with respect to the inclusion $\sqsubseteq$ on the channel valuations) is the one with $\pi_1(\varphi).t.ch = \langle m \rangle$ and $\langle \forall ch' \in C, t' \in [u,t] : ch' \neq ch \vee t' \neq t : \pi_1(\varphi).t'.ch' = \langle\rangle \rangle$.

The validity of the claim for arbitrary MSCs α follows by induction on the structure of α. The essential step here is to pick behaviors from the closed-world semantics of the operands of a composite MSC to obtain the semantics of the composite, and to observe that no MSC composition operator introduces a message on its own. ■

Proposition 46 (Syntactic criterion for preemption refinement)
For all messages $ch \triangleright m \in \langle MSG \rangle$, and MSCs $\alpha \in \langle MSC \rangle$ we have

$$(ch \triangleright m \notin msgs.\alpha) \Rightarrow (\alpha \equiv_u^{CW} \lceil \alpha \rceil^{\overline{M} \setminus \{ch \triangleright m\}})$$

and

$$(ch \triangleright m \notin msgs.\alpha) \Rightarrow (\alpha \leq_p^{CW} \alpha \stackrel{ch \triangleright m}{\rightarrow} \beta)$$

□

PROOF The first implication is a trivial consequence of Proposition 45, and the definitions of $\equiv_u^{CW}$ and $\lceil . \rceil^{\cdot}$. For the second implication we observe for all messages $ch \triangleright m \in \langle \text{MSG} \rangle$ and MSCs $\alpha \in \langle \text{MSC} \rangle$:

$$
\begin{array}{ll}
 & (\varphi, t) \in [\![\alpha]\!]_{u,CW} \wedge ch \triangleright m \notin msgs.\alpha \\
\Rightarrow & \text{(* Proposition 45 *)} \\
 & (\varphi, t) \in [\![\alpha]\!]_{u,CW} \wedge \langle \forall v \in [u, t] :: m \notin \pi_1(\varphi).v.ch \rangle \\
\Rightarrow & \text{(* definition of } [\![\alpha \stackrel{ch \triangleright m}{\rightarrow} \beta]\!]_u\text{, } (\varphi, t) \in [\![\alpha]\!]_{u,CW} \text{ *)} \\
 & (\varphi, t) \in [\![\alpha \stackrel{ch \triangleright m}{\rightarrow} \beta]\!]_u
\end{array}
$$

■

B.3.2. Safety and Liveness

In the following paragraphs we support the discussion of safety and liveness we have started in Section 6.3. We start by showing the equivalence of the two alternative definitions given for safety and liveness, respectively. To this end, we also study properties (idempotence, monotonicity, finite "sub-conjunctivity", finite disjunctivity) of the *prefc*-operator. We need these properties later in this section, where we prove our results on the propagation of safety and liveness properties with respect to MSC composition.

Alternative Characterizations for Safety and Liveness

In the proofs we carry out later we make extensive use of the alternative characterizations for safety and liveness given in Section 6.3. Here, we show that the alternatives are, indeed, equivalent. The alternative definitions are based on the *prefc*-operator; in the following three propositions we establish several important properties of this operator. After this we consider the alternative characterizations. Based on these we establish the well-known result that every property is the disjunction of a safety and a liveness property. The proofs we give here directly mimic the corresponding ones from [AS85, Rem92, Möl99].

Before we start, we note that $\langle \forall q \subseteq (\tilde{C} \times S)^\infty :: q \subseteq prefc.q \rangle$ holds. This follows immediately by inspection of the definition of *prefc* (where $\psi \in (\tilde{C} \times S)^\infty$, $q \subseteq (\tilde{C} \times S)^\infty$):

$$\psi \in prefc.q \equiv \langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in q :: \psi \downarrow t = \varphi \downarrow t \rangle \rangle$$

Proposition 47 (*prefc* is idempotent) *For all $q \subseteq (\tilde{C} \times S)^\infty$ the following equality holds:*

$$prefc.(prefc.q) = prefc.q$$

□

Proof We observe for every $q \subseteq (\tilde{C} \times S)^\infty$ and $\psi \in (\tilde{C} \times S)^\infty$:

$$\begin{aligned}
& \psi \in prefc.(prefc.q) \\
\equiv \quad & (* \text{ definition of } prefc *) \\
& \langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in prefc.q :: \psi \downarrow t = \varphi \downarrow t \rangle\rangle \\
\equiv \quad & (* \text{ predicate calculus } *) \\
& \langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in (\tilde{C} \times S)^\infty :: \varphi \in prefc.q \wedge \psi \downarrow t = \varphi \downarrow t \rangle\rangle \\
\equiv \quad & (* \text{ definition of } prefc *) \\
& \langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in (\tilde{C} \times S)^\infty :: \langle \forall t' \in \mathbb{N} :: \langle \exists \hat{\varphi} \in q :: \varphi \downarrow t' = \hat{\varphi} \downarrow t' \rangle\rangle \wedge \psi \downarrow t = \varphi \downarrow t \rangle\rangle \\
\equiv \quad & (* \text{ predicate calculus } *) \\
& \langle \forall t \in \mathbb{N} :: \langle \exists \hat{\varphi} \in q :: \psi \downarrow t' = \hat{\varphi} \downarrow t' \rangle\rangle \\
\equiv \quad & (* \text{ definition of } prefc *) \\
& \psi \in prefc.q
\end{aligned}$$

■

Proposition 48 (*prefc* is monotonic) *For all $p, q \subseteq (\tilde{C} \times S)^\infty$ the following implication holds:*

$$p \subseteq q \quad \Rightarrow \quad prefc.p \subseteq prefc.q$$

□

Proof We observe for every $p, q \subseteq (\tilde{C} \times S)^\infty$ and $\psi \in (\tilde{C} \times S)^\infty$:

$$\begin{aligned}
& \psi \in prefc.p \\
\equiv \quad & (* \text{ definition of } prefc *) \\
& \langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in p :: \psi \downarrow t = \varphi \downarrow t \rangle\rangle \\
\Rightarrow \quad & (* \ p \subseteq q \ *) \\
& \langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in q :: \psi \downarrow t = \varphi \downarrow t \rangle\rangle \\
\equiv \quad & (* \text{ definition of } prefc *) \\
& \psi \in prefc.q
\end{aligned}$$

■

Proposition 49 (*prefc* is finitely "sub-conjunctive") *For all $p, q \subseteq (\tilde{C} \times S)^\infty$ the following equality holds:*

$$prefc.(p \cap q) \subseteq prefc.p \cap prefc.q$$

□

Proof We observe for every $p, q \subseteq (\tilde{C} \times S)^\infty$ and $\psi \in (\tilde{C} \times S)^\infty$:

$$
\begin{array}{ll}
& \psi \in \mathit{prefc}.(p \cap q) \\
\equiv & (* \text{ definition of } \mathit{prefc} *) \\
& \langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in (p \cap q) :: \psi \downarrow t = \varphi \downarrow t \rangle \rangle \\
\equiv & (* \text{ set theory } *) \\
& \langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in (\tilde{C} \times S)^\infty :: \varphi \in p \wedge \varphi \in q \wedge \psi \downarrow t = \varphi \downarrow t \rangle \rangle \\
\Rightarrow & (* \text{ predicate calculus } *) \\
& \quad\; \langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in (\tilde{C} \times S)^\infty :: \varphi \in p \wedge \psi \downarrow t = \varphi \downarrow t \rangle \rangle \\
& \wedge \; \langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in (\tilde{C} \times S)^\infty :: \varphi \in q \wedge \psi \downarrow t = \varphi \downarrow t \rangle \rangle \\
\equiv & (* \text{ definition of } \mathit{prefc}, \text{ twice } *) \\
& \psi \in \mathit{prefc}.p \wedge \psi \in \mathit{prefc}.q
\end{array}
$$

■

With these preliminaries in place we can now prove the equivalence of the alternative characterizations for safety and liveness. This is the topic of the following two propositions.

Proposition 50 (Alternative characterization for safety)
For all $q \in (\tilde{C} \times S)^\infty$ the following equivalence holds:

$$
\begin{array}{ll}
& \langle \forall \psi \in (\tilde{C} \times S)^\infty : \psi \notin q : \langle \exists t \in \mathbb{N} :: \langle \forall \varphi \in (\tilde{C} \times S)^\infty :: (\psi \downarrow t) \frown \varphi \notin q \rangle \rangle \rangle \\
\equiv & (\mathit{prefc}.q = q)
\end{array}
$$

□

Proof We prove this equivalence by mutual implication.

<u>"⇒"</u>:
Let q be such that

$$\langle \forall \psi \in (\tilde{C} \times S)^\infty : \psi \notin q : \langle \exists t \in \mathbb{N} :: \langle \forall \varphi \in (\tilde{C} \times S)^\infty :: (\psi \downarrow t) \frown \varphi \notin q \rangle \rangle \rangle \qquad (\star)$$

holds; we show that this implies

$$\mathit{prefc}.q \subseteq q$$

Let $\psi \in \mathit{prefc}.q$, i.e. we have $\langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in q :: \psi \downarrow t = \varphi \downarrow t \rangle \rangle$. We observe:

$$
\begin{array}{ll}
& \psi \notin q \\
\Rightarrow & (* \text{ Assumption } (\star) *) \\
& \langle \exists t_0 \in \mathbb{N} :: \langle \forall \varphi \in (\tilde{C} \times S)^\infty :: (\psi \downarrow t_0) \frown \varphi \notin q \rangle \rangle \\
\equiv & (* \text{ predicate calculus } *)
\end{array}
$$

$\neg\langle\forall t_0 \in \mathbb{N} :: \langle\exists\varphi \in (\tilde{C} \times S)^\infty :: (\psi\downarrow t_0)\frown\varphi \in q\rangle\rangle$

$\equiv$ (∗ identify $\hat{\varphi}\uparrow t_0$ with φ ∗)

$\neg\langle\forall t_0 \in \mathbb{N} :: \langle\exists\hat{\varphi} \in q :: \psi\downarrow t_0 = \varphi\downarrow t_0\rangle\rangle$

$\equiv$ (∗ $\psi \in \mathit{prefc}.q$ ∗)

false

By taking the contrapositive, we have established $\psi \in q$, and thus $\mathit{prefc}.q \subseteq q$ holds. Because of $q \subseteq \mathit{prefc}.q$ for all $q \subseteq (\tilde{C} \times S)^\infty$ we obtain $\mathit{prefc}.q = q$ as desired.

"$\Leftarrow$":
Assume that $\mathit{prefc}.q \subseteq q$ holds, and let $\psi \in (\tilde{C} \times S)^\infty$ be arbitrary but fixed. We observe:

$\psi \notin q \;\Rightarrow\; \langle\exists t \in \mathbb{N} :: \langle\forall\varphi \in (\tilde{C} \times S)^\infty :: (\psi\downarrow t)\frown\varphi \notin q\rangle\rangle$

$\equiv$ (∗ contrapositive ∗)

$\langle\forall t \in \mathbb{N} :: \langle\exists\varphi \in (\tilde{C} \times S)^\infty :: (\psi\downarrow t)\frown\varphi \in q\rangle\rangle \;\Rightarrow\; \psi \in q$

$\equiv$ (∗ identify $\hat{\varphi}\uparrow t$ with φ ∗)

$\langle\forall t \in \mathbb{N} :: \langle\exists\hat{\varphi} \in q :: \psi\downarrow t = \hat{\varphi}\downarrow t\rangle\rangle \;\Rightarrow\; \psi \in q$

$\equiv$ (∗ definition of prefc ∗)

$\psi \in \mathit{prefc}.q \;\Rightarrow\; \psi \in q$

$\equiv$ (∗ $\mathit{prefc}.q = q$ ∗)

true

■

Proposition 51 (Alternative characterization for liveness) *Let $q \subseteq (\tilde{C} \times S)^\infty$ be a property. q is a liveness property if and only if*

$$\mathit{prefc}.q = (\tilde{C} \times S)^\infty$$

holds. □

Proof By definition of prefc we obtain that $\mathit{prefc}.q \subseteq (\tilde{C} \times S)^\infty$ holds. For the inclusion $(\tilde{C} \times S)^\infty \subseteq \mathit{prefc}.q$ we derive:

q is a liveness property

$\equiv$ (∗ definition of liveness ∗)

$\langle\forall\psi \in (\tilde{C} \times S)^\infty, t \in \mathbb{N} :: \langle\exists\varphi \in (\tilde{C} \times S)^\infty :: (\psi\downarrow t)\frown\varphi \in q\rangle\rangle$

$\equiv$ (∗ identify $\hat{\varphi}\uparrow t$ with φ, predicate calculus ∗)

$\langle\forall\psi \in (\tilde{C} \times S)^\infty, t \in \mathbb{N} :: \langle\exists\hat{\varphi} \in q :: \psi\downarrow t = \hat{\varphi}\downarrow t\rangle\rangle$

$\equiv$ (∗ definition of *prefc* ∗)
$\langle \forall \psi \in (\tilde{C} \times S)^\infty :: \psi \in prefc.q \rangle$
$\equiv$ (∗ set theory ∗)
$(\tilde{C} \times S)^\infty \subseteq prefc.q$

■

Proposition 52 (Property decomposition) *Let $q \subseteq (\tilde{C} \times S)^\infty$ be a property. Then there is a safety property $q_s \subseteq (\tilde{C} \times S)^\infty$ and a liveness property $q_l \subseteq (\tilde{C} \times S)^\infty$, such that*

$$q = q_s \cap q_l$$

holds. □

Proof We define two properties $\mathbb{S}.q \subseteq (\tilde{C} \times S)^\infty$ and $\mathbb{L}.q \subseteq (\tilde{C} \times S)^\infty$ as follows:

$$\mathbb{S}.q \stackrel{\mathsf{def}}{=} prefc.q$$

and

$$\psi \in \mathbb{L}.q \stackrel{\mathsf{def}}{=} \psi \in \mathbb{S}.q \Rightarrow \psi \in q$$

Then $q = \mathbb{S}.q \cap \mathbb{L}.q$ holds on account of:

$\psi \in \mathbb{S}.q \cap \mathbb{L}.q$
$\equiv$ (∗ definitions of $\mathbb{S}$ and $\mathbb{L}$ ∗)
$\psi \in prefc.q \wedge (\psi \in prefc.q \Rightarrow \psi \in q)$
$\equiv$ (∗ predicate calculus ∗)
$\psi \in prefc.q \wedge \psi \in q$
$\equiv$ (∗ $q \subseteq prefc.q$ ∗)
$\psi \in q$

In the following we prove that $\mathbb{S}.q$ is a safety property and $\mathbb{L}.q$ is a liveness property, from which the claim of the proposition follows.

<u>$\mathbb{S}.q$ is a safety property</u>:

We observe:

$prefc.(\mathbb{S}.q)$
$=$ (∗ definition of $\mathbb{S}.q$ ∗)
$prefc.(prefc.q)$

$=$ (* idempotence of *prefc* *)

$prefc.q$

$=$ (* definition of $\mathbb{S}.q$ *)

$\mathbb{S}.q$

<u>$\mathbb{L}.q$ is a liveness property</u>:

We show that $(\tilde{C} \times S)^\infty \subseteq prefc.(\mathbb{L}.q)$ holds (the other direction is trivial):

$\psi \in (\tilde{C} \times S)^\infty$

$\equiv$ (* predicate calculus *)

$\psi \in prefc.q \vee \psi \notin prefc.q$

$\equiv$ (* definition of $\notin$ *)

$\psi \in prefc.q \vee \psi \in ((\tilde{C} \times S)^\infty \setminus prefc.q)$

$\Rightarrow$ (* $q' \subseteq prefc.q'$ for arbitrary q' *)

$\psi \in prefc.q \vee \psi \in prefc.((\tilde{C} \times S)^\infty \setminus prefc.q)$

$\equiv$ (* *prefc* is finitely disjunctive, see below *)

$\psi \in prefc.(q \cup ((\tilde{C} \times S)^\infty \setminus prefc.q))$

$\equiv$ (* definition of $\mathbb{L}.q$ *)

$\psi \in prefc.(\mathbb{L}.q)$

<u>*prefc* is finitely disjunctive</u>:

Let $q_0, q_1 \subseteq (\tilde{C} \times S)^\infty$ be arbitrary properties, $\psi \in prefc.(q_0 \cup q_1)$, and assume that $\psi \notin prefc.q_0$ holds. We show that this is equivalent to $\psi \in prefc.q_1$ as follows:

$\psi \in prefc.(q_0 \cup q_1)$

$\equiv$ (* definition of *prefc* *)

$\langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in (q_0 \cup q_1) :: \psi \downarrow t = \varphi \downarrow t \rangle \rangle$

$\equiv$ (* predicate calculus, set theory *)

$\langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in (\tilde{C} \times S)^\infty :: (\varphi \in q_0 \vee \varphi \in q_1) \wedge \psi \downarrow t = \varphi \downarrow t \rangle \rangle$

$\equiv$ (* predicate calculus *)

$\langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in q_0 :: \psi \downarrow t = \varphi \downarrow t \rangle \vee \langle \exists \varphi \in q_1 :: \psi \downarrow t = \varphi \downarrow t \rangle \rangle$

$\equiv$ (* $\psi \notin prefc.q_0$ *)

$\langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in q_1 :: \psi \downarrow t = \varphi \downarrow t \rangle \rangle$

$\equiv$ (* definition of *prefc* *)

$\psi \in prefc.q_1$

■

Safety Preservers

The following proposition characterizes the MSC operators that preserve the safety of their operands.

Proposition 53 (Safety preserving operators) *Let* $\alpha, \beta \in \langle MSC \rangle$, $p \in \langle GUARD \rangle$, $\dagger \in \{\ ;\ ,\ |\ ,\ \sim\ ,\ \otimes\ \}$, $\texttt{<L>} \in \{\texttt{<*>}, \texttt{<m>}, \texttt{<m,n>}\}$ *with* $m, n \in \mathbb{N}$, $ch \triangleright m \in \langle MSG \rangle$, *and* $X \in \langle MSCNAME \rangle$ *with* $(X, \alpha) \in MSCR$. *Furthermore, let* $[\![\alpha]\!]$ *and* $[\![\beta]\!]$ *be safety properties. Then each of the following is a safety property:*

$$[\![p : \alpha]\!]$$
$$[\![\alpha \dagger \beta]\!]$$
$$[\![\alpha \uparrow_{\texttt{<L>}}]\!]$$
$$[\![\alpha \stackrel{ch \triangleright m}{\rightarrow} \beta]\!]$$
$$[\![\alpha \Uparrow_{ch \triangleright m}]\!]$$
$$[\![\rightarrow X]\!]$$

□

PROOF Let the assumptions be as stated in the proposition. For every $q \in \{[\![p : \alpha]\!], [\![\alpha \dagger \beta]\!], [\![\alpha \uparrow_{\texttt{<L>}}]\!], [\![\alpha \stackrel{ch \triangleright m}{\rightarrow} \beta]\!], [\![\alpha \Uparrow_{ch \triangleright m}]\!], [\![\rightarrow X]\!]\}$ we show that $prefc.q \subseteq q$ holds.

$\underline{q = [\![\alpha\ ;\ \beta]\!]}$:

The idea here is to distinguish the cases where α represents an infinite and a finite behavior, respectively. To this end, we observe for an arbitrary $\psi \in (\tilde{C} \times S)^\infty$:

$$
\begin{array}{ll}
& ((\psi \in prefc.[\![\alpha\ ;\ \beta]\!] \wedge (\psi, \infty) \in [\![\alpha]\!]_0) \Rightarrow \psi \in [\![\alpha\ ;\ \beta]\!]) \\
& \wedge\ ((\psi \in prefc.[\![\alpha\ ;\ \beta]\!] \wedge \langle \exists t \in \mathbb{N} :: (\psi, t) \in [\![\alpha]\!]_0 \rangle) \Rightarrow \psi \in [\![\alpha\ ;\ \beta]\!]) \\
\Rightarrow & (*\ \text{predicate calculus}\ *) \\
& (\quad (\psi \in prefc.[\![\alpha\ ;\ \beta]\!] \wedge (\psi, \infty) \in [\![\alpha]\!]_0) \\
& \quad \vee\ (\psi \in prefc.[\![\alpha\ ;\ \beta]\!] \wedge \langle \exists t \in \mathbb{N} :: (\psi, t) \in [\![\alpha]\!]_0 \rangle)) \Rightarrow \psi \in [\![\alpha\ ;\ \beta]\!] \\
\equiv & (*\ \text{predicate calculus}\ *) \\
& (\psi \in prefc.[\![\alpha\ ;\ \beta]\!] \wedge ((\psi, \infty) \in [\![\alpha]\!]_0 \vee \langle \exists t \in \mathbb{N} :: (\psi, t) \in [\![\alpha]\!]_0 \rangle)) \Rightarrow \psi \in [\![\alpha\ ;\ \beta]\!] \\
\Rightarrow & (*\ \psi \in prefc.[\![\alpha\ ;\ \beta]\!] \Rightarrow \langle \exists t \in \mathbb{N}_\infty :: (\psi, t) \in [\![\alpha]\!]_0 \rangle, \text{ see below}\ *) \\
& \psi \in prefc.[\![\alpha\ ;\ \beta]\!] \Rightarrow \psi \in [\![\alpha\ ;\ \beta]\!]
\end{array}
$$

This leaves us with the following three proof obligations:

1. $\psi \in prefc.[\![\alpha\ ;\ \beta]\!] \Rightarrow \langle \exists t \in \mathbb{N}_\infty :: (\psi, t) \in [\![\alpha]\!]_0 \rangle$

2. $(\psi \in prefc.[\![\alpha\ ;\ \beta]\!] \wedge (\psi, \infty) \in [\![\alpha]\!]_0) \Rightarrow \psi \in [\![\alpha\ ;\ \beta]\!]$

3. $(\psi \in \mathit{prefc}.[\![\alpha\,;\,\beta]\!] \wedge \langle \exists t \in \mathbb{N} :: (\psi, t) \in [\![\alpha]\!]_0\rangle) \Rightarrow \psi \in [\![\alpha\,;\,\beta]\!]$

We now discharge each of these proof obligations, in turn.

$\underline{\psi \in \mathit{prefc}.[\![\alpha\,;\,\beta]\!] \Rightarrow \langle \exists t \in \mathbb{N}_\infty :: (\psi, t) \in [\![\alpha]\!]_0\rangle}$:

$$
\begin{array}{ll}
 & \psi \in \mathit{prefc}.[\![\alpha\,;\,\beta]\!] \\
\Rightarrow & (*\ [\![\alpha\,;\,\beta]\!] \subseteq [\![\alpha]\!], \text{ see below; monotonicity of } \mathit{prefc}\ *) \\
 & \psi \in \mathit{prefc}.[\![\alpha]\!] \\
\Rightarrow & (*\ [\![\alpha]\!] \text{ is a safety property } *) \\
 & \psi \in [\![\alpha]\!] \\
\equiv & (*\ \text{definition of } [\![\alpha]\!]\ *) \\
 & \langle \exists t \in \mathbb{N}_\infty :: (\psi, t) \in [\![\alpha]\!]_0\rangle
\end{array}
$$

To see the validity of $[\![\alpha\,;\,\beta]\!] \subseteq [\![\alpha]\!]$ we observe:

$$
\begin{array}{ll}
 & \psi \in [\![\alpha\,;\,\beta]\!] \\
\equiv & (*\ \text{definition of } [\![\alpha\,;\,\beta]\!]\ *) \\
 & \langle \exists t \in \mathbb{N}_\infty :: (\psi, t) \in [\![\alpha\,;\,\beta]\!]_0\rangle \\
\equiv & (*\ \text{definition of } [\![\alpha\,;\,\beta]\!]_0\ *) \\
 & \langle \exists t, t' \in \mathbb{N}_\infty :: (\psi, t') \in [\![\alpha]\!]_0 \wedge (\psi, t) \in [\![\beta]\!]_{t'}\rangle \\
\Rightarrow & (*\ \text{predicate calculus } *) \\
 & \langle \exists t' \in \mathbb{N}_\infty :: (\psi, t') \in [\![\alpha]\!]_0\rangle \\
\equiv & (*\ \text{definition of } [\![\alpha]\!]\ *) \\
 & \psi \in [\![\alpha]\!]
\end{array}
$$

$\underline{(\psi \in \mathit{prefc}.[\![\alpha\,;\,\beta]\!] \wedge (\psi, \infty) \in [\![\alpha]\!]_0)) \Rightarrow \psi \in [\![\alpha\,;\,\beta]\!]}$:

We observe for every $\alpha, \beta \in \langle \mathrm{MSC}\rangle$ and $\psi \in (\tilde{C} \times S)^\infty$:

$$
\begin{array}{ll}
 & (\psi, \infty) \in [\![\alpha]\!]_0 \\
\Rightarrow & (*\ \text{predicate calculus, property of } [\![.]\!]_\infty\ *) \\
 & (\psi, \infty) \in [\![\alpha]\!]_0 \wedge (\psi, \infty) \in [\![\beta]\!]_\infty \\
\equiv & (*\ \text{definition of } [\![.\,;\,.]\!], \text{ Definition 1 } *) \\
 & (\psi, \infty) \in [\![\alpha\,;\,\beta]\!]
\end{array}
$$

$\underline{(\psi \in \mathit{prefc}.[\![\alpha\,;\,\beta]\!] \wedge \langle \exists t \in \mathbb{N} :: (\psi, t) \in [\![\alpha]\!]_0 \rangle) \Rightarrow \psi \in [\![\alpha\,;\,\beta]\!]}$:

We observe for every $\alpha, \beta \in \langle \mathrm{MSC} \rangle$ and $\psi \in (\tilde{C} \times S)^\infty$:

$$
\begin{array}{ll}
& \langle \exists \hat{t} \in \mathbb{N} :: \psi \in \mathit{prefc}.[\![\alpha\,;\,\beta]\!] \wedge (\psi, \hat{t}) \in [\![\alpha]\!]_0 \rangle \\
\equiv & (* \text{ definition of } \mathit{prefc} *) \\
& \langle \exists \hat{t} \in \mathbb{N} :: \\
& \quad \langle \forall t \in \mathbb{N} :: \\
& \quad\quad \langle \exists \varphi \in (\tilde{C} \times S)^\infty :: \\
& \quad\quad\quad \varphi \in [\![\alpha\,;\,\beta]\!] \wedge \psi \downarrow t = \varphi \downarrow t \rangle \rangle \wedge (\psi, \hat{t}) \in [\![\alpha]\!]_0 \rangle \\
\equiv & (* \text{ definition of } [\![\alpha\,;\,\beta]\!] *) \\
& \langle \exists \hat{t} \in \mathbb{N} :: \\
& \quad \langle \forall t \in \mathbb{N} :: \\
& \quad\quad \langle \exists \varphi \in (\tilde{C} \times S)^\infty, t', t'' \in \mathbb{N}_\infty :: \\
& \quad\quad\quad (\varphi, t') \in [\![\alpha]\!]_0 \wedge (\varphi, t'') \in [\![\beta]\!]_{t'} \wedge \psi \downarrow t = \varphi \downarrow t \rangle \rangle \wedge (\psi, \hat{t}) \in [\![\alpha]\!]_0 \rangle \\
\Rightarrow & (* \text{ predicate calculus, aiming at using } \hat{t} \text{ as the time point ``between'' } \alpha \text{ and } \beta *) \\
& \langle \exists \hat{t} \in \mathbb{N} :: \\
& \quad \langle \forall t \in \mathbb{N} : t \geq \hat{t} : \\
& \quad\quad \langle \exists \varphi \in (\tilde{C} \times S)^\infty, t', t'' \in \mathbb{N}_\infty :: \\
& \quad\quad\quad (\varphi, t') \in [\![\alpha]\!]_0 \wedge (\varphi, t'') \in [\![\beta]\!]_{t'} \wedge (\psi \uparrow \hat{t}) \downarrow (t - \hat{t}) = (\varphi \uparrow \hat{t}) \downarrow (t - \hat{t}) \rangle \rangle \\
& \quad \wedge (\psi, \hat{t}) \in [\![\alpha]\!]_0 \rangle \\
\Rightarrow & (* \text{ predicate calculus, fix } t' \text{ to } \hat{t} *) \\
& \langle \exists \hat{t} \in \mathbb{N} :: \\
& \quad \langle \forall t \in \mathbb{N} : t \geq \hat{t} : \\
& \quad\quad \langle \exists \varphi \in (\tilde{C} \times S)^\infty, t'' \in \mathbb{N}_\infty :: \\
& \quad\quad\quad (\varphi, \hat{t}) \in [\![\alpha]\!]_0 \wedge (\varphi, t'') \in [\![\beta]\!]_{\hat{t}} \wedge (\psi \uparrow \hat{t}) \downarrow (t - \hat{t}) = (\varphi \uparrow \hat{t}) \downarrow (t - \hat{t}) \rangle \rangle \\
& \quad \wedge (\psi, \hat{t}) \in [\![\alpha]\!]_0 \rangle \\
\equiv & (* \text{ predicate calculus, shift range of } t\text{; we aim at using the definition of } \mathit{prefc} *) \\
& \langle \exists \hat{t} \in \mathbb{N} :: \\
& \quad \langle \forall \tilde{t} \in \mathbb{N} : \tilde{t} \geq \hat{t} : \\
& \quad\quad \langle \exists \varphi \in (\tilde{C} \times S)^\infty, \tilde{t}'' \in \mathbb{N}_\infty :: \\
& \quad\quad\quad (\varphi, \hat{t}) \in [\![\alpha]\!]_0 \wedge (\varphi \uparrow \hat{t}, \tilde{t}'') \in [\![\beta]\!]_0 \wedge (\psi \uparrow \hat{t}) \downarrow \tilde{t} = (\varphi \uparrow \hat{t}) \downarrow \tilde{t} \rangle \rangle \\
& \quad \wedge (\psi, \hat{t}) \in [\![\alpha]\!]_0 \rangle \\
\Rightarrow & (* \text{ predicate calculus } *) \\
& \langle \exists \hat{t} \in \mathbb{N} :: \\
& \quad \langle \forall \tilde{t} \in \mathbb{N} : \tilde{t} \geq \hat{t} : \\
& \quad\quad \langle \exists \tilde{\varphi} \in (\tilde{C} \times S)^\infty, \tilde{t}'' \in \mathbb{N}_\infty :: \\
& \quad\quad\quad (\tilde{\varphi}, \tilde{t}'') \in [\![\beta]\!]_0 \wedge (\psi \uparrow \hat{t}) \downarrow \tilde{t} = \tilde{\varphi} \downarrow \tilde{t} \rangle \rangle \wedge (\psi, \hat{t}) \in [\![\alpha]\!]_0 \rangle
\end{array}
$$

$\equiv$ (* definition of *prefc* *)

$\langle \exists \hat{t} \in \mathbb{N} :: \psi \uparrow \hat{t} \in prefc.[\![\beta]\!] \wedge (\psi, \hat{t}) \in [\![\alpha]\!]_0 \rangle$

$\Rightarrow$ (* $[\![\beta]\!]$ is a safety property *)

$\langle \exists \hat{t} \in \mathbb{N} :: \psi \uparrow \hat{t} \in [\![\beta]\!] \wedge (\psi, \hat{t}) \in [\![\alpha]\!]_0 \rangle$

$\equiv$ (* definition of $[\![.]\!]$ *)

$\langle \exists \hat{t} \in \mathbb{N}, \hat{t}' \in \mathbb{N}_\infty :: (\psi, \hat{t}) \in [\![\alpha]\!]_0 \wedge (\psi \uparrow \hat{t}, \hat{t}') \in [\![\beta]\!]_0 \rangle$

$\equiv$ (* property of $[\![.]\!]_u$ (independence of absolute time) *)

$\langle \exists \hat{t} \in \mathbb{N}, \hat{t}' \in \mathbb{N}_\infty :: (\psi, \hat{t}) \in [\![\alpha]\!]_0 \wedge (\psi, \hat{t}' + \hat{t}) \in [\![\beta]\!]_{\hat{t}} \rangle$

$\Rightarrow$ (* definition of $[\![. \,;\, .]\!]_u$ *)

$\psi \in [\![\alpha \,;\, \beta]\!]_u$

This concludes the proof of $prefc.[\![\alpha \,;\, \beta]\!] \subseteq [\![\alpha \,;\, \beta]\!]$.

$\underline{q = [\![\alpha \mid \beta]\!]}$:

$\psi \in prefc.[\![\alpha \mid \beta]\!]$

$\equiv$ (* definition of *prefc* *)

$\langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in (\tilde{C} \times S)^\infty :: \varphi \in [\![\alpha \mid \beta]\!] \wedge \psi \downarrow t = \varphi \downarrow t \rangle \rangle$

$\equiv$ (* definition of $[\![\alpha \mid \beta]\!]$ *)

$$\begin{array}{l} \langle \forall t \in \mathbb{N} :: \\ \quad \langle \exists \varphi \in (\tilde{C} \times S)^\infty, t' \in \mathbb{N}_\infty :: \\ \quad\quad ((\varphi, t') \in [\![\alpha]\!]_0 \vee (\varphi, t') \in [\![\beta]\!]_0) \wedge \psi \downarrow t = \varphi \downarrow t \rangle \rangle \end{array}$$

$\Rightarrow$ (* predicate calculus *)

$$\begin{array}{ll} & \langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in (\tilde{C} \times S)^\infty, t' \in \mathbb{N}_\infty :: (\varphi, t') \in [\![\alpha]\!]_0 \wedge \psi \downarrow t = \varphi \downarrow t \rangle \rangle \\ \vee & \langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in (\tilde{C} \times S)^\infty, t' \in \mathbb{N}_\infty :: (\varphi, t') \in [\![\beta]\!]_0 \wedge \psi \downarrow t = \varphi \downarrow t \rangle \rangle \end{array}$$

$\equiv$ (* definition of *prefc* *)

$\psi \in prefc.[\![\alpha]\!] \vee \psi \in prefc.[\![\beta]\!]$

$\equiv$ (* $[\![\alpha]\!]$ and $[\![\beta]\!]$ are safety properties *)

$\psi \in [\![\alpha]\!] \vee \psi \in [\![\beta]\!]$

$\equiv$ (* definition of $[\![. \mid .]\!]$ *)

$\psi \in [\![\alpha \mid \beta]\!]$

$\underline{q = [\![p : \alpha]\!]}$:

$$
\begin{array}{ll}
& \psi \in \mathit{prefc}.[\![p : \alpha]\!] \\
\equiv & (* \text{ definition of } \mathit{prefc},\ [\![p : \alpha]\!] *) \\
& \langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in (\tilde{C} \times S)^\infty, t' \in \mathbb{N}_\infty :: (\varphi, t') \in [\![p : \alpha]\!]_0 \wedge \psi \downarrow t = \varphi \downarrow t \rangle\rangle \\
\equiv & (* \text{ definition of } [\![p : \alpha]\!]_0 *) \\
& \langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in (\tilde{C} \times S)^\infty, t' \in \mathbb{N}_\infty :: (\varphi, t') \in [\![\alpha]\!]_0 \wedge \pi_2(\varphi).0 \in [\![p]\!] \wedge \psi \downarrow t = \varphi \downarrow t \rangle\rangle \\
\Rightarrow & (* \text{ definition of } \mathit{prefc},\ [\![\alpha]\!], \text{ predicate calculus } *) \\
& \psi \in \mathit{prefc}.[\![\alpha]\!] \wedge \pi_2(\psi).0 \in [\![p]\!] \\
\Rightarrow & (* [\![\alpha]\!] \text{ is a safety property, definition of } [\![p : \alpha]\!] *) \\
& \psi \in [\![p : \alpha]\!]
\end{array}
$$

$\underline{q = [\![\alpha \sim \beta]\!]}$:

$\psi \notin [\![\alpha \sim \beta]\!]$ implies that there is no type correct oracle $bs \in (C \to \mathbb{B}^*)^\infty$, no $\psi_1, \psi_2 \in (\tilde{C} \times S)^\infty$, or no $t_1, t_2 \in \mathbb{N}_\infty$, such that both

$$((\mathit{filter}.(\pi_1(\varphi) \uparrow u).bs.\,\text{true}, (\pi_2(\varphi) \uparrow u)) \frown \psi_1, t_1) \in [\![\alpha]\!]_0$$

and

$$((\mathit{filter}.(\pi_1(\varphi) \uparrow u).bs.\,\text{false}, (\pi_2(\varphi) \uparrow u)) \frown \psi_2, t_2) \in [\![\beta]\!]_0$$

hold. This, in turn, implies that we have either

$$(\mathit{filter}.(\pi_1(\varphi) \uparrow u).bs.\,\text{true}, (\pi_2(\varphi) \uparrow u)) \frown \psi_1 \notin \mathit{prefc}.[\![\alpha]\!]$$

or

$$(\mathit{filter}.(\pi_1(\varphi) \uparrow u).bs.\,\text{false}, (\pi_2(\varphi) \uparrow u)) \frown \psi_2 \notin \mathit{prefc}.[\![\beta]\!]$$

because $[\![\alpha]\!]$ and $[\![\beta]\!]$ both are safety properties. From this, however, we easily conclude the validity of $\psi \notin \mathit{prefc}.[\![\alpha \sim \beta]\!]$ by inspection of the definition of *prefc* and $[\![\alpha \sim \beta]\!]$. By taking the contrapositive, we obtain $\psi \in \mathit{prefc}.[\![\alpha \sim \beta]\!] \Rightarrow \psi \in [\![\alpha \sim \beta]\!]$.

$\underline{q = [\![\alpha \otimes \beta]\!]}$:

Observe that $[\![\alpha \otimes \beta]\!]$ is the intersection of the sets $[\![\alpha]\!]$, $[\![\beta]\!]$, and q', where we define q' by:

$$
\begin{array}{l}
q' \stackrel{\text{def}}{=} \{\psi \in (\tilde{C} \times S)^\infty : \\
\quad \langle \exists t \in \mathbb{N}_\infty :: \\
\qquad ((\psi, t) \in [\![\alpha]\!]_0 \vee (\psi, t) \in [\![\beta]\!]_0) \\
\quad \Rightarrow (\langle \forall X \in (\mathit{msgs}.\alpha \cap \mathit{msgs}.\beta)^*, \hat{\psi} \in (\tilde{C} \times S)^\infty, ch \in C, t' \in [u, t] \cap \mathbb{N} :: \\
\qquad\qquad ((X \neq \langle\rangle) \wedge (\pi_1(\hat{\psi}).t'.ch = \pi_1(\psi).t'.ch \setminus X)) \\
\qquad \Rightarrow \langle \forall t'' \in \mathbb{N} :: (\hat{\psi}, t'') \notin [\![\alpha]\!]_u \wedge (\hat{\psi}, t'') \notin [\![\beta]\!]_u \rangle\rangle\rangle\}
\end{array}
$$

Because each of these is a safety property, which is given in the case of $[\![\alpha]\!]$ and $[\![\beta]\!]$, and is easy to see for q', we can conclude:

$$\begin{aligned}
& \quad \mathit{prefc}.[\![\alpha \otimes \beta]\!] \\
= & \quad (* \text{ definition of } [\![\alpha]\!], [\![\beta]\!], \text{ and } q' *) \\
& \quad \mathit{prefc}.([\![\alpha]\!] \cap [\![\beta]\!] \cap q') \\
\subseteq & \quad (* \; \mathit{prefc} \text{ is finitely ``sub-conjunctive'' } *) \\
& \quad \mathit{prefc}.[\![\alpha]\!] \cap \mathit{prefc}.[\![\beta]\!] \cap \mathit{prefc}.q' \\
= & \quad (* \; [\![\alpha]\!], [\![\beta]\!], \text{ and } q' \text{ are safety properties } *) \\
& \quad [\![\alpha]\!] \cap [\![\beta]\!] \cap q' \\
= & \quad (* \text{ definition of } [\![\alpha]\!], [\![\beta]\!], \text{ and } q' *) \\
& \quad [\![\alpha \otimes \beta]\!]
\end{aligned}$$

This shows that $[\![\alpha \otimes \beta]\!]$ is a safety property, if both $[\![\alpha]\!]$ and $[\![\beta]\!]$ are.

$q = [\![\alpha\!\uparrow_{\lhd I \rhd}]\!]$:

We show for all $n \in \mathbb{N}$ that $[\![\alpha\!\uparrow_{\lhd 0,n \rhd}]\!]$ is a pure safety property if $[\![\alpha]\!]$ is a pure safety property by induction on n. For the other finite loop ranges we reduce the claim to this result by means of the case distinction of the loop operator's definition, and the observation that ; preserves safety properties.

$n = 0$:

$$\begin{aligned}
& \quad \psi \in \mathit{prefc}.[\![\alpha\!\uparrow_{\lhd 0,0 \rhd}]\!] \\
\equiv & \quad (* \text{ definition of } [\![\alpha\!\uparrow_{\lhd 0,0 \rhd}]\!] *) \\
& \quad \psi \in \mathit{prefc}.[\![\mathbf{empty}]\!] \\
\equiv & \quad (* \; [\![\mathbf{empty}]\!] \text{ is a safety property } *) \\
& \quad \psi \in [\![\mathbf{empty}]\!] \\
\equiv & \quad (* \text{ definition of } [\![\alpha\!\uparrow_{\lhd 0,0 \rhd}]\!] *) \\
& \quad \psi \in [\![\alpha\!\uparrow_{\lhd 0,0 \rhd}]\!]
\end{aligned}$$

$n \rightsquigarrow (n+1)$:

$$\begin{aligned}
& \quad \psi \in \mathit{prefc}.[\![\alpha\!\uparrow_{\lhd 0,n+1 \rhd}]\!] \\
\equiv & \quad (* \; \alpha\!\uparrow_{\lhd 0,n+1 \rhd} \equiv_u \alpha \,;\, (\alpha\!\uparrow_{\lhd 0,n \rhd}) *) \\
& \quad \psi \in \mathit{prefc}.[\![\alpha \,;\, (\alpha\!\uparrow_{\lhd 0,n \rhd})]\!]
\end{aligned}$$

$\Rightarrow$ (* $[\![\alpha]\!]$ and $[\![\alpha\uparrow_{\lhd 0,n\rhd}]\!]$ are safety properties, ; preserves safety *)

$\psi \in [\![\alpha \,;\, (\alpha\uparrow_{\lhd 0,n\rhd})]\!]$

$\equiv$ (* $\alpha\uparrow_{\lhd 0,n+1\rhd} \;\equiv_u\; \alpha \,;\, (\alpha\uparrow_{\lhd 0,n\rhd})$ *)

$\psi \in [\![\alpha\uparrow_{\lhd 0,n+1\rhd}]\!]$

$q = [\![\alpha \stackrel{ch\rhd m}{\rightarrow} \beta]\!]$:

We make a case distinction on whether the preemptive message occurs in the execution under consideration or not.

In a first step, we assume $\langle \forall v \in \mathbb{N} : \langle \exists t \in \mathbb{N}_\infty :: (\psi, t) \in [\![\alpha]\!]_0\rangle : m \notin \pi_1(\psi).v.ch\rangle$ and observe under this assumption:

$\psi \in \mathit{prefc}.[\![\alpha \stackrel{ch\rhd m}{\rightarrow} \beta]\!]$

$\Rightarrow$ (* definition of $[\![\alpha \stackrel{ch\rhd m}{\rightarrow} \beta]\!]$ *)

$\psi \in \mathit{prefc}.[\![\alpha]\!]$

$\Rightarrow$ (* $[\![\alpha]\!]$ is a safety property *)

$\psi \in [\![\alpha]\!]$

$\Rightarrow$ (* definition of $[\![\alpha \stackrel{ch\rhd m}{\rightarrow} \beta]\!]$ *)

$\psi \in [\![\alpha \stackrel{ch\rhd m}{\rightarrow} \beta]\!]$

This establishes the result if $ch\rhd m$ does not occur in ψ during the time interval covered by α from time 0 on. Now we assume that $ch\rhd m$ does indeed occur during this time interval and observe:

$$\begin{array}{l}\langle \exists v \in \mathbb{N} : v = \min\{t' > 0 : m \in \pi_1(\psi).t'.ch\} \wedge (\psi, v-1) \in [\![\alpha]\!]_0^{v-1} : \\ \quad \langle \forall t \in \mathbb{N} :: \\ \qquad \langle \exists \varphi \in (\tilde{C} \times S)^\infty :: \\ \qquad\quad \varphi \in [\![\alpha \stackrel{ch\rhd m}{\rightarrow} \beta]\!] \wedge \psi{\downarrow}t = \varphi{\downarrow}t\rangle\rangle\rangle\end{array}$$

$\Rightarrow$ (* definition of $[\![\alpha \stackrel{ch\rhd m}{\rightarrow} \beta]\!]$ *)

$$\begin{array}{l}\langle \exists v \in \mathbb{N} : v = \min\{t' > 0 : m \in \pi_1(\psi).t'.ch\} \wedge (\psi, v-1) \in [\![\alpha]\!]_0^{v-1} : \\ \quad \langle \forall t \in \mathbb{N} :: \\ \qquad \langle \exists \varphi \in (\tilde{C} \times S)^\infty, \hat{t} \in \mathbb{N}_\infty, \hat{v} \in \mathbb{N} : \hat{v} = \min\{t'' > 0 : m \in \pi_1(\psi).t''.ch\} : \\ \qquad\quad (\varphi, \hat{v}-1) \in [\![\alpha]\!]_0^{\hat{v}-1} \wedge (\varphi, \hat{t}) \in [\![\beta]\!]_{\hat{v}} \wedge \psi{\downarrow}t = \varphi{\downarrow}t\rangle\rangle\rangle\end{array}$$

$\Rightarrow$ (* identify v and $\hat{v}$, consider $t > v$, predicate calculus *)

$\langle \exists v \in \mathbb{N} :: v = \min\{t' > 0 : m \in \pi_1(\psi).t'.ch\} \wedge (\psi, v-1) \in [\![\alpha]\!]_0^{v-1} \wedge \psi{\uparrow}v \in \mathit{prefc}.[\![\beta]\!]\rangle$

$\Rightarrow$ (* $[\![\beta]\!]$ is a safety property *)

$$\langle \exists v \in \mathbb{N} :: v = \min\{t' > 0 : m \in \pi_1(\psi).t'.ch\} \wedge (\psi, v-1) \in [\![\alpha]\!]_0^{v-1} \wedge \psi \uparrow v \in [\![\beta]\!] \rangle$$

$\Rightarrow$ (* definition of $[\![\alpha \stackrel{ch \triangleright m}{\rightarrow} \beta]\!]$ *)

$\psi \in [\![\alpha \stackrel{ch \triangleright m}{\rightarrow} \beta]\!]$

From this derivation the validity of

$$\psi \in prefc.[\![\alpha \stackrel{ch \triangleright m}{\rightarrow} \beta]\!] \;\Rightarrow\; \psi \in [\![\alpha \stackrel{ch \triangleright m}{\rightarrow} \beta]\!]$$

follows.

$q = [\![\alpha \Uparrow_{ch \triangleright m}]\!]$:

We want to show that $prefc.[\![\alpha \Uparrow_{ch \triangleright m}]\!] \subseteq [\![\alpha \Uparrow_{ch \triangleright m}]\!]$ holds. Due to the definition of $[\![\alpha \Uparrow_{ch \triangleright m}]\!]$ this amounts to proving the validity of

$$prefc.\langle \nu X :: \tau'_{pl}.X \rangle = \langle \nu X :: \tau'_{pl}.X \rangle$$

where we define $\tau'_{pl} : \mathcal{P}((\tilde{C} \times S)^\infty) \rightarrow \mathcal{P}((\tilde{C} \times S)^\infty)$ for all $\psi \in (\tilde{C} \times S)^\infty$ and $X \subseteq (\tilde{C} \times S)^\infty$ by

$$\psi \in \tau'_{pl}.X \stackrel{\text{def}}{=} \langle \exists t \in \mathbb{N}_\infty :: (\psi, t) \in \tau_{pl}.X \rangle$$

$\tau'_{pl}.X$ is the projection of $\tau_{pl}.X$ on the first coordinate of the latter's elements.

We follow the proof principle for fixpoints we have formulated in Proposition 35. As the predicate under consideration, we choose $P : \mathcal{P}((\tilde{C} \times S)^\infty) \rightarrow \mathbb{B}$ with

$$P.Q \stackrel{\text{def}}{=} prefc.Q = Q$$

According to Proposition 35 we have to discharge the following proof obligations:

(1) τ'_{pl} is monotonic

(2) P is admissible

(3) $P.(\tilde{C} \times S)^\infty$

(4) $\langle \forall X \subseteq (\tilde{C} \times S)^\infty :: P.X \Rightarrow P.(\tau'_{pl}.X) \rangle$

We deal with each of these proof obligations in turn.

τ'_{pl} is monotonic:

τ'_{pl} inherits its monotonicity directly from τ_{pl}.

<u>P is admissible</u>:

We have to show that $\langle \forall n \in \mathbb{N} :: P.X_n \rangle \Rightarrow P.(\bigcap_{n\in\mathbb{N}} X_n)$ holds for all infinite descending chains $(X_i)_{i\in\mathbb{N}}$ with $X_{i+1} \subseteq X_i$ for all $i \in \mathbb{N}$. Let $(X_i)_{i\in\mathbb{N}}$ be such a chain, $\psi \in (\tilde{C} \times S)^\infty$, and let $\langle \forall n \in \mathbb{N} :: P.X_n \rangle$ hold. We observe:

$$
\begin{array}{ll}
& \text{true} \\
\equiv & (*\ \text{set theory}\ *) \\
& \langle \forall k \in \mathbb{N} :: \bigcap_{n\in\mathbb{N}} X_n \subseteq X_k \rangle \\
\Rightarrow & (*\ \text{monotonicity of}\ \mathit{prefc}\ *) \\
& \langle \forall k \in \mathbb{N} :: \mathit{prefc}.(\bigcap_{n\in\mathbb{N}} X_n) \subseteq \mathit{prefc}.X_k \rangle \\
\Rightarrow & (*\ \langle \forall n \in \mathbb{N} :: P.X_n \rangle\ *) \\
& \langle \forall k \in \mathbb{N} :: \mathit{prefc}.(\bigcap_{n\in\mathbb{N}} X_n) \subseteq X_k \rangle \\
\Rightarrow & (*\ \text{set theory}\ *) \\
& \mathit{prefc}.(\bigcap_{n\in\mathbb{N}} X_n) \subseteq \bigcap_{n\in\mathbb{N}} X_n
\end{array}
$$

<u>$P.(\tilde{C} \times S)^\infty$</u>:

$$
\begin{array}{ll}
& P.(\tilde{C} \times S)^\infty \\
\equiv & (*\ \text{definition of}\ P\ *) \\
& \mathit{prefc}.(\tilde{C} \times S)^\infty = (\tilde{C} \times S)^\infty \\
\equiv & (*\ (\tilde{C} \times S)^\infty\ \text{is a trivial safety property}\ *) \\
& \text{true}
\end{array}
$$

<u>$\langle \forall X \subseteq (\tilde{C} \times S)^\infty :: P.X \Rightarrow P.(\tau'_{pl}.X) \rangle$</u>:

$$
\begin{array}{ll}
& \psi \in \mathit{prefc}.(\tau'_{pl}.X) \\
\equiv & (*\ \text{definition of}\ \mathit{prefc}\ *) \\
& \langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in (\tilde{C} \times S)^\infty :: \varphi \in \tau'_{pl}.X \wedge \psi{\downarrow}t = \varphi{\downarrow}t \rangle\rangle \\
\equiv & (*\ \text{definition of}\ \tau'_{pl}\ *) \\
& \langle \forall t \in \mathbb{N} :: \\
& \quad \langle \exists \varphi \in (\tilde{C} \times S)^\infty, t' \in \mathbb{N}_\infty :: \\
& \quad\quad (\quad ((\varphi, t') \in [\![\alpha]\!]_0 \wedge \langle \forall v \in [0, t'] :: m \notin \pi_1(\varphi).v.ch \rangle) \\
& \quad\quad\ \ \vee \langle \exists v \in \mathbb{N} :: \\
& \quad\quad\quad\quad\quad v = \min\{t'' \in \mathbb{N} : 0 < t'' \leq t' \wedge m \in \pi_1(\varphi).v.ch\} \\
& \quad\quad\quad\quad \wedge (\varphi, v-1) \in [\![\alpha]\!]_0^{v-1} \wedge (\varphi{\uparrow}v, t') \in X \rangle) \wedge \psi{\downarrow}t = \varphi{\downarrow}t \rangle\rangle
\end{array}
$$

$$
\begin{array}{ll}
\Rightarrow & \text{(* predicate calculus *)} \\
& \quad \langle \forall t \in \mathbb{N} :: \\
& \qquad \langle \exists \varphi \in (\tilde{C} \times S)^{\infty}, t' \in \mathbb{N}_{\infty} :: \\
& \qquad\quad (\varphi, t') \in [\![\alpha]\!]_0 \wedge \langle \forall v \in [0, t'] :: m \notin \pi_1(\varphi).v.ch \rangle \wedge \psi{\downarrow}t = \varphi{\downarrow}t \rangle\rangle \\
& \vee \ \langle \forall t \in \mathbb{N} :: \\
& \qquad \langle \exists \varphi \in (\tilde{C} \times S)^{\infty}, t' \in \mathbb{N}_{\infty}, v \in \mathbb{N} :: \\
& \qquad\qquad v = \min\{t'' \in \mathbb{N} : 0 < t'' \leq t' \wedge m \in \pi_1(\varphi).v.ch\} \\
& \qquad\quad \wedge (\varphi, v-1) \in [\![\alpha]\!]_0^{v-1} \wedge (\varphi{\uparrow}v, t') \in X \rangle \wedge \psi{\downarrow}t = \varphi{\downarrow}t \rangle \\
\Rightarrow & \text{(* definition of } \mathit{prefc}\text{, twice *)} \\
& \quad (\quad \psi \in \mathit{prefc}.[\![\alpha]\!] \\
& \qquad \wedge \langle \forall t \in \mathbb{N}_{\infty}, v \in \mathbb{N} : (\psi, t) \in [\![\alpha]\!]_0 \wedge m \in \pi_1(\psi).v.ch : v > t \rangle) \\
& \vee \ (\langle \exists v \in \mathbb{N} :: \\
& \qquad\quad v = \min\{t \in \mathbb{N} : m \in \pi_1(\psi).t.ch\} \\
& \qquad \wedge (\psi, v-1) \in [\![\alpha]\!]_0^{v-1} \wedge \psi{\uparrow}v \in \mathit{prefc}.X \rangle) \\
\equiv & \text{(* } [\![\alpha]\!] \text{ and } X \text{ are safety properties *)} \\
& \quad (\quad \psi \in [\![\alpha]\!] \\
& \qquad \wedge \langle \forall t \in \mathbb{N}_{\infty}, v \in \mathbb{N} : (\psi, t) \in [\![\alpha]\!]_0 \wedge m \in \pi_1(\psi).v.ch : v > t \rangle) \\
& \vee \ (\langle \exists v \in \mathbb{N} :: \\
& \qquad\quad v = \min\{t \in \mathbb{N} : m \in \pi_1(\psi).t.ch\} \\
& \qquad \wedge (\psi, v-1) \in [\![\alpha]\!]_0^{v-1} \wedge \psi{\uparrow}v \in X \rangle) \\
\equiv & \text{(* definition of } \tau'_{pl} \text{ *)} \\
& \psi \in \tau'_{pl}
\end{array}
$$

$q = [\![\rightarrow X]\!]$:

If $[\![\alpha]\!]$ is a safety property and $(X, \alpha) \in \mathit{MSCR}$ holds, then $[\![\rightarrow X]\!]$ is also a safety property, because we have $[\![\rightarrow X]\!] = [\![\alpha]\!]$ by definition of $[\![\rightarrow X]\!]$ under these circumstances.

■

Liveness Preservers

The following proposition is the dual to the preceding one, and characterizes the MSC operators that preserve the liveness of their operands.

Proposition 54 (Liveness preserving operators) *Let $\alpha, \beta \in \langle MSC \rangle$, $m, n \in \mathbb{N}_{\infty}$, $\lt L\gt \in \{\lt *\gt, \lt m\gt, \lt m,n\gt\}$, $ch \triangleright m \in \langle MSG \rangle$, and X $\in \langle MSCNAME \rangle$ with $(X, \alpha) \in$ MSCR. Furthermore, let $[\![\alpha]\!]$ and $[\![\beta]\!]$ be liveness properties. Then each of the following is*

a liveness property:

$$[\![\alpha\ ;\ \beta]\!]$$
$$[\![\alpha \mid \beta]\!]$$
$$[\![\alpha\ \sim\ \beta]\!]$$
$$[\![\alpha\uparrow_{<L>}]\!]$$
$$[\![\alpha \stackrel{ch\triangleright m}{\rightarrow} \beta]\!]$$
$$[\![\alpha\Uparrow_{ch\triangleright m}]\!]$$
$$[\![\rightarrow X]\!]$$

□

Proof Let the assumptions be as stated in the proposition. For every property $q \in \{[\![\alpha\ ;\ \beta]\!], [\![\alpha \mid \beta]\!], [\![\alpha\ \sim\ \beta]\!], [\![\alpha\uparrow_{<L>}]\!], [\![\alpha \stackrel{ch\triangleright m}{\rightarrow} \beta]\!], [\![\alpha\Uparrow_{ch\triangleright m}]\!], [\![\rightarrow X]\!]\}$ we show that $(\tilde{C} \times S)^\infty \subseteq \mathit{prefc}.q$ holds.

$q = [\![\alpha\ ;\ \beta]\!]$:

We begin by observing that for all $\psi \in (\tilde{C} \times S)^\infty$ and $\alpha, \beta \in \langle\text{MSC}\rangle$

$$(\psi, \infty) \in [\![\alpha]\!]_0 \ \Rightarrow\ (\psi, \infty) \in [\![\alpha\ ;\ \beta]\!]_0$$

holds. This is a trivial consequence of the definition of sequential composition and Definition 1. Thus, in the following we can concentrate on the subset of $[\![\alpha]\!]$ generated by executions with finite time bounds in $[\![\alpha]\!]_0$. Therefore, we assume that ψ is such an element, i.e. we have:

$$\langle \exists t \in \mathbb{N} :: (\psi, t) \in [\![\alpha]\!]_0 \rangle \qquad (\star\star)$$

Based on this assumption we derive

$\psi \in \mathit{prefc}.[\![\alpha\ ;\ \beta]\!]$

$\equiv$ (∗ definition of *prefc* ∗)

$\langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in (\tilde{C} \times S)^\infty :: \varphi \in [\![\alpha\ ;\ \beta]\!] \wedge \psi\!\downarrow\! t = \varphi\!\downarrow\! t \rangle\rangle$

$\Leftarrow$ (∗ definition of $[\![\alpha\ ;\ \beta]\!]$ ∗)

$\langle \forall t \in \mathbb{N} ::$
$\quad \langle \exists \varphi \in (\tilde{C} \times S)^\infty, t' \in \mathbb{N}, t'' \in \mathbb{N}_\infty ::$
$\quad\quad (\varphi, t') \in [\![\alpha]\!]_0 \wedge (\varphi, t'') \in [\![\beta]\!]_{t'} \wedge \psi\!\downarrow\! t = \varphi\!\downarrow\! t \rangle\rangle$

$\Leftarrow$ (∗ choose $\varphi = (\varphi_\alpha\!\downarrow\! t') \frown (\hat{\varphi}_\beta\!\uparrow\! t')$ and $\hat{t} = t'$, predicate calculus ∗)

$\langle \forall t \in \mathbb{N} :: \langle \exists \varphi_\alpha \in (\tilde{C} \times S)^\infty, t' \in \mathbb{N} :: (\varphi_\alpha, t') \in [\![\alpha]\!]_0 \wedge \psi\!\downarrow\! t = \varphi_\alpha\!\downarrow\! t \rangle\rangle$
$\wedge \langle \forall \hat{t} \in \mathbb{N} ::$
$\quad \langle \forall t \in \mathbb{N} :: \langle \exists \hat{\varphi}_\beta \in (\tilde{C} \times S)^\infty, \hat{t}' \in \mathbb{N}_\infty :: (\hat{\varphi}_\beta, \hat{t}') \in [\![\beta]\!]_{\hat{t}} \wedge \psi\!\downarrow\! t = \hat{\varphi}_\beta\!\downarrow\! t \rangle\rangle$

$\Leftarrow$ (* index shift, choose $\hat{\varphi}_\beta = (\psi \downarrow t) \frown \varphi_\beta$, and $\hat{t}' = \hat{t} + t'$ *)

$$\langle \forall t \in \mathbb{N} :: \langle \exists \varphi_\alpha \in (\tilde{C} \times S)^\infty, t' \in \mathbb{N} :: (\varphi_\alpha, t') \in [\![\alpha]\!]_0 \wedge \psi \downarrow t = \varphi_\alpha \downarrow t \rangle \rangle$$
$$\wedge \langle \forall \hat{t} \in \mathbb{N} :: \langle \forall t \in \mathbb{N} :: \langle \exists \varphi_\beta \in (\tilde{C} \times S)^\infty, t' \in \mathbb{N}_\infty :: (\varphi_\beta, t') \in [\![\beta]\!]_0 \wedge (\psi \uparrow \hat{t}) \downarrow t = \varphi_\beta \downarrow t \rangle \rangle \rangle$$

$\Leftarrow$ (* definition of *prefc*, Assumption ($\star\star$) *)

$\psi \in \mathit{prefc}.[\![\alpha]\!] \wedge \langle \forall \hat{t} \in \mathbb{N} :: \psi \uparrow \hat{t} \in \mathit{prefc}.[\![\beta]\!] \rangle$

$\Leftarrow$ (* $[\![\alpha]\!]$ and $[\![\beta]\!]$ are liveness properties *)

true

<u>$q = [\![\alpha \mid \beta]\!]$</u>:

$\psi \in \mathit{prefc}.[\![\alpha \mid \beta]\!]$

$\equiv$ (* definitions of *prefc* and $[\![\alpha \mid \beta]\!]$ *)

$\langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in (\tilde{C} \times S)^\infty, t' \in \mathbb{N}_\infty :: (\varphi, t') \in [\![\alpha \mid \beta]\!]_0 \wedge \psi \downarrow t = \varphi \downarrow t \rangle \rangle$

$\equiv$ (* definition of $[\![\alpha \mid \beta]\!]_0$ *)

$$\langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in (\tilde{C} \times S)^\infty, t' \in \mathbb{N}_\infty :: ((\varphi, t') \in [\![\alpha]\!]_0 \vee ((\varphi, t') \in [\![\beta]\!]_0) \wedge \psi \downarrow t = \varphi \downarrow t \rangle \rangle$$

$\Leftarrow$ (* predicate calculus *)

$$\langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in (\tilde{C} \times S)^\infty, t' \in \mathbb{N}_\infty :: (\varphi, t') \in [\![\alpha]\!]_0 \wedge \psi \downarrow t = \varphi \downarrow t \rangle \rangle$$
$$\vee \quad \langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in (\tilde{C} \times S)^\infty, t' \in \mathbb{N}_\infty :: (\varphi, t') \in [\![\beta]\!]_0 \wedge \psi \downarrow t = \varphi \downarrow t \rangle \rangle$$

$\equiv$ (* definition of *prefc*, twice *)

$\psi \in \mathit{prefc}.[\![\alpha]\!] \vee \psi \in \mathit{prefc}.[\![\beta]\!]$

$\Leftarrow$ (* $[\![\alpha]\!]$ and $[\![\beta]\!]$ are liveness properties *)

true

<u>$q = [\![\alpha \sim \beta]\!]$</u>:

Observe that $[\![\alpha\,;\,\beta]\!] \subseteq [\![\alpha \sim \beta]\!]$ holds (recall that $\alpha\,;\,\beta$ is a property refinement of $\alpha \sim \beta$). Thus, we derive:

$\psi \in \mathit{prefc}.[\![\alpha \sim \beta]\!]$

$\Leftarrow$ (* above observation, monotonicity of *prefc* *)

$\psi \in \mathit{prefc}.[\![\alpha\,;\,\beta]\!]$

$\Leftarrow$ (* see proof above, $[\![\alpha]\!]$ and $[\![\beta]\!]$ are liveness properties *)

true

$\underline{q \in \{[\![\alpha\uparrow_{<m,n>}]\!], [\![\alpha\uparrow_{<*>}]\!]\}}$:

We start with deriving the result for $q = [\![\alpha^n]\!]$, $n \in \mathbb{N}$ by induction on n:

$\underline{n = 0}$:

$$\begin{aligned} & \psi \in \mathit{prefc}.[\![\alpha^0]\!] \\ \equiv \quad & (*\ \text{definition of } \alpha^0\ *) \\ & \psi \in \mathit{prefc}.[\![\mathbf{empty}]\!] \\ \Leftarrow \quad & (*\ [\![\mathbf{empty}\ ]\!]\ \text{ is a liveness property } *) \\ & \text{true} \end{aligned}$$

$\underline{n \rightsquigarrow (n+1)}$:

$$\begin{aligned} & \psi \in \mathit{prefc}.[\![\alpha^{n+1}]\!] \\ \equiv \quad & (*\ \text{definition of } \alpha^{n+1}\ *) \\ & \psi \in \mathit{prefc}.[\![\alpha \ ; \ \alpha^n]\!] \\ \Leftarrow \quad & (*\ [\![\alpha]\!] \text{ and } [\![\alpha^n]\!] \text{ are liveness properties, } ; \text{ preserves liveness } *) \\ & \text{true} \end{aligned}$$

The result for arbitrary finite repetition ranges follows from the defining equations for the repetition operator, which reduce their argument to either **empty** or the term $\alpha^m \ ; \ \alpha^{n-m}$; moreover, $[\![\mathbf{empty}]\!]$ and $[\![\alpha^k]\!]$ are liveness properties for arbitrary $k \in \mathbb{N}$ (see above), and sequential composition preserves liveness.

$\underline{q = [\![\alpha\uparrow_{<\infty>}]\!]}$:

We sketch the outline of a proof strategy. We have to show that the infinite repetition of an MSC that defines a liveness property also yields a liveness property. The basic idea we follow is to reduce this problem to well-known liveness properties of temporal logic.

As a starting point we consider the special case $\alpha \overset{\mathsf{def}}{=} ch \triangleright m$. Below, we will see that message occurrence *defines* a liveness property (cf. Proposition 56), i.e. $(\tilde{C} \times S)^\infty \subseteq \mathit{prefc}.[\![ch \triangleright m]\!]$ holds.

Our first step is to observe the validity of the following equation:

$$ch \triangleright m \ ; \ (ch \triangleright m\uparrow_{<\infty>}) \ \equiv_u \ ch \triangleright m\uparrow_{<\infty>}$$

Its validity follows directly from the reduction lemma for greatest fixpoints as stated, for instance, in [Win93] (p. 237). By induction we obtain further the validity of

$$ch{\triangleright}m{\uparrow}_{<n>} \,;\, (ch{\triangleright}m{\uparrow}_{<\infty>}) \;\equiv_u\; ch{\triangleright}m{\uparrow}_{<\infty>}$$

for any $n \in \mathbb{N}$.

Recall from Proposition 2 that we have $(\varphi, t) \in [\![\beta]\!]_u \;\Rightarrow\; t \geq u$. In particular, for **empty** $\not\leq_p \beta$ we have $(\varphi, t) \in [\![\beta]\!]_u \;\Rightarrow\; t > u$ (**empty** is the only MSC term without "time progress"). This implies that $(\varphi, t) \in [\![ch{\triangleright}m]\!]_0 \;\Rightarrow\; t > 0$ holds in the special case we consider. By induction, we obtain the validity of $(\varphi, t) \in [\![ch{\triangleright}m{\uparrow}_{<n>}]\!]_0 \;\Rightarrow\; t > n-1$ for arbitrary $n \in \mathbb{N}$. The semantics of $ch{\triangleright}m{\uparrow}_{<n>}$ equals the semantics of the n-fold sequential composition of occurrences of $ch{\triangleright}m$. In particular, any φ with $(\varphi, t) \in [\![ch{\triangleright}m{\uparrow}_{<n>}]\!]_0$ contains at least n occurrences of $ch{\triangleright}m$ at n distinct time points.

Equipped with these preliminaries we observe for any $n \in \mathbb{N}$

$$\begin{array}{ll}
 & (\varphi, t) \in [\![ch{\triangleright}m{\uparrow}_{<\infty>}]\!] \\
\equiv & \text{(* see above *)} \\
 & ch{\triangleright}m{\uparrow}_{<n>} \,;\, (ch{\triangleright}m{\uparrow}_{<\infty>}) \\
\equiv & \text{(* definition of } [\![. \,;\, .]\!] \text{ *)} \\
 & \langle \exists t' \in \mathbb{N}_\infty :: (\varphi, t') \in [\![ch{\triangleright}m{\uparrow}_{<n>}]\!]_0 \;\wedge\; (\varphi, t) \in [\![ch{\triangleright}m{\uparrow}_{<\infty>}]\!]_{t'} \rangle \\
\Rightarrow & \text{(* above observations *)} \\
 & t > n \wedge |\{t' \in \mathbb{N} : m \in \pi_1(\varphi).t'.ch\}| \geq n
\end{array}$$

Because $n \in \mathbb{N}$ is arbitrary, this implies that

$$(\varphi, t) \in [\![ch{\triangleright}m{\uparrow}_{<\infty>}]\!]_0 \;\Rightarrow\; t = \infty \wedge |\{t' \in \mathbb{N} : m \in \pi_1(\varphi).t'.ch\}| = \infty \qquad (\star')$$

holds. Hence, elements of $[\![ch{\triangleright}m{\uparrow}_{<\infty>}]\!]$ have the form (φ, ∞) such that φ contains an infinite number of occurrences of $ch{\triangleright}m$.

It is a well-known fact from temporal logic that properties requiring an infinite number of occurrences of a certain condition are liveness properties (cf., for instance, [Eme90, CMP91]). To establish a closer relationship between our notion of properties and the one of temporal logic we could encode the condition "$ch{\triangleright}m$ has occurred" by means of an appropriate state (predicate) p; then the implication $(\star')$ reduces to $\mathbf{GF}p$ in linear time temporal logic (cf. [Eme90]).

This concludes the proof outline for the special case $\alpha = ch{\triangleright}m$. For arbitrary α such that $[\![\alpha]\!]$ is a liveness property the reduction to temporal logic proceeds along the same lines (and by induction on α's structure). We omit the full details of the proof for reasons of brevity.

$q = [\![\alpha \overset{ch \rhd m}{\rightarrow} \beta]\!]$:

Let $\psi \in (\tilde{C} \times S)^{\infty}$ and $t \in \mathbb{N}$ be arbitrary but fixed. We show that there is a $\varphi \in (\tilde{C} \times S)^{\infty}$ with $\psi \downarrow t = \varphi \downarrow t$ and $\varphi \in [\![\alpha \overset{ch \rhd m}{\rightarrow} \beta]\!]$.

We distinguish the following three cases:

(1) α's contribution has already occurred in $\psi \downarrow t$, but the preemptive message $ch \rhd m$ has *not* occurred until α's contribution is finished.

(2) α's contribution has occurred until time t, and the preemptive message $ch \rhd m$ has occurred before α's contribution was finished.

(3) α's contribution has not (completely) occurred in ψ until time t.

For each of these cases we describe how to construct an element $\varphi \in (\tilde{C} \times S)^{\infty}$ with $\psi \downarrow t = \varphi \downarrow t$ and $\varphi \in [\![\alpha \overset{ch \rhd m}{\rightarrow} \beta]\!]$.

case (1):

We assume there is a time $t' \in [0, t]$ such that

$$(\psi, t') \in [\![\alpha]\!]_0 \;\wedge\; \langle \forall v \in [0, t'] :: m \notin \pi_1(\psi).v.ch \rangle$$

Inspection of the preemption operator's definition immediately shows that, under these circumstances, we can set $\varphi \overset{\text{def}}{=} \psi$ to obtain $(\varphi, t') \in [\![\alpha \overset{ch \rhd m}{\rightarrow} \beta]\!]_0$; this, in turn, implies $\varphi \in [\![\alpha \overset{ch \rhd m}{\rightarrow} \beta]\!]$.

case (2):

Now we assume there is a time $t' \in [0, t]$ such that $(\psi, t') \in [\![\alpha]\!]_0$ holds. Furthermore, we assume the existence of a time $v \in \mathbb{N}$ with

$$v = \min\{v' \in [1, t'] : m \in \pi_1(\psi).v'.ch\}$$

In this constellation we have $(\psi, v-1) \in [\![\alpha]\!]_0^{v-1}$. Thus, to find a φ that satisfies the preemption operator's definition in this case, we only have to append the preemption handling according to β to $\psi \downarrow t$. More precisely, we select an element $\hat{\varphi} \in [\![\beta]\!]$, and construct φ as follows: $\varphi \overset{\text{def}}{=} \psi \downarrow t \frown \hat{\varphi}$. For this φ we observe the validity of all of the following:

- $v = \min\{v' \in [1, t'] : m \in \pi_1(\varphi).v'.ch\}$
- $(\varphi, v-1) \in [\![\alpha]\!]_0^{v-1}$
- $\langle \exists t' \in \mathbb{N}_\infty :: (\varphi, t') \in [\![\beta]\!]_v \rangle$

From this we immediately obtain that $\varphi \in [\![\alpha \stackrel{ch \triangleright m}{\rightarrow} \beta]\!]$ holds.

case (3):

Assume there is no $t' \in [0,t]$ such that $(\psi, t') \in [\![\alpha]\!]_0$ holds. Yet, because α is a liveness property, we know that there is a $t_\alpha \in \mathbb{N}_\infty$ with $(\psi, t_\alpha) \in [\![\alpha]\!]_0$.

This time, we construct φ by "forcing" occurrence of preemption and preemption handling in the extension of $\psi \downarrow t$. More precisely, we set $\varphi \stackrel{\mathsf{def}}{=} \psi \downarrow t \frown \kappa$ where κ is specified as follows:

$$\kappa \uparrow 1 \in [\![\beta]\!] \ \wedge \ \texttt{<}m\texttt{>} = \pi_1(\kappa).0.ch \ \wedge \ \langle \forall c \neq ch :: \pi_1(\kappa).0.c = \texttt{<>} \rangle$$

Then there is a time $v \in [1, t+1]$ such that all of the following hold:

- $v = \min\{v' > 0 : m \in \pi_1(\varphi).v'.ch\}$
- $(\varphi, v-1) \in [\![\alpha]\!]_0^{v-1}$
- $\langle \exists t' \in \mathbb{N}_\infty :: (\varphi, t') \in [\![\beta]\!]_v \rangle$

and thus we again obtain $\varphi \in [\![\alpha \stackrel{ch \triangleright m}{\rightarrow} \beta]\!]$ by the preemption operator's definition.

In summary, we obtain

$$\langle \forall \psi \in (\tilde{C} \times S)^\infty, t \in \mathbb{N} :: \langle \exists \varphi \in [\![\alpha \stackrel{ch \triangleright m}{\rightarrow} \beta]\!] :: \psi \downarrow t = \varphi \downarrow t \rangle \rangle$$

which shows that $[\![\alpha \stackrel{ch \triangleright m}{\rightarrow} \beta]\!]$ indeed is a liveness property.

$q = [\![\rightarrow X]\!]$:

If $[\![\alpha]\!]$ is a liveness property and $(X, \alpha) \in \mathit{MSCR}$ holds, then $[\![\rightarrow X]\!]$ is also a liveness property, because we have $[\![\rightarrow X]\!] = [\![\alpha]\!]$ by definition of $[\![\rightarrow X]\!]$ under these circumstances.
■

Hybrids

Operators involving guards, as well as the join operator yield combinations of safety and liveness properties; the following proposition makes this statement more precise.

Proposition 55 (Hybrid properties) *Let $\alpha, \beta \in \langle MSC \rangle$ be such that both $[\![\alpha]\!]$ and $[\![\beta]\!]$ are nontrivial liveness properties, and $p \in \langle GUARD \rangle$. Then, in general, each of the following is neither a pure safety nor a pure liveness property:*

$$[\![p : \alpha]\!]$$
$$[\![\alpha \uparrow_{\texttt{<}p\texttt{>}}]\!]$$
$$[\![\alpha \otimes \beta]\!]$$
□

PROOF For $q \in \{[\![p : \alpha]\!], [\![\alpha\uparrow_{<p>}]\!], [\![\alpha \otimes \beta]\!]\}$, such that the assumptions in the proposition hold, we show that there are nontrivial safety and liveness properties $q_S, q_L \subseteq (\tilde{C} \times S)^\infty$ with $q = q_S \cap q_L$.

$q = [\![p : \alpha]\!]$:

$[\![p : \alpha]\!]$ is the intersection of the nontrivial safety property $q_0 \stackrel{\text{def}}{=} \{\varphi \in (\tilde{C} \times S)^\infty : \pi_2(\varphi).0 \in [\![p]\!]\}$, and the nontrivial liveness property $[\![\alpha]\!]$:

$$[\![p : \alpha]\!] = q_0 \cap [\![\alpha]\!]$$

To see that q_0 is, indeed, a safety property, we observe:

$$\begin{array}{ll} & \psi \in \textit{prefc}.q_0 \\ \equiv & (\ast \text{ definition of } \textit{prefc} \ast) \\ & \langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in (\tilde{C} \times S)^\infty :: \varphi \in q_0 \wedge \psi \downarrow t = \varphi \downarrow t \rangle \rangle \\ \equiv & (\ast \text{ definition of } q_0 \ast) \\ & \langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in (\tilde{C} \times S)^\infty :: \pi_2(\varphi).0 \in [\![p]\!] \wedge \psi \downarrow t = \varphi \downarrow t \rangle \rangle \\ \Rightarrow & (\ast \text{ select } t = 0 \ast) \\ & \pi_2(\psi).0 \in [\![p]\!] \\ \equiv & (\ast \text{ definition of } q_0 \ast) \\ & \psi \in q_0 \end{array}$$

$q = [\![\alpha\uparrow_{<p>}]\!]$:

We observe the result for the special case $\alpha \stackrel{\text{def}}{=} ch \triangleright m$. To this end, we consider a predicate p that holds precisely if at least $n \in \mathbb{N}$ copies of message $ch \triangleright m \in \langle \text{MSG} \rangle$ have occurred, and examine the property defined by the MSC term $ch \triangleright m\uparrow_{<p>}$. We observe the validity of

$$[\![ch \triangleright m\uparrow_{<p>}]\!] = q_1 \cap q_2$$

where we define q_1 and q_2 as follows:

$$\begin{array}{lll} q_1 & \stackrel{\text{def}}{=} & \{\psi \in (\tilde{C} \times S)^\infty : \langle \forall t \in \mathbb{N} :: \psi_2(\psi).t \in [\![p]\!] \;\equiv\; \#(\{m\} \copyright \pi_1(\psi).ch) \geq n \rangle\} \\ q_2 & \stackrel{\text{def}}{=} & \{\psi \in [\![ch \triangleright m\uparrow_{<n>}]\!]\} \end{array}$$

Clearly, q_1 is a nontrivial safety property, and q_2 is a nontrivial liveness property. Even for $p \equiv$ true and certain $\alpha \in \langle \text{MSC} \rangle$ the property $[\![\alpha\uparrow_{<p>}]\!]$, which equals $[\![\alpha\uparrow_{<\infty>}]\!]$, is neither a pure safety, nor a pure liveness property. To see this, we consider the predicate p' that holds precisely if at most $n \in \mathbb{N}$ copies of the message $ch \triangleright m$ have occurred. For this choice

of p' we study the property defined by the MSC $((p' : ch \triangleright m) \mid ch' \triangleright m')\uparrow_{<\infty>}$. Clearly, we have

$$[\![((p' : ch \triangleright m) \mid ch' \triangleright m')\uparrow_{<\infty>}]\!] = q_3 \cap q_4$$

where we define q_3 and q_4 as follows:

$$q_3 \stackrel{\text{def}}{=} \{\psi \in (\tilde{C} \times S)^\infty : \langle \forall t \in \mathbb{N} :: \#(\{m\} \copyright \pi_1(\psi).ch) \leq n \rangle\}$$
$$q_4 \stackrel{\text{def}}{=} \{\psi \in [\![(ch \triangleright m \mid ch' \triangleright m')\uparrow_{<\infty>}]\!]\}$$

q_3 is a nontrivial safety property, and q_4 is a nontrivial liveness property. Thus,

$$[\![((p' : ch \triangleright m) \mid ch' \triangleright m')\uparrow_{<\infty>}]\!]$$

is a hybrid property.

<u>$q = [\![\alpha \otimes \beta]\!]$:</u>

Here, the safety part of the join semantics comes from the requirement that no redundant message may occur during the time interval covered by α and β. Thus, we obtain the validity of

$$[\![\alpha \otimes \beta]\!] = q_5 \cap q_6$$

where we define q_5 and q_6 as :

$$\begin{aligned} q_5 \stackrel{\text{def}}{=} \{\psi \in (\tilde{C} \times S)^\infty :\ & \\ & \langle \exists t \in \mathbb{N}_\infty : (\psi, t) \in [\![\alpha]\!]_0 \vee (\psi, t) \in [\![\beta]\!]_0 : \\ & \quad \langle \forall X \in (msgs.\alpha \cap msgs.\beta)^*, \hat{\psi} \in (\tilde{C} \times S)^\infty, ch \in C, t' \in [0, t] \cap \mathbb{N} :: \\ & \qquad ((X \neq <>) \wedge (\pi_1(\hat{\psi}).t'.ch = \pi_1(\psi).t'.ch \setminus X)) \\ & \qquad \Rightarrow \langle \forall t'' \in \mathbb{N} :: (\hat{\psi}, t'') \notin [\![\alpha]\!]_0 \wedge (\hat{\psi}, t'') \notin [\![\beta]\!]_0 \rangle\rangle \end{aligned}$$

and

$$q_6 \stackrel{\text{def}}{=} \{\psi \in (\tilde{C} \times S)^\infty : \langle \exists t_1, t_2 \in \mathbb{N}_\infty :: (\psi, t_1) \in [\![\alpha]\!]_0 \wedge (\psi, t_2) \in [\![\beta]\!]_0 \rangle\}$$

To see that q_5 is a safety property we assume that $\psi \notin q_5$ holds. Because both $[\![\alpha]\!]$ and $[\![\beta]\!]$ are liveness properties this can only happen if ψ is redundant in a message in $msgs.\alpha \cap msgs.\beta$ at some finite time point. No extension whatsoever of ψ beyond this time point can remedy this.

q_6 is a liveness property due to the following derivation:

$\psi \in prefc.q_6$

$\equiv$ (* definition of $prefc$ *)

$$
\begin{array}{ll}
& \langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in (\tilde{C} \times S)^\infty :: \varphi \in q_6 \wedge \psi \downarrow t = \varphi \downarrow t \rangle\rangle \\
\equiv & (* \text{ definition of } q_6 *) \\
& \langle \forall t \in \mathbb{N} :: \\
& \quad \langle \exists \varphi \in (\tilde{C} \times S)^\infty, t', t'' \in \mathbb{N}_\infty :: \\
& \qquad (\varphi, t') \in [\![\alpha]\!]_0 \wedge (\varphi, t'') \in [\![\beta]\!]_0 \wedge \psi \downarrow t = \varphi \downarrow t \rangle\rangle \\
\Leftarrow & (* \text{ predicate calculus } *) \\
& \quad\;\; \langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in (\tilde{C} \times S)^\infty, t' \in \mathbb{N}_\infty :: (\varphi, t') \in [\![\alpha]\!]_0 \wedge \psi \downarrow t = \varphi \downarrow t \rangle\rangle \\
& \wedge \;\; \langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in (\tilde{C} \times S)^\infty, t'' \in \mathbb{N}_\infty :: (\varphi, t'') \in [\![\beta]\!]_0 \wedge \psi \downarrow t = \varphi \downarrow t \rangle\rangle \\
\Leftarrow & (* \text{ definitions of } \mathit{prefc}, [\![\alpha]\!] \text{ and } [\![\beta]\!] *) \\
& \psi \in \mathit{prefc}.[\![\alpha]\!] \wedge \psi \in \mathit{prefc}.[\![\beta]\!] \\
\Leftarrow & (* [\![\alpha]\!] \text{ and } [\![\beta]\!] \text{ are liveness properties } *) \\
& \text{true}
\end{array}
$$

■

Liveness Generators

The following proposition indicates that message occurrence and trigger composition generate liveness properties.

Proposition 56 (Liveness generators) *Let $\alpha, \beta \in \langle MSC \rangle$, and $ch \triangleright m \in \langle MSG \rangle$. Furthermore, let $[\![\alpha]\!]$ and $[\![\beta]\!]$ be arbitrary nontrivial properties. Then each of the following is a liveness property:*

$$
\begin{array}{c}
[\![ch \triangleright m]\!] \\
[\![\alpha \;\mapsto\; \beta]\!]
\end{array}
$$

□

Proof For every $q \in \{[\![ch \triangleright m]\!], [\![\alpha \;\mapsto\; \beta]\!]\}$, and arbitrary $\alpha, \beta \in \langle \text{MSC} \rangle$ such that $[\![\alpha]\!]$ and $[\![\beta]\!]$ are nontrivial we show that $(\tilde{C} \times S)^\infty \subseteq \mathit{prefc}.q$ holds.

$\underline{q = [\![ch \triangleright m]\!]}$:

$$
\begin{array}{ll}
& \psi \in \mathit{prefc}.[\![ch \triangleright m]\!] \\
\equiv & (* \text{ definitions of } \mathit{prefc} \text{ and } [\![ch \triangleright m]\!] *) \\
& \langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in (\tilde{C} \times S)^\infty, t' \in \mathbb{N}_\infty :: (\varphi, t') \in [\![ch \triangleright m]\!]_0 \wedge \psi \downarrow t = \varphi \downarrow t \rangle\rangle \\
\equiv & (* \text{ definition of } [\![ch \triangleright m]\!]_0 *) \\
& \langle \forall t \in \mathbb{N} :: \\
& \quad \langle \exists \varphi \in (\tilde{C} \times S)^\infty, t' \in \mathbb{N}_\infty :: \\
& \qquad t' = \min\{v : v > 0 \wedge m \in \pi_1(\varphi).v.ch\} \wedge \psi \downarrow t = \varphi \downarrow t \rangle\rangle
\end{array}
$$

$\Leftarrow$ (* select $\varphi = \psi\downarrow t \frown \kappa$ with $\kappa \in (\tilde{C} \times S)^\infty$ such that $m \in \pi_1(\kappa).1.ch$ holds *)

true

<u>$q = [\![\alpha \mapsto \beta]\!]$:</u>

$\psi \in prefc.[\![\alpha \mapsto \beta]\!]$

$\equiv$ (* definition of *prefc* and $[\![\alpha \mapsto \beta]\!]$ *)

$\langle \forall t \in \mathbb{N} :: \langle \exists \varphi \in (\tilde{C} \times S)^\infty, \hat{t} \in \mathbb{N}_\infty :: (\varphi, \hat{t}) \in [\![\alpha \mapsto \beta]\!]_0 \land \psi\downarrow t = \varphi\downarrow t \rangle\rangle$

$\equiv$ (* definition of $[\![\alpha \mapsto \beta]\!]_0$ *)

$$\begin{array}{l} \langle \forall t \in \mathbb{N} :: \\ \quad \langle \exists \varphi \in (\tilde{C} \times S)^\infty, \hat{t} \in \mathbb{N}_\infty :: \\ \quad\quad \langle \forall t', t'' : \infty > t'' \geq t' \geq 0 : \\ \quad\quad\quad (\varphi, t'') \in [\![\alpha]\!]_{t'} \Rightarrow \langle \exists t''' : \infty > t''' > t'' : (\varphi, \hat{t}) \in [\![\beta]\!]_{t'''} \rangle\rangle \land \psi\downarrow t = \varphi\downarrow t \rangle\rangle \end{array}$$

$\Leftarrow$ (* set $\varphi = (\psi\downarrow t) \frown \kappa$ for any $(\kappa, \hat{t}) \in [\![\beta]\!]_{t+1}$ *)

true

■

B.4. From MSCs to Component Specifications

Chapter 7 contains two major propositions. The first asserts that MSC specifications are time guarded. The second gives a constructive criterion for determining the join-consistency of normalized MSCs. We prove these results in Section B.4.1 and B.4.2, respectively.

B.4.1. Time Guardedness Of MSCs

Proposition 57 (Time Guardedness of MSCs) *Let $\alpha, \beta \in \langle MSC\rangle$ be MSC terms, and $I, O \subseteq C$ sets of channels such that (I, O) is directed, and $R_\alpha : \overrightarrow{(I \cup O)} \rightarrow \mathbb{B}$ and $R_\beta : \overrightarrow{(I \cup O)} \rightarrow \mathbb{B}$ are time guarded. Furthermore, let $p \in \langle GUARD\rangle$, $ch{\triangleright}m \in \langle MSG\rangle$, and $<L> \in \{<p>, <n>, <m,n>\}$ hold for $m, n \in \mathbb{N}_\infty$. Then each of the following interaction interfaces (of the same signature as R_α and R_β) is also time guarded: $R_{\mathbf{empty}}$, $R_{\mathbf{any}}$, $R_{ch\triangleright m}$, $R_{\alpha\,;\,\beta}$, $R_{p:\alpha}$, $R_{\alpha\,|\,\beta}$, $R_{\alpha\,\sim\,\beta}$, $R_{\alpha\,\otimes\,\beta}$, $R_{\alpha\uparrow<L>}$, $R_{\alpha \xrightarrow{ch\triangleright m} \beta}$, $R_{\alpha\Uparrow ch\triangleright m}$, $R_{\alpha\,\mapsto\,\beta}$.* □

Proof We give an extended proof sketch. $R_{\mathbf{empty}}$ and $R_{\mathbf{any}}$ are trivially time guarded, because **empty** and **any** contain no interaction whatsoever under the closed world semantics. $R_{ch\triangleright m}$ is time guarded, because each element of $[\![ch{\triangleright}m]\!]_{0,CW}$ contains precisely one message occurrence with a delay of at least one time unit:

$$(\varphi, t) \in [\![ch{\triangleright}m]\!]_0 \Rightarrow \langle \exists t \in \mathbb{N} :: t = \min\{t' \in \mathbb{N} : t' > 0 \wedge \pi_1(\varphi).t'.ch\}\rangle$$

The delay becomes important in connection with sequential composition. Consider an element of the semantics of the sequential composition of α and β: $(\varphi, t) \in [\![\alpha\,;\,\beta]\!]_{0,CW}$. We know that there is a $t' \in \mathbb{N}_\infty$ such that both $(\varphi, t') \in [\![\alpha]\!]_0$ and $(\varphi, t) \in [\![\beta]\!]_{t'}$ hold. Recall that for $\alpha \not\equiv_u$ **empty** and $\beta \not\equiv_u$ **empty** we have $t' > 0$ and $t > t'$. Moreover, the first message contributed by β to φ occurs at least one time unit later than the last message α has contributed, due to the above observation for message occurrence. This explains the time guardedness of sequential composition. For the alternative operator we observe $R_{\alpha\,|\,\beta}.x \equiv R_\alpha.x \vee R_\beta.x$ and the disjunction of two time guarded interaction interfaces is time guarded. For guarded MSCs and the join of two MSCs the following two implications hold: $R_{p:\alpha}.x \Rightarrow R_\alpha.x$, $R_{\alpha\,\otimes\,\beta}.x \Rightarrow (R_\alpha.x \wedge R_\beta.x)$; because R_α and R_β are both time guarded, and conjunction preserves time guardedness, we obtain that also guarded MSCs and the join of two MSCs are time guarded. $R_{\alpha\,\sim\,\beta}$ is time guarded, because there is *no* mandatory order between messages contributed individually by α and β, but the *required* order (induced by the time guardedness of R_α and R_β) is maintained; the reason is that we can extract α's and β's contributions from the semantics of $\alpha \sim \beta$ independently. The loop constructs all reduce to sequential composition; therefore $R_{\alpha\uparrow<L>}$ is time guarded if R_α is. $R_{\alpha \xrightarrow{ch\triangleright m} \beta}$ is time guarded, because it either represents α alone (if $ch{\triangleright}m$ does not occur during the behavior corresponding to α), or it represents prefixes of α's behavior, sequentially composed with the behaviors of β. In the latter case, the delay of at least one time unit, introduced by

message occurrence in β, establishes the time guardedness of preemption. A similar line of thought shows that $R_{\alpha \Uparrow_{ch \triangleright m}}$ is time guarded. Finally, $R_{\alpha \mapsto \beta}$ is time guarded, because there is at least one time unit of delay between the occurrence of the last message of the trigger, and the first message of the triggered MSC. ■

B.4.2. Join Consistency

Proposition 58 (Join Consistency) *Let α and β be normalized MSCs. α and β are consistent with respect to join if and only if α and β are join-consistent with respect to every component appearing in α and β.* □

Proof Without loss of generality we assume that the normalized MSCs α and β contain precisely one component axis labeled F. Moreover, we assume that s_0^α and s_0^β are the start guards of α and β, respectively, and that s_T^α and s_T^β are the corresponding end guards. Let A_α and A_β be the automata obtained from α and β by application of the transformation procedure. We reduce the claim of the proposition to the following two proof obligations:

1. $[\![\alpha \otimes \beta]\!]_0 \neq \emptyset \quad \Rightarrow \quad (s_T^\alpha, s_T^\beta)$ is reachable in $\delta_{A_\alpha \otimes A_\beta}$,
2. (s_T^α, s_T^β) is reachable in $\delta_{A_\alpha \otimes A_\beta} \quad \Rightarrow \quad [\![\alpha \otimes \beta]\!]_0 \neq \emptyset$

We now discharge each of these proof obligations, in turn.

<u>$[\![\alpha \otimes \beta]\!]_0 \neq \emptyset \quad \Rightarrow \quad (s_T^\alpha, s_T^\beta)$ is reachable in $\delta_{A_\alpha \otimes A_\beta}$:</u>

Assume $\varphi \in [\![\alpha \otimes \beta]\!]$ holds for some $\varphi \in (\tilde{C} \times S)^\infty$. This implies, in particular, that there exist $t_1 \in \mathbb{N}$ and $t_2 \in \mathbb{N}$, such that $(\varphi, t_1) \in [\![\alpha]\!]_0$ and $(\varphi, t_2) \in [\![\beta]\!]_0$. This, in turn, implies the existence of $\varphi', \varphi'' \in (\tilde{C} \times S)^\infty$ such that both $(\varphi', t_1) \in [\![\alpha]\!]_{0,CW}$ and $(\varphi'', t_2) \in [\![\beta]\!]_{0,CW}$ hold. The step from φ to φ' involves, for instance, dropping all message occurrences outside the set $msgs.\alpha$, as well as all redundant messages from the set $msgs.\alpha$. We obtain φ'' from φ in the same way, now considering the messages outside and inside $msgs.\beta$, respectively.

Assume that (s_T^α, s_T^β) is not reachable from (s_0^α, s_0^β) in $\delta_{A_\alpha \otimes A_\beta}$. This implies the existence of a state $(s^\alpha, s^\beta) \in S_{A_\alpha} \times S_{A_\beta}$ with $(s^\alpha, s^\beta) \neq (s_T^\alpha, s_T^\beta)$, such that (s^α, s^β) is reachable from (s_0^α, s_0^β) but no other state is reachable from (s^α, s^β). Because α and β are normalized, δ_{A_α} and δ_{A_β} connect s_0^α and s_0^β with s_T^α and s_T^β, respectively, in a “linear” way, i.e. there are neither explicit loops, nor ϵ-transitions in the automata A_α and A_β. Hence, in the cross product $A_\alpha \otimes A_\beta$ the reachability of (s_T^α, s_T^β) from (s^α, s^β) can only fail if there is a message $ch \triangleright m \in (\hat{I}_\epsilon^{A_\alpha} \cap \hat{I}_\epsilon^{A_\beta})$ such that

$$(s_n^\alpha, s_n^\beta) \notin (\delta_{A_\alpha}(s^\alpha, ch \triangleright m, \epsilon), \delta_{A_\beta}(s^\beta, ch \triangleright m, \epsilon))$$

holds, or there exists a message $ch{\triangleright}m \in (\hat{O}_{\epsilon}^{A_\alpha} \cap \hat{O}_{\epsilon}^{A_\beta})$ such that

$$(s_n^\alpha, s_n^\beta) \notin (\delta_{A_\alpha}(s^\alpha, \epsilon, ch{\triangleright}m), \delta_{A_\beta}(s^\beta, \epsilon, ch{\triangleright}m))$$

holds for all $(s_n^\alpha, s_n^\beta) \in S_{A_\alpha} \times S_{A_\beta}$.

This, however, means that we cannot construct an execution $\hat{\varphi} \in (\tilde{C} \times S)^\infty$ such that both $\hat{\varphi} \in [\![A_\alpha]\!]$ and $\hat{\varphi} \in [\![A_\beta]\!]$ hold. This contradicts our observation above that there exist $\varphi' \in (\tilde{C} \times S)^\infty$ and times $t_1, t_2 \in \mathbb{N}$ with $(\varphi', t_1) \in [\![\alpha]\!]_{0,CW}$ and $(\varphi', t_2) \in [\![\beta]\!]_{0,CW}$, because by construction of A_α and A_β we easily obtain the validity of $\varphi' \in [\![A_\alpha]\!]$ and $\varphi' \in [\![A_\beta]\!]$.

This shows that $[\![\alpha \otimes \beta]\!] \neq \emptyset$ implies that (s_T^α, s_T^β) is reachable from (s_0^α, s_0^β) in $\delta_{A_\alpha \otimes A_\beta}$.

<u>(s_T^α, s_T^β) is reachable in $\delta_{A_\alpha \otimes A_\beta} \;\; \Rightarrow \;\; [\![\alpha \otimes \beta]\!]_0 \neq \emptyset$:</u>

Assume that (s_T^α, s_T^β) is reachable from (s_0^α, s_0^β) in $\delta_{A_\alpha \otimes A_\beta}$. Pick an element $\varphi \in [\![A_\alpha \otimes A_\beta]\!]$ with $\pi_2(\varphi)|_F.0 = (s_0^\alpha, s_0^\beta)$ and $t_T = \min\{t' \in \mathbb{N} : \pi_2(\varphi)|_F.t' = (s_T^\alpha, s_T^\beta)\}$. We show this implies the validity of $(\varphi, t_T) \in [\![\alpha \otimes \beta]\!]_0$.

Assume the opposite, i.e. $(\varphi, t_T) \notin [\![\alpha \otimes \beta]\!]_0$. This can happen if either of the outer conjuncts of the join operator's semantics yields false. The first one requires that there are times $t_1, t_2 \in \mathbb{N}$ such that both $(\varphi, t_1) \in [\![\alpha]\!]_0$ and $(\varphi, t_2) \in [\![\beta]\!]_0$ hold. This, however, cannot be false by construction of $\delta_{A_\alpha \otimes A_\beta}$. We have, in particular, $\varphi \in [\![A_\alpha \otimes A_\beta]\!] \Rightarrow (\hat{\varphi} \in [\![A_\alpha]\!] \wedge \hat{\varphi} \in [\![A_\beta]\!])$ where $\hat{\varphi}$ equals φ up to the restriction of the state label pairs from $S_{A_\alpha} \times S_{A_\beta}$ to their first and second component, respectively. Because $\hat{\varphi} \in [\![A_\alpha]\!] \Rightarrow (\hat{\varphi}, t_1) \in [\![\alpha]\!]_0$ for some $t_1 \in \mathbb{N}$ and $\hat{\varphi} \in [\![A_\beta]\!] \Rightarrow (\hat{\varphi}, t_2) \in [\![\beta]\!]_0$ for some $t_2 \in \mathbb{N}$ we obtain the validity of the first conjunct of the join operator's definition.

Therefore, if we assume $(\varphi, t_T) \notin [\![\alpha \otimes \beta]\!]_0$, the other conjunct must be false. Hence, there is a $\psi \in (\tilde{C} \times S)^\infty$, a time $t_0 \in \mathbb{N}$, and an element $ch{\triangleright}m \in msgs.\alpha \cap msgs.\beta$, such that both

$$\langle \forall t \in \mathbb{N} : t \neq t_0 : \psi.t = \varphi.t \rangle$$

and

$$\psi.t_0.ch = \langle\rangle \wedge \varphi.t_0.ch = \langle m \rangle$$

hold, with $(\psi, t_1') \in [\![\alpha]\!]_0$ or $(\psi, t_2') \in [\![\beta]\!]_0$ for some $t_1', t_2' \in \mathbb{N}$.

However, this implies that $\hat{\varphi}$ is then not an element of the closed world semantics of either α or β, i.e. $\hat{\varphi} \notin [\![\alpha]\!]_{0,CW}$ or $\hat{\varphi} \notin [\![\beta]\!]_{0,CW}$ holds. This contradicts our observation that $\hat{\varphi} \in [\![A_\alpha]\!]$ and $\hat{\varphi} \in [\![A_\beta]\!]$ holds, because we have $[\![A_\alpha]\!] \subseteq [\![\alpha]\!]_{CW}$ and $[\![A_\beta]\!] \subseteq [\![\beta]\!]_{CW}$.

Thus, we have established that if (s_T^α, s_T^β) is reachable from (s_0^α, s_0^β) in $\delta_{A_\alpha \otimes A_\beta}$, then $[\![\alpha \otimes \beta]\!]$ is nonempty. ∎

Bibliography

[AH93] Luca Aceto and Mathew Hennessy. Towards Action-Refinement in Process Algebras. *Information and Computation*, 103(2):204–269, 1993.

[AHP96] Rajeev Alur, Gerard J. Holzmann, and Doron Peled. An Analyzer for Message Sequence Charts. *Software — Concepts and Tools*, 17:70 – 77, 1996.

[ARS98] Camille Ben Achour, Colette Rolland, and Carine Souveyet. A Proposal for Improving The Quality of the Organization of Scenarios Collections. Technical Report 98-24, RWTH Aachen, 1998. CREWS Report Series (obtained via `http://sunsite.informatik.rwth-aachen.de/CREWS`).

[AS85] Bowen Alpern and Fred B. Schneider. Defining Liveness. *Information Processing Letters*, 21(4):181–185, 1985.

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilerbau*, volume I. Addison-Wesley, 1988.

[ATS99] Camille Ben Achour, Mustapha Tawbi, and Carine Souveyet. Bridging the Gap Between Users and Requirements Engineering: The Scenario-Based Approach. Technical Report 99-07, RWTH Aachen, 1999. CREWS Report Series (obtained via `http://sunsite.informatik.rwth-aachen.de/CREWS`).

[BAL96] Hanêne Ben-Abdallah and Stefan Leue. Architecture of a Requirements and Design Tool Based on Message Sequence Charts. Technical Report 96-13, University of Waterloo, 1996.

[BC96] Raymond J. A. Buhr and Ron S. Casselman. *Use CASE Maps for Object-Oriented Systems*. Prentice Hall, 1996.

[BDD+92] Manfred Broy, Frank Dederichs, Claus Dendorfer, Max Fuchs, Thomas F. Gritzner, and Rainer Weber. The Design of Distributed Systems. An Introduction to FOCUS – Revised Version –. Technical Report TUM-I9202-2, Technische Universität München, 1992.

[Ber97] Dorothea Beringer. *Modelling Global Behaviour With Scenarios In Object-Oriented Analysis*. PhD thesis, École Polytechnique Fédérale de Lausanne, 1997.

[BG88] Gérard Berry and Georges Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. Technical Report 842, INRIA, 1988.

[BGH+97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Proceedings OOPSLA'97 Workshop on Object-oriented Behavioral Semantics*, TUM-I9737, 1997.

[BGH+98] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and complete object interaction descriptions. *Computer Standards & Interfaces*, 19:335 – 345, 1998.

[BGK99] Manfred Broy, Radu Grosu, and Ingolf Krüger. Verfahren zum Automatischen Erzeugen eines Programms. Deutsches Patent, Aktenzeichen 198 37 871, 1999. (German Patent).

[BHKS97a] Manfred Broy, Christoph Hofmann, Ingolf Krüger, and Monika Schmidt. A graphical description technique for communication in software architectures. Technical Report TUM-I9705, Technische Universität München, 1997.

[BHKS97b] Manfred Broy, Christoph Hofmann, Ingolf Krüger, and Monika Schmidt. Using Extended Event Traces to Describe Communication in Software Architectures. In *Proceedings of the Asia-Pacific Software Engineering Conference and International Computer Science Conference*. IEEE Computer Society, 1997.

[BHS99] Manfred Broy, Franz Huber, and Bernhard Schätz. AutoFocus – Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme. *Informatik Forschung und Entwicklung*, 14(3):121–134, 1999.

[BK98] Manfred Broy and Ingolf Krüger. Interaction Interfaces – Towards a scientific foundation of a methodological usage of Message Sequence Charts. In J. Staples, M. G. Hinchey, and Shaoying Liu, editors, *Formal Engineering Methods (ICFEM'98)*, pages 2–15. IEEE Computer Society, 1998.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A System of Patterns. Pattern-Oriented Software Architecture.* Wiley, 1996.

[Boe88] Barry W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, pages 61 – 72, May 1988.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design. With Applications.* Addison-Wesley, 2nd edition, 1994.

[BP99] Max Breitling and Jan Philipps. Black Box Views of State Machines. Technical Report TUM-I9916, Technische Universität München, 1999.

[Bre99] Ruth Breu. *Konzepte, Techniken und Methodik des objektorientierten Entwurfs – Ein integrierter Ansatz.* Technische Universität München, 1999. Habilitationsschrift. (in German).

[Bro87] Manfred Broy. Some algebraic and functional hocuspocus with ABRACADABRA. Technical Report MIP-8717, Fakultät für Mathematik und Informatik, Universität Passau, 1987. also in: Information and Software Technology 32, 1990, pp. 686–696.

[Bro93] Manfred Broy. Interaction Refinement–The Easy Way. In Manfred Broy, editor, *Program Design Calculi*, volume 118 of *NATO ASI Series F*. Springer, 1993.

[Bro95] Manfred Broy. A Functional Rephrasing of the Assumption/Commitment Specification Style. Technical Report TUM-I9417, Technische Universität München, 1995.

[Bro97] Manfred Broy. The Specification of System Components by State Transition Diagrams. Technical Report TUM-I9729, Technische Universität München, 1997.

[Bro98] Manfred Broy. On the Meaning of Message Sequence Charts (Key Note). In Lahav Y. et al., editors, *Proceedings of the 1st Workshop of the SDL Forum Society Workshop on SDL & MSC*, volume I, pages 13–34, 1998.

[Bro99a] Manfred Broy. A Logical Basis for Modular Systems Engineering. In Manfred Broy and Ralf Steinbrüggen, editors, *Calculational System Design*, volume 173 of *NATO Science Series F*, pages 101–130. IOS Press, 1999.

[Bro99b] Manfred Broy. The Essence of Message Sequence Charts. (in preparation), 1999.

[BS00] Manfred Broy and Ketil Stølen. *Focus on System Development.* Springer, 2000. (to appear).

[CC92] Patrick Cousot and Radhia Cousot. Inductive Definitions, Semantics and Abstract Interpretation. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 83–94, Albuquerque, New Mexico, January 1992. ACM Press, New York, NY.

[CD94] Steve Cook and John Daniels. *Designing Object Systems. Object-Oriented Modelling with Syntropy.* Prentice Hall, 1994.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design. A Foundation.* Addison Wesley, 1988.

[CMP91] Edward Chang, Zohar Manna, and Amir Pnueli. The Safety-Progress Classification. In F.L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *In Logic and Algebra of Specifications*, NATO Advanced Science Institutes Series, pages 143–202. Springer, 1991.

[DDH72] Ole-Johan Dahl, Edsger W. Dijkstra, and C.A.R. Hoare. *Structured Programming*, chapter "Notes on Structured Programming" by Edsger W. Dijkstra, pages 1–82. Academic Press, 1972.

[DH99] Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. In *FMOODS'99 IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems*, 1999.

[Dij68] Edsger W. Dijkstra. *Selected Writings on Computing*, chapter EWD227. Stepwise Program Construction, pages 1–14. Springer, 1968.

[Dou98] Bruce Powell Douglass. *Real-Time UML. Developing Efficient Objects for Embedded Systems.* Addison Wesley, 1998.

[DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics.* Springer, 1990.

[DW98] Desmond D'Souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML– The Catalysis Approach.* Addison Wesley, 1998.

[EHS98] Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *SDL. Formal Object-oriented Language for Communicating Systems.* Prentice Hall, 1998.

[Eme90] E. Allen Emerson. Temporal and Modal Logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier, 1990.

[Fac95] Christian Facchi. *Methodik zur Formalen Spezifikation des ISO/OSI Schichtenmodells.* PhD thesis, Technische Universität München, 1995. (in German).

[Fei99] Loe M. G. Feijs. Generating FSMs from interworkings. *Distributed Computing*, 12:31–40, 1999.

[Fra86] Nissim Francez. *Fairness*. Texts and monographs in computer science. Springer, 1986.

[FS97] Martin Fowler and Kendall Scott. *UML Distilled. Applying the Standard Object Modeling Language*. Addison Wesley, 1997.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GHN93] Jens Grabowski, Dieter Hogrefe, and Robert Nahm. Test Case Generation with Test Purpose Specification by MSCs. In O. Færgemand and A. Sarma, editors, *SDL'93 – Using Objects*, Proceedings of the Sixth SDL Forum, 1993.

[GKRB96] Radu Grosu, Cornel Klein, Bernhard Rumpe, and Manfred Broy. State Transition Diagrams. Technical Report TUM-I9630, Technische Universität München, 1996.

[GKS99a] Radu Grosu, Ingolf Krüger, and Thomas Stauner. Hybrid Sequence Charts. Technical Report TUM-I9914, Technische Universität München, 1999.

[GKS99b] Radu Grosu, Ingolf Krüger, and Thomas Stauner. Requirements Specification of an Automotive System with Hybrid Sequence Charts. In *WORDS'99F, Fifth International Workshop on Object-oriented Real-time Dependable Systems*. IEEE, 1999.

[GKS00] Radu Grosu, Ingolf Krüger, and Thomas Stauner. Hybrid Sequence Charts. In *Proc. of the 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*. IEEE, 2000.

[GKSH99] Jens Grabowski, Beat Koch, Michael Schmitt, and Dieter Hogrefe. SDL and MSC based test generation for distributed test architectures. In R. Dssouli, G.V. Bochman, and Y. Lahav, editors, *SDL'99. The Next Millenium*, Proceedings of the 9th SDL-Forum. Elsevier, June 1999.

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, Technische Universität München, 1995.

[GRG93] Peter Graubmann, Ekkart Rudolph, and Jens Grabowski. Towards a Petri net based semantics definition for Message Sequence Charts. In O. Færgemand and A. Sarma, editors, *SDL'93 – Using Objects*, Proceedings of the Sixth SDL Forum, pages 179–190, 1993.

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[Hau97] Markus Haubner. Transformation von MSCs in Temporallogische Formeln. Master's thesis, Technische Universität München, 1997. (in German).

[HK99] David Harel and Hillel Kugler. Synthesizing Object Systems from LSC Specifications. (submitted), August 1999.

[HMS+98] Franz Huber, Sascha Molterer, Bernhard Schätz, Oscar Slotosch, and Alexander Vilbig. Traffic Lights - An AutoFocus Case Study. In *1998 International Conference on Application of Concurrency to System Design, pp. 282-294*. IEEE Computer Society, 1998.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[Hol95] Gerard J. Holzmann. Formal Methods for Early Fault Detection. In *TACAS'95*, volume 1135 of *LNCS*, pages 40 – 54. Springer, 1995.

[Hol96] Gerard J. Holzmann. Early Fault Detection Tools. *Software — Concepts and Tools*, 17:63 – 69, 1996.

[Hol97] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279 – 295, 1997.

[HP98] David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts. The* STATEMATE *approach*. McGraw-Hill, 1998.

[HSG+94] Pei Hsia, Jayarajan Samuel, Jerry Gao, David Kung, Yasufumi Toyoshima, and Chris Chen. Formal Approach to Scenario Analysis. *IEEE Software*, pages 33 – 41, March 1994.

[HSS96] Franz Huber, Bernhard Schätz, and Katharina Spies. AutoFocus – Ein Werkzeugkonzept zur Beschreibung verteilter Systeme. In U. Herzog and H. Hermanns, editors, *Formale Beschreibungstechniken für verteilte Systeme*, pages 165–174. Universität Erlangen-Nürnberg, 1996. Published in: Arbeitsberichte des Insituts für mathematische Maschinen und Datenverarbeitung, Vol. 29, Nr. 9.

[HSSS96] Franz Huber, Bernhard Schätz, Alexander Schmidt, and Katharina Spies. Autofocus – a tool for distributed systems specification. In B. Jonsson and J. Parrow, editors, *Proceedings FTRTFT'96 - Formal Techniques in Real-Time and Fault-Tolerant Systems, P. 467-470*. Springer Verlag, LNCS 1135, 1996.

[HU90] John E. Hopcroft and Jeffrey D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Addison-Wesley, 1990.

[HU99] Esfandiar Haghverdi and Hasan Ural. Submodule construction from concurrent system specifications. *Information and Software Technology*, 41:499–506, 1999.

[ISO87] ISO. Information Processing Systems – Open Systems Interconnection – service conventions, 1987.

[IT96] ITU-TS. Recommendation Z.120 : Message Sequence Chart (MSC). Geneva, 1996.

[IT98] ITU-TS. Recommendation Z.120 : Annex B. Geneva, 1998.

[IT99] ITU-TS. Recommendation Z.120 (11/99) : MSC 2000. Geneva, 1999.

[JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[JCJO92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering – A Use Case Driven Approach*. Addison Wesley, 1992.

[KGSB99] Ingolf Krüger, Radu Grosu, Peter Scholz, and Manfred Broy. From MSCs to Statecharts. In *DIPES'98*. Kluwer, 1999.

[Kle98] Cornel Klein. *Anforderungsspezifikation durch Transitionssysteme und Szenarien*. PhD thesis, Technische Universität München, 1998. (in German).

[KM93] Kai Koskimies and Erkki Mäkinen. Inferring State Machines From Trace Diagrams. Technical Report A-1993-3, University of Tampere. Department of Computer Science, July 1993.

[KM99] Ekkart Kindler and Axel Martens. Szenarios: Lokale Kriterien für globale Korrektheit. In Katharina Spies and Bernhard Schätz, editors, *Formale Beschreibungstechniken für verteilte Systeme. FBT'99*, 9. GI/ITG Fachgespräch, pages 113–122. Herbert Utz Verlag, June 1999.

[KMST96] Kai Koskimies, Tatu Männistö, Tarja Systä, and Jyrki Tuomi. On the Role of Scenarios in Object-Oriented Software Design. Technical Report A-1996-1, University of Tampere. Department of Computer Science, January 1996.

[Kru99a] Philippe Kruchten. *The Rational Unified Processs. An Introduction*. Addison Wesley, 1999.

[Krü99b] Ingolf Krüger. Towards the Methodical Usage of Message Sequence Charts. In Katharina Spies and Bernhard Schätz, editors, *Formale Beschreibungstechniken für verteilte Systeme. FBT'99*, 9. GI/ITG Fachgespräch, pages 123–134. Herbert Utz Verlag, June 1999.

[KSTM98] Kai Koskimies, Tarja Systä, Jyrki Tuomi, and Tatu Männistö. Automated Support for Modeling OO Software. *IEEE Software*, pages 87 – 94, January–February 1998.

[KW98] Piotr Kosiuczenko and Martin Wirsing. Towards an Integration of Message Sequence Charts and Timed Maude. In M.M. Tanik, J. Tauka, K. Itoh, M. Goedicke, W. Rossak, H. Ehrig, and H. Kurfess, editors, *3rd World Conference on Integrated Design and Process Technology, IDPT'98*, pages 88–95, Berlin, Austin: Society for Design and Process Science, 1998. (revised version to appear in Journal of IDPT, 2000).

[Lam77] Leslie Lamport. Proving the Correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, TSE-3(2):125–143, 1977.

[Lam94] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872 – 923, May 1994.

[Lam98] Leslie Lamport. Composition: A Way to Make Proofs Harder. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference*, volume 1536 of *LNCS*, pages 402–423. Springer, 1998.

[Lam99] Leslie Lamport. Specifying Concurrent Systems with TLA$^+$. In Manfred Broy and Ralf Steinbrüggen, editors, *Calculational System Design*, volume 173 of *NATO Science Series F*, pages 183–247. IOS Press, 1999.

[Leu95] Stefan Leue. *Methods and Semantics for Telecommunications Systems Engineering*. PhD thesis, Universität Bern, 1995.

[LL95] Peter B. Ladkin and Stefan Leue. Interpreting Message Flow Graphs. *Formal Aspects of Computing*, (5):473–509, 1995.

[LMR98] Stefan Leue, Lars Mehrmann, and Mohammad Rezai. Synthesizing ROOM Models from Message Sequence Chart Specifications. Technical Report 98-06, University of Waterloo, 1998.

[LT87] Nancy Lynch and Mark Tuttle. Hierarchical Correctness Proof for Distributed Algorithms. Technical Report MIT/LCS/TR-387, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA., 1987.

[LT89] Nancy Lynch and Mark Tuttle. An Introduction to Input/Output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[LV95] Nancy Lynch and Frits Vaandrager. Forward and Backward Simulations – Part I: Untimed Systems. *Information and Computation*, 121(3):214–233, 1995.

[LV96] Nancy Lynch and Frits Vaandrager. Forward and Backward Simulations – Part II: Timing-Based Systems. *Information and Computation*, 128(1):1–25, 1996.

[Lyn96] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[Mar92] Florence Maraninchi. Operational and Compositional Semantics of Synchronous Automaton Compositions. In W.R. Cleaveland, editor, *Proceedings CONCUR'92*, volume 630 of *LNCS*, pages 550–564. Springer, 1992.

[MB83] Philip Merlin and Gregor von Bochman. On the construction of submodule specifications and communication protocols. *ACM Transactions on Programming Languages and Systems*, 5(1):1–25, 1983.

[MC81] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7:417–426, 1981.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Number 92 in Lecture Notes in Computer Science. Springer, 1980.

[Möl99] Bernhard Möller. Algebraic Structures for Program Calculation. In Manfred Broy and Ralf Steinbrüggen, editors, *Calculational System Design*, volume 173 of *NATO Science Series F*, pages 25–97. IOS Press, 1999.

[MP95] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, 1995.

[MR94] Sjouke Mauw and Michel Adriaan Reniers. An Algebraic Semantics of Basic Message Sequence Charts. *The Computer Journal*, 37(4), 1994.

[MR96] Sjouke Mauw and Michel Adriaan Reniers. Refinement in Interworkings. In U. Montanari and V. Sassone, editors, *CONCUR'96*, volume 1119 of *LNCS*, pages 671–686. Springer, 1996.

[Mül98] Olaf Müller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory*. PhD thesis, Technische Universität München, 1998.

[NGH93] Robert Nahm, Jens Grabowski, and Dieter Hogrefe. Test Case Generation for Temporal Properties. Technical Report IAM-93-013, University of Berne, Institute for Informatics, Berne, Switzerland, June 1993.

[Pae97] Barbara Paech. A Framework for Interaction Description with Roles. Technical Report TUM-I9731, Technische Universität München, June 1997.

[Pan90] Paritosh K. Pandya. Some comments on the assumption-commitment framework for compositional verification of distributed programs. In *Proc. REX Workshop on Stepwise Refinement of Distributed Systems*, number 430 in Lecture Notes in Computer Science, pages 622–640, 1990.

[Pra86] Vaughan R. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

[RA98] Colette Rolland and Camille Ben Achour. Guiding the Construction of Textual Use Case Specifications. Technical Report 98-1, RWTH Aachen, 1998. CREWS Report Series (obtained via `http://sunsite.informatik.rwth-aachen.de/CREWS`).

[Rat97] Unified Modeling Language, Version 1.1. Rational Software Corporation, 1997.

[RBK99a] Bernhard Rumpe, Ruth Breu, and Ingolf Krüger. Applied Software Engineering Principles for UML. Tutorial at the TOOLS Europe'99, 29th International Conference, June 1999.

[RBK99b] Bernhard Rumpe, Ruth Breu, and Ingolf Krüger. Applied Software Engineering Principles for UML. Tutorial at the TOOLS Pacific'99, 32nd International Conference, November 1999.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modelling and Design*. Prentice-Hall, 1991.

[Rei82] Wolfgang Reisig. *Petri-Netze – eine Einführung*. Springer, 1982.

[Rem92] Martin Rem. A Personal Perspective of the Alpern-Schneider Characterization of Safety and Liveness. In Jayadev Misra, editor, *Beauty is our Business*, pages 365–372. Springer, 1992.

[Ren99] Michel Adriaan Reniers. *Message Sequence Chart. Syntax and Semantics*. PhD thesis, Eindhoven University of Technology, 1999.

[RGG99] Ekkart Rudolph, Jens Grabowski, and Peter Graubmann. Towards a Harmonization of UML-Sequence Diagrams and MSC. In R. Dssouli, G.V. Bochman, and Y. Lahav, editors, *SDL'99. The Next Millenium*, Proceedings of the 9th SDL-Forum. Elsevier, June 1999.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[RKW95] Björn Regnell, Kristofer Kimbler, and Anders Wesslén. Improving the Use Case Driven Approach to Requirements Engineering. In *Proceedings Requirements Engineering*. IEEE, 1995.

[Roy87] W.W. Royce. Managing the Development of Large Software Systems. In *Proc. ICSE*, pages 328 – 339. IEEE, 1987. reprinted from *IEEE WESCON*, 1970, pp. 1 – 9.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. PhD thesis, TU München, 1996. (in German).

[Sch98] Peter Scholz. *Design of Reactive Systems and their Distributed Implementation with Statecharts*. PhD thesis, Technische Universität München, 1998.

[SD93] Rainer Schlör and Werner Damm. Specification and verification of system level hardware designs using timing diagrams. In *Proc. European Conference on Design Automation*, pages 518–524, 1993.

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. Wiley, 1994.

[Sha98] Natarajan Shankar. Lazy Compositional Verification. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference*, volume 1536 of *LNCS*, pages 541–564. Springer, 1998.

[SHB96] Bernhard Schätz, Heinrich Hußmann, and Manfred Broy. Graphical Development of Consistent System Specifications. In J. Woodcock and M.-C. Gaudel, editors, *FME'96:Industrial Benefit and Advances in Formal Methods*, volume 1051 of *LNCS*. Springer, 1996.

[SKGH98] Michael Schmitt, Beat Koch, Jens Grabowski, and Dieter Hogrefe. Autolink – Putting formal test methods into practice. Schriftenreihe der Institute für Mathematik/Informatik A-98-04, Medical University of Lübeck, April 1998.

[SRS99] Thomas Stauner, Bernhard Rumpe, and Peter Scholz. Hybrid System Model. Technical Report TUM-I9903, Technische Universität München, 1999.

[Ste97] Robert Stephens. A Survey of Stream Processing. *Acta Informatica*, 34:491–541, 1997.

[Tho90] Wolfgang Thomas. Automata on Infinite Objects. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. Elsevier, 1990.

[UBE99] uBET Homepage, 1999. `http://cm.bell-labs.com/cm/cs/what/ubet`.

[V97] Entwicklungsstandard für IT-Systeme des Bundes, Vorgehensmodell. Allgemeiner Umdruck Nr. 250/1. Juni 1997, BWB IT I5.

[vdB94] Michael von der Beeck. A Comparison of Statecharts Variants. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Proc. Formal Techniques in Real-Time and Fault-Tolearnt Systems (FTRTFT'94)*, volume 863 of *LNCS*, pages 128–148. Springer, 1994.

[vG96] Robert J.H. van Glabbeek. *Comparative concurrency semantics and refinement of actions.* CWI Tracts, 1996.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction.* The MIT Press, 1993.

[Wir71] Niklaus Wirth. Program Development by Stepwise Refinement. *Communication of the ACM*, 14(4), 1971.

[Wir86] Niklaus Wirth. *Compilerbau.* Teubner, 1986.

[WK96] Martin Wirsing and Alexander Knapp. A Formal Approach to Object-Oriented Software Engineering. In José Meseguer, editor, *Proc. 1st Int. Wsh. Rewriting Logic and Its Applications*, volume 4 of *Electr. Notes Theo. Comp. Sci.*, pages 321–359, 1996. (revised version).

[XS98] Qiwen Xu and Mohalik Swarup. Compositional Reasoning Using the Assumption-Commitment Paradigm. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference*, volume 1536 of *LNCS*, pages 565–583. Springer, 1998.