
Message Sequence Chart

Syntax and Semantics

M.A. Reniers

Message Sequence Chart

Syntax and Semantics

Reniers, Michel Adriaan

Message Sequence Chart: Syntax and Semantics / Michel Adriaan Reniers. -

Eindhoven : Eindhoven University of Technology, 1999. - viii, 216 p.

Proefschrift. -

With summary in Dutch

IPA Dissertation Series 1999-07.

druk: UniversiteitsDrukkerij, Eindhoven

©1999 by Michel Adriaan Reniers, Eindhoven, The Netherlands.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior permission of the author.



This thesis has been supported by the Philips Research Laboratories Eindhoven. It has been carried out under the auspices of the Institute for Programming Research and Algorithmics (IPA).

Message Sequence Chart

Syntax and Semantics

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof. dr. M. Rem, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op maandag 7 juni 1999 om 16.00 uur

door

Michel Adriaan Reniers

geboren te Eindhoven

Dit proefschrift is goedgekeurd door de promotoren:

prof. dr. J. C. M. Baeten

en

prof. dr. ir. L. M. G. Feijs

Copromotor:

dr. S. Mauw

Preface

My stay at Eindhoven University of Technology has been an interesting experience in many ways. First of all, I felt at home right away. This is not surprising considering the following. I was born in Eindhoven, I have lived in Valkenswaard (close to Eindhoven) for twenty-three years, I graduated from Eindhoven University of Technology in the same department as where I became a Ph.D. student, I live in Eindhoven since the summer of 1994 and I am planning to stay in Eindhoven for several more years.

At all times during the period that I was a Ph.D. student I have had the opportunity to learn a lot. This has changed my outlook on computing science considerably. The freedom given to me by my supervisors has enabled me to focus on different topics in computing science from time to time. The willingness of my colleagues to discuss their or my work has created a nice working environment.

In very different ways, numerous people have contributed to the writing of this thesis. It goes too far to thank all of them separately. Therefore, I would like to express my gratitude to all who have contributed to the research and hard work that have been collected in this thesis.

Special thanks go to my family and girlfriend Evelien, and my supervisors Jos Baeten, Loe Feijs and Sjouke Mauw. Jos and Loe are my promotors and rightly so. I admire Jos for the work he has performed in process algebra and for the way in which he deals with his staff and Ph.D. students. I admire Loe for the way in which he pursues the application of formal methods to questions and problems from practice. I thank Sjouke for the way in which he has acted as a daily supervisor.

I would like to express my gratitude to Philips Research Laboratories Eindhoven for funding my Ph.D. position, for their interest in the topic of this thesis and for showing me their daily practice through two projects that took place at Philips Research Laboratories Eindhoven. This last experience has learned me that doing research in a university and doing research in an industrial setting are different issues.

I enjoyed the meetings of the MSC standardization group that took place regularly in Geneva. The participants from different companies were all very kind and made it very easy to integrate with them. The dinner and drinking sessions made it definitely worthwhile to travel to Geneva (by train) once in a while.

Finally, I would like to thank the members of my promotion committee for finding the time to read my thesis, to give me recommendations for improvements and for finding mistakes.

Michel Reniers,
April 20, 1999

Contents

Preface	i
Contents	iii
1 Introduction	1
2 Introduction to the language MSC	17
2.1 Introduction	17
2.2 Basic Message Sequence Charts	18
2.2.1 Graphical representation	18
2.2.2 Intuitive semantics	21
2.2.3 Textual representation	24
2.3 Additional basic concepts	29
2.3.1 Process creation and process termination	29
2.3.2 Timer handling	31
2.3.3 Incomplete message events	33
2.3.4 Conditions	35
2.4 Ordering facilities	37
2.4.1 Coregions	37
2.4.2 Causal orderings	40
2.5 Combining MSCs with composition constructs	45

2.5.1	Vertical, horizontal and alternative composition	45
2.5.2	MSC documents	50
2.5.3	Inline expressions	51
2.5.4	MSC reference expressions	56
2.5.5	High-level Message Sequence Charts	61
2.6	Gates	68
2.6.1	Formal gate definitions	68
2.6.2	MSC reference expressions and gates	73
2.6.3	Inline expressions and gates	79
2.7	Comments	82
2.8	Instance decomposition	83
2.9	Remarks on recommendation Z.120	84
2.9.1	Informal parts of the recommendation	84
2.9.2	Process creation and termination with composition	85
2.9.3	The keyword after	86
3	Process theory for Message Sequence Charts	89
3.1	Introduction	89
3.2	Operational semantics	90
3.2.1	Process expressions	90
3.2.2	State transformations	91
3.3	Deadlock, empty process and atomic actions	94
3.4	Delayed choice	94
3.5	Delayed parallel composition	96
3.6	Weak sequential composition	99
3.7	Generalization of the composition operators	104
3.8	Renaming operator	107
3.9	Repetitive behavior	108

3.9.1	Iteration	108
3.9.2	Unbounded repetition	109
3.10	Congruence and determinism	111
4	Semantics of Message Sequence Charts	115
4.1	Introduction	115
4.2	The approach	115
4.2.1	MSC documents	116
4.2.2	Message Sequence Charts	116
4.2.3	Message Sequence Chart bodies	116
4.2.4	Events	121
4.2.5	Coregions	121
4.2.6	MSC reference expressions	122
4.2.7	Inline expressions	125
4.2.8	High-level Message Sequence Charts	125
4.3	Semantics of an MSC document	126
4.4	Semantics of events	128
4.4.1	Local actions	128
4.4.2	Message events	129
4.4.3	Incomplete message events	131
4.4.4	Instance create and instance stop events	132
4.4.5	Timer events	133
4.4.6	Conditions	133
4.5	Semantics of causally ordered events	133
4.6	Vertical and horizontal composition	139
4.7	Semantics of coregions	142
4.8	Semantics of MSC bodies	143
4.9	Semantics of MSC reference expressions	145

4.10	Semantics of inline expressions	150
4.11	Semantics of HMSCs	152
4.12	Related work on the semantics of MSC	158
4.12.1	Petri-net semantics	158
4.12.2	Büchi automata semantics	158
4.12.3	Process algebra approach	159
4.12.4	Partial order semantics	160
5	Concluding remarks	161
	Bibliography	167
A	Textual syntax of MSC	177
A.1	Changes to the textual syntax	177
A.1.1	Parts of the language that are not treated	178
A.1.2	Irrelevant information	179
A.1.3	Shorthands	180
A.1.4	Extensions	181
A.1.5	Assumptions	182
A.2	Textual syntax for semantics definition	182
B	Proofs	187
B.1	Bisimulation modulo equational reasoning	187
B.2	Properties of \mp	191
B.2.1	Unit element for \mp	191
B.2.2	Commutativity of \mp	191
B.2.3	Associativity of \mp	192
B.2.4	Idempotency of \mp	193
B.3	Properties of \parallel	193

B.3.1	Unit element for \parallel	193
B.3.2	Commutativity of \parallel	194
B.3.3	Distributivity of \parallel over \mp	194
B.3.4	Associativity of \parallel	194
B.4	Properties of \circ	195
B.4.1	Unit element for \circ	195
B.4.2	Left-zero element for \circ	196
B.4.3	Distributivity of \circ over \mp	196
B.4.4	Associativity of \circ	196
B.5	Properties of generalized operators	198
B.5.1	Commutativity of \parallel^S	198
B.5.2	Distributivity of \parallel^S over \mp	198
B.5.3	Left-zero element for \circ^S	199
B.5.4	Distributivity of \circ^S over \mp	199
B.6	Properties of the repetition operators	201
B.6.1	Unfolding of iteration	201
B.6.2	Unfolding of unbounded repetition	202
B.6.3	Inclusion	203
B.6.4	Other properties	204
B.6.5	Other properties (II)	204
	Samenvatting	207
	Curriculum Vitae	211

1

Introduction

It is generally accepted that graphical representations are helpful in communicating information. In the technical sciences and engineering the use of graphical representation techniques for describing the objects under study or for describing relations between such objects is quite common. In cases where only few aspects of reality are of interest, the use of a graphical notation for expressing ideas can be of great help in understanding those.

Roughly speaking, there are two types of pictorial representations that are used in software engineering practice [FJM94]. These are pictures representing the structure of the system's description and pictures representing behavioral aspects of the system. Well-known and frequently used pictorial representations for behavioral aspects of systems are flow charts, Nassi-Shneidermann diagrams [NS73], transition diagrams [Kel76], Petri nets [Rei85], Statecharts [Har87], SDL [IT94], and sequence charts.

Advantages of pictorial representations can be that they are easy to learn, intuitively comprehensible, and that no mathematical background is required for obtaining an understanding of them. An important drawback of graphical representations is that different users of the notation can have different intuitions about the meaning of the picture. Very often it then suffices to explain in a few words what the intention of the drawing is. As a pictorial language grows, due to extensions, it becomes harder to find pictorial representations that are still intuitively clear to the users of the language. Also, with the growth of a language, there is an increased chance of misinterpretation due to the interaction of language features. Then, the development of a formal semantics might be a useful tool to control the language and to support its usage.

The subject of this thesis is the language Message Sequence Chart (MSC) and the definition of its formal semantics. Message Sequence Chart is a graphical language for the description of the interactions between system components. Every system component is represented by a vertical line called an instance. Along an instance,

time runs from top to bottom. Communication is *asynchronous* and no assumptions are made on the way in which this communication is achieved. Exchange of messages between the system components is described by arrows between the instances representing those components. The arrow is directed from the sending instance to the receiving instance. Implicitly, it is assumed that the sending of a message precedes the receiving of the message. In its simplest form, an MSC describes a class of traces of the system under consideration. In Chapter 2 the language MSC is introduced in its full complexity.

An example of an MSC is given in Figure 1.1. This MSC contains the instances $i1$, $i2$, $i3$, and $i4$. These instances exchange the messages $m1$, $m2$, and $m3$. Message $m1$ is sent by instance $i1$ to instance $i2$, message $m2$ is sent by instance $i3$ to instance $i2$, and message $m3$ is sent by instance $i3$ to instance $i4$. The MSC describes the transitive closure of the following orderings between the events contained:

1. the sending of message $m1$ precedes its receiving;
2. the sending of message $m2$ precedes its receiving;
3. the sending of message $m3$ precedes its receiving;
4. the receiving of message $m1$ precedes the receiving of message $m2$;
5. the sending of message $m3$ precedes the sending of message $m2$.

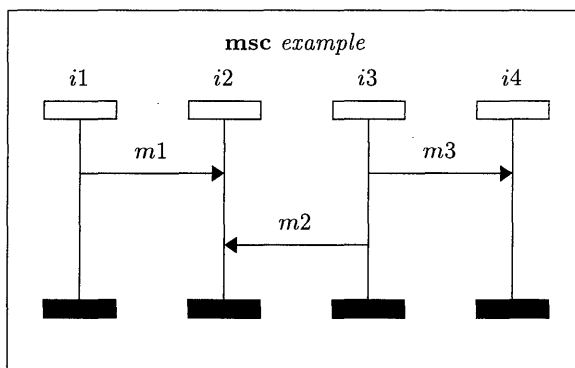


Figure 1.1: Simple Message Sequence Chart.

MSC is applied mostly in telecommunications and in software engineering, but the use of MSC is not restricted to these areas. In this introduction we present a short view on the developments that have led to the definition of the language MSC and its formal semantics. Furthermore, we indicate how the language MSC is used in the life cycle and discuss some related formalisms. At the end of this introduction we give the structure of the other chapters of this thesis.

Historical perspective: the ITU line

The language MSC is standardised by the International Telecommunication Union (ITU). The ITU is subdivided into sectors. A sector consists of study groups and a study group consists of questions. The development and standardization of MSC resides under question 9 of study group 10 of the Telecommunications sector of the ITU. Besides the language MSC the ITU standardizes a number of formalisms for use in the telecommunications domain. Examples of such standardised languages are SDL [IT94], CHILL [IT96c], and ASN.1 [IT88]. The standardization activities of MSC are an offspring of the standardization activities for the Specification and Description Language (SDL) by question 6 of study group 10 of ITU. For this reason, we first give a short (and rather incomplete) description of SDL.

The language SDL can be used to describe both the internal behavior of concurrent processes and the interaction between their interfaces. It is a state-oriented, formal language which is especially suited for event-driven real-time systems. The language SDL can be used with various design methodologies and many tools are available. SDL has two concrete representations: a program-like representation (PR form) and a graphical representation (GR form). In SDL a system is composed of blocks, channels, signal routes, and processes contained in these blocks. Channels and signal routes are the media through which signals are exchanged. Blocks describe a grouping of a number of smaller blocks or processes. In the block several processes may be included. The behavior of processes in telecommunication systems is modeled by, so called, communicating extended finite state machines. A process reacts by changing its state when accepting external stimuli. They are called extended because they can do more than just change state, such as generating responses, store and retrieve information, etc. An important assumption with respect to the communication mechanism in SDL is the following: An unbounded FIFO (First In First Out) input queue is associated with each process. If more than one signal arrives at the same time these are arbitrarily ordered. In Figure 1.2 some process descriptions are given. For more information on SDL we refer to [BHS91] and [FO94].

The SDL User Guidelines [CCI88c] contain a short section on sequence charts as one of the auxiliary diagrams that can be used in combination with SDL. At the SDL Forum held in Lisbon (Portugal) in 1989, Ekkart Rudolph and Jens Grabowski present a paper entitled "Putting Extended Sequence Charts to Practice" [GR89]. These Extended Sequence Charts are sequence charts extended with SDL symbols and other constructs.

In SDL, systems are described by providing an extended finite state machine for each of the processes. As a consequence, there is no good view on the interaction between the processes. In a sequence chart however, the focus is on the interaction between the processes and not so much on the internal behavior of these. The relation between an SDL description and an MSC is sketched informally in Figure 1.3. The MSC has been given in Figure 1.1 and the SDL process descriptions have already been given in Figure 1.2.

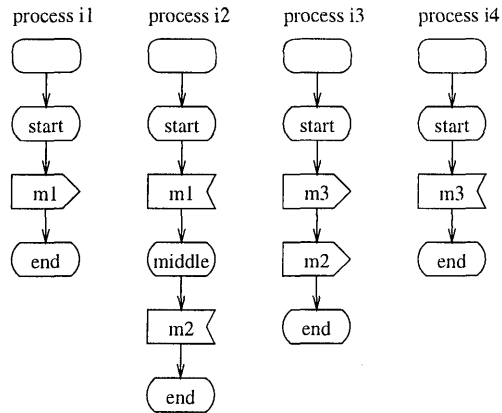


Figure 1.2: Example SDL description.

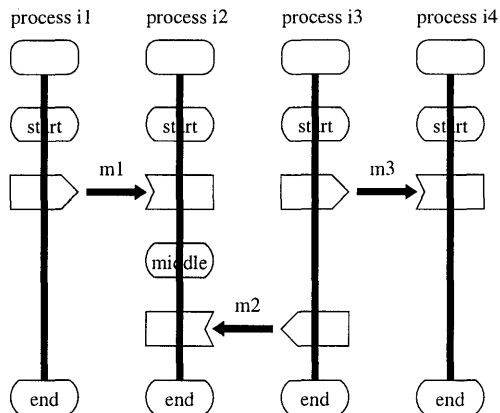


Figure 1.3: Sketch of the relation between SDL and MSC.

As a consequence of the increasing interest in sequence charts, the CCITT¹ approves the standardization of a new language called Message Sequence Chart in June 1990. The standardization activities for MSC are intended to be part of the new SDL Methodology Guidelines [Bel92] and not a recommendation on its own. The emphasis is on the basic constructs and an intuitively clear semantics for those.

Soon it is recognized that the standardization efforts for Message Sequence Chart go beyond the SDL methodology guidelines and the feeling that MSC should not only be related to SDL spreads. The formal decision to have a separate recommendation for Message Sequence Chart is taken at the study group 10 meeting in Geneva in February 1991.

At the closing session of the CCITT study period 1989-1992 in March 1993 the first recommendation for Message Sequence Chart, recommendation Z.120 [IT93], is approved by the World Telecommunication Standardization Conference (WTSC). According to popular belief, this recommendation has largely been written in a Spanish pub in Geneva. This first recommendation of MSC contains a rather informal graphical syntax definition, an abstract syntax definition, a concrete textual syntax definition, an informal explanation of the language, and several examples of MSC diagrams. This first recommendation for MSC contains the following features: MSC, instance, environment, message, timer set, timer reset, timeout, local action, coregion, condition, process creation, process stop, instance decomposition, and subMSC. We refer to Chapter 2 for an informal treatment of these notions.

A formal semantics definition is missing in the first recommendation for the language MSC. The most important reason for having a formal semantics for a language that is used to specify systems is the need for an unambiguous description of the meaning of a system description. Later we will return to the use of a formal semantics.

At the CCITT interims meeting in November 1992 it is recognised that the next study period (1993-1996) should be used to provide a formal semantics definition for the existing MSC recommendation.

Historical perspective: the Philips line

More or less independently of the developments on the language MSC within the ITU, in 1992, Philips Research Laboratories Eindhoven, Philips Kommunikations Industrie Nürnberg (PKI) and Eindhoven University of Technology start a project which aims for the definition of a formal syntax and semantics of the language *Interworking* and the development of tools.

The language *Interworking* is a graphical language in the style of MSC. In this language processes or system components are represented by vertical lines called entities. The interworking between the system components is indicated by means of horizontal arrows indicating *synchronous* communication. These arrows are drawn horizontally

¹CCITT is an abbreviation of Comité Consultatif International Télégraphique et Téléphonique. Nowadays the CCITT is called ITU-T.

from an entity to an entity. An Interworking concentrates mainly on the interactions of the involved entities and not so much on the internal behavior of the entities. An example of an Interworking is given in Figure 1.4 below².

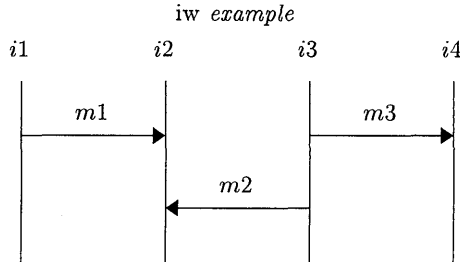


Figure 1.4: An example of an Interworking.

A collection of Interworkings describes the behavior of a system on a high level of abstraction. Each Interworking is a projection of a part of the communication behavior of a system onto a set of entities. Depicting all entities involved in the same Interworking usually results in a diagram that does not fit on the page. In a telecommunication context where for example SDL is used as description language, an entity may be a process or a set of processes combined into one functional block.

The language Interworking was developed in order to support the informal diagrams used at PKI which were used for requirements specification and design. Compared to other trace languages, Interworking has the advantage of a clear graphical layout and structuring. However, Interworkings are only suitable for the description of relatively small parts of the system behavior. In order to give a more complete description of the behavior of a system ways of combining Interworking diagrams into more complex processes are required. One of the reasons for developing an explicit language was that it turned out to be very hard to maintain a large collection of diagrams by hand. Several problems were encountered. First of all, manually drawing and updating large diagrams is a time-consuming and hence expensive activity. Secondly, diagrams that are linked to each other must be updated consistently. Therefore, consistency checks are needed. Thirdly, the relation between the diagrams in a collection is only implicit. Some diagrams describe successive behaviors of one part of the system, other diagrams define the concurrent behavior of different parts of the system, while still others describe the same behavior of the same part of the system, but at a different level of abstraction. Finally, there existed different interpretations of the meaning of even simple Interworkings. In order to solve the abovementioned problems, a formal semantics was proposed [MvWW93] and a tool set was developed [MW93]. The semantics is given via a translation into process algebra [BK84, BW90, BV95].

In [MR96] this semantics, which does not consider the notion of refinement and has some minor shortcomings, is extended. The use of process algebra for the semantics

²Graphically an Interworking is usually depicted without name, but textually an Interworking has a name [MvWW92]. We find it more convenient to depict the name of the Interworking in the diagram as well.

of Interworkings has been proved useful. Based on the process algebra semantics a prototype tool was developed for implementing *interworking sequencing* and *interworking merge*, the vertical and horizontal composition operator respectively, and for the previously mentioned consistency check.

The interworking sequencing (notation \circ_{iw}) of two Interworkings refers to the vertical composition of them. In this composition entities with the same name are connected to form one entity. The interworking sequencing is illustrated in Figure 1.5. Note that the result of the vertical composition of the Interworkings *ex1* and *ex2* is the Interworking *example* from Figure 1.4. Hence, we can also say that the Interworking *example* can be decomposed vertically into the Interworkings *ex1* and *ex2*.

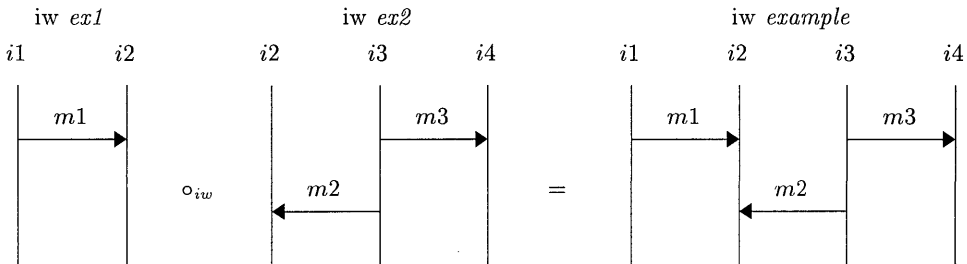


Figure 1.5: Vertical composition of Interworkings.

The interworking merge (notation \parallel_{iw}) refers to the horizontal composition of two Interworkings. The interworking merge of two Interworkings is called *consistent* if the same messages are exchanged in the same order between every pair of entities the Interworkings have in common. The result of such a composition is an Interworking where the entities the Interworkings have in common are placed on top of each other such that similar messages are identified. This is illustrated in Figure 1.6.

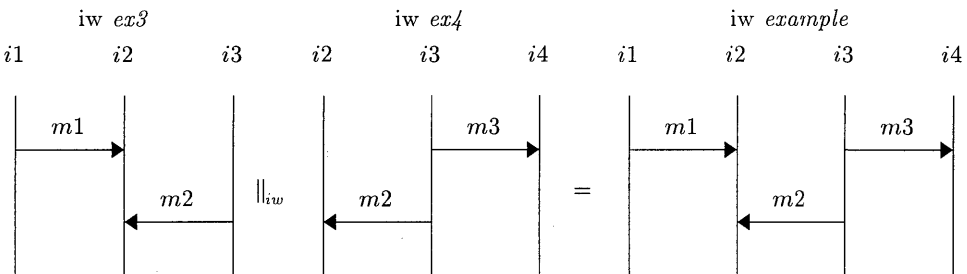


Figure 1.6: Horizontal composition of Interworkings.

In 1992, the results of this project were reported in Geneva during one of the experts meetings of study group 10 of the ITU. At the time, the question of a formal semantics definition of MSC became a topic in the standardization committee for MSC. A dis-

cussion on a proper model for the semantics of MSC is initiated and models based on automata theory [LL92c], Petri nets [GR92] and process algebra [Man93, MvWW92] are discussed. Partly due to the experiences gained with providing a formal semantics definition for the language Interworking the choice was made for a process algebra semantics. This semantics was to be developed at Eindhoven University of Technology. In April 1994, this semantics definition was completed and accepted by study group 10 for Annex B to recommendation Z.120 [MR94b]. In October 1994, “Z.120 Annex B: Algebraic Semantics of Message Sequence Charts” is published by ITU [IT95].

In September 1995, study group 10 accepts a formal definition of the syntax requirements (or static semantics) of Message Sequence Chart [Ren95b] for Annex C to recommendation Z.120 [IT96a]. This document formalizes the informal well-formedness requirements, as described in recommendation Z.120, for the textual syntax of MSC using predicate logic.

MSC takes off

In the meantime, creative minds all over the world have been thinking of new concepts to put into the language. At the rapporteurs meeting in Geneva in April 1996 a draft recommendation for MSC is accepted by study group 10. In October 1996, the WTSC accepts “Z.120: Message Sequence Chart” [IT96b]. In this new recommendation, the language MSC is extended with the following features:

- *causal orderings* (see Section 2.4.2). Causal orderings describe the ordering of two arbitrarily orderable events in an MSC. This feature can be used to describe that two events are ordered in a certain way where this is not necessarily due to communication or process behavior.
- *composition operators*. The 1996 recommendation contains operators for vertical and horizontal composition, and also for denoting alternatives and repetition. There are three ways of denoting these in the language MSC:
 - *MSC reference expressions* (see Section 2.5.4). MSC reference expressions are textual formulas describing the composition of a number of MSCs by referring to them by means of their names.
 - *Inline expressions* (see Section 2.5.3). In an inline expression the composition of a number of fragments of an MSC is indicated explicitly in the MSC. No references to the MSCs are used.
 - *High-level Message Sequence Charts (HMSC)* (see Section 2.5.5). In a HMSC the composition of a number of MSCs is described in a control-flow like format. MSCs are referenced by their names.

The extension of the language MSC with features for describing the composition of MSCs has been influenced greatly by the operators in the language Interworking. A difference between the composition operators in MSC and in Interworking is that the language MSC offers a graphical means of describing these compositions instead of a textual one.

- *gates* (see Section 2.6). The concept of gates has been introduced in the language MSC in order to describe communication and causal orderings that are not in the same scope of description. For example an instance in one MSC wishes to communicate with an instance in another MSC which is composed horizontally with the former MSC.

Besides these features also *lost* and *found messages* (see Section 2.3.3) are introduced and new symbols are defined for the timer events (see Section 2.3.2).

The introduction of these features means that the formal semantics definition for the 1992 recommendation has to be updated. Ideally, a feature should only be included in a language after its semantics has been researched, understood and accepted. This, however, is not the practice of standardization committees.

The language MSC is finding its way into practice rapidly. It goes too far to attribute this success to the ongoing standardization activities. On the contrary, we are reaching the point where the language is getting so complex that we run the risk that its main advantages, simplicity and overview, are lost.

Towards a formal semantics of MSC

The development of a formal semantics for MSC is relevant from several perspectives. From the perspective of language definition (and standardization), the development of a formal semantics itself detects ambiguities, omissions and contradictions ([Koy92]). Furthermore, a formal semantics enables the investigation of notions which are considered important for a proper language design, such as compositionality and orthogonality.

From the perspective of tool builders, it can be convenient to have a formal semantics definition as this can be used as a specification for some types of tools. For example the BNF (Backus-Naur Form) rules defining a textual representation can be used as a specification for the development of a parser. Similarly, the semantics definition can be considered as a specification for the development of tools whose functionality is related to the semantics definition. For a dynamic semantics the first tool that comes to mind is a simulator. But also for checking properties (formulated in a formalism of a logical nature) which refer to the behavior, like a tool for model checking, it is necessary to have a formal semantics. The availability of a standardized formal semantics definition enables the development of independent tools that interpret a specification as intended.

From the perspective of the user of the language, a formal semantics does not seem to have many advantages at first sight. In general most users of the language appreciate an informal explanation backed with a lot of examples more than a precise formal definition of the semantics. However, for a language for which commercial tools are available which are used in combination with a number of other formalisms and tools, it is essential that these tools are based on a formal definition of the language. This

does not only apply to the appearance of the language in, for example, a graphical or textual representation; it also applies to the dynamic behavior as is visible in tools via a simulator, a code generator, or in a testing tool. A formal semantics also enables the comparison of specifications through their semantics. This can be useful to relate specifications at different levels of abstraction. Formal verification or validation is only possible if a formal semantics is available. Specifications are often not used by only one user. They are communicated to other users as well. It is important that all users of the specification and the tools that are used in connection with such a specification have the same interpretation of the specification.

The development of a formal semantics definition requires the selection of an appropriate model. As the language MSC is used to describe the communication between distributed systems, it seems reasonable to restrict the choice for the semantics of MSC to well-known models of concurrency. Without attempting to be complete, these include Petri nets [Rei85], Mazurkiewicz traces [Maz88], event structures [Win87], labeled transition systems [Kel76], Büchi automata [Tho90], process algebras [Mil80, Hoa85, BW90], stream functions [Bro85], and I/O automata [Lyn96]. There exist many classifications of models of concurrency based on different criteria. A frequently used division of models of concurrency is into *total order* and *partial order* models.

In total order models all executions of actions are totally ordered in time. As a consequence there are no unrelated actions at the same time. In most total order theories concurrency is modeled by means of a notion of alternative composition and a notion of sequential composition. Partial order models do allow the simultaneous execution of events. Examples of partial order models are the already mentioned Petri nets, Mazurkiewicz traces, and event structures.

For the formal semantics of MSC92 the process algebra approach was chosen by study group 10 question 9 of ITU. There are several reasons for this choice:

- the process algebra semantics of the language Interworking was easy to understand and written down elegantly;
- the composition mechanisms interworking sequencing and interworking merge used in the semantics of Interworking seemed a reasonable basis for the development of similar operators for MSC;
- a clear commitment of the Formal Methods Group of Eindhoven University of Technology and of Philips Research Laboratories Eindhoven to actively participate in the standardization committee was given.

As mentioned before this has led to the standardization of a formal semantics of MSC92. This thesis reports on the research that has been performed in order to define a semantics for the language MSC96.

Message Sequence Chart in the life cycle

So far, we have only discussed the way in which the language MSC evolved in the past few years and in what way it has been influenced by the development of the language Interworking. Now, we will discuss the use of MSC in practice. MSC can be used in many phases of the software development process. Judging from the literature, especially the SDL Forums held biannually since 1981, we believe that the language MSC is most frequently used for requirements specification, visualization and simulation, verification and validation purposes, and the description of test cases. In Figure 1.7 this use of MSC is projected onto the well-known V-model. In this figure, time is going from left to right and the level of detail increases downwards.

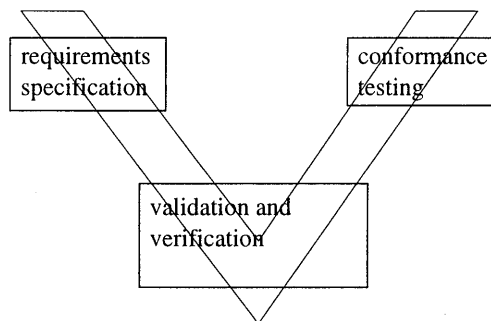


Figure 1.7: The use of MSC projected onto the V-model.

Requirements specification When initiating the development process for a system, one of the first things to do is to describe the behavior of the system on a high level of abstraction. The systems that are developed in the telecommunications industry are complex. Therefore, it is most often not feasible to give complete specifications of the system. In order to express at least some of the required behaviors of the system, scenarios can be given using the language MSC. In later stages of the development process, these MSCs can be used again for conformance testing, i.e., checking whether the system meets the requirements. In the literature several authors have indicated this use of MSC to be valuable in the system development process [Tak93, HBM93, Hau95, AN95, FMMvW98].

Related to the use of MSC for the description of requirements is the use of MSC in connection with use cases [JCJO92]. In the Unifying Modeling Language (UML) [BJR96] and related object oriented methods, use cases are a key ingredient in the development process. Use cases describe the typical interactions between the user and a system. Most of the time, use cases are simply described in natural language. In UML, sequence diagrams are used for visualizing a single use case. These sequence diagrams are very close to MSC. Currently, there is a debate on using MSCs for the formalization of use cases [AB95].

Visualization and simulation Because of their emphasis on communication and not on internal process behavior MSCs are very useful for presenting an overview of the

communication between the processes in a distributed system as for example results from a simulation. Examples of this use of MSC are the SDL Design Tool (SDT) by Telelogic [Tel96] and Object Geode by Verilog [Ver96] in the context of SDL, and the model checker Spin by Holzmann [Hol91]. In [TGH95], MSCs are used to visualize the execution sequences that result from partial order simulations of SDL descriptions. For this rather informal way of using MSC the language as it has been defined in the 1992 recommendation already contains sufficient features.

Validation and verification It is common practice to use MSCs for validation purposes. In such a case, an MSC is drawn and it is checked that the system (which is described in some formal language, for example SDL) can execute this sequence of events. Typically, MSCs which are generated during requirements specification can be used for this purpose.

In [Ek93], an algorithm is presented to perform a consistency check between an MSC and an SDL description. An MSC is considered consistent with an execution of an SDL description if the SDL execution contains all events that are contained in the MSC and the sequence of these events in the SDL execution is consistent with the partial ordering of these events as described by the MSC [Ren96a]. This algorithm is based on state space exploration techniques [Hol91] and has been implemented in the tool SDT. The notion of consistency between an SDL description and an MSC is also discussed in [Nah91].

Conformance testing In contrast to verification, which considers internals of the system, conformance testing performs a functional black box testing. The following explanation is mainly based on [Tog95]. Conformance testing is used when the internal structure of an implementation is not known or not known in full detail. A typical situation in which conformance testing is applied is the following. Suppose a manufacturer decides to develop a product for which it is required that it conforms to a standard – a situation which applies often in the telecommunications industry. In this case the specification is the set of requirements put down in the standard and the implementation is the product developed by the manufacturer. Through conformance testing it is checked if the product conforms to the requirements put down in the specification.

In conformance testing the expected behavior, in terms of observable events of the implementation, is described in a test suite, i.e. a set of test cases. A test case describes a tree of observable events and to each path in the tree it assigns a verdict which specifies whether the described behavior is correct or incorrect. Execution of the test case results in feeding the implementation with inputs and observing the generated observable events. This execution sequence of the implementation is then compared with the test case. The verdict of the corresponding path in the test tree is the outcome of the test execution.

Conformance testing consists of three tasks: the specification of the test suite, the execution of the test case, and the analysis of the verdicts. The first task describes what is expected of the conformance test, the second task is actually executing the test

cases on the implementation, and the third task is comparing the expected behavior with the actual behavior and obtaining a conclusion from the comparison. One of the biggest problems in conformance testing is obtaining a test suite from the specification. In the ITU recommendation Z.500 [IT97], this can be subdivided into two steps. The first step is the identification of a test purpose and the second is the derivation of a test suite.

A test purpose is a statement that describes what is to be tested. Test purposes should identify the 'important' behavior of the specification. Frequently used formalisms for the formalization of test purposes are temporal logic formulas and Message Sequence Charts. The use of MSC for the description of test purposes is advocated by the method SaMsTaG [GHN93, Gra94, Nah94, GSDH97] which has been developed at University of Berne in association with the Swiss PTT. A test purpose is a statement indicating what is to be tested.

From the specification and the test purposes the test suite is derived. A test case specifies all sequences of events that must be observed in order to achieve the test purpose (the test body). Furthermore, it specifies at least one sequence that leads the implementation under test (IUT) from the initial state to the initial state of the test purpose (the test preamble), and at least one sequence that leads the IUT back to the initial state (test postamble). For the (semi-)automated derivation of test cases it is important that the test purpose is formalized first. In the SaMsTaG method a complete test case can be generated from a system specification in SDL and a test purpose description in MSC. The test case is described using the Tree and Tabular Combined Notation (TTCN) [ISO91a]. A similar approach is followed by the HARPO toolkit [AMPV97, PAM97].

Among others the papers [GHNS95, FJ96, CLM97] use MSC for the description of test cases. In [EFM97] synchronous sequence charts, i.e. Interworkings, are used for this purpose. Another paper using MSC for testing is [SST97].

Related formalisms

The language MSC is a member of a large class of similar graphical notations, most of which are only defined informally. Examples are the previously mentioned Extended Sequence Charts and Interworkings, and also Siemens-SCs [Sie92], Time Sequence Diagrams (TSD) [ISO87, CCI88b, ISO91b, Fac95b], Information Flow Diagrams [CCI89], Message Flow Diagrams [CCHvK90], Arrow Diagrams [CCI88a], and use case diagrams [JCJO92, AB95]. Below we discuss some of these.

Time Sequence Diagrams Time Sequence Diagrams are a graphical representation employed to clarify the communication between service users and a server provider in the ISO/OSI basic reference model. Time Sequence Diagrams are a semi-formal means of describing a property of a service specification. Time Sequence Diagrams are defined in [ISO87, CCI88b] by means of some examples. Facchi has provided this formalism with a formal semantics [Fac95a]. With Time Sequence Diagrams timing precedences between service primitives, or events, can be expressed. Service

Access Points (SAP) are separated by means of vertical lines. These vertical lines also indicate an increase in time downwards. Two service primitives at different SAPs can be related by a solid line, which indicates an ordering in time (see Figure 1.8). If two events are connected by means of a wavy line, then these are not related with respect to time. A TSD expression describes a set of traces of events.

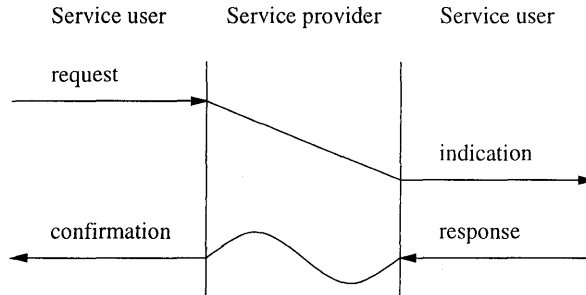


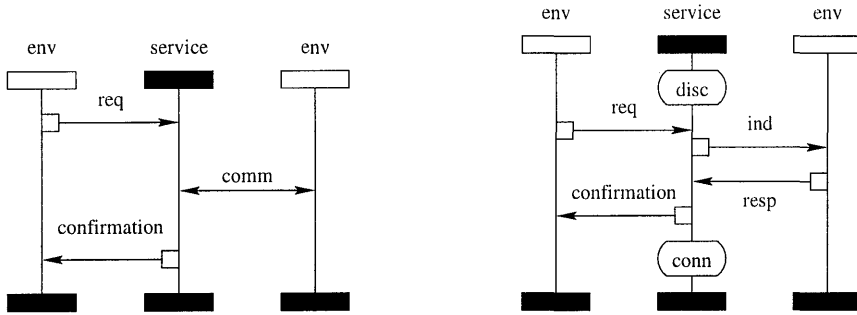
Figure 1.8: Example of a Time Sequence Diagram.

A service specification is described by means of a set of Time Sequence Diagrams. Therefore, the combination of Time Sequence Diagrams must be defined. These composition mechanisms are sequential composition, interleaving, conjunctive composition, alternative composition and repetition. The conjunctive composition of two TSDs gives all traces that are a trace of both TSDs. The repetition operator describes any arbitrary, possibly infinite fair interleaving of the TSD it is applied to.

Extended Sequence Charts Extended sequence charts are MSCs extended with some SDL symbols. A typical use of these ESCs is in a four phase refinement method to develop SDL specifications [GR89]. Starting from an ESC standard form (only instances, messages, and bidirectional arrows representing dialogs), a first refinement is to obtain an ESC in state form. This is an ESC extended with among others SDL state symbols. In this refinement step also the dialogs must be resolved by asynchronous communications. In the second refinement step an MSC in state input form is obtained by adding SDL input symbols to model the way in which SDL treats communication (an SDL input symbol represents the receipt of the message in the input queue of the instance). A final refinement step is to translate this state input form into SDL process descriptions. A simple example of the first three phases is given in Figure 1.9.

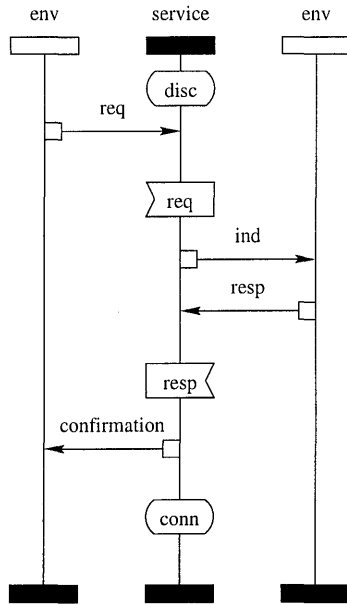
Structure of this thesis

This thesis is structured as follows. In Chapter 2, the language MSC as it is standardised by the ITU in October 1996 is introduced. The language elements are introduced piecewise and for each language element the graphical syntax is explained and illustrated, an informal explanation of the semantics is provided, and the textual syntax is explained and illustrated. This chapter presents the language MSC in an informal but nevertheless rigorous way. After finishing this chapter the reader should have a



(a) standard form

(b) state form



(c) state input form

Figure 1.9: The refinement method of Extended Sequence Charts.

good impression of the possibilities offered by the language MSC and of the meaning of an MSC diagram.

The core language of MSC, called *Basic Message Sequence Chart* (BMSC), is introduced first. A Basic Message Sequence Chart consists of communications between instances and local actions on instances only. In most languages comparable to MSC these features are present. Second, the additional basic concepts are introduced. These are the following: process creation and termination, timer events, incomplete messages and conditions. After these, the means offered by MSC to describe the ordering of events other than the orderings imposed via communication and the total ordering of events on an instance, are introduced. These are the core region for interleaved execution of events on an instance and causal orderings for the description of orderings between events (possibly from different instances). Then, the means offered by the language MSC to describe more complex systems are explained. These are inline expressions, MSC reference expressions and High-level Message Sequence Charts. All of these describe the composition of MSCs by means of operators. They differ in their graphical appearance and possibilities for application. Finally, the extension with gates is given. In connection with the composition mechanisms offered by MSC gates are used to describe communication between instances in different MSCs and the causal ordering of events on instances in different MSCs.

In Chapter 3 the process theory is developed that is used for the definition of the formal semantics of MSC in Chapter 4. This process theory consists of terms constructed from constant and function symbols (operators). The constants represent the events of the MSCs and the operators are used to describe the composition of these into more complex processes. The operators that are used are all close to the composition mechanisms of the language MSC. For example, the delayed choice operator defined in Section 3.4 represents the alternative composition of MSCs. The operators used in this thesis are all based on operators that can be found in the literature. By means of Plotkin-style deduction rules the dynamic behavior of processes described by these terms is defined in terms of the subsequent execution of actions (events). A notion of equivalence for the processes is defined similar to the notion of (strong) bisimilarity. Several equations between processes are given and these are proven bisimilar in Appendix B.

In Chapter 4 a denotational semantics of MSCs is defined. It consists of mappings $[[\]]$ which transform (a part of) an MSC in textual representation into a process term. The textual representation of MSC is reduced for the purpose of a concise description of the semantics definition. The reasoning that led to this reduced textual syntax and the resulting textual syntax are given in Appendix A. In the final section of this chapter we give a short review of some other approaches to the semantics of MSC.

2

Introduction to the language Message Sequence Chart

2.1 Introduction

In this chapter, the language Message Sequence Chart is introduced. First, the core language of Message Sequence Chart is introduced. This core language is called *Basic Message Sequence Chart* (BMSM). A Basic Message Sequence Chart is concerned with communications and local actions only. These are the features encountered in most languages comparable to Message Sequence Chart. The static requirements imposed on Basic Message Sequence Chart, as far as they are of importance to the definition of the formal semantics in Chapter 4, are given. These static requirements are not formalized. For MSC92 the static requirements are formalized in Annex C to recommendation Z.120 [Ren95a, Ren96b, IT96a].

Then, the additional basic concepts are introduced. These are process creation and termination, timer handling, incomplete message events and conditions. Next, core-gions and causal orderings are introduced. These are the means offered by MSC to describe ordering of events other than the orderings imposed via communication and the total ordering of events on an instance. Then, the more intricate possibilities of describing complex systems are considered. These are inline expressions, MSC reference expressions and High-level Message Sequence Charts. Finally, the extension of the language MSC with gates is given.

2.2 Basic Message Sequence Charts

A Basic Message Sequence Chart contains a finite collection of instances. An instance is an abstract entity on which message outputs, message inputs and local actions may be specified. The user of the language should determine for himself which instances have to be included in the description. This depends amongst others on his intentions. The collection of instances can, for example, reflect the physically available system components or, in other situations, supposedly logically based partitionings of the system. A first example of a Basic Message Sequence Chart is given in Figure 2.1. The vertical lines labeled by i_1, \dots, i_4 describe the instances of the MSC progressing in time from top to bottom. The arrows labelled by $m_0 \dots m_4$ between the instances describe messages that are exchanged. The frame surrounding the instances describes the border of the system. This frame is also used to describe interaction with the environment. For example, message m_0 is sent to the environment by instance i_1 . The box labelled by a denotes internal activity of instance i_2 .

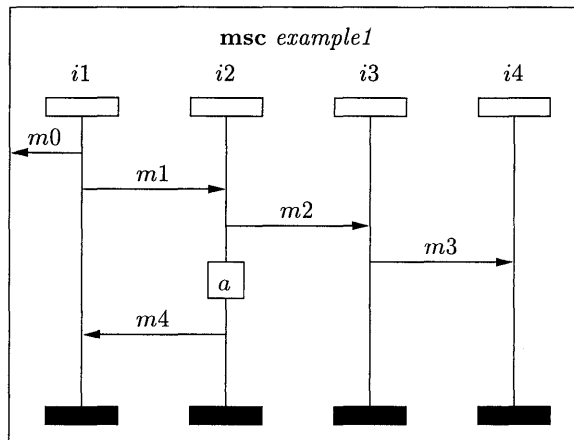


Figure 2.1: Example Basic Message Sequence Chart.

Next, we explain the graphical representation of Basic Message Sequence Chart. Then we describe their meaning, and finally we introduce the textual representation.

2.2.1 Graphical representation

Graphically an MSC is given by a frame containing a graphical representation of the instances. The name of the MSC following the keyword **msc** is placed inside this frame, usually above the instances. For an example see Figure 2.1.

In the graphical representation there are two ways to describe an instance. These are given in Figure 2.2 below. The first is a single vertical axis (line-form) and the second is the so-called column-form. The description of the instance starts with the

instance head symbol and ends with the *instance end* symbol. These do not describe creation and termination of the instance, but the start and end of the description. The representation of the instance and the instance head and instance end symbols should be aligned as indicated in Figure 2.2. Within one BMSC both representations of instances, line-form and column-form, may appear.

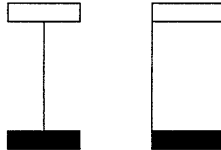


Figure 2.2: Instances: line-form and column-form.

To every instance an *instance heading* is associated. It consists of an *instance name* and optionally an *instance kind*. The instance kind consists of an optional *kind denominator* and an *identifier*. The kind denominator can be either one of the keywords **block**, **process**, **service** or **system** or an arbitrary *name*. The predefined kind denominators refer to the corresponding SDL constructs [IT94]. The identifier can be used to associate with an instance the name of the entity it represents. Semantically no meaning is attached to the use of these kind denominators and instance kinds.

The instance head may be placed above or inside the instance head symbol. It is also allowed to place the instance name inside the instance start symbol and the kind denominator above. In Figure 2.3 some valid placements are given where *i* is the instance name, *d* is the identifier of the instance kind, and **process** is the kind denominator of the instance kind. Between the instance name and the instance kind a colon may be placed. If the instance name is placed inside the instance start symbol and the instance kind is placed above, the colon should appear above the instance start symbol. In this thesis, we will only use the instance name. In all MSCs we place it above the instance head symbol.

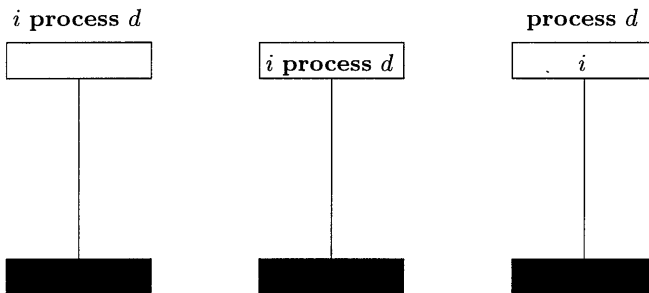


Figure 2.3: Placements of instance head.

Instances are referred to by means of the instance name. Therefore, the instance name must be unique within an MSC.

A local action is denoted by an *action* symbol on the axis with the *action character string* placed in it. A local action describes an internal activity of an instance. The action character string is an informal description for this internal activity. The precise syntax for action character strings is considered irrelevant in this thesis. When an action symbol is placed on an instance in line-form the instance axis is hidden. If the column-form is used for an instance, the width of the action symbol must coincide with the width of the column-form of the instance. Multiple actions on an instance must not overlap. See Figure 2.4 for examples.

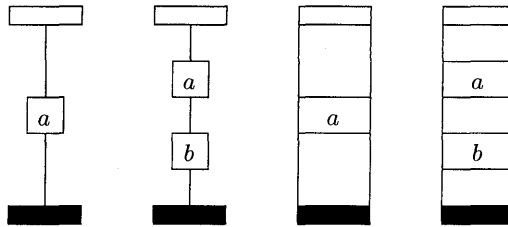


Figure 2.4: Placement of local actions on line-form and column-form instances.

A message between two instances is represented by an arrow which starts at the sending instance and ends at the receiving instance. An arrow representing a message may be horizontal or with downward slope. A message sent by an instance to the environment is represented by an arrow from the sending instance to the exterior of the Message Sequence Chart, i.e. the surrounding frame. A message received from the environment is represented by an arrow from the exterior of the Message Sequence Chart to the receiving instance. To every message a *message identification* is associated. It consists of a *message name* and optionally a *message instance name* and/or a *parameter list*. A message name and a message instance name are separated by a comma. The message instance name is used in the textual syntax for distinguishing between multiple occurrences of the same message name. The parameter list is denoted between brackets after the message name and possibly the message instance name. The message identification should be placed close to the message arrow. In Figure 2.5 some examples of the graphical syntax of messages are given. In this thesis, we will only use the message name.

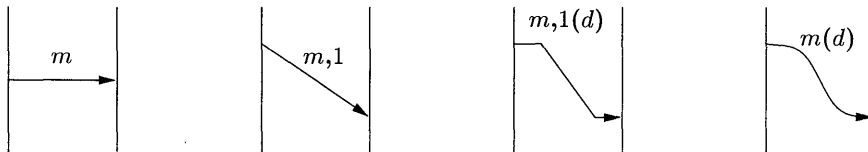


Figure 2.5: Appearance and placement of messages.

In general it is not allowed to have two or more events attached to one point of the instance axis in line-form and column-form or at the same height of the instance axes in the column-form. However, there is one exception to this rule. An *incoming event* and an *outgoing event* may be attached to the same point or at the same height. This

is interpreted as if the incoming event is drawn above the outgoing event. A message input event is an incoming event and a message output event is an outgoing event. Also the yet to be introduced found message events and timeout events are considered to be incoming events. The yet to be introduced lost message events, process creation events and timer set and reset events are outgoing events. In Figure 2.6 some examples are given.

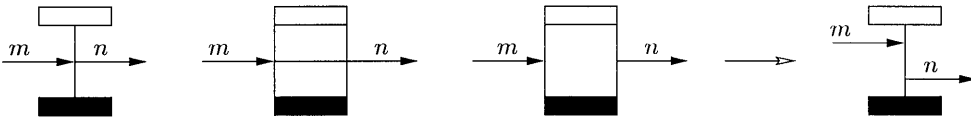


Figure 2.6: Interpretation of events at the same height.

With the use of the column-form instance the suggestion is put forward that each of the two vertical lines describes a total ordering of the events attached and that the events attached to different vertical lines are not ordered. This, however, is not the case. This is depicted in Figure 2.7 below. It is possible to draw events on an instance for which it can hardly be seen that they are not drawn at the same height.

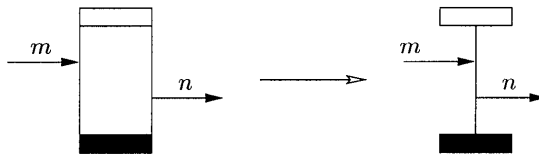


Figure 2.7: Interpretation of orderings on an instance in column-form.

2.2.2 Intuitive semantics

An MSC is intended to describe a number of executions of the events contained. As we have seen before these events can be local actions, message outputs and message inputs. An MSC does not only describe the events to be executed, it also contains information concerning the sequences in which they may be executed. One of the basic assumptions is that all events are executed instantaneously, i.e. it is assumed that the execution of an event consumes no time. Another important assumption is that no two events can be executed at the same time.

As explained before, an MSC consists of a number of instances on which events are specified. The meaning of such an instance is that it executes the events specified in the same ordering as they are given on the vertical axis from top to bottom. Thus one can say that the time along each instance axis is running from top to bottom. Therefore, the events specified on an instance are totally ordered in time. If we consider, for example, instance i_2 from the MSC given in Figure 2.1, then this means that instance i_2 executes the events “input of m_1 from instance j ”, “output of m_2 to instance i_3 ”, “action a ”, and “output of m_4 to instance i_1 ”, and also that these events

are executed in this order. Although an instance thus describes the execution of events while time progresses, the instance does not specify the elapse of time in between two consecutive events. It might be the case that the first event is executed after 5 minutes and that the second event is executed after 25 minutes. But alternatively, it could also be the case that the second event only happens somewhere next year.

The instances of an MSC operate independently of each other. No global notion of time is assumed. The only dependencies between the timing of the instances come from the restriction that a message must be sent before it is consumed. In Figure 2.1 this implies that message $m3$ is received by $i4$ only after it has been sent by $i3$, and, consequently, after the consumption of $m2$ by $i3$. Thus the events concerning $m1$ and $m3$ are ordered in time, while for the events of $m4$ and $m3$ no ordering is specified apart from the requirement that the output of a message occurs before its input. Because of the asynchronous communication, it would even be possible to first send $m3$, then send and receive $m4$, and finally receive $m3$. The execution of a local action is only restricted by the ordering of events on the instance it is defined on. The Basic Message Sequence Chart in Figure 2.8 defines the same execution sequences (from a semantic point of view) as the BMSC in Figure 2.1, but in an alternative drawing.

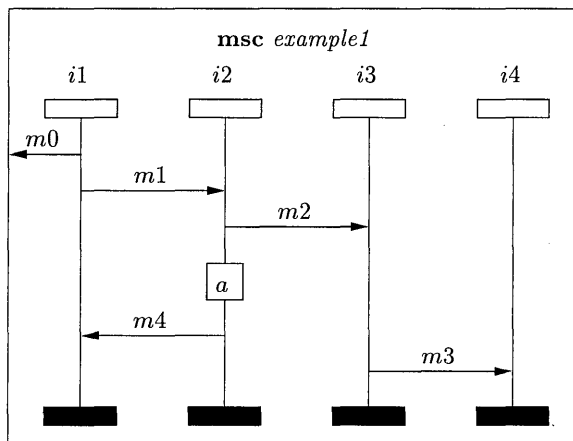


Figure 2.8: Example Basic Message Sequence Chart.

In MSC there is no notion of channels or buffering of communication. In [EMR97], a number of communication models for MSC is considered. The communication model adopted by recommendation Z.120 (without saying so) is that for every message there is an unbounded FIFO queue.

Another consequence of this mode of communication is that overtaking of messages is allowed, as expressed in Figure 2.9.

It is not allowed that a message output is causally depending on its corresponding message input, directly or via other messages [IT96b, IT96a, Ren95a]. This is the case if the temporal ordering of the events imposed by the Basic Message Sequence Chart

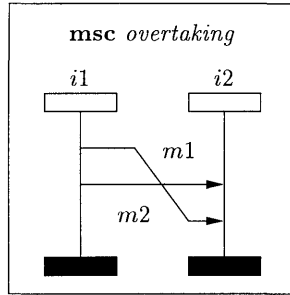


Figure 2.9: Basic Message Sequence Chart with overtaking.

specifies that a message input is executed before its corresponding message output. Such MSCs are often called *inconsistent*.

Consider the first diagram in Figure 2.10. Since the events which are specified on one instance are temporally ordered from top to bottom, the message input is executed before the corresponding message output. The diagram therefore violates the static requirement of consistency. In this example the message output is depending on its corresponding message input in a direct way.

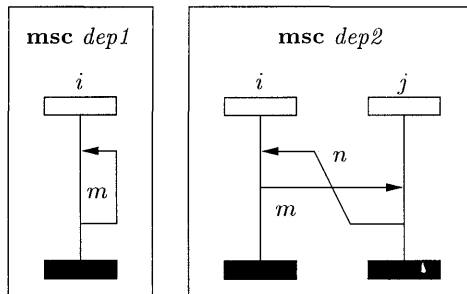


Figure 2.10: Two diagrams that violate the static requirements.

As an example of the indirect causal dependency between a message output and a message input the second diagram in Figure 2.10 is considered. Amongst others, there are the following temporal orderings:

1. the input of message *m* precedes the output of message *n*,
2. the output of message *n* precedes the input of message *n*, and
3. the input of message *n* precedes the output of message *m*.

Therefore, the diagram specifies that the input of message *m* precedes the output of message *m*. So the diagram violates the static requirements, and is therefore not

a (Basic) Message Sequence Chart. The drawing rules for messages [IT96b] already exclude this MSC. At this point it is impossible to draw an MSC which is inconsistent and does not violate the drawing rules. This example is given here because later, when causal orderings are introduced in Section 2.4.2, the drawing rules no longer prohibit the drawing of inconsistent MSCs.

2.2.3 Textual representation

Although the application of Message Sequence Charts is mainly focussed on the graphical representation, they have a concrete textual syntax. This representation was originally intended for exchanging Message Sequence Charts between computer tools only, but in this thesis it is used for the definition of the semantics.

With respect to the textual description, the language MSC offers two principal means to describe MSCs. First of all an MSC can be described by describing the behavior of all its instances in isolation. This way of describing an MSC is called *instance-oriented* and has been incorporated in the language from the beginning. In instance-oriented representation, the MSC from Figure 2.1 can be given by

```

msc example1 ;
  instance i1 ;
    out m0 to env ;
    out m1 to i2 ;
    in m4 from i2 ;
  endinstance ;
  instance i2 ;
    in m1 from i1 ;
    out m2 to i3 ;
    action 'a' ;
    out m4 to i1 ;
  endinstance ;
  instance i3 ;
    in m2 from i2 ;
    out m3 to i4 ;
  endinstance ;
  instance i4 ;
    in m3 from i3 ;
  endinstance ;
endmsc ;

```

In this example the instances are described in the order in which they are represented in the MSC. This is not required by the recommendation.

With the appearance of MSC96 also another way of representing MSCs has been incorporated: the so-called *event-oriented* description [Ek94]. With the event-oriented descriptions, just a list of events is given, for example as they are expected to occur in a trace of the system or as they are encountered while scanning the MSC from

top-to-bottom. Besides these two ways of describing MSC there is also the ‘artificial’ means to describe an MSC by mixing these two descriptions. An example of such a mixing is the following:

```
msc mixedsyntax ;  
i : instance ;  
i : action 'a' ;  
    instance k ;  
        out m to i ;  
    endinstance ;  
i : in m from k ;  
i : endinstance ;  
endmsc ;
```

In this example, instance *i* is described in event-oriented syntax and instance *k* in instance-oriented syntax.

In this thesis, the event-oriented textual syntax is used for the explanation of the textual syntax in this chapter and for the definition of a formal semantics in Chapter 4. There are several reasons to choose for the event-oriented syntax and not for the instance-oriented syntax. First of all, with the event-oriented syntax the so-called multi-instance events, i.e. events that are described/defined for a number of instances, need to be repeated for every instance they are defined on in the instance-oriented syntax. In the event-oriented syntax there is the possibility to describe them once for all instances involved. A second reason is related to the first one. For multi-instance events in the instance-oriented syntax, there necessarily are a lot of requirements which have to guarantee that a multi-instance event is defined for all involved instances. This is not necessary in the event-oriented syntax, as it is per definition defined for all instances involved.

The event-oriented syntax of MSC facilitates the description of the events contained in an MSC not per instance but in an arbitrary ordering which respects the ordering of events per instance. As a consequence it is even possible to describe a message input event before its corresponding message output event provided that these are attached to different instances. Although this seems counterintuitive there is a good reason to allow this (see Section 2.4.1).

The textual representation of an MSC consists of the keywords **msc** and **endmsc** and in between those an *msc head* and an *msc body*. The MSC head contains the name of the MSC, the *msc name*, and optionally an *msc inst interface*, which contains a list of instance names. The MSC body is defined differently for the three previously mentioned description styles. In the event-oriented syntax an MSC body consists of a (possibly empty) list of MSC statements. An MSC statement consists of an event definition. An event definition is an instance name followed by a non-empty list of instance events. Instance events are message events, local actions, and instance head and end statements. The latter two are the textual counterparts of the instance head and instance end symbol from the graphical syntax.

Textually a message between instances is described by a message output event and a message input event. The sender of message m is the instance with which the message output event is associated and the receiver of message m is the instance with which the message input event is associated. In the textual syntax for message output events the receiver of the message is called the *input address* and the sender the *output address*. If m is a message that is sent from instance i to instance j , textually the corresponding event definitions for the message output event and the message input event are “ $i : \text{out } m \text{ to } j$ ” and “ $j : \text{in } m \text{ from } i$ ”.

A message that is sent to the environment is described by a message output event only. For example a message m from instance i to the environment is described as “ $i : \text{out } m \text{ to env}$ ”. In this case the environment plays the role of output address of message m . Similarly a message that is received from the environment is described by the message input event only. For example, the receiving of a message m by instance j from the environment is described as: “ $j : \text{in } m \text{ from env}$ ”. The environment is considered the output address of the message.

In the graphical representation the correspondence between message output events and message input events is given by the arrow construction. In the textual representation a message output event and a message input event are corresponding if and only if

- the events have the same message identification,
- the instance on which the message output event is specified is the same as the instance indicated by the output address of the message input event, and
- the instance on which the message input event is specified is the same as the instance indicated by the input address of the message output event.

Thus a natural requirement on the textual representation of MSC is that for every message output event there is at most one corresponding message input event, and vice versa, for every message input event there is at most one message output event. As no dangling message output arrows and message input arrows are allowed, another natural requirement is that for every message output (input) event there is at least one corresponding message input (output) event. Note that for messages that are sent to the environment or that are received from the environment and for lost and found messages (see Section 2.3.3) this requirement does not have to be satisfied.

A local action is denoted by the keyword **action** followed by an action character string.

The MSC from Figure 2.1 can textually be represented by

```

msc example1 ;
i1 : instance ;
i2 : instance ;
i3 : instance ;

```

```
i4 : instance ;  
i1 : out m0 to env ;  
i1 : out m1 to i2 ;  
i2 : in m1 from i1 ;  
i2 : out m2 to i3 ;  
i3 : in m2 from i2 ;  
i3 : out m3 to i4 ;  
i4 : in m3 from i3 ;  
i2 : action 'a' ;  
i2 : out m4 to i1 ;  
i1 : in m4 from i2 ;  
i1 : endinstance ;  
i2 : endinstance ;  
i3 : endinstance ;  
i4 : endinstance ;  
endmsc ;
```

The grammar defining the event-oriented textual syntax of Basic Message Sequence Charts is given in Table 2.1. In this thesis, textual syntax is defined by means of rules in *Backus-Naur Form* (BNF). Nonterminals are indicated by putting them in between \langle and \rangle . Terminal productions are typeset using bold face. Words that are typeset using bold face are considered reserved keywords. A sequence of terminals or nonterminals denotes concatenation. Parts in between $[$ and $]$ are optional. The symbols $\{$ and $\}$ are used for grouping. Alternative productions for a nonterminal can be given using the symbol $|$. Repetition is indicated by means of $*$ or $+$. The symbol $*$ indicates that the group can be repeated any number of times (including zero) and the symbol $+$ indicates that the group is repeated at least once. The symbol $\langle \rangle$ denotes the empty string. If for one nonterminal more than one BNF rule is given, these should be considered alternative productions.

The nonterminals that end with the word 'name', for example $\langle \text{msc name} \rangle$, are all defined to be the same as $\langle \text{name} \rangle$. They are included to indicate a semantical interpretation, e.g., $\langle \text{msc name} \rangle$ represents the name of an MSC. Although recommendation Z.120 gives a number of BNF rules for the nonterminal $\langle \text{name} \rangle$, we will only mention that the productions of this nonterminal consist of sequences of letters, digits, several brackets and other symbols. For the explanation offered in this chapter it is only relevant to understand that it defines the name space for the different identifiers. The nonterminal $\langle \text{action character string} \rangle$ denotes a sequence of symbols between apostrophes. The action character string may contain spaces. Its precise definition is not interesting here.

The occurrence of an instance name preceding an instance head statement is considered to be the defining occurrence of that instance name. As was already indicated for the graphical syntax, there cannot be two or more instances in the MSC with the same instance name. Translated to the textual syntax this means that there cannot be two or more defining occurrences of an instance name. For each defining occurrence of an instance name there must be exactly one instance end statement preceded by that same instance name. Furthermore, that instance end statement must not

<p> $\langle \text{message sequence chart} \rangle ::= \text{msc } \langle \text{msc head} \rangle \langle \text{msc body} \rangle \text{endmsc } \langle \text{end} \rangle$ $\langle \text{msc head} \rangle ::= \langle \text{msc name} \rangle \langle \text{end} \rangle [\langle \text{msc interface} \rangle]$ $\langle \text{msc body} \rangle ::= \langle \text{msc statement} \rangle^*$ </p>
<p> $\langle \text{msc statement} \rangle ::= \langle \text{event definition} \rangle$ $\langle \text{event definition} \rangle ::= \langle \text{instance name} \rangle : \langle \text{instance event list} \rangle$ $\langle \text{instance event list} \rangle ::= \{ \langle \text{instance event} \rangle \langle \text{end} \rangle \}^+$ $\langle \text{instance event} \rangle ::= \langle \text{orderable event} \rangle \langle \text{non-orderable event} \rangle$ $\langle \text{orderable event} \rangle ::= \langle \text{message event} \rangle \langle \text{action} \rangle$ $\langle \text{non-orderable event} \rangle ::= \langle \text{instance head statement} \rangle \langle \text{instance end statement} \rangle$ </p>
<p> $\langle \text{message event} \rangle ::= \langle \text{message output} \rangle \langle \text{message input} \rangle$ $\langle \text{message output} \rangle ::= \text{out } \langle \text{msg identification} \rangle \text{ to } \langle \text{input address} \rangle$ $\langle \text{message input} \rangle ::= \text{in } \langle \text{msg identification} \rangle \text{ from } \langle \text{output address} \rangle$ $\langle \text{msg identification} \rangle ::= \langle \text{message name} \rangle [, \langle \text{message instance name} \rangle]$ $\quad [(\langle \text{parameter list} \rangle)]$ $\langle \text{parameter list} \rangle ::= \langle \text{parameter name} \rangle [, \langle \text{parameter list} \rangle]$ $\langle \text{output address} \rangle ::= \langle \text{instance name} \rangle \text{env}$ $\langle \text{input address} \rangle ::= \langle \text{instance name} \rangle \text{env}$ $\langle \text{action} \rangle ::= \text{action } \langle \text{action character string} \rangle$ </p>
<p> $\langle \text{instance head statement} \rangle ::= \text{instance } [\langle \text{instance kind} \rangle]$ $\langle \text{instance end statement} \rangle ::= \text{endinstance}$ </p>
<p> $\langle \text{msc interface} \rangle ::= [\langle \text{msc inst interface} \rangle]$ $\langle \text{msc inst interface} \rangle ::= \text{inst } \langle \text{instance list} \rangle \langle \text{end} \rangle$ $\langle \text{instance list} \rangle ::= \langle \text{instance name} \rangle [: \langle \text{instance kind} \rangle]$ $\quad [, \langle \text{instance list} \rangle]$ $\langle \text{instance kind} \rangle ::= [\langle \text{kind denominator} \rangle] \langle \text{identifier} \rangle$ $\langle \text{kind denominator} \rangle ::= \text{system} \text{block} \text{process} \text{service} \langle \text{name} \rangle$ </p>
<p> $\langle \text{end} \rangle ::= ;$ </p>

Table 2.1: The event-oriented textual syntax for Basic Message Sequence Charts.

occur before the instance head statement in the MSC. Instance names are used to associate events to instances in event definitions. All such occurrences of an instance name must be in between the corresponding instance head statement and instance end statement.

The instance names are used in output and input addresses of message events. For each such a reference there must be a defining occurrence of that instance name.

The textual representation of an MSC can contain an MSC instance interface. This MSC instance interface contains a number of descriptions of the instances as they are defined in the MSC body. It is required that the information presented in the MSC instance interface is consistent with the information provided in the MSC body by means of the instance head statements [IT96a]. In this thesis, the MSC instance interface will not be used.

2.3 Additional basic concepts

In this section, the other basic concepts are introduced. These are process creation and termination, timer handling, incomplete message events and conditions.

2.3.1 Process creation and process termination

In the language Message Sequence Chart a primitive is incorporated for the dynamic creation of an instance by another instance. Such a creation is denoted by a dashed arrow, the *createline* symbol, from the instance axis of the creating instance to the instance head symbol of the created instance. The createline symbol must be horizontal. An example is given in Figure 2.11. An instance can be created only once. As was the case for message events, a create event may be labelled with a parameter list. In case of a process create event the parameter list is placed close to the createline symbol. The parameter list is not enclosed in brackets (see Figure 2.11 for an example).

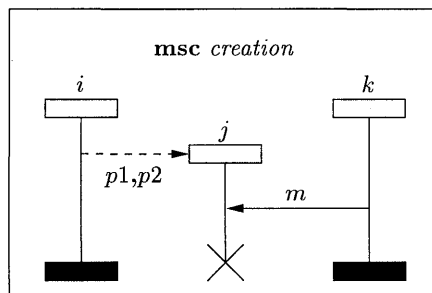


Figure 2.11: Process creation and termination.

An instance can terminate by executing a process stop event. Execution of a process stop is allowed only as the last event in the description of an instance. A process stop is denoted by replacing the instance end symbol by a cross, the *stop* symbol.

In Figure 2.11 a Message Sequence Chart with three instances is given. Instance *i* creates instance *j*, instance *k* sends a message *m* to instance *j*, and instance *j* receives the message *m* from instance *k* after it is created, and then terminates.

<code><create></code>	<code>::= create</code>	<code><instance name></code>	<code>[(<parameter list>)]</code>
<code><stop></code>	<code>::= stop</code>		
<code><orderable event></code>	<code>::=</code>	<code><create></code>	
<code><non-orderable event></code>	<code>::=</code>	<code><stop></code>	

Table 2.2: The textual syntax for process creation and termination.

In the textual representation the creation of an instance with name *j* is denoted by “**create j**” and the termination of an instance by “**stop**”. The textual grammar for Basic Message Sequence Charts in Table 2.1 is extended with the rules in Table 2.2. The event-oriented textual representation of the Message Sequence Chart in Figure 2.11 is given by

```

msc creation ;
i : instance ;
j : instance ;
k : instance ;
i : create j(p1,p2) ;
k : out m to j ;
j : in m from k ;
j : stop ;
i : endinstance ;
j : endinstance ;
k : endinstance ;
endmsc ;

```

With respect to process creation and termination the following static requirements are added. Only instances that are defined within a Message Sequence Chart may be created. An instance may be created only once and an instance may not create itself. A stop event may not be followed by other events from the same instance other than the instance end statement (which is not considered an event).

2.3.2 Timer handling

In a Message Sequence Chart several timer events can be described. These are the setting of a timer, a timer reset and the expiration of a timer.

In Figure 2.12 on instance i the setting of a timer T with duration d and its subsequent timer reset are specified, and on instance j the setting of a timer T and its subsequent timeout are specified. Timer events with the same name attached to different instances denote different timers.

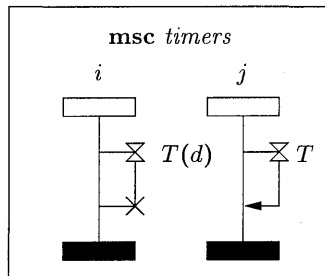


Figure 2.12: Message Sequence Chart with timer handling.

In the graphical syntax the timer events can be used stand-alone but also in combinations. We will first discuss the stand-alone occurrences of timer events. A timer set event is denoted by an *hourglass* symbol attached to the instance axis by means of a horizontal or bent line. A timer reset event is denoted by a *cross* symbol which is attached to the instance axis by means of a horizontal or bent line. A timeout is represented by an hourglass symbol which is attached to the instance axis by means of an horizontal or bent arrow from the hourglass symbol to the instance axis.

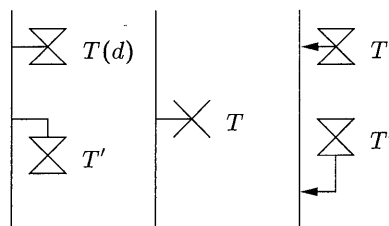


Figure 2.13: Timer events in stand-alone mode.

Examples of the stand-alone occurrences of the timer events are given in Figure 2.13. A timer event is labelled by an identifier, the *timer name* and optionally a *timer instance name*, that is placed aside the hourglass symbol or cross. The timer name and, if present, the timer instance name, are separated by a comma. We will call this combination of timer name and possibly timer instance name the *timer identifier* for easy reference. The timer instance name is used in the textual representation to be able to distinguish between timer events with the same timer name in the graphical

syntax which however are not corresponding. A timer set event may be labelled with an identifier for the duration, the *duration name*. The duration name is placed between brackets after the timer identifier.

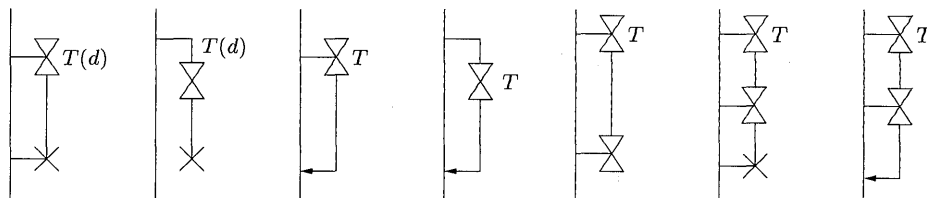


Figure 2.14: Combinations of timer events.

The graphical syntax of MSC also leaves room for combining timer events. This is achieved by connecting the timer events by means of a vertical line. Examples of such combinations are given in Figure 2.14. Two timer events are corresponding if they are associated with the same instance and they have the same timer identification. If an instance has two or more corresponding timer events then these form a timer combination. Note that for these combinations the timer identifier may be omitted from all but one timer event. A timer event is local to the instance it is specified on. It is not allowed to specify a timer set and a subsequent timeout or timer reset on different instances. On different instances the same timer identifier may be used. These indicate different timers.

The language MSC in its current form does not support the specification of a quantitative notion of time, the interpretation of the timer events is only symbolic. This means that set, reset and timeout are interpreted as events. Also, as no formal data language is available at the moment, the duration names that can be associated with a timer set event are symbolic. The user can write down any identifier there.

With industrial applications in mind, especially telecom applications, this is not satisfactory. The extension of the language MSC with a quantitative notion of time is investigated by several researchers [Men93, Sch95, AHP96, BJR96, BL97] and the extension of MSC with time is considered an important issue by the MSC standardization committee of the ITU. The approaches vary from associating time intervals with events, message delivery intervals with messages, time intervals with two consecutive events from the same instance, and time intervals with two arbitrary events in the MSC. In [AHP96] and [BL97] timing analysis is performed to determine if an MSC is timing consistent. An MSC is considered timing consistent if it is possible to give a timing assignment to the events that satisfies the timing constraints.

In the textual representation a timer set event is denoted by the keyword `set` followed by the timer identifier and optionally followed by the duration name between brackets. For example, the setting of a timer with timer name T , timer instance name t and duration name d on an instance i is represented textually by the event definition " i : `set T,t(d)`". A reset event and a timeout event are represented similarly apart from that these cannot have a duration name. The reset and timeout events that

correspond to the timer set event “ $i : \text{set } T, t(d)$ ” are given by “ $i : \text{reset } T, t$ ” and “ $i : \text{timeout } T, t$ ”, respectively. The grammar in Table 2.2 is extended with the rules in Table 2.3.

$\langle \text{timer statement} \rangle$	$::=$	$\langle \text{set} \rangle \mid \langle \text{reset} \rangle \mid \langle \text{timeout} \rangle$
$\langle \text{set} \rangle$	$::=$	set $\langle \text{timer name} \rangle$ [, $\langle \text{timer instance name} \rangle$] [($\langle \text{duration name} \rangle$)]
$\langle \text{reset} \rangle$	$::=$	reset $\langle \text{timer name} \rangle$ [, $\langle \text{timer instance name} \rangle$]
$\langle \text{timeout} \rangle$	$::=$	timeout $\langle \text{timer name} \rangle$ [, $\langle \text{timer instance name} \rangle$]
<hr/>		
$\langle \text{orderable event} \rangle$	$::=$	$\langle \text{timer statement} \rangle$

Table 2.3: The textual syntax for timer handling.

The Message Sequence Chart in Figure 2.12 is represented as follows:

```

msc timer ;
i  : instance ;
j  : instance ;
i  : set  $T(d)$  ;
i  : reset  $T$  ;
j  : set  $T$  ;
j  : timeout  $T$  ;
i  : endinstance ;
j  : endinstance ;
endmsc ;

```

Recommendation Z.120 states that if the timer name is not sufficient for a unique mapping the timer instance name has to be employed and that the setting of a timer always has to precede the corresponding timer events.

2.3.3 Incomplete message events

Besides the specification of successful transmission of messages also a lost message and a spontaneously found message can be described. A lost message is a message which is sent but will never be received by the other party in the communication. Symmetrically, a found message is a message which is received but where the output is unknown. A message identification is associated with the lost and found messages similarly as to the message identification for messages.

Graphically a lost message is indicated by a *lost message* symbol, i.e. an arrow from an instance axis to a black dot (“black hole”). To the black dot an *input address* may be associated. This input address, which is either an instance name or the environment,

represents the original destination of the message. A found message is indicated by a *found message* symbol, i.e. an arrow from an open dot (“white hole”) to an instance axis. An *output address* may be associated with the open dot. This output address, which is either an instance name or the environment, is the original source of the message. An example of the graphical representation of lost and found messages is given in Figure 2.15.

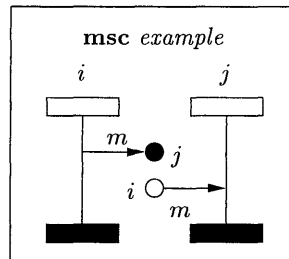


Figure 2.15: An MSC with lost and found messages.

Semantically these events are treated just as atomic events. It is not the case that a dynamic semantics is associated with messages such that they can result in lost and/or found messages. Thus these events are introduced to describe the situation where it is known that a message is lost or found.

Consider the MSC from Figure 2.15. On instance *i* the sending of a message *m* with destination *j* is described. However the corresponding receive event on instance *j* is missing. Similarly, instance *j* receives a message *m* which should have been sent by instance *i*, but on instance *i* the corresponding send event is missing. Although the lost and found message in this example seem to be complementary, this is not the interpretation from the recommendation. Thus the events are totally unrelated.

$\langle \text{incomplete message event} \rangle$	$::=$	$\langle \text{incomplete message output} \rangle$ $\langle \text{incomplete message input} \rangle$
$\langle \text{incomplete message output} \rangle$	$::=$	out $\langle \text{message identification} \rangle$ to lost [$\langle \text{input address} \rangle$]
$\langle \text{incomplete message input} \rangle$	$::=$	in $\langle \text{message identification} \rangle$ from found [$\langle \text{output address} \rangle$]
$\langle \text{orderable event} \rangle$	$::=$	$\langle \text{incomplete message event} \rangle$

Table 2.4: The textual syntax for incomplete messages.

The textual representation of the incomplete messages is very similar to the textual representation of messages (see Table 2.4). The event-oriented textual representation of the MSC of Figure 2.15 is given as follows:

```

msc example ;
i  : instance ;
j  : instance ;
i  : out m to lost j ;
j  : in m from found i ;
i  : endinstance ;
j  : endinstance ;
endmsc ;

```

It is not clear from the recommendation if the input address of a lost message and the output address of a found message can be names of instances that are not contained in the MSC. As this is irrelevant for the formal semantics presented in this thesis, we will assume that any instance name is allowed as the input address of a lost message event or as the output address of a found message event.

2.3.4 Conditions

Graphically a condition is represented by a *condition* symbol overlapping a number of instances (at least one) and containing a list of *condition names* (at least one) separated by commas. If an instance is not involved in a condition it is drawn through (Z.120 terminology). In Figure 2.16 an example of an MSC with a condition is given. This condition is associated with the instances *i* and *k*, but not with instance *j*.

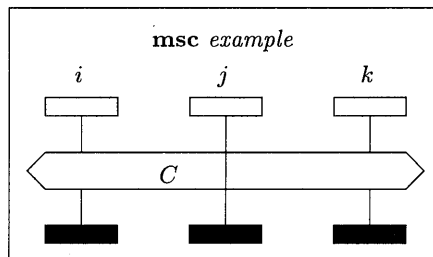


Figure 2.16: Graphical representation of conditions.

In the context of an HMSC conditions can be used to restrict the possible continuations of an MSC by means of final and initial condition identification. We refer to Section 2.5.5 for a thorough treatment of this use of conditions.

The textual representation of conditions is given in Table 2.5. A condition is a first example of an event that can be associated with more than one instance. This type of event is called a *multi instance event*. To facilitate the description of multi instance events without repeating them for every instance, the textual syntax is extended with the possibility to describe such an event for all instances involved. For example the condition from Figure 2.16 that involves the instances *i* and *k* can be described by "*i*, *k* : **condition** *C*". If a multi instance event is associated with all instances of the MSC then the list of instance names preceding the event can be replaced by the

keyword **all**. For example, a condition C that is associated with all instances of an MSC can be described by “**all : condition C** ”. The keyword **all** refers to all instances of the MSC [Ren95a, IT96a].

\langle condition \rangle	::=	\langle condition identification \rangle
\langle condition identification \rangle	::=	condition \langle condition name list \rangle
\langle condition name list \rangle	::=	\langle condition name \rangle { , \langle condition name \rangle }*
\langle event definition \rangle	::=	\langle instance name list \rangle : \langle multi instance event list \rangle
\langle instance name list \rangle	::=	\langle instance name \rangle { , \langle instance name \rangle }* all
\langle multi instance event list \rangle	::=	{ \langle multi instance event \rangle \langle end \rangle }+
\langle multi instance event \rangle	::=	\langle condition \rangle

Table 2.5: The textual syntax for conditions.

In the formal semantics of conditions in MSC92, the semantics of a condition is that of a comment. The reason for this interpretation of conditions is that there are many different interpretations of conditions in practice and that it was hard to obtain an interpretation that was accepted by the users [MR95]. However, in the informal semantics of MSC as given in recommendation Z.120 [IT93], conditions play the role of continuation points (see the SDL Methodology Guidelines [Bel92]).

In MSC96, basically nothing has changed. The only interpretation associated with conditions is the restrictive role for the vertical composition of MSCs as described in an HMSC (see Section 2.5.5). In practice however, several uses of conditions are found.

- Amongst others, Haugen [Hau95] uses conditions in an informal way to indicate a global initial system state or a global final system state. Informally, an MSC with a global initial condition that matches a global final condition of an MSC are considered to be possible continuations. If there are more possible continuations these are considered alternatives. The possible continuations can be displayed in a so-called roadmap. This is helpful in maintaining the collection of MSCs in an MSC document. A shortcoming of the roadmaps is that no starting point can be indicated. Roadmaps have later evolved into HMSC. Feijs [Fei97] suggests that conditions can be used for describing iteration.
- Another typical use of conditions is to stress the relation between an SDL description and an MSC describing its communication behavior. A local condition is then used to represent the states of an SDL process on the instance representing this process in an MSC. Examples of this use of conditions can be found in [RGG96] and also in Extended Sequence Charts as explained in Chapter 1. In these cases the condition is used as an informal, though useful, notation to establish a connection with a corresponding SDL description.

- Conditions are also used to guard the possible execution of an MSC. In [Mei96], the author uses local conditions for defining pre- and postconditions for MSCs and describes several ways of connecting MSCs based on these pre- and postconditions. It is assumed that every instance has a single local state variable, local data variables and local auxiliary variables. Furthermore it is assumed that the MSC contains no action symbols. Changes of data variables are described by means of pre- and postconditions only.

The top condition symbol of an instance describes the precondition and the bottom condition symbol the postcondition. The precondition can be described by means a proposition containing a conjunction of control state constraints and possibly a disjunction of data state constraints. A control state constraint is an equality of the control state variable and a control state identifier. The postcondition consists of exactly one control state constraint and possibly a conjunction of data state constraints.

Two MSCs are strongly connected if the postcondition of the first MSC implies the precondition of the second. Two MSCs are weakly connected if there exists a substitution on control and data variables such that after substitution the postcondition implies the precondition.

2.4 Ordering facilities

2.4.1 Coregions

So far the events specified on an instance were totally ordered in time. To enable the specification of unordered events on an instance the coregion is introduced. A coregion is a part of the instance axis for which the events specified within that part are assumed to be unordered in time. Within a coregion only *orderable events* may be specified such as message events, local actions, timer events, and process creates. An example of an event that may not be used in a coregion is the stop event. Also the instance head and end symbols cannot be allowed in a coregion. For MSC92 the only events that are allowed to be attached to a coregion are message events.

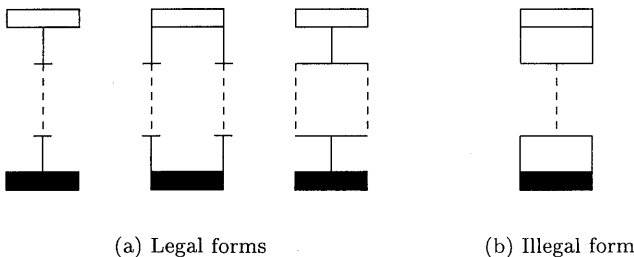


Figure 2.17: Graphical representations of coregions.

Graphically, for instances in line-form a coregion is indicated by drawing a part of the instance axis as a dashed line and for instances in column-form by drawing the same parts of the two vertical lines of the instance as dashed lines. There is also the possibility to use a column-form coregion with a line-form instance. Although there does not seem to be a good reason, the other combination, a line-form coregion with a column-form instance, is not allowed by recommendation Z.120. In Figure 2.17, examples of these forms, also of the illegal combination, are given.

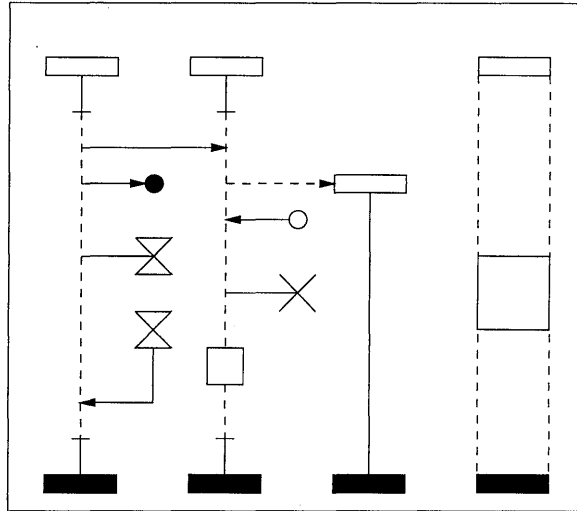


Figure 2.18: Placement of events on coregions.

The recommendation does not clearly state how the orderable events can be attached to a coregion for the three valid forms. The recommendation only illustrates this placement of events on a coregion for messages. Probably it is fair to assume that the placement of the orderable events on a coregion is similar as for instances. Examples of the placement of all orderable events on a coregion are given in Figure 2.18.

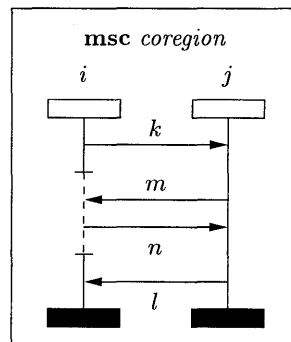


Figure 2.19: Message Sequence Chart with a coregion.

In Figure 2.19 an instance with a coregion is specified which contains an input of message m and an output of a message n . These two events are not ordered in time, but they are executed after the output of message k and before the input of message l . On instance j the events are totally ordered in time.

$\langle \text{coregion} \rangle$::=	concurrent $\langle \text{end} \rangle$ $\langle \text{coevent} \rangle^*$ endconcurrent
$\langle \text{coevent} \rangle$::=	$\langle \text{orderable event} \rangle$ $\langle \text{end} \rangle$
<hr/>		
$\langle \text{non-orderable event} \rangle$::=	$\langle \text{coregion} \rangle$

Table 2.6: The textual syntax for coregions.

In the textual representation a coregion is denoted by a list of the orderable events specified within the coregion started with the keyword **concurrent** and ended by the keyword **endconcurrent**. In Table 2.6 the rules for the extension with coregions are given. Note that it is possible that a coregion contains no events at all. The textual representation of the MSC from Figure 2.19 is as follows:

```

msc coregion ;
i  : instance ;
j  : instance ;
i  : out k to j ;
i  : concurrent ;
    in m from j ;
    out n to j ;
endconcurrent ;
i  : in l from j ;
j  : in k from i ;
j  : out m to i ;
j  : in n from i ;
j  : out l to i ;
i  : endinstance ;
j  : endinstance ;
endmsc ;

```

In Section 2.2 we mentioned that in the event-oriented textual representation it is possible to describe a message input event before its corresponding message output event. The MSC given in Figure 2.20 explains the necessity of this.

A textual representation of this MSC is the following:

```

msc in_before_out ;
i  : instance ;
j  : instance ;
i  : concurrent ;

```

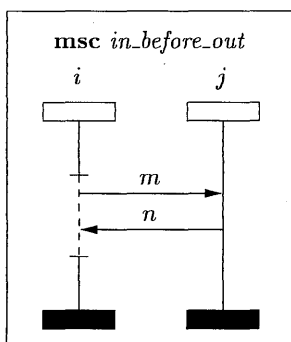



Figure 2.20: Message Sequence Chart with a coregion.

```

    out m to j ;
    in n from j ;
endconcurrent ;
j : in m from i ;
j : out n to i ;
i : endinstance ;
j : endinstance ;
endmsc ;

```

The textual description of the input of message n precedes the textual description of the corresponding output event. To prevent this, the description of the coregion and the output of message n can be switched. Since the input of message m has to precede the output of message n (these events are defined on the same instance), it is also necessary to place the input of message m before the coregion. Observe that after such a switch the input of message m precedes the output of message m in the textual description. The only way to describe the behavior of this MSC textually, in such a way that every output event precedes its corresponding input event, is by removing the coregion.

2.4.2 Causal orderings

With respect to the ordering of events from different instances the coregion does not offer more flexibility. Therefore, a mechanism is included in the language to describe a causal ordering between events. This mechanism can also be used for describing orderings between events from the same instance.

One of the reasons to describe arbitrary orderings of events on an instance is closely related to the use of refinements or instance decompositions (see Section 2.8). With instance decomposition, one instance is replaced by or refined into an MSC such that the external behavior of the instance on the one hand and the MSC on the other hand is identical. We would also like to be able to use the reverse operation, called

abstraction. With the instance decomposition as described in Section 2.8 and without more flexible ways of ordering events on an instance this can only be accomplished if the external behavior of the concrete MSC can be described by means of total orderings and coregions only. A simple example of an MSC where this is impossible is given in Figure 2.21. Suppose that we want to depict this MSC in a broader context

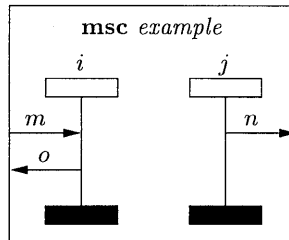


Figure 2.21: Example.

or on a higher level of abstraction by a single instance, say instance ij . This instance ij has to contain three events: (1) a message m which is received from the environment, (2) a message n which is sent to the environment, and (3) a message o which is also sent to the environment. However, there should also be an ordering of the receipt of message m before the sending of message o . This cannot be described by the language introduced so far.

Causal orderings are also introduced to facilitate the description of orderings between events when this ordering cannot be derived from the ordering of the events on an instance and the ordering by means of communication. For example if a local action a on instance i has to occur before a timeout event on instance j . Again, the features of the language discussed so far are not sufficient. The only way to describe this with the MSC-language introduced until now is by defining a communication from i to j where the output occurs after the local action and the input occurs before the timeout event. As MSCs are mostly used for high-level requirements specifications this is undesirable. Also, if many such orderings need to be specified, the additionally introduced communication overhead is disturbing.

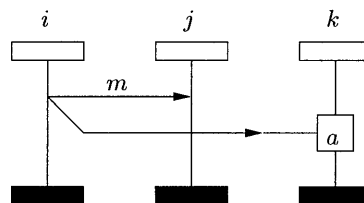


Figure 2.22: Example of a causal ordering.

Graphically a causal ordering of two events is represented by a *causal order* symbol, i.e. a solid line with an arrowhead in the middle (see Figure 2.22). This distinguishes it from normal messages where the arrowhead is placed on one end of the line. The

line may have any orientation and also be bent. The ordering line should be attached to the events that need to be ordered. Only orderable events can be ordered by means of a causal order symbol.

In case of a local action the causal order symbol can start or end at any point of the action symbol. In case of another orderable event the start or end of a causal order symbol coincides with the point of the instance where the event symbol is attached.

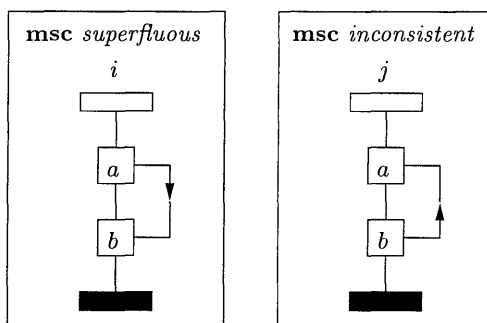


Figure 2.23: Examples of causal orderings within an instance.

The way to describe causal orderings as discussed above can also be used to describe the causal ordering of orderable events from the same instance. In cases where one of the events to be ordered is not inside a coregion, this either results in an inconsistent MSC or it results in an MSC for which the additional causal ordering is superfluous. Examples where two local actions on one instance are causally ordered are given in Figure 2.23. In the first MSC the causal ordering is superfluous as the local actions are already ordered by the total ordering of events on the instance. The fact that this causal ordering is superfluous does not mean that it is not allowed. The second MSC is inconsistent as the local actions are ordered in two conflicting ways. There is no drawing rule that prevents the user from drawing such an inconsistent MSC.

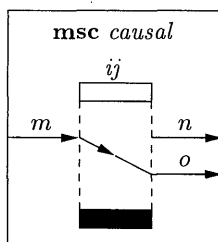


Figure 2.24: Causal ordering within a coregion.

A causal ordering between two events in the same coregion does give additional information. See Figure 2.24 for an example. The input of message m , the output of message n , and the output of message o are specified in a coregion and therefore unordered. But the causal ordering between the input of m and the output of o defines that the first precedes the latter. Note that although the output of n and the

output of o are vertically arranged on the same line they are not ordered. Instance ij represents the external behavior of the MSC given in Figure 2.21.

Thus by using the coregion in combination with the causal order construct, any partial ordering on orderable events on an instance can be described. Thereto, all events have to be specified in a coregion and causal orderings have to be added explicitly when two events have to be ordered. If such a coregion contains many events and many arrows the drawing easily gets confused. For an example see the coregion in Figure 2.25.

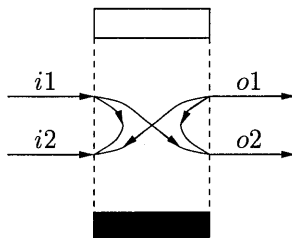


Figure 2.25: Causal ordering within an instance.

As an alternative the language MSC offers the possibility to leave the head of the arrow out. Thereby the order symbol is reduced to a line. The connection lines define the ordering of events from top to bottom. Also crossings of these lines have a meaning. Event a is ordered causally before event b if and only if there is non-increasing line going from a to b . The coregion from Figure 2.25 can then also be depicted as shown in Figure 2.26.

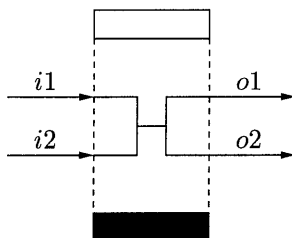


Figure 2.26: Causal ordering within an instance (alternative representation).

This shorthand notation is not always convenient. Especially in situations where the user is not prepared to shift events from the left-hand axis to the right-hand axis of the coregion and vice versa. See for example the coregion given in Figure 2.27. It cannot be represented in the arrowless way without shifting events from one axis of the coregion to the other axis of the coregion.

In the textual syntax (see Table 2.7) causal orderings are represented by using the keyword **before** followed by a list of event names. An *event name* refers to an event specified somewhere in the same MSC. Thus it can be an event from the same instance or an event from another instance. An event name can be associated with an event in the textual syntax by placing the event name just before the event.

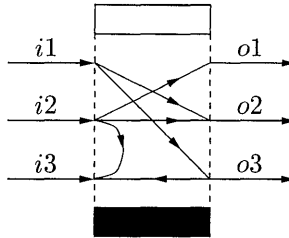


Figure 2.27: Causal ordering within an instance.

<code><orderable event></code>	<code>::=</code>	<code>[<event name>]</code> <code>{ <message event></code> <code> <incomplete message event></code> <code> <create></code> <code> <timer statement></code> <code> <action></code> <code>}</code> <code>[before <event name list>]</code>
<code><event name list></code>	<code>::=</code>	<code><order dest> [, <event name list>]</code>
<code><order dest></code>	<code>::=</code>	<code><event name></code>

Table 2.7: The textual syntax for causal orderings.

The MSC given in Figure 2.24 can be described textually as follows:

```
msc causal ;  
ij : instance ;  
ij : concurrent ;  
    in m from env before l ;  
    out n to env ;  
    l out o to env ;  
    endconcurrent ;  
ij : endinstance ;  
endmsc ;
```

The output of message *o* by instance *ij* to the environment has event name *l*. The input of message *m* by instance *ij* from the environment has to precede the event with event name *l*, i.e. the output of message *o* by instance *ij* to the environment.

The textual description of the causal orderings is asymmetrical in the sense that the causal ordering is only attached to the event that is associated with the starting point of the causal order arrow. Although this gives rise to short descriptions we feel that it should also be possible (maybe it should even be demanded) to describe the ordering for the other event. This can be achieved by extending the textual syntax with a keyword **after**. We will return to this subject in Section 2.9.3.

2.5 Combining MSCs with composition constructs

MSC based specifications often consist of many different MSCs, instead of one single MSC. MSC offers ways to group single MSCs into *MSC documents*. An MSC document is a collection of MSCs.

MSCs can be put in a wider context by means of *composition operators*. The three primitive operations are vertical composition, horizontal composition and alternative composition. In the language MSC these concepts of composing MSCs are manifest in different ways: in inline expressions, MSC reference expressions and High-level MSCs.

First the intuitive semantics of the operations vertical, horizontal and alternative composition is given. Then MSC documents are treated, followed by inline expressions and MSC references. The last part of this section describes the use of the composition mechanisms in High-level MSC.

2.5.1 Vertical, horizontal and alternative composition

In this section, we will focus on vertical, horizontal and alternative composition of MSCs. In the sections to follow a graphical and textual syntax is provided for these and other composition mechanisms by means of inline expressions, MSC reference

expressions and High-level MSCs. We explain the operations of horizontal and vertical composition and illustrate these by means of examples. The examples do not form a precise definition of the semantics. For a formal definition see Chapter 3.

Vertical composition The vertical composition of two MSCs refers to the operation of placing one MSC at the bottom of another one and then linking the instances they have in common thus obtaining a new MSC.

If the MSCs have *no* instances in common the meaning of the vertical composition is the same as an MSC with the instances of these MSCs placed next to each other. See Figure 2.28 for an example. MSC *first* has instances named *i* and *j* and MSC *second* has instances named *k* and *l*. The MSCs have no instances in common, so there are no links to be made. Thus vertical composition of MSCs does not necessarily mean that all events from the first MSC (in the example MSC *first*) have to be executed before any event from the second MSC (MSC *second*) can be executed. In the example this means that the sending of *n* might as well occur before the sending of *m*.

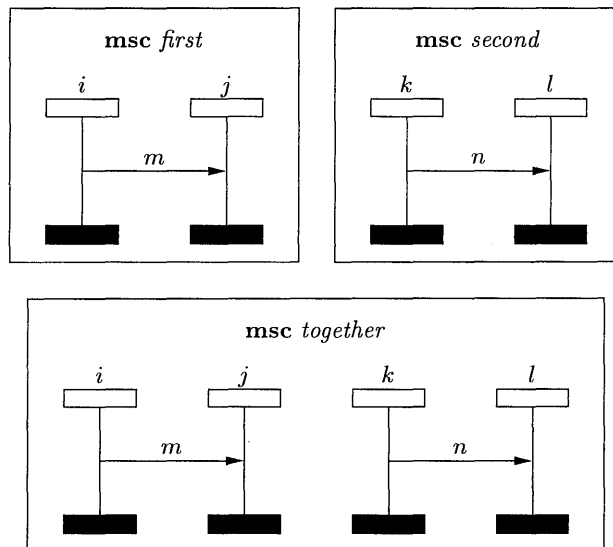


Figure 2.28: Vertical composition with disjoint instances.

Another case occurs if the MSCs have *all* instances in common. Then all events from an instance of the second MSC have to occur after the events from the same instance of the first MSC. For an example see Figure 2.29. The MSCs *first* and *second* have the instances *i*, *j*, and *k* in common. The reception of message *m* by instance *j* necessarily has to precede the reception of message *n* by instance *j* in the resulting MSC *together*. In this example it is still possible that the sending of message *n* by instance *i*, which is an event described in MSC *second*, is executed before the reception of message *m* by instance *j*, described in MSC *first*.

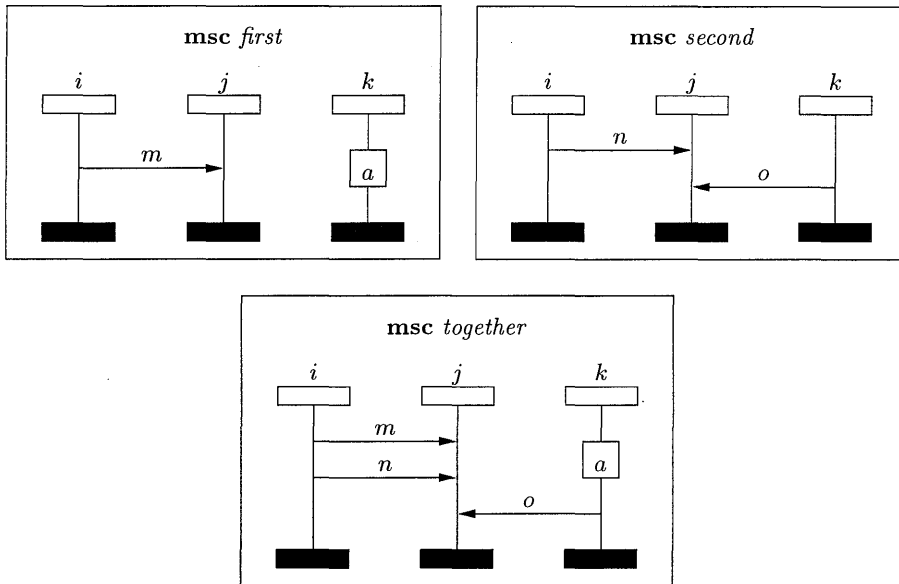


Figure 2.29: Vertical composition with the same instances.

Also the situation in which the MSCs have instances *in common* and also have *different* instances is allowed. For example the MSCs *first* and *second* from Figure 2.30 have the instance *j* in common, but instance *i* is only described for MSC *first* and instance *k* is only described for MSC *second*. The result of the vertical composition of the MSCs *first* and *second* is given as MSC *together* in the same figure.

Horizontal composition The horizontal composition of two MSCs refers to the operation of placing them next to each other. If the MSCs have some or all instances in common, it is assumed that the behavior of the common instance(s) is the interleaving of the behaviors of these instance(s) in the separate MSCs.

In the case that the MSCs have no instances in common, the horizontal composition is similar to the vertical composition (see Figure 2.28). For an example of the case where the MSCs have one instance in common, we refer to Figure 2.31.

In this example the MSCs *first* and *second* have the instance *j* in common. As stated before, the behavior of the shared instance is obtained by interleaving the events of the separate instance descriptions. This can be expressed in a coregion with general orderings as shown in MSC *together*.

For Interworkings, the interworking merge (\parallel_{iw}) of *first* and *second* (regarding them as Interworkings), cannot be expressed by a single Interworking. Instead a set of six Interworkings is needed. In these resulting Interworkings there is no need to express that the input of *m* precedes the output of *n* and that the output of *o* precedes the

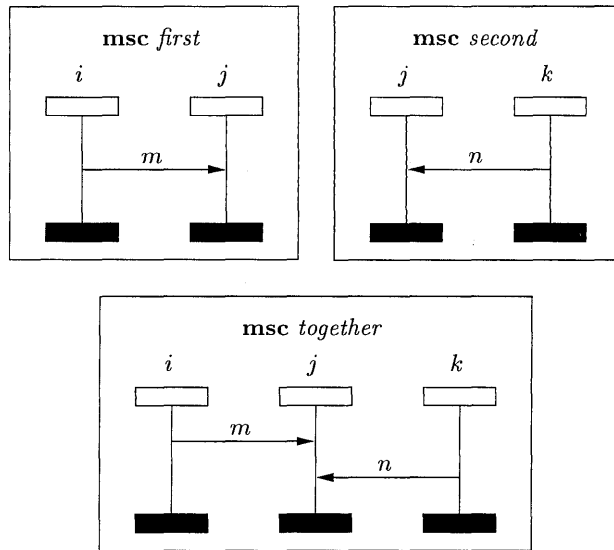


Figure 2.30: Vertical composition.

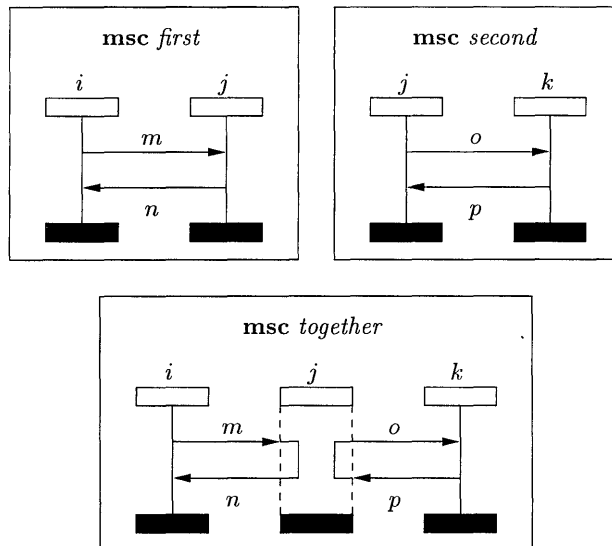


Figure 2.31: Horizontal composition with shared instance.

input of p as the communications are synchronous. Due to the introduction of the coregion, the result of horizontally composing two MSCs can be drawn as an MSC again if only orderable events are defined on the instances the MSCs have in common.

Alternative composition Usually a system is not described by means of one single MSC; instead a number of MSCs is used to describe several alternative scenarios. With the features of MSC introduced so far only alternative scenarios can be described over the same events. So each trace contains precisely the same events. For example, it is impossible to describe that either an event a or an event b is executed. A means to describe alternatives is by giving one MSC for each of the alternatives. Thus large piles of scenarios come into existence, for example when describing system requirements or when describing a system by giving different use cases [JCJO92].

In complex systems there are many points of deviating behavior. Therefore, it is important to be able to indicate at what point alternatives occur. For that reason the language MSC offers several possibilities to describe alternatives in an MSC. An important aspect of the meaning of the alternative composition mechanism in MSC is that the moment of choice between the different scenarios is postponed until that choice can no longer be avoided.

Consider the MSCs A and B as given in Figure 2.32. Each of these MSCs has one initial event, the sending of m and the sending of n respectively. The alternative composition of these MSCs now has two initial events: the sending of message m and the sending of message n . If the sending of message m is executed a choice is made for the execution of MSC A . On the other hand, if the sending of message n is executed, a choice is made in favour of MSC B . Thus, with the execution of an event which can be executed by only one of the alternatives, all alternatives that cannot execute this event are discarded.

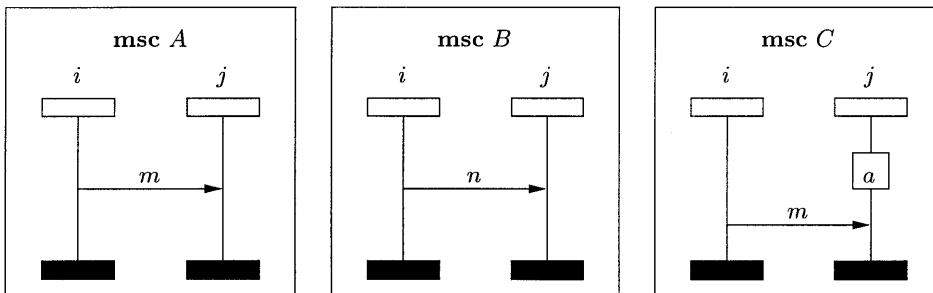


Figure 2.32: MSCs.

Now consider the MSCs A and C from Figure 2.32. If the local action on instance j of MSC C is executed necessarily a choice is made in favour of MSC C . But, if the sending of message m occurs, this event can originate from either MSC A or C , though it is not clear from which of the two MSCs it originates at the moment of execution of this event. Alternative composition in MSCs is defined in such a way

that no choice is made until this cannot be avoided. One could say that after the execution of the sending of message m there still are two alternatives: the parts of the MSCs A and C that remain to be executed. In this specific example now a choice has to be made as the MSCs have no initial events in common anymore.

2.5.2 MSC documents

In the following sections we will focus on the means offered by MSC96 to compose MSCs. As a consequence we must be able to describe more than one MSC. For this purpose Message Sequence Chart documents are used.

Graphically an MSC document is given as a *frame* symbol with a *document head* in it.

Textually, an *MSC document* consists of an *MSC document head* and an *MSC document body*. The MSC document head consists of the keyword **mscdocument** followed by an *MSC document name* and optionally followed by the keywords **related to** and an *sdl reference*. The SDL reference is, if MSC is used in combination with SDL, used for the identifier (pathname) of the SDL document to which the MSCs refer. In combinations of MSC with other languages/tools the SDL reference can be used for similar reference purposes. The MSC document body consists of a number of Message Sequence Charts. The textual grammar for MSC documents is given in Table 2.8. The nonterminal $\langle \text{text} \rangle$ is a sequence of letters, digits, spaces and other symbols. Its precise definition is not interesting here.

$\langle \text{msc document} \rangle$::=	$\langle \text{msc document head} \rangle$ $\langle \text{msc document body} \rangle$
$\langle \text{msc document head} \rangle$::=	$\langle \text{document head} \rangle$
$\langle \text{document head} \rangle$::=	mscdocument $\langle \text{msc document name} \rangle$ [related to $\langle \text{sdl reference} \rangle$] $\langle \text{end} \rangle$
$\langle \text{sdl reference} \rangle$::=	$\langle \text{sdl document identifier} \rangle$
$\langle \text{identifier} \rangle$::=	[$\langle \text{qualifier} \rangle$] $\langle \text{name} \rangle$
$\langle \text{qualifier} \rangle$::=	<< $\langle \text{text} \rangle$ >>
$\langle \text{msc document body} \rangle$::=	$\langle \text{message sequence chart} \rangle^*$

Table 2.8: The textual syntax for MSC documents.

For MSC documents the following static requirements are formulated. Within an MSC document there must not be two or more MSCs with the same name. Within the MSCs of an MSC document only references to MSCs specified within that MSC document may be used. An MSC may not be depending on itself, directly or through a number of references.

The textual representation of MSC documents in MSC92 differs slightly from the textual representation used in MSC96. In MSC92 the end of the MSC document

was marked with the keyword **endmscdocument**. As a consequence of this change, backward compatibility is lost at a point where this can hardly be motivated.

2.5.3 Inline expressions

Inline expressions provide a means to describe the composition of MSCs inside an MSC. The operators that can be used are amongst others the horizontal and alternative composition discussed before. The reason to omit vertical composition is that we already have a natural means to describe vertical composition inside an MSC.

Graphically an inline expression consists of an *inline expression* symbol that is attached to a number of instances (at least one). This inline expression symbol contains in the left-upper corner one of the keywords **alt**, **par**, **opt**, **exc** or **loop**. These keywords indicate the composition operation that is described by the inline expression. Inside the inline expression symbol the operands are described in the form of an anonymous MSC, i.e. an MSC without MSC name and without instance head and end symbols.

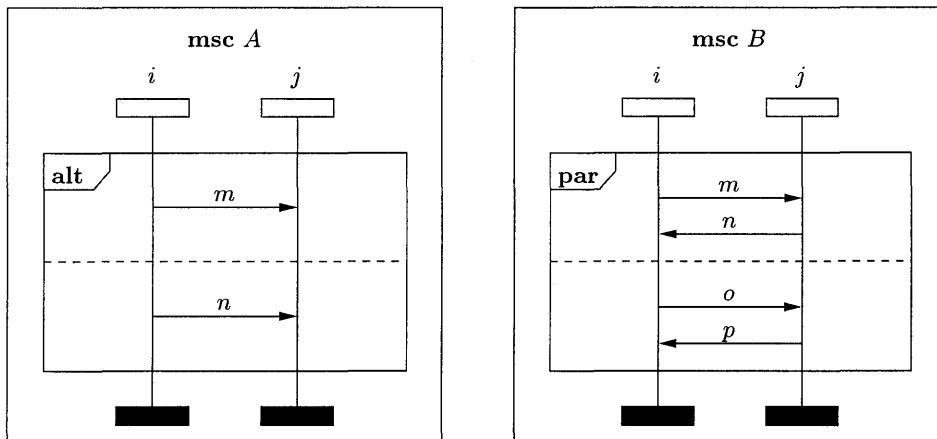


Figure 2.33: Examples of inline expressions.

Some examples of inline expressions are given in Figure 2.33. In MSC *A* an inline expression is attached to the instances *i* and *j*. This inline expression has the keyword **alt** in its upper left corner in order to indicate that the parts of the MSCs that are separated by means of the separator symbol are considered *alternatives*. In this particular example there are two operands. Different operands are separated by means of a horizontal dashed line, the *separator* symbol. The operands of the inline expression are described by associating events to the instance axes that are displayed inside the inline expression symbol. At this point it is not allowed to draw messages or causal orderings that cross the inline expression symbol or the separator symbols that occur therein. The first operand describes the sending of a message *m* by instance *i* and its subsequent reception by instance *j*. The second operand describes the sending of

a message n by instance i and its subsequent reception by instance j . The meaning of this MSC in terms of sequences of events that can be performed is that either the sending and reception of m or the sending and reception of n takes place but not both. As soon as the sending of one of the messages takes place it is known which operand is executed.

In MSC B the *horizontal composition* of two “MSCs” is indicated by means of the keyword **par**. In this case all events are executed in such a way that the orderings described by the first operand are respected and at the same time the orderings described by the second operand are respected. This mode of operation is often called interleaving. MSC B' from Figure 2.34 has the same behavior as MSC B from Figure 2.33.

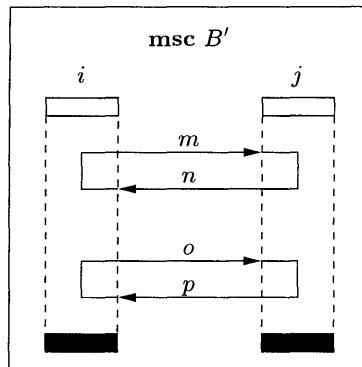


Figure 2.34: MSC equivalent to MSC B .

Both alternative and horizontal composition can have any finite, positive number of operands. These operands are all drawn inside the inline expression symbol and they are separated by a dashed horizontal line, the *separator* symbol. As in the recommendation no scheme for placing parentheses is indicated it is assumed that these operators are commutative and associative.

For optional composition and for repetition there is exactly one operand. This operand is described by means of the part of the MSC that is drawn inside the inline expression symbol. Examples of both are given in Figure 2.35.

MSC A describes an MSC where the sending and receiving of message m can occur, but does not have to occur, it is optional. In both cases message n is sent and received. MSC C from Figure 2.36 has the same behavior: the first alternative corresponds to the case that the optional part is executed, and the second alternative corresponds to the case that the optional part is neglected. In general every optional inline expression can be replaced by an alternative inline expression with two operands: one without events and one with the contents of the optional inline expression.

The inline loop expression in MSC B of Figure 2.35 describes that the sending and receiving of message m occurs zero, one or two times, followed by the sending and

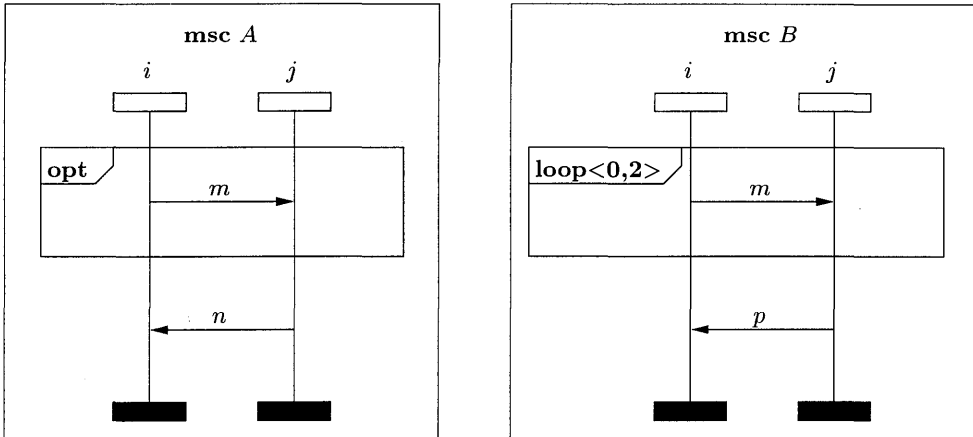


Figure 2.35: Examples of inline expressions.

receiving of message *p*. Intuitively the behavior of MSC *B* is the same as the behavior of MSC *D* from Figure 2.36.

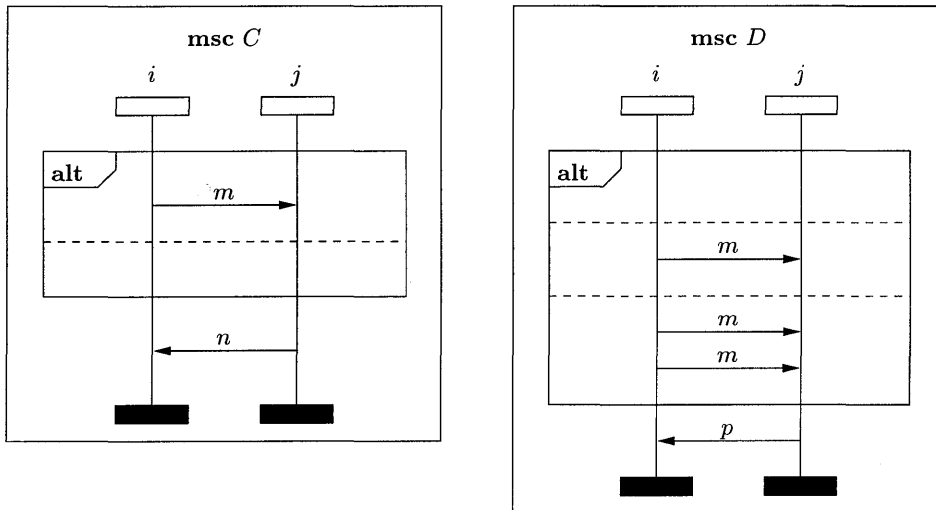


Figure 2.36: Examples of inline expressions.

The keyword **loop** is followed by a *loop boundary*. This loop boundary refers to the number of repeated vertical compositions of the operand of the inline expression. The loop boundary, if present, indicates the minimal and/or maximal number of vertical compositions of the operand. In the recommendation such a number can either be the keyword **inf**, representing infinity, or a sequence of *natural names*. A natural

name can be any label. For the semantics it is important to be able to interpret the sequences of natural names as natural numbers. In this thesis, we will only use decimal digits as natural names. The loop boundary can be of the form $\langle n \rangle$ or $\langle n, m \rangle$ where n and m are sequences of natural names or **inf**. The combination **loop** $\langle n, m \rangle$ means that the operand of the operator is executed at least n and at most m times. If the interpretation of the sequence of natural names n is greater than the interpretation of the sequence of natural names m then this means that the operand is executed zero times. The combination **loop** $\langle n \rangle$ can be viewed as a shorthand for the combination **loop** $\langle n, n \rangle$, i.e. the operand is executed exactly n times. If the loop boundary is omitted this is interpreted as the combination **loop** $\langle 1, \text{inf} \rangle$.

Inline expressions for exceptions are indicated by the keyword **exc**. For this kind of inline expressions not the inline expression symbol is used but the exception inline expression symbol (see Figure 2.37). The *exception inline expression* symbol differs from the inline expression symbol in that its lower horizontal line is dashed instead of solid. The reason is that the exception operator is a binary operator. Its first operand, the normal mode of operation, is indicated by the part of the MSC following the inline expression, and the second operand, the exceptional case is depicted in the inline expression.

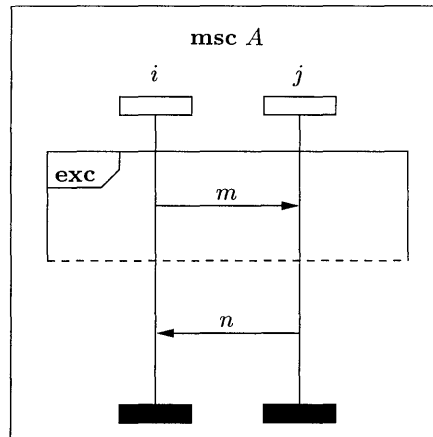


Figure 2.37: Example of an inline expression.

The intuition of an exception inline expression is that either the normal case is executed or the exceptional case is executed. Thus, an exception inline expression can be seen as the alternative composition of the two operands.

An exception inline expression must be associated with all instances in the MSC. In case of an exception inline expression which does not overlap all instances there might be difficulties in determining the part of the MSC following the inline expression. An example thereof is given in Figure 2.38. The local action a is drawn above the inline expression. The local action is not ordered relatively to any event of the inline expression and therefore it can be executed both before and after the events

of the inline expression. To circumvent situations where the second operand of the exception inline expression cannot be determined, the recommendation requires that if an exception inline expression is used it must be attached to all instances of the MSC.

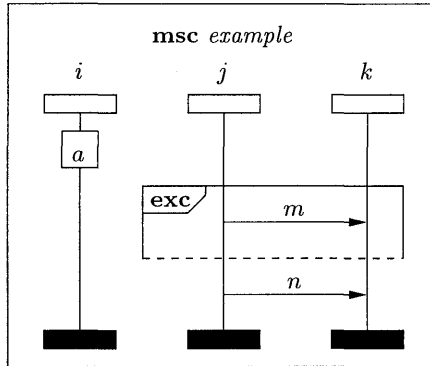


Figure 2.38: Illegal inline expression.

If an instance is not involved in the operands of an inline expression, then it is possible to hide the part of the instance axis of such an instance behind the inline expression. See Figure 2.39 for an example of an inline expression in which instance *j* does not participate.

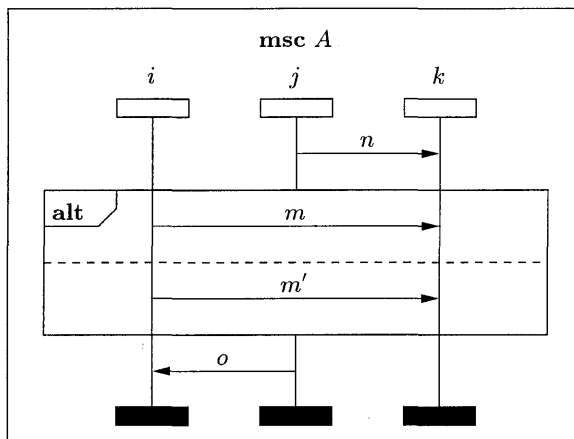


Figure 2.39: Example of an inline expression.

Inline expressions can be nested as long as the inner inline expression is contained completely in one operand of the outer inline expression.

The textual syntax of inline expressions is given in Table 2.9. For example, the textual representation of MSC *A* from Figure 2.33 is given by


```

msc A ;
i : instance ;
j : instance ;
i , j : alt begin ;
    i : out m to j ;
    j : in m from i ;
    alt ;
    i : out n to j ;
    j : in n from i ;
    alt end ;
i : endinstance ;
j : endinstance ;
endmsc ;

```

\langle inline expr \rangle	::=	\langle loop expr \rangle \langle opt expr \rangle \langle alt expr \rangle \langle par expr \rangle \langle exc expr \rangle
\langle loop expr \rangle	::=	loop [\langle loop boundary \rangle] begin \langle end \rangle \langle msc body \rangle loop end
\langle opt expr \rangle	::=	opt begin \langle end \rangle \langle msc body \rangle opt end
\langle exc expr \rangle	::=	exc begin \langle end \rangle \langle msc body \rangle exc end
\langle alt expr \rangle	::=	alt begin \langle end \rangle \langle msc body \rangle { alt \langle end \rangle \langle msc body \rangle } [*] alt end
\langle par expr \rangle	::=	par begin \langle end \rangle \langle msc body \rangle { par \langle end \rangle \langle msc body \rangle } [*] par end
\langle loop boundary \rangle	::=	\langle inf natural \rangle [, \langle inf natural \rangle] \rangle
\langle inf natural \rangle	::=	inf \langle natural name \rangle ⁺
\langle multi instance event \rangle	::=	\langle inline expr \rangle

Table 2.9: The textual syntax for inline expressions.

2.5.4 MSC reference expressions

An MSC reference expression can be used to refer to other MSCs in an MSC document by means of their unique MSC name. Graphically an MSC reference expression is

represented by a textual formula in a rounded frame, the *msc reference* symbol, which is placed on top of a number of instances.

Such a textual formula is an expression containing references to other MSCs in the MSC document via their MSC name, operators for composing MSCs: **alt**, **seq**, **par**, **opt**, **exc**, **empty**, and **loop** and parentheses for grouping subexpressions. MSC *A* in Figure 2.40 contains an MSC reference expression that is attached to the instances *i* and *j* and that contains the textual formula *B alt C* which refers to the alternative composition of the MSCs *B* and *C* (which are not displayed).

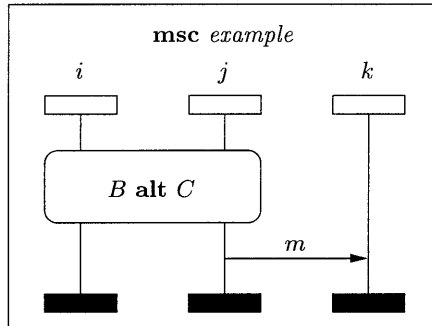


Figure 2.40: An example of an MSC reference expression.

The textual representation of formulas, which is also used in the graphical syntax of MSC reference expressions, is given in Table 2.10.

$\langle \text{msc ref expr} \rangle$	$::=$	$\langle \text{msc ref par expr} \rangle \{ \text{alt } \langle \text{msc ref par expr} \rangle \}^*$
$\langle \text{msc ref par expr} \rangle$	$::=$	$\langle \text{msc ref seq expr} \rangle \{ \text{par } \langle \text{msc ref seq expr} \rangle \}^*$
$\langle \text{msc ref seq expr} \rangle$	$::=$	$\langle \text{msc ref exc expr} \rangle \{ \text{seq } \langle \text{msc ref exc expr} \rangle \}^*$
$\langle \text{msc ref exc expr} \rangle$	$::=$	$[\text{exc}] \langle \text{msc ref opt expr} \rangle$
$\langle \text{msc ref opt expr} \rangle$	$::=$	$[\text{opt}] \langle \text{msc ref loop expr} \rangle$
$\langle \text{msc ref loop expr} \rangle$	$::=$	$[\text{loop } [\langle \text{loop boundary} \rangle]] \{ \text{empty} \}$
		$\quad \langle \text{msc name} \rangle$
		$\quad (\langle \text{msc ref expr} \rangle)$
		$\quad \}$

Table 2.10: The textual syntax for textual formula.

The binding power of the operators is in descending order as follows: **loop**, **opt**, **exc**, **seq**, **par**, **alt**. The binding power can be superseded by using parentheses. Examples of MSC reference expressions are:

- *A*;
- (*A alt B*) seq *C*;

- `loop <5,16> (A par B);`
- `A seq loop <3,inf> B.`

There are two requirements that must be satisfied with respect to the instances that are overlapped:

1. If an instance that is present in the enclosing MSC diagram is also present in the MSC reference expression, then the MSC reference symbol must be attached to this instance. An instance is present in an MSC reference expression if at least one of the MSCs that are referenced in the expression has an instance with that name.
2. If two MSC reference expressions in the same enclosing MSC diagram share an instance then this instance must be drawn in the enclosing MSC diagram.

Note that these requirements do not say that every instance that is present in the MSC reference expression must be visible in the enclosing MSC. The requirements also do not say that an MSC reference expression may not overlap an instance that is not present in the MSC reference expression.

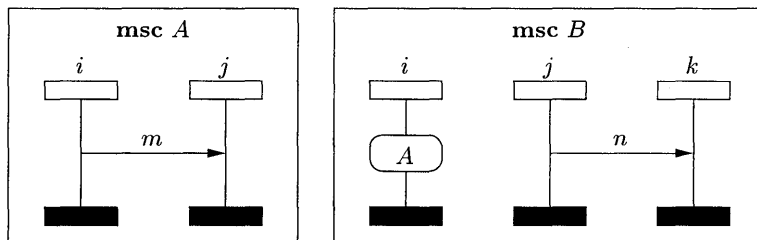


Figure 2.41: An example of an illegal MSC reference expression.

The first requirement is included to prevent the user from drawing an MSC where it is not clear how the events on an instance are ordered. An example is given in Figure 2.41. Suppose that an MSC *A* with instances *i* and *j* is given, and that a message *m* is transmitted from instance *i* to instance *j*. Then, in MSC *B*, it is not clear in what order the input of message *m* by *j* and the output of *n* by *j* are to be executed. This might be exactly what the user wants to indicate, but since it cannot be easily seen from MSC *B* that there exists another event on instance *j*, this is not allowed.

The second requirement is motivated similarly. As two MSC reference expressions share an instance it should be visible on the level of the enclosing MSC diagram how the events specified on these occurrences of the instance are ordered. Consider for example the MSCs shown in Figure 2.42. MSC *C* refers to the MSCs *A* and *B*. In MSC *C* only the instances *i* and *k* are drawn. Note that MSC *C* satisfies the first requirement. Both MSC *A* and MSC *B* have an instance *j*. In order to know how

the events from the two occurrences of instance j are ordered with respect to each other it is mandatory to display instance j in MSC C . Then the first requirement is no longer satisfied. In order to satisfy the first requirement the user must make sure that both references overlap this instance j in MSC C . The only two ways to do this are given as MSCs D and E in Figure 2.42. Diagrams where the MSC reference symbols overlap or cross are disallowed by the drawing rules.

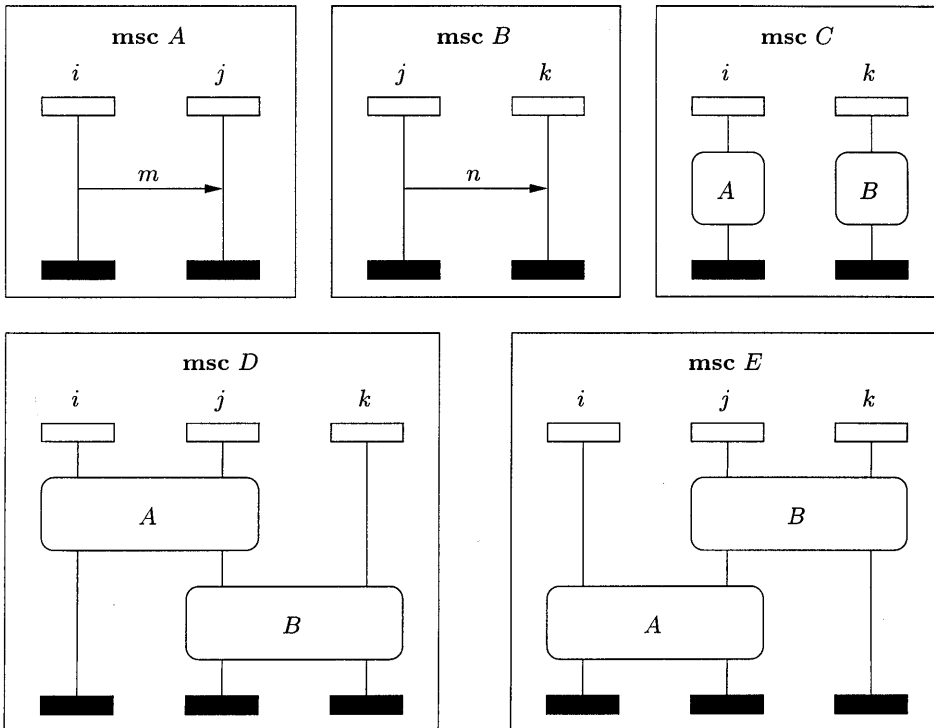


Figure 2.42: Example of MSCs: MSC C is illegal.

Next, we explain the meaning of the operators. For a precise description of their meaning we refer to Chapter 3 where for each of the operators a semantical equivalent is defined. The operators **seq**, **par** and **alt** refer to the notions of vertical composition, horizontal composition and alternative composition respectively. These have already been explained in Section 2.5.1. The meaning of the operators **exc**, **opt** and **loop** has already been explained in Section 2.5.3. The operator **empty** is a nullary operator, i.e. a constant, that refers to an MSC without events. Recommendation Z.120 does not specify which instances are contained in an empty MSC. There seem to be three choices: An empty MSC contains all instances, no instances, or the instances that are overlapped by the MSC reference expression the empty MSC appears in. The first choice is not acceptable as this could violate the static requirements mentioned above. The third choice is also not acceptable as the empty MSC can also be used in an HMSC and HMSCs do not specify instances. Thus, in this thesis, it is assumed that an empty MSC has no instances.

Textually an MSC reference expression is indicated by the keyword **reference** followed by the textual formula. See Table 2.11 for the textual syntax. The event-oriented description of MSC *D* of Figure 2.42 is given by

```

msc D ;
i  : instance ;
j  : instance ;
k  : instance ;
i , j: reference A ;
j , k reference B ;
i  : endinstance ;
j  : endinstance ;
k  : endinstance ;
endmsc ;

```

$\langle \text{msc reference} \rangle$	$::=$	reference $\langle \text{msc ref expr} \rangle$
$\langle \text{multi instance event} \rangle$	$::=$	$\langle \text{msc reference} \rangle$

Table 2.11: The textual syntax for MSC reference expressions.

In recommendation Z.120 a notion of parameter substitution is defined on MSC reference expressions. Parameter substitution can be useful as it increases the possibilities of reuse of MSCs. The extension of the grammar defining the textual formula that can be used in an MSC reference symbol is given in Table 2.12. There are parameter

$\langle \text{msc ref loop expr} \rangle$	$::=$	[loop [$\langle \text{loop boundary} \rangle$]] $\langle \text{msc name} \rangle$ $\langle \text{parameter substitution} \rangle$
$\langle \text{parameter substitution} \rangle$	$::=$	subst $\langle \text{substitution list} \rangle$
$\langle \text{substitution list} \rangle$	$::=$	$\langle \text{substitution} \rangle$ [, $\langle \text{substitution list} \rangle$]
$\langle \text{substitution} \rangle$	$::=$	$\langle \text{replace message} \rangle$ $\langle \text{replace instance} \rangle$ $\langle \text{replace msc} \rangle$
$\langle \text{replace message} \rangle$	$::=$	[msg] $\langle \text{message name} \rangle$ by $\langle \text{message name} \rangle$
$\langle \text{replace instance} \rangle$	$::=$	[inst] $\langle \text{instance name} \rangle$ by $\langle \text{instance name} \rangle$
$\langle \text{replace msc} \rangle$	$::=$	[msc] { empty $\langle \text{msc name} \rangle$ } by { empty $\langle \text{msc name} \rangle$ }

Table 2.12: The textual syntax for parameter substitution.

substitution mechanisms for message names, instance names and MSC names. No parameter substitution mechanism is provided for other names such as timer names and condition names. Parameter substitution is not allowed on arbitrary MSC reference expressions, but only on MSC references. At the same time the recommendation

states that if a parameter substitution is applied on an MSC reference it should also be applied on all MSCs referenced in the corresponding MSC. This corresponds to defining parameter substitution for arbitrary MSC reference expressions.

A parameter substitution consists of the keyword **subst** followed by a list of substitutions. A substitution can be the replacement of a message, an instance or an MSC (reference).







The meaning of MSC name substitution is a replacement of references to MSCs by references to MSCs. The meaning of instance and message name substitution is a replacement of these names by other names. The meaning of multiple parameter substitutions is the parallel application of parameter substitution. Thus the substitution “**subst A by B, B by A**” implements a simultaneous replacement of references to MSC *A* by references to MSC *B* and vice versa.

The recommendation does not state what it means if a substitution list contains more than one substitution for a name. An example is the expression “**A subst A by B, A by C**”. This should be disallowed explicitly.

2.5.5 High-level Message Sequence Charts

A High-level Message Sequence Chart (HMSC) is a graphical overview of the relation between the MSCs contained. It helps in keeping track of the control-flow. In an HMSC vertical, horizontal and alternative composition as well as repetitive behavior are captured in an attractive graphical layout: references to MSCs are related by means of arrows connecting them. One can look at HMSC as the synthesis of the roadmap approach [Rud95, RGG96] and the operator approach [Hau94].

Graphically an HMSC is a graph with several types of nodes connected by arrows. The types of nodes that can be included in an HMSC are the following:

- start node: 
- end node: 
- MSC reference node: 
- condition node: 
- connection node: 
- parallel frame: 

A first example of an HMSC is given in Figure 2.43.

Recommendation Z.120 states only one static requirement restricting the allowed diagrams: Every node in the HMSC must be reachable from the start node, i.e. the

graph must be connected. From the textual representation of HMSC, however, we additionally obtain the following requirements. Every HMSC has exactly one start node. The start node has no incoming arrows. An end node has no outgoing arrows. Every node that is not an end node has at least one outgoing arrow.

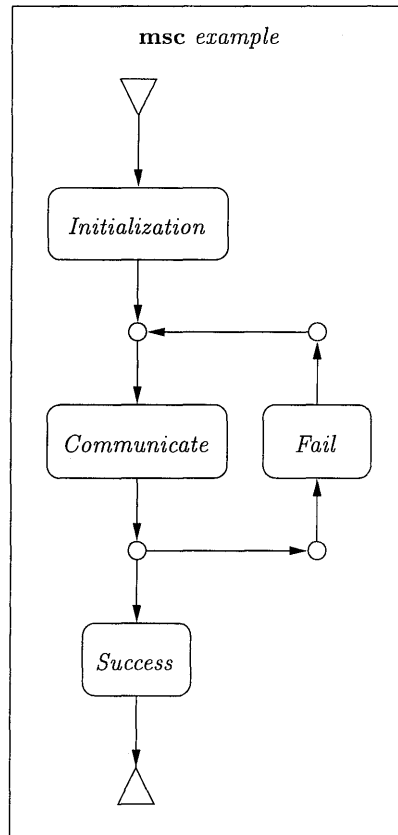


Figure 2.43: An example of an High-level Message Sequence Chart.

An MSC reference node contains any textual formula that can be used in an MSC reference expression (see Section 2.5.4). A condition node contains a non-empty list of condition names. A parallel frame contains at least one anonymous HMSC, i.e. an HMSC without name and without frame. The recommendation does not specify if different anonymous HMSCs can share nodes. In our opinion it is better not to share nodes between different anonymous HMSCs as this confuses the diagram.

Recommendation Z.120 requires that an arrow that connects two nodes always goes from the lower segment of a node to the upper segment of a node. In this thesis we also attach the arrows at other points to the nodes (as for example in Figure 2.43). Lines can be used instead of arrows for connecting nodes. A line connecting two nodes is always interpreted as an arrow from the upper node to the lower node. To avoid confusion, in this thesis, only arrows are used to connect nodes.

An HMSC describes relations between the contained MSCs in a graphically attractive way. Diagrams with many nodes and arrows can easily become unreadable for the human eye. By using connection nodes we can improve a lot on this problem. Connection nodes together with their incoming and outgoing arrows are a convenient shorthand. Every combination of an incoming and an outgoing edge of a connection node represents an arrow between the source of the incoming arrow and the destination of the outgoing arrow. A transformation of HMSCs with connection nodes to HMSCs without connection nodes can easily be given. An example of an HMSC with connection nodes is given in Figure 2.43.

An arrow between two nodes means that they are composed vertically. For the HMSC given in Figure 2.44 this means that the MSCs *A* and *B* are composed vertically.

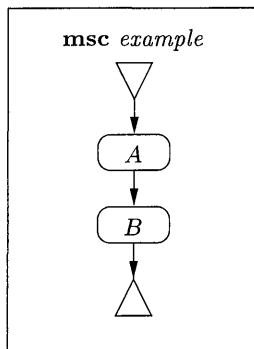


Figure 2.44: Vertical composition in an HMSC.

If a node has more than one outgoing arrow this indicates a number of alternatives with which this node can be composed vertically. The HMSC given in Figure 2.45 describes that MSC *A* is composed vertically with either MSC *B* or MSC *C*.

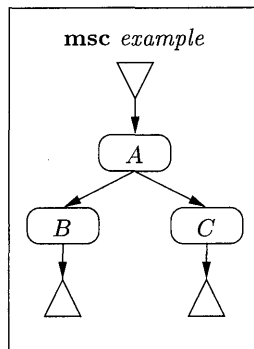


Figure 2.45: Alternatives in an HMSC.

Horizontal composition is denoted by the parallel frame. The parallel frame contains at least one anonymous HMSC. Each anonymous HMSC contained in the parallel frame denotes an operand for the horizontal composition. The anonymous HMSCs contained in such a frame are composed horizontally. The HMSC in Figure 2.46 describes that the MSCs *A* and *B* are composed horizontally.

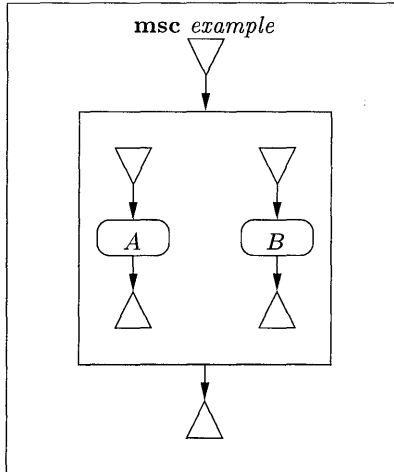


Figure 2.46: Horizontal composition in an HMSC.

With start nodes, end nodes and connection nodes no dynamic semantics is associated. The semantics of an MSC reference node is the semantics of the textual formula it contains. The semantics of these textual formula has already been explained in Section 2.5.4. The semantics of a parallel node has been explained above. Also with condition nodes no dynamic semantics is associated. Conditions however can be used in an HMSC to restrict the possible continuations. Thereto the following four requirements are formulated [IT96b]:

- If an MSC reference node in an HMSC is immediately preceded by a condition node then the condition names indicated inside the condition node must be a subset of the initial conditions of the MSC reference expression indicated inside the MSC reference node.
- If an MSC reference node in an HMSC is immediately followed by a condition node then the condition names indicated inside the condition node must be a subset of the final conditions of the MSC reference expression indicated inside the MSC reference node.
- If a parallel frame is immediately preceded by a condition node, then the condition names indicated inside the condition node must be a subset of the set of initial conditions of the parallel frame.

- If a parallel frame is immediately followed by a condition node, then the condition names indicated inside the condition node must be a subset of the set of final conditions of the parallel frame.

The set of initial (final) conditions of an MSC reference expressions is defined as follows:

- The set of initial (final) conditions of a reference to an MSC A is the set of initial (final) conditions of MSC A .
- The set of initial conditions of an expression X **seq** Y is the set of initial conditions of the expression X . The set of final conditions of an expression X **seq** Y is the set of final conditions of the expression Y .
- The set of initial (final) conditions of an expression X **alt** Y or X **par** Y is the intersection of the sets of initial (final) conditions of the MSC reference expressions X and Y .
- The set of initial (final) conditions of an expression **opt** X , **exc** X or **loop** X is the set of initial (final) conditions of the expression X .

The set of initial conditions of an HMSC is defined as follows. If the HMSC start node is followed immediately by one or more condition nodes, then the set of initial conditions is defined as the intersection of the sets of condition names indicated inside the condition nodes. If the HMSC start node is not immediately followed by a condition node then the set of initial conditions is defined to be the complete set of conditions. The set of initial conditions of a parallel frame is the intersection of the sets of conditions of the anonymous HMSCs in the parallel frame.

The set of final conditions of an HMSC is defined as follows. If the HMSC contains one or more condition nodes immediately preceding an end node, then the set of final conditions of the HMSC is defined to be the intersection of the sets of conditions indicated in these condition nodes. If there is no end node that is immediately preceded by a condition node the set of final conditions is defined to be the complete set of conditions. The set of final conditions of a parallel frame is the intersection of the sets of final conditions of the anonymous HMSCs inside the parallel frame.

The set of initial (final) conditions of an inline expression is defined as follows:

- The set of initial (final) conditions of an alternative or parallel inline expression is the intersection of the sets of initial (final) conditions of the operands.
- The set of initial (final) conditions of an optional, exception or loop inline expression is the set of initial (final) conditions of the operand.

If an MSC contains a global condition that precedes everything except the instance head symbols then the set of initial conditions is given by the condition names indicated in this condition. Otherwise, the set of initial conditions is defined to be the

complete set of conditions. If an MSC contains a global condition that follows everything except the instance end symbols then the set of final conditions is given by the condition names indicated in this condition. Otherwise, the set of final conditions is defined to be the complete set of conditions.

An example of a diagram that respects the requirements is given in Figure 2.47. The set of final conditions of MSC *A* is given by $\{C, F\}$, the set of initial conditions of MSC *B* by $\{C, I\}$ and the set of conditions in the condition node is given by $\{C\}$. Clearly, $\{C\} \subseteq \{C, F\}$ and $\{C\} \subseteq \{C, I\}$.

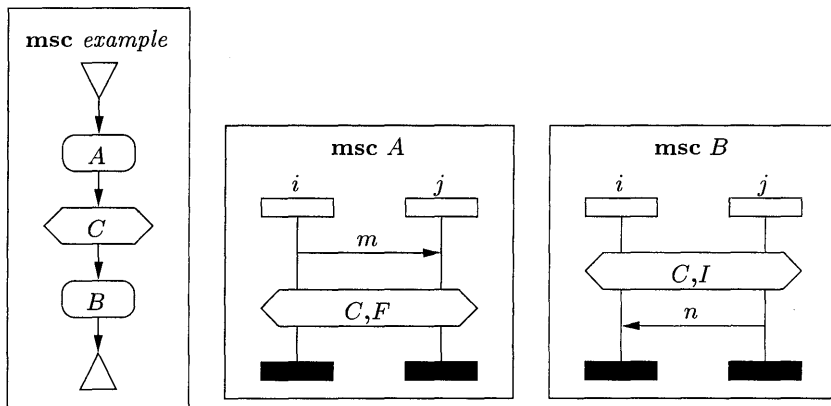


Figure 2.47: An HMSC that respects the requirements.

The arrows in an HMSC can form cycles. This indicates repetitive behavior. Figure 2.48 shows an HMSC that contains a cycle. This HMSC is equivalent to MSC *infinite* shown in the same figure.

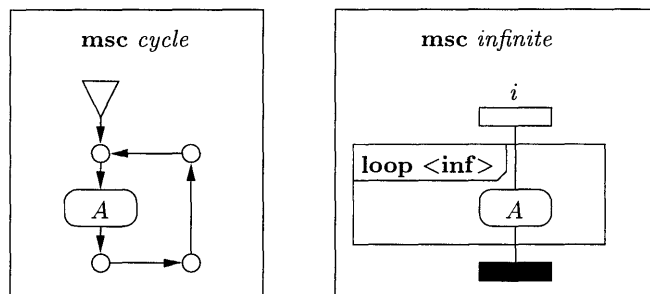


Figure 2.48: A cycle in an HMSC.

An HMSC is described textually by associating a *label name* with every node of the HMSC except the start symbol. The start symbol is implicitly named since there is only one start symbol for every HMSC. The connections between the start symbol and the other nodes are described first. For example if the start symbol has successor

nodes labeled with l_1, \dots, l_4 , this is described by “**expr** l_1 **alt** l_2 **alt** l_3 **alt** l_4 ;”. It indicates that there is an arrow from the start symbol to the nodes labeled with l_1, \dots, l_n . Then for every node of the HMSC in isolation its type/contents is described possibly followed by a list of its successor nodes in a “node expression”. For example if the HMSC contains a node labeled l with successor nodes labeled l' and l'' and this node labeled l is a reference to an MSC named A , then this is described as follows: “ $l : A$ **seq** (l' **alt** l'') ;”. The textual syntax of HMSCs is given in Figure 2.13.

<code><message sequence chart></code>	<code>::=</code>	mssc <code><mssc head></code> expr <code><mssc expression></code> endmssc <code><end></code>
<code><mssc expression></code>	<code>::=</code>	<code><start></code> <code><node expression></code> *
<code><start></code>	<code>::=</code>	<code><label name></code> { alt <code><label name></code> }* <code><end></code>
<code><node expression></code>	<code>::=</code>	<code><label name></code> : { <code><node></code> seq (<code><label name></code> { alt <code><label name></code> }*) end } <code><end></code>
<code><node></code>	<code>::=</code>	empty <code><mssc name></code> <code><par expression></code> condition <code><condition name list></code> connect (<code><mssc ref expr></code>)
<code><par expression></code>	<code>::=</code>	expr <code><mssc expression></code> endexpr { par expr <code><mssc expression></code> endexpr }*

Table 2.13: The textual syntax for High-level Message Sequence Charts.

The MSC from Figure 2.43 is represented textually by

```

mssc example ;
expr l0 ;
l0 : Initialization seq (l1) ;
l1 : connect seq (l2) ;
l2 : Communicate seq (l3) ;
l3 : connect seq (l4 alt l7) ;
l4 : connect seq (l5) ;
l5 : Fail seq (l6) ;
l6 : connect seq (l1) ;
l7 : connect seq (l8) ;
l8 : end ;
endmssc ;

```

A requirement on the textual representation of HMSC is that for every label name used to describe the successors of a node there should be a node labeled by that name.

2.6 Gates

2.6.1 Formal gate definitions

When describing industrial systems by means of Message Sequence Charts as presented so far one of the biggest problems is the number of instances and the number of events on these instances. The diagrams easily get too big to be handled, printed, read, etc. In order to solve this problem complex MSCs must be decomposed into smaller MSCs. In general, it is impossible to do this by means of horizontal or vertical composition without ever having to cut a message or causal ordering in two parts, where one part is located in the one component and another part is located in another component. An example of such an MSC is given in Figure 2.49. To facilitate this, gates are introduced.

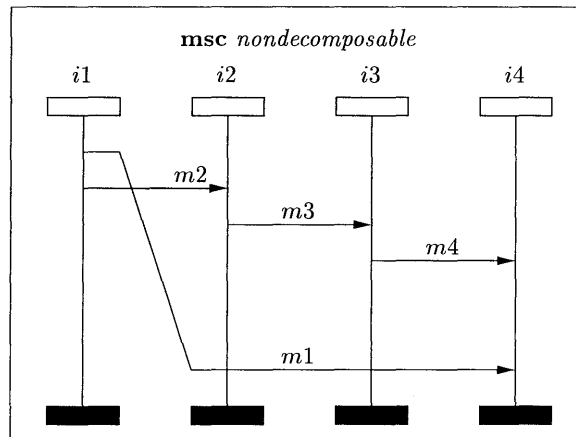


Figure 2.49: An MSC that cannot be decomposed.

Gates are implicitly or explicitly named parts of the environment. As such they can be used to describe the interface between an MSC and its environment. Any message arrow or causal order arrow attached to the frame of an MSC defines a gate. In the recommendation there are two types of gates: *message gates* and *order gates*. Message gates are used for message events and order gates are used for causal orderings.

Graphically an explicitly named gate is indicated by associating a gate name with the place where a message arrow or causal order arrow is attached to the frame of the MSC, i.e. the environment. A message gate always has a name, either explicitly given or implicitly defined. By associating a name with the gate on the frame of the MSC the *gate name* is explicitly defined. Otherwise the name is given implicitly by the direction of the message through the gate and the message identifier¹. For example if we have a message *m* from instance *i* to the frame of the MSC without explicitly

¹In the recommendation the implicit gate name is given by the direction of the message through the gate and the message name. The message name is not discriminating enough.

associating a gate name with the gate, implicitly the name is defined to be *out.m*. Notice that the direction of the message through the gate is chosen from the point of view of the environment; although we see a message *m* going into the gate, its direction is *out*. Examples of explicitly named message gates are the message gates *g1* and *g2* in Figure 2.50. Examples of explicitly named order gates are the order gates *g1* and *g2* in Figure 2.51. Graphically it is only possible to distinguish the two types of gates, message gates and order gates, by means of the type of arrow associated to it. If this is a message arrow the gate is a message gate; if it is a causal order arrow, the gate is an order gate.

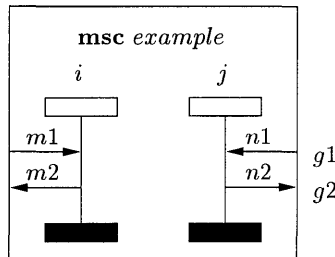


Figure 2.50: MSC to illustrate message gates.

Consider MSC *example* from Figure 2.50. For this MSC there are four message gates, two implicitly named on the left and two explicitly named on the right. The gates associated with the messages *m1* and *m2* that are sent to or received from the environment are implicitly named *in_m1* and *out_m2* respectively. The gates associated with the messages *n1* and *n2* are explicitly named by associating a gate name with the place where the messages are attached to the MSC frame.

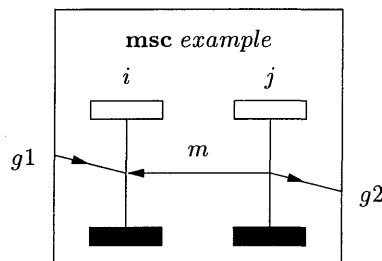


Figure 2.51: MSC to illustrate order gates.

In general, order gates are treated similarly as message gates, though there is an important difference. Order gates always have to be named explicitly. The reason for this is that it is impossible to determine whether two causally ordered events are corresponding, based on the events alone. With an order gate also a direction is associated from the event to be executed first to the event to be executed thereafter. It is required that gates that are connected when composing MSCs should be consistent in the sense that an order gate with direction *out* may only be connected to an order gate with direction *in* and vice versa. See Figure 2.51 for an example of an MSC with

two order gates $g1$ and $g2$. The gate indicated by the name $g1$ is called an *order in gate* and the gate with gate name $g2$ is an *order out gate*. Also in the case of order gates the direction is motivated from the perspective of the environment.

Besides message arrows from an instance to the environment and message arrows from the environment to an instance, it is also allowed to draw a message arrow from the environment to the environment. In this case no event is associated with the arrow.

So far we have only considered gates as the input address of message output events and as the output address of message input events. The recommendation also allows a gate to be used as the input address of a lost message output event or as the output address of a found message input event. An example of these is given in Figure 2.52. Recommendation Z.120 is not so clear in the representation of lost and

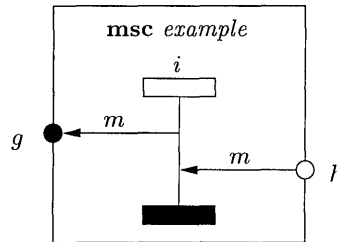


Figure 2.52: Lost and found messages and gates.

found messages with a gate as an output or input address. We assume that it is the intention to represent those graphically in this way since otherwise they cannot be distinguished from lost and found message events that are sent to an instance.

It is even possible to draw an MSC where a lost message output event or a found message input event is attached to the frame of the MSC. Examples of these are given in Figure 2.53. Also in this case no events are associated with the arrows.

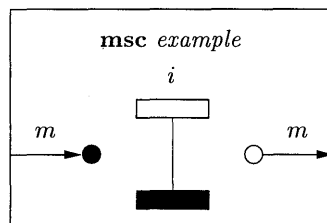


Figure 2.53: Lost and found messages on the MSC frame.

As already explained in the motivation, the intended use of gates is to compose and decompose large specifications and descriptions into more tractable pieces. This will become more apparent in Section 2.6.2 and Section 2.6.3 on the combination of gates with MSC reference expressions and inline expressions respectively. For now we will only discuss the aspects related to gates on the MSC-level without these composition

mechanisms. For the semantics of gates we refer to the upcoming two sections where gates are attached to MSC reference expressions and inline expressions.

Textually, the name of a message gate can be used as an output or input destination for message output and message input events. An implicitly named gate is textually indicated by the keyword **env**. An explicitly named gate is textually represented by the keywords **env** and **via** followed by a gate name. For example, the sending of message *m1* from instance *i* to the environment in MSC *example* in Figure 2.50 is denoted by “*i* : **in** *m1* **from env**” and the sending of message *n2* from instance *j* to the gate *g2* by “*j* : **out** *n2* **to env via g2**”. Textually the MSC from Figure 2.50 can be represented by

```

msc example ;
i  : instance ;
j  : instance ;
i  : in m1 from env ;
i  : out m2 to env ;
i  : in n1 from env via g1 ;
i  : out n2 to env via g2 ;
i  : endinstance ;
j  : endinstance ;
endmsc ;

```

The MSC interface which consists of an optional MSC instance interface is extended with an optional MSC gate interface. The MSC gate interface contains a list of MSC gate definitions. A MSC gate definition is composed of the keyword **gate** followed by a message or order gate. Such a message gate consists of an optional gate name followed by a message output or input event. This message output or input event is described from the perspective of the gate. Thus a message *m* from a gate *g* to an instance *i* is in the MSC gate interface described as “**gate g out m to i**”. An implicitly named gate can be described similarly by leaving the gate name out. An order gate consists of a gate name and, in the case of an order in gate, the keyword **before** followed by an order destination.

The recommendation states that the MSC gate interface provides a definition of message gates contained in the MSC. However, most of the examples in the recommendation do not live up to the expectations obtained from this statement: mostly there simply is no MSC gate interface although there are messages to the environment. The MSC gate interface for the MSC of Figure 2.50 can be described as follows:

```

gate out m1 to i ;
gate in m2 from i ;
gate g1 out n1 to j ;
gate g2 in n2 from j ;

```

Within an MSC it is also allowed to draw a message from the environment to the environment. In the textual representation such “messages” clearly represent exceptional cases. As there is no instance which sends or receives the message it is impossible to

describe the defining occurrences of the implicitly or explicitly defined gates in the MSC body. For this purpose the MSC gate interface is used. A message m from a gate g to a gate h is described by “**gate g out m to env via h** ” and/or “**gate h in m from env via g** ”. Similarly a message from an implicitly defined gate to an implicitly defined gate is described by “**gate out m to env**” and/or “**gate in m from env**”.

Textually, a causal order arrow from an event on an instance i to a gate g is described by “ **$i : e$ before env via g** ”. Thus, the keywords **env** and **via** followed by a gate name can be used to describe the destination of the causal order arrow. However, a causal order arrow from a gate g on the frame of an MSC to an event e on an instance i cannot be described in a similar way. The reason is that the textual syntax lacks a keyword **after**. At the moment, such a causal order arrow can only be described in the MSC gate interface: “**gate g before l** ” where l is an event name that is associated with the event e . The MSC from Figure 2.51 can be described by

```

msc example ;
gate g1 before l ;
i : instance ;
j : instance ;
i : l in m from j ;
j : out m to i before env via g2 ;
i : endinstance ;
j : endinstance ;
endmsc ;

```

Thereby, the MSC gate interface necessarily contains a defining occurrence of a gate and a causal ordering. There is a simple, elegant solution however. If the textual syntax of MSC is extended with a keyword **after** which can be used on all places where **before** is allowed, then the gate $g1$ can be described in the MSC body by “ **$i : in m from j after env via g1$** ”. In Section 2.9.3 we elucidate on the extension of the textual syntax with a keyword **after**.

A causal order arrow can be drawn from a gate to another gate. Textually such a causal order can only be described in the MSC gate interface. For example a causal arrow from a gate g to a gate h is described by

```

msc example ;
gate g before env via h ;
gate h ;
endmsc ;

```

For lost message output events and found message input events the input address and output address respectively can also be an implicitly or explicitly named gate. The textual syntax for these output and input addresses is identical to the syntax for message output and input events. The lost and found messages that are attached to the MSC frame symbol must be represented textually in the MSC gate interface. Also these are described from the point of view of the environment. The MSC from Figure 2.53 is represented textually by

```

msc example ;
gate out m to lost ;
gate in m from found ;
i : instance ;
i : endinstance ;
endmsc ;

```

The textual syntax for gates is given in Table 2.14.

\langle msc interface \rangle	::=	[\langle msc inst interface \rangle] \langle msc gate interface \rangle
\langle msc gate interface \rangle	::=	\langle msc gate def \rangle *
\langle msc gate def \rangle	::=	gate { \langle msg gate \rangle \langle order gate \rangle } \langle end \rangle
\langle msg gate \rangle	::=	\langle def in gate \rangle \langle def out gate \rangle
\langle order gate \rangle	::=	\langle def order in gate \rangle \langle def order out gate \rangle
\langle def in gate \rangle	::=	[\langle gate name \rangle] out \langle msg identification \rangle to \langle input dest \rangle
\langle def out gate \rangle	::=	[\langle gate name \rangle] in \langle msg identification \rangle from \langle output dest \rangle
\langle def order out gate \rangle	::=	\langle gate name \rangle
\langle def order in gate \rangle	::=	\langle gate name \rangle before \langle order dest \rangle
\langle output dest \rangle	::=	found [\langle output address \rangle] \langle output address \rangle
\langle output address \rangle	::=	env via \langle gate name \rangle
\langle input dest \rangle	::=	lost [\langle input address \rangle] \langle input address \rangle
\langle input address \rangle	::=	env via \langle gate name \rangle
\langle order dest \rangle	::=	env via \langle gate name \rangle

Table 2.14: The textual syntax for formal gate definitions.

2.6.2 MSC reference expressions and gates

In the previous section we have seen how gate definitions can be described both graphically and textually. In this section, we will extend the syntax for MSC reference expressions with gates. An MSC reference expression is indicated graphically by a textual formula in an MSC reference symbol. As the MSCs referenced in the textual formula can have gates, it should be possible to connect gates from referenced MSCs. For this purpose actual gates are used. An actual gate is defined by connecting a message arrow with the MSC reference expression symbol. By placing a gate name close to the point of connection an explicitly named actual gate is defined. If the gate name is omitted an implicitly named actual gate is defined. In Figure 2.54 the different occurrences of gates are named.

The actual gates of an MSC reference expression may connect to corresponding constructs in the enclosing MSC. An actual message gate (on an MSC reference symbol) may connect to another actual message gate, an instance, or a message gate definition (implicitly or explicitly named) of the enclosing MSC by means of a message arrow. Similarly, an actual order gate may connect to another actual order gate, an orderable

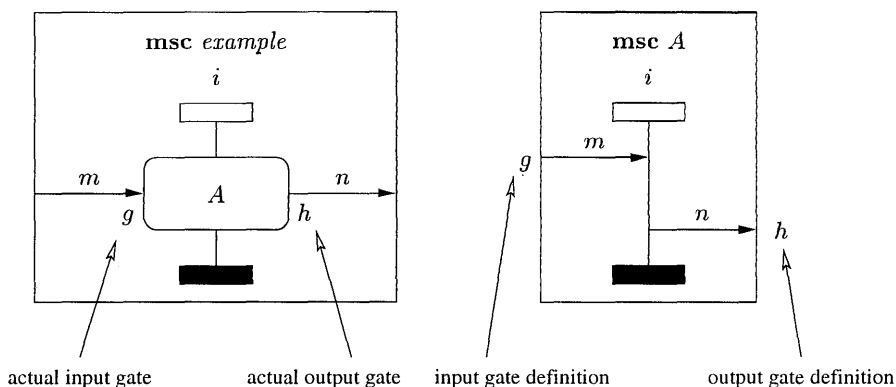


Figure 2.54: Terminology on gates.

event, or an order gate definition of the enclosing MSC by means of a causal order arrow.

A message arrow can only be connected to an MSC reference symbol if at least one of the MSCs that are referenced has a corresponding gate. If a message m is sent to an actual input gate g of an MSC reference expression, then the MSC reference expression must contain a reference to an MSC with an input gate definition of gate g for a message m . If a message n is received from an actual output gate h of an MSC reference expression, then the MSC reference expression must contain a reference to an MSC with an output gate definition of gate h for a message n . For implicitly named message gates similar requirements hold. In that case there must be an implicitly named gate with the same message identifier. Examples of the graphical appearance of such connections are given in Figure 2.55.

If a message arrow is connected to an MSC reference symbol and more than one of the MSCs referenced in this MSC reference symbol have a corresponding gate, then it is required that the instances to which these gates are connected internally (in the referenced MSCs) are identical. The reason for this additional requirement is that we are not capable of distinguishing between the two occurrences of the same gate, neither in the graphical syntax nor in the textual syntax.

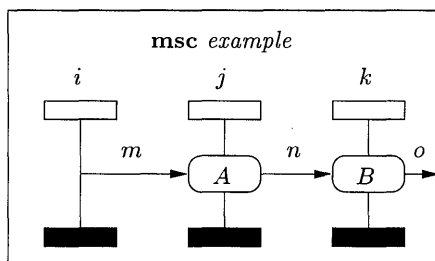


Figure 2.55: Gates on MSC reference expressions.

It is important to define the gates of an MSC reference expression as the above explanation refers to this notion. The set of gates of an MSC reference expression is the union of the sets of gates of the MSCs referenced by that expression.

It is allowed to connect two message gates from the same MSC reference expression in an enclosing MSC. An example of this situation is the MSC given in Figure 2.56. It is also possible to connect gates from different MSCs that are referenced in the same MSC reference expression.

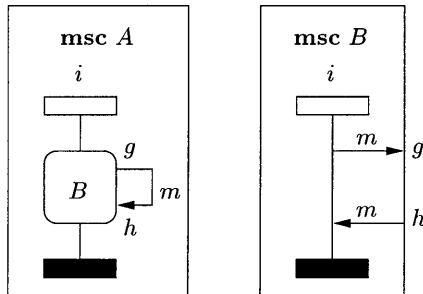


Figure 2.56: Connecting gates from the same MSC reference expression.

MSC reference expressions with gates that are connected on the outside of the MSC reference symbol describe how a message or causal order arrow is continued outside the MSC reference symbol. For the MSC in Figure 2.57 a message arrow is drawn from instance i to the MSC reference expression. This means that a message m is sent by instance i to the receiver of the corresponding message input event in MSC A. In this case this is instance j .

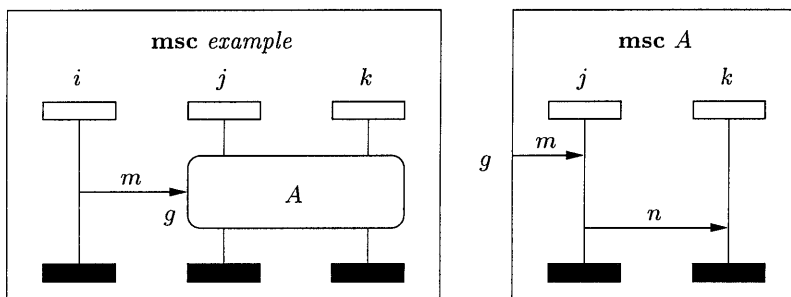


Figure 2.57: Connecting a gate.

It is also possible to connect the gates of two MSC reference expressions by means of a message arrow (see Figure 2.55). If a gate of an MSC reference A is connected to a gate of an MSC reference expression B by means of a message arrow with message identifier n this means that the output of message n inside MSC A is connected to the input of message n inside MSC B . Note that according to the requirements these have to exist.

A last possibility is to connect a gate from an MSC reference expression with a gate of the MSC. This means that the message output or input event is sent to or received from the environment of the enclosing MSC. Also, if a gate of an MSC reference expression is not connected this implicitly means that it is connected to the environment of the enclosing MSC. Examples of both situations are given in Figure 2.58. From a semantics point of view the two MSCs *A* are equivalent.

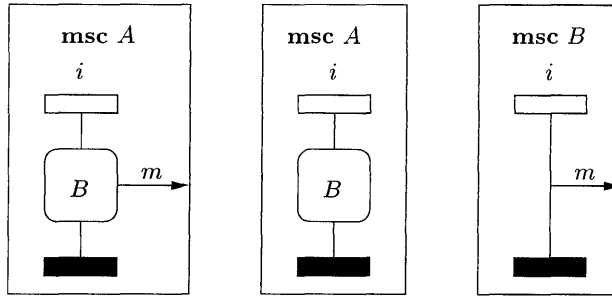


Figure 2.58: Propagation of a gate to the environment.

So far we have only indicated the meaning of connecting gates in the case that the MSC reference expression is only a reference to an MSC by means of its name. However, MSC reference expressions can easily become more complex. For example the MSC reference expression can be the alternative composition of two MSC reference expressions by means of the keyword **alt**. It can be the case that one MSC reference expression has a gate *g* and the other has no such gate. An example of this situation is given in Figure 2.59.

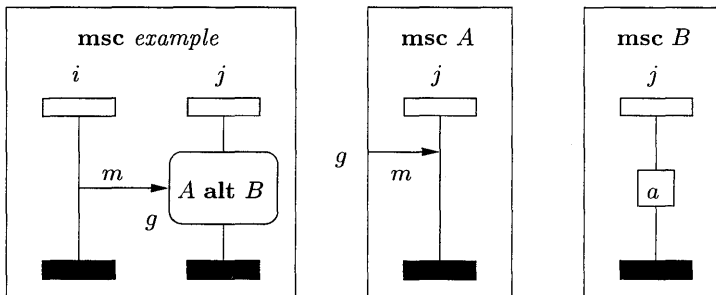


Figure 2.59: Connecting a gate.

In case that the MSC *A* is selected for execution, the MSC can only perform the sending of message *m* and its subsequent reception. On the other hand, if MSC *B* is selected, we expect the execution of local action *a* and the output of message *m* in an arbitrary order. Note that in this case, the input of message *m* does not take place. This gives rise to message output events without corresponding message input event.

Thus it is possible that a message is sent by an instance to an instance while the receiver instance never receives the message. Conversely, the situation where a mes-

sage is received from an instance while it has never been sent cannot occur. If such a situation arises a deadlock results.

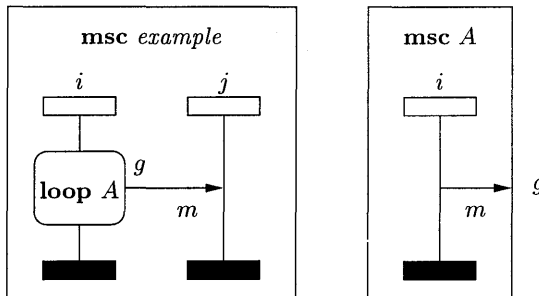


Figure 2.60: Gates and loops.

In Figure 2.60 an MSC *example* is given that refers to an MSC *A* by means of the MSC reference expression **loop A**. Instance *j* receives a message from the gate *g*. In MSC *A* a message is sent to a gate *g*. As a consequence MSC *example* expresses that message *m* is sent an arbitrary number of times, but at least once, to instance *j* and that instance *j* receives message *m* exactly once. The MSC does not specify which occurrence of the sending of message *m* is received. The other occurrences of the sending of message *m* are never received.

<code><msc reference></code>	<code>::=</code>	reference [<code><msc reference identification></code> :] <code><msc ref expr></code> [<code><reference gate interface></code>]
<code><msc reference identification></code>	<code>::=</code>	<code><msc reference name></code>
<code><reference gate interface></code>	<code>::=</code>	{ <code><end gate></code> <code><ref gate></code> }*
<code><ref gate></code>	<code>::=</code>	<code><actual out gate></code> <code><actual in gate></code> <code><actual order out gate></code> <code><actual order in gate></code>
<code><actual out gate></code>	<code>::=</code>	[<code><gate name></code>] out <code><msg identification></code> to <code><input dest></code>
<code><actual in gate></code>	<code>::=</code>	[<code><gate name></code>] in <code><msg identification></code> from <code><output dest></code>
<code><actual order out gate></code>	<code>::=</code>	<code><gate name></code> before <code><order dest></code>
<code><actual order in gate></code>	<code>::=</code>	<code><gate name></code>
<code><output address></code>	<code>::=</code>	<code><reference identification></code> [via <code><gate name></code>]
<code><input address></code>	<code>::=</code>	<code><reference identification></code> [via <code><gate name></code>]
<code><order dest></code>	<code>::=</code>	<code><reference identification></code> via <code><gate name></code>
<code><reference identification></code>	<code>::=</code>	reference <code><msc reference identification></code>

Table 2.15: The textual syntax for actual gates on MSC reference expressions.

The textual syntax of MSC reference expressions with gates is given in Table 2.15. It extends the description of MSC reference expressions without gates (Table 2.11) with an optional *MSC reference identification* and with an optional *reference gate interface*.

The MSC reference identification is used to unambiguously identify an MSC reference expression. If a gate on an MSC reference symbol acts as output or input address of a message arrow or as the destination of a causal order arrow, this is described textually by the keyword **reference** followed by an MSC reference identification and, in case the gate is explicitly named, by the keyword **via** and the gate name. The defining occurrence of the MSC reference identification therefore has to be unique.

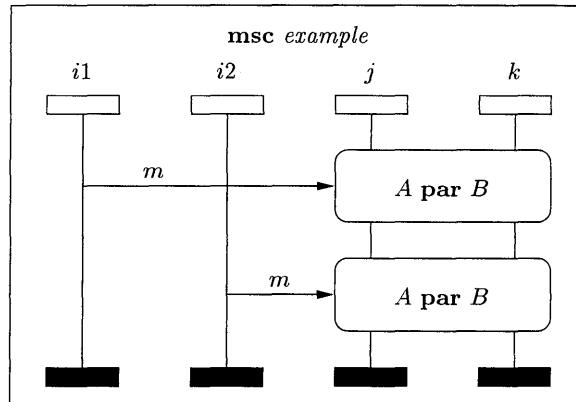


Figure 2.61: MSC where MSC reference identifications are needed.

Consider for example the MSC from Figure 2.61. Graphically it is immediately clear that the output of *m* by instance *i1* refers to the first occurrence of the expression *A par B* and the output of *m* by instance *i2* is to the second occurrence of the expression *A par B*. Textually we need a means to distinguish these two references which have the same appearance. Thereto the MSC reference identification is used. In this example we use *parallel1* and *parallel2* as MSC reference identifications for the first and second occurrence of the expression *A par B* respectively. Textually this MSC is described as follows:

```

msc example ;
i1 : instance ;
i2 : instance ;
j  : instance ;
k  : instance ;
i1 : out m to reference parallel1 ;
i2 : out m to reference parallel2 ;
j , k reference parallel1 : A par B ;
j , k reference parallel2 : A par B ;
i1 : endinstance ;
i2 : endinstance ;
j  : endinstance ;
k  : endinstance ;
endmsc ;

```

With every MSC reference expression a reference gate interface can be associated. This interface describes how the gates of the MSCs that are referenced in the MSC

reference expression are connected externally. If a gate of the MSC reference expression is not connected externally, no entry in the reference gate interface is required. Syntactically the entries in this interface are described similar to the descriptions of the gates in the MSC gate interface.

2.6.3 Inline expressions and gates

Graphically an inline expression is indicated by an inline expression symbol or an exception inline expression symbol. A message arrow or causal order arrow that is attached to the inline expression symbol constitutes a gate definition. At the same time a continuation of this arrow in the enclosing MSC describes a connection of this gate. Thus, for inline expressions the definition of a gate (of the anonymous MSC) coincides with the use of the gate (the actual gate). As was the case for gates on MSC reference symbols the actual gates can be named explicitly or implicitly. In Figure 2.62 the gate definitions and actual gates are indicated.

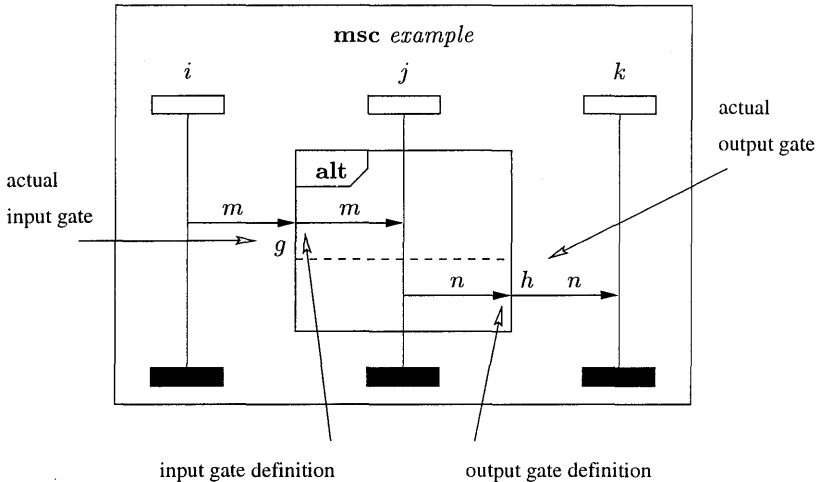


Figure 2.62: Terminology of gates on inline expressions.

The inline expression in Figure 2.63 has two implicitly named gates. These are both connected outside the inline expression symbol by means of message arrows.

If a message arrow or causal order arrow is connected to the inline expression symbol internally, but not externally this indicates that the gate propagates to the frame of the enclosing MSC. The gate name remains the same.

If an inline expression has multiple occurrences of the same gate in different operands, then either there is an external connection for exactly one occurrence of the gate which is supposed to apply to all occurrences, or none of the occurrences of the gate is connected externally in which case it is assumed that the occurrences of the gate propagate to the enclosing frame.

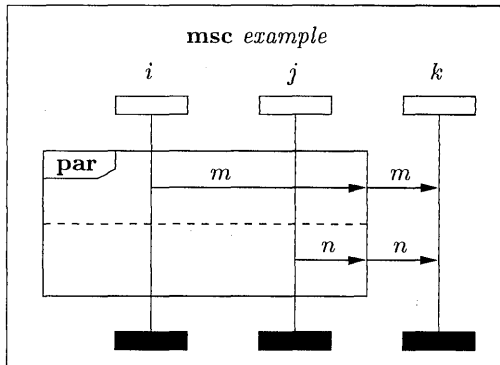


Figure 2.63: Example of an inline expression with gates.

For all occurrences of a gate on an inline expression the internal address of the different occurrences of this gate must be identical. The reason for this requirement is that in the textual syntax there is no means to distinguish the different occurrences of the gate.

The textual syntax of inline expressions with gates is given in Table 2.16. The introduction of an *inline expression identification* in the textual representation of inline expressions with gates is motivated similarly as the introduction of the MSC reference identification in the previous section. If a gate of an inline expression is the output or input address of a message arrow this is described by means of the keyword **inline** followed by the inline expression identifier and, in case the gate is explicitly named, by the keyword **via** and the gate name.

For each operand of the inline expression an *inline gate interface* can be described. Such an inline gate interface describes both the internal and external connections of the gates on the inline expression symbol.

The MSC from Figure 2.63 is textually described by

```

msc example ;
i  : instance ;
j  : instance ;
k  : instance ;
i, j : par begin l ;
      gate in m from i external out m to k ;
i  :   out m to env ;
      par
        gate in n from j external out n to k ;
j  :   out n to env ;
      par end ;
k  : in m from inline l ;
k  : in n from inline l ;
i  : endinstance ;

```

```

j : endinstance ;
k : endinstance ;
endmsc ;

```

<code><loop expr></code>	<code>::= loop [<loop boundary>] begin [<inline expr identification>] <end> [<inline gate interface>] <msc body> loop end</code>
<code><opt expr></code>	<code>::= opt begin [<inline expr identification>] <end> [<inline gate interface>] <msc body> opt end</code>
<code><exc expr></code>	<code>::= exc begin [<inline expr identification>] <end> [<inline gate interface>] <msc body> exc end</code>
<code><alt expr></code>	<code>::= alt begin [<inline expr identification>] <end> [<inline gate interface>] <msc body> { alt <end> [<inline gate interface>] <msc body> }* alt end</code>
<code><par expr></code>	<code>::= par begin [<inline expr identification>] <end> [<inline gate interface>] <msc body> { par <end> [<inline gate interface>] <msc body> }* par end</code>
<code><inline expr identification></code>	<code>::= <inline expr name></code>
<code><inline gate interface></code>	<code>::= { gate <inline gate> <end> }⁺</code>
<code><inline gate></code>	<code>::= <inline out gate> <inline in gate> <inline order out gate> <inline order in gate></code>
<code><inline out gate></code>	<code>::= <def out gate> [external out <msg identification> to <input dest>]</code>
<code><inline in gate></code>	<code>::= <def in gate> [external in <msg identification> from <output dest>]</code>
<code><inline order out gate></code>	<code>::= <gate name> [external before <order dest>]</code>
<code><inline order in gate></code>	<code>::= <gate name> before <order dest> [external]</code>
<code><reference identification></code>	<code>::= inline <inline expr identification></code>

Table 2.16: The textual syntax for actual gates on inline expressions.

2.7 Comments

The MSC language offers several features for presenting informal comments. These are

- the “comment” which can be used in both graphical and textual descriptions,
- the “text” which can also be used in both graphical and textual representation, and
- the “note” which occurs only in the textual descriptions.

In Figure 2.64 we give the symbols for informally describing comments which can be used in the graphical syntax. The *comment* symbol can be attached to an enormous amount of symbols. The *text* symbol is placed somewhere in the MSC stand-alone, but usually in the right-upper corner. Both can contain an arbitrary text (generated by the nonterminal $\langle \text{text} \rangle$).

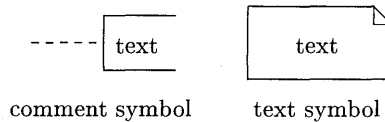


Figure 2.64: Symbols for representing comments.

The textual syntax for comments, text and notes is given in Table 2.17. The comment is in the textual syntax allowed on all places where the nonterminal $\langle \text{end} \rangle$ is used. The note can be inserted before or after any lexical unit. It is an arbitrary text in between $/^*$ and $*/$.

$\langle \text{comment} \rangle$::=	comment $\langle \text{character string} \rangle$
$\langle \text{text definition} \rangle$::=	text $\langle \text{character string} \rangle \langle \text{end} \rangle$
$\langle \text{end} \rangle$::=	[$\langle \text{comment} \rangle$] ;
$\langle \text{msc statement} \rangle$::=	$\langle \text{text definition} \rangle$
$\langle \text{msc statement} \rangle$::=	$\langle \text{text definition} \rangle$

Table 2.17: The textual syntax for comments.

In the recommendation the use of comment symbols in the graphical syntax and the occurrences of the nonterminal $\langle \text{end} \rangle$ is not consistent. For example the recommendation does not allow the comment symbol to be attached to the MSC frame symbol, although the textual description allows a comment to be associated with the MSC. Probably there are more such situations.

2.8 Instance decomposition

An instance in an MSC can refer to entities at different levels of abstraction. To enable the description of the relation between different levels of abstraction, a mechanism for decomposing an instance into a collection of instances is included in recommendation Z.120.

An instance at a high level of abstraction can represent a number of instances at a lower level of abstraction and vice versa. In the language MSC such an instance is called a *decomposed instance*. Graphically, this is indicated by putting a reference to an MSC in the instance head symbol by means of the keywords **decomposed as** followed by the name of an MSC. This name refers to an MSC which describes the decomposed instance at a lower level of abstraction. This MSC is called the *refining MSC*. The keyword **as** and the MSC name can be omitted. In that case, it is assumed that the decomposed instance is described in an MSC with the same name as the decomposed instance. An example of a decomposed instance and a refining MSC is given in Figure 2.65.

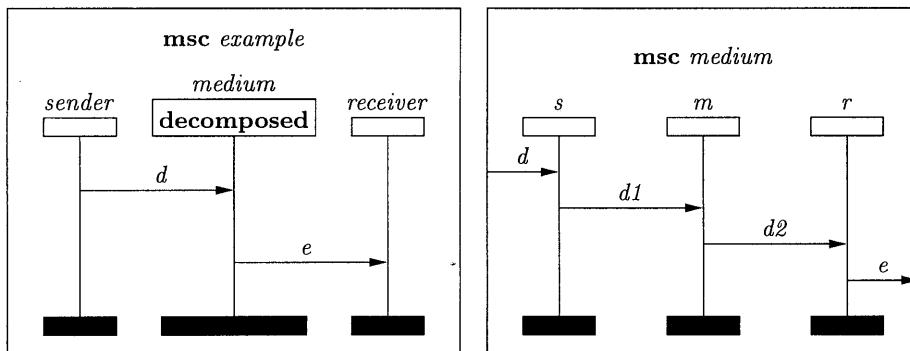


Figure 2.65: A decomposed instance and its refining MSC.

For every decomposed instance in an MSC document there must be a refining MSC in the same MSC document. For every message output of the decomposed instance there has to be a corresponding message output to the environment in the refining MSC. Similarly, for every message input of the decomposed instance there has to be a corresponding message input from the environment in the refining MSC.

The ordering described between message output and message input events on the decomposed instance has to correspond to the ordering described for their corresponding events in the refining MSC. For the example in Figure 2.65 this clearly is the case.

In the textual representation, an instance that is decomposed is labelled with the keywords **decomposed as** followed by the name of an MSC². Again, the keyword **as** and the MSC name can be omitted indicating that the refining MSC has the same

²Recommendation Z.120 uses the nonterminal (message sequence chart name) instead of (msc name). This mistake is corrected in Table 2.18.

name as the decomposed instance. In Table 2.18 the rules for the extension with instance decomposition are given.

$\langle \text{instance head statement} \rangle$	$::=$	instance [$\langle \text{instance kind} \rangle$] $\langle \text{decomposition} \rangle$
$\langle \text{decomposition} \rangle$	$::=$	decomposed [$\langle \text{substructure reference} \rangle$]
$\langle \text{substructure reference} \rangle$	$::=$	as $\langle \text{msc name} \rangle$

Table 2.18: The textual syntax for instance decomposition.

MSC *example* from Figure 2.65 is textually represented by

```

msc example ;
  sender :   instance ;
  medium :  instance decomposed ;
  receiver : instance ;
  sender :   out d to medium ;
  medium :  in d from sender ;
  medium :  out e to receiver ;
  receiver : in e from medium ;
  sender :   endinstance ;
  medium :  endinstance ;
  receiver : endinstance ;
endmsc ;

```

The following static requirements are formulated. With every decomposed instance a refining MSC name has to be defined in the MSC document. On a decomposed instance no create events may be specified. A decomposed instance may not be created. After replacing all decomposed instances of an MSC by their corresponding MSCs the resulting MSC has to respect all previously mentioned requirements. A decomposed instance may not be refined by the MSC it is defined in, directly or via a number of refinements.

2.9 Remarks on recommendation Z.120

2.9.1 Informal parts of the recommendation

Several parts of the language definition of MSC96 are defined informally in the recommendation [Mau96]. These are the definition of the graphical syntax (including the drawing rules), the static semantics, the relation between graphical and textual syntax, and the dynamic semantics. The fact that the textual syntax is the only part of the language definition of MSC96 that is defined formally is the reason to base the definition of the dynamic semantics on it.

There are some serious drawbacks related to the informal definition of the language. First, the user is mainly interested in the graphical representation of MSC and therefore it would be an advantage to base the definition of the dynamic semantics on the graphical syntax. This however is not possible as the graphical syntax is defined informally. Secondly, as there is no formal definition of the relation between the graphical and textual syntax and the textual syntax is mainly used for exchanging MSCs between tools, it is possible that the user is confronted with “different” MSC diagrams in different tools which however are based on the same textual representation. Due to the lack of a formally defined relation between the graphical and textual syntax each tool builder has to make this relation explicit himself. As a consequence different interpretations come into existence. Third, the lack of a formal definition of the static semantics has a similar disadvantage. One tool may consider a given MSC in textual representation well-formed while another tool may reject it. Last, but certainly not least important, and related to the second disadvantage, is the fact that it is hard to establish if the two syntaxes (graphical and textual) have equal expressive power, i.e. there is a diagram for each MSC in textual representation and vice versa.

With the graphical syntax of MSC as it is presented in the recommendation it is possible to draw ambiguous MSCs [LRH97]. This is another reason to base the definition of the dynamic semantics on the textual syntax.

2.9.2 Process creation and termination with composition

In [LRH97] the authors describe an interesting inconsistency between the graphical syntax and the textual syntax. It is not possible to describe a created or terminated instance inside an inline expression following the graphical syntax. In the textual syntax, however, this is not explicitly disallowed.

In MSC reference expressions it is allowed to describe created and terminated instances for both the graphical and textual syntax. MSC reference expressions and inline expressions are different ways of describing composed MSCs. As the composition mechanisms in both are the same except for the vertical composition, which is not available for inline expressions, the two mechanisms should also have the same requirements.

There is a good reason to disallow the description of created instances and terminated instances. Suppose that two MSCs are composed vertically and that they contain the same instances. If in the second MSC an instance is created then it is not clear what this means in the context of the vertical composition as the instance already exists in the first MSC. Similarly, if the first MSC contains a terminated instance it is not clear what this means for the instance with the same name in the second MSC. Also in the case of the other composition mechanisms situations arise in which the semantics is not clear.

We propose to disallow the use of process creation and stop events inside inline expressions and inside any MSC that is referenced by an MSC reference expression.

Furthermore, as also HMSCs describe the composition of MSCs by means of vertical, horizontal and alternative composition, also the MSCs referenced in an HMSC should respect this requirement.

This requirement is formulated in a very strict way and there are many situations that are disallowed by it which are not harmful at all. For example, if in the second MSC in a vertical composition an instance is created which does not exist in the first MSC then this is acceptable as this situation can easily be interpreted. It is conceivable that less strict requirements can be formulated that only exclude the harmful situations. A drawback of a less strict requirement is that it is harder for the user to determine if the use of a process creation or termination is allowed at a certain point.

2.9.3 The keyword **after**

As mentioned in Section 2.4.2 the textual syntax of causal orderings is asymmetrical. A causal ordering is described only for the event that is associated with the starting point of the causal order arrow. This asymmetry also forces the user to describe a causal order arrow from a gate in the environment in the MSC gate interface (see Section 2.6.1).

We propose to extend the textual syntax with a keyword **after** and we propose to change the textual syntax in such a way that a causal ordering can be described for each event that is associated with the causal order arrow.

As an example of the changes, the MSC of Figure 2.24 can be represented textually by

```

msc causal ;
ij : instance ;
ij : concurrent ;
    k in m from env before l ;
    out n to env ;
    l out o to env after k ;
    endconcurrent ;
ij : endinstance ;
endmsc ;

```

As a consequence it is no longer necessary to describe a causal order arrow from a gate in the environment in the MSC gate interface. The concrete changes to the textual syntax are listed in Appendix A.

The MSC of Figure 2.51 can be represented textually by:

```

msc example ;
i : instance ;
j : instance ;
i : l in m from j after env via g1 ;

```

```
j : out m to i before env via g2 ;  
i : endinstance ;  
j : endinstance ;  
endmsc ;
```


3

Process theory for Message Sequence Charts

3.1 Introduction

In this chapter, we present the process theory which is used in Chapter 4 as the semantical framework. A number of operators is defined operationally in the style of Plotkin [Plo81]. These operators are based on the means of composing Message Sequence Charts as explained informally in the previous chapter. For example, we introduce a delayed choice operator which plays the role of alternative composition in MSCs. All of these operators are based on operators that can be found in the literature.

There are several reasons for defining these operators by means of an operational semantics. First of all, an operational model of Message Sequence Charts will make it easier to link Message Sequence Charts to other formalisms since almost every formalism makes use of a structured operational semantics or a structured operational semantics can easily be provided. Examples are process algebras [Hoa85, Mil89, BW90], Petri nets [Rei85, Jen92, Hee94], automata, etc.

A second important reason is that the use of a process algebra is more difficult in this case. This is due to the combination of several features of the language MSC. We can still give some axioms for reasoning with the semantics of Message Sequence Charts which are sound, but a complete set of axioms for deciding equivalence of the process terms is probably impossible.

The operators introduced in this chapter are based on the basic features present in MSC96 for composing small MSCs into more complex MSCs. These basic features are alternative, horizontal and vertical composition. Their semantical equivalents are de-

layed choice, generalized delayed parallel composition and generalized weak sequential composition. Based on these, also operators are introduced for providing semantics to the means offered by MSC96 to describe infinite behavior. These operators are iteration and unbounded repetition.

This chapter is structured as follows. In Section 3.2 we introduce some terminology on operational semantics as it is used in this thesis. Then, in the following sections, we introduce a number of operators and give some properties of those. These operators are delayed choice, delayed parallel composition, weak sequential composition, generalizations of the last two operators, renaming, iteration and unbounded repetition. In Section 3.10, we prove some properties of the operational semantics, especially that every process term is deterministic.

3.2 Operational semantics

In this section, some terminology is introduced with respect to the mathematical framework used to define an operational semantics. Both terminology and notation have been taken from [BV95]. The goal of an operational semantics is, given an expression denoting a process in a certain state, to describe all possible activities that can be performed by the process in that state and to describe the state of the process after such an activity.

3.2.1 Process expressions

The process terms that are used to represent the states are given by a signature and a way of constructing terms from the constant and function symbols in this signature.

Definition 3.2.1.1 The signature Σ consists of

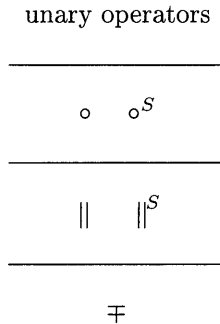
- constants:
 - empty process ε ;
 - deadlock δ ;
 - atomic actions from a set A ;
- unary operators:
 - renaming ρ_f for every injective function $f : A \rightarrow A$;
 - iteration \circledast ;
 - unbounded repetition ∞ ;
- binary operators:
 - delayed choice \mp ;

- delayed parallel composition \parallel ;
- weak sequential composition \circ ;
- generalized parallel composition \parallel^S for every $S \subseteq A \times \mathbb{N} \times A$;
- generalized weak sequential composition \circ^S for every $S \subseteq A \times \mathbb{N} \times A$.

The set of constants A is considered a parameter of the theory. From a signature, terms can be constructed according to some construction rules.

Definition 3.2.1.2 Let Σ be a signature and let V be a set of variables. A Σ -term is defined inductively as follows: For every variable $x \in V$, x is a Σ -term, for every constant $c \in \Sigma$, c is a Σ -term, and for every n -ary function symbol $f \in \Sigma$ and t_1, \dots, t_n Σ -terms, $f(t_1, \dots, t_n)$ is a Σ -term. A closed Σ -term is a Σ -term in which no variables occur. The set of all Σ -terms is denoted by $\mathcal{O}(\Sigma)$ and the set of all closed Σ -terms is denoted by $\mathcal{C}(\Sigma)$. If it is clear from the context which signature is intended we also use term and closed term instead of Σ -term and closed Σ -term.

For the binary operators we use infix notation, e.g. we write $x \mp y$ instead of $\mp(x, y)$. The operators have the following relative binding power ordered from top to bottom by decreasing binding power:



Thus the process expression $a \circ b \parallel c \mp d$ should be read as $((a \circ b) \parallel c) \mp d$. Operators that are indicated at the same line have an equal binding power. For operators with equal binding power brackets are associated from the right. Thus the process expression $a \circ b \circ^S c \circ d$ should be read as $a \circ (b \circ^S (c \circ d))$.

3.2.2 State transformations

The activities that are considered for the operational semantics of MSC96 are the execution of an event and the termination of the MSC. Also the states resulting after such activities are described by expressions. If from a state s an event a can be performed and the resulting state is represented by the expression s' , then this is

usually denoted by $s \xrightarrow{a} s'$. If in a given state s the process is capable of terminating immediately and successfully, this is indicated by means of $s\downarrow$.

The predicate $\downarrow \subseteq \mathcal{C}(\Sigma)$ is called the *termination predicate* as it indicates that a process has the possibility to terminate immediately and successfully. If we assume that all events are represented by atomic actions from the set A , the ternary relation $\rightarrow \xrightarrow{-} - \subseteq \mathcal{C}(\Sigma) \times A \times \mathcal{C}(\Sigma)$ is called the *transition relation*.

The predicate and the relations are defined by means of deduction rules (operational rules). A deduction rule is of the form $\frac{H}{C}$ where H is a set of premises and C is the conclusion. Each individual premise and the conclusion are of the form $s \xrightarrow{a} s'$ or $s\downarrow$ for arbitrary $s, s' \in \mathcal{C}(\Sigma)$ and $a \in A$. Such a deduction rule should be interpreted as follows: If all premises are true, the conclusion holds by definition. A special kind of deduction rule appears if the set of premises is empty ($H = \emptyset$). Such a deduction rule is also called a deduction axiom and usually simply denoted by the conclusion C . An example of a deduction axiom is deduction axiom (At 1) given in Table 3.1:

$$\frac{}{a \xrightarrow{a} \varepsilon.}$$

This deduction axiom expresses that a process that is in a state represented by the atomic action a can perform event a and thereby evolves into a state represented by the expression ε . This expression ε indicates a state in which no events can be performed but in which it is possible to terminate successfully and immediately. This is expressed by the deduction axiom (E 1) also from Table 3.1:

$$\frac{}{\varepsilon\downarrow.}$$

These are the only rules for expressions $a \in A$ and ε . The expression ε is used to denote an MSC without events.

Clearly the process a cannot yet terminate and the process ε cannot perform events. Note that these *negative* results are not explicitly defined. The following convention applies: If it is impossible to derive $s\downarrow$, then by definition *not* $s\downarrow$, which is denoted by $s \not\downarrow$. Similarly, if it is impossible to derive $s \xrightarrow{a} s'$, then by definition *not* $s \xrightarrow{a} s'$. This is usually denoted as $s \not\xrightarrow{a} s'$. Such negative results can also be used in the set of premises, and then these are called *negative premises*. If we want to express that a process represented by the expression s can perform a transition labelled with a and we are not interested in the resulting state, this is denoted by $s \xrightarrow{a}$. Formally, it means that there exists a state s' such that $s \xrightarrow{a} s'$. Then $s \xrightarrow{a}$ should be read as there does not exist a state s' such that $s \xrightarrow{a} s'$, or for all states s' we have $s \not\xrightarrow{a} s'$. These abbreviations extend to the relation $\rightarrow \xrightarrow{-} - \subseteq \mathcal{C}(\Sigma) \times A \times \mathcal{C}(\Sigma)$ to be introduced in Section 3.6.

For term deduction systems without negative premises, the notion of a deduction is quite straightforward [GV92]. In the case of term deduction systems with negative

premises, this is not so easy. It is no longer obvious which set of positive formulas can be deduced using the deduction rules. Groote [Gro90] showed that if for each rule the conclusion is in some sense more difficult than each of the premises, there always is a well-defined set of formulas that is deducible. This notion of being less difficult is called a stratification.

In this chapter the term deduction system under consideration has the signature Σ (see Definition 3.2.1.1) and the deduction rules as defined in the various tables throughout the chapter. The term deduction system is stratifiable (see the proof of Theorem 3.10.1). On top of this term deduction system a notion of equivalence is defined. This notion of equivalence is called *bisimilarity*. It originates from Park [Par81], but we use the formulation of [BV95].

Definition 3.2.2.1 (Bisimulation relation) A binary relation $B \subseteq \mathcal{C}(\Sigma) \times \mathcal{C}(\Sigma)$ is called a bisimulation relation if for all $a \in A$ and $s, t \in \mathcal{C}(\Sigma)$ with sBt the following conditions hold

$$\begin{aligned} \forall s' \in \mathcal{C}(\Sigma) (s \xrightarrow{a} s' \Rightarrow \exists t' \in \mathcal{C}(\Sigma) (t \xrightarrow{a} t' \wedge s'Bt')), \\ \forall s' \in \mathcal{C}(\Sigma) (s \xrightarrow{\dots} s' \Rightarrow \exists t' \in \mathcal{C}(\Sigma) (t \xrightarrow{\dots} t' \wedge s'Bt')), \\ \forall t' \in \mathcal{C}(\Sigma) (t \xrightarrow{a} t' \Rightarrow \exists s' \in \mathcal{C}(\Sigma) (s \xrightarrow{a} s' \wedge s'Bt')), \\ \forall t' \in \mathcal{C}(\Sigma) (t \xrightarrow{\dots} t' \Rightarrow \exists s' \in \mathcal{C}(\Sigma) (s \xrightarrow{\dots} s' \wedge s'Bt')), \end{aligned}$$

and

$$\begin{aligned} s \downarrow \Rightarrow t \downarrow, \\ t \downarrow \Rightarrow s \downarrow. \end{aligned}$$

Two closed terms $p, q \in \mathcal{C}(\Sigma)$ are bisimilar, notation $p \Leftrightarrow q$, if there exists a bisimulation relation B such that pBq .

Theorem 3.2.2.2 (Equivalence) Bisimilarity is an equivalence relation.

Proof Let $s, t, u \in \mathcal{C}(\Sigma)$. Using the identity on closed terms it is easily established that $t \Leftrightarrow t$ and hence bisimilarity is reflexive. Suppose that $s \Leftrightarrow t$. This means that there exists a bisimulation relation B such that sBt . Then clearly also the relation B^{-1} is a bisimulation relation and $tB^{-1}s$. Thus, bisimilarity is symmetrical. Suppose that $s \Leftrightarrow t$ and $t \Leftrightarrow u$. Then there exist bisimulation relations B_1 and B_2 such that sB_1t and tB_2u . Clearly, also the relation $B_1 \circ B_2$ is a bisimulation relation and $s(B_1 \circ B_2)u$. Thus, bisimilarity is transitive. \square

3.3 Deadlock, empty process and atomic actions

In this section we introduce the smallest building blocks of the term algebra. These are divided into the special constants and the atomic actions. There are two special constants: δ and ε . The *deadlock* constant δ represents a process that cannot execute an event and cannot terminate. The *empty* process ε represents a process that cannot execute an event, but contrary to deadlock it terminates successfully.

The set of atomic actions is a parameter of the term algebra. In the context of Message Sequence Charts it is chosen to represent the events of the MSC language such as output and input of a message, timer statements and local actions. As is the case with MSC, each smallest event is defined on an instance. To mimic this in the term algebra the existence of a total mapping $\ell : A \rightarrow Id$ is assumed which associates to an atomic action an identifier representing an instance name.

$$\frac{}{\varepsilon \downarrow} \text{(E 1)} \qquad \frac{}{a \xrightarrow{a} \varepsilon} \text{(At 1)}$$

Table 3.1: Deduction rules for constants: $a \in A$.

The deduction rules for the constants are given in Table 3.1. As indicated before, the empty process ε is capable of terminating immediately and successfully. This is expressed by deduction axiom (E 1). An atomic action a can execute event a and thereby it evolves into the empty process: $a \xrightarrow{a} \varepsilon$ ((At 1)). As the deadlock constant is not capable of executing an event nor capable of terminating, there are no rules for δ in the table. In fact, since there are no deduction rules for δ , we obtain $\delta \not\xrightarrow{a}$, for $a \in A$, and $\delta \not\downarrow$.

3.4 Delayed choice

The structured operational semantics associated with delayed choice by means of the deduction rules presented in Table 3.2 illustrate the purposes of this operator eminently. The deduction rules for \xrightarrow{a} clearly express that $x \mp y$ can perform an a -transition, thereby resolving the choice, if exactly one of its operands can, and in the case that both operands can perform an a -transition, the choice is not yet resolved.

The deduction rules for the termination predicate and the transition relation from Table 3.2 have been taken from [BM95] where the delayed choice operator was introduced in the setting of bisimulation semantics as a means of composing MSCs. The deduction rules (DC 1) and (DC 2) express that the alternative composition of two processes has the option to terminate if and only if at least one of the alternatives has this option.

$\frac{x\downarrow}{x \mp y\downarrow} \text{(DC 1)}$	$\frac{y\downarrow}{x \mp y\downarrow} \text{(DC 2)}$	
$\frac{x \xrightarrow{a} x', y \xrightarrow{a}}{x \mp y \xrightarrow{a} x'} \text{(DC 3)}$	$\frac{x \xrightarrow{a}, y \xrightarrow{a} y'}{x \mp y \xrightarrow{a} y'} \text{(DC 4)}$	$\frac{x \xrightarrow{a} x', y \xrightarrow{a} y'}{x \mp y \xrightarrow{a} x' \mp y'} \text{(DC 5)}$

Table 3.2: Deduction rules for delayed choice.

Example 3.4.1 The process $a \mp \varepsilon$ has an option to terminate as the second alternative has this option. On the contrary the process $a \mp b$ does not have an option to terminate as none of its alternatives can terminate.

The deduction rules (DC 3) and (DC 4) express that, in the situation that exactly one of the alternatives can execute an action a , the alternative composition can execute this event as well and that the execution of this event resolves the choice.

Example 3.4.2 Let a and b be two different atomic actions. The process $a \mp b$ can execute the action a and the action b . In both cases the action can be executed by only one of the alternatives. Thus in both cases making a choice between the alternatives cannot be avoided. Operationally this is seen as follows:

$$a \mp b \xrightarrow{a} \varepsilon \quad \text{and} \quad a \mp b \xrightarrow{b} \varepsilon.$$

Deduction rule (DC 5) deals with the situation that both alternatives can execute an action a . It states that, in that case, the alternative composition can execute a and, moreover, that there remain two alternatives.

Example 3.4.3 The process $a \mp a$ has two alternatives both of which can execute action a . The choice between the alternatives is not resolved. Operationally this can be seen as follows:

$$a \mp a \xrightarrow{a} \varepsilon \mp \varepsilon.$$

Theorem 3.4.4 (Properties of \mp) For all closed terms $s, t, u \in \mathcal{C}(\Sigma)$ we have the following properties:

- deadlock is a unit element for delayed choice: $t \mp \delta \Leftrightarrow t$ and $\delta \mp t \Leftrightarrow t$;
- delayed choice is commutative: $s \mp t \Leftrightarrow t \mp s$;
- delayed choice is associative: $(s \mp t) \mp u \Leftrightarrow s \mp (t \mp u)$;
- delayed choice is idempotent: $t \mp t \Leftrightarrow t$.

Proof These properties are proved in Appendix B.2. ⊠

In [BM95], Baeten and Mauw introduce the delayed choice operator in the framework of $\text{BPA}_{\delta\varepsilon}$ in branching time semantics. The deduction rules are the same as given in Table 3.2.

Baeten and Mauw provide axioms for bisimilarity in which they use auxiliary operators \boxtimes and \triangleleft . These auxiliary operators are useful in eliminating delayed choice in favour of nondeterministic choice. This elimination character of the axioms renders them absolutely useless in a setting in which nondeterministic choice is not a basic operator. That is why we have chosen to express the properties of delayed choice directly.

So far, we have seen that some nice properties hold for delayed choice. Also for nondeterministic choice we would have obtained that deadlock is a unit element and that nondeterministic choice is idempotent, commutative and associative. However, as was already demonstrated by Baeten and Mauw, the delayed choice is not idempotent. The counterexample they give is the process $x \equiv (a \cdot b) + (a \cdot c)$. Observe that $((a \cdot b) + (a \cdot c)) \mp ((a \cdot b) + (a \cdot c)) \not\equiv (a \cdot b) + (a \cdot c)$. Nevertheless, for the process theory defined in this chapter, idempotency of delayed choice holds (Theorem 3.4.4). This is due to the fact that all closed process terms are deterministic (See Theorem 3.10.3). The fact that idempotency of delayed choice holds for deterministic processes (see Definition 3.10.2) was already claimed by Baeten and Mauw in their conclusions. Furthermore, for deterministic processes, bisimulation equivalence and trace equivalence coincide [Eng85]. Also, when considering trace equivalence, the operators delayed choice and nondeterministic choice coincide.

3.5 Delayed parallel composition

The delayed parallel composition of two processes is the interleaved execution of the events of the processes while maintaining the ordering of events as specified by the processes in isolation. This operator is a delayed version of the interleaving operators normally used. If both processes that are composed by means of delayed parallel composition can perform the same event, it is not visible which of the two is actually executed. In other words, a delayed choice is made between the two occurrences. In this aspect the delayed parallel composition operator used for the semantics of MSC differs from the interleaving operators of ACP-style process algebras [BW90]. The deduction rules for the delayed parallel composition operator are given in Table 3.3.

Deduction rule (DP 1) expresses that the delayed parallel composition of two processes has an option to terminate if and only if both processes have this option.

Example 3.5.1 The process $(a \mp \varepsilon) \parallel (b \mp \varepsilon)$ has an option to terminate as both $a \mp \varepsilon$ and $b \mp \varepsilon$ have this option. Operationally this is seen as follows: $a \mp \varepsilon \downarrow$ and $b \mp \varepsilon \downarrow$ and

$\frac{x\downarrow, y\downarrow}{x \parallel y\downarrow} \text{(DP 1)}$	$\frac{x \xrightarrow{a} x', y \xrightarrow{a}}{x \parallel y \xrightarrow{a} x' \parallel y} \text{(DP 2)}$
$\frac{x \xrightarrow{a} x', y \xrightarrow{a} y'}{x \parallel y \xrightarrow{a} x' \parallel y \mp x \parallel y'} \text{(DP 3)}$	$\frac{x \xrightarrow{a}, y \xrightarrow{a} y'}{x \parallel y \xrightarrow{a} x \parallel y'} \text{(DP 4)}$

Table 3.3: Deduction rules for delayed parallel composition.

therefore by deduction rule (DP 1) also $(a \mp \varepsilon) \parallel (b \mp \varepsilon)\downarrow$. The process $a \parallel \varepsilon$ does not have an option to terminate as the left-hand side a of the delayed parallel composition does not have this option ($a \not\downarrow$) and therefore deduction rule (DP 1) is not applicable.

The deduction rules (DP 2) and (DP 4) express that if exactly one of the operands of a delayed parallel composition can execute an action a , then the delayed parallel composition can execute this a as well and it is known which operand has actually executed a .

Example 3.5.2 The process $a \parallel b$ is capable of performing action a and thereby it evolves into the process $\varepsilon \parallel b$. But it is also possible for this process to perform action b and then the process $a \parallel \varepsilon$ remains.

The deduction rule (DP 3) expresses that, in a situation that both operands can execute an action a , the delayed parallel composition can execute an a . It also expresses that it is not known which operand executed a . This is seen in the deduction rule by the term $x' \parallel y \mp x \parallel y'$. The first alternative results from the execution of a by process x and the second from the execution of a by process y . The fact that the process $x \parallel y$ evolves into the process $x' \parallel y \mp x \parallel y'$ indicates that it is not known which a has been executed.

Example 3.5.3 An example illustrating the delayed nature of the delayed parallel composition is the process $a \parallel a$. It can perform the following sequence of transitions:

$$a \parallel a \xrightarrow{a} \varepsilon \parallel a \mp a \parallel \varepsilon \xrightarrow{a} \varepsilon \parallel \varepsilon \mp \varepsilon \parallel \varepsilon \downarrow.$$

Theorem 3.5.4 (Properties of \parallel) For all closed terms $s, t, u \in \mathcal{C}(\Sigma)$ we have the following properties:

- the empty process is a unit for delayed parallel composition: $\varepsilon \parallel t \Leftrightarrow t$ and $t \parallel \varepsilon \Leftrightarrow t$;
- delayed parallel composition is commutative: $s \parallel t \Leftrightarrow t \parallel s$;

- delayed parallel composition distributes over delayed choice:

$$(s \mp t) \parallel u \Leftrightarrow s \parallel u \mp t \parallel u$$

and

$$s \parallel (t \mp u) \Leftrightarrow s \parallel t \mp s \parallel u;$$

- delayed parallel composition is associative: $(s \parallel t) \parallel u \Leftrightarrow s \parallel (t \parallel u)$.

Proof These properties are proved in Appendix B.3. □

In [BW90], an extension of the process algebra $\text{BPA}_{\delta\epsilon}$ with interleaving merge is defined. The resulting process algebra is called $\text{PA}_{\delta\epsilon}$. The interleaving merge operator there is characterized by the following operational rules:

$$\frac{x \downarrow, y \downarrow}{x \parallel y \downarrow}, \quad \frac{x \xrightarrow{a} x'}{x \parallel y \xrightarrow{a} x' \parallel y}, \quad \frac{y \xrightarrow{a} y'}{x \parallel y \xrightarrow{a} x \parallel y'}.$$

With these operational rules it is possible that a process, due to interleaving, makes a nondeterministic choice between the execution of an a from x or an a from y . Consider for example the process $(a \cdot b) \parallel (a \cdot c)$ where \cdot denotes strong sequential composition as it appears in most process theories. This process has two ways to perform an initial a event: $(a \cdot b) \parallel (a \cdot c) \xrightarrow{a} b \parallel (a \cdot c)$ and $(a \cdot b) \parallel (a \cdot c) \xrightarrow{a} (a \cdot b) \parallel c$. This is due to the unfolding of the merge operator into the nondeterministic choice of the processes $x \parallel y$ and $y \parallel x$, where \parallel is an operator that behaves like \parallel except that the first event must be executed by the left-hand side process. In the setting we discuss in this chapter we only have the delayed choice at our disposal. If the situation arises where both operands can execute an event a initially, we want the choice between these to be delayed. Thus we had to propose the following deduction rule

$$\frac{x \xrightarrow{a} x', y \xrightarrow{a} y'}{x \parallel y \xrightarrow{a} x' \parallel y \mp x \parallel y'}.$$

Then the original deduction rules have to be adapted to exclude the possibility that the “other” process also is able to execute event a . For example, the second deduction rule from above became

$$\frac{x \xrightarrow{a} x', y \not\xrightarrow{a}}{x \parallel y \xrightarrow{a} x' \parallel y}.$$

This rule expresses that an a event from process x is executed by itself, only if process y is not capable of performing an a event at this moment. A similar adjustment has to be made for the third, symmetric, rule for parallel composition.

With respect to the process $a \cdot b \parallel a \cdot c$ where \parallel is intended to be the delayed parallel composition we only have

$$a \cdot b \parallel a \cdot c \xrightarrow{a} b \parallel a \cdot c \mp a \cdot b \parallel c.$$

Although these processes do have the same set of traces for the different interpretations of \parallel they are not bisimilar.

3.6 Weak sequential composition

In order to explain the weak sequential composition operator, it is necessary to consider the purpose of this operator in the semantics of MSC. The weak sequential composition operator is introduced to represent the vertical composition of MSCs. It has a behavior similar to the delayed parallel composition operator, but in addition it maintains the ordering of events from instances that the MSCs have in common. Thus an event on instance i in the second MSC can only take place in situations where all events on instance i (if any) in the first MSC have already taken place.

However, there is a complication with respect to alternatives. Suppose that an MSC A is given that describes two alternatives. The first alternative only describes a local action a on instance i and the second alternative only contains a local action b on instance j . Suppose that this MSC is composed vertically with an MSC B that only contains a local action c on instance i . The vertical composition of the first alternative of MSC A with MSC B should not allow the execution of local action c as it must be preceded by local action a . The vertical composition of the second alternative of MSC A with MSC B can execute local action c as there are no events in the second alternative of MSC A that must precede the execution of local action c . Thus, one alternative of MSC A does not allow the execution of local action c and one alternative does allow the execution of local action c . The expected result is that the execution of local action c is allowed and moreover that if local action c is executed the first alternative disappears.

In an MSC every event is associated with an instance on which it is defined. In the operational semantics this is incorporated by assuming a mapping $\ell : A \rightarrow I$, where I represents the set of all instance names, which associates with an atomic action $a \in A$ the name $\ell(a)$ of the instance it is defined on.

In order to deal with this aspect of the weak sequential composition operator the *permission relation* $\dashv\vdash \xrightarrow{a} \dashv\vdash \subseteq \mathcal{C}(\Sigma) \times A \times \mathcal{C}(\Sigma)$ is used. The proposition $x \dashv\vdash \xrightarrow{a} x'$ states that an event a is allowed to precede the execution of the events of x even if this event is composed after x by means of weak sequential composition. The reason to allow such a bypass is that the process x has an alternative that does not execute events from the location of a , i.e. $\ell(a)$. In the case that process x permits the bypass by event a , the actual execution of event a disables all alternatives of x that do not allow the bypass. The proposition $x \dashv\vdash \not\xrightarrow{a}$ indicates that x does not allow the bypass of action a . The reason is that all alternatives of x execute an event from the location of a , i.e. $\ell(a)$.

The deduction rules for the permission relation, for all operators introduced so far, are given in Table 3.4 and the deduction rules for weak sequential composition are

given in Table 3.5. The empty process permits the execution of any event. An event b permits the execution of any event a if it is defined on another location, i.e. $\ell(a) \neq \ell(b)$. The delayed choice of two processes permits an event if at least one of the processes permits the event. If an alternative does not permit the event, it is disabled (removed). The delayed parallel and weak sequential composition of processes permit an event if both processes permit the event. Note that the constant δ does not permit the execution of any event. In this sense it indicates a global deadlock.

Example 3.6.1 The process $a \mp b$ with $\ell(a) \neq \ell(b)$ permits the action b' with $\ell(b) = \ell(b')$. This permission results in the disabling of alternative b . This can be summarized as follows: $a \mp b \xrightarrow{b'} a$.

Example 3.6.2 The process $(a \mp b) \circ a$ with $\ell(a) \neq \ell(b)$ does not permit the action a' with $\ell(a) = \ell(a')$. The reason is that both alternatives ($a \circ a$ and $b \circ a$) can perform an event from the location of a' . Hence, $(a \mp b) \circ a \not\xrightarrow{a'}$.

$$\begin{array}{ccc}
\frac{}{\varepsilon \xrightarrow{a} \varepsilon} \text{(E 2)} & \frac{\ell(a) \neq \ell(b)}{b \xrightarrow{a} b} \text{(At 2)} & \frac{x \xrightarrow{a} x', y \xrightarrow{a} y'}{x \mp y \xrightarrow{a} x' \mp y'} \text{(DC 6)} \\
\\
\frac{x \xrightarrow{a} x', y \xrightarrow{a} y'}{x \mp y \xrightarrow{a} x'} \text{(DC 7)} & & \frac{x \xrightarrow{a} y, y \xrightarrow{a} y'}{x \mp y \xrightarrow{a} y'} \text{(DC 8)} \\
\\
\frac{x \xrightarrow{a} x', y \xrightarrow{a} y'}{x \parallel y \xrightarrow{a} x' \parallel y'} \text{(DP 5)} & & \frac{x \xrightarrow{a} x', y \xrightarrow{a} y'}{x \circ y \xrightarrow{a} x' \circ y'} \text{(WS 5)}
\end{array}$$

Table 3.4: Deduction rules for the permission relation.

Deduction rule (WS 1) expresses that the weak sequential composition of two processes has an option to terminate if and only if both processes have this option.

Example 3.6.3 The process $\varepsilon \circ (a \mp \varepsilon)$ has the option to terminate as both operands have this option: $\varepsilon \downarrow$ and $a \mp \varepsilon \downarrow$.

The deduction rules (WS 2), (WS 3) and (WS 4) deal with the transitions of the vertical composition of two processes. In the case that x can execute a and either y cannot execute a or x does not allow the execution of a by y , only the execution of a by x can take place. This is expressed by deduction rule (WS 2).

Example 3.6.4 Suppose that $\ell(a) \neq \ell(b)$. The process $a \circ b$ can execute action a and evolves into the process $\varepsilon \circ b$ since $a \xrightarrow{a} \varepsilon$ and $b \not\xrightarrow{a}$.

$$\frac{x\downarrow, y\downarrow}{x \circ y\downarrow} \text{ (WS 1)}$$

$$\frac{x \xrightarrow{a} x', x \cdots \xrightarrow{a} \vee y \xrightarrow{a}}{x \circ y \xrightarrow{a} x' \circ y} \text{ (WS 2)} \quad \frac{x \xrightarrow{a} x', x \cdots \xrightarrow{a} x'', y \xrightarrow{a} y'}{x \circ y \xrightarrow{a} x' \circ y \mp x'' \circ y'} \text{ (WS 3)}$$

$$\frac{x \xrightarrow{a}, x \cdots \xrightarrow{a} x', y \xrightarrow{a} y'}{x \circ y \xrightarrow{a} x' \circ y'} \text{ (WS 4)}$$

Table 3.5: Deduction rules for weak sequential composition.

In the case that both x and y can execute action a and x allows the execution of a by y , there are two possibilities for executing action a . A delayed choice of the individual occurrences of action a results. This is expressed by deduction rule (WS 3).

Example 3.6.5 Suppose that $\ell(a) \neq \ell(b)$. The process $(a \mp b) \circ a$ can execute action a and thereby evolves into the process $\varepsilon \circ a \mp b \circ \varepsilon$. The first alternative of the resulting process describes the result of the execution of a by $a \mp b$ and the second alternative describes the result of the execution of a by the process a . Note that due to the execution of the second a , the alternative a from $a \mp b$ is not present anymore since $a \mp b \cdots \xrightarrow{a} b$.

In the case that x cannot execute an action a , y can execute a and x permits the execution of a , there is one possibility of executing a . This is expressed by deduction rule (WS 4).

Example 3.6.6 Suppose that $\ell(a) \neq \ell(b)$. The process $a \circ b$ can execute an action b since the second operand of the vertical composition can ($b \xrightarrow{b} \varepsilon$) and the first operand allows this ($a \cdots \xrightarrow{b} a$). The resulting process after the execution of action b is $a \circ \varepsilon$.

Theorem 3.6.7 (Properties of \circ) For all closed terms $s, t, u \in \mathcal{C}(\Sigma)$ we have the following properties:

- the empty process is a unit element for weak sequential composition: $\varepsilon \circ t \Leftrightarrow t$ and $t \circ \varepsilon \Leftrightarrow t$;
- deadlock is a left-zero element for weak sequential composition: $\delta \circ t \Leftrightarrow \delta$;
- weak sequential composition distributes over delayed choice:

$$(s \mp t) \circ u \Leftrightarrow s \circ u \mp t \circ u$$

and

$$s \circ (t \mp u) \Leftrightarrow s \circ t \mp s \circ u;$$

- weak sequential composition is associative: $(s \circ t) \circ u \Leftrightarrow s \circ (t \circ u)$.

Proof These properties are proved in Appendix B.4. □

The n -times repeated application of weak sequential composition x^n is introduced as a shorthand. No operational rules are given for this operator.

Definition 3.6.8 Let $n \in \mathbb{N}$. Then for $x \in \mathcal{C}(\Sigma)$ the process x^n is defined inductively as follows:

$$x^n = \begin{cases} \varepsilon & \text{if } n = 0, \\ x \circ x^{n-1} & \text{if } n > 0. \end{cases}$$

The standard operator for describing sequential composition in ACP-based process algebras is denoted by \cdot . In this thesis we call it *strong sequential composition*. Operationally this operator is described by

$$\frac{x \downarrow, y \downarrow}{x \cdot y \downarrow}, \quad \frac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y} \quad \text{and} \quad \frac{x \downarrow, y \xrightarrow{a} y'}{x \cdot y \xrightarrow{a} y'}.$$

For axioms for bisimilarity we refer to [BW90]. The strong sequential composition of two processes x and y behaves like process x and upon termination of x it starts behaving like process y . There are two important differences with weak sequential composition. First, the strong sequential composition of two deterministic processes does not have to be deterministic. For example the process $(a \mp \varepsilon) \cdot a$ can execute action a and there can be two different resulting processes, viz. $\varepsilon \cdot a$ or ε . The first of these two possibilities is manifest when the left-hand operand of the strong sequential composition executes action a and the second possibility appears if the left-hand operand terminates and the right-hand operand executes action a . The second difference is that no action from process y can be executed before x has the option to terminate. By defining the permission relation to hold only if there is a summand that can terminate, this difference is overcome. In other words, if $\ell(a) = \ell(b)$ for all $a, b \in A$, then weak sequential composition and strong sequential composition behave similarly with respect to the execution of events by process y . They are still different with respect to determinism.

To capture the notion of placing an Interworking underneath another one Mauw, Van Wijk and Winter defined the *interworking sequencing* (\circ_{iw}) [MvWW93]. The interpretation of $x \circ_{iw} y$ is that actions of y which are independent of x do not have to wait for another to proceed, even if they are composed sequentially. The most interesting operational rule for interworking sequencing is the following

$$\frac{I(a) \notin E(x), y \xrightarrow{a} y'}{x \circ_{iw} y \xrightarrow{a} x \circ_{iw} y'}$$

where $E(x)$ denotes the set of instances (entities in the terminology of Interworkings, locations in our terminology) occurring in process x . In our terminology the predicate $I(a) \notin E(x)$ can be paraphrased as: $\ell(a) \neq \ell(b)$ for every action b that can be executed (after an arbitrary number of steps) by process x . This way the permission to overtake the execution of x is formulated with respect to all branches in the execution of x , and not as we have for the weak sequential composition, with respect to the existence of a branch which allows overtaking. We could express this by saying that interworking sequencing is based on a static permission relation whereas weak sequential composition as we see it, is based on a dynamical notion of permission. However, as all traces of an Interworking consist of the same atomic actions [MvWW93], the notions of dynamic and static permission are equivalent for Interworkings. Another difference is in the permission of δ . With interworking sequencing δ permits the execution of any atomic action whereas with weak sequential composition δ permits no atomic actions. For interworking sequencing the law $\delta \circ_{iw} x = x \circ_{iw} \delta$ holds. For weak sequential composition we have the law $\delta \circ x = \delta$.

A third difference is that the interworking sequencing is not determinism preserving. An important consequence of these differences is that the right-distribution of interworking sequencing over delayed choice does not hold, whereas it does hold for weak sequential composition.

Rensink and Wehrheim [RW94] define a special operator for sequential composition which is defined relative to a dependency relation over the atomic actions. This operator is called *weak sequential composition*, and in their paper denoted by \cdot . In this thesis we will denote this operator by \otimes to prevent confusion and we will call it *weak sequencing*. The interpretation is, as is the case for weak sequential composition, that actions which are independent do not have to wait for another to proceed, even if they are composed sequentially. The permission relation is dynamic and in fact both the idea and the notation for using it for weak sequential composition stem from the paper by Rensink and Wehrheim. However there are two important differences with weak sequential composition. First, choices introduced by weak sequencing are not delayed as is the case for weak sequential composition. Secondly, Rensink and Wehrheim have a family of deadlock constants $\mathbf{0}_S$. Each deadlock constant is labeled with a set S of actions that are permitted. Thus the deadlock constant δ is in their algebra given by $\mathbf{0}_\emptyset$. The constant ε is present in their algebra as $\mathbf{0}_A$. Also for weak sequencing the right-distributivity of weak sequencing over delayed choice does not hold.

In the discussion of related work we have considered three other operators for sequential composition. We have seen that these differ from weak sequential composition in three ways:

- choices that are introduced by the composition are delayed or not;
- permission is static or dynamic;
- the permissions of deadlock constants.

The choice for a delayed choice operator instead of a nondeterministic choice operator and the choice for a dynamic permission relation instead of a static permission relation are both motivated from the intuitive meaning of MSC. The use of delayed choice is mainly motivated from the use of MSC for the description of system requirements, scenarios and test cases [BM95]. In these situations the user of the language probably has a perception of alternatives that corresponds to delayed choice instead of nondeterministic choice.

The choice for a dynamic permission relation is mainly motivated by the desire to have the distributivity of weak sequential composition and delayed parallel composition over delayed choice. The user interprets an MSC with alternatives in it as a set of MSCs where each MSC describes exactly one scenario without alternatives. This motivates the choice for a dynamic permission relation as this enables the distributivity of weak sequential composition and delayed parallel composition over delayed choice.

The deadlock constant is introduced for the purpose of disabling alternatives due to the resolving of choices by the execution of a permitted event, as explained before. As a consequence it must be the case that δ itself does not permit any event. We expect that the constants $\mathbf{0}_S$ of Rensink and Wehrheim are necessary when we aim for a complete axiomatization of bisimulation equivalence. Otherwise it might be difficult to establish that the processes $a \circ^{b \rightarrow a} \varepsilon$ and $a' \circ^{c \rightarrow a'} \varepsilon$ are bisimilar for any different $a, a', b, c \in A$ such that $\ell(a) = \ell(a')$.

3.7 Generalization of the composition operators

In this section generalized versions of the delayed parallel composition operator and the weak sequential composition operator are defined. These operators must be generalized in order to describe the ordering of events based on other grounds than the fact that they are from the same location. In MSC this situation applies to messages and causal orderings.

The operators for delayed parallel and weak sequential composition are generalized by labeling them with a set of ordering requirements. An ordering requirement is a triple of the form $a \xrightarrow{n} b$ where a and b are different atomic actions and n is a natural number. As a notational shorthand $a \xrightarrow{0} b$ is written as $a \mapsto b$. Often the curly brackets of the set of ordering requirements are simply omitted.

The reason to include this “counter” n in the ordering requirements is that in MSC the two events to be ordered, say a and b , can (in certain situations) be executed a different number of times. In such a case we need to ensure that the number of executions of event b is not greater than the number of executions of event a . The counter indicates the difference between the number of executions of a and b .

$$\begin{array}{c}
\frac{x \downarrow, y \downarrow}{x \parallel^S y \downarrow} \text{(HC 1)} \quad \frac{x \xrightarrow{a} x', y \xrightarrow{a}, \text{enabled}(a, S)}{x \parallel^S y \xrightarrow{a} x' \parallel^{\text{upd}(a, S)} y} \text{(HC 2)} \\
\\
\frac{x \xrightarrow{a} x', y \xrightarrow{a} y', \text{enabled}(a, S)}{x \parallel^S y \xrightarrow{a} x' \parallel^{\text{upd}(a, S)} y \mp x \parallel^{\text{upd}(a, S)} y'} \text{(HC 3)} \\
\\
\frac{x \xrightarrow{a}, y \xrightarrow{a} y', \text{enabled}(a, S)}{x \parallel^S y \xrightarrow{a} x \parallel^{\text{upd}(a, S)} y'} \text{(HC 4)} \quad \frac{x \cdots \xrightarrow{a} x', y \cdots \xrightarrow{a} y'}{x \parallel^S y \cdots \xrightarrow{a} x' \parallel^S y'} \text{(HC 5)}
\end{array}$$

Table 3.6: Deduction rules for generalized parallel composition.

One difference between the deduction rules for \parallel^S and \circ^S and the deduction rules for \parallel and \circ is that the execution of an event c is restricted to the situations where $\text{enabled}(c, S)$ holds. The predicate $\text{enabled}(c, S)$ holds if and only if there is no ordering requirement in the set S that does not allow the execution of event a . An ordering requirement $a \xrightarrow{n} b$ does not allow the execution of an event c if and only if $c \equiv b$ and $n = 0$. Because the counter is zero event c (or b) can only be executed after event a has been executed (at least once).

As the counter n of an ordering requirement $a \xrightarrow{n} b$ denotes the difference in the number of times that a and b have been executed already, it is influenced by the execution of a and b . Execution of a means that the counter must be increased by one and execution of b means that the counter must be decreased by one. The effect of the execution of an event a on the counters in the set of ordering requirements S is denoted by $\text{upd}(a, S)$.

The deduction rules for the generalized parallel composition operator are given in Table 3.6 and the deduction rules for the generalized weak sequential composition operator are given in Table 3.7. The auxiliary predicate enabled and the auxiliary mapping upd are defined below.

Definition 3.7.1 For $a \in A$ and $S \subseteq A \times \mathbb{N} \times A$ a set of ordering requirements the mappings $\text{enabled} : A \times \mathcal{P}(A \times \mathbb{N} \times A) \rightarrow \text{IB}$ and $\text{upd} : A \times \mathcal{P}(A \times \mathbb{N} \times A) \rightarrow \mathcal{P}(A \times \mathbb{N} \times A)$ are defined as follows:

$$\begin{aligned}
\text{enabled}(a, S) &\iff \forall b, c \in A, n \in \mathbb{N} \ b \xrightarrow{n} c \in S \implies (c \not\equiv a \vee n > 0), \\
\text{upd}(a, S) &= \{b \xrightarrow{n} c \mid b \xrightarrow{n} c \in S \wedge b \not\equiv a \wedge c \not\equiv a\} \\
&\cup \{b \xrightarrow{n-1} c \mid b \xrightarrow{n} c \in S \wedge c \equiv a \wedge n > 0\} \\
&\cup \{b \xrightarrow{n+1} c \mid b \xrightarrow{n} c \in S \wedge b \equiv a\}.
\end{aligned}$$

Note that for both operators the deduction rules are similar to the deduction rules for their non-generalized counterparts. In fact, $\parallel^\emptyset = \parallel$ and $\circ^\emptyset = \circ$. This can be seen by realizing that $\text{enabled}(a, \emptyset)$ and $\text{upd}(a, \emptyset) = \emptyset$.

$$\frac{x \downarrow, y \downarrow}{x \circ^S y \downarrow} \text{(VC 1)} \quad \frac{x \cdots \overset{a}{\rightarrow} x', y \cdots \overset{a}{\rightarrow} y'}{x \circ^S y \cdots \overset{a}{\rightarrow} x' \circ^S y'} \text{(VC 5)}$$

$$\frac{x \overset{a}{\rightarrow} x', x \cdots \overset{a}{\rightarrow} x'', y \overset{a}{\rightarrow} y', \text{enabled}(a, S)}{x \circ^S y \overset{a}{\rightarrow} x' \circ^{\text{upd}(a, S)} y \mp x'' \circ^{\text{upd}(a, S)} y'} \text{(VC 3)}$$

$$\frac{x \overset{a}{\rightarrow}, x \cdots \overset{a}{\rightarrow} x', y \overset{a}{\rightarrow} y', \text{enabled}(a, S)}{x \circ^S y \overset{a}{\rightarrow} x' \circ^{\text{upd}(a, S)} y'} \text{(VC 4)}$$

$$\frac{x \overset{a}{\rightarrow} x', x \cdots \overset{a}{\rightarrow} \forall y \overset{a}{\rightarrow}, \text{enabled}(a, S)}{x \circ^S y \overset{a}{\rightarrow} x' \circ^{\text{upd}(a, S)} y} \text{(VC 2)}$$

Table 3.7: Deduction rules for generalized weak sequential composition.

Example 3.7.2 Consider the process $?m \parallel^{!m \mapsto ?m} !m$. If the ordering requirement is not considered, i.e., the process $?m \parallel !m$ is considered, the actions $!m$ and $?m$ would be executed in any order. However, the presence of the requirement $!m \mapsto ?m$ blocks the execution of $?m$ as long as $!m$ has not been executed. Thus the only possible execution for this process is

$$?m \parallel^{!m \mapsto ?m} !m \xrightarrow{!m} ?m \parallel^{!m \mapsto ?m} \varepsilon \xrightarrow{?m} \varepsilon \parallel^{!m \mapsto ?m} \varepsilon \downarrow.$$

Example 3.7.3 Consider the process $a \circ^{a \mapsto b} (b \circ b)$ with $\ell(a) \neq \ell(b)$. For this process the only possible execution is

$$a \circ^{a \mapsto b} (b \circ b) \xrightarrow{a} \varepsilon \circ^{a \mapsto b} (b \circ b) \xrightarrow{b} \varepsilon \circ^{a \mapsto b} (\varepsilon \circ b).$$

Because of the ordering requirement $a \mapsto b$ event a must be executed before event b can be executed. Because event a can be executed only once event b can be executed at most once.

Theorem 3.7.4 (Properties of \parallel^S and \circ^S) For all closed terms $s, t, u \in \mathcal{C}(\Sigma)$ and $S \subseteq A \times IN \times A$ we have the following properties:

- generalized delayed parallel composition is commutative: $x \parallel^S y \Leftrightarrow y \parallel^S x$;
- generalized delayed parallel composition distributes over delayed choice:

$$(s \mp t) \parallel^S u \Leftrightarrow s \parallel^S u \mp t \parallel^S u$$

and

$$s \parallel^S (t \mp u) \Leftrightarrow s \parallel^S t \mp s \parallel^S u;$$

- deadlock is a left-zero element for generalized weak sequential composition: $\delta \circ^S t \Leftrightarrow \delta$;
- generalized weak sequential composition distributes over delayed choice:

$$(s \mp t) \circ^S u \Leftrightarrow s \circ^S u \mp t \circ^S u$$

and

$$s \circ^S (t \mp u) \Leftrightarrow s \circ^S t \mp s \circ^S u.$$

Proof These properties are proved in Appendix B.5. □

3.8 Renaming operator

In this section we define the renaming operator ρ_f , for $f : A \rightarrow A$ a given total injective mapping on atomic actions. A mapping $f : A \rightarrow A$ is called injective if for all $a, b \in A$, if $f(a) = f(b)$ then $a = b$. In the case that $a \in \text{rng}(f)$ we denote the unique $b \in A$ with $f(b) = a$ by $f^{-1}(a)$.

If the process x can execute an atomic action a , then the process $\rho_f(x)$ can execute the atomic action $f(a)$. Similarly, if the process x permits the execution of an event a , then the process $\rho_f(x)$ permits the execution of event $f(a)$. The process $\rho_f(x)$ terminates if and only if the process x terminates. The deduction rules for the renaming operator are given in Table 3.8.

$\frac{x \downarrow}{\rho_f(x) \downarrow}$	$\frac{x \xrightarrow{a} x'}{\rho_f(x) \xrightarrow{f(a)} \rho_f(x')}$	$\frac{x \xrightarrow{a} x'}{\rho_f(x) \xrightarrow{f(a)} \rho_f(x')}$
---	--	--

Table 3.8: Deduction rules for renaming.

The reason for requiring that the mapping f is injective is that otherwise the result of renaming a deterministic process is not necessarily deterministic. This is illustrated by the following example.

Example 3.8.1 Consider the process $x \equiv a \mp b \circ c$. Consider the mapping f with $f(a) = d$, $f(b) = d$ and which is the identity otherwise. Then, by the above deduction rules $\rho_f(x) \xrightarrow{d} \rho_f(\varepsilon)$ since $x \xrightarrow{a} \varepsilon$ and $f(a) = d$ and $\rho_f(x) \xrightarrow{d} \rho_f(\varepsilon \circ c)$ since $x \xrightarrow{b} \varepsilon \circ c$ and $f(b) = d$. The process $\rho_f(x)$ is not deterministic since the processes $\rho_f(\varepsilon)$ and $\rho_f(\varepsilon \circ c)$ are not equivalent.

The example indicates that, if two initial actions a and b , that are different for process x , can be renamed into the same action d , then the renamed process $\rho_f(x)$ is not deterministic anymore. The injectivity criterion ensures that it is not possible that two actions are identical after a renaming if they are not identical before the renaming.

3.9 Repetitive behavior

3.9.1 Iteration

The process x^\circledast represents the process that consists of any number of vertical compositions of process x . This includes the possibility of executing x zero times and also the possibility of executing x infinitely often. The choice of how many times the process x is executed, however, is delayed. The deduction rules for iteration are presented in Table 3.9. The operation of the iteration operator is closely related to the operation of the weak sequential composition and the delayed choice as will be clear from the explanation of the deduction rules.

$\frac{}{x^\circledast \downarrow}$ (IT 1)	
$\frac{x \xrightarrow{a} x', x \cdots \overset{a}{\dashrightarrow}}{x^\circledast \xrightarrow{a} x' \circ x^\circledast}$ (IT 2)	$\frac{x \xrightarrow{a} x', x \cdots \overset{a}{\dashrightarrow} x''}{x^\circledast \xrightarrow{a} x''^\circledast \circ (x' \circ x^\circledast)}$ (IT 3)
$\frac{x \cdots \overset{a}{\dashrightarrow}}{x^\circledast \cdots \overset{a}{\dashrightarrow} \varepsilon}$ (IT 4)	$\frac{x \cdots \overset{a}{\dashrightarrow} x'}{x^\circledast \cdots \overset{a}{\dashrightarrow} x'^\circledast}$ (IT 5)

Table 3.9: Deduction rules for iteration.

The process x^\circledast has the option to execute x zero times and thus, it has the option to terminate successfully and immediately. This is what is expressed by deduction rule (IT 1).

The process x^\circledast can perform an event a if the process x can do so. To determine what the resulting process will be, it is of importance whether x also permits the event a . Suppose that $x \xrightarrow{a} x'$. In the case that x does not permit event a , i.e. $x \cdots \overset{a}{\dashrightarrow}$, the only possibility for executing a is the a from the first x . The resulting process then clearly is $x' \circ x^\circledast$ (see deduction rule (IT 2)).

In the case that x does permit a and thereby evolves into x'' , i.e. $x \cdots \overset{a}{\dashrightarrow} x''$, there are many possibilities for executing a . The choice between all these possible executions of

a is delayed. The resulting process is given by $x''^{\otimes} \circ x' \circ x^{\otimes}$ (see deduction rule (IT 3)). The resulting process $x''^{\otimes} \circ (x' \circ x^{\otimes})$ describes that first any number of occurrences of x permit the execution of a (resulting in x''^{\otimes}) and then a is finally executed by some occurrence of x (resulting in x').

If process x does not permit the execution of action a , then x^{\otimes} permits the execution of action a (IT 4). The reason for this is that x^{\otimes} has the empty process ε as one of its alternatives. If, on the other hand, the process x does permit the execution of a and thereby evolves into x' , then x^{\otimes} also permits the execution of a and it evolves into x'^{\otimes} (IT 5).

Example 3.9.1.1 Consider the process a^{\otimes} . This process describes an arbitrary number of executions of action a . Only the first occurrence of a can be executed as $a \cdots \xrightarrow{a}$. Thus $a^{\otimes} \xrightarrow{a} \varepsilon \circ a^{\otimes}$.

Example 3.9.1.2 Consider the process $(a \circ b)^{\otimes}$ where $\ell(a) \neq \ell(b)$. The first occurrence of b can be executed as a allows this ($a \cdots \xrightarrow{b} a$). The other occurrences of b cannot be executed as the previous occurrences of b prohibit this ($a \circ b \cdots \xrightarrow{b}$). Thus, $(a \circ b)^{\otimes} \xrightarrow{b} (a \circ \varepsilon) \circ (a \circ b)^{\otimes}$.

Example 3.9.1.3 Consider the process $(a \mp b)^{\otimes}$ where $\ell(a) \neq \ell(b)$. Then $a \mp b \cdots \xrightarrow{a} b$ and $a \mp b \xrightarrow{a} \varepsilon$. Deduction rule (IT 3) then gives $(a \mp b)^{\otimes} \xrightarrow{a} b^{\otimes} \circ (\varepsilon \circ (a \mp b)^{\otimes})$. This result can be explained as follows. First, an arbitrary number of copies of $a \mp b$ allow the execution of event a . Each of the copies of $a \mp b$ evolves into the process b ($(a \mp b)^{\otimes} \cdots \xrightarrow{a} b^{\otimes}$). Then, event a is actually executed by a copy of $a \mp b$. This copy of $a \mp b$ evolves into the process ε ($a \mp b \xrightarrow{a} \varepsilon$). The deduction rule expresses that an arbitrary occurrence of a can be executed and that as a consequence all previous occurrences of a are removed.

3.9.2 Unbounded repetition

The unbounded repetition of the process x , i.e., x^{∞} , corresponds to the notion where fresh copies of x are composed by means of weak sequential composition ad infinitum. The fact that the operation of unbounded repetition is so closely linked with the operation of weak sequential composition is visible in the deduction rules presented in Table 3.10.

First, we consider the transition relation. There are only two relevant (disjoint) cases. The first is where x can execute an a event and x does not permit an a event, and the second is where x can execute an a event and also permits an a event. The other case, i.e., where x cannot execute an a event, does not give rise to a transition of x^{∞} as none of the copies of x can execute the a event.

Suppose that x can perform an a event and thereby evolves into x' and suppose that x does not permit an a event. Then, following the deduction rules for weak sequential

$$\frac{x \xrightarrow{a} x', x \cdots \xrightarrow{a}}{x^\infty \xrightarrow{a} x' \circ x^\infty} \text{(UR 1)} \quad \frac{x \xrightarrow{a} x', x \cdots \xrightarrow{a} x''}{x^\infty \xrightarrow{a} x''^{\otimes} \circ (x' \circ x^\infty)} \text{(UR 2)}$$

$$\frac{x \cdots \xrightarrow{a} x'}{x^\infty \cdots \xrightarrow{a} x'^\infty} \text{(UR 3)}$$

Table 3.10: Deduction rules for unbounded repetition.

composition, the process x^∞ can only execute the a event from the first copy of x . Thus x^∞ performs the a event as well and thereby evolves into the process $x' \circ x^\infty$. This is expressed by deduction rule (UR 1).

Alternatively, if x permits an event a and thereby evolves into x'' , there are in principle infinitely many possibilities for the execution of the a event, due to the permission for a , each of the copies can perform the a event. Thus, the deduction rule expresses that one of the copies of x will perform the a event. All preceding copies thus evolve into x'' . Thus the process x^∞ evolves into the process $x''^{\otimes} \circ (x' \circ x^\infty)$ after the execution of action a . This is expressed by deduction rule (UR 2).

The deduction rule for the permission relation (UR 3) is based directly on the deduction rule for weak sequential composition. Note that the process x^∞ cannot terminate, not even if x can terminate.

Example 3.9.2.1 An example of the situation where an event can be executed even if it is composed vertically with an unbounded repetition is the process $a^\infty \circ b$ with $\ell(a) \neq \ell(b)$. Clearly, $a^\infty \cdots \xrightarrow{b} a^\infty$. Therefore, we obtain $a^\infty \circ b \xrightarrow{b} a^\infty \circ \varepsilon$.

Example 3.9.2.2 The process $(a \mp b)^\infty \circ b$ with $\ell(a) \neq \ell(b)$ can execute b and thereby evolves into the process $(a^{\otimes} \circ (\varepsilon \circ (a \mp b)^\infty)) \circ b \mp a^\infty \circ \varepsilon$. The first summand of the resulting process describes that one of the occurrences of b from the process $(a \mp b)^\infty$ can be executed and the second summand describes the situation that the b following the unbounded repetition is executed.

A convenient shorthand is the expression $x^{[m,n]}$ where $x \in \mathcal{C}(\Sigma)$ and $m, n \in \mathbb{N} \cup \{\infty\}$. This expression indicates that at least m and at most n copies of x are composed by means of weak sequential composition. For example, the expression $x^{[2,4]}$ represents the expression $x \circ x \mp x \circ (x \circ x) \mp x \circ (x \circ (x \circ x))$. If the minimal number of repetitions exceeds the maximal number of repetitions it is assumed that x is executed zero times.

Definition 3.9.2.3 Let $m, n \in \mathbb{N}$. Then, for $x \in \mathcal{C}(\Sigma)$, the process $x^{[m,n]}$ is defined inductively as follows:

$$x^{[m,n]} = \begin{cases} \varepsilon & \text{if } m > n, \\ x^m & \text{if } m = n, \\ x^m \mp x^{[m+1,n]} & \text{if } m < n. \end{cases}$$

Additionally, we define the following shorthands for $m \in \mathbb{N}$ and $x \in \mathcal{C}(\Sigma)$

$$\begin{aligned} x^{[m,\infty]} &= x^m \circ x^\otimes, \\ x^{[\infty,n]} &= \varepsilon, \\ x^{[\infty,\infty]} &= x^\infty. \end{aligned}$$

Theorem 3.9.2.4 (Properties of \otimes and ∞) For closed terms $t \in \mathcal{C}(\Sigma)$ we have the following properties

1. $t^\otimes \Leftrightarrow \varepsilon \mp t \circ t^\otimes$;
2. $t^\infty \Leftrightarrow t \circ t^\infty$;
3. $t^\otimes \mp t^\infty \Leftrightarrow t^\otimes$;
4. $t^\otimes \circ t^\otimes \Leftrightarrow t^\otimes$;
5. $t^\otimes \circ t^\infty \Leftrightarrow t^\infty$.

Proof These properties are proved in Appendix B.6. \(\square\)

3.10 Congruence and determinism

In this section we present two important properties of our process theory. These are congruence and determinism. Congruence refers to the fact that we can use already established equalities also in a broader context. For example, as $x \circ \varepsilon \Leftrightarrow x$, we can also obtain $(x \circ \varepsilon) \circ y \Leftrightarrow x \circ y$.

Definition 3.10.0.5 (Congruence) Let Σ be a signature. An equivalence relation R on the set of closed Σ -terms is called a congruence if for all n -ary function symbols $f \in \Sigma$ and closed Σ -terms $x_1, \dots, x_n, y_1, \dots, y_n$ we have

$$(\forall_{1 \leq i \leq n} x_i R y_i) \Rightarrow f(x_1, \dots, x_n) R f(y_1, \dots, y_n).$$

Theorem 3.10.1 (Congruence) Bisimulation is a congruence with respect to the function symbols from the signature Σ .

Proof For the proof of this theorem, we use Verhoef's congruence theorem for structured operational semantics [Ver95]. If a term deduction system is in *panth* format and stratifiable, then bisimilarity is a congruence with respect to all function symbols occurring in the signature. Strictly speaking the term deduction system is not in *panth* format. The reason for this is that we use the logical connective \vee in the set of hypotheses of the deduction rules (WS 2) and (VC 2). This \vee in the hypotheses can be seen as a shorthand notation for two deduction rules. So the deduction rule of the form

$$\frac{H \cup \{h_1 \vee h_2\}}{C}$$

is a shorthand notation for the deduction rules

$$\frac{H \cup \{h_1\}}{C} \quad \text{and} \quad \frac{H \cup \{h_2\}}{C}.$$

It remains to provide a stratification for the term deduction system. We define the mapping $S : \mathcal{C}(\Sigma) \rightarrow \mathbb{N}$ for $t, t' \in \mathcal{C}(\Sigma)$ and $a \in A$ as follows:

$$\begin{aligned} S(t\downarrow) &= 1, \\ S(t \xrightarrow{a} t') &= 1 + n(t), \\ S(t \xrightarrow{\dots^a} t') &= 1 + n(t). \end{aligned}$$

The mapping $n : \mathcal{C}(\Sigma) \rightarrow \mathbb{N}$ is for $t, t' \in \mathcal{C}(\Sigma)$, $a \in A$, f an injective mapping and $S \subseteq A \times \mathbb{N} \times A$ defined as follows:

$$\begin{aligned} n(\varepsilon) &= n(\delta) = n(a) = 1, \\ n(\rho_f(t)) &= n(t^\circledast) = n(t^\infty) = n(t) + 1, \\ n(t \mp t') &= n(t \parallel t') = n(t \circ t') = n(t \parallel^S t') = n(t \circ^S t') = n(t) + n(t') + 1. \end{aligned}$$

It can easily be checked that this mapping S is a stratification for the transformed term deduction system. \square

Definition 3.10.2 (Determinism) A process $t \in \mathcal{C}(\Sigma)$ is called *deterministic* if and only if for all $a \in A$ and $t_1, t_2 \in \mathcal{C}(\Sigma)$

- if $t \xrightarrow{a} t_1$ and $t \xrightarrow{a} t_2$, then $t_1 \equiv t_2$, and
- if $t \xrightarrow{\dots^a} t_1$ and $t \xrightarrow{\dots^a} t_2$, then $t_1 \equiv t_2$.

Theorem 3.10.3 (Determinism) Every process $t \in \mathcal{C}(\Sigma)$ is deterministic.

Proof This theorem can be proved by induction on the structure of closed term t . Inspection of the deduction rules indicates that every operator from the signature Σ is determinism preserving, that is, if s_1, \dots, s_n are deterministic processes and f is an

n -ary operator, then $f(s_1, \dots, s_n)$ is also deterministic. This can be seen as follows. Suppose that $f(s_1, \dots, s_n)$ is not deterministic. Then, by definition, it must be the case that $f(s_1, \dots, s_n) \xrightarrow{a} p$ and $f(s_1, \dots, s_n) \xrightarrow{a} p'$ for some $a \in A$ and $p, p' \in \mathcal{C}(\Sigma)$ such that $p \neq p'$. This can only be the case if two different deduction rules can be applied to the term $f(s_1, \dots, s_n)$ at the same time. The only deduction rules that can be applied to the term $f(s_1, \dots, s_n)$ are the deduction rules for which the left-hand side of the conclusion is of the form $f(x_1, \dots, x_n)$ and for which the sets of hypotheses are satisfied. From the deduction rules we can immediately see that it cannot be the case that two deduction rules are applicable since the sets of hypotheses cannot be satisfied at the same time. Hence, the process $f(s_1, \dots, s_n)$ is deterministic. With respect to the permission relation a similar reasoning can be applied. \square

4

Semantics of Message Sequence Charts

4.1 Introduction

In this chapter, a denotational semantics of the language MSC is defined. It consists of a family of mappings $\llbracket \cdot \rrbracket$ which transform (part of) an MSC in textual representation into a process expression over the signature introduced in Chapter 3. The semantics is defined compositionally. This means that the semantics of a piece of textual syntax is only defined in terms of the information available in this piece of syntax.

Before we start with the formal definition of the semantics in Section 4.3 and further, we first explain our approach to the semantics in the following section.

4.2 The approach

For the purpose of defining the formal semantics we use the textual syntax as presented in Appendix A.2. This syntax is different from the textual syntax as presented in recommendation Z.120. The changes can be subdivided into several categories:

- Not treated. Not treated in this chapter are the instance-oriented textual syntax, instance decomposition, substitution, and the combination of incomplete message events with gates.
- Removing irrelevant information. For a complete list of the parts of the textual syntax of Z.120 that are considered irrelevant see Appendix A.1.2. An example of irrelevant information are the instance head and instance end statements.

- Shorthands. Several parts of the textual syntax of MSC can be seen as a shorthand for another (larger) piece of textual syntax. For the formal semantics definition it is convenient to assume that these are replaced by their unabbreviated versions. An example is the omission of a loop boundary in an inline expression with the keyword **loop**. This is an abbreviation of a similar construction with a loop boundary $\langle 1, \text{inf} \rangle$. A list of such shorthands that are removed from the textual syntax is given in Appendix A.1.3.
- Extensions. Introduction of the keyword **after** for the description of causal orderings. For the consequences of this extension see Appendix A.1.4.

Furthermore, some assumptions that are imposed on the textual syntax are listed in Appendix A.1.5.

4.2.1 MSC documents

An MSC document contains a finite number of MSCs. In an MSC, references to other MSCs can be used by means of the unique MSC names. A reference to an MSC with name A can be dealt with semantically by substituting the MSC name with the body of the MSC with that name. However, the approach that is followed in this chapter is such that for every MSC in the MSC document an equation is given that associates with an MSC with name A the equation $\bar{A} = S$ where \bar{A} is a variable associated with the MSC with name A and S is the semantics of the body of this MSC. As a consequence the semantics of an MSC document thus consists of a set of equations.

It is not allowed that an MSC refers to itself, directly or via a number of other references. Therefore, the equations are not recursion equations; they only define a complex process expression as a constant.

4.2.2 Message Sequence Charts

Both the semantics of an MSC with name A in the context of such an MSC document and the semantics of a reference to such an MSC are given by the constant \bar{A} . This approach allows to consider the semantics of an MSC document by considering the semantics of every MSC in isolation.

4.2.3 Message Sequence Chart bodies

The body of an MSC in event-oriented textual representation basically consists of a list of event definitions. The intuition behind such a list of event definitions is that these can be thought of as being composed vertically in the same order as the event definitions appear in the event-oriented representation. The approach that is followed to obtain the semantics of an MSC body can then be paraphrased in the

following way: an MSC body is the vertical composition of the event definitions that are contained.

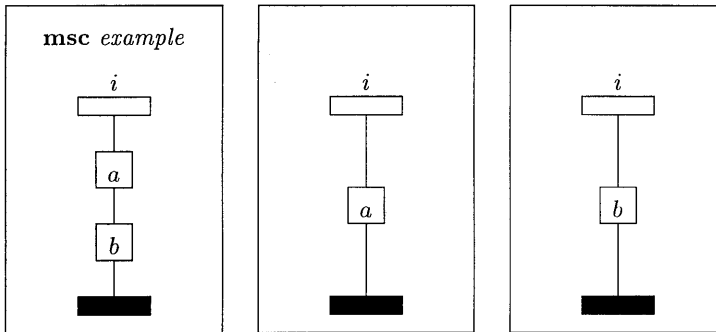


Figure 4.1: Decomposition of an instance.

Example 4.2.3.1 Consider the MSC given in Figure 4.1. It contains one instance with two local actions. Textually, this MSC is represented by

```
msc example ;
i : action 'a' ;
i : action 'b' ;
endmsc ;
```

It contains the two event definitions “ $i : \mathbf{action} 'a'$ ” and “ $i : \mathbf{action} 'b'$ ”. Each of these can be viewed as a nameless MSC as depicted in Figure 4.1 as well. The original MSC can be obtained from the MSC fragments by means of vertical composition. The semantics of the MSC is obtained by composing the semantics of the MSC fragments by means of weak sequential composition. Due to the fact that both local actions are defined on the same instance, the weak sequential composition operator maintains the ordering that local action a is executed before local action b .

Example 4.2.3.2 Consider the MSC given in Figure 4.2. It contains two instances, each of which has a local action. Textually, this MSC is represented by

```
msc example ;
i : action 'a' ;
j : action 'b' ;
endmsc ;
```

or by

```
msc example ;
j : action 'b' ;
i : action 'a' ;
endmsc ;
```

In both cases, the textual representation contains the event definitions “ $i : \mathbf{action} 'a'$ ” and “ $j : \mathbf{action} 'b'$ ”. A graphical representation of those is also given in Figure 4.2. As these events are defined on different instances their vertical composition has the same result as their horizontal composition. The semantics of the MSC is again obtained by composing the semantics of the MSC fragments by means of weak sequential

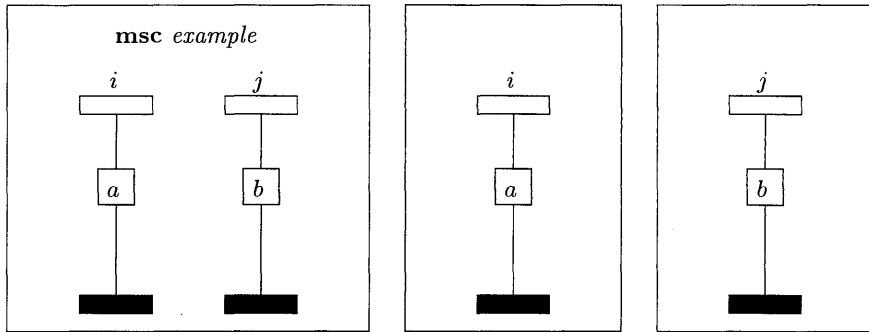


Figure 4.2: Decomposition of two unrelated events.

composition. As the local actions are defined on different instances and this is reflected in the atomic actions representing them, the weak sequential composition of these atomic actions results in their interleaved execution.

In the first example the MSC fragments from which the MSC is composed are related through the fact that they have an instance in common. In the second example the two MSC fragments are not related at all. The vertical composition of two MSC fragments results in the linking of instances with the same name. For the first example this means that the order between the local actions a and b is maintained. However, there are also situations in which there are more relations between the MSC fragments. Examples of these are messages and causal orderings where the two events that are involved reside in different MSC fragments.

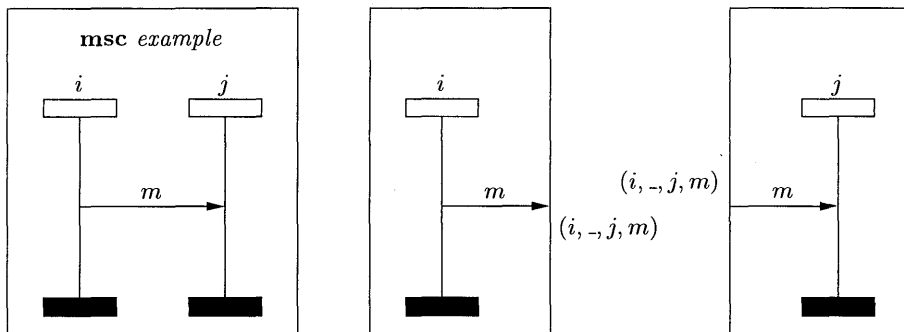


Figure 4.3: Decomposition of a message.

Example 4.2.3.3 Consider the MSC given in Figure 4.3. It contains two instances and one message. The message consists of two event definitions in the textual representation. These are “ i : out m to j ” and “ j : in m from i ”. In the textual representation of the MSC these can occur in any order in the MSC body. Semantically however, it is expected that the output of m takes place before the input of m . This is achieved semantically by computing an ordering requirement and labeling

the generalized weak sequential composition operator by the computed requirement. The requirement is computed from the semantics of the MSC fragments in isolation. In Figure 4.3 the connections of the dangling message events with the frame of the MSC fragments are labeled with the information that is used to find that these events together constitute a message. In Section 4.6 it is explained in more detail how the ordering requirement is computed from the semantics of the MSC fragments.

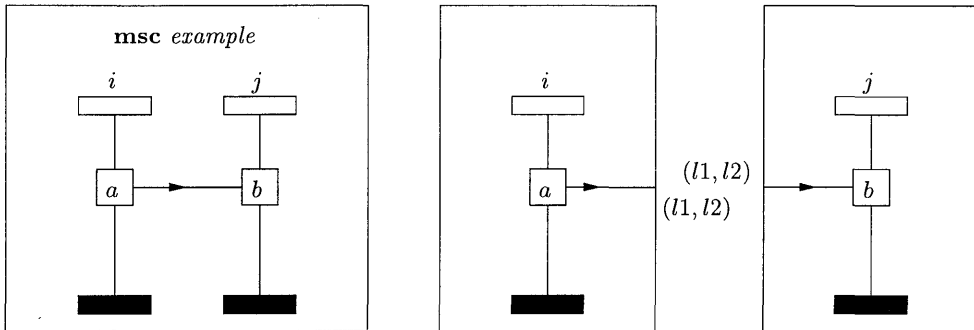


Figure 4.4: Decomposition of a causal ordering.

Example 4.2.3.4 Consider the MSC given in Figure 4.4. It contains two instances. Each instance contains a local action. The local actions are ordered causally. The event definitions that occur in the textual representation are “*i* : *l1* action ‘*a*’ before *l2*” and “*j* : *l2* action ‘*b*’ after *l1*”. The MSC fragments are represented graphically in Figure 4.4. The connections of the causal order arrows with the frames of the MSC fragments are labeled by the information that is used to find that these two events are ordered causally. Semantically, this information is attached to the atomic actions representing the local actions (see Section 4.5). The vertical composition of the MSC fragments that correspond to these event definitions should incorporate the requirement that local action *a* must precede local action *b*. Also this type of requirements is computed from the semantics of the MSC fragments.

In the previous examples the MSC body only contains two event definitions. In general, however, an MSC body contains an arbitrary (but finite) number of event definitions. The approach followed for arbitrary MSC bodies is basically the same as the approach sketched in the previous examples. Semantics is provided for an MSC body by vertically composing the semantics of the first event definition with the semantics of the remaining part of the MSC body. The linking of instances with the same name is taken care of by the generalized weak sequential composition operator. The ordering requirements that are due to corresponding message events and corresponding causally ordered events in different MSC fragments are explicitly added as a label to the generalized weak sequential composition operator.

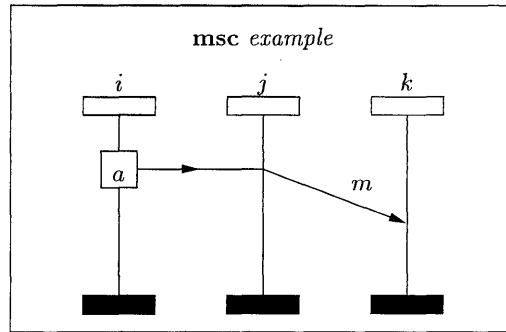


Figure 4.5: An example MSC.

Example 4.2.3.5 In Figure 4.5 an MSC is given and in Figure 4.6 its decomposition into three fragments is given by means of horizontal dashed lines. The textual syntax of each of the MSC fragments is given in the same figure. The MSC is decomposed

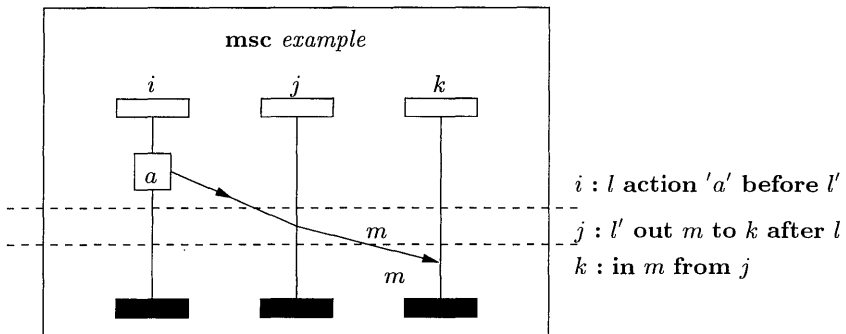


Figure 4.6: Attributed example MSC.

into three fragments. Each fragment describes one event. The textual representations of the fragments contain enough information to establish how the dangling message arrows and causal order arrows are to be connected. For example the fact that local action a precedes the output of message m is available in the event names l and l' and the parts of the textual syntax that describe " $l \dots$ **before** l' " and " $l' \dots$ **after** l ". In isolation the three fragments could be represented as given in Figure 4.7. In this figure, dangling arrows are connected with the frame around the MSC fragment and the information that is necessary for determining whether the dangling arrows should be connected is described close to the connection with the frame.

In the approach towards the definition of the formal semantics one event definition is almost an MSC on its own. It differs from an MSC in the following aspects:

1. It does not have a name.
2. It can have dangling message arrows and dangling causal order arrows.

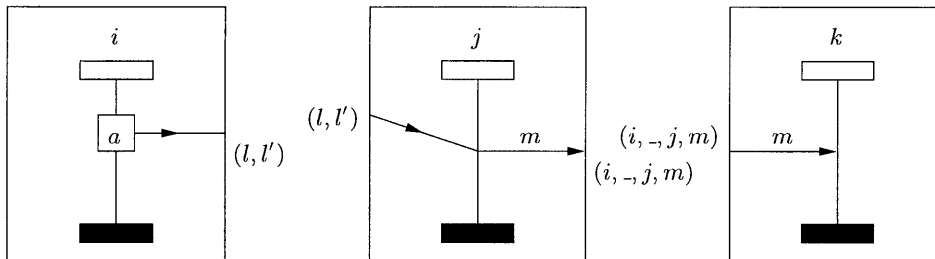


Figure 4.7: Decomposition of the example MSC.

In the presented examples only very simple MSC fragments appeared. In principle, also more complex MSC fragments are used in the semantics definition. The following MSC fragments are distinguished:

1. Single instance events: An event attached to an instance with some dangling causal arrows.
2. Vertical composition: The vertical composition of two MSC fragments is again considered an MSC fragment. In such a vertical composition corresponding dangling arrows are connected and the required orderings are maintained.
3. Coregions.
4. Multi instance events: A multi instance event is attached to a number of instances (at least one). There are three different multi instance events:
 - (a) conditions;
 - (b) MSC reference expressions;
 - (c) inline expressions.

4.2.4 Events

The single instance events are in the semantics denoted by atomic actions. The semantics of single instance events is considered in Section 4.4. These atomic actions can be labeled by an event name and a set denoting the dangling causal ordering arrows (see Section 4.5). This is necessary as this information is needed when single instance events are composed vertically or horizontally.

4.2.5 Coregions

A coregion contains a number of single instance events. These events are all defined on the same instance and they are supposed to be executed in parallel. Nevertheless,

it can be the case that the events in a coregion are ordered. Reasons for such an ordering between events can be that they are corresponding message events or that they are causally ordered.

The semantics of a coregion is obtained by considering the events in the coregion as MSC fragments that are composed horizontally. The ordering requirements are obtained in the same way as they are obtained for vertically composed MSC fragments.

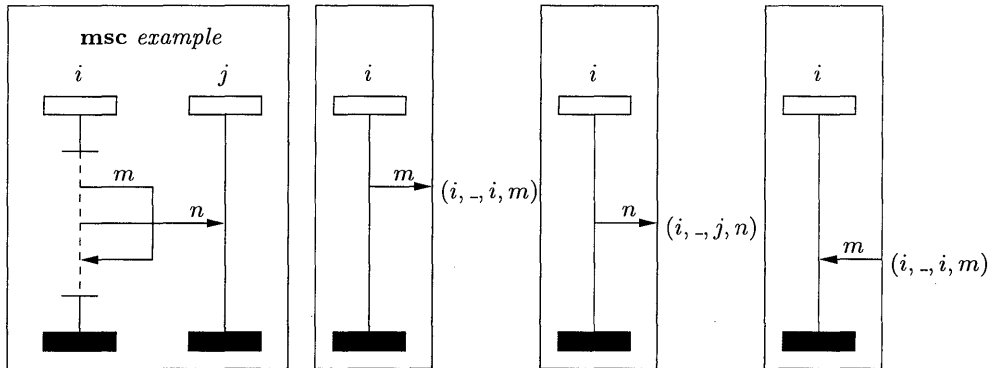


Figure 4.8: Decomposition of a coregion.

Example 4.2.5.1 Consider the MSC given in Figure 4.8. The coregion contains three events. To obtain the semantics of the coregion we decompose it into MSC fragments. These are given in Figure 4.8 as well. The semantics of the coregion is obtained from the semantics of the MSC fragments by means of horizontal composition. From the semantics of the MSC fragments the necessary ordering requirements are computed in a similar way as for the MSC bodies before.

The horizontal composition mechanism used for obtaining a coregion from its events is defined in Section 4.6. The semantics of a coregion is formally described in Section 4.7.

4.2.6 MSC reference expressions

An MSC reference expression is a textual formula which describes a composition of MSCs by means of a number of operators. The smallest building blocks of MSC reference expressions are references to other MSCs by means of their MSC name. Semantically, these are dealt with by means of variables. This also means that an equation must be given for such a variable. This is the reason for associating a specification with an MSC document. The operators **alt**, **seq**, **par** and **loop** $\langle m, n \rangle$ are treated semantically by replacing them by constructs from the process theory developed in the previous chapter. This way a process expression is obtained for the textual formula described in the MSC reference symbol.

However, it is possible that messages and causal order arrows are connected to the MSC reference expression symbol by means of a gate. In such cases at least one of the referenced MSCs must contain a corresponding event. It is necessary to take the ordering requirement due to this connection via the gate into account. This is achieved by replacing the message gate and order gate definitions by actual message and order gates as described textually in the MSC reference gate interface. The approach is illustrated by the following example.

Example 4.2.6.1 Consider the MSCs given in Figure 4.9. MSC *B* has an MSC reference expression which refers to MSC *A*. Instance *i* in MSC *A* sends a message *m* to a gate *g*. In MSC *B* instance *j* receives a message *m* from a gate *g* on the MSC reference symbol. This gives rise to an ordering requirement. MSC *B* can

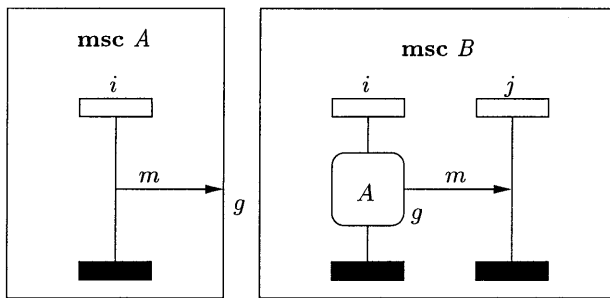


Figure 4.9: MSC with an MSC reference expression with a gate.

be thought of as being composed of two MSC fragments. These are depicted in Figure 4.10. The semantics of the MSC fragment containing the MSC reference

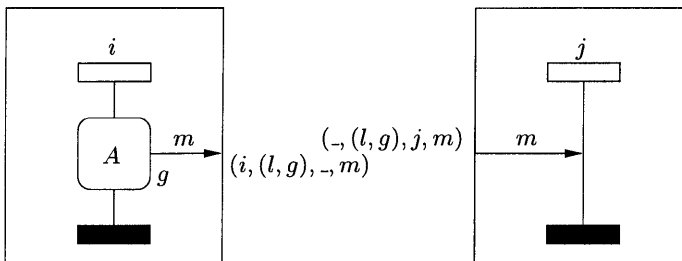


Figure 4.10: MSC fragments for MSC *B*.

expression is obtained by replacing the destination of the message *m*, that is gate *g* on the environment, in MSC *A* by an actual gate. This actual gate is a combination of the MSC reference identification and the name of the gate in MSC *A*. The reason for using the MSC reference identification is that there can be many references to MSC *A*. For each of these the gates can be connected in a different way. This implies that different instantiations of the same gate definition have to be distinguished. After this instantiation of the gate definitions by actual gates, the recipe described for obtaining the semantics of an MSC body can be followed.

In the previous example and also in the discussion on the semantics of MSC bodies we are looking for corresponding events that are located in the different MSC fragments. Now, with MSC reference expressions, the situation arises that two events from the same MSC fragment are corresponding since two gates of the MSC reference expression are connected by means of a message arrow or causal order arrow. An example is given in MSC *B* in Figure 4.11. Of course we can adapt the computation of the ordering requirements such that also corresponding events in the same MSC fragment give rise to an ordering requirement, but we rather determine those separately. The reason not to proceed in this direction is that we have no criterion to distinguish between already enforced ordering requirements and new ordering requirements. Thus, all ordering requirements that are already enforced would be taken into account again. Technical details can be found in Section 4.9.

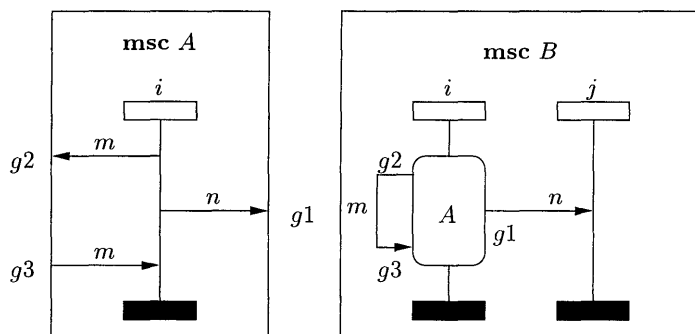


Figure 4.11: An MSC reference expression with self-connected gates.

Example 4.2.6.2 Consider the MSCs given in Figure 4.11. In this example the gates g_2 and g_3 , which are both on the MSC reference symbol, are connected. Textually, this connection is described in the MSC reference gate interface. MSC *B* is decomposed into two MSC fragments, which are given in Figure 4.12. The semantics of the first

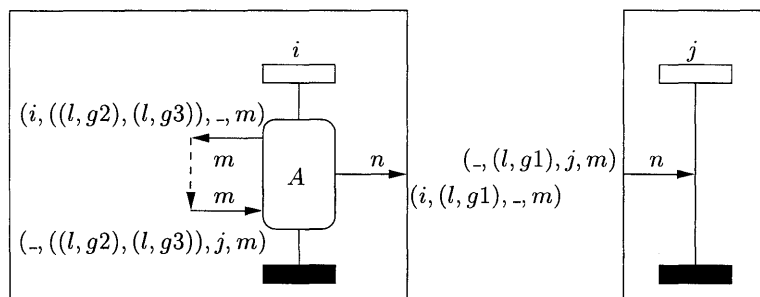


Figure 4.12: MSC fragments for MSC *B*.

MSC fragment is obtained as follows. First the semantics of the textual formula in the MSC reference symbol is computed. Then, based on the MSC reference gate interface, gate definitions are replaced by actual gates. Finally, the ordering requirements that

are due to the connection of two gates on the MSC reference symbol have been taken into account. In Figure 4.12 this is indicated by means of the dashed arrow connecting the message arrows from the gates $g2$ and $g3$.

The approach for the semantics of MSC reference expressions can be summarized as follows. First, the semantics of the textual formula in the MSC reference symbol is computed. Then, based on the MSC reference gate interface, gate definitions are replaced by actual gates. Finally, orderings of events from the MSC reference expression due to the connection of gates on the MSC reference symbol are computed and integrated.

4.2.7 Inline expressions

Inline expressions are a different graphical representation for MSC reference expressions. There are only two interesting differences. First, the operands of the operators that are used in inline expressions are not references to MSCs but MSC bodies. Secondly, the inline expression gate interface is not described textually on one place. Its description is distributed over the operands. The first difference is overcome easily. Instead of considering the semantics of a textual formula we compute the semantics of the MSC bodies, as described before. The second difference is only an artificial one. We explained before that we give semantics to gates under several restrictions. These restrictions guarantee that we in fact can combine the distributed inline expression gate interfaces into one overall inline gate interface. With those remarks it can be understood that the approach towards the semantics of inline expressions is similar to the approach for MSC reference expressions.

4.2.8 High-level Message Sequence Charts

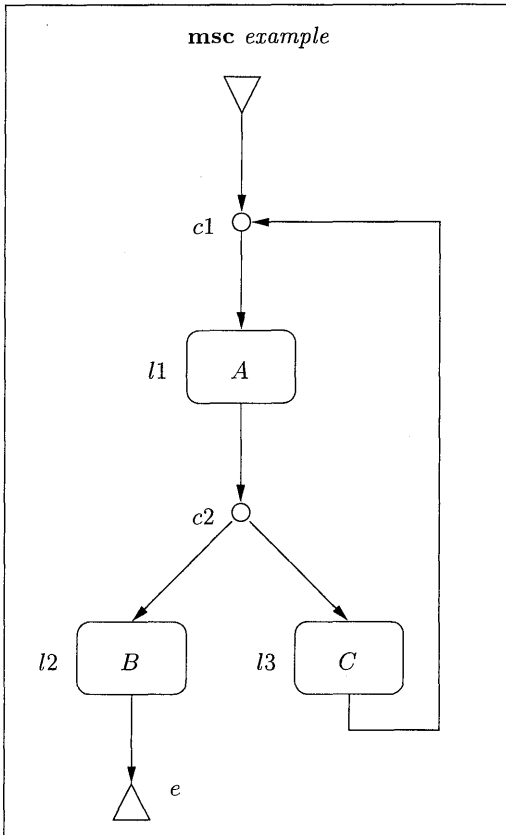
On an abstract level an HMSC consists of a number of nodes and arrows between those. There are different types of nodes. Each type of node is treated differently in the semantics. The arrows between the nodes are used to describe vertical composition and alternative composition. If a node has only one outgoing arrow, then this indicates vertical composition. If a node has multiple outgoing arrows, this indicates alternative vertical compositions.

For an HMSC the semantics is obtained by transforming it into an HMSC with a very specific structure. This transformation is not defined directly on HMSCs but on edge-labelled graphs. For the “HMSC” that results after the transformation a process expression can be given easily. We will illustrate this by an example. For more details on the transformation itself we refer to Section 4.11.

Example 4.2.8.1 Consider the HMSC given in Figure 4.13. The textual representation of this MSC is also given in this figure. The label names used in the textual

representation for describing the nodes are indicated in the HMSC as well. With this HMSC eventually the following process expression is associated:

$$(\bar{A} \circ \bar{C})^{\oplus} \circ (\bar{A} \circ \bar{B}).$$



```

msc example ;
expr c1 ;
c1 : connect seq (l1) ;
l1 : A seq (c2) ;
c2 : connect seq (l2 alt l3) ;
l2 : B seq (e) ;
l3 : C seq (c1) ;
e : end ;
endmsc ;

```

Figure 4.13: An annotated HMSC.

4.3 Semantics of an MSC document

The semantics associated with an MSC document is a set of equations defining constants. For every MSC in the MSC document a constant is introduced. For an MSC with name id , this constant is denoted as \bar{id} .

The mapping MSC associates with an MSC document a set of pairs of MSC names with their textual representation, as they appear in that MSC document. Note that for each MSC name there can be at most one pair in which that MSC name occurs.

Definition 4.3.1 The mapping $MSC : \mathcal{L}(\langle \text{msc document} \rangle) \rightarrow \mathcal{P}(\mathcal{L}(\langle \text{msc name} \rangle) \times \mathcal{L}(\langle \text{message sequence chart} \rangle))$ is for $docid \in \mathcal{L}(\langle \text{msc document name} \rangle)$ and $docbody \in \mathcal{L}(\langle \text{msc document body} \rangle)$ defined as follows:

$$MSC(\mathbf{mscdocument} \textit{ docid} ; \textit{ docbody}) = MSC(\textit{ docbody}).$$

The mapping $MSC : \mathcal{L}(\langle \text{msc document body} \rangle) \rightarrow \mathcal{P}(\mathcal{L}(\langle \text{msc name} \rangle) \times \mathcal{L}(\langle \text{message sequence chart} \rangle))$ is for $msc \in \mathcal{L}(\langle \text{message sequence chart} \rangle)$ and $docbody \in \mathcal{L}(\langle \text{msc document body} \rangle)$ defined inductively as follows:

$$\begin{aligned} MSC() &= \emptyset, \\ MSC(\textit{ msc docbody}) &= \{(Name(\textit{ msc}), \textit{ msc})\} \cup MSC(\textit{ docbody}), \end{aligned}$$

where the mapping $Name : \mathcal{L}(\langle \text{message sequence chart} \rangle) \rightarrow \mathcal{L}(\langle \text{msc name} \rangle)$ is for $id \in \mathcal{L}(\langle \text{msc name} \rangle)$, $mscbody \in \mathcal{L}(\langle \text{msc body} \rangle)$ and $mscexpr \in \mathcal{L}(\langle \text{msc expression} \rangle)$ defined as follows:

$$\begin{aligned} Name(\mathbf{msc} \textit{ id} ; \textit{ mscbody} \mathbf{endmsc} ;) &= \textit{ id}, \\ Name(\mathbf{msc} \textit{ id} ; \mathbf{expr} \textit{ mscexpr} \mathbf{endmsc} ;) &= \textit{ id}. \end{aligned}$$

As an MSC document cannot contain two or more MSCs with the same MSC name this set of pairs can be considered a mapping. In the sequel we will write $MSC(id)$ if we mean msc such that $(id, msc) \in MSC(doc)$. Note that we must be certain that we only do this for id such that there actually is an MSC with that name in the MSC document.

The mapping Eqs associates to an MSC document the set of recursive equations that describe the semantics of the MSCs in the MSC document. For an MSC (not an HMSC) this equation is of the form $\overline{id} = S$ where id is the name of the MSC and S is the semantics of the body of the MSC. The definition of the mapping Eqs for HMSCs is given in Definition 4.11.9.

Definition 4.3.2 For all $docid \in \mathcal{L}(\langle \text{msc document name} \rangle)$ and $docbody \in \mathcal{L}(\langle \text{msc document body} \rangle)$

$$Eqs(\mathbf{mscdocument} \textit{ docid} ; \textit{ docbody}) = Eqs(\textit{ docbody}).$$

For $msc \in \mathcal{L}(\langle \text{message sequence chart} \rangle)$ and $docbody \in \mathcal{L}(\langle \text{msc document body} \rangle)$

$$\begin{aligned} Eqs() &= \emptyset, \\ Eqs(\textit{ msc docbody}) &= Eqs(\textit{ msc}) \cup Eqs(\textit{ docbody}). \end{aligned}$$

For $id \in \mathcal{L}(\langle \text{msc name} \rangle)$ and $mscbody \in \mathcal{L}(\langle \text{msc body} \rangle)$

$$Eqs(\mathbf{msc} \textit{ id} ; \textit{ mscbody} \mathbf{endmsc} ;) = \{\overline{id} = \llbracket \textit{ mscbody} \rrbracket\}.$$

The semantics of an MSC msc with MSC name id from a given MSC document doc is then given by the constant \overline{id} in the set of equations $Eqs(doc)$. We will denote such a constant \overline{id} with respect to a set of equations E by $\langle \overline{id} | E \rangle$.

Definition 4.3.3 Let $doc \in \mathcal{L}(\langle \text{msc document} \rangle)$. For $msc \in \mathcal{L}(\langle \text{message sequence chart} \rangle)$ such that $(Name(msc), msc) \in MSC(doc)$

$$\llbracket msc \rrbracket_{doc} = \langle \overline{Name(msc)} \mid Eqs(doc) \rangle.$$

The way in which the semantics of MSC documents and MSCs is treated in this section makes it possible to deal with references to an MSC by using the appropriate constant for the semantics. For example an MSC reference expression to an MSC with name A is semantically represented by \bar{A} .

4.4 Semantics of events

In this section the semantics of events is defined. In the recommendation several types of events are distinguished. The first distinction is between single instance events and multi instance events. A single instance event is an event that is defined on exactly one instance. A multi instance event is an event that can be defined on one or more instances. Besides this distinction there is also a distinction between orderable and non-orderable events. An orderable event is an event that can be used in a causal ordering and a non-orderable event is an event that may not be used in a causal ordering. In Table 4.1 the events that are present in the language MSC are classified with respect to orderability and the number of instances they can be defined on.

event	single instance	multi instance
non-orderable	instance stop	condition
orderable	local action (incomplete) message event instance create timer events	

Table 4.1: Classes of events.

4.4.1 Local actions

Local actions are represented in the semantics by atomic actions from the set A_{act} defined below. A local action that is defined on an instance i with action name a is denoted by $action(i, a)$.

Definition 4.4.1.1 The set A_{act} is defined as follows:

$$A_{act} = \{ action(i, a) \mid i \in \mathcal{L}(\langle \text{instance name} \rangle) \wedge a \in \mathcal{L}(\langle \text{action character string} \rangle) \}.$$

Definition 4.4.1.2 Let $i \in \mathcal{L}(\langle \text{instance name} \rangle)$. Then, for all $a \in \mathcal{L}(\langle \text{action character string} \rangle)$

$$\llbracket \mathbf{action} \ 'a' \rrbracket_i = \mathit{action}(i, a).$$

4.4.2 Message events

The atomic actions that represent message output and message input events have four parameters. For a message output event the following information is maintained:

1. the name of the instance on which the event is executed;
2. an abstract representation of the gate via which the message is sent (if available);
3. the name of the instance that should receive the message (if available);
4. the name of the message.

Similarly, for a message input the following information is maintained:

1. the name of the instance on which the event is executed;
2. an abstract representation of the gate via which the message is received (if available);
3. the name of the instance that should send the message (if available);
4. the name of the message.

Message output and input events, as they occur in the textual syntax, have either a gate part or a sender or receiver instance name, but not both. However, in connecting gates on MSC reference expressions and inline expressions an instance name becomes available as the sender or receiver of a message. In such cases the gate part is still relevant for distinguishing multiple occurrences of such a message. An example of this is given in Figure 4.14. If the gate part is not maintained in the atomic actions representing the message output events then these cannot be distinguished anymore. As a consequence it is impossible to distinguish MSC B from MSC C . As this example indicates, it is not sufficient to maintain the name of the gate via which the message is sent. For this purpose the reference identification is added to the gate name. The reference identification must therefore be unique within the MSC document.

Textually, for the input address of a message output event there are three possibilities. If it is an instance name then the message is not sent via a gate and the receiver instance name is known. This is indicated in the gate part by $_$. If the input address of a message output event is a gate g in the environment this is indicated by means of $\mathit{env}(g)$. If the input address of a message output event is an actual gate g of an

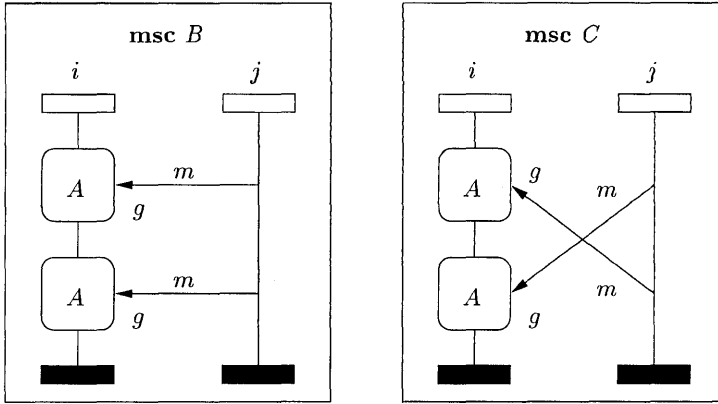


Figure 4.14: Necessity of gate names.

MSC reference expression or inline expression with reference identification l , then this is indicated by (l, g) . Actual gates are represented by elements of the set \mathcal{AG} which is defined below. This set also contains elements of the form $((l, g), (l', g'))$ where l and l' are reference identifications and g and g' are gate names. These are added explicitly for the purpose of finding corresponding message output and message input events (see Section 4.6).

Clearly, for the output address of message input events similar possibilities exist. The different notations for the representation of the gate parameter of the message output and input events are combined in the set \mathcal{AMG} which is also defined below.

Definition 4.4.2.1 The set \mathcal{AG} is defined as follows:

$$\begin{aligned} \mathcal{AG} = & \mathcal{L}(\langle \text{reference identification} \rangle) \times \mathcal{L}(\langle \text{gate name} \rangle) \\ & \cup (\mathcal{L}(\langle \text{reference identification} \rangle) \times \mathcal{L}(\langle \text{gate name} \rangle))^2. \end{aligned}$$

The set \mathcal{AMG} is defined as follows:

$$\mathcal{AMG} = \{-, env(g) \mid g \in \mathcal{L}(\langle \text{gate name} \rangle)\} \cup \mathcal{AG}.$$

Definition 4.4.2.2 The sets A_{out} and A_{in} are defined as follows:

$$\begin{aligned} A_{out} = & \{out(i, -, j, m), out(i, env(g), -, m), out(i, G, -, m), out(i, G, j, m) \\ & \mid i, j \in \mathcal{L}(\langle \text{instance name} \rangle) \wedge g \in \mathcal{L}(\langle \text{gate name} \rangle) \wedge G \in \mathcal{AG} \\ & \wedge m \in \mathcal{L}(\langle \text{message name} \rangle)\}, \\ A_{in} = & \{in(i, -, j, m), in(-, env(g), j, m), in(-, G, j, m), in(i, G, j, m) \\ & \mid i, j \in \mathcal{L}(\langle \text{instance name} \rangle) \wedge g \in \mathcal{L}(\langle \text{gate name} \rangle) \wedge G \in \mathcal{AG} \\ & \wedge m \in \mathcal{L}(\langle \text{message name} \rangle)\}. \end{aligned}$$

The atomic actions $out(i, -, j, m)$ and $in(i, -, j, m)$ represent a message output and message input event respectively for a message m that is sent directly from instance i

to instance j . The atomic actions $out(i, env(g), -, m)$ and $in(-, env(g), i, m)$ represent a message m that is sent to or received from the environment via a gate with name g . The atomic actions $out(i, (l, g), -, m)$ and $in(-, (l, g), i, m)$ represent the sending and receiving of a message m to an unknown instance via an actual gate with name g on an MSC reference expression or inline expression with reference identification l . The reasons for including the other atomic actions will become clear in Section 4.9.

Definition 4.4.2.3 Let $i \in \mathcal{L}(\langle \text{instance name} \rangle)$. Then, for $m \in \mathcal{L}(\langle \text{message name} \rangle)$, $j \in \mathcal{L}(\langle \text{instance name} \rangle)$, $g \in \mathcal{L}(\langle \text{gate name} \rangle)$ and $l \in \mathcal{L}(\langle \text{reference identification} \rangle)$

$$\begin{array}{ll}
\llbracket \text{out } m \text{ to } j \rrbracket_i & = out(i, -, j, m), \\
\llbracket \text{out } m \text{ to env via } g \rrbracket_i & = out(i, env(g), -, m), \\
\llbracket \text{out } m \text{ to reference } l \text{ via } g \rrbracket_i & = out(i, (l, g), -, m), \\
\llbracket \text{out } m \text{ to inline } l \text{ via } g \rrbracket_i & = out(i, (l, g), -, m), \\
\llbracket \text{in } m \text{ from } j \rrbracket_i & = in(j, -, i, m), \\
\llbracket \text{in } m \text{ from env via } g \rrbracket_i & = in(-, env(g), i, m), \\
\llbracket \text{in } m \text{ from reference } l \text{ via } g \rrbracket_i & = in(-, (l, g), i, m), \\
\llbracket \text{in } m \text{ from inline } l \text{ via } g \rrbracket_i & = in(-, (l, g), i, m).
\end{array}$$

Please note that there is no difference in notation in the semantics of a gate g on an MSC reference expression or on an inline expression. Throughout the semantics the role of MSC reference expressions and inline expressions will be similar.

4.4.3 Incomplete message events

Lost message output events and found message input events are represented by atomic actions from the sets A_{lost} and A_{found} respectively.

Definition 4.4.3.1 The sets A_{lost} and A_{found} are defined as follows:

$$\begin{array}{ll}
A_{lost} & = \{lost(i, j, m), lost(i, -, m), lost(i, env, m) \\
& \quad | i, j \in \mathcal{L}(\langle \text{instance name} \rangle) \wedge m \in \mathcal{L}(\langle \text{message name} \rangle)\}, \\
A_{found} & = \{found(i, j, m), found(-, j, m), found(env, j, m) \\
& \quad | i, j \in \mathcal{L}(\langle \text{instance name} \rangle) \wedge m \in \mathcal{L}(\langle \text{message name} \rangle)\}.
\end{array}$$

The first parameter of these atomic actions refers to the sender of the message, the second parameter refers to the receiver of the message and the third parameter represents the message name. For a lost message output event it is possible that the receiver is an instance, the environment or unknown. If the receiver is unknown this is indicated by $-$. Similarly, for a found message input the sender can be an instance, the environment or unknown.

Definition 4.4.3.2 Let $i \in \mathcal{L}(\langle \text{instance name} \rangle)$. Then, for $m \in \mathcal{L}(\langle \text{message name} \rangle)$ and $j \in \mathcal{L}(\langle \text{instance name} \rangle)$

$$\begin{aligned}
\llbracket \text{out } m \text{ to lost} \rrbracket_i &= \text{lost}(i, -, m), \\
\llbracket \text{out } m \text{ to lost } j \rrbracket_i &= \text{lost}(i, j, m), \\
\llbracket \text{out } m \text{ to lost env} \rrbracket_i &= \text{lost}(i, \text{env}, m), \\
\llbracket \text{in } m \text{ from found} \rrbracket_i &= \text{found}(-, i, m), \\
\llbracket \text{in } m \text{ from found } j \rrbracket_i &= \text{found}(j, i, m), \\
\llbracket \text{in } m \text{ from found env} \rrbracket_i &= \text{found}(\text{env}, i, m).
\end{aligned}$$

Recommendation Z.120 allows gates as the input address of lost messages and as the output address of found messages. Also, lost and found messages can be attached to the frame of an MSC reference expression or an inline expression. In this thesis such lost and found messages are not treated.

4.4.4 Instance create and instance stop events

Instance create events are represented by atomic actions from the set A_{cr} and instance stop events are represented by atomic actions from the set A_{stop} .

Definition 4.4.4.1 The sets A_{cr} and A_{stop} are defined as follows:

$$\begin{aligned}
A_{cr} &= \{ \text{create}(i, j, p), \text{create}(i, j, -) \\
&\quad | i, j \in \mathcal{L}(\langle \text{instance name} \rangle) \wedge p \in \mathcal{L}(\langle \text{parameter list} \rangle) \}, \\
A_{stop} &= \{ \text{stop}(i) | i \in \mathcal{L}(\langle \text{instance name} \rangle) \}.
\end{aligned}$$

The first parameter of these atomic actions represents the instance on which the event is defined. In case of a create event the second parameter of the atomic action is the name of the created instance and the third parameter represents the parameter list. If the parameter list is not specified for a create event, this is indicated in the atomic action by denoting the third parameter by $-$.

Definition 4.4.4.2 Let $i \in \mathcal{L}(\langle \text{instance name} \rangle)$. Then, for $j \in \mathcal{L}(\langle \text{instance name} \rangle)$ and $p \in \mathcal{L}(\langle \text{parameter list} \rangle)$,

$$\begin{aligned}
\llbracket \text{create } j \rrbracket_i &= \text{create}(i, j, -), \\
\llbracket \text{create } j(p) \rrbracket_i &= \text{create}(i, j, p), \\
\llbracket \text{stop} \rrbracket_i &= \text{stop}(i).
\end{aligned}$$

Recommendation Z.120 states that for a created instance none of its events can be executed before the corresponding create event on the creating instance has been executed. This ordering requirement is not formalized in this thesis. There are several possibilities to extend the approach in such a way that this requirement is also taken care of.

4.4.5 Timer events

Timer events are represented by atomic actions from the set A_{timer} .

Definition 4.4.5.1 The set A_{timer} is defined as follows:

$$A_{timer} = \{set(i, t, d), set(i, t, -), reset(i, t), timeout(i, t) \\ | i \in \mathcal{L}(\langle\text{instance name}\rangle) \wedge t \in \mathcal{L}(\langle\text{timer name}\rangle) \\ \wedge d \in \mathcal{L}(\langle\text{duration name}\rangle)\}.$$

The first parameter of these atomic actions represents the name of the instance on which the timer event is defined, the second parameter represents the name of the timer and, in case of a timer set event, the third parameter represents the duration name associated with the timer set event. If no duration name is associated with the timer set event this is denoted by $-$. If in a timer event no duration name occurs this is represented in the atomic action by denoting its last parameter by $-$.

Definition 4.4.5.2 Let $i \in \mathcal{L}(\langle\text{instance name}\rangle)$. Then, for $t \in \mathcal{L}(\langle\text{timer name}\rangle)$ and $d \in \mathcal{L}(\langle\text{duration name}\rangle)$,

$$\begin{aligned} \llbracket \text{set } t \rrbracket_i &= set(i, t, -), \\ \llbracket \text{set } t(d) \rrbracket_i &= set(i, t, d), \\ \llbracket \text{reset } t \rrbracket_i &= reset(i, t), \\ \llbracket \text{timeout } t \rrbracket_i &= timeout(i, t). \end{aligned}$$

4.4.6 Conditions

Although conditions are not really events, they are only used as a means to restrict vertical composition in HMSCs, they are best treated in this section. With a condition no atomic action is associated. As a condition does not disallow any further events it is represented by the empty process ε .

Definition 4.4.6.1 Then, for $cl \in \mathcal{L}(\langle\text{condition name list}\rangle)$

$$\llbracket \text{condition } cl \rrbracket = \varepsilon.$$

4.5 Semantics of causally ordered events

Semantically, events are represented by atomic actions. These atomic actions can have parameters which play a symbolic role. For example the output of a message with name m by instance i with receiver instance j is represented by $out(i, -, j, m)$. The corresponding message input event is represented by $in(i, -, j, m)$. With these parameters enough information is available to decide whether a message output and

a message input are corresponding. For the correspondence of events that are involved in a causal ordering this is not so easy. For example if a local action with name a on instance i must precede a local action with name b on instance j then this cannot be determined from the atomic actions $action(i, a)$ and $action(j, b)$ representing these events. This implies that additional information has to be maintained.

There are three situations that need to be considered:

- the other end of the causal ordering is an event attached to an instance;
- the other end of the causal ordering is a gate on the frame of the MSC;
- the other end of the causal ordering is an actual gate on the frame of an MSC reference expression or an inline expression.

For each of these situations different information is available. Therefore, three different representations are used. Additionally, this has the advantage that the three situations can be distinguished.

In the first situation both events that are involved in the causal ordering are known via the event names. Therefore the causal ordering can easily be represented via the event names. For example, the event " $i_1 : l_1 e_1$ **before** l_2 " describes that the event e_1 with event name l_1 is causally ordered before an unknown event with event name l_2 . This is represented by labeling the atomic action representing the event e_1 with the pair $l_1 \mapsto l_2$. The corresponding event, say " $i_2 : l_2 e_2$ **after** l_1 ", is labeled with the pair $l_1 \mapsto l_2$ as well. Thus it is easy to establish that these two events are ordered.

In the second situation only one of the events is available. In a broader context, however, the gate may be connected to another gate or event and then both events will be available. Thus, even although there is only one event, it still is necessary to maintain the information. An example of this situation is the event " $i_1 : l_1 e_1$ **before env vi-a** g ". The available information in this case is that the event with event name l_1 is ordered before an event that might be connected to gate g . This is represented by $l_1 \mapsto env(g)$. In Section 4.9 and in Section 4.10, where MSC reference expressions and inline expressions are treated respectively, we will see that if an MSC is placed in a context in which the gate g is connected the information by which the atomic actions are labeled, will be changed accordingly.

The third situation is comparable to the second situation. In this case however, it is known that the order arrow connects to an actual gate. Textually this is indicated by a reference to an MSC reference expression or an inline expression by means of a reference identification. An example is the event " $i_1 : l_1 e_1$ **before reference** l_2 **vi-a** g ". As there can be more than one occurrence of gate g due to multiple references to MSCs, the reference identification is essential information. The causal ordering is represented by the pair $l_1 \mapsto (l_2, g)$.

With an orderable event an event name can be associated. These event names are used to refer to an event when describing a causal ordering. The event names are also

necessary to distinguish multiple occurrences of the same causally ordered event. An example thereof is given in Example 4.5.1 below.

Example 4.5.1 In order to explain the reason that the event names are necessary to distinguish multiple occurrences of atomic actions representing a causally ordered event, consider the MSCs in Figure 4.15. If the two occurrences of local action a on

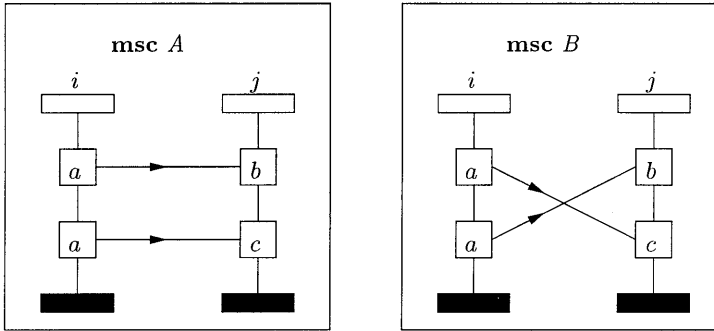


Figure 4.15: Necessity of event names.

instance i cannot be distinguished semantically, then it is impossible to distinguish the MSCs A and B . These MSCs should be distinguished as MSC A has the trace

$$\text{action}(i, a) \text{ action}(j, b) \text{ action}(i, a) \text{ action}(j, c),$$

while MSC B does not have this trace.

Textually the event name is the only means to distinguish the two occurrences of the same event. Therefore, the atomic actions representing causally ordered events are labeled by the event name that is used in the textual syntax for describing the causal ordering. As one event can be involved in many causal orderings the atomic action is labeled with a set of ordering requirements. Also this set of ordering requirements is relevant for distinguishing multiple occurrences of the same atomic action. An example to indicate this necessity is given in Example 4.5.2.

Example 4.5.2 Suppose that MSC C only contains a local action a on instance i which is connected by a causal order arrow with a gate g on the environment. Then, the two occurrences of local action a in MSC A in Figure 4.16 cannot be distinguished based on the event name alone. Without taking the reference identification of the MSC reference identifications into account the MSCs A and B cannot be distinguished textually. MSC A has a trace $\text{action}(i, a), \text{action}(j, b), \text{action}(i, a), \text{action}(j, c)$, whereas MSC B does not have this trace.

The set of ordering requirements for the atomic actions representing the local action a contain this reference identification. Thus, the two occurrences of local action a can be distinguished by looking at the set of ordering requirements.

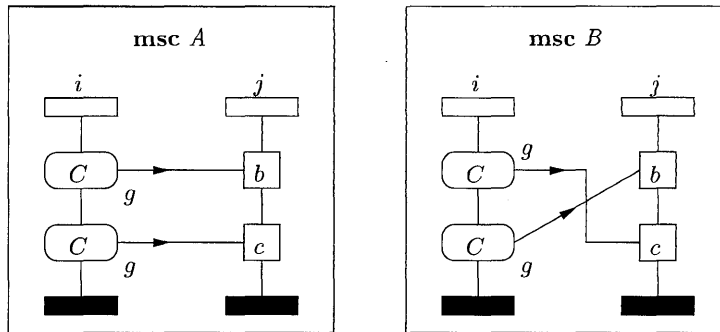


Figure 4.16: Necessity of set of ordering requirements.

The sets \mathcal{AOA} and \mathcal{AOR} represent the information that is provided textually when an event is causally ordered. An abstract order address, that is an element of the set \mathcal{AOA} , describes one half of a causal ordering. An abstract ordering requirement, that is an element of the set \mathcal{AOR} , describes both halves of a causal ordering. The abstract ordering requirement $d \mapsto d'$ should be interpreted as follows: the address represented by the abstract order address d is causally ordered before the address represented by the abstract order address d' .

Definition 4.5.3 The set \mathcal{AOA} is defined as follows:

$$\begin{aligned} \mathcal{AOA} &= \mathcal{L}(\langle \text{event name} \rangle) \\ &\cup \{ \text{env}(g) \mid g \in \mathcal{L}(\langle \text{gate name} \rangle) \} \\ &\cup \mathcal{AG}. \end{aligned}$$

The set \mathcal{AOR} is defined as follows:

$$\mathcal{AOR} = \mathcal{AOA} \times \mathcal{AOA}.$$

Not all elements of \mathcal{AOR} will appear in the semantics. Usually we omit the curly brackets from a set $O \subseteq \mathcal{AOR}$ and write $d \mapsto d'$ for an element $(d, d') \in \mathcal{AOR}$. Thus, the set of abstract ordering requirements $\{(l1, l2), (l1, \text{env}(g)), (l3, g), l1\}$ is written as $l1 \mapsto l2, l1 \mapsto \text{env}(g), (l3, g) \mapsto l1$.

The mapping S associates with an order destination, as it is represented in the textual syntax, an element of the set \mathcal{AOA} , that is, an abstract order address, as explained informally before.

Definition 4.5.4 The mapping $S : \mathcal{L}(\langle \text{order dest} \rangle) \rightarrow \mathcal{AOA}$ is for all $e \in \mathcal{L}(\langle \text{event name} \rangle)$, $l \in \mathcal{L}(\langle \text{reference identification} \rangle)$ and $g \in \mathcal{L}(\langle \text{gate name} \rangle)$, defined as follows:

$$\begin{aligned} S(e) &= e, \\ S(\text{env via } g) &= \text{env}(g), \\ S(\text{reference } l \text{ via } g) &= (l, g), \\ S(\text{inline } l \text{ via } g) &= (l, g). \end{aligned}$$

Then, some notation is introduced for the set of all atomic actions and for labelled atomic actions.

Definition 4.5.5 (Labelled atomic actions) The set A is defined as follows:

$$A = A_{act} \cup A_{out} \cup A_{in} \cup A_{lost} \cup A_{found} \cup A_{cr} \cup A_{stop} \cup A_{timer}.$$

The sets LA , LA_{out} , LA_{in} and LA_{msg} are defined as follows:

$$\begin{aligned} LA &= \{a, a_e, a_e^O \mid a \in A \wedge e \in \mathcal{L}(\langle \text{event name} \rangle) \wedge O \subseteq \mathcal{AOR}\}, \\ LA_{out} &= \{a, a_e, a_e^O \mid a \in A_{out} \wedge e \in \mathcal{L}(\langle \text{event name} \rangle) \wedge O \subseteq \mathcal{AOR}\}, \\ LA_{in} &= \{a, a_e, a_e^O \mid a \in A_{in} \wedge e \in \mathcal{L}(\langle \text{event name} \rangle) \wedge O \subseteq \mathcal{AOR}\}, \\ LA_{msg} &= LA_{out} \cup LA_{in}. \end{aligned}$$

The above definition allows the atomic actions representing instance stop events to be labeled. However, as instance stop events can not be used in a causal ordering, no such labeled atomic actions will be used in the semantics of MSC.

The set of labelled atomic actions LA is the instantiation of the set of atomic actions A for the process theory. Next we define how the mapping ℓ , which associates with an atomic action the instance it is defined on, is instantiated.

Definition 4.5.6 The mapping $\ell : A \rightarrow \mathcal{L}(\langle \text{instance name} \rangle)$ is for $i, j \in \mathcal{L}(\langle \text{instance name} \rangle)$, $a \in \mathcal{L}(\langle \text{action character string} \rangle)$, $G \in \mathcal{AG}$, $m \in \mathcal{L}(\langle \text{message name} \rangle)$, $p \in \mathcal{L}(\langle \text{parameter list} \rangle)$, $t \in \mathcal{L}(\langle \text{timer name} \rangle)$ and $d \in \mathcal{L}(\langle \text{duration name} \rangle)$ defined as follows:

$$\begin{array}{ll} \ell(\text{action}(i, a)) &= i, & \ell(\text{found}(-, j, m)) &= j, \\ \ell(\text{out}(i, G, j, m)) &= i, & \ell(\text{found}(\text{env}, j, m)) &= j, \\ \ell(\text{out}(i, G, -, m)) &= i, & \ell(\text{create}(i, j, p)) &= i, \\ \ell(\text{in}(i, G, j, m)) &= j, & \ell(\text{create}(i, j, -)) &= i, \\ \ell(\text{in}(-, G, j, m)) &= j, & \ell(\text{stop}(i)) &= i, \\ \ell(\text{lost}(i, j, m)) &= i, & \ell(\text{set}(i, t, d)) &= i, \\ \ell(\text{lost}(i, -, m)) &= i, & \ell(\text{set}(i, t, -)) &= i, \\ \ell(\text{lost}(i, \text{env}, m)) &= i, & \ell(\text{reset}(i, t)) &= i, \\ \ell(\text{found}(i, j, m)) &= j, & \ell(\text{timeout}(i, t)) &= i. \end{array}$$

The mapping $\ell : LA \rightarrow \mathcal{L}(\langle \text{instance name} \rangle)$ is for $a \in A$, $e \in \mathcal{L}(\langle \text{event name} \rangle)$ and $O \subseteq \mathcal{AOR}$ defined as follows:

$$\begin{aligned} \ell(a) &= \ell(a), \\ \ell(a_e) &= \ell(a), \\ \ell(a_e^O) &= \ell(a). \end{aligned}$$

The semantics of an ordered event is obtained as follows. The event that is ordered is translated into an atomic action as defined in Section 4.4. This atomic action is labelled with the event name and a set of abstract ordering requirements.

Definition 4.5.7 (Ordered events) Let $i \in \mathcal{L}(\langle \text{instance name} \rangle)$. Then, for all $l \in \mathcal{L}(\langle \text{event name} \rangle)$, $enl, enl' \in \mathcal{L}(\langle \text{event name list} \rangle)$ and $e \in \mathcal{L}(\langle \text{orderable event} \rangle)$,

$$\begin{aligned} \llbracket l \text{ e before } enl \rrbracket_i &= (\llbracket e \rrbracket_i)_l^{\text{before}_l(enl)}, \\ \llbracket l \text{ e after } enl \rrbracket_i &= (\llbracket e \rrbracket_i)_l^{\text{after}_l(enl)}, \\ \llbracket l \text{ e before } enl \text{ after } enl' \rrbracket_i &= (\llbracket e \rrbracket_i)_l^{\text{before}_l(enl) \cup \text{after}_l(enl')}. \end{aligned}$$

where the mappings $\text{before}_l, \text{after}_l : \mathcal{L}(\langle \text{event name list} \rangle) \rightarrow \mathcal{P}(\mathcal{AOR})$ are, for $d \in \mathcal{L}(\langle \text{order dest} \rangle)$ and $enl \in \mathcal{L}(\langle \text{event name list} \rangle)$, defined inductively as follows:

$$\begin{aligned} \text{before}_l(d) &= \{(l, S(d))\}, \\ \text{before}_l(d, enl) &= \{(l, S(d))\} \cup \text{before}_l(enl), \\ \text{after}_l(d) &= \{(S(d), l)\}, \\ \text{after}_l(d, enl) &= \{(S(d), l)\} \cup \text{after}_l(enl). \end{aligned}$$

Example 4.5.8 Consider the MSC given in Figure 4.17. In the MSC also the event names and reference identification as they are used in the textual syntax are indicated. Local action a on instance j is represented in the semantics by the atomic action

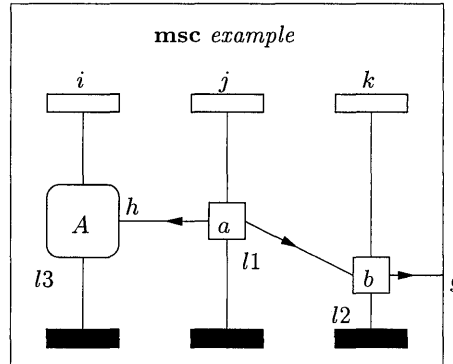


Figure 4.17: Annotated example MSC.

$\text{action}(j, a)_{l1}^{\{l1 \mapsto (l3, h), l1 \mapsto l2\}}$. Local action b on instance k is represented by the atomic action $\text{action}(k, b)_{l2}^{\{l1 \mapsto l2, l2 \mapsto env(g)\}}$.

The atomic actions representing the ordered events are labeled by the event name and the set of ordering requirements. We have indicated by means of the examples 4.5.1 and 4.5.2 that this is necessary. There are alternatives however. The set of ordering requirements is only relevant for distinguishing atomic actions in so far that the reference identification occurs in it. If we label the atomic action with this reference identification directly, it is possible to get rid of the set of ordering requirements as soon as all its orderings have been taken into account.

4.6 Vertical and horizontal composition

If two MSC fragments are composed vertically or horizontally, it is possible that the MSC fragments contain corresponding message events or corresponding causally ordered events. A message output event and a message input event are considered to be corresponding if they have the same message name and either the same sender instance and receiver instance, or the message output is sent to a gate which is connected to the gate from which the message input is received. In a similar way it can be established that two causally ordered events are corresponding. In Definition 4.6.1 these notions are formalized.

Definition 4.6.1 The relation $\circ \rightarrow \circ \subseteq LA \times LA$ is the smallest relation that satisfies: for all $i, j \in \mathcal{L}(\langle \text{instance name} \rangle)$, $m \in \mathcal{L}(\langle \text{message name} \rangle)$, $G \in \mathcal{AG}$, $O, O' \subseteq \mathcal{AOR}$ and $e, e' \in \mathcal{L}(\langle \text{event name} \rangle)$

$$\begin{aligned} out(i, -, j, m)_e^O \circ \rightarrow \circ in(i, -, j, m)_{e'}^{O'}, \\ out(i, G, -, m)_e^O \circ \rightarrow \circ in(-, G, j, m)_{e'}^{O'}. \end{aligned}$$

The relation $\circ \rightarrow \circ \subseteq LA \times LA$ is for all $a, b \in A$, $O, O' \subseteq \mathcal{AOR}$ and $e, e' \in \mathcal{L}(\langle \text{event name} \rangle)$ defined by

$$a_e^O \circ \rightarrow \circ b_{e'}^{O'} \iff (e, e') \in O \cap O' \vee (e, e') \in O \circ O',$$

where relation composition $\circ : \mathcal{P}(\mathcal{AOR}) \rightarrow \mathcal{P}(\mathcal{AOR})$ is for $O, O' \subseteq \mathcal{AOR}$ defined by

$$O \circ O' = \{(s, u) \mid \exists t \in \mathcal{AG} (s, t) \in O \wedge (t, u) \in O'\}.$$

It would have been plausible to define $out(i, G, j, m) \circ \rightarrow \circ in(i, G, j, m)$ as well because the events that are represented by these atomic actions together form a message m that is sent from instance i to instance j via an actual gate G . The reason not to do so is that the relation $\circ \rightarrow \circ$ is used to determine if two atomic actions from different MSC fragments are connected. If this is the case, this gives rise to an ordering requirement and, for the second clause, a renaming of these atomic actions. The atomic actions $out(i, G, j, m)$ and $in(i, G, j, m)$ cannot occur in the semantics of different MSC fragments. As a matter of fact, they are the result of applying the just mentioned renaming to the atomic actions $out(i, G, -, m)$ and $in(-, G, j, m)$.

Two events with event names l and l' are causally ordered if the atomic actions representing them both have the abstract ordering requirement $l \mapsto l'$ (see the first clause) or there exists an actual gate G such that the first has an abstract ordering requirement $l \mapsto G$ and the second has an abstract ordering requirement $G \mapsto l'$ (see the second clause).

The mapping \mathcal{M} associates with a process the set of atomic actions that refer to the message output and message input events that occur in the process. The mapping \mathcal{O} associates with a process the set of all atomic actions that refer to an ordered event.

Definition 4.6.2 The mapping $\mathcal{M} : \mathcal{C}(\Sigma) \rightarrow \mathcal{IP}(LA_{msg})$ is for $\otimes \in \{\mp, \circ^S, \parallel^S \mid S \subseteq LA \times IN \times LA\}$, $\odot \in \{\oplus, \infty, [m,n] \mid m, n \in IN \cup \{\infty\}\}$, $a \in LA$ and $x, y \in \mathcal{C}(\Sigma)$ defined as follows:

$$\begin{aligned} \mathcal{M}(\varepsilon) &= \emptyset, \\ \mathcal{M}(\delta) &= \emptyset, \\ \mathcal{M}(a) &= \begin{cases} \{a\} & \text{if } a \in LA_{msg}, \\ \emptyset & \text{otherwise,} \end{cases} \\ \mathcal{M}(x \otimes y) &= \mathcal{M}(x) \cup \mathcal{M}(y), \\ \mathcal{M}(x^\odot) &= \mathcal{M}(x). \end{aligned}$$

The mapping $\mathcal{O} : \mathcal{C}(\Sigma) \rightarrow \mathcal{IP}(LA)$ is for $\otimes \in \{\mp, \circ^S, \parallel^S \mid S \subseteq LA \times IN \times LA\}$, $\odot \in \{\oplus, \infty, [m,n] \mid m, n \in IN \cup \{\infty\}\}$, $a \in A$, $e \in \mathcal{L}(\langle \text{event name} \rangle)$, $O \subseteq \mathcal{AOR}$ and $x, y \in \mathcal{C}(\Sigma)$ defined as follows:

$$\begin{aligned} \mathcal{O}(\varepsilon) &= \emptyset, \\ \mathcal{O}(\delta) &= \emptyset, \\ \mathcal{O}(a_e^O) &= \begin{cases} \{a_e^O\} & \text{if } O \neq \emptyset, \\ \emptyset & \text{otherwise,} \end{cases} \\ \mathcal{O}(a_e) &= \emptyset, \\ \mathcal{O}(a) &= \emptyset, \\ \mathcal{O}(x \otimes y) &= \mathcal{O}(x) \cup \mathcal{O}(y), \\ \mathcal{O}(x^\odot) &= \mathcal{O}(x). \end{aligned}$$

If two MSC fragments are composed vertically or horizontally it can be the case that one of them contains a message output event and the other a corresponding message input event. In that case the ordering requirement that the message output event precedes the message input event must be taken into account. This is achieved by finding the pairs of atomic actions that refer to a message output or input event (using the mapping \mathcal{M} and the relation $\circ \rightarrow \circ$). Such a pair then gives rise to an ordering requirement. Similarly if the MSC fragments contain corresponding ordered events this also gives rise to an ordering requirement. The mappings *MsgReq* and *OrdReq* are used to obtain the ordering requirements that must be taken into account when two MSC fragments are composed due to the requirement that an output precedes the corresponding input and due to causal order relations between orderable events.

Definition 4.6.3 The mapping $MsgReq : \mathcal{C}(\Sigma) \times \mathcal{C}(\Sigma) \rightarrow \mathcal{IP}(LA_{out} \times LA_{in})$ is for $x, y \in \mathcal{C}(\Sigma)$ defined as follows:

$$\begin{aligned} MsgReq(x, y) &= \{o \mapsto i \mid o \circ \rightarrow \circ i \wedge o \in \mathcal{M}(x) \wedge i \in \mathcal{M}(y)\} \\ &\cup \{o \mapsto i \mid o \circ \rightarrow \circ i \wedge o \in \mathcal{M}(y) \wedge i \in \mathcal{M}(x)\}. \end{aligned}$$

The mapping $OrdReq : \mathcal{C}(\Sigma) \times \mathcal{C}(\Sigma) \rightarrow \mathcal{IP}(LA \times LA)$ is for $x, y \in \mathcal{C}(\Sigma)$ defined as follows:

$$\begin{aligned} OrdReq(x, y) &= \{s \mapsto d \mid s \circ \rightarrow \circ d \wedge s \in \mathcal{O}(x) \wedge d \in \mathcal{O}(y)\} \\ &\cup \{s \mapsto d \mid s \circ \rightarrow \circ d \wedge s \in \mathcal{O}(y) \wedge d \in \mathcal{O}(x)\}. \end{aligned}$$

Recall that $o \mapsto i$ and $s \mapsto d$ are shorthands for $o \xrightarrow{0} i$ and $s \xrightarrow{0} d$.

If the connection of a message output event and a message input event is established via a gate it is necessary to change the atomic actions in such a way that the atomic action for the message output event is updated with the receiver instance name and the atomic action for the message input event is updated with the sender instance name. Before the connection was established these names were not known and therefore indicated by $_$. Given two processes x and y the mapping $f(x, y)$ associates with every atomic action a possibly renamed atomic action. Note that for output events this renaming only applies to the receiver instance part and for input events only to the sender instance part.

Definition 4.6.4 Let $x, y \in \mathcal{C}(\Sigma)$. Then, the mapping $f(x, y) : LA \rightarrow LA$ is for $i, j \in \mathcal{L}(\langle \text{instance name} \rangle)$, $m \in \mathcal{L}(\langle \text{message name} \rangle)$, $G \in \mathcal{AG}$, $S, S' \subseteq \mathcal{AOR}$, $a \in LA$ and $e, e' \in \mathcal{L}(\langle \text{event name} \rangle)$ defined as follows

- if $out(i, G, _, m)_e^S \in \mathcal{M}(x) \wedge in(_, G, j, m)_{e'}^{S'} \in \mathcal{M}(y)$ or $out(i, G, _, m)_e^S \in \mathcal{M}(y) \wedge in(_, G, j, m)_{e'}^{S'} \in \mathcal{M}(x)$, for some j, S', e' , then

$$f(x, y)(out(i, G, _, m)_e^S) = out(i, G, j, m)_e^S;$$

- if $out(i, G, _, m)_{e'}^{S'} \in \mathcal{M}(x) \wedge in(_, G, j, m)_e^S \in \mathcal{M}(y)$ or $out(i, G, _, m)_{e'}^{S'} \in \mathcal{M}(y) \wedge in(_, G, j, m)_e^S \in \mathcal{M}(x)$, for some i, S', e' , then

$$f(x, y)(in(_, G, j, m)_e^S) = in(i, G, j, m)_e^S;$$

- and in all other cases

$$f(x, y)(a) = a.$$

The mapping f is well-defined due to the requirements that, in an MSC document, there are no two gate definitions with the same name and there are no two MSC reference expressions or inline expressions with the same reference identification. A formal proof of this statement will be tedious and will be hardly more convincing.

The vertical and horizontal composition of MSC fragments is described by means of the operators \bullet and \parallel . As can be seen in Definition 4.6.5 below, the ordering requirements and the necessary renaming are computed from the arguments of the operator.

Definition 4.6.5 For $x, y \in \mathcal{C}(\Sigma)$

$$\begin{aligned} x \bullet y &= \rho_{f(x, y)}(x \circ \text{MsgReq}(x, y) \cup \text{OrdReq}(x, y) y), \\ x \parallel y &= \rho_{f(x, y)}(x \parallel \text{MsgReq}(x, y) \cup \text{OrdReq}(x, y) y). \end{aligned}$$

The mappings \mathcal{M} and \mathcal{O} are used to obtain from a process expression the atomic actions representing message events and causally ordered events respectively. These are used to obtain information about the connections between two MSC fragments.

Instead of obtaining this information from the atomic actions that occur in the process expressions it would also have been possible to maintain a set of atomic actions, which are not yet connected, for each MSC fragment. An advantage of this approach would be that also the description of message and causal arrows for which no event is available could be achieved. A disadvantage of this approach is that every MSC fragments has to be represented by a pair, i.e. a process expression and an interface description.

In [RGG95], the authors present three ways of composing MSCs horizontally. These are the environmental merge (\parallel_{env}), the synchronization merge (\parallel_{syn}) and the synchronization condition merge (\parallel_{sync}). The environmental merge is close to the horizontal composition of MSCs as used in recommendation Z.120 and therefore corresponds to the operator \parallel . Output and input events sent to and received from the environment are connected based on gate name identification. The synchronization merge is similar to the interworking merge. The events of the MSCs are interleaved but for messages between instances that occur in both MSCs a synchronization takes place. The synchronization condition merge is similar to the synchronization merge with the difference that synchronization does not take place on messages but on explicitly added synchronization points on the instances. The three operators are presented on the level of MSC by means of some examples only.

4.7 Semantics of coregions

A coregion contains a number (possibly zero) of orderable events. These events are defined on the same instance, but are nevertheless not ordered for that reason. It is however possible that a coregion contains both the output and the input of a message or both events involved in a causal ordering. The semantics of a coregion is thus the horizontal composition of the semantics of its events.

Definition 4.7.1 Let $i \in \mathcal{L}(\langle \text{instance name} \rangle)$. Then, for $e \in \mathcal{L}(\langle \text{orderable event} \rangle)$ and $coevents \in \mathcal{L}(\langle \text{coevent list} \rangle)$

$$\begin{aligned} \llbracket \text{concurrent; endconcurrent} \rrbracket_i &= \varepsilon, \\ \llbracket \text{concurrent; } e \text{ ; } coevents \text{ endconcurrent} \rrbracket_i &= \llbracket e \rrbracket_i \llbracket \text{concurrent; } coevents \\ &\quad \text{endconcurrent} \\ &\quad \rrbracket. \end{aligned}$$

Example 4.7.2 Consider a coregion on instance i which contains the input of message m , a local action a and the output of message m . Then, the semantics of this coregion is given by

$$out(i, -, i, m) \parallel^{R_2} \left(action(i, a) \parallel^{R_1} out(i, -, i, m) \right),$$

where $R_1 = \emptyset$ since $\mathcal{O}(action(i, a)) = \emptyset$ and $R_2 = \{out(i, -, i, m) \mapsto in(i, -, i, m)\}$.

4.8 Semantics of MSC bodies

An MSC body is a possibly empty list of event definitions. As explained before such a list of event definitions is interpreted as a list of MSC fragments that are composed vertically.

Definition 4.8.1 For $eventdef \in \mathcal{L}(\langle \text{event definition} \rangle)$ and $mscbody \in \mathcal{L}(\langle \text{msc body} \rangle)$

$$\begin{aligned} \llbracket \] \rrbracket &= \varepsilon, \\ \llbracket eventdef\ mscbody \rrbracket &= \llbracket eventdef \rrbracket \bullet \llbracket mscbody \rrbracket. \end{aligned}$$

In composing an event definition with an MSC body it can be the case that gates are connected.

There are two types of event definitions that are considered in this section: single instance events and multi instance events. Single instance events are instance events that are defined on one instance. In order to associate an atomic action with the defining instance as a parameter to these single instance events the defining instance is determined and the semantic mapping is labeled with it. Multi instance events are events that are defined on a non-empty set of instances. There is no use for labeling the semantic mapping with these instances as in any relevant case the instances appear again in the textual description of the multi instance event itself.

Definition 4.8.2 For $i \in \mathcal{L}(\langle \text{instance name} \rangle)$, $ilist \in \mathcal{L}(\langle \text{instance name list} \rangle)$, $ievent \in \mathcal{L}(\langle \text{instance event} \rangle)$ and $multiinstanceevent \in \mathcal{L}(\langle \text{multi instance event} \rangle)$

$$\begin{aligned} \llbracket i : ievent ; \rrbracket &= \llbracket ievent \rrbracket_i, \\ \llbracket ilist : multiinstanceevent ; \rrbracket &= \llbracket multiinstanceevent \rrbracket. \end{aligned}$$

Example 4.8.3 (Simple communication) Consider an MSC A with two instances i and j and one message m from instance i to instance j . There are two event-oriented textual representations for this MSC:

$$\begin{array}{lcl} \text{msc } A ; & & \text{msc } A ; \\ i \quad : \text{ out } m \text{ to } j ; & \text{and} & j \quad : \text{ in } m \text{ from } i ; \\ j \quad : \text{ in } m \text{ from } i ; & & i \quad : \text{ out } m \text{ to } j ; \\ \text{endmsc } ; & & \text{endmsc } ; \end{array}$$

Using the first textual representation the recursive equation

$$\bar{A} = out(i, -, j, m) \circ^{out(i, -, j, m) \mapsto in(i, -, j, m)} in(i, -, j, m)$$

is obtained and using the second textual representation the recursive equation

$$\bar{A} = in(i, -, j, m) \circ^{out(i, -, j, m) \mapsto in(i, -, j, m)} out(i, -, j, m)$$

is obtained. The semantics of the MSC is in both cases given by \overline{A} . Operationally the first can be depicted as

$$\overline{A} \quad \begin{array}{l} \text{out}(i, \xrightarrow{-} j, m) \\ \text{in}(i, \xrightarrow{-} j, m) \end{array} \quad \begin{array}{l} \varepsilon \circ \text{out}(i, \xrightarrow{-} j, m) \xrightarrow{1} \text{in}(i, \xrightarrow{-} j, m) \\ \varepsilon \circ \text{out}(i, \xrightarrow{-} j, m) \mapsto \text{in}(i, \xrightarrow{-} j, m) \end{array} \quad \begin{array}{l} \text{in}(i, \xrightarrow{-} j, m) \\ \varepsilon \downarrow \end{array}$$

and the second as

$$\overline{A} \quad \begin{array}{l} \text{in}(i, \xrightarrow{-} j, m) \\ \text{in}(i, \xrightarrow{-} j, m) \end{array} \quad \begin{array}{l} \text{in}(i, \xrightarrow{-} j, m) \circ \text{out}(i, \xrightarrow{-} j, m) \xrightarrow{1} \text{in}(i, \xrightarrow{-} j, m) \\ \varepsilon \circ \text{out}(i, \xrightarrow{-} j, m) \mapsto \text{in}(i, \xrightarrow{-} j, m) \end{array} \quad \begin{array}{l} \varepsilon \\ \varepsilon \downarrow \end{array}$$

Observe that in both cases the same traces can be performed.

Example 4.8.4 (Causal ordering) Consider the MSC from Figure 4.4. Suppose that this MSC is textually represented by

```

mcs example ;
i : l1 action 'a' before l2;
j : l2 action 'b' after l1;
endmcs ;

```

This MSC consists of two MSC fragments. These fragments are semantically represented by

$$\text{action}(i, a)_{l1}^{\{l1 \mapsto l2\}}$$

and

$$\text{action}(j, b)_{l2}^{\{l1 \mapsto l2\}}.$$

Observe that

$$\mathcal{O} \left(\text{action}(i, a)_{l1}^{\{l1 \mapsto l2\}} \right) = \left\{ \text{action}(i, a)_{l1}^{\{l1 \mapsto l2\}} \right\}$$

and

$$\mathcal{O} \left(\text{action}(j, b)_{l2}^{\{l1 \mapsto l2\}} \right) = \left\{ \text{action}(j, b)_{l2}^{\{l1 \mapsto l2\}} \right\}.$$

Then, the following set of ordering requirements is obtained:

$$R = \left\{ \text{action}(i, a)_{l1}^{\{l1 \mapsto l2\}} \mapsto \text{action}(j, b)_{l2}^{\{l1 \mapsto l2\}} \right\}.$$

Thus, the expression representing the semantics of the MSC, is the following:

$$\text{action}(i, a)_{l1}^{\{l1 \mapsto l2\}} \circ^R \text{action}(j, b)_{l2}^{\{l1 \mapsto l2\}}.$$

4.9 Semantics of MSC reference expressions

Textually an MSC reference expression consists of a textual formula containing MSC names and operators, an MSC reference identification and a reference gate interface. The semantics of the textual formula itself is rather easy as a semantical equivalent has been defined for each of the composition operators that can occur in this formula.

We assume the existence of an interpretation $\iota : \mathcal{L}(\langle \text{inf natural} \rangle) \rightarrow \mathbb{N} \cup \{\infty\}$ which associates with every sequence of natural names a natural number and with the keyword **inf** the constant ∞ . Furthermore we extend the normal ordering $<$ on \mathbb{N} to an ordering $<$ on $\mathbb{N} \cup \{\infty\}$ by taking $n < \infty$ for all $n \in \mathbb{N}$.

In the examples we will only use the decimal digits as natural names. The interpretation ι of these is straightforward and therefore left implicit.

Definition 4.9.1 For all $m, n \in \mathcal{L}(\langle \text{inf natural} \rangle)$, $m\text{screfexpr} \in \mathcal{L}(\langle \text{msc ref expr} \rangle)$, $par \in \mathcal{L}(\langle \text{msc ref par expr} \rangle)$, $seq \in \mathcal{L}(\langle \text{msc ref seq expr} \rangle)$, $loop \in \mathcal{L}(\langle \text{msc ref loop expr} \rangle)$, $b \in \mathcal{L}(\langle \text{expr body} \rangle)$ and $m\text{scname} \in \mathcal{L}(\langle \text{msc name} \rangle)$,

$$\begin{aligned} \llbracket par \text{ alt } m\text{screfexpr} \rrbracket &= \llbracket par \rrbracket \mp \llbracket m\text{screfexpr} \rrbracket, \\ \llbracket seq \text{ par } par \rrbracket &= \llbracket seq \rrbracket \parallel \llbracket par \rrbracket, \\ \llbracket loop \text{ seq } seq \rrbracket &= \llbracket loop \rrbracket \circ \llbracket seq \rrbracket, \\ \llbracket loop \langle m, n \rangle b \rrbracket &= \llbracket b \rrbracket^{\iota(m), \iota(n)}, \\ \\ \llbracket \text{empty} \rrbracket &= \varepsilon, \\ \llbracket m\text{scname} \rrbracket &= \overline{m\text{scname}}, \\ \llbracket (m\text{screfexpr}) \rrbracket &= \llbracket m\text{screfexpr} \rrbracket. \end{aligned}$$

Example 4.9.2 The semantics of the textual formula

reference ($A \text{ alt empty}$) **par** $B \text{ seq } C$

is given by the process $\overline{A} \mp \varepsilon \parallel \overline{B} \circ \overline{C}$.

Example 4.9.3 The semantics of the textual formula

reference loop $\langle 5, 3 \rangle A \text{ seq } B$

is given by the process $\overline{A}^{[5,3]} \circ \overline{B}$ which cannot perform any events from MSC A .

If gates of an MSC reference expression are connected on the outside, the gate definitions in the MSCs referenced by the textual formula become actual gates. The semantics of the textual formula contains these gate definitions as the via part of message output events and message input events and as labels of the orderings with which atomic actions can be labelled. For such message gates three different types of connection can exist.

1. A gate can be connected to the environment of the enclosing MSC fragment.
2. A gate can be connected to an instance in the enclosing MSC fragment.
3. A gate can be connected to an actual gate of an MSC reference expression or inline expression in the enclosing MSC fragment.

These three situations are depicted in Figure 4.18 for the MSC reference expression with textual formula A .

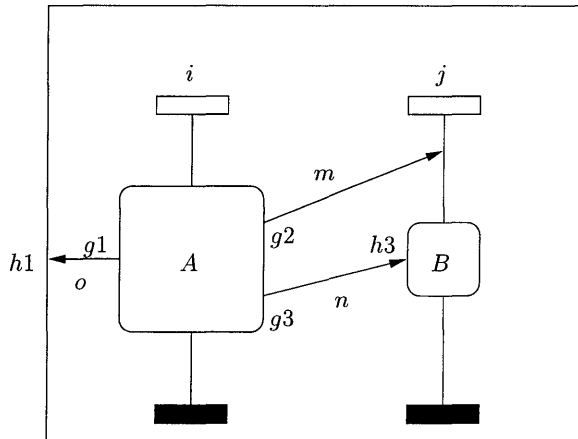


Figure 4.18: Three types of connecting gates.

In the semantics of the textual formula A the gates $g1$, $g2$ and $g3$ appear as $env(g1)$, $env(g2)$ and $env(g3)$ respectively. In the context of the MSC which contains this MSC reference expression these gates are not necessarily connections to the environment anymore. Only gate $g1$ is connected to the environment (again). In order to indicate this situation we replace all occurrences of $env(g1)$, $env(g2)$ and $env(g3)$ by more appropriate and convenient gate names. This renaming is based on the information that is available in the reference gate interface.

1. The gate with name $g1$ is connected externally to the environment via a gate with name $h1$. Therefore, all occurrences of $env(g1)$ in the semantics of A are replaced by $env(h1)$.
2. The gate with name $g2$ is connected externally to instance j by means of a message arrow. The intuition is that the output of message m in A is received by instance j . Thus, this communication will become internal. This is part of the reason why $env(g2)$ is replaced by $(l, g2)$. Another reason is that we must be able to distinguish the actual gates of references to an MSC in different MSC reference expressions. As the MSC reference identification is unique, the combination of the MSC reference identification and the gate name is a nice name for the actual gate. Looking at the semantics of the message input event

on instance j we find that it also has a via part $(l, g2)$. So additionally, but on purpose, we have created the situation in which we can establish which output and input event together make one communication. Looking back this (partly) motivates the definition of $\circ \rightarrow \circ$ (Definition 4.6.1).

3. For similar reasons the occurrences of $env(g3)$ are replaced by $((l, g3), (l', h3))$ where l' is the MSC reference identification of the MSC reference expression on instance j . Also the occurrences of $env(h3)$ in the semantics of this second MSC reference expression are replaced by $((l, g3), (l', h3))$. This again, gives us a nice way to establish correspondence of the message output event and the message input event.

The information needed for the renamings discussed above is available in the reference gate interface. For the example MSC it contains the entries: “gate $g1$ out o to env via $h1$ ”, “gate $g2$ out m to j ” and “gate $g3$ out n to reference l' via $h3$ ”.

The mapping G that is given in Definition 4.9.6 abstracts from the textual representation of the reference gate interface and turns it into a set of connections, that is an abstract gate interface (AGI , see Definition 4.9.4). In case of a message gate such a connection consists of two abstract message addresses (AMA , see Definition 4.9.4). An abstract message address represents a starting or ending point of a message arrow. A connection with a gate h in the environment is indicated by $env(h)$, a connection with an instance j by j and a connection with an MSC reference expression or an inline expression by its identification and the gate used on it. The pairs are ordered such that an arrow is drawn from the first address to the second address.

In case of a causal order gate a connection consists of two abstract order addresses (AOA , see Definition 4.5.3). Also for causal order gates three different connections exist. These are the connection to the environment of the enclosing MSC fragment, the connection to an orderable event in the enclosing MSC fragment and the connection to an actual gate of an MSC reference expression or inline expression. The first and the third case are treated in the same way as message gates are treated. In the second case, the gate definition is replaced by the event name of the orderable event to which the causal order gate is connected in the enclosing MSC fragment.

Definition 4.9.4 The set AMA is defined as follows:

$$\begin{aligned} AMA &= \mathcal{L}(\text{(instance name)}) \\ &\cup \{env(g) \mid g \in \mathcal{L}(\text{(gate name)})\} \\ &\cup AG. \end{aligned}$$

The set AGI is defined as follows:

$$AGI = \mathcal{P}((AMA \times AMA) \cup (AOA \times AOA)).$$

The mapping S associates with an output or input address, as it occurs in the textual representation, an abstract message address.

Definition 4.9.5 The mapping $S : \mathcal{L}(\langle \text{output address} \rangle) \cup \mathcal{L}(\langle \text{input address} \rangle) \rightarrow \mathcal{AMA}$ is for all $i \in \mathcal{L}(\langle \text{instance name} \rangle)$, $g \in \mathcal{L}(\langle \text{gate name} \rangle)$ and $l \in \mathcal{L}(\langle \text{reference identification} \rangle)$ defined by

$$\begin{aligned} S(i) &= i, \\ S(\mathbf{env\ via\ } g) &= env(g), \\ S(\mathbf{reference\ } l \mathbf{\ via\ } g) &= (l, g), \\ S(\mathbf{inline\ } l \mathbf{\ via\ } g) &= (l, g). \end{aligned}$$

The mapping G associates with a reference gate interface an abstract gate interface.

Definition 4.9.6 Let $l \in \mathcal{L}(\langle \text{reference identification} \rangle)$. The mapping $G_l : \mathcal{L}(\langle \text{reference gate interface} \rangle) \rightarrow \mathcal{AGI}$ is for $refgate \in \mathcal{L}(\langle \text{ref gate} \rangle)$ and $gates \in \mathcal{L}(\langle \text{reference gate interface} \rangle)$ defined inductively by

$$\begin{aligned} G_l() &= \emptyset, \\ G_l(\mathbf{;\ gate\ } refgate \mathbf{\ gates}) &= \{G_l(refgate)\} \cup G_l(gates). \end{aligned}$$

Let $l \in \mathcal{L}(\langle \text{reference identification} \rangle)$. The mapping $G_l : \mathcal{L}(\langle \text{ref gate} \rangle) \rightarrow \mathcal{AGI}$ is for all $g \in \mathcal{L}(\langle \text{gate name} \rangle)$, $m \in \mathcal{L}(\langle \text{message name} \rangle)$, $a \in \mathcal{L}(\langle \text{output address} \rangle) \cup \mathcal{L}(\langle \text{input address} \rangle)$ and $d \in \mathcal{L}(\langle \text{order dest} \rangle)$ defined by

$$\begin{aligned} G_l(g \mathbf{\ out\ } m \mathbf{\ to\ } a) &= \{(l, g), S(a)\}, \\ G_l(g \mathbf{\ in\ } m \mathbf{\ from\ } a) &= \{S(a), (l, g)\}, \\ G_l(g \mathbf{\ before\ } d) &= \{(l, g), S(d)\}, \\ G_l(g \mathbf{\ after\ } d) &= \{S(d), (l, g)\}. \end{aligned}$$

In the following definition a mapping *via* is defined. This mapping implements the renaming of the gate definitions of the referenced MSCs into actual gates or other gate definitions following the lines explained before. For *via* to be well-defined it is necessary that there are no two gate definitions with the same gate name, not even if they have another direction. It is also necessary that there are no two different external connections for a given gate g on the MSC reference expression. The mapping *via* is extended to the sets of ordering requirements by which the atomic actions are labelled and to the atomic actions in the obvious way.

Definition 4.9.7 Let $l \in \mathcal{L}(\langle \text{reference identification} \rangle)$ and let $gates \subseteq \mathcal{AGI}$. The mapping $via(l, gates) : (\mathcal{AMG} \cup \mathcal{AOA}) \rightarrow (\mathcal{AMG} \cup \mathcal{AOA})$ is for $g, h \in \mathcal{L}(\langle \text{gate name} \rangle)$, $i, j \in \mathcal{L}(\langle \text{instance name} \rangle)$, $l' \in \mathcal{L}(\langle \text{reference identification} \rangle)$ and $G \in \mathcal{AMG} \cup \mathcal{AOA}$ defined as follows:

$$\begin{aligned} via(l, gates)(env(g)) &= \begin{cases} env(h) & \text{if } ((l, g), env(h)) \in gates \\ & \text{or } (env(h), (l, g)) \in gates \text{ for some } h, \\ (l, g) & \text{if } ((l, g), j) \in gates \\ & \text{or } (i, (l, g)) \in gates \text{ for some } i, j, \\ ((l, g), (l', h)) & \text{if } ((l, g), (l', h)) \in gates \text{ for some } l', h, \\ ((l', h), (l, g)) & \text{if } ((l', h), (l, g)) \in gates \text{ for some } l', h, \\ env(g) & \text{otherwise,} \end{cases} \\ via(l, gates)(G) &= G \quad \text{otherwise.} \end{aligned}$$

Let $l \in \mathcal{L}$ (reference identification) and let $gates \subseteq AGI$. The mapping $via(l, gates) : LA \rightarrow LA$ is for $i, j \in \mathcal{L}$ (instance name), $g \in \mathcal{L}$ (gate name), $m \in \mathcal{L}$ (message name), $O \subseteq \mathcal{AOR}$, $e \in \mathcal{L}_\lambda$ (event name) and $a \in A$ defined as follows:

$$\begin{aligned} via(l, gates) (out(i, env(g), j, m)_e^O) &= out(i, via(l, gates)(env(g)), j, m)_e^{via(l, gates)(O)}, \\ via(l, gates) (in(i, env(g), j, m)_e^O) &= in(i, via(l, gates)(env(g)), j, m)_e^{via(l, gates)(O)}, \\ via(l, gates) (a_e^O) &= a_e^{via(l, gates)(O)} \quad \text{otherwise.} \end{aligned}$$

Let $l \in \mathcal{L}$ (reference identification) and let $gates \subseteq AGI$. The mapping $via(l, gates) : \mathcal{P}((AMG \cup AOA) \times (AMG \cup AOA)) \rightarrow \mathcal{P}((AMG \cup AOA) \times (AMG \cup AOA))$ is for $O \subseteq \mathcal{AOR}$ defined as follows:

$$via(l, gates)(O) = \{(via(l, gates)(g_1), via(l, gates)(g_2)) \mid (g_1, g_2) \in O\}.$$

Using the above definitions the semantics of the MSC reference expression where the gate definitions are replaced by actual gates as indicated by the MSC reference gate interface can then be described by $\rho_{via(l, G_l(gates))}(\llbracket mscrefexpr \rrbracket)$ where l is the MSC reference identification, $gates$ is the reference gate interface and $mscrefexpr$ is the textual formula.

However, it is possible that two gates of the MSC reference expression are connected. Therefore, an ordering requirement must be added to the semantics and, if this is a connection between message gates, atomic actions have to be renamed. The mapping $g(x)$ defined below gives the necessary renaming and the mapping $R(x)$ defines the (not yet renamed) ordering requirements.

Definition 4.9.8 The mapping $R : \mathcal{C}(\Sigma) \rightarrow \mathcal{P}(LA \times LA)$ is for $x \in \mathcal{C}(\Sigma)$ defined as follows:

$$\begin{aligned} R(x) = & \{out(i, ((l, g), (l, g')), -, m)_e^O \mapsto in(-, ((l, g), (l, g')), j, m)_{e'}^{O'} \\ & \mid out(i, ((l, g), (l, g')), -, m)_e^O \in \mathcal{M}(x) \\ & \wedge in(-, ((l, g), (l, g')), j, m)_{e'}^{O'} \in \mathcal{M}(x)\} \\ \cup & \{a_e^{O \cup \{(e, ((l, g), (l, g')))\}} \mapsto b_{e'}^{O' \cup \{((l, g), (l, g')), e'\}} \\ & \mid a_e^{O \cup \{(e, ((l, g), (l, g')))\}} \in \mathcal{O}(x) \wedge b_{e'}^{O' \cup \{((l, g), (l, g')), e'\}} \in \mathcal{O}(x)\}. \end{aligned}$$

The definition of the mapping R is similar to the definition of the mappings $MsgReq$ and $OrdReq$ used for the definition of the vertical composition of MSC fragments (see Definition 4.6.5). The difference is that this time we are only interested in ordering requirements due to the connection of gates which are both from the MSC reference expression. This is manifest in the definition of R by only considering actual gates of the form $((l, g), (l, g'))$. Similarly, in the definition below, the renaming of the atomic actions only applies to message events with a gate part of this form.

In order to simplify the following definition we define $\mathcal{L}_\lambda(\langle X \rangle) = \mathcal{L}(\langle X \rangle) \cup \{\lambda\}$ where $\lambda \notin \mathcal{L}(\langle X \rangle)$. We use the following notations: a_e can be written as a_e^\emptyset and a can be written as a_λ^\emptyset .

Definition 4.9.9 Let $x \in \mathcal{C}(\Sigma)$. The mapping $g(x): LA \rightarrow LA$ is for $i, j \in \mathcal{L}(\langle \text{instance name} \rangle)$, $m \in \mathcal{L}(\langle \text{message name} \rangle)$, $l \in \mathcal{L}(\langle \text{reference identification} \rangle)$, $g, h \in \mathcal{L}(\langle \text{gate name} \rangle)$, $O, O' \subseteq \mathcal{AOR}$, $e, e' \in \mathcal{L}_\lambda(\langle \text{event name} \rangle)$ and $a \in LA$ defined as follows

- if $out(i, ((l, g), (l, h)), -, m)_e^O, in(-, ((l, g), (l, h)), j, m)_{e'}^{O'} \in \mathcal{M}(x)$ for some j, O', e' , then

$$g(x) (out(i, ((l, g), (l, h)), -, m)_e^O) = out(i, ((l, g), (l, h)), j, m)_e^O;$$

- if $out(i, ((l, g), (l, h)), -, m)_{e'}^{O'}, in(-, ((l, g), (l, h)), j, m)_e^O \in \mathcal{M}(x)$ for some i, O', e' , then

$$g(x) (in(-, ((l, g), (l, h)), j, m)_e^O) = in(i, ((l, g), (l, h)), j, m)_e^O;$$

- and in all other cases

$$g(x)(a) = a.$$

Due to the strict requirements on the use of gates in MSC this mapping $g(x)$ is well-defined.

The semantics of an MSC reference expression is obtained by taking into account the ordering requirements due to the connection of gates of the MSC reference expression and the required renaming as expressed in the following definition.

Definition 4.9.10 For all $l \in \mathcal{L}(\langle \text{reference identification} \rangle)$, $mscrefexpr \in \mathcal{L}(\langle \text{msc ref expr} \rangle)$ and $gates \in \mathcal{L}(\langle \text{reference gate interface} \rangle)$

$$\llbracket \text{reference } l : mscrefexpr \text{ gates} \rrbracket = \rho_g(\rho_v(\llbracket mscrefexpr \rrbracket) \circ^R \varepsilon),$$

where $v = via(l, G_l(gates))$,
 $g = g(\rho_v(\llbracket mscrefexpr \rrbracket))$,
 $R = R(\rho_v(\llbracket mscrefexpr \rrbracket))$.

4.10 Semantics of inline expressions

The semantics of inline expressions is easily obtained from the semantics of the arguments of an inline expression by combining them by means of the semantical equivalent of the operation indicated in the inline expression. The operation indicated with the keyword **alt** is interpreted by the operator delayed choice \mp , the operation indicated by **par** is interpreted as delayed parallel composition \parallel and the operation **loop** $\langle m, n \rangle$ by the operator $^{[m, n]}$.

The approach for the semantics of inline expressions is similar to the way MSC reference expressions are treated. The main difference is in the computation of the abstract gate interface. The description of the external connections of the gates of the inline expression is distributed over the arguments.

Definition 4.10.1 Let $l \in \mathcal{L}(\langle \text{reference identification} \rangle)$. The mapping $G_l : \mathcal{L}(\langle \text{alt list} \rangle) \rightarrow \mathcal{AGI}$ is for $gates \in \mathcal{L}(\langle \text{inline gate interface} \rangle)$, $b \in \mathcal{L}(\langle \text{msc body} \rangle)$ and $altlist \in \mathcal{L}(\langle \text{alt list} \rangle)$ defined inductively by:

$$\begin{aligned} G_l(gates \ b) &= G_l(gates), \\ G_l(gates \ b \ \mathbf{alt} \ ; \ altlist) &= G_l(gates) \cup G_l(altlist). \end{aligned}$$

Let $l \in \mathcal{L}(\langle \text{reference identification} \rangle)$. The mapping $G_l : \mathcal{L}(\langle \text{par list} \rangle) \rightarrow \mathcal{AGI}$ is for $gates \in \mathcal{L}(\langle \text{inline gate interface} \rangle)$, $b \in \mathcal{L}(\langle \text{msc body} \rangle)$ and $parlist \in \mathcal{L}(\langle \text{par list} \rangle)$ defined inductively by:

$$\begin{aligned} G_l(gates \ b) &= G_l(gates), \\ G_l(gates \ b \ \mathbf{par} \ ; \ parlist) &= G_l(gates) \cup G_l(parlist). \end{aligned}$$

Let $l \in \mathcal{L}(\langle \text{reference identification} \rangle)$. The mapping $G_l : \mathcal{L}(\langle \text{inline gate interface} \rangle) \rightarrow \mathcal{AGI}$ is for $inlinegate \in \mathcal{L}(\langle \text{inline gate} \rangle)$ and $gates \in \mathcal{L}(\langle \text{inline gate interface} \rangle)$ defined as follows:

$$\begin{aligned} G_l() &= \emptyset, \\ G_l(\mathbf{gate} \ inlinegate \ ; \ gates) &= G_l(inlinegate) \cup G_l(gates). \end{aligned}$$

Let $l \in \mathcal{L}(\langle \text{reference identification} \rangle)$. The mapping $G_l : \mathcal{L}(\langle \text{inline gate} \rangle) \rightarrow \mathcal{AGI}$ is for $g \in \mathcal{L}(\langle \text{gate name} \rangle)$, $m, m' \in \mathcal{L}(\langle \text{message name} \rangle)$, $s \in \mathcal{L}(\langle \text{output address} \rangle)$, $d \in \mathcal{L}(\langle \text{input address} \rangle)$ and $o, o' \in \mathcal{L}(\langle \text{order dest} \rangle)$ defined as follows:

$$\begin{aligned} G_l(g \ \mathbf{in} \ m \ \mathbf{from} \ s \ \mathbf{external} \ \mathbf{out} \ m' \ \mathbf{to} \ d) &= \{(l, g), S(d)\}, \\ G_l(g \ \mathbf{out} \ m \ \mathbf{to} \ d \ \mathbf{external} \ \mathbf{in} \ m' \ \mathbf{from} \ s) &= \{S(s), (l, g)\}, \\ G_l(g \ \mathbf{after} \ o \ \mathbf{external} \ \mathbf{before} \ o') &= \{(l, g), S(o')\}, \\ G_l(g \ \mathbf{before} \ o \ \mathbf{external} \ \mathbf{after} \ o') &= \{S(o'), (l, g)\}. \end{aligned}$$

Please note that the recommendation allows the use of different message names in the internal and external connection of a gate. A static requirement that forbids this should be defined.

Definition 4.10.2 (Inline loop expr.) For $m, n \in \mathcal{L}(\langle \text{inf natural} \rangle)$, $l \in \mathcal{L}(\langle \text{reference identification} \rangle)$, $gates \in \mathcal{L}(\langle \text{inline gate interface} \rangle)$ and $b \in \mathcal{L}(\langle \text{msc body} \rangle)$

$$\llbracket \mathbf{loop} \ \langle m, n \rangle \ \mathbf{begin} \ l \ ; \ gates \ b \ \mathbf{loop} \ \mathbf{end} \rrbracket = \rho_g(\rho_v(\llbracket b \rrbracket^{l(m), l(n)})) \circ^R \varepsilon,$$

where $v = \text{via}(l, G_l(gates))$, $g = g(\rho_v(\llbracket b \rrbracket^{l(m), l(n)}))$ and $R = R(\rho_v(\llbracket b \rrbracket^{l(m), l(n)}))$.

Definition 4.10.3 (Inline alternative expr.) For $l \in \mathcal{L}(\langle \text{reference identification} \rangle)$, $altlist \in \mathcal{L}(\langle \text{alt list} \rangle)$, $gates \in \mathcal{L}(\langle \text{inline gate interface} \rangle)$ and $b \in \mathcal{L}(\langle \text{msc body} \rangle)$

$$\llbracket \mathbf{alt} \ \mathbf{begin} \ l \ ; \ altlist \ \mathbf{alt} \ \mathbf{end} \rrbracket = \rho_g(\rho_v(\llbracket altlist \rrbracket)) \circ^R \varepsilon,$$

$$\begin{aligned} \llbracket gates \ b \rrbracket &= \llbracket b \rrbracket, \\ \llbracket gates \ b \ \mathbf{alt} \ ; \ altlist \rrbracket &= \llbracket b \rrbracket \mp \llbracket altlist \rrbracket, \end{aligned}$$

where $v = \text{via}(l, G_l(altlist))$, $g = g(\rho_v(\llbracket altlist \rrbracket))$ and $R = R(\rho_v(\llbracket altlist \rrbracket))$.

Definition 4.10.4 (Inline parallel expr.) For all $l \in \mathcal{L}(\langle \text{reference identification} \rangle)$, $parlist \in \mathcal{L}(\langle \text{par list} \rangle)$, $gates \in \mathcal{L}(\langle \text{inline gate interface} \rangle)$ and $b \in \mathcal{L}(\langle \text{msc body} \rangle)$

$$\begin{aligned} \llbracket \mathbf{par\ begin\ } l ; parlist \mathbf{\ par\ end} \rrbracket &= \rho_g(\rho_v(\llbracket parlist \rrbracket) \circ^R \varepsilon), \\ \llbracket gates\ b \rrbracket &= \llbracket b \rrbracket, \\ \llbracket gates\ b\ \mathbf{par} ; parlist \rrbracket &= \llbracket b \rrbracket \parallel \llbracket parlist \rrbracket, \end{aligned}$$

where $v = via(l, G_l(parlist))$, $g = g(\rho_v(\llbracket parlist \rrbracket))$ and $R = R(\rho_v(\llbracket parlist \rrbracket))$.

4.11 Semantics of High-level MSCs

Textually an HMSC is described by associating a label with every node except the start node. The start node is described first in the textual syntax by simply listing its successor nodes in a label name list. Then all other nodes are described. Such a description consists of the label name associated with the node followed by a description of the type of the node and a label name list representing the label names of the successor nodes.

If a node has successor nodes then these are interpreted as alternative vertical compositions. For example if a node labeled l has two successor nodes labeled l_1 and l_2 this means that the node l is vertically composed with either node l_1 or l_2 .

Every HMSC is represented by a graph. The nodes of the graph represent the nodes of the HMSC. The edges represent the edges of the HMSC. The type of the node is taken into account as a label on the edges between two nodes. In the textual representation a unique label name is associated with every node, except the start nodes, in the HMSC. This label name is also used to identify the nodes of the graph that corresponds to the HMSC. The start node is not represented in the set of nodes of the graph. Instead the successor nodes of the start node in the HMSC are identified as initial nodes in the corresponding graph. The end nodes of the HMSC are represented in the corresponding graph by final nodes. The label of an edge in the graph of an HMSC between the nodes with label names l_1 and l_2 is based on the type of node that is associated with label name l_1 . If this node is an MSC reference node, the label of the node in the graph is simply the semantics of the textual formula that is written in the node. In all other cases the label is denoted as ε .

Definition 4.11.1 An edge-labeled graph is a quadruple (V, E, I, F) where

- V is a finite set of nodes;
- $E \subseteq V \times \mathcal{C}(\Sigma) \times V$ is a finite set of labeled edges;
- $I \subseteq V$ is a set of initial nodes;
- $F \subseteq V$ is a set of final nodes.

A pair $(v, t, w) \in E$ is often written as $v \xrightarrow{t} w$. In this thesis we assume that for any graph (V, E, I, F) we encounter, the set of nodes can only contain label names. Hence, we define $\mathcal{V} = \mathcal{L}(\langle \text{label name} \rangle)$ and assume $V \subseteq \mathcal{V}$.

Let $G = (V, E, I, F)$. We define $\text{succ}_G(v) = \{w \in V \mid \exists e \in \mathcal{C}(\Sigma) v \xrightarrow{e} w\}$, and we usually omit the subscript G if it is clear from the context.

A graph is called connected if every node of the graph is reachable from an initial node. A final node has no outgoing edges.

In the following definitions we define mappings *Nodes*, *Edges*, *Initial* and *Final* which associate to an MSC expression the sets of nodes, edges, initial nodes and final nodes respectively.

Definition 4.11.2 (Nodes of the graph) The mapping $\text{Nodes}: \mathcal{L}(\langle \text{msc expression} \rangle) \rightarrow \mathcal{P}(\mathcal{V})$ is, for $\text{labels} \in \mathcal{L}(\langle \text{label name list} \rangle)$ and $\text{nodedefs} \in \mathcal{L}(\langle \text{node expression list} \rangle)$, defined by

$$\text{Nodes}(\text{labels}; \text{nodedefs}) = \text{Nodes}(\text{nodedefs}).$$

The mapping $\text{Nodes}: \mathcal{L}(\langle \text{node expression list} \rangle) \rightarrow \mathcal{P}(\mathcal{V})$ is, for all $\text{nodedef} \in \mathcal{L}(\langle \text{node expression} \rangle)$ and $\text{nodedefs} \in \mathcal{L}(\langle \text{node expression list} \rangle)$, defined inductively by

$$\begin{aligned} \text{Nodes}() &= \emptyset, \\ \text{Nodes}(\text{nodedef } \text{nodedefs}) &= \text{Nodes}(\text{nodedef}) \cup \text{Nodes}(\text{nodedefs}). \end{aligned}$$

The mapping $\text{Nodes}: \mathcal{L}(\langle \text{node expression} \rangle) \rightarrow \mathcal{P}(\mathcal{V})$ is, for $l \in \mathcal{L}(\langle \text{label name} \rangle)$, $\text{node} \in \mathcal{L}(\langle \text{node} \rangle)$ and $\text{labels} \in \mathcal{L}(\langle \text{label name list} \rangle)$, defined by

$$\begin{aligned} \text{Nodes}(l: \text{node } \text{seq } (\text{labels});) &= \{l\}, \\ \text{Nodes}(l: \text{node } \text{end};) &= \{l\}. \end{aligned}$$

Definition 4.11.3 (Edges of the graph) The mapping $\text{Edges}: \mathcal{L}(\langle \text{msc expression} \rangle) \rightarrow \mathcal{P}(\mathcal{V} \times \mathcal{C}(\Sigma) \times \mathcal{V})$ is, for $\text{labels} \in \mathcal{L}(\langle \text{label name list} \rangle)$ and $\text{nodedefs} \in \mathcal{L}(\langle \text{node expression list} \rangle)$, defined by

$$\text{Edges}(\text{labels}; \text{nodedefs}) = \text{Edges}(\text{nodedefs}).$$

The mapping $\text{Edges}: \mathcal{L}(\langle \text{node expression list} \rangle) \rightarrow \mathcal{P}(\mathcal{V} \times \mathcal{C}(\Sigma) \times \mathcal{V})$ is, for $\text{nodedef} \in \mathcal{L}(\langle \text{node expression} \rangle)$ and $\text{nodedefs} \in \mathcal{L}(\langle \text{node expression list} \rangle)$, defined inductively by

$$\begin{aligned} \text{Edges}() &= \emptyset, \\ \text{Edges}(\text{nodedef } \text{nodedefs}) &= \text{Edges}(\text{nodedef}) \cup \text{Edges}(\text{nodedefs}). \end{aligned}$$

The mapping $\text{Edges}: \mathcal{L}(\langle \text{node expression} \rangle) \rightarrow \mathcal{P}(\mathcal{V} \times \mathcal{C}(\Sigma) \times \mathcal{V})$ is, for all $l \in \mathcal{L}(\langle \text{label name} \rangle)$, $\text{node} \in \mathcal{L}(\langle \text{node} \rangle)$ and $\text{labels} \in \mathcal{L}(\langle \text{label name list} \rangle)$, defined by

$$\begin{aligned} \text{Edges}(l: \text{node } \text{seq } (\text{labels});) &= \{(l, \llbracket \text{node} \rrbracket, l') \mid l' \in \text{Succ}(\text{labels})\}, \\ \text{Edges}(l: \text{node } \text{end};) &= \emptyset. \end{aligned}$$

where the mapping $Succ : \mathcal{L}(\langle \text{label name list} \rangle) \rightarrow \mathcal{P}(\mathcal{V})$ is, for $l \in \mathcal{L}(\langle \text{label name} \rangle)$ and $list \in \mathcal{L}(\langle \text{label name list} \rangle)$, defined inductively by

$$\begin{aligned} Succ(l) &= \{l\}, \\ Succ(l \text{ alt } list) &= \{l\} \cup Succ(list). \end{aligned}$$

For a definition of the mapping $\llbracket \cdot \rrbracket$ we refer to Definition 4.11.7.

The auxiliary mapping $Succ$ defined in the previous definition is also used in the definition of the mapping $Initial$ which associates with an HMSC the set of nodes which are immediate successor nodes of the start node. In the graph representing the HMSC these are considered initial nodes.

Definition 4.11.4 (Initial nodes) The mapping $Initial : \mathcal{L}(\langle \text{msc expression} \rangle) \rightarrow \mathcal{P}(\mathcal{V})$ is, for $labels \in \mathcal{L}(\langle \text{label name list} \rangle)$ and $nodedefs \in \mathcal{L}(\langle \text{node expression list} \rangle)$, defined by

$$Initial(labels; nodedefs) = Succ(labels).$$

Definition 4.11.5 (Final nodes) The mapping $Final : \mathcal{L}(\langle \text{msc expression} \rangle) \rightarrow \mathcal{P}(\mathcal{V})$ is, for $labels \in \mathcal{L}(\langle \text{label name list} \rangle)$ and $nodedefs \in \mathcal{L}(\langle \text{node expression list} \rangle)$, defined by

$$Final(labels; nodedefs) = Final(nodedefs).$$

The mapping $Final : \mathcal{L}(\langle \text{node expression list} \rangle) \rightarrow \mathcal{P}(\mathcal{V})$ is, for all $nodedef \in \mathcal{L}(\langle \text{node expression} \rangle)$ and $nodedefs \in \mathcal{L}(\langle \text{node expression list} \rangle)$, defined inductively by

$$\begin{aligned} Final() &= \emptyset, \\ Final(nodedef nodedefs) &= Final(nodedef) \cup Final(nodedefs). \end{aligned}$$

The mapping $Final : \mathcal{L}(\langle \text{node expression} \rangle) \rightarrow \mathcal{P}(\mathcal{V})$ is, for $l \in \mathcal{L}(\langle \text{label name} \rangle)$, $node \in \mathcal{L}(\langle \text{node} \rangle)$ and $labels \in \mathcal{L}(\langle \text{label name list} \rangle)$, defined by

$$\begin{aligned} Final(l: node \text{ seq } (labels);) &= \emptyset, \\ Final(l: node \text{ end};) &= \{l\}. \end{aligned}$$

Using the above definitions we can formally define the transformation of an HMSC into a graph.

Definition 4.11.6 The mapping $Graph$ is, for all $mname \in \mathcal{L}(\langle \text{msc name} \rangle)$ and $mscexpr \in \mathcal{L}(\langle \text{msc expression} \rangle)$, defined by

$$Graph(mscexpr) = (\begin{array}{l} Nodes(mscexpr) \\ , Edges(mscexpr) \\ , Initial(mscexpr) \\ , Final(mscexpr) \end{array}).$$

The semantics of a node in an HMSC depends on the type of node. Start nodes, condition nodes, connector nodes and end nodes do not describe the execution of events. Therefore their semantics is given by the empty process ε . An MSC reference node describes the composition of a number of MSCs by means of a textual formula. The semantics of these textual formula has been given in Section 4.9. A parallel frame node describes the horizontal composition of a number of sub-HMSCs. The semantics of one such sub-HMSC is given by considering it as an HMSC by itself.

Definition 4.11.7 (Semantics of a node) The mapping $\llbracket \cdot \rrbracket : \mathcal{L}(\langle \text{node} \rangle) \rightarrow \mathcal{C}(\Sigma)$ is, for $mname \in \mathcal{L}(\langle \text{mname} \rangle)$, $parexpr \in \mathcal{L}(\langle \text{par expression} \rangle)$, $cond \in \mathcal{L}(\langle \text{condition} \rangle)$ and $mscrefexpr \in \mathcal{L}(\langle \text{mref expr} \rangle)$, defined by

$$\begin{aligned} \llbracket \text{empty} \rrbracket &= \varepsilon, \\ \llbracket mname \rrbracket &= \overline{mname}, \\ \llbracket parexpr \rrbracket &= \llbracket parexpr \rrbracket, \\ \llbracket cond \rrbracket &= \varepsilon, \\ \llbracket \text{connect} \rrbracket &= \varepsilon, \\ \llbracket (mscrefexpr) \rrbracket &= \llbracket mscrefexpr \rrbracket. \end{aligned}$$

The mapping $\llbracket \cdot \rrbracket : \mathcal{L}(\langle \text{par expression} \rangle) \rightarrow \mathcal{C}(\Sigma)$ is, for $mscexpr \in \mathcal{L}(\langle \text{mref expr} \rangle)$ and $parexpr \in \mathcal{L}(\langle \text{par expression} \rangle)$, defined inductively by

$$\begin{aligned} \llbracket \text{expr } mscexpr \text{ endmscexpr} \rrbracket &= \llbracket Graph(mscexpr) \rrbracket, \\ \llbracket \text{expr } mscexpr \text{ endmscexpr par } parexpr \rrbracket &= \llbracket Graph(mscexpr) \rrbracket \parallel \llbracket parexpr \rrbracket. \end{aligned}$$

The graph that results from applying the mapping $Graph$ to an HMSC, is connected by definition, and final nodes of such a graph have no outgoing edges.

The transformation of an HMSC into a graph described formally above is now illustrated by means of the following example.

Example 4.11.8 Consider the HMSC in Figure 4.19. Besides the HMSC also the labels associated with each node and the textual syntax of the HMSC are presented in the figure. With this HMSC the graph (V, E, I, F) is associated, where

$$\begin{aligned} V &= \{L1, L2, L3, L4\}, \\ E &= \{ (L1, \overline{disconnected}, L2), \\ &\quad (L1, \overline{disconnected}, L3), \\ &\quad (L2, \overline{message_lost}, L4), \\ &\quad (L3, \overline{time_out}, L4), \\ &\quad (L4, \overline{disconnection}, L1) \}, \\ I &= \{L1\}, \\ F &= \emptyset. \end{aligned}$$

The semantics of an HMSC is then expressed in terms of the semantics of the graph corresponding to the HMSC. A transformation of such a graph into a term is presented shortly.

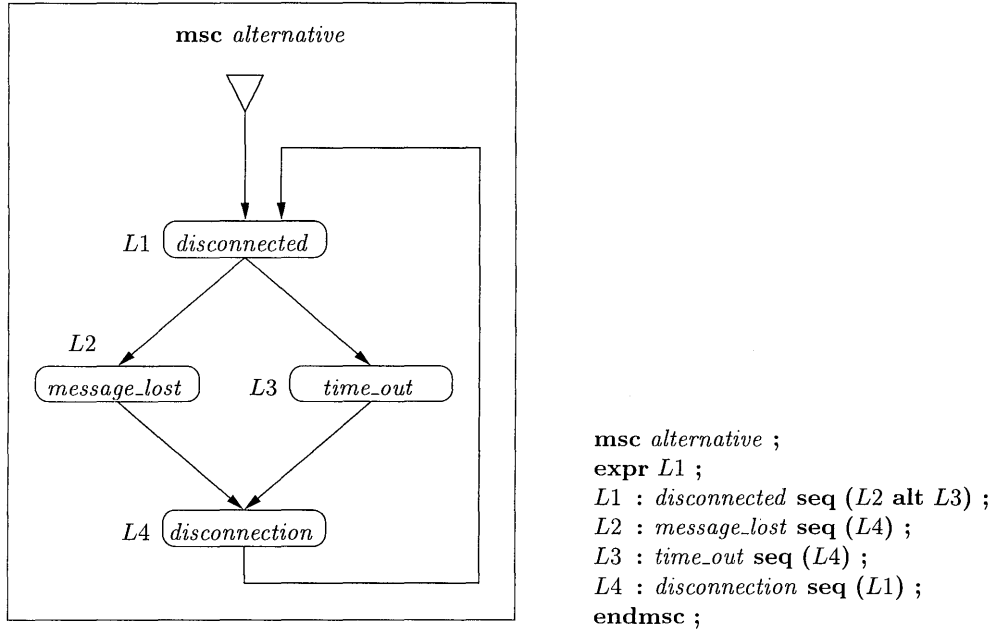


Figure 4.19: HMSC with a loop.

Definition 4.11.9 (Semantics of an HMSC) For $mscname \in \mathcal{L}(\langle \text{msc name} \rangle)$ and $mscexpr \in \mathcal{L}(\langle \text{msc expression} \rangle)$

$$Eqs(\text{msc } mscname; \text{expr } mscexpr \text{ endmsc};) = \{\overline{mscname} = \llbracket \text{Graph}(mscexpr) \rrbracket\}.$$

The semantics of the graph corresponding to an HMSC is obtained by associating an expression with the graph in a way very similar to the way a regular expression is obtained from an automaton. A simple recursive definition of the semantics of a graph is difficult because of the cycles that can be in the graph.

First the graph is transformed into a certain normal form for which the definition of the semantics is easy. This transformation is based on eliminating nodes from the graph.

We present this transformation in the form of a rewrite rule.

Definition 4.11.10 (Elimination of node) Let $G = (V, E, I, F)$ be a graph. Let $v \in V \setminus I$ such that $\text{succ}(v) \not\subseteq \{v\}$. Then $(V, E, I, F) \mapsto_v (V', E', I', F')$ where

$$V' = V \setminus \{v\};$$

$$E' = \left\{ n_1 \xrightarrow{t} n_2 \in E \mid n_1 \neq v \wedge n_2 \neq v \right\}$$

$$\cup \left\{ n_1 \xrightarrow{e_1 \circ e_2} n_2 \mid n_1 \xrightarrow{e_1} v \in E \wedge v \xrightarrow{e_2} n_2 \in E \wedge v \notin \text{succ}(v) \right\}$$

$$\cup \left\{ n_1 \xrightarrow{t} n_2 \mid n_1 \xrightarrow{e_1} v, v \xrightarrow{e_2} n_2 \in E \wedge v \in \text{succ}(v) \wedge t = e_1 \circ \left(\prod_{v \xrightarrow{e} v \in E} e \right)^{\otimes} \circ e_2 \right\};$$

$$I' = I;$$

$$F' = F.$$

Repeated application of this rewriting rule to a graph will result in a normal form as the number of nodes of the graph decreases with one for every application of the rewrite rule. For such a graph in normal form it must be the case that there is no node v left such that the rewrite rule is applicable. Hence, every node of the graph is an initial node or a node with no successor nodes other than itself or with no successor nodes at all.

Theorem 4.11.11 If $G = (V, E, I, F)$ is a connected graph and $G \mapsto_v G'$ for some $v \in V$ and graph G' , then G' is also connected.

Proof The node v that is eliminated is not initial. As G is connected the node v and all its immediate predecessors are reachable from an initial node. By eliminating v the successor nodes of v can become unconnected. By construction every successor of v has an incoming edge from a predecessor of v and is thus reachable from an initial node. \square

In the following definition with a graph in normal form a process expression is associated. The semantics of such a graph in normal form is the delayed choice of the semantics of the initial nodes. The semantics that is associated with a node of the graph depends on the loops of the node and the successor(s) of the node.

Definition 4.11.12 Let $G = (V, E, I, F)$ be graph in normal form. Then the following expression is associated with the graph:

$$\llbracket G \rrbracket = \prod_{i \in I} \llbracket i \rrbracket_G,$$

where

$$\llbracket i \rrbracket_G = \begin{cases} \varepsilon^\infty & \text{if } \text{succ}(i) = \emptyset \wedge i \notin F, \\ \varepsilon & \text{if } \text{succ}(i) = \emptyset \wedge i \in F, \\ \left(\bigwedge_{i \xrightarrow{e} i \in E} e \right)^\infty & \text{if } \text{succ}(i) = \{i\} \wedge i \notin F, \\ \left(\bigwedge_{i \xrightarrow{e} i \in E} e \right)^\circledast & \text{if } \text{succ}(i) = \{i\} \wedge i \in F, \\ \bigwedge_{i \xrightarrow{e} j \in E, i \neq j} e \circ \llbracket j \rrbracket_G & \text{if } i \notin \text{succ}(i) \wedge \text{succ}(i) \not\subseteq \{i\}, \\ \left(\bigwedge_{i \xrightarrow{e} i \in E} e \right)^\circledast \circ \left(\bigwedge_{i \xrightarrow{e} j \in E, i \neq j} e \circ \llbracket j \rrbracket_G \right) & \text{if } i \in \text{succ}(i) \wedge \text{succ}(i) \not\subseteq \{i\}. \end{cases}$$

4.12 Related work on the semantics of MSC

4.12.1 Petri-net semantics

In [GRG93] a translation of MSC92 into labelled occurrence nets is provided. Basically, for each event occurring in the MSC a transition is included in the labelled occurrence net. Two subsequent events on an instance are connected by means of a place. These places are labelled with the name of the instance of the two events they connect.

Two corresponding message events are also connected by means of a place. These places are labelled by the message name, the sender instance name and the receiver instance name.

The transformation of an MSC into a labelled occurrence net is straightforward for the language elements of MSC92. It is not so clear if an update of this semantics to the structural concepts of MSC96 is feasible. First, with MSC96 infinite behavior can be described and this is impossible with labelled occurrence nets. The definition of the alternative composition mechanism of MSC (delayed choice) on the level of Petri nets is a complex task. Causal orderings, vertical and horizontal composition should not pose any problems on a Petri-net semantics.

4.12.2 Büchi automata semantics

Ladkin and Leue [LL92b, LL92c, LL94, LL95b] present a semantics of Message Sequence Chart in the following way. Informally, a Message Sequence Chart is translated into a next-event/signal graph, i.e. a graph with different types of edges for commu-

nication and ordering on an instance. This ne/sig graph is then translated into a global state transition graph. The global state transition graph acts as the transition graph of a Büchi automaton [Tho90]. Büchi automata can be defined with different acceptance criteria which are related to various reliability assumptions for communication. Ladkin and Leue state that the MSC language is underspecified in the sense that these reliability properties of communication are not defined explicitly. In our opinion this is not the case. It is not hard to find clues in the recommendation that point to a completely reliable communication mechanism. It is however a recognized problem that the reliability assumptions in MSC cannot be specified by the user.

In [LL92a, LL95a], Ladkin and Leue describe the possibility to add temporal logic formulas [MP91] to MSC specifications to express safety and liveness properties. They propose to replace the description of reliability assumptions with respect to communication by means of Büchi acceptance criteria by temporal logic formulas.

Also in [LL95a], they present four issues concerning the semantics of Message Sequence Chart. In essence, these issues relate to their claim that a Message Sequence Chart must be finite state. The motivation of this claim however only refers to very low-level problems that can be expected when implementing an MSC. In our opinion the language MSC is not suited for the description of systems at a low level of abstraction. The semantics of a language which is to be used at a high level of abstraction should not be motivated by implementation issues on a low level of abstraction.

4.12.3 Process algebra approach

De Man [Man93] uses a process algebra approach towards the semantics of Message Sequence Charts. First, a textual representation for MSC is defined to serve as the basis for the formal semantics. This textual syntax consists of events and event prefixing (;) and it can be considered a more abstract version of the event-oriented textual syntax of MSC. Operators for alternative composition ($\{\}$), sequential composition (\gg), disruption ($\{>$), parallel composition ($\|\|$) and repetitive behavior (*loop*) are defined for MSCs.

The constant ε which represents the empty sequence and the constants that represent the events of an MSC together with event prefixing and alternative composition form a kernel language. The semantics of the other operators is defined in terms of translations of those into the kernel language.

The author does not provide a model for his algebra, nor does he define when two MSCs are equivalent.

In [MR94a], Mauw and Reniers present a formal semantics of Basic Message Sequence Charts based on the process algebra ACP_ε [BW90]. The sequential composition operator \cdot is used to describe the ordering of events on an instance. The parallel composition operator $\|\|$ is used to describe the horizontal composition of the instances. The ordering of the output of a message before its corresponding input is described

by using the state operator λ_S . So, if the semantics of the instances of an MSC is given by the terms I_1, \dots, I_n , then the semantics of the complete MSC is given by

$$\lambda_{\emptyset}(I_1 \parallel \dots \parallel I_n).$$

The operator λ_S collects in S the output events it has encountered and it only allows the execution of an input event if the corresponding output event has been encountered before. In this thesis the operators \cdot , \parallel and λ_S have been combined into \circ^S and \parallel^S respectively.

In [MR97b], the operational rules for generalized delayed parallel composition \parallel^S and generalized weak sequential composition \circ^S are given and the approach to the semantics as used in this thesis is explained.

In [MR97a], the authors present a semantics for High-Level Message Sequence Charts based on recursive equations. However, it turns out that this is not the intended semantics. The definition of the permission relation on recursive equations is extremely difficult and no solution is found there yet. Another difference with the approach in this thesis is that the operator used for horizontal composition there is the normal interleaving merge of *ACP*. This operator does not maintain determinism as we would like.

4.12.4 Partial order semantics

In [AHP96], the authors argue that the semantics of an MSC depends on the communication architecture of the system described. Besides the visual order ($<$) specified by the MSC, which corresponds to the standardized interpretation, they introduce an enforced order (\ll) and an inferred order (\sqsubset). The enforced order contains all event pairs that are guaranteed to occur in the order specified by the communication architecture. It maintains the ordering of corresponding message output and input events. The inferred order contains pairs of events that are likely to be assumed by the user to occur in that order. Different semantic interpretations correspond to different ways of obtaining the enforced and inferred order from the visual order.

Although the authors are right that MSC is used with different communication architectures in mind, we are reluctant to vary the interpretation of an MSC. The reason for our reluctance is that using the same diagrams with different interpretations can be confusing for the user. Therefore, at least, a way should be found to indicate the communication model inside the MSC.

A different, but nevertheless related, approach was followed in [EMR97]. There it is defined if an MSC, with the visual order interpretation, can be realized with a given communication architecture. Also a hierarchy is presented of realizability in different communication models.

5

Concluding remarks

In this thesis we have given a detailed introduction of the ITU-standardized language MSC. The graphical syntax as well as the event-oriented textual syntax have been explained in Chapter 2. This chapter also contains an informal explanation of the meaning of an MSC. Necessarily, the contents of this chapter are based on recommendation Z.120 [IT96b], which contains the definition of the language MSC.

In the following chapter, Chapter 3, a number of constants and operators is introduced by means of operational rules in the style of Plotkin. Some properties of the operators are given. These operators are related directly to the composition constructs used in the language MSC.

In Chapter 4, a mapping of an MSC document in event-oriented textual syntax into a process term is defined. This chapter thus contains the formal semantics definition of the language MSC. The formal semantics definition covers almost completely the language as it is standardised by the ITU [IT96b]. We did not associate a semantics with the combination of timer events and we did not consider messages and orderings from the environment to the environment.

It is hard to make remarks on the correctness of this formal semantics. In general, such a formal semantics is the basic step from an informal representation of the semantics to a formal description thereof. However, it is possible to increase the confidence in the correctness of the formal semantics. One way of doing so is relating the formal semantics definition to another formal semantics definition. This requires that the mathematical frameworks that are used for defining the formal semantics definitions can be compared. This in itself is already a topic worthwhile studying. Another way of increasing confidence is through the consensus of a group of persons that are considered experts on MSC. This is the kind of consensus that is obtained for the semantics of MSC as presented in this thesis. Although this is not a mathematically convincing argument for the correctness of the formal semantics, it is the second best option available.

The believe in the correctness of the formal semantics can be increased further by showing that some properties that were already expected by the same experts indeed hold. In the case of MSC this could be transformation rules for HMSCs or basic properties of the operators such as commutativity and associativity of delayed choice.

The formal semantics defined in this thesis can be used for several purposes. Once a formal semantics has been defined and agreed upon, it can act as a reference manual for the meaning of MSCs. If two parties disagree on the meaning of an MSC, the formal semantics definition can be inspected to determine the standardised meaning of the MSC.

The formal semantics definition can be used to check if the functionality offered by tools is consistent with the standardised semantics. For example, it can be used to check if the execution sequences of a simulator are also given by the formal semantics.

The formal semantics associates a process term with an MSC in event-oriented textual representation. For this process term an operational semantics is defined. Implementation of the transformation of an MSC in textual representation to a process term and implementation of the operational semantics lead to the ingredients that are needed to build a simulator. An experiment in this direction for Basic Message Sequence Charts has been performed using the tool ASF+SDF [MvdM95]. This experiment was based on the instance-oriented textual syntax of BMSC and the process algebra semantics of Mauw and Reniers [MR94a]. It is expected that the development of a similar prototype tool based on the event-oriented syntax and the formal semantics presented in this thesis is feasible as well. Currently, experiments in this direction take place at Eindhoven University of Technology.

Via the operational semantics, a transformation of an MSC into a transition system is defined. Such a transition system can be considered a first step in the direction of applying model checking techniques on MSCs. An important aspect there is the development of a language for expressing properties of an MSC. A first initiative in this direction is [CPRO95] where the MSC language itself is used for the specification of properties.

Using some theorems presented in Chapter 3, transformations can be defined on MSCs. In Figure 5.1 a number of such transformations is given for MSC reference nodes with textual formula into multiple MSC reference nodes with simpler textual formula. With the formal semantics definition it is possible to formally prove that these transformations preserve the semantics of the MSC.

Other transformations that can be based on the properties of the operators are the following:

- Adding and removing connection nodes (see Figure 5.2). Any arrow between two nodes in an HMSC can be replaced by two arrows between these nodes by using a fresh connection node. This transformation is based on the fact that the empty process is a unit for weak sequential composition (see Theorem 3.6.7). A consequence of this transformation is that it is possible to change an HMSC

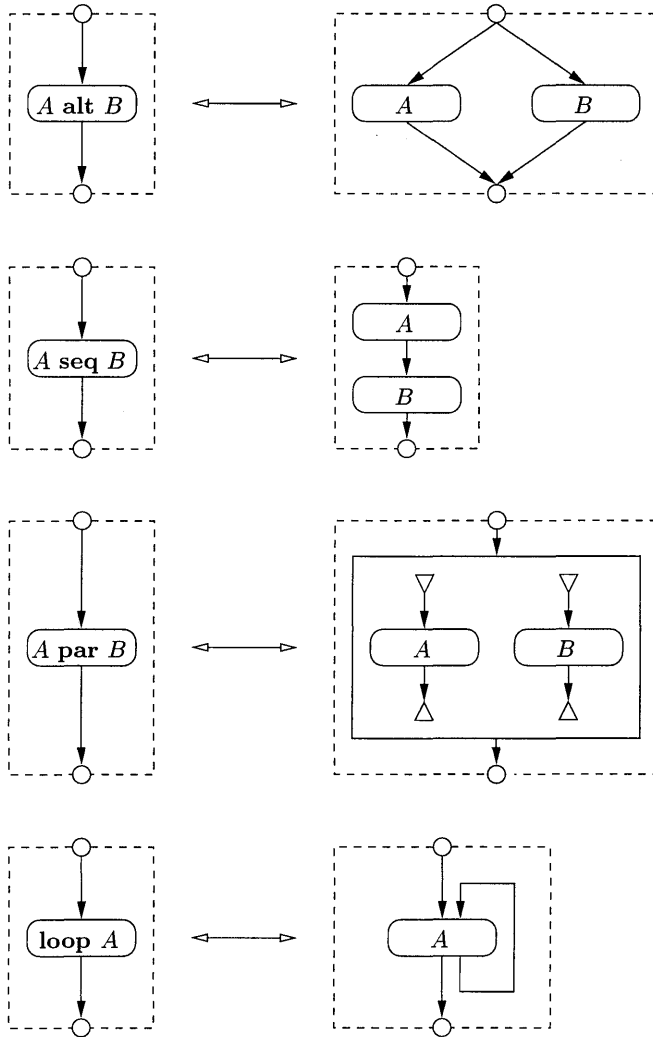


Figure 5.1: Transformations replacing textual formula by HMSC nodes.

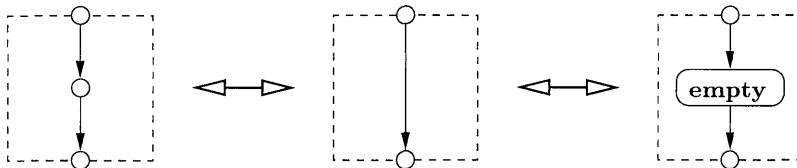


Figure 5.2: Adding and removing connection nodes.

into an equivalent HMSC where every non-connection node has exactly one incoming arrow (except the start node) and exactly one outgoing arrow (except the end nodes). Instead of adding or removing a connection node also an MSC reference node with the textual formula **empty** can be used as the semantics of such a textual formula is the same as the semantics of a connection node. More transformation based on this law are given in Figure 5.3.

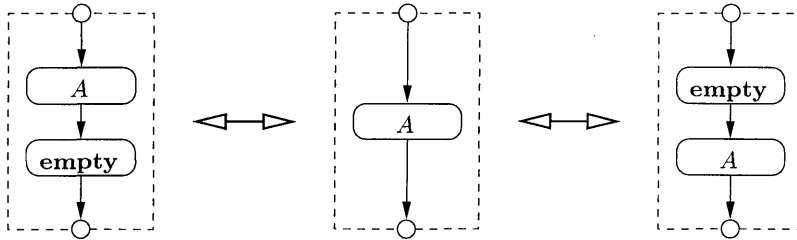


Figure 5.3: Transformations due to unit element.

- Idempotency of delayed choice (Theorem 3.4.4). The fact that delayed choice is idempotent leads to transformations as sketched in Figure 5.4.

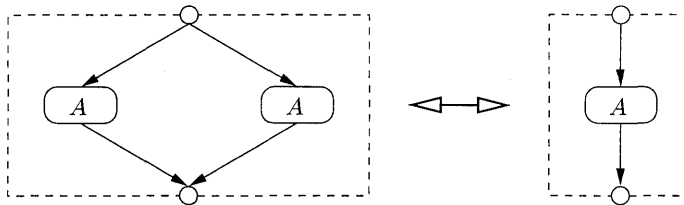


Figure 5.4: Transformation due to idempotency of delayed choice.

- Distributivity of delayed parallel composition over delayed choice (see Theorem 3.5.4). This leads to transformations as sketched in Figure 5.5.
- Distributivity of weak sequential composition over delayed choice (see Theorem 3.6.7). This leads to transformations as sketched in Figure 5.6.

As MSC is often used in combination with other formalisms, it is interesting to study the relation between formalisms. Having a formal semantics is a prerequisite for such a study. Also, the description of transformations between formalisms benefits from a formal basis.

In the use of the graphical and textual syntax of High-level Message Sequence Charts implicitly a number of properties of the composition mechanisms are assumed to hold.

A node can have any positive, finite number of outgoing arrows. These denote alternative continuations. Both graphically and textually there is no means of grouping these

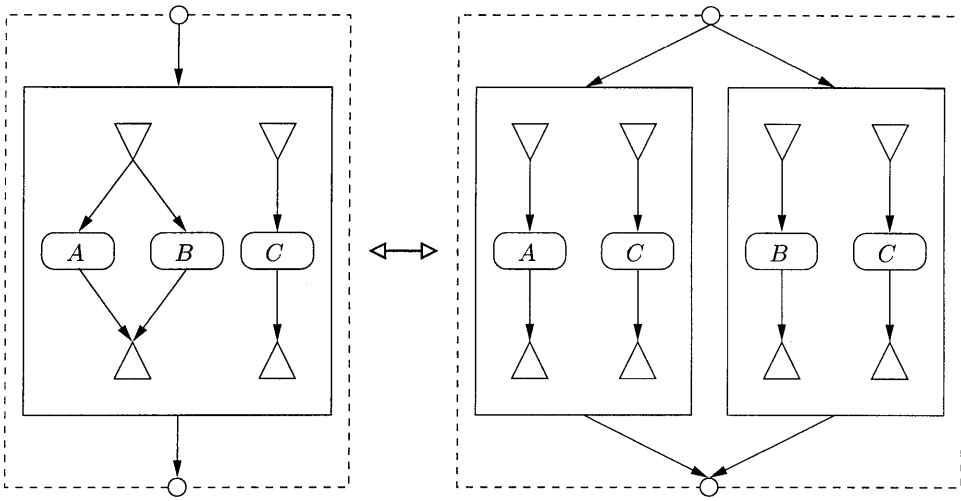


Figure 5.5: Transformation due to distributivity of \parallel over \mp .

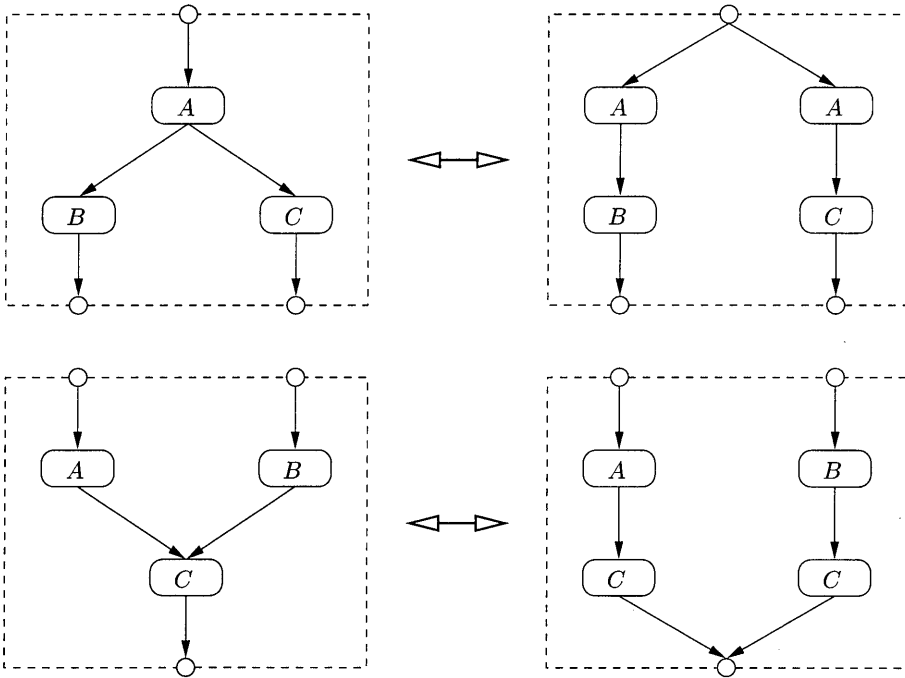


Figure 5.6: Transformations due to distributivity of \circ^S over \mp .

alternatives by means of parentheses. Thus, either there must be an implicit way of grouping those or the grouping does not make a difference. As the recommendation states nothing about the first option, it is assumed that the grouping of alternatives does not make a difference. In order for the semantics to be well-defined it thus has to be the case that delayed choice (the operator used for describing alternatives) is associative. This is stated in Theorem 3.5.4.

A similar situation applies to the use of the parallel frame in HMSCs. It can contain any positive, finite number of anonymous HMSCs and again no means of grouping those is provided. Thus, it is required that delayed parallel composition is associative. This is also stated in Theorem 3.5.4.

An HMSC appears as a graph with several types of labelled nodes. The user expects that HMSCs which are isomorphic (in the sense that their underlying graphs are isomorphic), are considered equivalent. This is indeed the case for the semantics associated with HMSC in this thesis. For this it is important that we have the following properties for the operators used in the semantics of HMSCs.

- Delayed choice is commutative. Without commutativity of delayed choice the HMSCs that are represented textually by

<pre> msc <i>example</i> ; expr <i>l1</i> alt <i>l2</i> ; <i>l1</i> : <i>A</i> seq (<i>e</i>) ; <i>l2</i> : <i>B</i> seq (<i>e</i>) ; <i>e</i> : end ; endmsc ; </pre>	and	<pre> msc <i>example</i> ; expr <i>l2</i> alt <i>l1</i> ; <i>l1</i> : <i>A</i> seq (<i>e</i>) ; <i>l2</i> : <i>B</i> seq (<i>e</i>) ; <i>e</i> : end ; endmsc ; </pre>
---	-----	---

would not necessarily be equivalent although their graphical representations are isomorphic.

- Delayed choice is associative.
- Delayed parallel composition is commutative.
- Delayed parallel composition is associative.

Bibliography

- [AB95] M. Andersson and J. Bergstrand. Formalizing Use Cases with Message Sequence Charts. Master's thesis, Lund Institute of Technology, 1995.
- [AHP96] R. Alur, G.J. Holzmann, and D. Peled. An analyzer for Message Sequence Charts. *Software - Concepts and Tools*, 17(2):70–77, 1996.
- [AMPV97] E. Algaba, M. Monedero, E. Pérez, and O. Valcárel. HARPO: Testing tools development. In M. Kim, S. Kang, and K. Hong, editors, *Testing of Communicating Systems*, IFIP TC6 Tenth International Workshop on Testing of Communicating Systems, pages 318–323, Cheju Island, Korea, September 1997. Chapman & Hall.
- [AN95] G. Amsjø and A. Nyeng. SDL-based software development in Siemens A/S – experience of introducing rigorous use of SDL and MSC. In R. Bræk and A. Sarma, editors, *SDL'95 - with MSC in CASE*, Proceedings of the Seventh SDL Forum, pages 339–348, Oslo, 1995. Amsterdam, North-Holland.
- [Bel92] F. Belina. SDL Methodology Guidelines. Technical report, CCITT, May 1992.
- [BHS91] F. Belina, D. Hogrefe, and A. Sarma. *SDL - with applications from protocol specification*. The BCS Practitioners Series. Prentice-Hall International, London/Englewood Cliffs, 1991.
- [BJR96] G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language for Object-Oriented Development*. RATIONAL Software Corporation, 1996.
- [BK84] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1/3):109–137, 1984.
- [BL97] H. Ben-Abdallah and S. Leue. Timing constraints in Message Sequence Chart specifications. In T. Mizuno, N. Shiratori, T. Higashino, and A. Togashi, editors, *Formal Description Techniques and Protocol Specification, Testing and Verification*, Proceedings of FORTE X and PSTV XVII '97, pages 91–106, Osaka, 1997. Chapman & Hall.

- [BM95] J.C.M. Baeten and S. Mauw. Delayed choice: an operator for joining Message Sequence Charts. In D. Hogrefe and S. Leue, editors, *Formal Description Techniques VII*, Proceedings of the Seventh IFIP WG 6.1 International Conference on Formal Description Techniques, pages 340–354. Berne, Chapman & Hall, 1995.
- [Bro85] M. Broy. Extensional behaviour of concurrent, nondeterministic, communicating systems. In M. Broy, editor, *Control flow and data flow: concepts of distributed programming*, volume F14 of *NATO ASI series*, pages 229–276. Springer-Verlag, 1985.
- [BV95] J.C.M. Baeten and C. Verhoef. Concrete process algebra. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Semantic Modelling*, volume 4 of *Handbook of Logic in Computer Science*, pages 149–268. Oxford University Press, 1995.
- [BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.
- [CCHvK90] A.A.R. Cockburn, W. Citrin, R.F. Hauser, and J. von Känel. An environment for interactive design of communication architectures. In L. Logrippo, R.L. Probert, and H. Ural, editors, *Protocol Specification, Testing and Verification*, volume 10 of *Proc. IFIP WG 6.1 Tenth International Symposium*, pages 115–127, Ottawa, 1990. North-Holland.
- [CCI88a] CCITT. *Recommendation Q.699: Interworking between the Digital Subscriber Layer 3 Protocol and the Signalling System No. 7 ISDN User Part CCITT*. CCITT, Geneva, 1988.
- [CCI88b] CCITT. *Recommendation X.210: Open System Interconnection layer service definition conventions*. Blue Book FASCICLE VIII.4, Recommendations X.200 - X.219. CCITT, November 1988.
- [CCI88c] CCITT. *Recommendation Z.100: Specification and Description Language (SDL) - Annex D: SDL User Guidelines*. CCITT, Geneva, 1988.
- [CCI89] CCITT. *Recommendation Q.65: Stage 2 of the Method for the Characterization of Services Supported by an ISDN*. CCITT, Geneva, 1989.
- [CLM97] A. Cavalli, B. Lee, and T. Macavei. Test generation for the SSCOP-ATM networks protocol. In A. Cavalli and A. Sarma, editors, *SDL'97: Time for Testing - SDL, MSC and Trends*, Proceedings of the Eighth SDL Forum, pages 277–288, Evry, 1997. Amsterdam, North-Holland.
- [CPRO95] P. Combes, S. Pickin, B. Renard, and F. Olsen. MSCs to express service requirements as properties on a SDL model: Application to service interaction detection. In R. Bræk and A. Sarma, editors, *SDL'95 - with MSC in CASE*, Proceedings of the Seventh SDL Forum, pages 243–256, Oslo, 1995. Amsterdam, North-Holland.

- [EFM97] A. Engels, L.M.G. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In E. Brinksma, editor, *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1217 of *Lecture Notes in Computer Science*, pages 384–398. Springer-Verlag, 1997.
- [Ek93] A. Ek. Verifying Message Sequence Charts with the SDT validator. In O. Færgemand and A. Sarma, editors, *SDL'93 - Using Objects*, Proceedings of the Sixth SDL Forum, pages 237–249, Darmstadt, 1993. Amsterdam, North-Holland.
- [Ek94] A. Ek. Event-oriented textual syntax. Technical Report TD 44, ITU-T Meeting Study Group 10, October 1994.
- [EMR97] A. Engels, S. Mauw, and M.A. Reniers. A hierarchy of communication models for Message Sequence Charts. In T. Mizuno, N. Shiratori, T. Higashino, and A. Togashi, editors, *Formal Description Techniques and Protocol Specification, Testing and Verification*, Proceedings of FORTE X and PSTV XVII '97, pages 75–90, Osaka, 1997. Chapman & Hall.
- [Eng85] J. Engelfriet. Determinacy \rightarrow (observation equivalence = trace equivalence). *Theoretical Computer Science*, 36(1):21–25, 1985.
- [Fac95a] C. Facchi. Formal semantics of Time Sequence Diagrams. Technical Report TUM-I9540, Institut für Informatik, Technische Universität München, 1995.
- [Fac95b] C. Facchi. *Methodik zur formalen Spezifikation des ISO/OSI Schichtenmodells*. PhD thesis, Technische Universität München, 1995.
- [Fei97] L.M.G. Feijs. Synchronous sequence charts in action. *Information and Software Technology*, 39:583–606, 1997.
- [FJ96] L.M.G. Feijs and M. Jumelet. A rigorous and practical approach to service testing. In B. Baumgarten, H. Burkhardt, and A. Giessler, editors, *Testing of Communicating Systems*, IFIP TC6 Ninth International Workshop on Testing of Communicating Systems, pages 175–190. Chapman & Hall, 1996.
- [FJM94] L.M.G. Feijs, H.B.M. Jonkers, and C.A. Middelburg. *Notations for Software Design*. Formal approaches to computing and information technology. Springer-Verlag, 1994.
- [FMMvW98] L.M.G. Feijs, F.A.C. Meijs, J.R. Moonen, and J.J. van Wamel. Conformance testing of a multimedia chip using PHACT. In A. Petrenko and N. Yevtushenko, editors, *Testing of Communicating Systems*, pages 193–210, 1998.
- [FO94] O. Færgemand and A. Olsen. Introduction to SDL'92. *Computer Networks and ISDN Systems*, 26(9):1143–1167, 1994.

- [GHN93] J. Grabowski, D. Hogrefe, and R. 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, pages 253–265, Darmstadt, 1993. Amsterdam, North-Holland.
- [GHNS95] J. Grabowski, D. Hogrefe, I. Nussbaumer, and A. Spichiger. Test case specification based on MSCs and ASN.1. In R. Bræk and A. Sarma, editors, *SDL'95 - with MSC in CASE*, Proceedings of the Seventh SDL Forum, pages 307–322, Oslo, 1995. Amsterdam, North-Holland.
- [GR89] J. Grabowski and E. Rudolph. Putting extended sequence charts to practice. In O. Færgemand and M.M. Marques, editors, *SDL'89 - The Language at Work*, Proceedings of the Fourth SDL Forum, pages 3–10, Lisbon, 1989. Amsterdam, North-Holland.
- [GR92] P. Graubmann and E. Rudolph. Comments on a Petri Net based Message Sequence Chart Semantics. Technical report, CCITT Interims Meeting, Geneva, 1992.
- [Gra94] J. Grabowski. *Test Case Generation and Test Case Specification with Message Sequence Charts*. PhD thesis, Universität Bern, 1994.
- [GRG93] P. Graubmann, E. Rudolph, and J. 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, Darmstadt, 1993. Amsterdam, North-Holland.
- [Gro90] J.F. Groote. Transition system specifications with negative premises. In J.C.M. Baeten and J.W. Klop, editors, *CONCUR'90 - Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 332–341, Amsterdam, 1990. Springer-Verlag.
- [GSDH97] J. Grabowski, R. Scheuer, Z.R. Dai, and D. Hogrefe. Applying SaM-sTaG to the B-ISDN protocol SSCOP. In M. Kim, S. Kang, and K. Hong, editors, *Testing of Communicating Systems*, IFIP TC6 Tenth International Workshop on Testing of Communicating Systems, pages 397–415, Cheju Island, Korea, September 1997. Chapman & Hall.
- [GV92] J.F. Groote and F.W. Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100(2):202–260, 1992.
- [Har87] D. Harel. Statecharts, a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
- [Hau94] Ø. Haugen. MSC structural concepts. Technical Report TD 9006, ITU-T Experts Meeting SG 10, Turin, 1994.

- [Hau95] Ø. Haugen. Using MSC-92 effectively. In R. Bræk and A. Sarma, editors, *SDL'95 - with MSC in CASE*, Proceedings of the Seventh SDL Forum, pages 37–49, Oslo, 1995. Amsterdam, North-Holland.
- [HBM93] Ø Haugen, R. Bræk, and G. Melby. The SISU project. In O. Færgemand and A. Sarma, editors, *SDL'93 - Using Objects*, Proceedings of the Sixth SDL Forum, pages 479–489, Darmstadt, 1993. Amsterdam, North-Holland.
- [Hee94] K.M. van Hee. *Information Systems Engineering: A Formal Approach*. Cambridge University Press, Cambridge, 1994.
- [Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall International, 1985.
- [Hol91] G.J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall International, 1991.
- [ISO87] ISO. *Information processing systems – Open Systems Interconnection – service conventions*. ISO, 1987.
- [ISO91a] ISO. *ISO/IEC JTC 1/SC 21: Information Technology - Open Systems Interconnection - Conformance Testing Methodology and Framework - Part 3: The Tree and Tabular Combined Notation, IS 9646-3*. ISO, Geneva, 1991.
- [ISO91b] ISO. *Revised Text of CD 10731, Information Technology - Open Systems Interconnection - Conventions for the Definition of OSI Service Conventions, ISO/IEC JTC 1/SC 21 N 6341*, 1991.
- [IT88] ITU-T. *Recommendation X.208: Specification of Abstract Syntax Notation One (ASN.1)*. ITU-T, Geneva, 1988.
- [IT93] ITU-T. *Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-T, Geneva, September 1993.
- [IT94] ITU-T. *Recommendation Z.100: Specification and Description Language (SDL)*. ITU-T, Geneva, June 1994.
- [IT95] ITU-T. *Recommendation Z.120 Annex B: Algebraic semantics of Message Sequence Charts*. ITU-T, Geneva, April 1995.
- [IT96a] ITU-T. *Recommendation Z.120 Annex C: Static semantics of Message Sequence Charts*. ITU-T, Geneva, October 1996.
- [IT96b] ITU-T. *Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-T, Geneva, October 1996.
- [IT96c] ITU-T. *Recommendation Z.200: CCITT High Level Language (CHILL)*. ITU-T, Geneva, October 1996.

- [IT97] ITU-T. *Recommendation Z.500: Framework on formal methods in conformance testing*. ITU-T, Geneva, May 1997.
- [JCJO92] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison-Wesley, Reading, 1992.
- [Jen92] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1992.
- [Kel76] R.M. Keller. Formal verification of parallel programs. *Journal of the ACM*, 19(7):371–384, 1976.
- [Koy92] R. Koymans. *Specifying Message Passing and Time-Critical Systems with Temporal Logic*, volume 651 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1992.
- [LL92a] P.B. Ladkin and S. Leue. An analysis of Message Sequence Charts. Technical Report IAM-92-013, University of Berne, 1992.
- [LL92b] P.B. Ladkin and S. Leue. An automaton interpretation of Message Sequence Charts. Technical Report IAM-92-012, University of Berne, 1992.
- [LL92c] P.B. Ladkin and S. Leue. Interpreting Message Sequence Charts. Technical Report IBM RJ 8965, IBM Almaden Research Center, San Jose, 1992.
- [LL94] P.B. Ladkin and S. Leue. What do Message Sequence Charts mean? In R.L. Tenney, P.D. Amer, and M.Ü. Uyar, editors, *Formal Description Techniques VI*, IFIP Transactions C, Proceedings Sixth International Conference on Formal Description Techniques, pages 301–316, Boston, 1994. Amsterdam, North-Holland.
- [LL95a] P.B. Ladkin and S. Leue. Four issues concerning the semantics of Message Flow Graphs. In D. Hogrefe and S. Leue, editors, *Formal Description Techniques VII*, Proceedings of the Seventh IFIP WG 6.1 International Conference on Formal Description Techniques, pages 355–369. Berne, Chapman & Hall, 1995.
- [LL95b] P.B. Ladkin and S. Leue. Interpreting message flow graphs. *Formal Aspects of Computing*, 7(5):473–509, 1995.
- [LRH97] S. Loidl, E. Rudolph, and U. Hinkel. MSC'96 and beyond - a critical look. In A. Cavalli and A. Sarma, editors, *SDL'97 : Time for Testing - SDL, MSC and Trends*, Proceedings of the Eighth SDL Forum, pages 213–227, Evry, 1997. Amsterdam, North-Holland.
- [Lyn96] N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

- [Man93] J. de Man. Towards a formal semantics of Message Sequence Charts. In O. Færgemand and A. Sarma, editors, *SDL'93 - Using Objects*, Proceedings of the Sixth SDL Forum, pages 157–165, Darmstadt, 1993. Amsterdam, North-Holland.
- [Mau96] S. Mauw. The formalization of Message Sequence Charts. *Computer Networks and ISDN Systems*, 28(12):1643–1657, 1996. Special issue on SDL and MSC, guest editor Ø. Haugen.
- [Maz88] A. Mazurkiewicz. Basic notions of trace theory. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear time, branching time and partial orders in logics and models for concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 285–363. Springer-Verlag, 1988.
- [Mei96] F.A.Ch. Meijs. Message Sequence Chart Enhancements. Technical Report RWB-506-ir-95071, Information and Software Technology, Philips Research, 1996.
- [Men93] N. Meng-Siew. Reasoning with timing constraints in Message Sequence Charts. Master's thesis, University of Stirling, 1993.
- [Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.
- [Mil89] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice-Hall International, 1989.
- [MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.
- [MR94a] S. Mauw and M.A. Reniers. An algebraic semantics of Basic Message Sequence Charts. *The Computer Journal*, 37(4):269–277, 1994.
- [MR94b] S. Mauw and M.A. Reniers. An algebraic semantics of Message Sequence Charts. Technical Report TD 9009, ITU-T Experts Meeting, Turin, 1994.
- [MR95] S. Mauw and M.A. Reniers. Thoughts on the meaning of conditions. Technical Report TD 9016, ITU-T Experts Meeting SG 10, St. Petersburg, April 1995.
- [MR96] S. Mauw and M.A. Reniers. Refinement in Interworkings. In U. Montanari and V. Sassone, editors, *CONCUR'96, Proceedings of the Seventh Conference on Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 671–686, Pisa, Italy, 1996. Springer-Verlag.
- [MR97a] S. Mauw and M.A. Reniers. High-level Message Sequence Charts. In A. Cavalli and A. Sarma, editors, *SDL'97 : Time for Testing - SDL, MSC and Trends*, Proceedings of the Eighth SDL Forum, pages 291–306, Evry, 1997. Amsterdam, North-Holland.

- [MR97b] S. Mauw and M.A. Reniers. Operational semantics for MSC'96. In A. Cavalli and D. Vincent, editors, *Tutorials of the Eighth SDL Forum SDL'97: Time for Testing - SDL, MSC and Trends*, pages 135–152, Evry, 1997. Amsterdam, Institut national des télécommunications. To appear in *Computer Networks and ISDN Systems*, June 1999.
- [MvdM95] S. Mauw and E.A. van der Meulen. Generating tools for Message Sequence Charts. In R. Bræk and A. Sarma, editors, *SDL'95 - with MSC in CASE*, Proceedings of the Seventh SDL Forum, pages 51–62, Oslo, 1995. Amsterdam, North-Holland.
- [MvWW92] S. Mauw, M. van Wijk, and T. Winter. Syntax and semantics of synchronous Interworkings. Technical Report RWB-508-re-92436, Information and Software Technology, Philips Research, 1992.
- [MvWW93] S. Mauw, M. van Wijk, and T. Winter. A formal semantics of synchronous Interworkings. In O. Færgemand and A. Sarma, editors, *SDL'93 - Using Objects*, Proceedings of the Sixth SDL Forum, pages 167–178, Darmstadt, 1993. Amsterdam, North-Holland.
- [MW93] S. Mauw and T. Winter. A prototype toolset for Interworkings. *Philips Telecommunication Review*, 51(3):41–45, 1993.
- [Nah91] R. Nahm. Consistency analysis of Message Sequence Charts and SDL-systems. In O. Færgemand and R. Reed, editors, *SDL'91 - Evolving Methods*, Proceedings of the Fifth SDL Forum, pages 261–271. Amsterdam, North-Holland, 1991.
- [Nah94] R. Nahm. *Conformance Testing Based on Formal Description Techniques and Message Sequence Charts*. PhD thesis, Universität Bern, 1994.
- [NS73] I. Nassi and B. Shneidermann. Flowchart techniques for structured programming. *ACM SIGPLAN Notices*, 8(8):12–16, 1973.
- [PAM97] E. Pérez, E. Algaba, and M. Monedero. A pragmatic approach to test generation. In M. Kim, S. Kang, and K. Hong, editors, *Testing of Communicating Systems*, IFIP TC6 Tenth International Workshop on Testing of Communicating Systems, pages 365–380, Cheju Island, Korea, September 1997. Chapman & Hall.
- [Par81] D.M.R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proceedings 5th GI Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1981.
- [Plo81] G.D. Plotkin. A structural approach to operational semantics. Technical Report DIAMI FN-19, Computer Science Department, Aarhus University, 1981.
- [Rei85] W. Reisig. *Petri Nets: An Introduction*, volume 4 of *EATCS Monographs in Theoretical Computer Science*. Springer-Verlag, Berlin, 1985.

- [Ren95a] M. A. Reniers. Syntax requirements of Message Sequence Charts. In R. Bræk and A. Sarma, editors, *SDL'95 - with MSC in CASE*, Proceedings of the Seventh SDL Forum, pages 63–74, Oslo, 1995. Amsterdam, North-Holland.
- [Ren95b] M.A. Reniers. Syntax requirements of Message Sequence Charts. Technical Report TD GEN/10-23, ITU-T Joint Rapporteurs Meeting, September 1995.
- [Ren96a] M.A. Reniers. Evaluation of the SDT validator. Technical Report RWB-510-ir-95048, Information and Software Technology, Philips Research, 1996.
- [Ren96b] M.A. Reniers. Static semantics of Message Sequence Charts. Technical Report CSR 96-19, Eindhoven University of Technology, Department of Computing Science, 1996.
- [RGG95] E. Rudolph, P. Graubmann, and J. Grabowski. Message Sequence Chart: composition techniques versus OO-techniques - 'tema con variazioni'. In R. Bræk and A. Sarma, editors, *SDL'95 - with MSC in CASE*, Proceedings of the Seventh SDL Forum, pages 77–88, Oslo, 1995. Amsterdam, North-Holland.
- [RGG96] E. Rudolph, P. Graubmann, and J. Grabowski. Tutorial on Message Sequence Charts. *Computer Networks and ISDN Systems*, 28(12):1629–1641, 1996. Special issue on SDL and MSC, guest editor Ø. Haugen.
- [Rud95] E. Rudolph. MSC roadmaps. Towards a synthesized solution. Technical Report TD 9017, Experts Meeting ITU-T SG 10, Geneva, 1995.
- [RW94] A. Rensink and H. Wehrheim. Weak sequential composition in process algebras. In B. Jonsson and J. Parrow, editors, *CONCUR'94: Concurrency Theory*, volume 836 of *Lecture Notes in Computer Science*, pages 226–241, Uppsala, 1994. Springer-Verlag.
- [Sch95] Chr. Schaffer. MSC/RT: A real-time extension to Message Sequence Charts (MSCs). Technical Report 129-95, Johannes Kepler University, Department of Systems Theory and Information Engineering, Linz, 1995.
- [Sie92] Siemens AG (Germany). *Comments on Recommendation Z.120 Message Sequence Chart (MSC)*. Geneva, November 1992.
- [SST97] S. Shiba, Y. Shigeta, and W. Tanaka. Switching software test environment using MSC. In A. Cavalli and A. Sarma, editors, *SDL'97: Time for Testing - SDL, MSC and Trends*, Proceedings of the Eighth SDL Forum, pages 183–196, Evry, 1997. Amsterdam, North-Holland.
- [Tak93] B. Takacs. Use of SDL in an Object Oriented Design Process during the development of a prototype switching system. In O. Færgemand

- and A. Sarma, editors, *SDL'93 - Using Objects*, Proceedings of the Sixth SDL Forum, pages 79–88, Darmstadt, 1993. Amsterdam, North-Holland.
- [Tel96] Telelogic AB. *SDT 3.1 Reference Manual*. Malmö, Sweden, 1996.
- [TGH95] D. Toggweiler, J. Grabowski, and D. Hogrefe. Partial order simulation of SDL specifications. In R. Bræk and A. Sarma, editors, *SDL'95 - with MSC in CASE*, Proceedings of the Seventh SDL Forum, pages 293–306, Oslo, 1995. Amsterdam, North-Holland.
- [Tho90] A. Thomas. Automata on onfinite objects. In J. van Leeuwen, editor, *Formal models and semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 4, pages 132–191. Elsevier Science Publishers B.V., 1990.
- [Tog95] D. Toggweiler. *Efficient Test Generation for Distributed Systems specified by Automata*. PhD thesis, University of Berne, 1995.
- [Ver95] C. Verhoef. A congruence theorem for structured operational semantics with predicates and negative premises. *Nordic Journal of Computing*, 2(2):274–302, 1995.
- [Ver96] Verilog. *ObjectGEODE Toolset Documentation*, 1996.
- [Win87] G. Winskel. Event structures. In Brauer, Reissig, and G. Rozenberg, editors, *Petri nets: Applications and relationships to other models of concurrency*, volume 255 of *Lecture Notes in Computer Science*, pages 325–392. Springer-Verlag, 1987.

Appendix A

Textual syntax of MSC for the semantics

In this appendix we present the textual syntax that has actually been used for the definition of the formal semantics. In Section A.2 the textual syntax is given and in Section A.1 the changes that have led to this textual syntax are explained.

A.1 Changes to the textual syntax

The textual syntax of MSC as presented in recommendation Z.120 is changed in several aspects for the definition of the formal semantics. These changes can be subdivided into several categories. In Section A.1.1, we explain the changes to the textual syntax due to the fact that certain concepts are not treated in the formal semantics in this thesis. In Section A.1.2, we optimize the textual syntax by removing irrelevant information. In Section A.1.3, we explain the optimization of the textual syntax by considering certain constructions as abbreviations of other constructions. In Section A.1.4, we extend the textual syntax for the purpose of defining the formal semantics. In Section A.1.5, we explain the assumptions that have led to a further simplification of the textual syntax.

Besides the changes presented in the following sections also reformulations of the BNF rules have taken place in order to facilitate the definition of the formal semantics. These reformulations are replacing a nonterminal in the right-hand sides of BNF rules by its productions, reformulating a BNF rule such that it facilitates inductive definitions and the introduction of new nonterminals to facilitate definitions. All changes explained below are given with respect to the textual syntax of MSC as presented in recommendation Z.120.

A.1.1 Parts of the language that are not treated

Instance-oriented representation

The textual syntax of MSC offers the possibility to describe an MSC in an instance-oriented way, in an event-oriented way and even by mixing these two description styles. For the definition of the semantics it is assumed that the MSC is represented in an event-oriented way. This restriction has big consequences for the textual syntax that is used for the definition of the formal semantics. These consequences are listed below:

- The MSC statements that are produced by the sequence of nonterminals ⟨old instance head statement⟩ ⟨instance event list⟩ are used to give the user of the language MSC the possibility to describe an instance in isolation. This combination is removed as an alternative for the productions of the nonterminal ⟨msc statement⟩.
- The shared conditions, shared MSC reference expressions and shared inline expressions are only used for the instance-oriented textual syntax and can be omitted as alternative productions in the rule for ⟨non-orderable event⟩.
- As a consequence of the above omissions a number of nonterminals is not necessary anymore. These are removed.

Instance decomposition

In this thesis no semantics is provided for instance decomposition. As a consequence it is not necessary to indicate that an instance is decomposed by means of the productions of the nonterminal ⟨decomposition⟩ in the BNF rule for the nonterminal ⟨instance head statement⟩.

Substitution

In this thesis no semantics is provided for the substitution mechanism in MSC reference expressions. The optional use of the nonterminal ⟨parameter substitution⟩ in the BNF rule for the nonterminal ⟨msc ref loop expr⟩ is therefore removed.

Incomplete message events and gates

In this thesis no semantics is provided for lost and found message events that are sent to or received from the environment. This has several consequences for the textual syntax.

- A lost message event can only be sent to an instance or the environment without a gate name being associated with it. Similarly, a found message event can only be received from an instance or the environment without a gate name being associated with it. Therefore the BNF rules for the nonterminals (incomplete message output) and (incomplete message input) is replaced by the rules

$$\begin{aligned} \langle \text{incomplete message output} \rangle ::= & \text{ out } \langle \text{msg identification} \rangle \\ & \text{ to lost } [\langle \text{instance name} \rangle \mid \text{env}] \\ \langle \text{incomplete message input} \rangle ::= & \text{ in } \langle \text{msg identification} \rangle \\ & \text{ from found } [\langle \text{instance name} \rangle \mid \text{env}] \end{aligned}$$

- As incomplete message events cannot be sent to or received from the environment the nonterminals (output dest) and (input dest) can be simplified to the nonterminals (output address) and (input address) respectively.

A.1.2 Irrelevant information

In the textual syntax of MSC at several places information is provided that is irrelevant for the semantics. For the purpose of defining the semantics of MSC it is assumed that the MSCs do not contain this type of information.

- All parts of the textual syntax that specify comments are removed. The BNF rule for the nonterminal (end) is replaced by the BNF rule

$$\langle \text{end} \rangle ::= \text{ ; }$$

As a consequence all occurrences of the nonterminal (end) are replaced by the terminal ;. Furthermore, the possibility to have a text definition as an MSC statement is removed.

- Graphical parts of the concrete grammar are removed. These are the nonterminals (document head area) and (msc diagram) which occur in the BNF rules for the nonterminals (msc document head) and (msc document body), respectively.
- The part of the MSC document head that contains references to external sources is removed. The BNF rule for the nonterminal (document head) is replaced by the BNF rule

$$\langle \text{document head} \rangle ::= \text{ mscdocument } \langle \text{msc document name} \rangle \text{ ;}$$

- The optional MSC interface ((msc interface)) is removed as it contains no information that is relevant to the definition of the formal semantics. This is only possible due to the extension of the textual syntax with a keyword **after** as explained in Section A.1.4.

- The part of the syntax referring to instance head and end statements (the non-terminals \langle instance head statement \rangle and \langle instance end statement \rangle) is removed. The information which instances are described in the MSC is only used as additional information that is useful when drawing an MSC starting from the textual representation. Also, this information is used to interpret the keyword **all**. We assume that all occurrences of the keyword **all** are replaced by the corresponding list of instance names (see Section A.1.3).

A.1.3 Shorthands

In the textual syntax of MSC at a number of places shorthands can be used in the textual syntax. For the purpose of defining semantics these can be treated as if they were replaced by their unabbreviated representations.

- The textual syntax for event definitions is restricted to contain exactly one instance event or multi instance event.

$$\begin{aligned} \langle \text{event definition} \rangle ::= & \langle \text{instance name} \rangle : \langle \text{instance event} \rangle ; \\ & | \langle \text{instance name list} \rangle : \langle \text{multi instance event} \rangle ; \end{aligned}$$

The original event definitions that have more than one instance event or multi instance event can be replaced according to the following scheme:

$$\begin{array}{ccc} i : e_1; & \text{is replaced by} & i : e_1; \\ e_2; & & i : e_2; \\ \vdots & & \vdots \quad \vdots \\ e_n; & & i : e_n; \end{array}$$

A similar scheme is used for replacing the event definitions with more than one multi instance event. As a consequence the nonterminals \langle instance event list \rangle and \langle multi instance event list \rangle are redundant.

- The possibility to use the keyword **all** as a means to refer to all instances defined in the MSC is removed. It is assumed that all occurrences of this keyword are replaced by a list of instance names. The BNF rule for the nonterminal \langle instance name list \rangle is replaced by the rule

$$\begin{aligned} \langle \text{instance name list} \rangle ::= & \langle \text{instance name} \rangle \\ & | \langle \text{instance name} \rangle , \langle \text{instance name list} \rangle \end{aligned}$$

- The possibility to use the keyword **loop** with only one inf-natural is removed.

$$\langle \text{loop boundary} \rangle ::= \langle \text{inf natural} \rangle , \langle \text{inf natural} \rangle >$$

The loop boundaries with one inf-natural can be replaced by a loop boundary with two inf-naturals according to the following scheme: $\langle k \rangle$ is replaced by $\langle k, k \rangle$.

- The possibility to use the keyword **loop** without specifying a loop boundary and the possibility to use the loop boundary without using the keyword **loop** are removed. An occurrence of the keyword **loop** without a loop boundary is considered a shorthand for the combination **loop** <1,inf >. An occurrence of a loop boundary *l* without the keyword **loop** is considered a shorthand for the combination **loop** *l*.
- The option inline expression is considered a shorthand for an alternative inline expression with two operands where the second operand is an empty MSC. The exception inline expression is considered a shorthand for an alternative inline expression where the second operand is the part of the MSC following the exception inline expression.
- The option MSC reference expression is considered a shorthand for an alternative MSC reference expression with two operands where the second operand is an empty MSC. The exception MSC reference expression is considered a shorthand for an alternative MSC reference expression where the second operand is the part of the MSC following the exception MSC reference expression.

A.1.4 Extensions

In favour of symmetry, the textual syntax is adapted in such a way that besides the already present before part, for orderable events, an additional after part is created such that both events in a causal ordering have the information that they are causally ordered. This change has several consequences for the textual syntax:

- The BNF rule for the nonterminal <orderable event> is replaced by the rule

$$\begin{aligned} \langle \text{orderable event} \rangle ::= & [\langle \text{event name} \rangle] \\ & \{ \langle \text{message event} \rangle \\ & | \langle \text{incomplete message event} \rangle \\ & | \langle \text{create} \rangle \\ & | \langle \text{timer statement} \rangle \\ & | \langle \text{action} \rangle \\ & \} \\ & [\text{before } \langle \text{event name list} \rangle] \\ & [\text{after } \langle \text{event name list} \rangle] \end{aligned}$$

- The BNF rules for nonterminals <actual order in gate>, <inline order out gate> and <inline order in gate> are replaced by the rules

$$\begin{aligned} \langle \text{actual order in gate} \rangle ::= & \langle \text{gate name} \rangle \text{ after } \langle \text{order dest} \rangle \\ \langle \text{inline order out gate} \rangle ::= & \langle \text{gate name} \rangle \text{ after } \langle \text{order dest} \rangle \\ & [\text{external before } \langle \text{order dest} \rangle] \\ \langle \text{inline order in gate} \rangle ::= & \langle \text{gate name} \rangle \text{ before } \langle \text{order dest} \rangle \\ & [\text{external after } \langle \text{order dest} \rangle] \end{aligned}$$

A.1.5 Assumptions

- It is assumed that the message name alone is sufficient for establishing the correspondence between message input and message output events. As a consequence the optional message instance name and parameter list are removed.
- It is assumed that the MSC reference names and the inline expression names are unique with respect to the MSC document. The nonterminals \langle msc reference name \rangle and \langle inline expr name \rangle are replaced by the nonterminal \langle ref name \rangle .
- It is assumed that the timer name alone is sufficient for establishing if timer events correspond. Thus the nonterminal \langle timer instance name \rangle is removed.
- It is assumed that every MSC reference expression or inline expression has an MSC reference identification or an inline expression identification respectively.
- It is assumed that there are no implicitly defined gates. As a consequence the via-parts in the BNF rules for \langle output address \rangle and \langle input address \rangle become obligatory. Also, the optional gate name in the BNF rules for the nonterminals \langle actual out gate \rangle , \langle actual in gate \rangle , \langle def out gate \rangle and \langle def in gate \rangle become obligatory.
- It is assumed that all external and internal connections of gates of an inline expression are described in its inline gate interfaces. For MSC reference expressions it is assumed that all external connections are described in its MSC reference gate interface.

A.2 Textual syntax for semantics definition

If there are multiple rules for one nonterminal then this should be read as an extension and not as a replacement.

MSC documents

```

 $\langle$ msc document $\rangle$  ::= mscdocument  $\langle$ msc document name $\rangle$  ;
                     $\langle$ msc document body $\rangle$ 
 $\langle$ msc document body $\rangle$ ::=  $\langle$   $\rangle$  |  $\langle$ message sequence chart $\rangle$   $\langle$ msc document body $\rangle$ 

```

Message Sequence Charts

```

 $\langle$ message sequence chart $\rangle$ ::= msc  $\langle$ msc name $\rangle$  ;  $\langle$ msc body $\rangle$  endmsc ;

```

Events

$\langle \text{action} \rangle$::=	action $\langle \text{action character string} \rangle$
$\langle \text{message event} \rangle$::=	$\langle \text{message output} \rangle$ $\langle \text{message input} \rangle$
$\langle \text{message output} \rangle$::=	out $\langle \text{message name} \rangle$ to $\langle \text{input address} \rangle$
$\langle \text{message input} \rangle$::=	in $\langle \text{message name} \rangle$ from $\langle \text{output address} \rangle$
$\langle \text{incomplete message event} \rangle$::=	$\langle \text{incomplete message output} \rangle$ $\langle \text{incomplete message input} \rangle$
$\langle \text{incomplete message output} \rangle$::=	out $\langle \text{message name} \rangle$ to lost [$\langle \text{instance name} \rangle$ env]
$\langle \text{incomplete message input} \rangle$::=	in $\langle \text{message name} \rangle$ from found [$\langle \text{instance name} \rangle$ env]
$\langle \text{create} \rangle$::=	create $\langle \text{instance name} \rangle$ [($\langle \text{parameter list} \rangle$)]
$\langle \text{stop} \rangle$::=	stop
$\langle \text{timer statement} \rangle$::=	$\langle \text{set} \rangle$ $\langle \text{reset} \rangle$ $\langle \text{timeout} \rangle$
$\langle \text{set} \rangle$::=	set $\langle \text{timer name} \rangle$ [($\langle \text{duration name} \rangle$)]
$\langle \text{reset} \rangle$::=	reset $\langle \text{timer name} \rangle$
$\langle \text{timeout} \rangle$::=	timeout $\langle \text{timer name} \rangle$
$\langle \text{condition} \rangle$::=	condition $\langle \text{condition name list} \rangle$
$\langle \text{output address} \rangle$::=	$\langle \text{instance name} \rangle$ env via $\langle \text{gate name} \rangle$ $\langle \text{reference identification} \rangle$ via $\langle \text{gate name} \rangle$
$\langle \text{input address} \rangle$::=	$\langle \text{instance name} \rangle$ env via $\langle \text{gate name} \rangle$ $\langle \text{reference identification} \rangle$ via $\langle \text{gate name} \rangle$
$\langle \text{reference identification} \rangle$::=	reference $\langle \text{ref name} \rangle$ inline $\langle \text{ref name} \rangle$
$\langle \text{parameter list} \rangle$::=	$\langle \text{parameter name} \rangle$ [, $\langle \text{parameter list} \rangle$]
$\langle \text{condition name list} \rangle$::=	$\langle \text{condition name} \rangle$ { , $\langle \text{condition name} \rangle$ }*

Causally ordered events

$\langle \text{orderable event} \rangle$::=	$\langle \text{message event} \rangle$ $\langle \text{incomplete message event} \rangle$ $\langle \text{create} \rangle$ $\langle \text{timer statement} \rangle$ $\langle \text{action} \rangle$
$\langle \text{ordered event} \rangle$::=	$\langle \text{event name} \rangle$ $\langle \text{orderable event} \rangle$ before $\langle \text{event name list} \rangle$ $\langle \text{event name} \rangle$ $\langle \text{orderable event} \rangle$ after $\langle \text{event name list} \rangle$ $\langle \text{event name} \rangle$ $\langle \text{orderable event} \rangle$ before $\langle \text{event name list} \rangle$ after $\langle \text{event name list} \rangle$
$\langle \text{event name list} \rangle$::=	$\langle \text{order dest} \rangle$ [, $\langle \text{event name list} \rangle$]
$\langle \text{order dest} \rangle$::=	$\langle \text{event name} \rangle$ env via $\langle \text{gate name} \rangle$ $\langle \text{reference identification} \rangle$ via $\langle \text{gate name} \rangle$

Coregions

⟨coregion⟩ ::= **concurrent** ; ⟨coevent list⟩ **endconcurrent**
 ⟨coevent list⟩ ::= ⟨⟩ | ⟨orderable event⟩ ; ⟨coevent list⟩

MSC bodies

⟨msc body⟩ ::= ⟨⟩ | ⟨event definition⟩ ⟨msc body⟩
 ⟨event definition⟩ ::= ⟨instance name⟩ : ⟨instance event⟩ ;
 | ⟨instance name list⟩ : ⟨multi instance event⟩ ;
 ⟨instance event⟩ ::= ⟨orderable event⟩ | ⟨non-orderable event⟩
 ⟨non-orderable event⟩ ::= ⟨stop⟩ | ⟨coregion⟩
 ⟨multi instance event⟩ ::= ⟨condition⟩ | ⟨msc reference⟩ | ⟨inline expr⟩
 ⟨instance name list⟩ ::= ⟨instance name⟩
 | ⟨instance name⟩ , ⟨instance name list⟩

MSC reference expressions

⟨msc reference⟩ ::= **reference** ⟨ref name⟩ :
 ⟨msc ref expr⟩ ⟨reference gate interface⟩
 ⟨msc ref expr⟩ ::= ⟨msc ref par expr⟩
 | ⟨msc ref par expr⟩ **alt** ⟨msc ref expr⟩
 ⟨msc ref par expr⟩ ::= ⟨msc ref seq expr⟩
 | ⟨msc ref seq expr⟩ **par** ⟨msc ref par expr⟩
 ⟨msc ref seq expr⟩ ::= ⟨msc ref loop expr⟩
 | ⟨msc ref loop expr⟩ **seq** ⟨msc ref seq expr⟩
 ⟨msc ref loop expr⟩ ::= [**loop** ⟨loop boundary⟩] ⟨expr body⟩
 ⟨expr body⟩ ::= **empty** | ⟨msc name⟩ | (⟨msc ref expr⟩)
 ⟨loop boundary⟩ ::= <⟨inf natural⟩ , ⟨inf natural⟩ >
 ⟨inf natural⟩ ::= **inf** | ⟨natural name⟩+
 ⟨reference gate interface⟩ ::= ⟨⟩ | ; **gate** ⟨ref gate⟩ ⟨reference gate interface⟩
 ⟨ref gate⟩ ::= ⟨actual out gate⟩ | ⟨actual in gate⟩
 | ⟨actual order out gate⟩ | ⟨actual order in gate⟩
 ⟨actual out gate⟩ ::= ⟨gate name⟩ **out** ⟨message name⟩
 to ⟨input address⟩
 ⟨actual in gate⟩ ::= ⟨gate name⟩ **in** ⟨message name⟩
 from ⟨output address⟩
 ⟨actual order out gate⟩ ::= ⟨gate name⟩ **before** ⟨order dest⟩
 ⟨actual order in gate⟩ ::= ⟨gate name⟩ **after** ⟨order dest⟩

Inline expressions

```

<inline expr> ::= <loop expr> | <alt expr> | <par expr>
<loop expr> ::= loop <loop boundary> begin <ref name> ;
               <inline gate interface> <msc body>
               loop end
<alt expr> ::= alt begin <ref name> ; <alt list> alt end
<alt list> ::= <inline gate interface> <msc body>
               | <inline gate interface> <msc body> alt ; <alt list>
<par expr> ::= par begin <ref name> ; <par list> par end
<par list> ::= <inline gate interface> <msc body>
               | <inline gate interface> <msc body> par ; <par list>

<inline gate interface> ::= <> | gate <inline gate> ; <inline gate interface>
<inline gate> ::= <inline out gate> | <inline in gate>
                | <inline order out gate> | <inline order in gate>
<inline out gate> ::= <def out gate> external out <message name>
                    to <input address>
<inline in gate> ::= <def in gate> external in <message name>
                    from <output address>
<inline order out gate> ::= <gate name> after <order dest> external
                          before <order dest>
<inline order in gate> ::= <gate name> before <order dest> external
                          after <order dest>
<def out gate> ::= <gate name> in <message name>
                  from <output address>
<def in gate> ::= <gate name> out <message name>
                 to <input address>

```

High-level Message Sequence Charts

```

<message sequence chart> ::= msc <msc name> ;
                             expr <msc expression>
                             endmsc ;
<msc expression> ::= <start> <node expression list>
<start> ::= <label name list> ;
<node expression list> ::= <> | <node expression> <node expression list>
<node expression> ::= <label name> :
                    { <node> seq ( <label name list> ) | end } ;

```

```
<node> ::= empty
          | <msc name>
          | <par expression>
          | condition <condition name list>
          | connect
          | ( <msc ref expr> )
<par expression> ::= expr <msc expression> endexpr
                    | expr <msc expression> endexpr
                    | par <par expression>
<label name list> ::= <label name> | <label name> alt <label name list>
```

Appendix B

Proofs

In this appendix a number of proofs is presented that are omitted from Chapter 3. In order to make the proofs easier we first present the notion of bisimulation modulo equational reasoning. In short, this notion allows us to use already established equalities between terms in the proofs of other equalities as long as no cyclic reasoning is performed.

B.1 Bisimulation modulo equational reasoning

For the terminology and notation used in this section we refer to [BV95].

Definition B.1.1 Let $T = (\Sigma, D)$ be a term deduction system with stratification S and let $D = (T_p, T_r)$. Let E be a set of equations over Σ . A relation $B \subseteq \mathcal{C}(\Sigma) \times \mathcal{C}(\Sigma)$ is a bisimulation relation modulo a set of equations E if for all $s, t \in \mathcal{C}(\Sigma)$ with sBt the following conditions hold. For all $R \in T_r$

$$\forall_{s' \in \mathcal{C}(\Sigma)} (T_S \models sRs' \Rightarrow \exists_{t' \in \mathcal{C}(\Sigma)} (T_S \models tRt' \wedge E \cup Eq(B) \vdash s' = t')),$$

$$\forall_{t' \in \mathcal{C}(\Sigma)} (T_S \models tRt' \Rightarrow \exists_{s' \in \mathcal{C}(\Sigma)} (T_S \models sRs' \wedge E \cup Eq(B) \vdash s' = t')),$$

and for all $P \in T_p$

$$T_S \models Ps \Rightarrow T_S \models Pt,$$

$$T_S \models Pt \Rightarrow T_S \models Ps.$$

We have used $Eq(B)$ to denote the set of equations that is obtained by considering each pair of B as an equation: $Eq(B) = \{p = q \mid pBq\}$.

Theorem B.1.2 Let Σ be a signature. Let (Σ, D) be a stratifiable term deduction system in panth format. Let E be a set of equations over the signature Σ that is sound with respect to \Leftrightarrow . Then, for all $p, q \in \mathcal{C}(\Sigma)$, if $T_S \models p \Leftrightarrow_{/E} q$, then $T_S \models p \Leftrightarrow q$.

Proof Since the equations of the set E are sound we have the existence of a relation B_e for each axiom $e \in E$ such that B_e is a bisimulation relation for the equation e . Suppose that S is a stratification for the deduction system T . Let B be a bisimulation relation modulo E such that pBq .

In order to prove that $p \Leftrightarrow q$, we need to define a relation B' and prove that this relation relates p and q and that it is a bisimulation relation. We define the relation B' as follows: B' is the smallest relation that satisfies:

- $B \subseteq B'$;
- $B_e \subseteq B'$ for all $e \in E$;
- $Id \subseteq B'$;
- B' is symmetric;
- B' is transitive;
- for any n -ary function symbol $f \in \Sigma$, if $p_i B' q_i$ for all $1 \leq i \leq n$, then $f(p_1, \dots, p_n) B' f(q_1, \dots, q_n)$.

First we prove that such a relation B' exists. Thereto, suppose that B_1 and B_2 are relations that satisfy the above criteria. Then we must establish that also $B_1 \cap B_2$ satisfies these criteria.

- $B \subseteq B_1 \cap B_2$ since $B \subseteq B_1$ and $B \subseteq B_2$.
- for every $e \in E$ we have $B_e \subseteq B_1 \cap B_2$ since $B_e \subseteq B_1$ and $B_e \subseteq B_2$.
- $Id \subseteq B_1 \cap B_2$ since $Id \subseteq B_1$ and $Id \subseteq B_2$;
- Suppose that $p(B_1 \cap B_2)q$. Then clearly pB_1q and pB_2q . Therefore qB_1p and qB_2p . Thus $q(B_1 \cap B_2)p$.
- Suppose that $p(B_1 \cap B_2)q$ and $q(B_1 \cap B_2)r$. Then clearly pB_1q and qB_1r and pB_2q and qB_2r and therefore also pB_1r and pB_2r . Therefore $p(B_1 \cap B_2)r$.
- Suppose that $p_i(B_1 \cap B_2)q_i$ for $1 \leq i \leq n$. Then $p_i B_1 q_i$ and $p_i B_2 q_i$ for $1 \leq i \leq n$. Thus we have $f(p_1, \dots, p_n) B_1 f(q_1, \dots, q_n)$ and $f(p_1, \dots, p_n) B_2 f(q_1, \dots, q_n)$ and therefore $f(p_1, \dots, p_n) (B_1 \cap B_2) f(q_1, \dots, q_n)$.

This illustrates that there is at most one smallest relation B' that satisfies the above criteria. Together with the fact that $\mathcal{C}(\Sigma) \times \mathcal{C}(\Sigma)$ satisfies the above criteria we have

that there is exactly one smallest relation that satisfies the criteria. This enables us to use induction on the structure of the definition of B' to prove that B' is a bisimulation relation.

Thus we have to prove:

$$\begin{aligned} \forall_{s,t \in \mathcal{C}(\Sigma)} sB't &\implies \forall_{P \in S_p} T_S \models Ps \implies T_S \models Pt \wedge T_S \models Pt \implies T_S \models Ps \\ &\quad \wedge \\ &\quad \forall_{R \in S_r} \forall_{s' \in \mathcal{C}(\Sigma)} T_S \models sRs' \implies \exists_{t' \in \mathcal{C}(\Sigma)} T_S \models tRt' \wedge s'B't' \\ &\quad \wedge \\ &\quad \forall_{t' \in \mathcal{C}(\Sigma)} T_S \models tRt' \implies \exists_{s' \in \mathcal{C}(\Sigma)} T_S \models sRs' \wedge s'B't' \end{aligned}$$

Let $s, t \in \mathcal{C}(\Sigma)$ with $sB't$. Then we continue the proof by induction on the structure of B' .

1. Suppose that sBt . Since B is a bisimulation modulo E we have for all $R \in T_r$

$$\forall_{s' \in \mathcal{C}(\Sigma)} (T_S \models sRs' \implies \exists_{t' \in \mathcal{C}(\Sigma)} (T_S \models tRt' \wedge E \cup Eq(B) \vdash s' = t')),$$

$$\forall_{t' \in \mathcal{C}(\Sigma)} (T_S \models tRt' \implies \exists_{s' \in \mathcal{C}(\Sigma)} (T_S \models sRs' \wedge E \cup Eq(B) \vdash s' = t')),$$

and for all $P \in T_p$

$$T_S \models Ps \implies T_S \models Pt,$$

$$T_S \models Pt \implies T_S \models Ps.$$

Thus all that has to be shown is that $E \cup Eq(B) \vdash s' = t'$ implies that $s'B't'$. This is easy as the proof of $E \cup Eq(B) \vdash s' = t'$ can be mimicked precisely in B' .

2. Suppose that $sB_e t$ for some $e \in E$. Trivial since B_e is a bisimulation relation and $B_e \subseteq B'$.
3. Suppose that $sB't$ due to the reflexivity of B' . Then $s \equiv t$. Trivial since the diagonal is a bisimulation relation.
4. Suppose that $sB't$ due to the symmetry of B' . This case is trivial by induction.
5. Suppose that $sB't$ due to $sB'r$ and $rB't$. By induction also trivial.
6. Suppose that $sB't$ due to the existence of an n -ary function symbol $f \in \Sigma$ and $s_i, t_i \in S$ ($1 \leq i \leq n$) such that $s_i B' t_i$ and $s \equiv f(s_1, \dots, s_n)$ and $t \equiv f(t_1, \dots, t_n)$. Then we proceed by transfinite induction on $PF(T)$, the positive formulas of T .

Let $P \in S_p$. Suppose that $T_S \models Ps$. Thus there is a proof for $T_S \models Ps$. By case analysis on the last applied deduction rule or axiom.

- (a) The deduction axiom that is applied last is

$$\frac{}{Px}$$

with a substitution σ such that $\sigma(x) = s$. Define σ' as follows: $\sigma'(x) = t$ and $\sigma'(v) = \sigma(v)$ otherwise. Then clearly also Pt .

- (b) The deduction axiom that is applied last is

$$\frac{}{Pf(x_1, \dots, x_n)}$$

with a substitution σ such that $\sigma(x_i) = s_i$. Define σ' as follows: $\sigma'(x) = t$ and $\sigma'(v) = \sigma(v)$ otherwise. Then clearly also Pt .

- (c) The deduction rule that is applied last is

$$\frac{P_k s_k, t_l R_l y_l, \neg P_m u_m, v_n \neg R_n}{Px}$$

with a substitution σ such that

$$\begin{aligned} \sigma(x) &= s, \\ P_k \sigma(s_k), \\ \sigma(t_l) R_l \sigma(y_l), \\ \neg P_m \sigma(u_m), \\ \sigma(v_n) \neg R_n. \end{aligned}$$

Define σ' as follows: $\sigma'(x) = t$ and $\sigma'(v) = \sigma(v)$ otherwise. Since

$$\begin{aligned} \sigma(s_k) B' \sigma'(s_k), \\ \sigma(t_l) B' \sigma'(t_l), \\ \sigma(y_l) B' \sigma'(y_l), \\ \sigma(u_m) B' \sigma'(u_m), \\ \sigma(v_n) B' \sigma'(v_n) \end{aligned}$$

and the term deduction system is stratifiable we have by induction that $\sigma'(s_k)$, $\sigma'(t_l) R_l \sigma'(y_l)$, $\neg P_m \sigma'(u_m)$ and $\sigma'(v_n) \neg R_n$. Therefore, the same deduction rule is applicable: so Pt .

- (d) The deduction rule that is applied last is

$$\frac{P_k s_k, t_l R_l y_l, \neg P_m u_m, v_n \neg R_n}{Pf(x_1, \dots, x_n)}$$

with a substitution σ such that

$$\sigma(x_i) = s_i, P_k \sigma(s_k), \sigma(t_l) R_l \sigma(y_l), \neg P_m \sigma(u_m), \sigma(v_n) \neg R_n.$$

Define σ' as follows: $\sigma'(x) = t$ and $\sigma'(v) = \sigma(v)$ otherwise. Since

$$\begin{aligned} &\sigma(s_k)B'\sigma'(s_k), \\ &\sigma(t_l)B'\sigma'(t_l), \\ &\sigma(y_l)B'\sigma'(y_l), \\ &\sigma(u_m)B'\sigma'(u_m), \\ &\sigma(v_n)B'\sigma'(v_n) \end{aligned}$$

and the term deduction system is stratifiable we have by induction that $\sigma'(s_k)$, $\sigma'(t_l)R_l\sigma'(y_l)$, $\neg P_m\sigma'(u_m)$ and $\sigma'(v_n)\neg R_n$. Therefore, the same deduction rule is applicable: so Pt .

A similar proof can be given to show that $Pt \implies Ps$.

Similarly we prove that $sRs' \implies tRt' \wedge s'B't'$. \(\square\)

B.2 Properties of \mp

B.2.1 Unit element for \mp

Proof Since the proofs of $t \mp \delta \Leftrightarrow t$ and $\delta \mp t \Leftrightarrow t$ are symmetrical we only give the first one. Consider the relation $B = \{(s \mp \delta, s) \mid s \in \mathcal{C}(\Sigma)\} \cup Id_{\mathcal{C}(\Sigma)}$. The proof that this relation is a bisimulation relation is easy. Consider a pair of the form $(s \mp \delta, s)$ for arbitrary $s \in \mathcal{C}(\Sigma)$. The transfer conditions for both directions are treated simultaneously. First, as $\delta \xrightarrow{a} s$, $s \mp \delta \xrightarrow{a} p'$ if and only if $s \xrightarrow{a} p'$ and note that $p'Bp'$. Secondly, as $\delta \xrightarrow{a} s$, $s \mp \delta \xrightarrow{a} p'$ if and only if $s \xrightarrow{a} p'$ and again note that $p'Bp'$. Finally, as $\delta \not\downarrow$, $s \mp \delta \not\downarrow$ if and only if $s \not\downarrow$. \(\square\)

B.2.2 Commutativity of \mp

Proof Let $B = \{(s \mp t, t \mp s) \mid s, t \in \mathcal{C}(\Sigma)\} \cup Id_{\mathcal{C}(\Sigma)}$. We will show that for all pairs in the relation B the transfer properties hold, i.e., that B is a bisimulation relation. Thus consider a pair of the form $(s \mp t, t \mp s)$ for arbitrary $s, t \in \mathcal{C}(\Sigma)$. We will only treat the transfer conditions from left to right. The transfer conditions in the other direction are treated similarly.

- Suppose that $s \mp t \xrightarrow{a} p$ for some $a \in A$ and $p \in \mathcal{C}(\Sigma)$. Inspection of the deduction rules gives that one of the following must be the case:
 - $s \xrightarrow{a} s'$ and $t \xrightarrow{a} t'$ for some $s', t' \in \mathcal{C}(\Sigma)$ such that $p \equiv s' \mp t'$. In this case we have $t \mp s \xrightarrow{a} t' \mp s'$ and $s' \mp t' B t' \mp s'$.
 - $s \xrightarrow{a} s'$ and $t \not\rightarrow$ for some $s' \in \mathcal{C}(\Sigma)$ such that $p \equiv s'$. Then, $t \mp s \xrightarrow{a} s'$ and $s' B s'$.

- $s \xrightarrow{a}$ and $t \xrightarrow{a} t'$ for some $t' \in \mathcal{C}(\Sigma)$ such that $p \equiv t'$. Then, $t \mp s \xrightarrow{a} t'$ and $t'Bt'$.
- Suppose that $s \mp t \xrightarrow{a} p$ for some $a \in A$ and $p \in \mathcal{C}(\Sigma)$. Inspection of the deduction rules gives that one of the following must be the case:
 - $s \xrightarrow{a} s'$ and $t \xrightarrow{a} t'$ for some $s', t' \in \mathcal{C}(\Sigma)$ such that $p \equiv s' \mp t'$. In this case we have $t \mp s \xrightarrow{a} t' \mp s'$ and $s' \mp t'Bt' \mp s'$.
 - $s \xrightarrow{a} s'$ and $t \xrightarrow{a}$ for some $s' \in \mathcal{C}(\Sigma)$ such that $p \equiv s'$. Then, $t \mp s \xrightarrow{a} s'$ and $s'Bs'$.
 - $s \xrightarrow{a}$ and $t \xrightarrow{a} t'$ for some $t' \in \mathcal{C}(\Sigma)$ such that $p \equiv t'$. Then, $t \mp s \xrightarrow{a} t'$ and $t'Bt'$.
- Suppose that $s \mp t \downarrow$. Inspection of the deduction rules gives that $s \downarrow$ or $t \downarrow$. In both cases also $t \mp s \downarrow$. \square

B.2.3 Associativity of \mp

Proof Consider the relation $B = \{((s \mp t) \mp u, s \mp (t \mp u)) \mid s, t, u \in \mathcal{C}(\Sigma)\} \cup Id_{\mathcal{C}(\Sigma)}$.

- Suppose that $(s \mp t) \mp u \xrightarrow{a} p$ for some $a \in A$ and $p \in \mathcal{C}(\Sigma)$. Inspection of the deduction rules gives that this must be due to one of the following:
 - $s \mp t \xrightarrow{a} p'$ and $u \xrightarrow{a} u'$ for some $p', u' \in \mathcal{C}(\Sigma)$ such that $p \equiv p' \mp u'$. Further inspection gives that $s \mp t \xrightarrow{a} p'$ is due to one of the following:
 - * $s \xrightarrow{a} s'$ and $t \xrightarrow{a} t'$ for some $s', t' \in \mathcal{C}(\Sigma)$ such that $p' \equiv s' \mp t'$. In this case we also have $t \mp u \xrightarrow{a} t' \mp u'$ and hence $s \mp (t \mp u) \xrightarrow{a} s' \mp (t' \mp u')$. Note that $p \equiv p' \mp u' \equiv (s' \mp t') \mp u'$ and $(s' \mp t') \mp u'Bs' \mp (t' \mp u')$.
 - * $s \xrightarrow{a} s'$ and $t \xrightarrow{a}$ for some $s' \in \mathcal{C}(\Sigma)$ such that $p' \equiv s'$. Then $t \mp u \xrightarrow{a} u'$ and $s \mp (t \mp u) \xrightarrow{a} s' \mp u'$. Also $p \equiv p' \mp u' \equiv s' \mp u'$. Note that $s' \mp u'Bs' \mp u'$.
 - * $s \xrightarrow{a}$ and $t \xrightarrow{a} t'$ for some $t' \in \mathcal{C}(\Sigma)$ such that $p' \equiv t'$. This case is similar to the previous one.
 - $s \mp t \xrightarrow{a} p'$ and $u \xrightarrow{a}$ for some $p' \in \mathcal{C}(\Sigma)$ such that $p \equiv p'$. Further inspection gives that $s \mp t \xrightarrow{a} p'$ is due to one of the following:
 - * $s \xrightarrow{a} s'$ and $t \xrightarrow{a} t'$ for some $s', t' \in \mathcal{C}(\Sigma)$ such that $p' \equiv s' \mp t'$. In this case we also have $t \mp u \xrightarrow{a} t'$ and hence $s \mp (t \mp u) \xrightarrow{a} s' \mp t'$. Note that $p \equiv p' \equiv s' \mp t'$ and $s' \mp t'Bs' \mp t'$.
 - * $s \xrightarrow{a} s'$ and $t \xrightarrow{a}$ for some $s' \in \mathcal{C}(\Sigma)$ such that $p' \equiv s'$. Then $t \mp u \xrightarrow{a}$ and $s \mp (t \mp u) \xrightarrow{a} s'$. Also $p \equiv p' \equiv s'$. Note that $s'Bs'$.
 - * $s \xrightarrow{a}$ and $t \xrightarrow{a} t'$ for some $t' \in \mathcal{C}(\Sigma)$ such that $p' \equiv t'$. This case is similar to the previous one.

- $s \mp t \xrightarrow{a}$ and $u \xrightarrow{a} u'$ for some $u' \in \mathcal{C}(\Sigma)$ such that $p \equiv u'$. Inspection of the deduction rules gives that it must be the case that $s \xrightarrow{a}$ and $t \xrightarrow{a}$. Thus, $t \mp u \xrightarrow{a} u'$ and $s \mp (t \mp u) \xrightarrow{a} u'$. Note that $p \equiv p' \equiv u'$ and that $u' Bu'$.
- Suppose that $(s \mp t) \mp u \xrightarrow{a} p$ for some $a \in A$ and $p \in \mathcal{C}(\Sigma)$. This proof is treated in the same way as the first case.
- Suppose that $(s \mp t) \mp u \downarrow$. Inspection of the deduction rules gives that it must be the case that $s \mp t \downarrow$ or $u \downarrow$. If $u \downarrow$, then $t \mp u \downarrow$, and hence $s \mp (t \mp u) \downarrow$. If $s \mp t \downarrow$ it must be the case that $s \downarrow$ or $t \downarrow$. In both cases we easily obtain $s \mp (t \mp u) \downarrow$.

The proofs for the transfer conditions in the other direction are similar and therefore omitted. \square

B.2.4 Idempotency of \mp

Proof We define the relation B as follows:

$$B = \{(t \mp t, t) \mid t \in \mathcal{C}(\Sigma)\} \cup Id_{\mathcal{C}(\Sigma)}.$$

- Suppose that $t \mp t \xrightarrow{a} p'$ for some $a \in A$ and $p' \in \mathcal{C}(\Sigma)$. This must be due to $t \xrightarrow{a} t'$ and $t \xrightarrow{a} t''$ for some $t', t'' \in \mathcal{C}(\Sigma)$ such that $p' \equiv t' \mp t''$. By Theorem 3.10.3 we have $t' \equiv t''$. Note that $p \equiv t' \mp t'' \equiv t' \mp t''$ and $t' \mp t' Bt'$.
- Suppose that $t \xrightarrow{a} t'$ for some $a \in A$ and $t' \in \mathcal{C}(\Sigma)$. Then also, $t \mp t \xrightarrow{a} t' \mp t'$. Note that $t' \mp t' Bt'$.

The proof that the transfer property for the permission relation holds is similar. Also the transfer property of the termination relation is simple: $t' \mp t' \downarrow$ if and only if $t' \downarrow$ or $t' \downarrow$ if and only if $t' \downarrow$. \square

B.3 Properties of \parallel

B.3.1 Unit element for \parallel

Proof We will give the proof for $\varepsilon \parallel t \Leftrightarrow t$. The proof for $t \parallel \varepsilon \Leftrightarrow t$ is similar. Consider the relation $R = \{(\varepsilon \parallel t, t) \mid t \in \mathcal{C}(\Sigma)\}$.

- Suppose that $\varepsilon \parallel t \xrightarrow{a} p$ for some $a \in A$ and $p \in \mathcal{C}(\Sigma)$. Since $\varepsilon \xrightarrow{a}$ this can only be due to $t \xrightarrow{a} t'$ for some $t' \in \mathcal{C}(\Sigma)$ such that $p \equiv \varepsilon \parallel t'$. Note that $\varepsilon \parallel t' R t'$.

- Suppose that $t \xrightarrow{a} q$ for some $a \in A$ and $q \in \mathcal{C}(\Sigma)$. Since $\varepsilon \xrightarrow{a}$ we have $\varepsilon \parallel t \xrightarrow{a} \varepsilon \parallel q$. Note that $\varepsilon \parallel qRq$.
- Suppose that $\varepsilon \parallel t \xrightarrow{a} p$ for some $a \in A$ and $p \in \mathcal{C}(\Sigma)$. This must be due to $\varepsilon \xrightarrow{a} p'$ and $t \xrightarrow{a} p''$ for some $p', p'' \in \mathcal{C}(\Sigma)$ such that $p \equiv p' \parallel p''$. We have $p' \equiv \varepsilon$. Thus $t \xrightarrow{a} p''$ and $\varepsilon \parallel p''Rp''$.
- Suppose that $t \xrightarrow{a} q$ for some $a \in A$ and $q \in \mathcal{C}(\Sigma)$. Then, since $\varepsilon \xrightarrow{a} \varepsilon$, we have $\varepsilon \parallel t \xrightarrow{a} \varepsilon \parallel q$ and $\varepsilon \parallel qRq$.
- Suppose that $\varepsilon \parallel t \downarrow$. This must be due to $\varepsilon \downarrow$ and $t \downarrow$.
- Suppose $t \downarrow$. Since $\varepsilon \downarrow$, also $\varepsilon \parallel t \downarrow$. □

B.3.2 Commutativity of \parallel

Proof This property is an instantiation of the commutativity of generalized weak sequential composition (Theorem 3.5.4, proof in Appendix B.5.1) where the set of requirements is taken to be empty, that is, $S = \emptyset$. □

B.3.3 Distributivity of \parallel over \mp

Proof The property is an instantiation of the distributivity of generalized delayed parallel composition over delayed choice (see Theorem 3.5.4, proof in Appendix B.5.2) where the set of requirements is taken to be empty, that is, $S = \emptyset$. □

B.3.4 Associativity of \parallel

Proof We define the following relation.

$$B = \{(s \parallel t) \parallel u, s \parallel (t \parallel u) \mid s, t, u \in \mathcal{C}(\Sigma)\}.$$

This relation is a bisimulation relation modulo the equations $(x \mp y) \parallel z = x \parallel z \mp y \parallel z$ and $x \parallel (y \mp z) = x \parallel y \mp x \parallel z$. The proof thereof is similar, though easier, than the proof given for the associativity of weak sequential composition in Appendix B.4.4. This is due to the more restricted role of the permission relation, giving less case distinctions. □

B.4 Properties of \circ

B.4.1 Unit element for \circ

Proof We define the relation B as follows:

$$B = \{(\varepsilon \circ t, t) \mid t \in \mathcal{C}(\Sigma)\}.$$

The proof that this relation is indeed a bisimulation is rather trivial.

Consider a pair $(\varepsilon \circ t, t)$ for arbitrary $t \in \mathcal{C}(\Sigma)$.

- Suppose that $\varepsilon \circ t \xrightarrow{a} p$ for some $a \in A$ and $p \in \mathcal{C}(\Sigma)$. Inspection of the deduction rules gives that either one of the following should be the case:
 - $\varepsilon \xrightarrow{a} p'$ and $\varepsilon \xrightarrow{a} \forall t \xrightarrow{a}$ for some $p' \in \mathcal{C}(\Sigma)$ such that $p \equiv p' \circ t$. This case cannot occur since $\varepsilon \not\xrightarrow{a}$.
 - $\varepsilon \xrightarrow{a} p'$, $\varepsilon \xrightarrow{\dots^a} p''$ and $t \xrightarrow{a} t'$ for some $p', p'', t' \in \mathcal{C}(\Sigma)$ such that $p \equiv p' \circ t \mp p'' \circ t'$. This case cannot occur since $\varepsilon \not\xrightarrow{a}$.
 - $\varepsilon \xrightarrow{\dots^a}, \varepsilon \xrightarrow{\dots^a} p'$ and $t \xrightarrow{a} t'$ for some $p', t' \in \mathcal{C}(\Sigma)$ such that $p \equiv p' \circ t'$. From $\varepsilon \xrightarrow{\dots^a} p'$ we obtain $p' \equiv \varepsilon$. We have $t \xrightarrow{a} t'$. Note that $p \equiv p' \circ t' \equiv \varepsilon \circ t'$. Note that $\varepsilon \circ t' B t'$.
- Suppose that $t \xrightarrow{a} t'$ for some $a \in A$ and $t' \in \mathcal{C}(\Sigma)$. Then, since $\varepsilon \xrightarrow{\dots^a}$ and $\varepsilon \xrightarrow{\dots^a} \varepsilon$, we have $\varepsilon \circ t \xrightarrow{a} \varepsilon \circ t'$. Note that $\varepsilon \circ t' B t'$.
- Suppose that $\varepsilon \circ t \xrightarrow{\dots^a} p$ for some $a \in A$ and $p \in \mathcal{C}(\Sigma)$. This must be due to $\varepsilon \xrightarrow{\dots^a} p'$ and $t \xrightarrow{\dots^a} p''$ for some $p', p'' \in \mathcal{C}(\Sigma)$ such that $p \equiv p' \circ p''$. Note that necessarily $p' \equiv \varepsilon$, and hence $p \equiv \varepsilon \circ p''$. We have $t \xrightarrow{\dots^a} p''$. Note that $\varepsilon \circ p'' B p''$.
- Suppose that $t \xrightarrow{\dots^a} q$ for some $a \in A$ and $q \in \mathcal{C}(\Sigma)$. Since $\varepsilon \xrightarrow{\dots^a} \varepsilon$, we have $\varepsilon \circ t \xrightarrow{\dots^a} \varepsilon \circ q$. Note that $\varepsilon \circ q B q$.
- Suppose that $\varepsilon \circ t \downarrow$. This must be due to $\varepsilon \downarrow$ and $t \downarrow$. Hence $t \downarrow$.
- Suppose that $t \downarrow$. From the deduction rules we also have $\varepsilon \downarrow$, and hence $\varepsilon \circ t \downarrow$.

We define the relation B as follows:

$$B = \{(t \circ \varepsilon, t) \mid t \in \mathcal{C}(\Sigma)\}.$$

Consider a pair $(t \circ \varepsilon, t)$ for arbitrary $t \in \mathcal{C}(\Sigma)$.

- Suppose that $t \circ \varepsilon \xrightarrow{a} p$ for some $a \in A$ and $p \in \mathcal{C}(\Sigma)$. Since $\varepsilon \xrightarrow{a}$, this must be due to $t \xrightarrow{a} t'$ for some $t' \in \mathcal{C}(\Sigma)$ such that $p \equiv t' \circ \varepsilon$. We have $t \xrightarrow{a} t'$. Note that $t' \circ \varepsilon B t'$.
- Suppose that $t \xrightarrow{a} q$ for some $a \in A$ and $q \in \mathcal{C}(\Sigma)$. Then, since $\varepsilon \xrightarrow{a}$, also $t \circ \varepsilon \xrightarrow{a} q \circ \varepsilon$. Note that $q \circ \varepsilon B q$.
- Suppose that $t \circ \varepsilon \cdots \xrightarrow{a} p$ for some $a \in A$ and $p \in \mathcal{C}(\Sigma)$. This must be due to $t \cdots \xrightarrow{a} t'$ and $\varepsilon \cdots \xrightarrow{a} p'$ for some $t', p' \in \mathcal{C}(\Sigma)$ such that $p \equiv t' \circ p'$. Note that $p' \equiv \varepsilon$. Thus $p \equiv t' \circ \varepsilon$. We have $t \cdots \xrightarrow{a} t'$. Note that $t' \circ \varepsilon B t'$.
- Suppose that $t \cdots \xrightarrow{a} q$ for some $a \in A$ and $q \in \mathcal{C}(\Sigma)$. Then, since $\varepsilon \cdots \xrightarrow{a} \varepsilon$, also $t \circ \varepsilon \cdots \xrightarrow{a} q \circ \varepsilon$. Note that $q \circ \varepsilon B q$.
- Suppose that $t \circ \varepsilon \downarrow$. This must be due to $t \downarrow$ and $\varepsilon \downarrow$. Thus $t \downarrow$.
- Suppose that $t \downarrow$. Since $\varepsilon \downarrow$ we also have $t \circ \varepsilon \downarrow$. □

B.4.2 Left-zero element for \circ

Proof This property is an instantiation of the property that deadlock is a left-zero for generalized weak sequential composition where the set of requirements is taken to be empty, that is $S = \emptyset$. □

B.4.3 Distributivity of \circ over \mp

Proof This property is an instantiation of the distributivity of generalized weak sequential composition over delayed choice where the set of requirements is taken to be empty, that is $S = \emptyset$. The proof of the distributivity of delayed choice over generalized weak sequential composition is given in Appendix B.5.4. □

B.4.4 Associativity of \circ

Proof

We define the relation B as follows: $B = \{((s \circ t) \circ u, s \circ (t \circ u)) \mid s, t, u \in \mathcal{C}(\Sigma)\}$. We prove that B is a bisimulation relation modulo the equations: $(x \mp y) \circ z = x \circ z \mp y \circ z$ and $x \circ (y \mp z) = x \circ y \mp x \circ z$.

There to, suppose that $p R q$. Then $p \equiv (s \circ t) \circ u$ and $q \equiv s \circ (t \circ u)$ for some $s, t, u \in \mathcal{C}(\Sigma)$. Suppose that $p \xrightarrow{a} p'$ for some $a \in A$ and $p' \in \mathcal{C}(\Sigma)$, i.e., $(s \circ t) \circ u \xrightarrow{a} p'$. This must be due to one of the following:

- $s \circ t \xrightarrow{a} p''$ and $s \circ t \cdots \xrightarrow{a} \forall u \xrightarrow{a}$ for some $p'' \in \mathcal{C}(\Sigma)$ such that $p' \equiv p'' \circ u$. In turn $s \circ t \xrightarrow{a} p''$ is due to one of the following:
 - $s \xrightarrow{a} s'$ and $s \cdots \xrightarrow{a} \forall t \xrightarrow{a}$ for some $s' \in \mathcal{C}(\Sigma)$ such that $p'' \equiv s' \circ t$. If $s \cdots \xrightarrow{a} s''$ for some $s'' \in \mathcal{C}(\Sigma)$, then $t \cdots \xrightarrow{a} \forall u \xrightarrow{a}$ and $t \xrightarrow{a}$ and therefore $s \circ (t \circ u) \xrightarrow{a} s' \circ (t \circ u)$. If $s \cdots \xrightarrow{a}$ then also $s \circ (t \circ u) \xrightarrow{a} s' \circ (t \circ u)$. Note that $\vdash p' \equiv (s' \circ t) \circ u = s' \circ (t \circ u)$.
 - $s \xrightarrow{a} s', s \cdots \xrightarrow{a} s''$ and $t \xrightarrow{a} t'$ for some $s', s'', t' \in \mathcal{C}(\Sigma)$ such that $p'' \equiv s' \circ t \mp s'' \circ t'$. Then $t \cdots \xrightarrow{a} \forall u \xrightarrow{a}$ and therefore $s \circ (t \circ u) \xrightarrow{a} s' \circ (t \circ u) \mp s'' \circ (t' \circ u)$. Note that $\vdash (s' \circ t \mp s'' \circ t') \circ u = s' \circ (t \circ u) \mp s'' \circ (t' \circ u)$.
 - $s \xrightarrow{a}, s \cdots \xrightarrow{a} s''$ and $t \xrightarrow{a} t'$ for some $s'', t' \in \mathcal{C}(\Sigma)$ such that $p'' \equiv s'' \circ t'$. Then $t \cdots \xrightarrow{a} \forall u \xrightarrow{a}$ and therefore $s \circ (t \circ u) \xrightarrow{a} s'' \circ (t' \circ u)$. Note that $\vdash (s'' \circ t') \circ u = s'' \circ (t' \circ u)$.
- $s \circ t \xrightarrow{a} p'', s \circ t \cdots \xrightarrow{a} p'''$ and $u \xrightarrow{a} u'$ for some $p'', p''', u' \in \mathcal{C}(\Sigma)$ such that $p' \equiv p'' \circ u \mp p''' \circ u'$. Since $s \circ t \cdots \xrightarrow{a} p'''$ we have that $s \cdots \xrightarrow{a} s''$ and $t \cdots \xrightarrow{a} t''$ for some $s'', t'' \in \mathcal{C}(\Sigma)$ such that $p''' \equiv s'' \circ t''$. In turn $s \circ t \xrightarrow{a} p''$ is due to one of the following:
 - $s \xrightarrow{a} s'$ and $s \cdots \xrightarrow{a} \forall t \xrightarrow{a}$ for some $s' \in \mathcal{C}(\Sigma)$ such that $p'' \equiv s' \circ t$. Then $s \circ (t \circ u) \xrightarrow{a} s' \circ (t \circ u) \mp s'' \circ (t'' \circ u')$. Note that $\vdash (s' \circ t) \circ u \mp (s'' \circ t'') \circ u' = s' \circ (t \circ u) \mp s'' \circ (t'' \circ u')$.
 - $s \xrightarrow{a} s', s \cdots \xrightarrow{a} s^*$ and $t \xrightarrow{a} t'$ for some $s', s^*, t' \in \mathcal{C}(\Sigma)$ such that $p'' \equiv s' \circ t \mp s^* \circ t'$. Due to permission determinism we have $s^* \equiv s''$. Then $s \circ (t \circ u) \xrightarrow{a} s' \circ (t \circ u) \mp s'' \circ (t' \circ u \mp t'' \circ u')$. Note that $\vdash p' \equiv (s' \circ t \mp s'' \circ t') \circ u \mp (s'' \circ t'') \circ u' = s' \circ (t \circ u) \mp s'' \circ (t' \circ u \mp t'' \circ u')$.
 - $s \xrightarrow{a}, s \cdots \xrightarrow{a} s^*$ and $t \xrightarrow{a} t'$ for some $s^*, t' \in \mathcal{C}(\Sigma)$ such that $p'' \equiv s^* \circ t'$. Due to permission determinism we have $s^* \equiv s''$. Then $s \circ (t \circ u) \xrightarrow{a} s'' \circ (t' \circ u \mp t'' \circ u')$. Note that $\vdash p' \equiv (s'' \circ t') \circ u \mp (s'' \circ t'') \circ u' = s'' \circ (t' \circ u \mp t'' \circ u')$.
- $s \circ t \xrightarrow{a}, s \circ t \cdots \xrightarrow{a} p''$ and $u \xrightarrow{a} u'$ for some $p'', u' \in \mathcal{C}(\Sigma)$ such that $p' \equiv p'' \circ u'$. First of all we obtain $s \cdots \xrightarrow{a} s''$ and $t \cdots \xrightarrow{a} t''$ for some $s'', t'' \in \mathcal{C}(\Sigma)$ such that $p'' \equiv s'' \circ t''$. Then $s \xrightarrow{a}$ and, since $s \cdots \xrightarrow{a}$ also $t \xrightarrow{a}$. Therefore $s \circ (t \circ u) \xrightarrow{a} s'' \circ (t'' \circ u')$. Note that $\vdash p' \equiv (s'' \circ t'') \circ u' = s'' \circ (t'' \circ u')$.

The other five statements are proved similarly. \square

B.5 Properties of \parallel^S and \circ^S

B.5.1 Commutativity of \parallel^S

Proof Let $B = \{(s \parallel^S t, t \parallel^S s) \mid s, t \in \mathcal{C}(\Sigma) \wedge S \subseteq A \times \mathbb{N} \times A\}$. We prove that B is a bisimulation relation modulo the equation $x \mp y = y \mp x$.

Consider a pair of the form $(s \parallel^S t, t \parallel^S s)$ for arbitrary $s, t \in \mathcal{C}(\Sigma)$ and arbitrary $S \subseteq A \times \mathbb{N} \times A$. Let $S' = \text{upd}(a, S)$.

- Suppose that $s \parallel^S t \xrightarrow{a} p'$ for some $a \in A$ and $p' \in \mathcal{C}(\Sigma)$. Inspection of the deduction rules gives that this must be due to one of the following:
 - $s \xrightarrow{a} s', t \xrightarrow{a} t'$ and $\text{enabled}(a, S)$ for some $s', t' \in \mathcal{C}(\Sigma)$ such that $p' \equiv s' \parallel^{S'} t' \mp s \parallel^{S'} t'$. Clearly then also $t \parallel^S s \xrightarrow{a} t' \parallel^{S'} s \mp t \parallel^{S'} s'$ and $\vdash s' \parallel^{S'} t' \mp s \parallel^{S'} t' = t \parallel^{S'} s' \mp t \parallel^{S'} s \parallel^{S'} t'$.
 - $s \xrightarrow{a} s', t \xrightarrow{a}$ and $\text{enabled}(a, S)$ for some $s' \in \mathcal{C}(\Sigma)$ such that $p' \equiv s' \parallel^{S'} t$. Then, we also have $t \parallel^S s \xrightarrow{a} t \parallel^{S'} s'$ and $\vdash s' \parallel^{S'} t = t \parallel^{S'} s'$.
 - $s \xrightarrow{a}, t \xrightarrow{a} t'$ and $\text{enabled}(a, S)$ for some $t' \in \mathcal{C}(\Sigma)$ such that $p' \equiv s \parallel^{S'} t'$. Then, we also have $t \parallel^S s \xrightarrow{a} t' \parallel^{S'} s$ and $\vdash s \parallel^{S'} t' = t' \parallel^{S'} s$.
- Suppose that $s \parallel^S t \xrightarrow{\dots^a} p'$ for some $a \in A$ and $p' \in \mathcal{C}(\Sigma)$. Inspection of the deduction rules gives that this must be due to $s \xrightarrow{\dots^a} s'$ and $t \xrightarrow{\dots^a} t'$ for some $s', t' \in \mathcal{C}(\Sigma)$ such that $p' \equiv s' \parallel^{S'} t'$. Then also $t \parallel^S s \xrightarrow{\dots^a} t' \parallel^{S'} s'$ and $\vdash s' \parallel^{S'} t' = t' \parallel^{S'} s'$.
- Suppose that $s \parallel^S t \downarrow$. This must be due to $s \downarrow$ and $t \downarrow$. Then also $t \parallel^S s \downarrow$.

The proofs that the transfer conditions also hold in the other direction are similar and therefore omitted. \square

B.5.2 Distributivity of \parallel^S over \mp

Proof We define the following relations.

$$\begin{aligned} B_1 &= \{(s \mp t) \parallel^S u, s \parallel^S u \mp t \parallel^S u \mid s, t, u \in \mathcal{C}(\Sigma) \wedge S \subseteq A \times \mathbb{N} \times A\}, \\ B_2 &= \{(s \parallel^S (t \mp u), s \parallel^S t \mp s \parallel^S u) \mid s, t, u \in \mathcal{C}(\Sigma) \wedge S \subseteq A \times \mathbb{N} \times A\}. \end{aligned}$$

These relations are bisimulations modulo the equations $x \mp y = y \mp x$ and $(x \mp y) \mp z = x \mp (y \mp z)$. The proofs thereof are similar, though easier, than the proof given for

the distributivity of delayed choice over the generalized weak sequential composition operator in Appendix B.5.4. This is due to the more restricted role of the permission relation, giving less case distinctions. \square

B.5.3 Left-zero element for \circ^S

Proof Consider the relation $B = \{(\delta \circ^S t, \delta) \mid t \in \mathcal{C}(\Sigma) \wedge S \subseteq A \times IN \times A\}$. We will prove that the conditions for B to be a bisimulation hold. For arbitrary $a \in A$ and terms of the form $\delta \circ^S t$ we easily obtain $\delta \circ^S t \xrightarrow{a}$, $\delta \circ^S t \dots \xrightarrow{a}$ and $\delta \circ^S t \not\downarrow$. We also obtain $\delta \xrightarrow{a}$, $\delta \dots \xrightarrow{a}$ and $\delta \not\downarrow$. \square

B.5.4 Distributivity of \circ^S over \mp

Proof First we prove the right-distributivity of delayed choice over generalized weak sequential composition, as this is the most difficult proof of the two. For the left-distributivity we only give the definition of the relation to be a bisimulation.

We define the relation B as follows:

$$B = \{(s \mp t) \circ^S u, s \circ^S u \mp t \circ^S u \mid s, t, u \in \mathcal{C}(\Sigma) \wedge S \subseteq A \times IN \times A\}.$$

Then we show that B is a bisimulation modulo the equations $x \mp y = y \mp x$ and $(x \mp y) \mp z = x \mp (y \mp z)$. Thereto, let $p, q \in \mathcal{C}(\Sigma)$ such that pBq and $p \equiv (s \mp t) \circ^S u$ and $q \equiv s \circ^S u \mp t \circ^S u$ for some $s, t, u \in \mathcal{C}(\Sigma)$ and $S \subseteq A \times IN \times A$. Let $S' = \text{upd}(a, S)$. Then $(s \mp t) \circ^S u \xrightarrow{a} p'$ is due to one of the following:

- $s \mp t \xrightarrow{a} p'_1$, $s \mp t \dots \xrightarrow{a}$ $\forall u \xrightarrow{a}$ and $\text{enabled}(a, S)$ for some $p'_1 \in \mathcal{C}(\Sigma)$ such that $p' \equiv p'_1 \circ^{S'} u$. In turn $s \mp t \xrightarrow{a} p'_1$ is due to one of the following:
 - $s \xrightarrow{a} s'$ and $t \xrightarrow{a} t'$ for some $s', t' \in \mathcal{C}(\Sigma)$ such that $p'_1 \equiv s' \mp t'$. Then $s \circ^S u \xrightarrow{a} s' \circ^{S'} u$ and $t \circ^S u \xrightarrow{a} t' \circ^{S'} u$ and hence $s \circ^S u \mp t \circ^S u \xrightarrow{a} s' \circ^{S'} u \mp t' \circ^{S'} u$. Note that $\vdash p' \equiv p'_1 \circ^{S'} u \equiv (s' \mp t') \circ^{S'} u = s' \circ^{S'} u \mp t' \circ^{S'} u$.
 - $s \xrightarrow{a} s'$ and $t \not\downarrow$ for some $s' \in \mathcal{C}(\Sigma)$ such that $p'_1 \equiv s'$. Then $s \circ^S u \xrightarrow{a} s' \circ^{S'} u$ and $t \circ^S u \not\downarrow$. Note that $\vdash p' \equiv p'_1 \circ^{S'} u \equiv s' \circ^{S'} u$.
 - $s \not\downarrow$ and $t \xrightarrow{a} t'$ for some $t' \in \mathcal{C}(\Sigma)$ such that $p'_1 \equiv t'$. Similar to the previous case.
- $s \mp t \xrightarrow{a} p'_1$, $s \mp t \dots \xrightarrow{a} p''_1$, $u \xrightarrow{a} u'$ and $\text{enabled}(a, S)$ for some $p'_1, p''_1, u' \in \mathcal{C}(\Sigma)$ such that $p' \equiv p'_1 \circ^{S'} u \mp p''_1 \circ^S u'$. In turn $s \mp t \xrightarrow{a} p'_1$ is due to one of the following:

- $s \xrightarrow{a} s'$ and $t \xrightarrow{a} t'$ for some $s', t' \in \mathcal{C}(\Sigma)$ such that $p'_1 \equiv s' \mp t'$. Also, $s \mp t \xrightarrow{a} p'_1$ is due to one of the following:
 - * $s \xrightarrow{a} s''$ and $t \xrightarrow{a} t''$ for some $s'', t'' \in \mathcal{C}(\Sigma)$ such that $p'_1 \equiv s'' \mp t''$. Since $s \xrightarrow{a} s'$, $s \xrightarrow{a} s''$ and $u \xrightarrow{a} u'$ we have $s \circ^S u \xrightarrow{a} s' \circ^{S'} u \mp s'' \circ^{S'} u'$. Similarly we have $t \circ^S u \xrightarrow{a} t' \circ^{S'} u \mp t'' \circ^{S'} u'$. Thus we have $s \circ^S u \mp t \circ^S u \xrightarrow{a} (s' \circ^{S'} u \mp s'' \circ^{S'} u') \mp (t' \circ^{S'} u \mp t'' \circ^{S'} u')$. Note that $\vdash p' \equiv p'_1 \circ^{S'} u \mp p'_1 \circ^{S'} u' \equiv (s' \mp t') \circ^{S'} u \mp (s'' \mp t'') \circ^{S'} u' = (s' \circ^{S'} u \mp s'' \circ^{S'} u') \mp (t' \circ^{S'} u \mp t'' \circ^{S'} u')$.
 - * $s \xrightarrow{a} s''$ and $t \xrightarrow{a} \dashrightarrow$ for some $s'' \in \mathcal{C}(\Sigma)$ such that $p'_1 \equiv s''$. Since $s \xrightarrow{a} s'$, $s \xrightarrow{a} s''$ and $u \xrightarrow{a} u'$ we have $s \circ^S u \xrightarrow{a} s' \circ^{S'} u \mp s'' \circ^{S'} u'$. Since $t \xrightarrow{a} t'$, $t \xrightarrow{a} \dashrightarrow$ and $u \xrightarrow{a} u'$ we have $t \circ^S u \xrightarrow{a} t' \circ^{S'} u$. Thus we have $s \circ^S u \mp t \circ^S u \xrightarrow{a} (s' \circ^{S'} u \mp s'' \circ^{S'} u') \mp t' \circ^{S'} u$. Note that $\vdash p' \equiv p'_1 \circ^{S'} u \mp p'_1 \circ^{S'} u' \equiv (s' \mp t') \circ^{S'} u \mp s'' \circ^{S'} u' = (s' \circ^{S'} u \mp s'' \circ^{S'} u') \mp t' \circ^{S'} u$.
 - * $s \xrightarrow{a} \dashrightarrow$ and $t \xrightarrow{a} t''$ for some $t'' \in \mathcal{C}(\Sigma)$ such that $p'_1 \equiv t''$. Similar to the previous case.
- $s \xrightarrow{a} s'$ and $t \xrightarrow{a} \dashrightarrow$ for some $s' \in \mathcal{C}(\Sigma)$ such that $p'_1 \equiv s'$. Also, $s \mp t \xrightarrow{a} p'_1$ is due to one of the following:
 - * $s \xrightarrow{a} s''$ and $t \xrightarrow{a} t''$ for some $s'', t'' \in \mathcal{C}(\Sigma)$ such that $p'_1 \equiv s'' \mp t''$. Since $s \xrightarrow{a} s'$, $s \xrightarrow{a} s''$ and $u \xrightarrow{a} u'$ we have $s \circ^S u \xrightarrow{a} s' \circ^{S'} u \mp s'' \circ^{S'} u'$. Since $t \xrightarrow{a} \dashrightarrow$, $t \xrightarrow{a} t''$ and $u \xrightarrow{a} u'$ we have $t \circ^S u \xrightarrow{a} t'' \circ^{S'} u'$. Thus $s \circ^S u \mp t \circ^S u \xrightarrow{a} (s' \circ^{S'} u \mp s'' \circ^{S'} u') \mp t'' \circ^{S'} u'$. Note that $\vdash p' \equiv p'_1 \circ^{S'} u \mp p'_1 \circ^{S'} u' \equiv s' \circ^{S'} u \mp (s'' \mp t'') \circ^{S'} u' = (s' \circ^{S'} u \mp s'' \circ^{S'} u') \mp t'' \circ^{S'} u'$.
 - * $s \xrightarrow{a} s''$ and $t \xrightarrow{a} \dashrightarrow$ for some $s'' \in \mathcal{C}(\Sigma)$ such that $p'_1 \equiv s''$. Since $s \xrightarrow{a} s'$, $s \xrightarrow{a} s''$ and $u \xrightarrow{a} u'$ we have $s \circ^S u \xrightarrow{a} s' \circ^{S'} u \mp s'' \circ^{S'} u'$. Since $t \xrightarrow{a} \dashrightarrow$ and $t \xrightarrow{a} \dashrightarrow$ we have $t \circ^S u \xrightarrow{a} \dashrightarrow$. Thus $s \circ^S u \mp t \circ^S u \xrightarrow{a} s' \circ^{S'} u \mp s'' \circ^{S'} u'$. Note that $\vdash p' \equiv p'_1 \circ^{S'} u \mp p'_1 \circ^{S'} u' \equiv s' \circ^{S'} u \mp s'' \circ^{S'} u'$.
 - * $s \xrightarrow{a} \dashrightarrow$ and $t \xrightarrow{a} t''$ for some $t'' \in \mathcal{C}(\Sigma)$ such that $p'_1 \equiv t''$. Since $s \xrightarrow{a} s'$, $s \xrightarrow{a} \dashrightarrow$ and $u \xrightarrow{a} u'$ we have $s \circ^S u \xrightarrow{a} s' \circ^{S'} u$. Since $t \xrightarrow{a} \dashrightarrow$, $t \xrightarrow{a} t''$ and $u \xrightarrow{a} u'$ we have $t \circ^S u \xrightarrow{a} t'' \circ^{S'} u'$. Thus $s \circ^S u \mp t \circ^S u \xrightarrow{a} s' \circ^{S'} u \mp t'' \circ^{S'} u'$. Note that $\vdash p' \equiv p'_1 \circ^{S'} u \mp p'_1 \circ^{S'} u' \equiv s' \circ^{S'} u \mp t'' \circ^{S'} u'$.
- $s \xrightarrow{a} \dashrightarrow$ and $t \xrightarrow{a} t'$ for some $t' \in \mathcal{C}(\Sigma)$ such that $p' \equiv t'$. Similar to the previous case.
- $s \mp t \xrightarrow{a} \dashrightarrow$, $s \mp t \xrightarrow{a} p''$, $u \xrightarrow{a} u'$ and $\text{enabled}(a, S)$ for some $p'', u' \in \mathcal{C}(\Sigma)$ such that $p' \equiv p'' \circ^{S'} u'$. Since $s \mp t \xrightarrow{a} \dashrightarrow$ we have $s \xrightarrow{a} \dashrightarrow$ and $t \xrightarrow{a} \dashrightarrow$. Also $s \mp t \xrightarrow{a} p''$ must be due to one of the following:
 - $s \xrightarrow{a} s'$ and $t \xrightarrow{a} t'$ for some $s', t' \in \mathcal{C}(\Sigma)$ such that $p'' \equiv s' \mp t'$. Then $s \circ^S u \xrightarrow{a} s' \circ^{S'} u'$ and $t \circ^S u \xrightarrow{a} t' \circ^{S'} u'$ and hence $s \circ^S u \mp t \circ^S u \xrightarrow{a} s' \circ^{S'} u' \mp t' \circ^{S'} u'$. Note that $\vdash p' \equiv p'' \circ^{S'} u' \equiv (s' \mp t') \circ^{S'} u' = s' \circ^{S'} u' \mp t' \circ^{S'} u'$.
 - $s \xrightarrow{a} s'$ and $t \xrightarrow{a} \dashrightarrow$ for some $s' \in \mathcal{C}(\Sigma)$ such that $p'' \equiv s'$. Then $s \circ^S u \xrightarrow{a} s' \circ^{S'} u'$ and $t \circ^S u \xrightarrow{a} \dashrightarrow$, thus $s \circ^S u \mp t \circ^S u \xrightarrow{a} s' \circ^{S'} u'$. Note that $\vdash p' \equiv p'' \circ^{S'} u' \equiv s' \circ^{S'} u'$.

- $s \xrightarrow{a}$ and $t \xrightarrow{a} t'$ for some $t' \in \mathcal{C}(\Sigma)$ such that $p'' \equiv t'$. Similar to the previous case.

The vice versa part of the first statement can be proved similarly. The proof is omitted.

The approach for the proof of the second statement is similar. Suppose that $(s \mp t) \circ^S u \xrightarrow{a} p'$ for some $p' \in \mathcal{C}(\Sigma)$. Then $(s \mp t) \circ^S u \xrightarrow{a} p'$ is due to $s \mp t \xrightarrow{a} p''$ and $u \xrightarrow{a} u'$ for some $p'', u' \in \mathcal{C}(\Sigma)$ such that $p' \equiv p'' \circ^S u'$. In turn $s \mp t \xrightarrow{a} p''$ is due to one of the following:

- $s \xrightarrow{a} s'$ and $t \xrightarrow{a} t'$ for some $s', t' \in \mathcal{C}(\Sigma)$ such that $p'' \equiv s' \mp t'$. Then $s \circ^S u \mp t \circ^S u \xrightarrow{a} s' \circ^S u' \mp t' \circ^S u'$. Note that $\vdash p' \equiv p'' \circ^S u' \equiv (s' \mp t') \circ^S u' = s' \circ^S u' \mp t' \circ^S u'$.
- $s \xrightarrow{a} s'$ and $t \xrightarrow{a}$ for some $s' \in \mathcal{C}(\Sigma)$ such that $p'' \equiv s'$. Then $s \circ^S u \mp t \circ^S u \xrightarrow{a} s' \circ^S u'$. Note that $\vdash p' \equiv p'' \circ^S u' \equiv s' \circ^S u'$.
- $s \xrightarrow{a}$ and $t \xrightarrow{a} t'$ for some $t' \in \mathcal{C}(\Sigma)$ such that $p'' \equiv t'$. Then $s \circ^S u \mp t \circ^S u \xrightarrow{a} t' \circ^S u'$. Note that $\vdash p' \equiv p'' \circ^S u' \equiv t' \circ^S u'$.

Again, the proof for the vice versa part of the second statement is similar and therefore omitted.

The proof for the third statement is even simpler. Suppose that $(s \mp t) \circ^S u \downarrow$. This must be due to $s \mp t \downarrow$ and $u \downarrow$. In turn $s \mp t \downarrow$ must be due to $s \downarrow$ or $t \downarrow$. Thus we have $s \circ^S u \downarrow$ or $t \circ^S u \downarrow$, and therefore $s \circ^S u \mp t \circ^S u \downarrow$. The proof of the vice versa part is again similar.

For left-distributivity we define the relation B as follows: $B = \{(s \circ^S (t \mp u), s \circ^S t \mp s \circ^S u) \mid s, t, u \in \mathcal{C}(\Sigma) \wedge S \subseteq A \times IN \times A\}$. □

B.6 Properties of the repetition operators

B.6.1 Unfolding of iteration

Proof We define the relation B as follows:

$$B = \{(t^\otimes, \varepsilon \mp t \circ t^\otimes) \mid t \in \mathcal{C}(\Sigma)\}.$$

Then we prove that B is a bisimulation relation modulo the equations $(x \mp y) \circ z = x \circ z \mp y \circ z$, $(x \circ y) \circ z = x \circ (y \circ z)$, and $\varepsilon \circ x = x$.

There to consider the pair $(t^\otimes, \varepsilon \mp t \circ t^\otimes)$ for an arbitrary $t \in \mathcal{C}(\Sigma)$.

- Suppose that $t^\otimes \xrightarrow{a} p'$ for some $a \in A$ and $p' \in \mathcal{C}(\Sigma)$. Inspection of the deduction rules gives that this must be due to one of the following:
 - $t \xrightarrow{a} t'$ and $t \xrightarrow{a} \varepsilon$ for some $t' \in \mathcal{C}(\Sigma)$ such that $p' \equiv t' \circ t^\otimes$. Then $t \circ t^\otimes \xrightarrow{a} t' \circ t^\otimes$ and thus $\varepsilon \mp t \circ t^\otimes \xrightarrow{a} t' \circ t^\otimes$. Note that $\vdash t' \circ t^\otimes = t' \circ t^\otimes$.
 - $t \xrightarrow{a} t'$ and $t \xrightarrow{a} t''$ for some $t', t'' \in \mathcal{C}(\Sigma)$ such that $p' \equiv t''^\otimes \circ (t' \circ t^\otimes)$. Then $\varepsilon \mp t \circ t^\otimes \xrightarrow{a} t' \circ t^\otimes \mp t'' \circ (t''^\otimes \circ (t' \circ t^\otimes))$. Note that $\vdash t''^\otimes \circ (t' \circ t^\otimes) = (\varepsilon \mp t'' \circ t''^\otimes) \circ (t' \circ t^\otimes) = t' \circ t^\otimes \mp (t'' \circ t''^\otimes) \circ (t' \circ t^\otimes) = t' \circ t^\otimes \mp t'' \circ (t''^\otimes \circ (t' \circ t^\otimes))$.
- Suppose that $t^\otimes \xrightarrow{a} p'$ for some $a \in A$ and $p' \in \mathcal{C}(\Sigma)$. This must be due to one of the following:
 - $t \xrightarrow{a} t'$ for some $t' \in \mathcal{C}(\Sigma)$ such that $p' \equiv t'^\otimes$. Then also $\varepsilon \mp t \circ t^\otimes \xrightarrow{a} \varepsilon \mp t' \circ t'^\otimes$. Note that $\vdash t'^\otimes = \varepsilon \mp t' \circ t'^\otimes$.
 - $t \xrightarrow{a} \varepsilon$. In that case necessarily $p' \equiv \varepsilon$. Then also $\varepsilon \mp t \circ t^\otimes \xrightarrow{a} \varepsilon$. Note that $\vdash \varepsilon = \varepsilon$.
- Suppose that $t^\otimes \downarrow$. Also $\varepsilon \mp t \circ t^\otimes \downarrow$ since $\varepsilon \downarrow$.
- Suppose that $\varepsilon \mp t \circ t^\otimes \xrightarrow{a} q'$ for some $a \in A$ and $q' \in \mathcal{C}(\Sigma)$. Inspection of the deduction rules gives that this is due to one of the following:
 - $t \xrightarrow{a} t'$ and $t \xrightarrow{a} \forall t^\otimes \xrightarrow{a}$ for some $t' \in \mathcal{C}(\Sigma)$ such that $q' \equiv t' \circ t^\otimes$. Then, $t \xrightarrow{a} \varepsilon$, since $t \xrightarrow{a}$ implies $t^\otimes \xrightarrow{a}$. Then also $t^\otimes \xrightarrow{a} t' \circ t^\otimes$. Note that $\vdash t' \circ t^\otimes = t' \circ t^\otimes$.
 - $t \xrightarrow{a} t'$, $t \xrightarrow{a} t''$ and $t^\otimes \xrightarrow{a} q''$ for some $t', t'', q'' \in \mathcal{C}(\Sigma)$ such that $q' \equiv t' \circ t^\otimes \mp t'' \circ q''$. Since $t \xrightarrow{a} t'$ and $t \xrightarrow{a} t''$ we have $q'' \equiv t''^\otimes \circ (t' \circ t^\otimes)$ and $t^\otimes \xrightarrow{a} t''^\otimes \circ (t' \circ t^\otimes)$. Thus $q' \equiv t' \circ t^\otimes \mp t'' \circ (t''^\otimes \circ (t' \circ t^\otimes))$. Note that $\vdash t''^\otimes \circ (t' \circ t^\otimes) = t' \circ t^\otimes \mp t'' \circ (t''^\otimes \circ (t' \circ t^\otimes))$.
 - $t \xrightarrow{a} \varepsilon$, $t \xrightarrow{a} t'$ and $t^\otimes \xrightarrow{a} q''$ for some $t', q'' \in \mathcal{C}(\Sigma)$ such that $q' \equiv t' \circ q''$. This case cannot occur since if $t \xrightarrow{a} \varepsilon$ then $t^\otimes \xrightarrow{a} \varepsilon$.
- Suppose that $\varepsilon \mp t \circ t^\otimes \xrightarrow{a} q'$ for some $a \in A$ and $q' \in \mathcal{C}(\Sigma)$. This must be due to one of the following:
 - $t \xrightarrow{a} \varepsilon$. Then $q' \equiv \varepsilon$. Then also $t^\otimes \xrightarrow{a} \varepsilon$. Note that $\vdash \varepsilon = \varepsilon$.
 - $t \xrightarrow{a} t'$ for some $t' \in \mathcal{C}(\Sigma)$ such that $q' \equiv \varepsilon \mp t' \circ t'^\otimes$. Then also $t^\otimes \xrightarrow{a} t'^\otimes$. Note that $\vdash t'^\otimes = \varepsilon \mp t' \circ t'^\otimes$.
- Suppose that $\varepsilon \mp t \circ t^\otimes \downarrow$. Also $t^\otimes \downarrow$. \(\square\)

B.6.2 Unfolding of unbounded repetition

Proof We define the relation B as follows:

$$B = \{(t^\infty, t \circ t^\infty) \mid t \in \mathcal{C}(\Sigma)\}.$$

Then we prove that B is a bisimulation relation modulo the equations $(x \mp y) \circ z = x \circ z \mp y \circ z$, $\varepsilon \circ x = x$, $(x \circ y) \circ z = x \circ (y \circ z)$ and $x^\otimes = \varepsilon \mp x \circ x^\otimes$. Let $R = \{(t^\infty, t \circ t^\infty) \mid t \in \mathcal{C}(\Sigma)\}$.

There to consider a pair of the form $(t^\infty, t \circ t^\infty)$ for arbitrary $t \in \mathcal{C}(\Sigma)$.

- Suppose that $t^\infty \xrightarrow{a} p'$ for some $a \in A$ and $p' \in \mathcal{C}(\Sigma)$. This is due to one of the following:
 - $t \xrightarrow{a} t'$ and $t \xrightarrow{\dots^a} p'$ for some $t' \in \mathcal{C}(\Sigma)$ such that $p' \equiv t' \circ t^\infty$. Then also $t \circ t^\infty \xrightarrow{a} t' \circ t^\infty$. Note that $\vdash t' \circ t^\infty = t' \circ t^\infty$.
 - $t \xrightarrow{a} t'$ and $t \xrightarrow{\dots^a} t''$ for some $t', t'' \in \mathcal{C}(\Sigma)$ such that $p' \equiv t''^\otimes \circ (t' \circ t^\infty)$. Then $t \circ t^\infty \xrightarrow{a} t' \circ t^\infty \mp t'' \circ (t''^\otimes \circ (t' \circ t^\infty))$. Note that $\vdash t''^\otimes \circ (t' \circ t^\infty) = t' \circ t^\infty \mp t'' \circ (t''^\otimes \circ (t' \circ t^\infty))$.
- Suppose that $t^\infty \xrightarrow{\dots^a} p'$. This must be due to $t \xrightarrow{\dots^a} t''$ for some $t'' \in \mathcal{C}(\Sigma)$ such that $p' \equiv t''^\infty$. Then also $t \circ t^\infty \xrightarrow{\dots^a} t'' \circ t''^\infty$. Note that $\vdash t''^\infty = t'' \circ t''^\infty$.
- Suppose that $t^\infty \downarrow$. This case cannot occur.

The proof that the transfer conditions hold in the other direction are similar and therefore omitted. \square

B.6.3 Inclusion

Proof We define the relation B as follows:

$$B = \{(t^\otimes \mp t^\infty, t^\otimes) \mid t \in \mathcal{C}(\Sigma)\}.$$

Then we prove that B is a bisimulation relation modulo the equation $x \circ (y \mp z) = x \circ y \mp x \circ z$.

Consider a pair of the form $(t^\otimes \mp t^\infty, t^\otimes)$ for an arbitrary $t \in \mathcal{C}(\Sigma)$.

With respect to the transfer conditions for the transition relation two cases can be distinguished. First, if $t \xrightarrow{a} t'$ and $t \xrightarrow{\dots^a} t''$, then $t^\otimes \mp t^\infty \xrightarrow{a} t''^\otimes \circ (t' \circ t^\otimes) \mp t''^\otimes \circ (t' \circ t^\infty)$ and $t^\otimes \xrightarrow{a} t''^\otimes \circ (t' \circ t^\otimes)$. Note that $\vdash t''^\otimes \circ (t' \circ t^\otimes) \mp t''^\otimes \circ (t' \circ t^\infty) = t''^\otimes \circ (t' \circ t^\otimes \mp t' \circ t^\infty) = t''^\otimes \circ (t' \circ (t^\otimes \mp t^\infty)) = t''^\otimes \circ (t' \circ t^\otimes)$. Secondly, if $t \xrightarrow{a} t'$ and $t \xrightarrow{\dots^a} p'$, then $t^\otimes \mp t^\infty \xrightarrow{a} t' \circ t^\otimes \mp t' \circ t^\infty$ and $t^\otimes \xrightarrow{a} t' \circ t^\otimes$. Note that $\vdash t' \circ t^\otimes \mp t' \circ t^\infty = t' \circ (t^\otimes \mp t^\infty) = t' \circ t^\otimes$.

With respect to the transfer conditions for the permission relation two cases can be distinguished. First, if $t \xrightarrow{\dots^a} t''$, then $t^\otimes \mp t^\infty \xrightarrow{\dots^a} t''^\otimes \mp t''^\infty$ and $t^\otimes \xrightarrow{\dots^a} t''^\otimes$. Note

that $\vdash t''^{\otimes} \mp t''^{\infty} = t''^{\otimes}$. Secondly, if $t \xrightarrow{a}$, then $t^{\otimes} \mp t^{\infty} \xrightarrow{a} \varepsilon$ and $t^{\otimes} \xrightarrow{a} \varepsilon$. Note that $\vdash \varepsilon = \varepsilon$.

The transfer conditions for the termination predicate are satisfied trivially as $t^{\otimes} \mp t^{\infty} \downarrow$ if and only if $t^{\otimes} \downarrow \vee t^{\infty} \downarrow$ if and only if $t^{\otimes} \downarrow$. \square

B.6.4 Other properties

Proof We define the relation B as follows:

$$B = \{(t^{\otimes} \circ t^{\otimes}, t^{\otimes}) \mid t \in \mathcal{C}(\Sigma)\}.$$

Then we prove that B is a bisimulation relation modulo the equations $x \mp x = x$, $\varepsilon \circ x = x$ and $(x \circ y) \circ z = x \circ (y \circ z)$.

There to consider the pair $(t^{\otimes} \circ t^{\otimes}, t^{\otimes})$ for arbitrary $t \in \mathcal{C}(\Sigma)$.

For the transfer conditions for the transition relation two cases can be distinguished. First, suppose that $t \xrightarrow{a} t'$ and $t \xrightarrow{a}$. Then, $t^{\otimes} \circ t^{\otimes} \xrightarrow{a} (t' \circ t^{\otimes}) \circ t^{\otimes}$ and $t^{\otimes} \xrightarrow{a} t' \circ t^{\otimes}$. Note that $\vdash (t' \circ t^{\otimes}) \circ t^{\otimes} = t' \circ (t^{\otimes} \circ t^{\otimes}) = t' \circ t^{\otimes}$. Secondly, suppose that $t \xrightarrow{a} t'$ and $t \xrightarrow{a} t''$. Then $t^{\otimes} \circ t^{\otimes} \xrightarrow{a} (t''^{\otimes} \circ (t' \circ t^{\otimes})) \circ t^{\otimes} \mp t''^{\otimes} \circ (t''^{\otimes} \circ (t' \circ t^{\otimes}))$ and $t^{\otimes} \xrightarrow{a} t''^{\otimes} \circ (t' \circ t^{\otimes})$. Note that $\vdash (t''^{\otimes} \circ (t' \circ t^{\otimes})) \circ t^{\otimes} \mp t''^{\otimes} \circ (t''^{\otimes} \circ (t' \circ t^{\otimes})) = t''^{\otimes} \circ (t' \circ t^{\otimes})$.

For the transfer conditions for the permission relation two cases can be distinguished. First, suppose that $t \xrightarrow{a}$. Then, $t^{\otimes} \circ t^{\otimes} \xrightarrow{a} \varepsilon \circ \varepsilon$ and $t^{\otimes} \xrightarrow{a} \varepsilon$. Note that $\vdash \varepsilon \circ \varepsilon = \varepsilon$. Secondly, suppose that $t \xrightarrow{a} t''$. Then, $t^{\otimes} \circ t^{\otimes} \xrightarrow{a} t''^{\otimes} \circ t''^{\otimes}$ and $t^{\otimes} \xrightarrow{a} t''^{\otimes}$. Note that $\vdash t''^{\otimes} \circ t''^{\otimes} = t''^{\otimes}$. \square

B.6.5 Other properties (II)

Proof We define the relation B as follows:

$$B = \{(t^{\otimes} \circ t^{\infty}, t^{\infty}) \mid t \in \mathcal{C}(\Sigma)\}.$$

Then, we prove that B is a bisimulation relation modulo the equations $(x \circ y) \circ z = x \circ (y \circ z)$, $\varepsilon \circ x = x$, $x \mp x = x$ and $x^{\otimes} \circ x^{\otimes} = x^{\otimes}$.

There to, consider a pair of the form $(t^{\otimes} \circ t^{\infty}, t^{\infty})$ for an arbitrary $t \in \mathcal{C}(\Sigma)$.

With respect to the transfer conditions for the transition relation we distinguish two cases. First, suppose that $t \xrightarrow{a} t'$ and $t \xrightarrow{a}$. Then $t^{\otimes} \circ t^{\infty} \xrightarrow{a} (t' \circ t^{\otimes}) \circ t^{\infty}$ and $t^{\infty} \xrightarrow{a} t' \circ t^{\infty}$. Note that $\vdash (t' \circ t^{\otimes}) \circ t^{\infty} = t' \circ (t^{\otimes} \circ t^{\infty}) = t' \circ t^{\infty}$. Secondly, suppose

that $t \xrightarrow{a} t'$ and $t \xrightarrow{\dots a} t''$. Then, $t^{\otimes} \circ t^{\infty} \xrightarrow{a} (t''^{\otimes} \circ (t' \circ t^{\otimes})) \circ t^{\infty} \mp t''^{\otimes} \circ (t''^{\otimes} \circ (t' \circ t^{\infty}))$ and $t^{\infty} \xrightarrow{a} t''^{\otimes} \circ (t' \circ t^{\infty})$. Note that $\vdash (t''^{\otimes} \circ (t' \circ t^{\otimes})) \circ t^{\infty} \mp t''^{\otimes} \circ (t''^{\otimes} \circ (t' \circ t^{\infty})) = t''^{\otimes} \circ (t' \circ t^{\infty})$.

The transfer conditions for the permission relation and the termination predicate are trivial. \square

Samenvatting

Het onderwerp van dit proefschrift is de taal Message Sequence Chart (MSC) en de definitie van haar formele semantiek. Message Sequence Chart is een grafische taal voor de beschrijving van de interactie van systeemcomponenten. Elke systeemcomponent wordt gerepresenteerd door een verticale lijn, de proceslijn. Langs een proceslijn loopt de tijd van boven naar beneden. Communicatie wordt verondersteld asynchroon te verlopen. Er worden geen aannames gemaakt met betrekking tot de wijze waarop communicatie wordt bewerkstelligd. De uitwisseling van boodschappen tussen systeemcomponenten wordt weergegeven door pijlen tussen de betrokken proceslijnen. De pijl is gericht van de proceslijn van de zendende systeemcomponent naar de proceslijn van de ontvangende systeemcomponent. Impliciet wordt er verondersteld dat het verzenden van een boodschap plaatsvindt voor de ontvangst van de boodschap.

De taal MSC wordt gestandaardiseerd door de ITU, de International Telecommunication Union. In maart 1993 verscheen de eerste standaard met betrekking tot de taal MSC [IT93]. Deze standaard bevat een informele definitie van de grafische syntax en formele abstracte syntax en concrete syntax definities. Daarnaast bevat deze standaard een informele uitleg van de betekenis van een MSC en enkele voorbeelden van de grafische en tekstuele syntax. Een belangrijke tekortkoming van deze eerste standaard is een formele definitie van de betekenis van een MSC. In oktober 1994 wordt hier iets aan gedaan middels de acceptatie van een formele semantiek van MSC gebaseerd op procesalgebra [IT95] als standaard Z.120 Annex B.

In april 1996 wordt de taal MSC uitgebreid met een groot aantal compositiemechanismen [IT96b]. Als gevolg daarvan is de formele definitie van de betekenis van een MSC zoals beschreven in Z.120 Annex B [IT95] niet langer toereikend.

In dit proefschrift wordt in Hoofdstuk 1 ingegaan op de ontwikkelingen welke geleid hebben tot de definitie van de taal MSC en haar formele semantiek. Ook wordt kort ingegaan op het gebruik van de taal MSC en op enkele aan MSC gerelateerde formalismen.

In Hoofdstuk 2 wordt een uitvoerige informele uitleg van de taal MSC gegeven. Deze uitleg heeft betrekking op de grafische syntax, de informele betekenis van de taalconstructies en de tekstuele syntax. Naast de elementen van de taal MSC welke al in de eerste standaard aanwezig waren, is ook een uitleg gegeven van de constructies welke

voor het eerst in de tweede standaard (1996) verschenen. Daartoe behoren onder andere

- uitgebreidere faciliteiten voor het ordenen van activiteiten,
- faciliteiten voor het (de)componeren van MSCs,
- een mechanisme om de uitwisseling van boodschappen tussen systeemcomponenten welke in verschillende MSCs beschreven zijn weer te geven, en
- een mechanisme om activiteiten welke in verschillende MSCs beschreven zijn te ordenen.

In Hoofdstuk 3 wordt een procestheorie gedefinieerd welke in Hoofdstuk 4 gebruikt zal worden voor de definitie van de betekenis van MSCs. Deze procestheorie bestaat uit termen opgebouwd uit constanten en operaties. De constanten die gebruikt worden zijn:

- ε voor het proces dat niets doet behalve termineren,
- δ voor het proces dat niets doet en zelfs niet kan termineren, en
- $a \in A$ atomaire acties welke de activiteiten die in de MSCs beschreven zijn representeren.

De operaties die gebruikt worden zijn sterk gebaseerd op de compositiemechanismen uit de taal MSC:

- \mp (delayed choice) voor de modelering van alternatieven,
- \circ en \circ^S voor de beschrijving van verticale compositie van MSCs
- \parallel and \parallel^S voor de beschrijving van horizontale compositie van MSCs
- \otimes en ∞ voor de beschrijving van de repetitie van een MSC

De constanten en operatoren worden voorzien van een operationele semantiek die beschrijft welke activiteiten een proces kan uitvoeren. Voor de operationele semantiek van de procestheorie in dit hoofdstuk zijn de volgende activiteiten van belang: het uitvoeren van een atomaire actie, het kunnen termineren en het toestaan dat een ander proces bepaalde acties uitvoert. Deze activiteiten worden formeel gedefinieerd door middel van een deductiesysteem.

Gebaseerd op de activiteiten beschreven door middel van de operationele semantiek wordt een notie van gelijkheid gedefinieerd op processen welke sterke gelijkheid vertoont met bisimulatie.

In Hoofdstuk 4 wordt de semantiek van de taal MSC gedefinieerd door een afbeelding te geven van syntactische objecten uit de taal MSC naar het domein van de procesexpressies die in Hoofdstuk 3 gedefinieerd is. Voor deze afbeelding worden niet de grafische objecten gebruikt maar de tekstuele representaties van deze objecten. De reden hiervoor is dat de grafische syntax informeel is. Het basisidee volgens welke deze afbeelding werkt is dat elk object uit de taal MSC opgedeeld kan worden in een aantal activiteiten en dat de relatie tussen deze activiteiten nauwkeurig geadmineistreerd kan worden. Voor elk van deze activiteiten wordt een atomaire actie in de procestheorie gedefinieerd. Door middel van de operaties uit de procestheorie kunnen de atomiare acties weer samengevoegd worden tot een procesexpressie die de betekenis van het MSC object weergeeft.

Het proefschrift besluit met enkele afsluitende opmerkingen, een uitgebreide lijst met referenties en enkele bijlagen.

Curriculum Vitae

11 June 1970 Born in Eindhoven, Noord-Brabant, The Netherlands.

September 1982 - August 1988 Secondary school (VWO), Hertog Jancollege in Valkenswaard.

September 1988 - April 1994 Master's degree (cum laude) in technical computing science, specialization 'formal methods' at Eindhoven University of Technology, The Netherlands.

May 1994 - April 1998 PhD, Eindhoven University of Technology, The Netherlands.

From May 1998 Senior research assistant at CWI (Dutch Foundation for Mathematics and Computing Science), Amsterdam, The Netherlands.

Titles in the IPA Dissertation Series

The State Operator in Process Algebra

J. O. Blanco

Faculty of Mathematics and Computing Science, TUE, 1996-1

Transformational Development of Data-Parallel Algorithms

A. M. Geerling

Faculty of Mathematics and Computer Science, KUN, 1996-2

Interactive Functional Programs: Models, Methods, and Implementation

P. M. Achten

Faculty of Mathematics and Computer Science, KUN, 1996-3

Parallel Local Search

M. G. A. Verhoeven

Faculty of Mathematics and Computing Science, TUE, 1996-4

The Implementation of Functional Languages on Parallel Machines with Distrib. Memory

M. H. G. K. Kessler

Faculty of Mathematics and Computer Science, KUN, 1996-5

Distributed Algorithms for Hard Real-Time Systems

D. Alstein

Faculty of Mathematics and Computing Science, TUE, 1996-6

Communication, Synchronization, and Fault-Tolerance

J. H. Hoepman

Faculty of Mathematics and Computer Science, UvA, 1996-7

Reductivity Arguments and Program Construction

H. Doornbos

Faculty of Mathematics and Computing Science, TUE, 1996-8

Functorial Operational Semantics and its Denotational Dual

D. Turi

Faculty of Mathematics and Computer Science, VUA, 1996-9

Single-Rail Handshake Circuits

A. M. G. Peeters

Faculty of Mathematics and Computing Science, TUE, 1996-10

A Systems Engineering Specification Formalism

N. W. A. Arends

Faculty of Mechanical Engineering, TUE, 1996-11

Normalisation in Lambda Calculus and its Relation to Type Inference

P. Severi de Santiago

Faculty of Mathematics and Computing Science, TUE, 1996-12

Abstract Interpretation and Partition Refinement for Model Checking

D. R. Dams

Faculty of Mathematics and Computing Science, TUE, 1996-13

Topological Dualities in Semantics

M. M. Bonsangue

Faculty of Mathematics and Computer Science, VUA, 1996-14

Algorithms for Graphs of Small Treewidth

B. L. E. de Fluiter

Faculty of Mathematics and Computer Science, UU, 1997-01

Process-algebraic Transformations in Context

W. T. M. Kars

Faculty of Computer Science, UT, 1997-02

A Generic Theory of Data Types

P. F. Hoogendijk

Faculty of Mathematics and Computing Science, TUE, 1997-03

The Evolution of Type Theory in Logic and Mathematics

T. D. L. Laan

Faculty of Mathematics and Computing Science, TUE, 1997-04

Preservation of Termination for Explicit Substitution

C. J. Bloo

Faculty of Mathematics and Computing Science, TUE, 1997-05

Discrete-Time Process Algebra

J. J. Vereijken

Faculty of Mathematics and Computing Science, TUE, 1997-06

A Functional Approach to Syntax and Typing

F. A. M. van den Beuken

Faculty of Mathematics and Informatics, KUN, 1997-07

Ins and Outs in Refusal Testing

A.W. Heerink

Faculty of Computer Science, UT, 1998-01

A Discrete-Event Simulator for Systems Engineering

G. Naumoski and W. Alberts

Faculty of Mechanical Engineering, TUE, 1998-02

Scheduling with Communication for Multiprocessor Computation

J. Verriet

Faculty of Mathematics and Computer Science, UU, 1998-03

An Asynchronous Low-Power 80C51 Microcontroller

J. S. H. van Gageldonk

Faculty of Mathematics and Computing Science, TUE, 1998-04

In Terms of Nets: System Design with Petri Nets and Process Algebra

A. A. Basten

Faculty of Mathematics and Computing Science, TUE, 1998-05

Inductive Datatypes with Laws and Subtyping – A Relational Model

E. Voermans

Faculty of Mathematics and Computing Science, TUE, 1999-01

Towards Probabilistic Unification-based Parsing

H. ter Doest

Faculty of Computer Science, UT, 1999-02

Algorithms for the Simulation of Surface Processes

J.P.L. Segers

Faculty of Mathematics and Computing Science, TUE, 1999-03

Recombinative Evolutionary Search

C.H.M. van Kemenade

LIACS, faculty of Mathematics and Natural Sciences, Leiden University, 1999-04

Learning Reliability: a Study on Indecisiveness in Sample Selection

E.I. Barakova

Faculty of Mathematics and Natural Sciences, RUG, 1999-05

Schedulere Optimization in Real-Time Distributed Databases

M.P. Bodlaender

Faculty of Mathematics and Computing Science, TUE, 1999-06

Message Sequence Chart: Syntax and Semantics

M.A. Reniers

Faculty of Mathematics and Computing Science, TUE, 1999-07

Stellingen

behorende bij het proefschrift

Message Sequence Chart

Syntax and Semantics

van

M.A. Reniers

1

De taal Message Sequence Chart dankt zijn populariteit niet aan het bestaan van een standaard.

2

Een voorstel tot uitbreiding van een formele taal zoals MSC met een taalconstructie dient ten minste de volgende zaken te omvatten:

- een motivatie voor de toevoeging van de taalconstructie;
- een voorstel voor de syntax van de taalconstructie;
- een informele omschrijving van de betekenis van de taalconstructie;
- een uitgebreid overzicht van de interactie van de taalconstructie met alle taalconstructies die al in de taal aanwezig zijn;
- een duidelijke omschrijving van de restricties waaraan het gebruik van de taalconstructie onderhevig is;
- voorbeelden welke de vorige punten illustreren.

Pas als een dergelijke gedetailleerde bestudering plaats heeft gevonden dient overwogen te worden daadwerkelijk tot toevoeging over te gaan. In de praktijk van standaardisatie van de taal MSC blijkt al besloten te worden tot toevoeging op grond van slechts de eerste twee van de bovengenoemde punten.

3

Alur, Holzmann en Peled geven, gebaseerd op verschillende aannamen over de manier waarop communicatie tot stand komt, drie verschillende semantiekken voor Message Sequence Charts met alleen communicatie [AHP96]. Het is beter om de uitgangspunten die de drie semantiekken voortbrengen expliciet in de syntax van de taal op te nemen zodat de gebruiker zelf de mogelijkheid heeft het gewenste communicatiemodel te specificeren.

4

Het is mogelijk om gegeven een MSC vast te stellen in welke communicatiemodellen het gedrag dat dit MSC beschrijft, gerealiseerd kan worden [EMR97].

5

Het is niet waarschijnlijk dat de procesalgebra $BPA_{\delta\epsilon}$ [BW90] op een conservatieve wijze uitgebreid kan worden met tijd onder de aanname dat de constanten uit $BPA_{\delta\epsilon}$ in de theorie met tijd willekeurige hoeveelheden tijd kunnen laten passeren voordat de actie daadwerkelijk uitgevoerd wordt.

6

De axiomas voor de interne stap in discrete tijd, relatieve tijd procesalgebra zoals gepresenteerd in [BBR] vormen een volledige axiomatisering van 'rooted branching tail bisimulation' op gesloten procestermen.

7

Er dient meer aandacht te zijn voor de correctheidsbewijzen van axiomatiseringen in de procesalgebra [BV95, RW94, Gro97, Ver97].

8

De invoering van notebooks aan de Technische Universiteit Eindhoven is een slechte zaak voor zowel de fysieke als geestelijke gesteldheid van de studenten. Bovendien is de kantine in het hoofdgebouw er minder gezellig van geworden.

9

De diverse herzieningen van het curriculum van de opleiding technische informatica aan de Technische Universiteit Eindhoven waarin de wiskundige component steeds verder teruggebracht wordt, leidt tot afgestudeerden welke niet over het vereiste abstractieniveau en het logisch deductievermogen beschikken om op een gestructureerde wijze met software- en systeemontwikkeling om te gaan.

10

De tendens om een deel van de artikelen die verschenen zijn in de proceedings van een conferentie ook als special issue in een tijdschrift te publiceren, doet afbreuk aan de kwaliteit van publicaties in tijdschriften.

11

Het zou promovendi toegestaan moeten zijn te promoveren aan de Technische Universiteit Eindhoven op een proefschrift zonder stellingen.

Referenties

- [AHP96] R. Alur, G.J. Holzmann, and D. Peled. An analyzer for Message Sequence Charts. *Software - Concepts and Tools*, 17(2):70–77, 1996.
- [BBR] J.C.M. Baeten, J.A. Bergstra, and M.A. Reniers. Discrete time process algebra with silent step. To appear in G. Plotkin, C. Stirling and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*.
- [BK84] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1/3):109–137, 1984.
- [BV95] J.C.M. Baeten and C. Verhoef. Concrete process algebra. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Semantic Modelling*, volume 4 of *Handbook of Logic in Computer Science*, pages 149–268. Oxford University Press, 1995.
- [BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.
- [EMR97] A. Engels, S. Mauw, and M.A. Reniers. A hierarchy of communication models for Message Sequence Charts. In T. Mizuno, N. Shiratori, T. Higashino, and A. Togashi, editors, *Formal Description Techniques and Protocol Specification, Testing and Verification*, Proceedings of FORTE X and PSTV XVII '97, pages 75–90, Osaka, 1997. Chapman & Hall.
- [Gro97] J.F. Groote. The syntax and semantics of timed μ CRL. Technical Report SEN-R9709, CWI, 1997.
- [RW94] A. Rensink and H. Wehrheim. Weak sequential composition in process algebras. In B. Jonsson and J. Parrow, editors, *CONCUR'94: Concurrency Theory*, volume 836 of *Lecture Notes in Computer Science*, pages 226–241, Uppsala, 1994. Springer-Verlag.
- [Ver97] J.J. Vereijken. *Discrete-Time Process Algebra*. PhD thesis, Eindhoven University of Technology, 1997.