# Languages for analysis and testing of event sequences

# Languages for Analysis and Testing of Event Sequences

Engels, Andreas Gerhardus
Languages for Analysis and Testing of Event Sequences
Ph.D. thesis, Eindhoven University of Technology, 2001


IPA Dissertation Series 2001-07
druk: UniversiteitsDrukkerij, Eindhoven


The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

# Languages for Analysis and Testing of Event Sequences

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 dinsdag 29 mei 2001 om 16.00 uur

door

Andreas Gerhardus Engels

geboren te Borger

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.ir. L.M.G. Feijs
en
prof.dr. J.C.M. Baeten

Copromotor:
dr. S. Mauw

# Preface

Science is work of man. If there is one thing I have come to realize these last few years, it is that. It has been a hard lesson. My four years here in Eindhoven as a PhD student started out very well. Articles were written, sent in, published. A success story in the making, or so it seemed. But as time passed, things went less and less the way we would want them to go. Projects became slow, or I even halted for a while – only to find that starting again was even harder than continuing was originally. We all knew that something was wrong, or at least I did. But what it was became clear only when things really got out of hand one day.

I had been under too much stress. And although my work was not the main cause, my situation did make it impossible for me to keep working, and in fact had already done so for several months. I spent most of the next half year on sick leave for it. Given these circumstances, it is almost a miracle that this booklet has come about with only half a year's delay.

I would like to thank everybody who has helped me get through this period of my life. In particular, I would like to thank my parents, who provided me with a place where I could recuperate and find the strength to carry on. Also great thanks to all my colleagues. When times were bad, you coped with me and helped me quiet down again. When times were better, you were friends to me and we had fun together. Very special thanks to Sjouke Mauw, who has always been a pleasure to work with, both from a personal and from a professional point of view. Jos, Marcella (thank you for the tea), Dragan, Martijn, Susanna and Tim, thanks for your help in keeping or regaining quietness. Jan-Joris, you provided me with a very pleasant atmosphere in the first phase of my times here. I would also like to thank IPA for financially enabling my work, and the faculty of mathematics and computer science for various support.

It is good to know that there are friends to support me, and I would like to thank Pieter, Benedikt, Tim and Satomi and her family for giving me this support as well as the pleasant times we could share.

Much of the material in this thesis has come about in cooperation with, or with technical support from, other people. We mention Sjouke Mauw, Loe Feijs, Michel Reniers, Thijs Cobben en Rogier Vermeulen with whom we have worked together, while other people who have given useful technical help with one or more chapters, are Piet Bakker, Roel Bloo, Victor Bos, Jan Docekal, Herman Geuvers, Jan-Friso Groote, Øysten Haugen, Thierry Jeron, Clive Jervis, Bart Knaack, Erik Kwast, Frans Meijs, Jaco van der Pol, M. de Vreugd, and especially Joost-Pieter Katoen, whose

remarks did much to improve the thesis.

A number of chapters have already been published before. Chapter 2 has been published as [EFM97]. Chapter 5 has been published as [EMR97a] and in a shortened form as [EMR97b]. Chapter 6 has partly been published as [Eng00], and is partly based on [EFM99]. Chapter 7 has been published as [Eng98], and Chapter 8 as [CE98]. Chapter 3, which is joint work with Thijs Cobben, Loe Feijs and Rogier Vermeulen, Chapter 4 and part of Chapter 6 have not been published before.

The MSCs in this thesis have been created using the LaTeX MSC macro package [BM99].

# Contents

# Chapter 1

# Introduction

In computer science, it is important to be able to describe what a computer system does, or what it should do. If a system or program is simple, or if only a very general description is necessary, a natural language such as English might suffice, but when systems grow to even moderate sizes, natural language gets too cumbersome to use. Some of the disadvantages of natural language as a specification language are:

- Natural language descriptions are often inexact. One text often has more than one interpretation.
- When descriptions becomes long, it becomes hard to find an overview.
- It is hard to define with mathematical precision whether a given system corresponds to a description in natural language.
- Natural language descriptions are not well-equipped to decide properties of a system from its description.
- The structures of natural language do not correspond to the natural structures of computer systems.

To resolve these and similar problems, formal languages have been introduced. These are dedicated languages, based on mathematics, for the description of computer systems. A formal language consists of two parts, the first is a formal syntax, the second a formal semantics. The syntax defines what descriptions in the language look like, while the semantics define what such a description actually means. In most cases, such semantics consist of a formal translation into some other mathematical formalism.

In this thesis, we will both look at the construction of formal languages, with an emphasis on the task of finding a good semantics (that is, a semantics that in the first place corresponds with the intended intuitive meaning of the language and in the second place is easy to work with), and at their usage. For both tasks a few case studies have been conducted. The language construction part consists of a number of extensions to an existing language as well as the creation of a small, new language for a small sub-domain, while the usage part consists of an attempt to derive properties

of a system from a formal specification and an attempt to use formal languages and methods for the creation of test sequences.

The work in this thesis is partly based on practical problems that were encountered over time. This holds in particular for the Chapters 6 and 8. Both data and disrupt/interrupt were felt within the ITU (International Telecommunications Union) to be possibly useful extensions to the MSC (Message Sequence Charts) language. Partly on our own initiative and partly because of questions from within the MSC community, the Eindhoven formal methods group explored especially the semantic consequences of these changes. A similar background holds for Chapter 3. We have studied the current testing process at KPN (the largest Dutch service provider in the area of telecommunication), and the study started from an idea to improve this process.

One language that we will in particular look into is MSC (Message Sequence Chart) [IT00, RGG96b]. This is a graphical language which in particular describes the communication behaviour of a system. As such, it is very useful for the description of communication protocols, and for the specification of distributed systems where communication is the most important aspect. A strong point of MSC is that it combines a graphical syntax, which is relatively easy to understand for humans, with a strict formal semantics, which enables automated analysis by computers.

In theory, one could derive a program from a specification in a formal way, or prove its correctness mathematically, but in practice systems are often built and then tested, rather than proven correct. Several reasons for this discrepancy between theory and practice can be given:

Formal methods and their possibilities are often not known.

Many formal methods do not scale up very well.

Certain errors and properties, such as hardware errors and timing properties, are hard or impossible to find without testing.

Correctness of a system must sometimes be ascertained by companies that do not have the actual code, and can only look at the system as a 'black box'.

Although testing often takes up a considerable portion of the devekionebt process (in some cases more than half of it), it is often remarkably little formalised and automated. Test traces are still often designed by hand, and sometimes even the outcomes are checked by hand. If these could be automated, more, or more complicated, test traces could be checked in the same time, and thus the quality of testing could be improved and/or the time needed for testing could be reduced. There already exists a formal language for the specification of test traces, namely TTCN (Tree and Tabular Combined Notation) [KW91]. Deriving TTCN from a system description is (in most cases) not a complicated task, but finding test traces or sets of test traces with certain properties (for example, being in some sense 'complete' or finding some specific errors) often is.

## 1.1 Organisation of the Thesis

This thesis roughly consists of two parts. The first part, discussing the subject of testing, consists of Chapters 2 and 3. The second part, about MSC, is comprised of the Chapters 4 through 5.

The first part contains two Chapters. In the first (Chapter 2), we look at Model Checking as a possible tool for the derivation of test traces. The number of states of a system is often very large, and finding test traces with given properties can therefore be difficult. Because model checking tools have many methods to deal with this 'state space explosion', we look whether they can be used to find test traces to certain behaviours of these systems. Chapter 3 looks at a later phase of the testing process. We introduce *LOGAN*, a language developed at the Eindhoven University formal methods group. It is originally developed for the testing of telephone systems, to partly replace the manual check of test logs. The idea of the language is that it can be used to automatically find the events in a test log that correspond to a single call.

Chapter 4 gives an introduction of MSC, with a look into its history, an overview of some of its main features and an introduction to the formal semantics. The next chapter shows, how formal language descriptions of a system can be used to determine properties of that system. We will derive the buffering architecture of a system from its MSC description. The Chapter creates a hierarchy of these architectures.

The last three Chapters discuss some (existing or possible future) additions to MSC. First, Chapter 6 discusses the introduction of data in MSC. By this introduction, it is now possible to use variables and parameters in MSC descriptions. The Chapter tells how this addition was done, and why it was done that way. There is also a semantics of this aspect of MSC being developed. The next two Chapters discuss some other extensions that could in the future be added to MSC. Chapter 7 introduces message refinement, which introduces a refinement method that makes it possible to look at protocols at different levels of abstraction. Chapter 8 introduces disrupt and interrupt, which can be used to describe situations where one behaviour is stopped half-way to start another type of behaviour, a situation that is currently hard or impossible to describe in MSC. Both Chapters contain a discussion of the semantics of the features that are introduced.

# Chapter 2

# Test Derivation Using Model Checking

## 2.1 Introduction

In this chapter, we will discuss test derivation, and more specifically test derivation for telecommunication systems. Testing is necessary in several phases of a development process. In the first place there is the testing during the specification phase. Central questions in this respect are whether the specification follows the requirements, and whether any logical errors are present in the specification. In telecommunication systems a probable cause for those logical errors is feature interaction [Mid94], that is, the effect that different features (variations on the basic protocol) have on one another. This could for example happen if one feature changes a variable another feature uses or changes as well.

Secondly, it is also necessary to test whether the implementation conforms to the specification. Feature interaction is again an important subject, for example through the sharing of (necessarily finite) resources. It is this second testing phase that will be investigated in this chapter.

One problem in testing is the creation of a suitable test-set, a set of test traces to be checked. Manual generation of test traces is a lot of work, so it is natural to look for computer support. In relatively small cases this is perfectly feasible: there are techniques and tools that, given a formal specification, generate a complete set of test traces. See for example [Nah94]. Assuming that the implementation has as many states as the specification, a positive result of the test can be considered a correctness proof. In many practical cases there is a so-called state space explosion, that causes the number and/or the length of the traces to be (much) larger than can be dealt with. In this case one has to choose which traces are and which are not to be tested. This selection of interesting traces requires much insight in the problem at hand, so cannot be automated. Still, support in this process will be useful.

Our aim is not to create yet more new tools, but to find and link existing tools that suit this purpose. At present this also means we do not go beyond prototyping.

In particular we will try to use tools from model checking to generate traces. model checking provides us with serious tools with a good theoretical foundation and the possibility to work with large examples (so we can more easily cope with the problems of the state space explosion). Another important advantage is that tools to translate the output of SPIN (the model checker we used) [Hol96] into useful formats, either exist or are easily created. In short, this chapter will be about the usage of existing tools for the purpose of test trace generation for implementation checking of systems. We will use this for systems that are far beyond full state space checking, that is, where a full state space check can not be reached or even approximated.

As an example to apply our methods to we have chosen Intelligent Networks. This is an important application in which conformance testing, as well as other tests, are a necessity. Because of the regular addition of features these tests also have to be repeated during the lifespan of the network. Moreover, these features can have unexpected interactions, so every time a test has to be done of the system as a whole. The addition of features often causes an exponential growth of the state space, so a state space explosion will be almost a certainty. We used a model of a telephone service, with two features: Originating Call Screening (OCS) and Hotline. We have successfully applied our method, and the developed prototype, to this simplified example.

## 2.2 Methodology

In this section, we give an overview of the testing methodology that is proposed in this chapter.

The starting point will be the specification of the system under study. We have taken an SDL [IT94, SRS89] specification as our input. The first step is to translate this specification (manually) into a form understood by the model checker. Since we used the model checker SPIN [Hol96], we translated this SDL-specification into Promela, the modeling language of SPIN. The structure of a Promela model (several parallel processes, which can communicate both through shared variables and through channels), also fits neatly with SDL-descriptions. The Promela-code was created from the SDL-specification by hand, but a few macros were used to bring the Promela code closer to the SDL-code. Our correctness criterium for the implementation will be, that every possible trace of the implementation must also be a trace of the SDL model (and thus of the Promela model, which is assumed to be equivalent), as far as its external behaviour is concerned. Furthermore, the system may not deadlock if the SDL model does not deadlock. Because we will be assuming that the SDL description is deterministic, this actually means that the traces must be equivalent.

During this translation, and even during the creation of the specification itself, it is a good thing to already start looking at the testing goals. Sometimes auxiliary variables are necessary to count the number of times a certain step in the process has been taken or is being taken (as the value of such a variable might be part of the testing goal). Also the degree of simplification might differ, depending on what is to be tested.

In this model we also incorporate a so-called stimulation process. This is an added process, that regulates the external inputs and/or the independent actions of the system to be tested. It sends messages to the other processes, either through a

specialised channel or through the change of some variable, that normally work as a trigger for performing some activity. For example, in our test case, a model of a telephony service, the stimulation process regulates which calls are to be attempted. The stimulation process is restricted to only those parts of the system that actually are under outside control. Every message that is in the stimulation process, must correspond to a communication from the outside world, or an outside-controlled part of the system, to the system.

Next, we have to develop a test purpose. This consists of the desired characteristics of the test traces to be developed. Of course, these characteristics have to be chosen such that the test traces to be found have a high chance of catching implementation errors, that is, they should describe a situation where the system is likely to behave differently from the specification in the case of errors. Because of this, it is important to guess which kind of errors are most likely to occur, or most important to be found.

In their most simple form these testing purposes consist of a property or a set of properties for the final state of the trace, but they could also be more complicated, for example a series of states (distinguished by their properties) that have to be traversed, or an added restriction on the states before the final one. The only consideration is that it must be possible to write the testing purposes down in (temporal) logic.

These two additions (stimulation process and testing purpose) give two ways of controlling the test trace developed. The stimulation process describes the search space for traces, while the testing purpose regulates which kind of traces are actually generated. Of course these two will be connected: On the one hand one can cause the stimulation process to make only those actions happen which bring the testing purpose closer, thus making the number of possibilities checked smaller, or one can disallow the most trivial ways of reaching the testing purpose, thus finding other, possibly more interesting, traces. Finally, one can re-use the same testing purpose by using it together with different stimulators.

We now take a model checker (in our example SPIN), and take the negation of the testing purpose as a so-called never-claim. In normal usage of model checkers the never-claim is an asserted logical invariant of the model, and should therefore never become false (hence the name never-claim). The model checker then runs the model, checking whether the never-claim ever becomes false, and presenting a trace that makes the never-claim false if this is the case. Here we take the negation of the testing purpose, which will cause the trace found to be one in which the negation of the testing purpose is false, and hence the testing purpose itself is reached. In general there will be more than one test trace possible that fulfills the testing purpose. In that case the model checker will make an essentially nondeterministic choice (although some model checkers might allow one to find all traces, or one particular (such as the shortest)).

From an output of SPIN (which contains all the information about the trace that is found) we create Interworkings (IW) [MvWW93], a (TUE and Philips) local variant of synchronous MSC-like diagrams (see Chapter 4 for a detailed introduction of MSC). The reason we do this, is that our final goal is to derive a test description in TTCN [KW91], and there is a tool available [FJ96] to translate IW into TTCN. The SPIN-output also contains some MSCs, which can be used for a quick scan of the trace found, and thus can give help for human control of the test generation process.

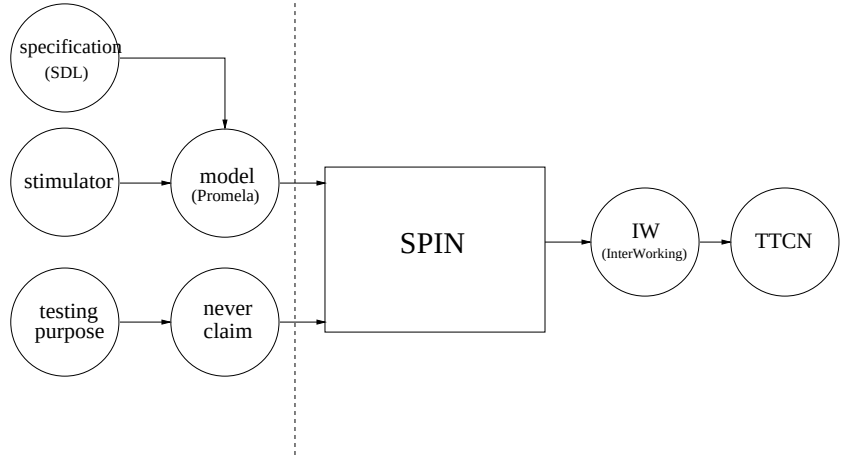



Figure 2.1: General method

A scheme of the method can be found in Figure 2.1. The part on the left left of the dashed line is mainly manual work, the part on the right is done by tools.

## 2.3 Case Study: Testing Intelligent Networks

In this section we apply our method to an example from the field of Intelligent Networks (IN). We have modeled a telephony service with two features, OCS (Originating Call Screening) and HOT (Hotline). Using our methods, we will derive a single test trace.

### 2.3.1 Intelligent Networks

Because of the ever-growing amount of possibilities of telephone services, a new paradigm for telephony and connected telecommunication has been developed: Intelligent Networks. The following citation from [SMC+96] characterises the IN-concept:

> Intelligent Network (IN) services are customised telephone services, like e.g., 1) 'Free-Phone', where the receiver of the call can be billed if some conditions are met, 2) 'Universal Private Telephone', enabling groups of customers to define their own private net within the public net, or 3) 'Partner Lines', where a number of menus leads to the satisfaction of all desires. The realisation of these services is quite complex and error prone.
>
> The current trend in advanced IN services clearly evolves towards decoupling Service Processing Systems from the switch network (see e.g. [CK94]). The reasons for this tendency lie in the growing need for decentralisation of the service processing, in the demand for quick customisation of the offered services, and in the requirement of rapid availability of the modified or reconfigured services.

> Service Creation Environments for the creation of IN-services are usually based on classical 'Clipboard-Architecture' environments, where services are graphically constructed, compiled, and successively tested. Two extreme approaches characterise the state of the art: The first approach guarantees consistency, but the creation process is strongly limited in its flexibility to compose Service Independent Building Blocks (SIBs) to new services. The second approach allows flexible compositions of services, but there is little or no feedback on the correctness of the service under creation during the development: the validation is almost entirely located after the design is completed. Thus the resulting test phase is lengthy and costly.

For more information on IN we refer to e.g. [BW94, CK94].

In [VWK95] the need for service testing in the context of IN is explained and a framework for testing telecommunication services is presented, where it is stressed that next to the service itself, the underlying platform and the already existing services should be tested as well.

### 2.3.2 A Simple Model

We will model an Intelligent Network with two special services (which we call 'features'), namely OCS (Originating Call Screening) and a simplified version of HOT (Hotline). In Originating Call Screening, phone calls can be blocked by the receiver depending on the originator of the call. In Hotline, the dialing of frequently used numbers is made easier by causing another (smaller) number to result in the same connection. In our model, the Hotline will be established on dialing any number that is not a service number. Adding a feature can be done without much problems (although it will increase the size of the space state, and thus might cause the generation time for the test trace to increase slightly).

In our model we will decouple the SSP (Service Switching Point), which connects the user with the services, and the SCP (Service Control Point), which physically contains the services. This decoupling is suggested within the literature (see above), because the implementation and maintenance of the system is easier if the (stable) basic functions are decoupled from the (dynamically added) features. For the model there would be only little change if we had not implemented this decoupling, but for the test trace generation this might very well be important, since an overload of the connection between SSP and SCP might be a cause of errors.

In Figure 2.2 we see the architecture of an Intelligent Network as it is represented by our model. The SSP is responsible for the connections between the telephones, and the connections between the telephones and the SCP. It can be modeled in for example SDL.

Through a special channel, ss7, the SSP is connected to the SCP. The SCP checks which, if any, features have to be used in a given call. In our model the SDP (Service Data Point), which does the maintenance of the features, that is, keeping track of which features are enabled for whom and how they are configured, and the SCP have been combined into one process. The SCP is normally modeled by Service Independent Building Blocks (SIBs) [SMC+96].

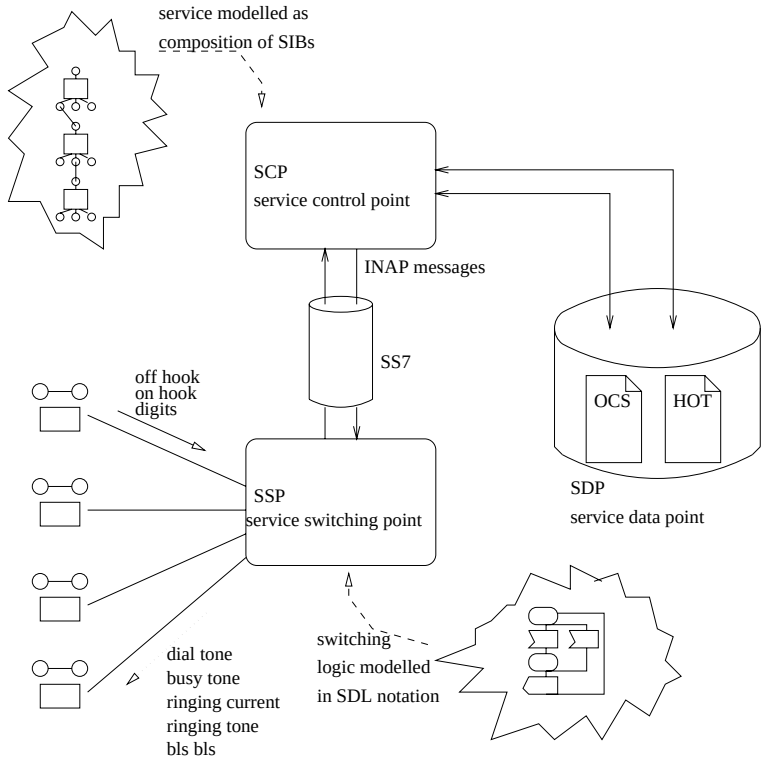



Figure 2.2: IN architecture

In Figure 2.3 we see an SDL-model of the SSP as we have modeled it. This model is based on [RV94]. Identifier A here stands for the person attempting to make the call, B for the person being called. The SSP is idle until there is an off-hook message (A off hook), after which a dial tone is sent to phone A, and then the SSP is in the await_digits-state. This one can end in two ways, namely by A putting the phone on hook, and by A dialing a number. In the first case the (attempted) call ends, and the SSP becomes idle again. In the second case, the SSP checks whether the line called is busy; if it is it generates a busy tone and waits for an on_hook, otherwise the second phone starts ringing and a conversation is attempted. The rest of the figure reads likewise.

This scheme is simplified from the form we used in our model in a few ways: Firstly, there can be more than one attempt for making a talk. Because of this, many copies of this scheme are running at the same time, one for each call attempted. Secondly, not all actions (the sending of tones and talk across the telephone lines) are shown. In the third place, this only specifies the behavior in absence of any special features. The presence of features influences the effect in the following ways:

- Hotline changes the line with which a connection is attempted.
- OCS makes 'called line busy?' true even when the line is not busy.

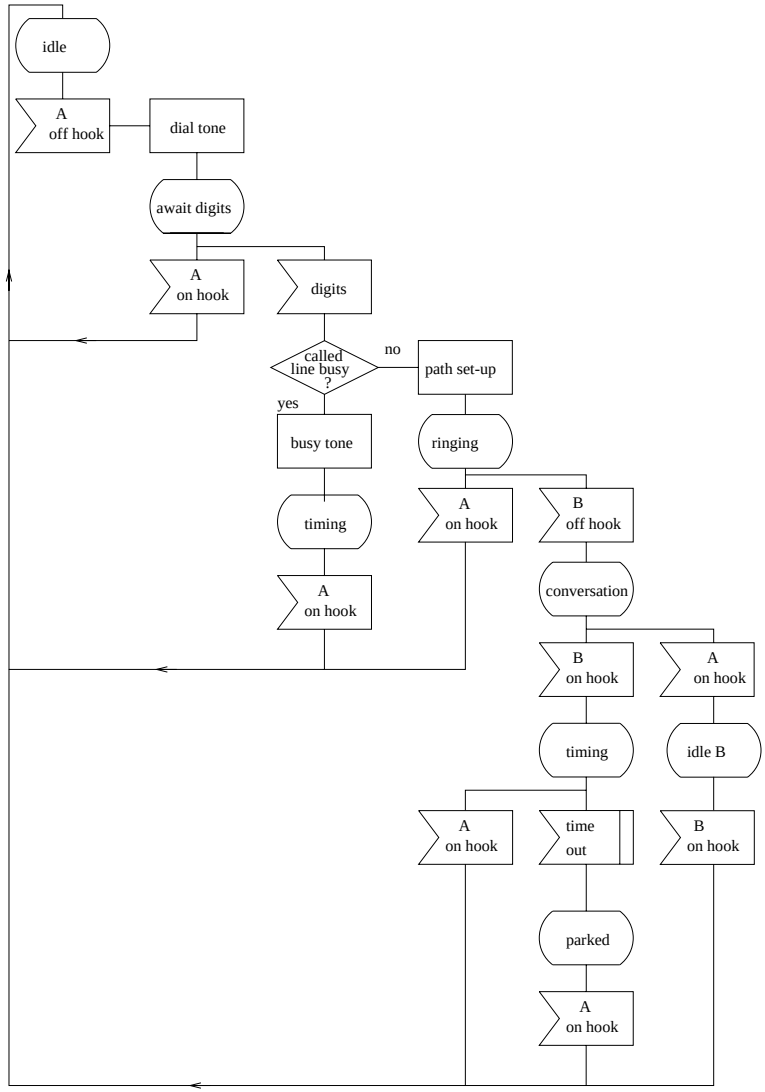



Figure 2.3: SDL-model of the SSP

- After the digits are dialed, it is first checked whether they are the digits for adding or removing a feature. If so, the feature is enabled/disabled/changed, and the phone waits for an on_hook.

We translated this SDL-model into PROMELA-code. We will not give the complete (5-page) PROMELA-code of our model here, but only a few representative parts. First, we give a part of the SSP-code:

```
STATE(ringing)
:: signal[A]?on_hook  -> atomic {
                           line[A] = silent;
                           line[B] = silent;
                           busy[A] = false;
                           busy[B] = false;
                           NEXTSTATE(idle)
                         }
:: intern[B]?off_hook -> atomic {
                           line[A] = blabla(B);
                           line[B] = blabla(A);
                           BCP_account[A]++;
                           NEXTSTATE(conversation)
                         }
ENDSTATE
```

In Figure 2.3 this state ('ringing') can be found a bit above the middle (below the action 'path set-up'). Take special notice of the variable BCP_account[A]. It is incremented each time A has made a successful attempt to engage in a talk. It has no function in the model, but we include it because (part of) our testing goal will be that the number of calls by one subscriber exceeds a certain number.

The model of the SCP is shown below (OCS is a 5 times 5 boolean array, HOT is an integer array of length 5):

```
do
:: ss7?feature(A,B) -> if
                       :: B/100 == 66 -> HOT[A] = (B - 6600)
                       :: B/100 == 88 -> OCS[B - 8800,A] = true
                       :: B/100 == 89 -> OCS[B - 8900,A] = false
                       fi
:: ss7?check(A,B) ->   s = OCS[A,B];
                       ss7!checked(!s,_);
                       if
                       :: (s == 1) -> IN_account[B]++
                       :: else -> skip
                       fi
:: ss7?lookup(A,B) ->  if
                       :: (HOT[A] == A) -> ss7!lookuped(A,B)
                       :: (HOT[A] != A) -> ss7!lookuped(A,HOT[A])
                       fi
od
```

First, there are a few arrays: OCS[A,B] is true iff B is blocking messages from A; HOT[A] is the Hotline A has (if it has A as its value, A does not have a Hotline). These arrays are filled in the skipped part.

The SCP gets its orders from the SSP through the ss7-channel. The message 'feature(A,B)' adds or removes a feature, the message 'check(A,B)' asks whether A

is allowed to call B, and the message 'lookup(A,B)' tells A has dialed the number B, and asks with which phone a connection should actually be attempted.

Through this same channel the SCP sends the results back to the SSP. If the order was a check, it sends 'checked(!s,_)', where s is true iff B is blocking messages from A, while the second one ('_') is a dummy variable, which is only needed because the ss7 is a 3-variable channel. If the order was a lookup, then 'lookuped(A,H)' is sent, where H is A's hotline if any, and B otherwise, so in fact it is sending the number that will be the real receiver of the message.

As before, there is an auxiliary variable: IN_ACCOUNT[B], which counts the number of calls to B that have been blocked by B.

The stimulation process controls the amount of non-determinism in the system. An example of a stimulation process can be found below:

```
proctype stimula()
{
        call[2]!6603;           /* 2 has Hotline to 3  */
        call[4]!8801;           /* 1 should not call 4 */
        do
        :: call[4]!8901         /* 1 may call 4 again  */
        :: call[1]!4
        od
}
```

The action call[A]!B sends a message to phone A, telling it to attempt to make a call in which it dials number B. So the stimulation process above first orders phone 2 to create a Hotline to 3, then orders phone 4 to create an OCS towards 1, and then goes through a cycle, every time either ordering phone 4 to stop its OCS towards 1, or ordering phone 1 to attempt a call to phone 4.

This is of course just one example of a stimulation process. We have worked with several different processes in order to get different traces.

### 2.3.3 Generating a Test Sequence

As an example, we will generate an interesting trace. As a working hypothesis we assumed that problems were likely to arise due to mistaken allocation of shared resources, especially if some resource was used too extensively. This leads to testing goals like 'There are $n$ SDP-accesses taking place' However, because our main goal was the testing of the feasibility of the general method, we have only used the simplest cases in practice, such as:

- Phone A has made a successful call
- Phone A has made two successful calls
- Phone A and B have been connected in a successful call
- An SDP-access is taking place

In practice more complex situations have to be checked. This might cause a longer computation time, because the minimal length of a trace that has the desired

properties is longer, and the testing purpose is more complicated. Neither seems to be really problematic, though.

As an example we take the testing goal 'Phone 1 has made a successful call', with the stimulation process as described above.

In SPIN the testing goal can be implemented as a "never-claim".

```
#define ALWAYS(P) never { do :: P :: !(P) -> break od }
#define CLAIM(A) (BCP_account[A] < 1)
ALWAYS(CLAIM(1))
```

SPIN will look for traces in which the process defined by the never-claim has ended. In our definition of ALWAYS(P) this means that P has been false at some place of the trace – in fact, at the last step of the trace. So if we make our claim ALWAYS(P), then SPIN will be looking for a trace that ends with a situation in which P is NOT true. As we want to have a trace in which phone 1 has made a (successful) call, this P must be 'Phone 1 has not made a call', which, because of the addition of the variable BCP_account[A] into our model, simply translates into 'BCP_account[1]<1'.

This model was run using SPIN (XSpin). It did find a trace to a state in which the BCP-account of telephone 1 is at least 1. The main part of the SPIN-output consists of listings of the following form. It is in fact a complete list of the actions taken by the various processes, with some information added (the line of code where the action is described, the value of variables that have changed, etcetera).

```
1: proc - (:never:) line 319 "pan_in" (state 1) [((BCP_account[1]<1))]
2: proc 1 (:init:) line 323 "pan_in" (state 1) [(run phone(0))]
3: proc - (:never:) line 319 "pan_in" (state 1) [((BCP_account[1]<1))]
4: proc 2 (phone) line  93 "pan_in" (state 1) [self = self]
            phone(2):friend = 0
            phone(2):state = 0
            phone(2):self = 0
5: proc - (:never:) line 319 "pan_in" (state 1) [((BCP_account[1]<1))]
6: proc 2 (phone) line  94 "pan_in" (state 2) [((self==0))]
etc.
```

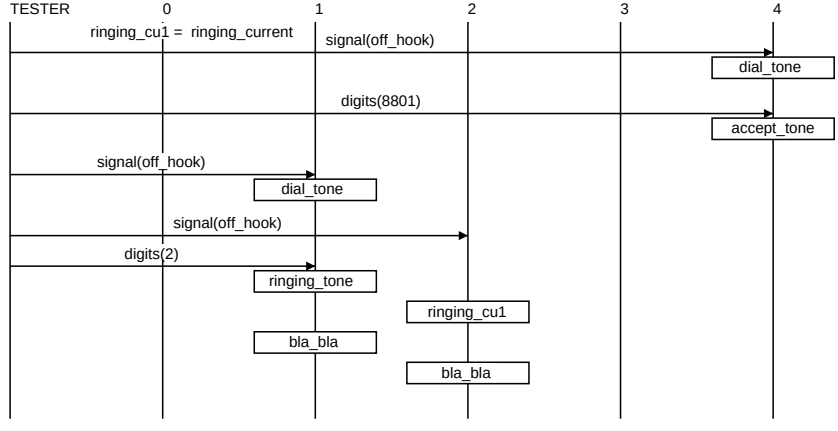



Figure 2.4: Interworking of test run after inversion

XSpin enables us to inspect this trace as an MSC, which we will not display here. We used a series of Unix shell-scripts and existing tools to transform the trace into an Interworking. To this Interworking we applied an 'inversion' [FJ96], which transforms the output lines of the various processes into input lines. This facilitates the use in a testing environment, because we can now regard them as orders to do certain actions, instead of the actions themselves. We decided to receive line-states as (observation) actions. This resulted in the Interworking shown in Figure 2.4.

Tools exist to translate this into TTCN. For the case at hand the TTCN looks as follows:

```
+-------------------------------------------------------------------------+
|Test Case FEATURE_INTERACTION_TEST 1                                     |
+-------------------------------------------------------------------------+
|Test Case Name : FIT 1                                                   |
|Group          : \1                                                      |
|Purpose        : 1st demo use SPIN FI TESTING                            |
|Default        :                                                         |
|Comments       :                                                         |
+-------------------------------------------------------------------------+
|Nr | Label | Behavior Descriptions        | Constraints Ref | Verdict     |

              4!signal(off_hook)
               [line 4 = dial_tone]                          (PASS)
                4!digits(8801)
                 [line 4 = accept_tone]                      (PASS)
                  1!signal(off_hook)
                   [line 1 = dial_tone]                      (PASS)
                    2!signal(off_hook)
                     1!digits(2)
                      [line 1 = ringing_tone]                (PASS)
                       [line 2 = ringing_current]            (PASS)
                        [line 1 = bla_bla]                   (PASS)
                         [line 2 = bla_bla]                  PASS
                         [OTHERWISE]                         FAIL
                        [OTHERWISE]                          (FAIL)
                       [OTHERWISE]                           (FAIL)
                      [OTHERWISE]                            (FAIL)
                     [OTHERWISE]                             (FAIL)
                    [OTHERWISE]                              (FAIL)
                   [OTHERWISE]                               (FAIL)     |
+-------------------------------------------------------------------------+
```

In general it is not the case that the TTCN generated from the trace in such a straightforward manner is directly correct as a test. The problem is the correct assignments of verdicts to the alternatives, all of which are made FAIL initially. We discuss three approaches to deal with this problem.

The first approach is as follows: subdivide the trace into two parts, an initial part which serves for setting-up the services and contextual connections, followed by a second part, usually much shorter, which characterises the intended behaviour of the system. For example in the test case FIT1 given above, the two observation actions [line1 = bla_bla] and [line2 = bla_bla] should be interpreted as a characterisation of the intended behaviour (call established), so the alternatives of the second part could keep the assigned FAIL verdicts. The verdicts of the alternatives of the steps of the first part can be turned into INCONCLUSIVE.

The second approach is a further refinement of this. The generated TTCN is only considered as a draft of the correct test case, which is to be obtained by checking the

verdicts and adding more alternatives (some of which may also get PASS). This is the approach of [FJ96], where it is shown in detail how a simulator is used (during multiple runs of the simulator) to find out in how far the crucial steps are deterministic, and if not, what the interesting alternative behaviours are (in the step-by-step method of [FJ96], these are the steps 13, simulate alternatives and 14, complete the TTCN description).

The third approach to the problem of verdict assignment to alternatives is to adapt the Model Checker and make it produce trees or graphs rather than sequences. This is the approach of the tool TGV [FJJV97].

So far we have only used temporal claims of a particularly simple kind, viz. invariants (such as ALWAYS(INV(1))). So we ought to discuss whether temporal claims in general are useful as well. In the classical usage of a Model Checker, i.e. for verification purposes, it will attempt to falsify claims like: "*it is always true that when the sender transmits a message, the receiver will eventually accept it*". For test generation purposes however, the temporal claim could be used to say for example: "*it is always true that when A is in state trying-to-reach-B, the SSP (Service Switching Point) will eventually connect A and B*". Of course this claim need not be true, e.g. because it is precisely the purpose of certain services (like OCS) to prevent connections from A to B to happen. Therefore, such a temporal claim, when falsified, results a trace leading to a state where this 'prevention' service has been put into operation.

## 2.4 Conclusions

Model checking can be useful as a technique for generating test traces. This can be done using existing tools, at least on prototype level. A reservation has to be made on the point of the scaling-up of the tools, because we have only tested small examples. Also the time and memory consumption of the method have not yet been investigated. If we want to use this in practice we will probably need more specialised tools, and we must be able to connect them to service-creation environments.

We found that our way of working (selecting a trace leading to an interesting state) is a promising one. This way, a part of the hard work of the creation of test traces can be automated. Traces can be selected to agree with given testing purposes without having to step down too far in abstraction.

A restriction to the applicability of the method, at least in the current form, is that the application to be tested should react deterministically to the test input. The reason for this is that otherwise a trace in which an error has occurred cannot be distinguished from one in which the internal non-determinism has caused the system to react different from the derived test trace, but still within the specification. If the system is not deterministic, the method is still useful, but in that case more manual work is needed to complete the test case. This step could of course also be automated. One possibility could be to check all supposed failure traces with the model checker again to see whether they still fit on the system. This method is described more extensively in [CSE96].

Our method supports a part of the test traject, namely the derivation of a test trace from a given test purpose. Formulating the test purpose, the stimulation process

and the model remains a task that has to be done by hand, and requires an amount of domain specific knowledge.

Another way of working might be introducing deliberate errors in the SPIN program, which are supposed to model possible errors in the design, and creating a trace in which the error occurs. We could call this 'negative testing', because in this case we are constructing traces we want the real design NOT to be able to follow, while in the constructs given until now, we wanted the design to follow the trace we gave it. An example of such a negative test is a trace that leads to a connection between two subscribers while the called party is refusing calls from the calling party by using the OCS feature.

However, we think that positive testing is more suitable to combination with the given method, because for negative testing we need to make many more assumptions about the kind of errors that might occur. In negative testing we need a rather specific idea about WHAT errors can occur, in positive testing we only need to hypothesise on WHEN they occur. When looking for specific errors, negative testing is the way to go, but if the purpose is to make a general check of a system, positive testing is much more useful.

One objection to our method could be that in order to generate a trace satisfying the property checked, the Model Checker risks searching the entire state space, which may be infeasible (the problem of the state space is often stated as an argument for the need of testing in the first place). Although this is true in principle, the important observation is that a Model Checker such as SPIN has powerful techniques built into it (such as the supertrace algorithm) to cope with the state space problem. In our opinion it is important that (if testing cannot be made superfluous by other means, for any reason whatsoever), the testers should use powerful and high-level tools as well; in particular this holds for the intermediate situation where fully automated testing is infeasible and where fully manual test generation is too costly.

## 2.5 Related Work

Several other authors have made attempts to use Model Checking for test generation. Although different methods are proposed, the basic idea is always that model checking tools are used to easily find traces to a state with some given desired properties.

In [CSE96], no complete method is given for using model checking for testing. Rather, the authors mention model checking's possibilities for the generation of test cases as well as for other aspects of testing (checking of the validity of test traces and selecting test traces among a greater number of them). The methodology that would come most closely to the ideas in this chapter would be to derive a number $n$ of boolean variables on the system state, and find traces to each of the $2^n$ possible combinations of values of these variables that can actually occur, taking these as test traces. The advantage over more random methods of test generation is that there is likely a better coverage of all aspects of the system.

The method in [ABM98] is closely related to the abovementioned idea of 'negative testing'. In these articles, so-called mutation operators create a variant of the original specification, and these mutated specifications are then compared to see whether there is a trace to make them diverge from the original specification. These traces are then

used as test traces. In [BOY00], a number of mutation operators are defined, and it is checked which one gives the best coverage when applied to an example system.

Two methods are proposed in [GH99]. The first is to use the negation of some known properties of the system. The second, in our opinion more interesting, method is to check a path to every branch of a decision. An extra variable is added, which gets a different value in every branch, and paths are found that lead to the various values of these variables.

Unlike the abovementioned articles, in [VT00] model checking is not used to rapidly find traces with a certain property. Rather, Promela is here used mostly as a specification language, and the main reason that model checking tools have been chosen in favour of other tools, is that they provide methods to store the state space efficiently.

The various methodologies have different application domains. Most of the abovementioned methods are especially adapted to medium-sized systems, where a test set that is more or less exhaustive is still possible. Our method is more applicable for large systems, where such an ideal is far out of reach, and tests have to be restricted by necessity to just a part of the system, and where it is therefore of great importance that tests are focused on those situations that are most likely to show errors.

# Chapter 3

# LOGAN: A LOG ANalysis Language

## 3.1 Introduction

Telecommunication systems are very complex, which makes testing important. Testing typically involves the design or automatic generation and selection of suitable test cases, i.e. tests that cover much of the system's behavior, the application of these tests to the system, and analysis of the test results. Ideally, the expected outcome of a test case is specified when that test case is created, so that analysis of the test results boils down to comparing the real outcome of a test case with its expected outcome.

We will examine a real life test result analysis problem which arises from a non ideal method of testing. At the Test&Release center of KPN Telecom, a representative copy of the Dutch public telephony network, called TESTNET, is used to test the execution of tariffing and call registration. Figure 3.1 shows how these tests are carried out. Firstly, a test script, which describes a certain test case, is made. This script can then be executed by a Call Generator, a system that can make calls via the TESTNET network. TESTNET produces so-called Call Data Records (CDRs). A CDR is created each time a successful call is terminated (a successful call being a call in which a connection was established between two or more subscribers). It contains information about that call which is used for tariffing, such as the subscribers that were involved and the duration of the call. Test result analysis for this type of test consists of checking the correctness of the contents of the CDRs that were produced during the test.

If the 'ideal' testing process were followed, each test script would be accompanied by the CDRs that are expected to be produced by TESTNET. These could then be compared with the CDRs that were actually produced during the test. At KPN T&R, however, no expected CDRs are specified before the execution of a test; a possible reason for this, is that system specifications, on which test prediction should be based, are missing or unclear. Instead, the CDRs are compared with other data produced during the test, namely a log file that contains the signals that the Call Generator and

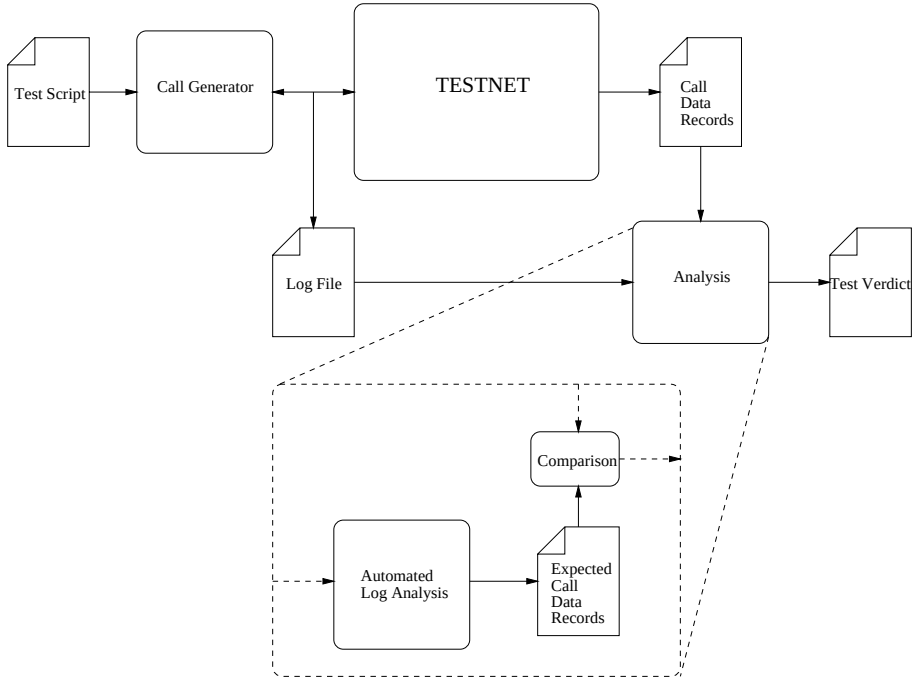



Figure 3.1: CDR testing on TESTNET, and a proposal to automate the analysis of test results

TESTNET exchanged. The analysis consists of finding signal traces of successfully terminated calls in the log file, matching these with the corresponding CDRs, and determining if the information in a CDR is correct, based on the information in the trace. This analysis is carried out manually.

Accepting the fact that real test prediction for the CDR testing process is not something that can be implemented in the near future, we will investigate the possibility of automated analysis of log files. Figure 3.1 (the dashed box) shows how this could fit in the current analysis procedure. Given a log file, the automated analysis produces 'expected' CDRs which can be compared with the real CDRs.

We will only concern ourselves with finding signal traces corresponding to successful calls, as the construction of a CDR, given a trace, should not be that difficult. What is left is a type of *pattern matching* problem, which leaves us in fact with two problems: what is a *pattern* in our case, so, what kind of traces do we want to find, and how do we do the *matching*, so, how do we actually find them?

We define a *pattern description language* that enables us to describe the properties of the traces we are interested in, and present an algorithm that, given a pattern description and a log file, finds all the traces in that log file that match the pattern.

## 3.2 Finding Call Traces in Log Files

The process that we want to automate is the search for successful calls in a log file. Figure 3.2 shows a fragment of a log file. It is a sequence of signals, where each signal is sent by a user (Call Generator) to the system (TESTNET), or received by a user from the system. The number with which each signal starts identifies the user; it is his telephone number. The form of a signal does not show if it was *sent* or *received* by a user. For anybody familiar with telephony, the name of a signal should be a clear indication of its direction. Moreover, the direction of signals is not important for our problem.

```
30:Off_Hook
30:Dial_Tone
10:Off_Hook
30:Dial(32)
10:Dial_Tone
30:Busy_Tone
10:Dial(20)
10:Ringing_Tone
30:On_Hook
20:Ringing
30:Off_Hook
20:Ringing
30:Dial_Tone
20:Off_Hook
30:Dial(20)
30:Busy_Tone
10:On_Hook
30:On_Hook
20:On_Hook
```

Figure 3.2: Fragment of a log file

In reality the signals in a log file are accompanied by time stamps, but we do not show these because they do not play a role in the problem of finding successful calls (they *do* play a role in the problem of generating CDRs for these successful calls).

The log file shown in Figure 3.2 contains one successfully terminated call, from user `10` to user `20` to be precise. In Figure 3.3(A) the same log file is depicted, but with the *call trace* of that successful call highlighted. With each successful call made during the test corresponds a sequence of signals, a call trace, in the log file that was produced. Finding successful calls means finding such traces.

### 3.2.1 Characteristic Sequences

So how do we recognise a sequence of signals as the trace of a successful call? As a first attempt, we notice that the trace of a successful call will contain a certain subsequence of events that identifies it as such. Such a subsequence we will call a *characteristic sequence*. A characteristic sequence of a successful phone call could for example be the pattern:

```
A:Dial_Tone; A:Dial(B); B:Off_Hook; (A:On_Hook or B:On_Hook);
```

The variables `A` and `B` here stand for two different subscribers. They function as parameters of the pattern, and must match phone numbers in a log file. We will thus need some kind of parametric pattern matching [Bak96].

In the pattern above, one can recognise the typical scenario of a normal phone call: the `A` party receives a dial tone, it dials the phone number of the `B` party, which responds by going off hook, and, finally, the call is terminated by one of the parties going on hook. In Figure 3.3(B) a characteristic sequence matching this description is highlighted. It identifies the trace highlighted in Figure 3.3(A) as a successful call.

(A)

```
30:Off_Hook
30:Dial_Tone
10:Off_Hook
30:Dial(32)
10:Dial_Tone
30:Busy_Tone
10:Dial(20)
10:Ringing_Tone
30:On_Hook
20:Ringing
30:Off_Hook
20:Ringing
30:Dial_Tone
20:Off_Hook
30:Dial(20)
30:Busy_Tone
10:On_Hook
30:On_Hook
20:On_Hook
```

(B)

```
30:Off_Hook
30:Dial_Tone
10:Off_Hook
30:Dial(32)
10:Dial_Tone
30:Busy_Tone
10:Dial(20)
10:Ringing_Tone
30:On_Hook
20:Ringing
30:Off_Hook
20:Ringing
30:Dial_Tone
20:Off_Hook
30:Dial(20)
30:Busy_Tone
10:On_Hook
30:On_Hook
20:On_Hook
```

Figure 3.3: (A) A call trace in a log file, (B) A characteristic sequence in a call trace

However, finding characteristic sequences is not enough to solve the problem of finding successful calls. On the one hand, they recognise too little, because certain legitimate calls will not match the above pattern. On the other hand, they recognise too much, because events that are actually unrelated might 'accidently' form a pattern like the one described above.

### 3.2.2 Problem 1: Other Call Types

In present day telephony, 'normal' phone calls are not the only calls being made. Telephony systems have been enhanced, and keep being enhanced, with all kinds of special services like *call forwarding*, *call waiting*, and *automatic ring back*. The use of such a service in a call can lead to a successful call that does not match the pattern we have given. We will illustrate this by giving an example of the use of call forwarding.

In call forwarding, a subscriber can issue the system to forward all calls made to his telephone to another telephone. He can do this by dialing the code `*21`, followed by the phone number of the new destination. We will use the signal `A:Dial*21(B)` to

denote the activation of the call forwarding service, where A is the subscriber, and B the new destination, and A:Dial#21 for the deactivation of the call forwarding feature by subscriber A.

(A)

```
⋮
10:Off_Hook
30:Dial(32)
10:Dial_Tone
30:Busy_Tone
10:Dial(20)
10:Ringing_Tone
30:On_Hook
50:Ringing
30:Off_Hook
50:Ringing
30:Dial_Tone
50:Off_Hook
30:Dial(20)
30:Busy_Tone
10:On_Hook
30:On_Hook
50:On_Hook
```

(B)

```
20:Dial*21(50)
30:Off_Hook
10:Off_Hook
30:Dial(32)
10:Dial_Tone
30:Busy_Tone
10:Dial(20)
10:Ringing_Tone
30:On_Hook
50:Ringing
30:Off_Hook
50:Ringing
30:Dial_Tone
50:Off_Hook
30:Dial(20)
30:Busy_Tone
10:On_Hook
30:On_Hook
50:On_Hook
```

Figure 3.4: (A) A call trace of a forwarded call, (B) A characteristic sequence of signals for call forwarding in a call trace

Figure 3.4(A) shows the trace of a forwarded call. Although this is a legal call, it does not contain a match for the pattern we have defined. The only difference of this trace with the normal call trace depicted in Figure 3.3(A), is the phone that answers the call, 50 instead of 20. Apparently, phone 20 has been forwarded to phone 50. If the activation of the service took place before the test was executed, there is no record of the activation in the log file and there is little hope of identifying the call trace of Figure 3.4(A) as a successful (forwarded) call. If the activation took place during the test, the log file will show this. The following pattern then seems a good candidate for identifying forwarded calls:

B:Dial*21(C); A:Dial_Tone; A:Dial(B); C:Off_Hook; A:On_Hook *or* C:On_Hook;

In Figure 3.4(B) a subsequence of the trace of a forwarded call is highlighted that matches this pattern.

This example shows that different call types require different patterns. There is however the well known problem of *feature interaction* [CV93, Mid94]. Services, also known as features, can interact with each other in a call, and it is possible that this results in a call trace that can not be recognised with any of the patterns designed for the individual services. So, a combination of services can, in a way, give rise to yet another call type, and since services can be combined in many ways, if we want to find all such calls, we will need to define a lot of patterns, maybe even an infinite number of them.

A good example of a feature interaction causing problems, is the interaction of call forwarding with itself. Using call forwarding, it is possible to make a chain of forwarded phones. A phone can be forwarded to a phone that is forwarded to yet another phone that is forwarded to..., and so on. In our forwarded calls in Figure 3.4, phone `20` was forwarded to phone `50`. Had phone number `50` been forwarded to yet another phone itself, then our call forwarding pattern would have been recognised by neither a normal call scenario nor our call forwarding scenario.

As a matter of fact, the kind of pattern matching that we want to do, which is related to string matching and sequence matching [KMP77, BM77, Wat95, DFG+97], is not really suited for finding calls that are forwarded through long forwarding chains. If one wants to find calls with any number of forwardings (or whichever other combination of (unrestrictedly) many services), only methods that go beyond merely pattern matching will be able to detect all possible combinations.

### 3.2.3 Problem 2: Coherence of Characteristic Sequences

The first problem mentioned at the end of Section 3.2.1, i.e. that of recognizing too little with our pattern, can thus be solved by defining different patterns for different call types, at least to some extent. We will now deal with the second problem mentioned at the end of Subsection 3.2.1, i.e. that of recognizing unrelated events as if they were part of a pattern.

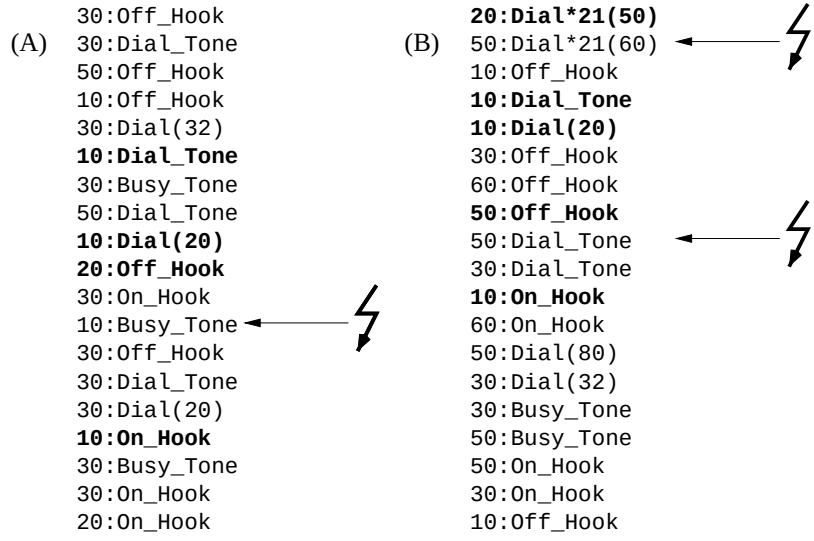



Figure 3.5: Erroneous recognition of a normal call (A), and a forwarded call (B)

Examine the log files shown in Figure 3.5. Log file (A) contains a characteristic signal sequence of a normal call, but it does *not* contain a successful normal call. The `off_hook` signal, in this case, is not the response to the incoming call but the initiation of another call. So, a number of unrelated signals are mistaken as a witness

of the presence of the call trace of some successful normal call. Something similar holds for log file (B), where a forwarded call is erroneously recognised.

We want to be able to determine whether a characteristic signal sequence in a log file really is part of one call, or is just a collection of signals from different (successful or unsuccessful) calls. Figure 3.5 shows how this could be done. The `10:Busy_Tone` signal in log file (A) indicates that the highlighted sequence can never be a witness of a normal call. Its occurrence between the `20:Off_Hook` and `10:On_Hook` signals shows that something other than a successful call from 10 to 20 is taking place. In log file (B), both the `50:Dial*21(60)` and the `50:Dial_Tone` signals indicate that the sequence indicated does not actually signify a successfully forwarded call.

So, the presence of certain signals at certain positions within the log file segment occupied by a characteristic sequence, can tell us that that characteristic sequence is not a witness of a successful call. The pattern language that we define in the next section features such signals, which we will call *negative* signals (as opposed to the *positive* signals in a characteristic sequence).

Finally, Figure 3.6 shows how the approach of using positive and negative signals relates to the system under test, TESTNET. If we regard this system as a huge state machine, the positive signals identify state transitions on some path that eventually leads to a desired final state (where a successful call (of some type) is terminated, so where a CDR should be created by TESTNET), whereas the negative signals cause state transitions that "lead away from the path".

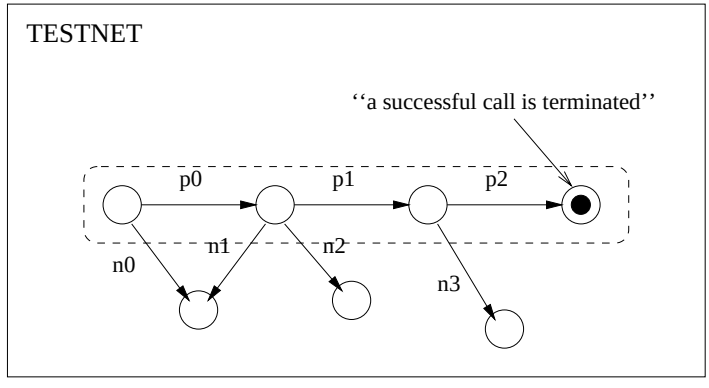



Figure 3.6: How positive and negative signals relate to the system under test

## 3.3 A Pattern Language: LOGAN

These concepts of positive and negative signals have been incorporated in a pattern description language, which we wall call *LOGAN* (LOG ANalysis). We will introduce this language first by giving an example pattern for a normal call. A formal definition will follow later in this chapter.

```
PATTERN normal_call
```

```
BEGIN
  A:Dial_Tone;
    NOT A:Busy_Tone,A:On_Hook;
  A:Dial(B);
    NOT A:Busy_Tone,A:On_Hook;
  B:Off_Hook;
    NOT A:Busy_Tone,B:Busy_Tone,A:On_Hook,B:On_Hook;
  A:On_Hook,B:On_Hook;
END
```

In this pattern we clearly recognise the specification of the characteristic sequences of normal calls that we presented earlier. Although we have already presented some specifications of characteristic sequences, we have not yet defined what kind of specifications we use. Any formalism in which a set of traces can be specified could be used, for example, regular expressions, finite state machines, grammars, Message Sequence Charts (MSC) [IT00, RGG96a], process algebra [BW90] or a large number of other formalisms. *LOGAN* uses a very simple, but because of that same reason also rather weak formalism, that of a list of signal sets. The normal call pattern above uses a list of signal sets to specify the characteristic sequences of a normal call: {`A:Dial_Tone`}; {`A:Dial(B)`}; {`B:Off_Hook`}; {`A:On_Hook`, `B:On_Hook`}. If $S_0; \ldots; S_n$ is a list of signal sets, it represents a set of traces $\{s_0; \ldots; s_n \mid \ \forall i : 0 \leq i \leq n : s_i \in S_i\}$.

What is really new in the pattern description, is the specification of negative signals. In between the positive signal sets of the pattern, sets of negative signals are specified, which are preceded with the keyword `NOT` to distinguish them from sets of positive signals. The idea is that not only the list of positive signals should 'fit', but also no negative signals should appear on the place where they are specified. For example, between the signals `A:Dial(B)` and `B:Off_Hook` the signals `A:Busy_Tone` and `A:On_Hook` may not occur. The log file in Figure 3.5A should not be considered to contain the pattern described above, because of the occurrence of a `10:Busy_Tone` signal between the `20:Off_Hook` and `10:On_Hook` signals.

The reader can easily verify that all the negative signals in this pattern are signals that, when encountered within a characteristic sequence, indicate that the sequence is not coherent. It is more difficult to see whether all possible incoherent characteristic sequences of a normal call in a log file, are indeed 'rejected' by the pattern.

Before we give the formal syntax of *LOGAN*, we will first give a second example, a pattern for call forwarding:

```
PATTERN call_forward
BEGIN
    NOT C:Dial*21(*);
  B:Dial*21(C);
    NOT B:Dial*21(*),B:Dial#21,C:Dial*21(*);
  A:Dial_Tone;
    NOT A:Busy_Tone,A:On_Hook,B:Dial*21(*),B:Dial#21,C:Dial*21(*);
  A:Dial(B);
    NOT A:Busy_Tone,A:On_Hook,B:Dial*21(*),B:Dial#21,C:Dial*21(*);
  C:Off_Hook;
```

```
    NOT A:Busy_Tone,C:Busy_Tone,A:On_Hook,C:On_Hook;
  A:On_Hook,C:On_Hook;
END
```

Again, the specification of the characteristic sequences of forwarded calls that we gave earlier, can be recognised in this pattern. There are however two new elements in this pattern, compared to the last one: A negative signal set precedes the first positive signal set, and wildcards (`*`) are used to denote *any* subscriber.

The `NOT C:Dial*21(*)` that starts off the pattern states that `C`, which is the destination to which `B` is going to forward his calls, is not forwarded itself (to anyone) before `B` forwards his calls to `C`. So, a negative signal set preceding the first positive signal set makes perfectly good sense and is very useful, as this example shows. We will not allow a negative signal set *after* the last positive signal set. Though from a 'pattern matching' point of view, there is nothing wrong with this, it does not make much sense from the 'state machine' point of view (Figure 3.6). From this point of view, we want to detect that TESTNET has reached some state where a successful call is terminated. If we are not already in such a state we can only get there if *something happens*, not if *something will not happen in the future*. From a practical point of view such a restriction seems reasonable: we do not want to have to wait indefinitely long in the future before deciding whether or not something is a valid call.

### 3.3.1 Syntax of LOGAN

The syntax of *LOGAN* is given in Table 3.1, in the form of a context-free grammar. The grammar is rather straightforward. The two *LOGAN* pattern examples we have given cover most of the language, so the grammar does not reveal anything radically new. Worth mentioning perhaps is that signals can have an arbitrary number of arguments, as is expressed by the rules for *ACT* and *ARGs*, and that concrete telephone numbers can be used wherever a variable or wildcard can be used, as is expressed by the rule for *SUB*.

| | | |
|---|---|---|
| *PAT* | ::= | `PATTERN` *NAM* `BEGIN` *BOD* `END` |
| *NAM* | ::= | $[\texttt{a}\ldots\texttt{z},\texttt{A}\ldots\texttt{Z},\texttt{0}\ldots\texttt{9},\texttt{\_}]^+$ |
| *BOD* | ::= | $[NEG\ \ POS]^*$ |
| *NEG* | ::= | $\varepsilon$ \| `NOT` *SIGs* `;` |
| *POS* | ::= | *SIGs* `;` |
| *SIGs* | ::= | *SIG* \| *SIG*`,`*SIGs* |
| *SIG* | ::= | *SUB*`:`*ACT* |
| *SUB* | ::= | *ID* \| `*` \| $[\texttt{0}\ldots\texttt{9}]^+$ |
| *ID* | ::= | $\texttt{A}\ldots\texttt{Z}$ |
| *ACT* | ::= | *SIGNAM* \| *SIGNAM*`(`*ARGs*`)` |
| *SIGNAM* | ::= | $[\texttt{a}\ldots\texttt{z},\texttt{A}\ldots\texttt{Z},\texttt{0}\ldots\texttt{9},\texttt{\_},*,\#\ ]^+$ |
| *ARGs* | ::= | *SUB* \| *SUB*`,`*ARGs* |

Table 3.1: The syntax of *LOGAN*

### 3.3.2 Tabular Form

For practical purposes, we propose an alternative notation for *LOGAN* patterns which we will call tabular form. Here is the tabular form of `normal_call`:

| normal_call | | |
|---|---|---|
| | Busy_Tone | On_Hook |
| A:Dial_Tone | A | A |
| A:Dial(B) | A | A |
| B:Off_Hook | A,B | A,B |
| A:On_Hook,B:On_Hook | | |

The correspondence between textual and tabular form should not be hard to grasp. The name of the pattern is in the upper left field of the table. The positive signal sets are all in the first column. The negative signal sets are represented by the second, third, etc. columns. Each negative signal is split in its 'subject', i.e. the receiving or sending subscriber, and the name of the signal. So the `A` in the third row and second column of the table means that `A:Busy_Tone` may not occur between `A:Dial_Tone` and `A:Dial(B)`. We feel that, with its two-dimensional representation of patterns, the tabular format provides a more user friendly way of writing and reading patterns. The main reason for this is that subsequent negative signal sets often contain the same signals, and this property is readily apparent from the tabular format. If this was not a property of patterns, then the tabular format would probably be much less readable.

The following example in tabular format is a pattern describing a successful call in which the Call Waiting Hookflash service is activated. This means that while subscriber `B` is connected to subscriber `A`, a third person, say `C`, can call `B`. `B` will then hear a soft warning tone, and when `B` hookflashes, `A` will be put 'on hold' and `B` and `C` can talk. `B` can switch many times between `A` and `C` by hookflashing. The example describes the situation where `B` switches once from `A` to `C` and after termination of the call with `C`, switches back to `A`. Of course, many other call waiting scenarios are possible. In order to find these, we would have to write other patterns, or, better, find *one* pattern that captures the essence of all, or at least a lot of, call waiting scenarios. We will come back to this issue in Sections 3.7 and 3.8.

| call_w_hkflsh | | |
|---|---|---|
| | Busy_Tone | On_Hook |
| A:Dial_Tone | A | A |
| A:Dial(B) | A | A |
| B:Off_Hook | A,B | A,B |
| C:Dial_Tone | A,B,C | A,B,C |
| C:Dial(B) | A,B,C | A,B,C |
| B:Warning_Tone | A,B,C | A,B,C |
| B:Hookflash | A,B,C | A,B,C |
| A:Hold_Tone | A,B,C | A,B,C |
| C:On_Hook | A,B | A,B |
| B:Hookflash | A,B | A,B |
| A:On_Hook,B:On_Hook | | |

Note, that the normal call pattern is present in this call waiting hookflash pattern (rows 3, 4, 5 and 13). The first leg of the call, i.e. the conversation between `A` and `B` will therefore also be detected by the normal call pattern, but this is not the case for the second leg, i.e. the conversation between `C` and `B`. This second leg does not match the normal call pattern because `B` responds to the incoming call with a `Hookflash` instead of an `Off_Hook`.

## 3.4 Formal Semantics of LOGAN

We will know proceed to a formal semantics of *LOGAN*. In the preceding sections we have more than once used the term *witness of a pattern*, meaning a sequence of signals in a log file that indicates the presence of the pattern. This term will be central in our definition of a formal semantics. However, before giving a formal definition of the witness concept for *LOGAN* patterns, we will first give a mathematical description of *LOGAN* patterns.

**Definition 3.4.1 (*LOGAN* pattern)** With a *LOGAN* pattern, containing $k$ positive signal sets, we associate a pair $(P, N)$, where $P = (P_0, P_1, \ldots, P_{k-1})$ is a list of non-empty sets of signals, and $N = (N_0, N_1, \ldots, N_{k-1})$ is a list of, possibly empty, sets of signals. For all $0 \leq i < k$, $P_i$ contains the signals of the $i$-th positive signal set of the *LOGAN* pattern, and $N_i$ contains the signals of the negative signal set that precedes $P_i$. In the remainder of this chapter we will simply call such $(P, N)$ pairs *LOGAN* patterns.

In order to match a *LOGAN* pattern with a signal sequence in a log file we have to establish a relation between the variables in that pattern, which represent telephone numbers, and the actual telephone numbers in the log file. For this, we will use *valuations*.

**Definition 3.4.2 (Valuation)** Let $(P, N)$ be a *LOGAN* pattern. A valuation for this pattern is a partial injection $v : \mathit{vars}(P) \nrightarrow \mathit{Ext}$, where *Ext* is the set of extensions,

i.e. telephone numbers, and $vars(P)$ denotes the set of variables that appear in the positive signal sets of the pattern.

So, a valuation assigns telephone numbers to variables in a pattern. The fact that valuations are injections implies that different variables in a pattern represent different telephone numbers. They are partial functions, because in order to match a pattern with a particular signal sequence in a log file not all variables in the positive signal sets need to have an assignment.

We now define how variables and wildcards in a pattern can be replaced by actual telephone numbers, which will allow us to match actual signals from a log file with signal sets of a pattern (which may contain variables and wildcards).

**Definition 3.4.3 (Substitution and matching)** Let $(P, N)$ be a *LOGAN* pattern, and $v$ a valuation for that pattern. Let $S$ be some signal set in $(P, N)$ (positive or negative). Applying the valuation $v$ to the signal set $S$ yields a signal set $v(S)$ obtained from $S$ by substituting extensions for variables as is prescribed by $v$ and 'expanding' all wildcards.

By expanding, we mean that all possible substitutions of extensions for wildcards are included in $v(S)$. Here is an example of a substitution, where valuation $v = \{\texttt{A} \mapsto \texttt{1024}, \texttt{D} \mapsto \texttt{1060}\}$.

$$v(\{\texttt{A:Dial(B)}, \texttt{D:Dial(*)}\}) = \{\texttt{1024:Dial(B)}, \texttt{1060:Dial(}e\texttt{)} \mid e \in \mathit{Ext}\}$$

So, given the valuation $v$, `1060:Dial(1914)` matches $\{\texttt{A:Dial(B)}, \texttt{D:Dial(*)}\}$, because $\texttt{1060:Dial(1648)} \in v(\{\texttt{A:Dial(B)}, \texttt{D:Dial(*)}\})$. Note, that under the valuation $v$, `1024:Dial(1918)` does *not* match this signal set (although we can easily extend the valuation so that it does).

Now that we have a convenient mathematical notation for *LOGAN* patterns, valuations that assign extensions to the variables in a pattern, and the notion of signals matching signal sets (given a valuation), we are able to give a formal definition of a witness of a *LOGAN* pattern.

**Definition 3.4.4 (Witness)** Let $(P, N) = ((P_0, \ldots, P_k), (N_0, \ldots, N_k))$ be a *LOGAN* pattern, and $L = [s_0, \ldots, s_n]$ a log file. A *witness* of $(P, N)$ on $L$ is a pair $(f, v)$, where $f : \{0, \ldots, k\} \to \{0, \ldots, n\}$ is a so-called *witness function*, and $v : vars(P) \nrightarrow \mathit{Ext}$ is a valuation, such that for all $i$ $(0 \leq i \leq k)$, $j$ $(0 \leq j \leq k)$ and $x$ $(0 \leq x \leq n)$:

1. $i < j \Rightarrow f(i) < f(j)$
2. $s_{f(i)} \in v(P_i)$
3. $minimal(v, f, P)$
4. $f(i-1) < x < f(i) \Rightarrow s_x \notin v(N_i)$ (define $f(-1) = -1$).

where $minimal(v, f, P) \equiv (\forall\, w \subset v :: \neg(\forall\, i : 0 \leq i \leq k : s_{f(i)} \in w(P_i)))$. The demand that a valuation is "minimal" assures that all the variable assignments in the valuation are necessary. As shorthand for "$(f, v)$ is a witness of $(P, N)$ on $L$" we will use $(f, v) : L \models (P, N)$.

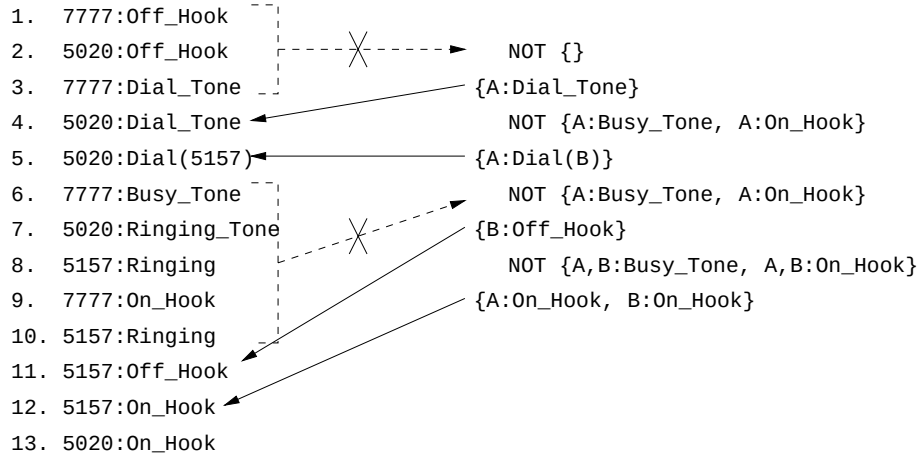



Figure 3.7: a witness with valuation $\{\mathtt{A} \mapsto \mathtt{5020}, \mathtt{B} \mapsto \mathtt{5157}\}$

Figure 3.7 depicts witness $(\{0 \mapsto 4, 1 \mapsto 5, 2 \mapsto 11, 3 \mapsto 12\}, \{\mathtt{A} \mapsto \mathtt{5020}, \mathtt{B} \mapsto \mathtt{5157}\})$. Because the domain of a witness function is a finite initial segment of the natural numbers, we identify witness functions with lists over the natural numbers. We can say that Figure 3.7 depicts witness $(\langle 4, 5, 11, 12\rangle, \{\mathtt{A} \mapsto \mathtt{5020}, \mathtt{B} \mapsto \mathtt{5157}\})$ and in the sequel we will also use list operators on witness functions, yielding expressions like $|f|$ (length of a list), $tail(f)$ (tail of a list), and $f +\!\!+ g$ (concatenation of two lists).

## 3.5 Algorithm

In this section we will show an algorithm to find all the witnesses of a *LOGAN* pattern in a log file. We will first look at an algorithm for a subset of *LOGAN* which we will call $LOGAN_c$. $LOGAN_c$ is equivalent to *LOGAN* except for not using variables and wildcards. Thus, syntactically, $LOGAN_c$ is like *LOGAN* except for the rule $SUB ::= ID \mid * \mid [\mathtt{0} \ldots \mathtt{9}]^+$, which is replaced by the rule $SUB ::= [\mathtt{0} \ldots \mathtt{9}]^+$. On this subset we can construct a basic algorithm for sequences matching *LOGAN*-style 'pos-neg' patterns, and leave the extra complication of variable substitution for later.

Because $LOGAN_c$ patterns do not contain variables and wildcards, valuations do not play a role in finding witnesses. All witnesses of such a pattern are of the form $(f, \emptyset)$. A witness for a $LOGAN_c$ pattern is actually just a witness function, and we will therefore write $f : L \models (P, N)$, thereby meaning $(f, \emptyset) : L \models (P, N)$.

The pre-condition and post-condition for our algorithm will be:

**con** $L$: log file, $(P, N)$: a $LOGAN_c$ pattern
**var** $F$: set of witness functions

**PRE**: $L = [s_0, \ldots, s_n] \ \wedge \ (P, N) = ((P_0, \ldots, P_k), (N_0, \ldots, N_k))$
**POST**: $F = \{f \mid f : L \models (P, N)\}$

Basically, the algorithm works as follows: it traverses the log file and maintains the set of witness functions of *prefixes* of the pattern on the part of the log file scanned so far. We will use the following notation to denote pattern prefixes: $(P,N) \restriction m = ((P_0,\ldots,P_{m-1}),(N_0,\ldots,N_{m-1}))$. We will actually design an algorithm that satisfies the following post-condition:

**POST'**: $F = \{f \mid f : L \models (P,N) \restriction |f|\}$

The intended post-condition **POST** can easily be reached from **POST'** by the rule $F := \{f \mid f \in F \wedge |f| = k+1\}$.

We introduce variables $l_1$ and $l_2$, representing the part of $L$ already processed and the part that still has to be done, respectively. The following invariants will hold during the application of the algorithm:

**var** $l_1, l_2$ : log file

**INV$_0$**: $l_1 +\!\!+ l_2 = L$
**INV$_1$**: $F = \{f \mid f : l_1 \models (P,N) \restriction |f| \wedge \mathit{extra}(f, l_1, N)\}$

where $\mathit{extra}(f,l,N) \equiv |f| = k+1 \ \vee \ (\forall\, j : f(|f|-1) < j < |l| : \ l(j) \notin N_{|f|})$.

The $\mathit{extra}(f,l_1,N)$ clause in **INV$_1$** is an extension of clause (4) of definition 3.4.4, and it expresses that for all witness functions $f \in F$ that are not complete witnesses yet, the part of the log file processed after the last signal witnessed by $f$, may not contain signals in the 'lookahead' negative signal set $N_{|f|}$.

For $l_2 = \langle\rangle$ the post-condition follows from these invariants, so $l_2 \neq \langle\rangle$ will be a suitable guard for the algorithm. Because the empty pattern is the only pattern that matches the empty log file, and the empty witness function the only corresponding witness, $F := \{\langle\rangle\}$, $l_1 := \langle\rangle$, and $l_2 := L$ will do as initialization.

Now, for within the repetition of the algorithm, we have to find assignments to $F$, $l_1$ and $l_2$, that satisfy the invariants and assure termination of the algorithm. Heading for termination we choose $l_1 := l_1 +\!\!+ \langle l_2(0)\rangle$ and $l_2 := \mathit{tail}(l_2)$, which leaves us with the task of computing $\{f \mid l_1 +\!\!+ \langle l_2(0)\rangle \models (P,N) \restriction |f| \wedge \mathit{extra}(f, l_1 +\!\!+ \langle l_2(0)\rangle, N)\}$. By splitting this set in two sets, one containing the witness functions $f$ that refer to the $|l_1|$-th element of the log file (i.e. $l_2(0)$, the signal currently being inspected), and the other one containing the ones that do not, we can derive the following equivalent expression:

$$\begin{array}{c}\{f +\!\!+ \langle |l_1| \rangle \mid f \in F \wedge \ |f| \leq k \ \wedge \ l_2(0) \in P_{|f|}\} \cup \\ \{f \mid f \in F \ \wedge \ (|f| = k+1 \ \vee \ l_2(0) \notin N_{|f|})\}\end{array}$$

The first set of this union shows how witness functions in $F$ can be extended, with a reference to the signal under inspection. The second set expresses that a witness function in $F$ remains in $F$ unless it is an incomplete witness and the signal under inspection is in the 'lookahead' negative signal set. Here is the complete algorithm:

$l_1 := \langle\rangle$; $l_2 := L$; $F := \{\langle\rangle\}$;
**while** $l_2 \neq \langle\rangle$ **do**
$\quad F := \{f +\!\!+ \langle |l_1| \rangle \mid f \in F \wedge \ |f| \leq k \wedge \ l_2(0) \in P_{|f|}\} \cup$

$\{f \mid f \in F \wedge\ (|f| = k+1 \vee\ l_2(0) \notin N_{|f|})\};$
$l_1 := l_1 +\!\!+ \langle l_2(0) \rangle;\ l_2 := tail(l_2)$
**od**

For practical usage it is better to transform this algorithm into a more readable form, where a number of set operations have been eliminated. The assignment to $F$ is replaced by a sequence of assignments for each element of the set in a straightforward manner. We also get rid of the two lists $l_1$ and $l_2$ in the representation of the algorithm, instead introducing an integer variable $m = |l_1|$.

$F := \{\langle\rangle\};$
**for** $m = 0$ **to** $n$
  $G := \emptyset;$
  **for each** $f \in F$
    $l := |f|;$
    **if** $l \leq k$ **cand** $s_m \in P_l$ **then** $G := G \cup \{f +\!\!+ \langle m \rangle\};$
    **if** $l = k+1$ **cor** $s_m \notin N_l$ **then** $G := G \cup \{f\}$
  **next**
  $F := G$
**next**

## 3.6 Variable Substitution

We now get to the problem of adding variables and wildcards to the algorithm derived in the previous section. Extending the algorithm so that it can handle wildcards is actually quite easy. When we defined substitution in Section 3.4, we introduced the notion of expanding a signal set to get rid of wildcards. If we apply this expansion to the signal sets present in the algorithm, we have an algorithm that also works for patterns with wildcards (In an implementation of this algorithm the 'expansion' will, of course, have to be implemented by a simple pattern matching procedure, and not an actual expansion procedure). Extending the algorithm so that it can handle variables is more difficult. The major difficulty, as we will see, is that before a value is assigned to a variable in order to match a signal from the log file to a signal in a positive signal set, a negative signal set can already have imposed some restrictions on the values that the variable may attain.

First, we give a specification for the algorithm. It is basically the specification of the previous algorithm, but with witness functions replaced by complete witnesses, i.e. witness functions and valuations.

**con** $L$: log file, $(P, N)$: a *LOGAN* pattern
**var** $F$: set of witnesses

**PRE**: $L = [s_0, \ldots, s_n] \ \wedge\ (P, N) = ((P_0, \ldots, P_k), (N_0, \ldots, N_k))$
**POST**: $F = \{(f, v) \mid (f, v) : L \models (P, N)\}$

We will have to get the valuations into the algorithm somehow. At first glance, this does not seem to be such a big problem. Just pair the empty witness function in

the initialization with the empty valuation, and extend a valuation with the proper variable assignments if thereby we can match a signal from the log file with a signal in a positive signal set.

### 3.6.1 Constraints

Consider scanning a log file containing `1080:Dial_Tone;1080:Busy_Tone` for the normal call pattern. The first signal matches the first positive signal of this pattern if we choose the valuation $\{\texttt{A} \mapsto \texttt{1080}\}$. The second signal, however, matches the next negative signal set if this valuation is applied to it. This means that we have to abandon the witness that we are constructing. It is no longer valid.

Now consider scanning a log file containing `1080:Dial*21(1030);...;1060:Dial*21(1080)` for the call forwarding pattern. The last signal matches the first positive signal of the pattern if we choose valuation $\{\texttt{B} \mapsto \texttt{1060}, \texttt{C} \mapsto \texttt{1080}\}$. Then, however, we have a conflict with the first negative signal set of the call forwarding pattern, the one preceding the first positive signal. This set contains `C:Dial*21(*)` and the first signal in the log file fragment matches this signal set given our valuation. This means that, again, we have to abandon the witness we are constructing. This example shows that we somehow have to remember that valuation $\{\texttt{C} \mapsto \texttt{1080}\}$ is no longer allowed after we have encountered the `1080:Dial*21(1030)` signal in the log file. Besides the *positive* information about the values of the variables, i.e. the valuations, we also have to keep track of *negative* information about the values of variables.

With the help of a (probably quite exotic) fragment of a pattern we will explain how we can use *constraints* as carriers for the negative information. Here is the example:

```
    NOT A:dial*21(B), B:dial*21(A);
  ...
    NOT C:dial(B), A:dial(*);
  ...
```

Suppose that we look for this pattern in a log file, and that the witness we are constructing demands that the first negative signal set may not contain `1080:Dial*21(1060)`, while `1024:Dial(1050)` may not be contained in the second one. The valuations that become forbidden because of the first signal matching the first negative signal set can be characterised by the following formula of propositional logic:

$$(\texttt{A} = \texttt{1080} \ \wedge \ \texttt{B} = \texttt{1060}) \ \vee \ (\texttt{B} = \texttt{1080} \ \wedge \ \texttt{A} = \texttt{1060})$$

So, after we have scanned the first signal (future) valuations have to satisfy the following *constraint*:

$$\neg((\texttt{A} = \texttt{1080} \ \wedge \ \texttt{B} = \texttt{1060}) \ \vee \ (\texttt{B} = \texttt{1080} \ \wedge \ \texttt{A} = \texttt{1060}))$$

We can transform this proposition into an equivalent one which is in *Conjunctive Normal Form* (i.e. written as a conjunction of disjunctions):

$$(\texttt{A} \neq \texttt{1080} \ \vee \ \texttt{B} \neq \texttt{1060}) \ \wedge \ (\texttt{B} \neq \texttt{1080} \ \vee \ \texttt{A} \neq \texttt{1060})$$

We can do the same for the second signal and the second negative signal set. Matching these yields the following constraint (in CNF):

$$(\texttt{C} \neq 1024 \ \vee \ \texttt{B} \neq 1050) \ \wedge \ \texttt{A} \neq 1024$$

Given a signal and a negative signal set, we can always produce a logical formula in conjunctive normal form that characterises the forbidden valuations. If the signal and the signal set do not match we get the formula *true* (*true* and *false* are both considered to be in CNF). If the signal matches with one or more signals in the set, then, for each of these matches, there is a (smallest) valuation that establishes this match and that can be characterised by a formula of the form $(X_0 = e_0 \wedge \ldots \wedge X_n = e_n)$. The *set* of (smallest) valuations that cause a match of the signal and the signal set can then be characterised by the disjunction of all these formulas. We turn the formula we then get into a constraint by placing a negation in front of it, and transforming it into a formula in CNF using De Morgan's laws ($\neg(A \wedge B) = \neg A \vee \neg B$ and $\neg(A \vee B) = \neg A \wedge \neg B$).

Let us return to the example. We said that the witness under construction demands that the first negative signal set may not contain `1080:Dial*21(1060)`, and that the second one may not contain `1024:Dial(1050)`. For each of these demands we have constructed a constraint. The valuation of the witness must satisfy both constraints, or in other words, it must satisfy their *conjunction*, in this case $(\texttt{A} \neq 1080 \vee \texttt{B} \neq 1060) \wedge (\texttt{B} \neq 1080 \vee \texttt{A} \neq 1060) \wedge (\texttt{C} \neq 1024 \vee \texttt{B} \neq 1050) \wedge \texttt{A} \neq 1024$.

The example has shown us that we can impose restrictions on the valuation we associate with a witness, by also associating a *constraint* with it, which is a logical formula in CNF that expresses which assignments to variables are forbidden (and which not).

**Definition 3.6.1 (Constraints)** Let $V$ be a set of variables. The set of all constraints over $V$ is denoted by $\mathit{Prop}(V)$, and it consists of all $C$ formed according to the following BNF rules:

$$\begin{array}{lcl} C & ::= & (C \ \underline{\wedge} \ C) \ \mid \ D \\ D & ::= & \underline{\textbf{true}} \ \mid \ \underline{\textbf{false}} \ \mid \ (D \ \underline{\vee} \ D) \ \mid \ \underline{X \neq e} \text{ for some } X, e. \end{array}$$

We have used underlining to emphasize the fact that we are dealing with syntactic categories. A constraint is *not* a boolean expression, it *represents* one.

Our definition of constraints permits that the values **true** and **false** occur in a constraint. This has been done with the evaluation of constraints (or of parts of a constraint) in mind. The next two definitions concern this evaluation of constraints.

**Definition 3.6.2 (Constraints and valuations)** Using the recursive structure of constraints, we define how a valuation $v$ is applied to a constraint, producing another constraint:

- $v(C_1 \ \underline{\wedge} \ C_2) = v(C_1) \ \underline{\wedge} \ v(C_2)$
- $v(D_1 \ \underline{\vee} \ D_2) = v(D_1) \ \underline{\vee} \ v(D_2)$
- $v(\underline{\textbf{true}}) = \underline{\textbf{true}}$

- $v(\underline{\textbf{false}}) = \underline{\textbf{false}}$
- $v(\underline{X \neq e}) = \begin{cases} \underline{X \neq e} & \text{if } X \notin dom(v) \wedge \forall Y \in dom(v) : v(Y) \neq e \\ \underline{\textbf{true}} & \text{if } X \in dom(v) \wedge v(X) \neq e, \\ \underline{\textbf{false}} & \text{if } X \in dom(v) \wedge v(X) = e, \\ \underline{\textbf{true}} & \text{if for some } Y \in dom(v), Y \neq X \wedge v(Y) = e \end{cases}$

Applying a valuation $v$ to a constraint $C$ yields a constraint equal to $C$ in structure, but with the $\underline{X \neq e}$ clauses replaced by $\underline{\textbf{true}}$'s and $\underline{\textbf{false}}$'s where the valuation permits it. Note, that we use the fact that $v$ is an *injection*, i.e. $v(X) \neq v(Y)$ for $X \neq Y$. By applying a valuation to a constraint we can check whether that valuation satisfies the constraint or not.

Intuitively, we feel that two constraints have the same *meaning* if the logical propositions they represent are equivalent. Consequently, we can simplify a constraint using the rules of propositional logic without changing its meaning.

**Definition 3.6.3 (Simplification)** Let $V$ be a set of variables. The function $simp : Prop(V) \rightarrow Prop(V)$ simplifies a constraint, using the rules of logic for interaction of *true* and *false* with $\wedge$ and $\vee$. Following the recursive structure of constraints we define *simp* as follows:

- $simp(\underline{\textbf{true}}) = \underline{\textbf{true}}$, $simp(\underline{\textbf{false}}) = \underline{\textbf{false}}$, and $simp(\underline{X \neq e}) = \underline{X \neq e}$.
- $simp(C_1 \underline{\wedge} C_2) =$
  $\begin{cases} \underline{\textbf{false}} & \text{if } simp(C_1) = \underline{\textbf{false}} \text{ or } simp(C_2) = \underline{\textbf{false}}, \\ \underline{\textbf{true}} & \text{if } simp(C_1) = simp(C_2) = \underline{\textbf{true}}, \\ simp(C_1) & \text{if } simp(C_1) \notin \{\underline{\textbf{true}}, \underline{\textbf{false}}\},\ simp(C_2) = \underline{\textbf{true}}, \\ simp(C_2) & \text{if } simp(C_1) = \underline{\textbf{true}},\ simp(C_2) \notin \{\underline{\textbf{true}}, \underline{\textbf{false}}\}, \\ simp(C_1) \underline{\wedge} simp(C_2) & \text{if } simp(C_1), simp(C_2) \notin \{\underline{\textbf{true}}, \underline{\textbf{false}}\} \end{cases}$
- $simp(D_1 \underline{\vee} D_2) =$
  $\begin{cases} \underline{\textbf{true}} & \text{if } simp(D_1) = \underline{\textbf{true}} \text{ or } simp(D_2) = \underline{\textbf{true}}, \\ \underline{\textbf{false}} & \text{if } simp(D_1) = simp(D_2) = \underline{\textbf{false}}, \\ simp(D_1) & \text{if } simp(D_1) \notin \{\underline{\textbf{true}}, \underline{\textbf{false}}\},\ simp(D_2) = \underline{\textbf{false}}, \\ simp(D_2) & \text{if } simp(D_1) = \underline{\textbf{false}},\ simp(D_2) \notin \{\underline{\textbf{true}}, \underline{\textbf{false}}\}, \\ simp(D_1) \underline{\vee} simp(D_2) & \text{if } simp(D_1), simp(D_2) \notin \{\underline{\textbf{true}}, \underline{\textbf{false}}\} \end{cases}$

It is quite clear from the definition of *simp* that if it is applied on a constraint $C$ it yields a constraint $C'$ with the same meaning, with the additional property that either $C' = \underline{\textbf{true}}$ or $C' = \underline{\textbf{false}}$ or $C'$ does not contain the constants $\underline{\textbf{true}}$ and $\underline{\textbf{false}}$ at all.

With the help of the concept of applying valuations to constraints and the concept of simplification we define a formal semantics for constraints, based on valuations.

**Definition 3.6.4 (Constraint semantics)** Given a set $V$ of variables, and $C : \mathit{Prop}(V)$ a constraint on those variables, we define the semantics $[\![\ ]\!]$ of $C$ with respect to $V$ as follows:

$$[\![C]\!]_V = \{v : V \to \mathit{Ext} \mid \mathit{simp}(v(C)) \neq \underline{\mathbf{false}}\}$$

So, the semantics of a constraint is the set of valuations that do not falsify the constraint.

We say that valuation $v$ *satisfies* constraint $C$ iff $v \in [\![C]\!]_V$, and that constraint $C_1$ is *weaker* than constraint $C_2$ iff $[\![C_2]\!]_V \subseteq [\![C_1]\!]_V$.

### 3.6.2 An Algorithm with Variable Substitution

Armed with the concept of constraints, we can now extend the algorithm of Section 3.5 so that it can handle all *LOGAN* patterns. To this end we give a definition of a witness that includes constraints. This definition sheds some light on how the valuation of a (partial) witness may be extended. It is only with the algorithm and the construction of witnesses in mind that this definition makes any sense. As a means to explain the witness concept it would be correct but also quite absurd.

**Definition 3.6.5 ('Constrained' Witnesses)** Given a *LOGAN* pattern $(P, N) = ((P_0, \ldots, P_k), (N_0, \ldots, N_k))$, and a log file (prefix) $L = [s_0, \ldots, s_n]$, a *constrained witness* of $(P, N)$ on $L$ is a 3-tuple $(f, v, C)$, where $f : \{0, \ldots, k\} \to \{0, \ldots, n\}$ is a witness function, $v : \mathit{vars}(P) \nrightarrow \mathit{Ext}$ a valuation, and $C : \mathit{Prop}(\mathit{vars}(N))$ a constraint, such that for all $i$ $(0 \leq i \leq k)$, $j$ $(0 \leq j \leq k)$, $x$ $(0 \leq x \leq n)$, and $w$ $(v \subseteq w)$:

1. $i < j \Rightarrow f(i) < f(j)$
2. $s_{f(i)} \in v(P_i)$
3. $\mathit{minimal}(v, f, P)$
4. $\mathit{simp}(v(C)) \neq \underline{\mathbf{false}}$
5. $f(i-1) < x < f(i) \Rightarrow (s_x \in w(N_i) \Rightarrow \mathit{simp}(w(C)) = \underline{\mathbf{false}})$ (define $f(-1) = -1$).
6. $\mathit{weakest}(C, f, v, N)$

where $\mathit{weakest}(C, f, v, N) \equiv (\forall\, C' : C'\text{satisfies clauses (4) and (5)} : [\![C']\!] \subseteq [\![C]\!])$. Clause (5) takes the extension of valuations into account by stating that constraint $C$ must prohibit certain extensions $w$ of valuation $v$.

As shorthand for "$(f, v, C)$ is a constrained witness of $(P, N)$ on $L$" we will use $(f, v, C) : L \models (P, N)$. Note, that clauses (4) and (5) imply clause (4) from Definition 3.4.4 (substitute $v$ for $w$ in clause (5)), so we have that $(f, v, C) : L \models (P, N) \Rightarrow (f, v) : L \models (P, N)$. We also have $(f, v) : L \models (P, N) \Rightarrow (\exists\, C :: (f, v, C) : L \models (P, N))$. Because of these two implications (soundness and completeness) we can safely compute constrained witnesses instead of ordinary witnesses.

So, a constrained witness is just an ordinary witness with some extra information that says something about how the witness may be extended. We can therefore give the following specification, which is more or less equivalent to the one given earlier in this section:

**con** $L$: log file, $(P, N)$: a *LOGAN* pattern
**var** $F$: set of constrained witnesses

**PRE**: $L = [s_0, \ldots, s_n] \ \wedge \ (P, N) = ((P_0, \ldots, P_k), (N_0, \ldots, N_k))$
**POST**: $F = \{(f, v, C) \mid (f, v, C) : L \models (P, N)\}$

We replace the post-condition by the following post-condition (like we did when we derived the first algorithm):

**POST'**: $F = \{(f, v, C) \mid (f, v, C) : L \models (P, N) \uparrow |f|\}$

This post-condition gives rise to the following invariants, analogous to the invariants we had for the first algorithm:

**INV$_0$**: $l_1 +\!\!+ l_2 = L$
**INV$_1$**: $F = \{(f, v, C) \mid (f, v, C) : l_1 \models (P, N) \uparrow |f| \wedge \ extra(f, v, C, l_1, N)\}$

where

$$extra(f, v, C, l, N) \equiv (\forall\, j : f(|f| - 1) < j < |l| : (\forall\, w : v \subset w : l(j) \in w(N_{|f|}) \Rightarrow simp(w(C)) = \underline{\textbf{false}})) \quad (\text{define } N_{k+1} = \emptyset)$$

The algorithm derived with these invariants looks a lot like the algorithm of Section 3.5. Here it is (using the more practical format immediately):

$F := \{(\langle\rangle, \emptyset, \underline{\textbf{true}})\}$;
**for** $m = 0$ **to** $n$
  $G := \emptyset$;
  **for each** $(f, v, C) \in F$
    $l := |f|$;
    **if** $l \leq k$ **then**
      **for each** $w : minext(w, v, s_m, P_l)$
        $C' := simp(w(C))$;
        **if** $C' \neq \underline{\textbf{false}}$ **then** $G := G \cup \{(f +\!\!+ \langle m \rangle, w, C')\}$
      **next**;
    **if** $l = k + 1$ **then** $G := G \cup \{(f, v, C)\}$
    **else**
      $C' := simp(v(C \ \underline{\triangle} \ constraint(s_m, N_l)))$;
      **if** $C' \neq \underline{\textbf{false}}$ **then** $G := G \cup \{(f, v, C')\}$
  **next**
  $F := G$
**next**

Let us examine how this algorithm differs from the old one. In the initialization the empty witness function has been replaced by a 3-tuple consisting of the empty witness function, the empty valuation, and the constraint **true**. Note, that the empty valuation is a minimal valuation, and that **true** is the weakest constraint possible. Definition 3.6.5 requires this.

Next, we see that the **if** $l \leq k$ **cand** $s_m \in P_l$ **then** $G := G \cup \{f +\!\!+ \langle m \rangle\}$ statement of the original algorithm has been replaced by a repetition. In the original statement, the witness function $f$ was extended if signal $s_m$ was in signal set $P_l$. Here we consider all valuations $w$ that are minimal extensions of valuation $v$ that make $s_m$ match $P_l$:

$$minext(w, v, s, P) \equiv v \subseteq w \ \wedge \ s \in w(P) \ \wedge \ (\forall\, w' : v \subseteq w' \subset w : s \notin w'(P))$$

It is then checked if the extended valuation $w$ satisfies the constraint we have, and if this is the case $s_m$ and $P_l$ really match, and the witness function $f$ can be extended. The valuation and constraint associated with this extended $f$, are extended valuation $w$ and constraint $C'$, the result of applying $w$ to $C$ and simplification. Note, that we could use $C$ instead of $C'$. These constraints do not mean the same, i.e. $[\![C]\!] \neq [\![C']\!]$, but we do have $(\forall\, w' : w \subseteq w' : w' \in [\![C]\!] \equiv w' \in [\![C']\!])$. So we could say, that they mean the same if we take into account the valuation that has been constructed so far.

Finally, the **if** $l = k + 1$ **cor** $s_m \notin N_l$ **then** $G := G \cup \{f\}$ statement of the original algorithm is replaced. This statement expresses that witness function $f$ remains a valid witness if signal $s_m$ does not match negative signal set $N_l$. In the new algorithm it has to be checked if $s_m$ does not match $N_l$, given valuation $v$. Furthermore, the constraint $C$ has to be strengthened with the forbidden future assignments that make $s_m$ and $N_l$ match. In Subsection 3.6.1 we described how $constraint(s_m, N_l)$ can be constructed. Formally, this construction can be expressed as follows:

$$constraint(s, N) = (\underline{\bigwedge} n \in N, w : minext(w, \emptyset, s, \{n\}) : (\underline{\bigvee} X, e : X \in dom(w) \ \wedge \ w(X) = e : \underline{X \neq e}))$$

What we have just said about the interchangeability of $C'$ and $C$, here holds for $C'$ and $C \ \underline{\wedge}\ constraint(s_m, N_l)$.

## 3.7 Implementation and Testing

The algorithm has been implemented in *C*, and it was tested on some patterns and some handcrafted log files. The disadvantage of handcrafted log files, is that they are small, and made with the recognition of patterns in mind, which is okay for testing the correctness of the algorithm, but not for testing the pattern description capabilities of *LOGAN*. Therefore, we conducted an experiment with an SDL [BH88, IT94] specification of a switch with a Call Waiting service. This specification, made by N. Goga in the context of *Côte de Resyste*, a research project on the testing of reactive systems, is an extensive one and covers also very exotic call waiting scenarios. The idea was to automatically produce some large log files, using this specification and SDT, the SDL toolset from Telelogic [Tel95]. The set up of the experiment is depicted in Figure 3.8.

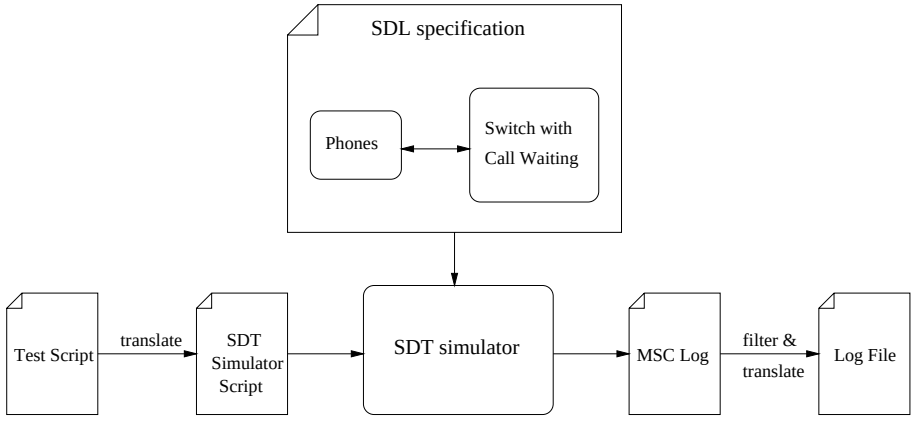



Figure 3.8: using an SDL specification to produce log files

We wrote test scripts, describing different kinds of Call Waiting scenarios, which were automatically translated to SDT simulator scripts. An SDT simulator script contains a list of commands, mostly signals that have to be sent to the SDL system. These scripts were fed to the simulator running a simulation of the switch. In each simulation run, the simulator produced an MSC log, showing all the signal exchanges that took place in the SDL system during the simulation. From these MSCs we automatically created log files by extracting the signals that we were interested in (those exchanged between the phones and the switch) and translating them to the proper format.

### 3.7.1 Some Test Results

We created one 'large' test script containing six successful call waiting scenarios. Since the first part of a call waiting scenario is also a normal call, it automatically contained six successful normal call scenarios as well. With the normal call pattern of Section 3.3 we found all 6 normal calls in the log file that was produced.

The call waiting pattern of Section 3.3, however, proved to be much too strict. With it, we only found 1 call in the log file. The pattern requires that some subscriber `A` sets up a call with subscriber `B`, and that after that, a third subscriber, `C`, tries to set up a call with `B`. Other possibilities, which the pattern does not cover, are that `B` himself starts the call with `A`, and that `A` and `C` call `B` more or less simultaneously.

We therefore wrote a new pattern for call waiting in which there is no reference to party `A`. Of course, this means that with this pattern we only detect the second leg of a call waiting scenario, but we know that the first leg of such a scenario can be detected with the normal call pattern.

With the new pattern 8 witnesses were found, 5 (out of 6) true witnesses, but also 3 false ones. The 3 false witnesses were due to identification of the former `A` party, which we removed from the pattern, with the `C` party, so these patterns consisted of parts of first and second legs mistakenly recognised as one single second leg.

We added a negative `B:Off_Hook` signal to the pattern that assures that the `B`

party is already engaged in a call, when the call of the C party arrives. With this addition the 3 false witnesses were no longer recognised with the pattern. Here is the final version of the call waiting pattern:

```
PATTERN Call_Waiting
BEGIN
  C:Dial_Tone;
    NOT C:Busy_Tone,C:On_Hook;
  C:Dial(B);
    NOT B:Off_Hook,B:Busy_Tone,B:On_Hook,B:Warning_Tone,
        C:Busy_Tone,C:On_Hook;
  B:Warning_Tone;
    NOT B:Busy_Tone,B:On_Hook,C:Busy_Tone,C:On_Hook,B:Hookflash;
  B:Hookflash;
    NOT B:Busy_Tone,B:On_Hook,C:Busy_Tone,C:On_Hook,C:Connect;
  C:Connect;
    NOT B:Busy_Tone,B:On_Hook,C:Busy_Tone,C:On_Hook;
  C:On_Hook,B:On_Hook;
END
```

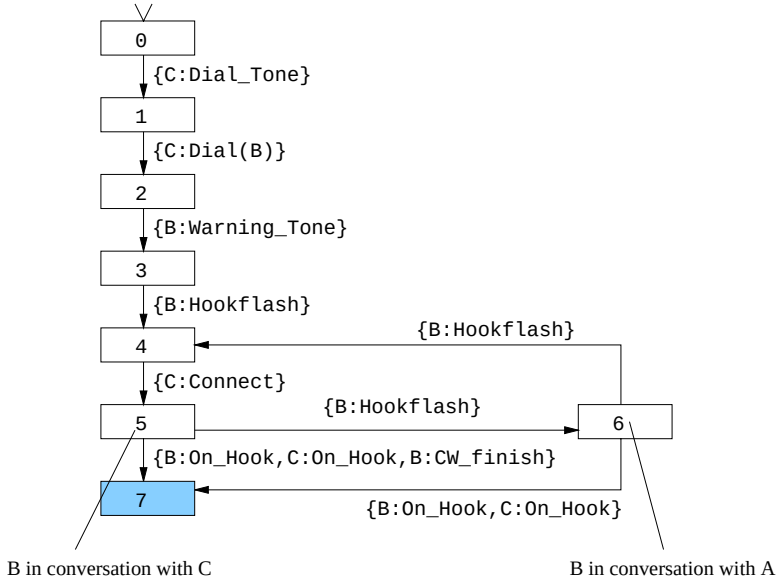
With the call waiting pattern we just presented, 5 out of 6 true witnesses were found, and no false witnesses, which is quite satisfactory as we do not expect to find every call. The one call that we did not find was a call that was not terminated with a `C:On_Hook` or `B:On_Hook`, but with a `B:CW_finish` signal. With this signal the `B` party can terminate the active leg of the call, whereas a `B:On_Hook` would terminate both legs.

## 3.8 A Language Extension

If we would add the `B:CW_finish` signal to the last positive signal set of the pattern, we would get a pattern that recognises all 6 calls in our log file. However, such a pattern might in other cases produce false witnesses as well. The reason for this is, that the `B:CW_finish` signal can be directed to the `C` party *or* the `A` party, and we need it to be directed to the `C` party.

If we want to add this signal to our pattern, we would have to know the state of the `B` party: is it currently connected to the `A` party or the `C` party? This information can be received from the number of hookflashes made by `B`, but in *LOGAN* this information cannot be represented.

A possibly interesting extension of *LOGAN* would therefore be the addition of explicit states and state transitions. *LOGAN* would then get the expressive power of finite state machines and regular expressions. Figure 3.9 shows what a pattern for call waiting could look like in such an extension of *LOGAN*. Note the two possible transitions leading to the final (grey) state, one originating from a state where `B` is in conversation `C`. Here the `B:CW_finish` signal can be used to terminate the connection between `B` and `C`. The other originates from a state where `B` is in conversation with `A`. Here the `B:CW_finish` signal does not terminate the connection between `B` and `C`.




```
0. {}
1. {C:Busy_Tone,C:On_Hook}
2. {B:Off_Hook,B:Busy_Tone,B:On_Hook,B:Warning_Tone,C:Busy_Tone,C:On_Hook}
3. {B:Busy_Tone,B:On_Hook,C:Busy_Tone,C:On_Hook,B:Hookflash}
4. {B:Busy_Tone,B:On_Hook,C:Busy_Tone,C:On_Hook,C:Connect}
5. {B:Busy_Tone,B:On_Hook,C:Busy_Tone,C:On_Hook,B:CW_finish}
6. {B:Busy_Tone,B:On_Hook,C:Busy_Tone,C:On_Hook}
7. -
```

Figure 3.9: a state machine-like pattern description of call waiting

Extending *LOGAN* to such a state machine-like form would have other advantages. Often a certain part of the pattern can have 2 or more forms. For example, in the call forwarding scenario, `C:Dial*21(*)` is given as a negative signal. However, such a signal on itself would not be problematic, provided it is followed by a `C:Dial#21` signal. This possibility could be added, but that would mean adding another pattern, and because it can occur on 4 different places, the total would then be $2^4 = 16$ different patterns. In real-life examples, there may be even more such minor variations, which makes the number of possible variations grow explosively. If state machine-like patterns are used, all these variations, and even variations containing different features, might be included in one single pattern, each variation requiring the addition of one or two extra 'states' (negative event sets) rather than a doubling of the number of patterns.

The algorithm we have given can be easily, although not trivially, extended to cover state machine-like *LOGAN*. To do so, the next positive possibility should be any 'state' that can be reached from the current one, rather than (as in the current algorithm) always the next one.

## 3.9 Conclusions

We have defined a pattern description language, *LOGAN*, in which we can express properties of call traces. For different call types, especially calls in which different services are active, different patterns can be defined. Basically, a pattern describes the characteristic sequences in call traces of a certain call type, using so-called *negative* signals to ensure the coherence of these characteristic sequences.

By giving some examples of *LOGAN* patterns for well known call types, and some examples of log files, we have demonstrated the use of *LOGAN*. We also, briefly, addressed the problem of feature interaction, showing the limitations of the pattern matching approach to finding call traces. Even simple types of interaction, like that of Call Forwarding with itself, cannot be covered by this type of pattern matching, especially when interactions can go to an arbitrary far degree.

Another problem is that finding correct patterns may be hard. Still, a short time of experimentation would normally solve this. A larger problem is that some features could either not be described in full, or would require a large number relatively similar patterns. This problem might be overcome by extending *LOGAN* into a state machine-like form. This would improve the pattern description capabilities of *LOGAN* considerably.

We also designed and implemented an algorithm for finding *LOGAN* patterns in log files. This algorithm, together with the *LOGAN* language, could be the basis for tools that support the testing process described in the introduction.

Further work in this area could be to investigate if, and how, CDRs can be computed automatically given a pattern (witness). Another option might be to automatically generate patterns from a description in SDL or some other similar language.

# Chapter 4

# The MSC Language

## 4.1 Introduction

Many languages have been designed to describe the behaviour of information systems. Using such a language, one can describe the high-level behaviour of a system without having to worry (yet) about the exact implementation details. One such language is Message Sequence Chart (MSC) [RGG96a, IT00], which is the subject of the following chapters. It differs from other languages in two important aspects. In the first place it puts emphasis on the communication between processes, not paying much attention to the internal behaviour of these processes. This way, it specialises on systems in which communication is important. Because many systems nowadays have a distributed nature, this holds for many systems. One area where it is much used, and the one for which it was originally created, is telecommunication systems. In the second place, MSC provides a graphical representation, rather than just a textual description. Because of this, it can be more easily and intuitively understood by human users. Still, behind this graphical syntax lies an exact meaning and a well-defined semantics. Because of this, it can also be well understood by tools such as SDT [Tel95].

MSC-like diagrams have a long history in formal descriptions of information systems, but the official Message Sequence Chart language has been developed in the early nineties within the ITU (International Telecommunication Union) and its predecessor, the CCITT (Comité Consultatif International Télégraphique et Téléphonique)

In this Chapter, a short overview will be given of the history of the language, of a number of its constructs and of its semantics. But first, we will give an example of a simple MSC, to give an impression of what the language looks like.

An example of an MSC is given in Figure 4.1.

The vertical lines in the MSC ($i$, $j$ and $k$) denote the so-called *instances*, which represent the processes, objects or systems whose behaviour is described. The arrows between them denote messages between the instances. These messages are the basic constructs of MSC, but many other features, such as timers, are also included. In Figure 4.1 there is one, simple example of such a feature: $a$ denotes some otherwise unspecified local event at instance $i$.

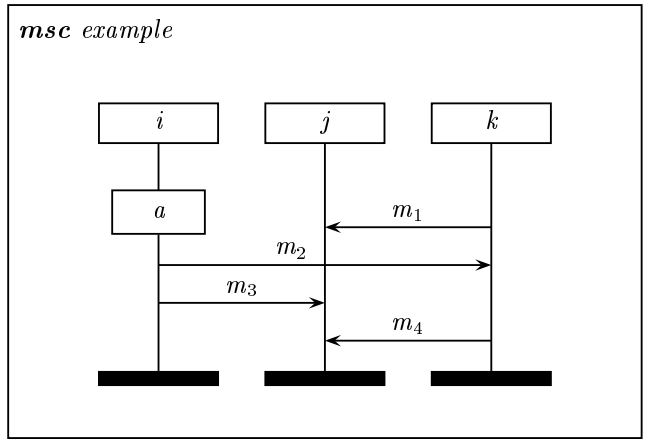



Figure 4.1: An example of an MSC

Time runs from the top to the bottom, but does not have to run at the same speed at each instance. For example, message $m_2$ must be received before message $m_4$ is sent, because the receipt of $m_2$ is above the sending of $m_4$ at the same instance, but $m_4$ may be sent before $m_1$ is received. The sending of $m_4$ is below the receipt of $m_1$, but at a different instance, so there is no ordering. The only ordering that exists between different instances is that each message has to be sent before it is received.

MSCs are used in different contexts. The original purpose of MSC when it was first formalised, was to describe requirements in the early phases of the development process. It was intended to be an addition to SDL (Specification and Description Language), where the two languages would be used in different phases of the development process: MSC early on, when requirements and global specifications are made, SDL later on, when specifications are closer to the final implementation.

However, the language is now used in many more applications. To name a few: the description of the actual behaviour of an existing system, especially in the context of testing, the generation of test cases [GHN93], the specification of protocols and the formalisation of use cases [RAB96], and the display of simulation traces [VGMF00].

## 4.2 History

MSC-like diagrams (often taken together under the name 'Sequence Charts') have been in existence for a long time [Lam78]. They have been used in various contexts, either as a stand-alone description of a standard, or as illustrations to more formal descriptions in languages like SDL [IT94, BH88, SRS89, BHS91], Estelle [ISO88a, BD87] or LOTOS [ISO88b, EVD89]. Because Sequence Charts were so widely used, but often in different variants, a need was felt for a more formal basis for these diagrams. That way, their usage could be harmonised across various users and institutions, in a way that would moreover be formally defined.

In 1989, at the fourth SDL Forum, a proposal was made [GR89] to start developing such a formalised sequence chart language. Not only would a standardisation overcome the (mostly syntactic) differences between the various languages, it would also make tool support [Ek93, Loi96] possible, and provide possibilities to define a formal mapping between MSC and SDL specifications [Gra90, Kri91, Nah91].

In 1990 such a plan was approved by the CCITT, and responsibility for the language was given to the CCITT Study Group X, which also was responsible for SDL. In 1992, the first version of the MSC language, containing a number of basic constructs, was formally approved as CCITT Recommendation Z.120 [IT93].

The standardisation of the language, and even more importantly, the resulting possibilities for tool support [Tel95, Ver96, Pel98], led to a remarkable growth in the use of the language. However, with this it also became clear that the language was not complete enough to fully describe an information system, not even at the high levels where it was supposed to be most useful. In the next four-year period, from 1992 to 1996, a number of extensions to the language was therefore discussed [MS93, Rüf94, Mei95, Rud95, Sch95, Far96].

Another important step in this period was the creation of a formal semantics for MSC. Several semantics were proposed [Til91, dM93, MvWW93, GRG93, LL94, MR94a], and the process algebra semantics [MR94b, BM95] was agreed upon, and officially adopted in 1995 [IT95]. This semantic view also provided one of the most important extensions of MSC, namely that with composition mechanisms, such as HMSC [Rud95, MR97a].

This and other language extensions were included in a new version of the language, which appeared in 1996 [IT96, HL97], MSC'96. However, further extensions were still wanted. In 2000 a new version (MSC2000) [IT00, Hau00] of the language was introduced. It contained a number of extensions, the most important of those being the inclusion of time information [SRM97, Sil98, GDO98], representation of data [EFM99, Eng00] and object-oriented features such as flow of control [RGG99]. It is hoped that this last extension will make a unification of MSC with time sequence diagrams from UML [BRJ98] possible. In this thesis, we will look into the way data has been included in the MSC language in Chapter 6. In Chapters 7 and 8, we will be looking at message refinement and disrupt and interrupt, two more proposals for extension of the MSC language, which were not included in the language – although of course they still might be included in the future.

In the meantime, research on MSC has also continued. The existing semantics for MSC have been extended to cover the MSC'96 language [MR97b, Ren99, IT98], and some new possible semantical frameworks for MSC have been introduced [Kos97, Hey98, KL98, Klu99, Hey00]. Much research has been going into the automatic or semi-automatic generation of specifications in SDL or other languages from MSC descriptions [SD97, RKG97, LMR98, KRBG98, Fei99, AKB99, KGSB99, MZ99, Man99, Mus99, HJ00]. Other research checked how certain properties of a system could be known from its MSC description, such as race conditions [AHP96], process divergence and non-local choice [LL95, BAL97b] (however, note that the notion of safe realisability, as defined in [AEY00] seems to cover the actual problems caused by non-local choice better), necessary buffers [EMR97b] (see Chapter 5 of this thesis), implementability by locally specified elements [KRBG98], and the existence of possible unspecified

behaviours [AEY00]. There are also some more general results: It has been found how to check a complete MSC description against a partial one [LP97] and whether traces of MSC contain some with a specific behaviour [AY99]. In [MPS98, MP00] there are some results on decidability of properties of systems described in MSC.

At the same time, research has also gone on into the applications of MSC, and it now includes such diverse areas as requirements specification [GW96], system design and software engineering [GRG91, MT96, MRW00, VGMF00], specification of test purposes [GHNS95, GSDH97, SEG+98, GH00, RSG00a], visualisation of test cases [Heg95, GW98, RSG00b, GGR01], formalisation of Use Cases [AB95, RAB96, BC00, Fei00], detection of feature interaction in telecommunication systems [BB97] and workflow analysis [Aal99], while attempts are being made in the area of natural language analysis [End00]. There are also attempts to combine MSC with sequence diagrams from UML [RGG99, Hau01], and MSC has been introduced as a graphical syntax [GW98, RSG00b, SG01, BRS01] for TTCN [KW91].

## 4.3 An Overview of the MSC Language

### 4.3.1 Basic Constructs: Messages

In Figure 4.2 we see another example of a Message Sequence Chart. It shows the process of giving a test to a student by a teacher.

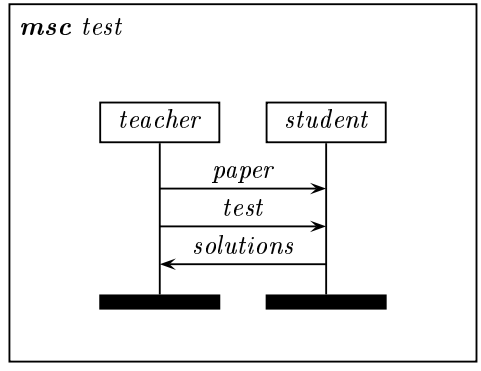



Figure 4.2: An example of an MSC

The vertical lines are called *instances*, and show the various entities whose behaviours are described by the MSC. In this case there are two instances, one is called '*teacher*' and the other '*student*'. The blocks at the top and bottom have no special meaning, they just show the beginning and end of the description of the instance – which not necessarily coincides with the beginning or end of the instance itself. The arrows show messages that are sent. In this case the messages are the paper and the test that are given by the teacher to the student, and the completed test that is given back to the teacher.

In this diagram time is running from top to bottom. That is, first the paper is given, then the test, and finally the student gives back his solutions. However, one should note that:

1. Sending a message and receiving it are considered two separate actions. That is, some time passes in between, and other actions may happen in between.

2. Time runs separately on each instance. That is, events (like the sending and receipt of messages) that are on the same instance are ordered as they appear from top to bottom in the diagram, but events on different instances need not be. Their order is not specified. The only order that exists between messages on different instances is the ordering that is caused by the fact that a message needs to be sent before it can be received.

For example, in the MSC above the teacher may send the test before the paper is received. Although it is higher in the diagram, the reception of the paper is on a different process, so their different positions do not need to correspond with an actual temporal ordering. Of course, in this example it is not very realistic that there will be much time between the moment the paper leaves the teacher and the time it reaches the student, but we could for example think of the materials as being sent through the mail – in that case the teacher could send the exercises while the paper was still under way.

On the other hand, the teacher must have given the paper before the student can receive the test, because the paper must be sent (given) before the test can be sent, and the test must be sent before it can be received.

The meaning of an MSC is determined by the possible traces, that is the various orders in which events can take place. In Figure 4.2 there are exactly two:

1. sending '`paper`', receiving '`paper`', sending '`test`', receiving '`test`', sending '`solutions`', receiving '`solutions`'

2. sending '`paper`', sending '`test`', receiving '`paper`', receiving '`test`', sending '`solutions`', receiving '`solutions`'

It is allowed for messages to cross, or overtake one another. In that case, the message that is sent first, is received last. What is not allowed, is a cyclic dependency, that is, two events for which (directly or indirectly) both the first has to come before the second and the second before the first. Such events would cause the MSC to become meaningless, since no trace would be possible.

### 4.3.2 Local Actions

A very simple extension of the language is the *local action*. This is simply something that happens at one instance, and has no effects elsewhere. It is shown as a block, and can be found in Figure 4.1. The action $a$ is here something that is done by instance $i$, and does not influence any other instance directly.

### 4.3.3 Co-region

Sometimes one does not want to specify exactly in which order events on one instance take place. For example, if we extend our MSC *test* with a second student, doing the same test, we do not want to specify which student is the first to finish and submit her completed test.

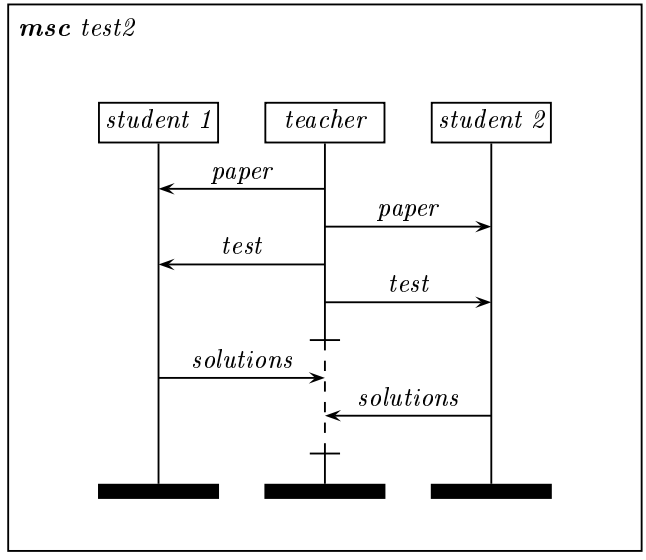



Figure 4.3: MSC with coregion

In Figure 4.3 the dashed part of the line showing the teacher's behaviour is a so-called co-region. Events in a coregion are not ordered, so the reception of the two completed tests can occur in any order.

### 4.3.4 MSC References

When an MSC grows large, it may become hard to read. Several additions are made to make it possible to break an MSC into pieces.

One way to do this is by describing parts of the MSC separately. For this an MSC reference expression is used. This is a box, replacing part of the description of one or more instances, containing the name of a separate MSC that describes the behaviour of the instances involved. For example, the MSC `test` could be part of a larger `course` MSC, as shown in Figure 4.4.

The box in the MSC `course` is the reference MSC, referring to the MSC `test` (by way of giving its name – see figure 4.2 for a possible content of this MSC). One should think this as some kind of shorthand notation, what happens in the MSC reference expression is described by the MSC `test`.

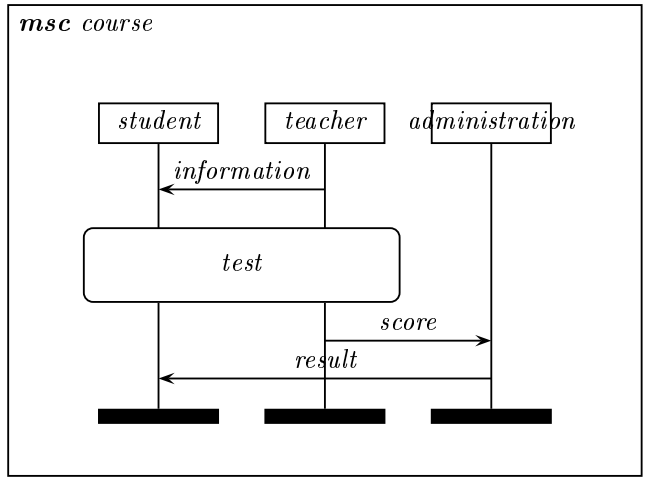



Figure 4.4: MSC with reference MSC

### 4.3.5 Inline Expressions

Inline expressions are sections of MSC where a choice, loop or other special construct takes place. For example, the abovementioned MSC with two students doing their test at the same time, could also be implemented with an inline expression as shown in Figure 4.5.

The square construct with '`par`' in the upper left corner, is the inline expression. The text in the upper left tells us the type of inline expression, '`par`' means that it is a parallel inline expression, that is, the two (or more) parts of the inline expression (separated by the dashed line) have to be done in parallel.

Other inline expressions are:

Optional (`opt`): The actions inside the inline expression may be executed or skipped.

Alternative (`alt`): Exactly one of the parts is chosen.

Exception (`exc`): The actions inside the inline expression may be executed instead of those in the surrounding MSC.

Loop (`loop`): The actions inside the inline expression must be executed a number of times (the minimum and maximum number, which must be either a natural or the special value `infinity`, are given).

Of these inline expression, only the parallel and alternative inline expressions have parts; the other ones consist of just a single box, without a separator.

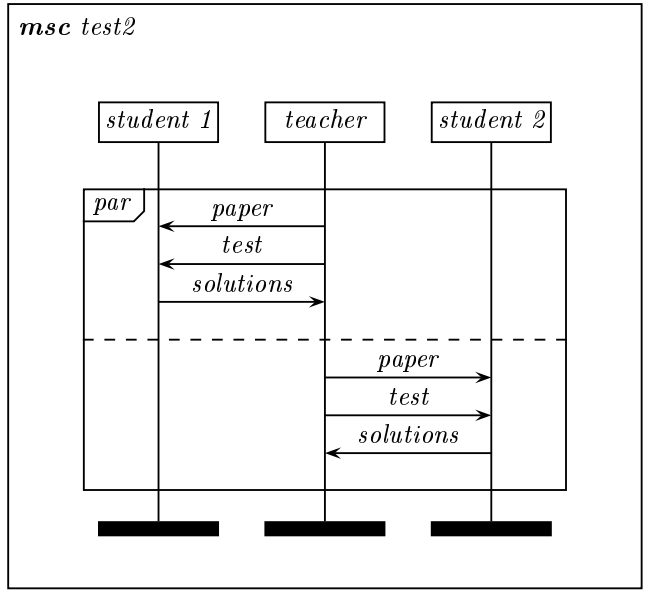



Figure 4.5: MSC with Inline Expression

### 4.3.6 High-Level MSCs

One can go even further, and split up the complete MSC into parts. This way one gets a picture with a succession of MSCs that have to be gone through in a certain order. To make this possible, High-level MSCs (HMSCs) have been introduced [MR97a]. An example of an HMSC is shown in Figure 4.6.

To read a diagram like the one in Figure 4.6, one starts at the start symbol $\bigtriangledown$. Following the vertical line, we first get to the reference MSC `teaching`, after this to MSC `test`, and finally to the end symbol $\bigtriangleup$. The meaning of this, is that the MSCs `teaching` and `test` are combined into one, `teaching` happening first, and `test` after it.

One should note that the way the constituting MSCs are combined implies an ordering in time, but again the ordering holds only per instance. Thus, when all events for the teacher in the MSC `teaching` have been done, `teacher` can start doing actions from `test`, whether or not the student still has actions to do from `teaching`.

But the possibilities of HMSC are larger than just the sequential ordering of several MSCs. In the first place, the MSCs that are being referenced may be HMSCs themselves, thus allowing for more than two levels of description. But what is more important are their possibilities of specifying choices and loops.

When our hypothetical student has gone through the test, and seen his result, it is not unlikely that he will choose to do the test again, and maybe repeatedly do so until he passes. This possibility is shown in Figure 4.7.

When we go through this figure from the top to the bottom, we first see a small circle. This is a connection point. Here, the MSC continues along the line going down,

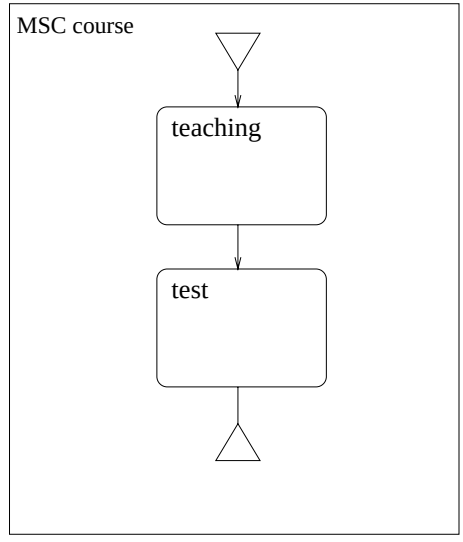



Figure 4.6: High-level MSC

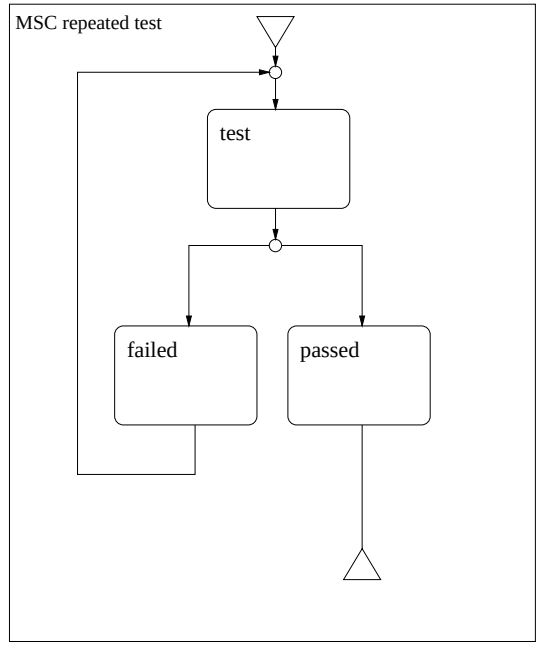



Figure 4.7: HMSC with loop

irrespective of the question from which of the two arrows it came. After going throug '`test`', there is another connection point, this time with two exits. This denotes a choice. Execution of the MSC continues either through the left or through the right path, but not both. Finally, the arrow that goes up again from '`failed`' creates a loop structure.

Of course one could also make more complicated structures, with intertwined loops, loop escapes, etcetera.

### 4.3.7 Further MSC Constructs

There are several more MSC constructs that have not been included in this introduction. These include causal orders, that can be used to force a temporal order between otherwise unconnected actions, and instance refinement, where it can be specified in which way a single instance can be decomposed into several separate ones. An introduction on MSC, where the subjects treated in this chapter are being treated together with other aspects of the language, is [RGG96b].

A number of new language elements has been introduced in MSC2000. One of them is data, which will be discussed more extensively in Chapter 6. There are also extensions regarding (relative and absolute) time, flow of control and the overall structure of a document consisting of various MSCs. These are discussed in [Hau00].

## 4.4 Formal Semantics

In this section, we will give an overview of the official process algebra semantics of MSC. A more extensive discussion can be found in [Ren99], which contains reasons for various rules, historical notes, properties of the semantics and examples, as well as a complete semantics, of which we will show only the most important parts.

For the semantics of MSC, each MSC is translated into an expression in process algebra [BW90]. This process algebra does however contain a number of operators specifically for MSC. The semantics itself is operational, consisting of rules of the form $\frac{\text{cond}}{x \xrightarrow{a} y}$, which can be translated as "When the conditions 'cond' are true, a system in the state $x$ can do a step of the type $a$ to state $y$". The basic rules are $\frac{}{a \xrightarrow{a} \epsilon}$, which says that an event $a$ can do step $a$, and then results in the empty process $\epsilon$, and $\frac{}{\epsilon \downarrow}$, which means that $\epsilon$ can terminate, that is, succesfully end without doing any further actions.

The process algebra for MSC differs somewhat from normal process algebra. In the first place, the semantics of MSC are completely deterministic, that is, if a process has both the option of doing $a$ followed by $b$ and the option of doing $a$ followed by $c$, then after doing $a$, it still has both the option of doing $b$ and the option of doing $c$. This is unlike normal process algebra, where the process $a \cdot b + a \cdot c$ can do $a$ to $b$, or to $c$, but not to $a + c$. MSC therefore uses the delayed choice operator $\mp$ [BM95]. The deduction rules for $\mp$ are:

$$\frac{x\downarrow}{x \mp y\downarrow} \qquad \frac{y\downarrow}{x \mp y\downarrow} \qquad \frac{x \xrightarrow{a} x', y \not\xrightarrow{a}}{x \mp y \xrightarrow{a} x'} \qquad \frac{x \not\xrightarrow{a}, y \xrightarrow{a} y'}{x \mp y \xrightarrow{a} y'} \qquad \frac{x \xrightarrow{a} x', y \xrightarrow{a} y'}{x \mp y \xrightarrow{a} x' \mp y'}$$

We will first look at a simple example, just a number of local action, see Figure 4.8.

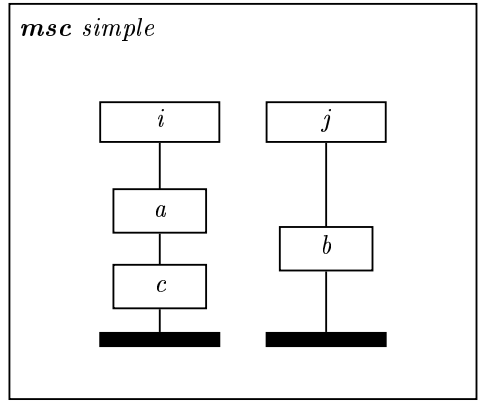



Figure 4.8: A simple MSC

First, the various events of the MSC are translated into the corresponding process algebra events. For an action, this simply is $action(i, a)$, with $i$ being the instance on which the action takes place, and $a$ the text in the action box. Next, the MSC is split up into parts, which are connected with the 'weak sequencing' operator $\circ$. Thus, the MSC above is translated into the process algebra expression $action(i, a) \circ action(j, b) \circ action(i, c)$.

The weak sequencing operator $x \circ y$ has as its semantics that actions from $x$ are always possible, while actions from $y$ are possible if and only if there are no actions from $x$ on the same instance. To translate this into process algebra, an extra relation, the permission relations $\cdots\!\!\rightarrow$ is added. $x \overset{a}{\cdots\!\!\rightarrow} y$ means that $x$ allows $a$ even if it is 'after' $x$ in the MSC, and this results in $x$ changing into $y$ ($y$ can be unequal to $x$ if $x$ contains a choice, which may be a choice between options some of which do and some of which do not allow $a$). The basic SOS-rules for the permission relation are: $\frac{l(a) \neq l(b)}{b \overset{a}{\cdots\!\!\rightarrow} b}$ and $\frac{}{\epsilon \overset{a}{\cdots\!\!\rightarrow} \epsilon}$. Here, $l(a)$ is the instance on which the action $a$ takes place. Thus, the empty process permits anything, while an action stops other actions on the same instance, but allows actions on different instances.

Three rules exist for the delayed choice regarding the permission relation, depending on whether the part to the left of the $\mp$, the part to the right, or both permit the

action:

$$\frac{x \overset{a}{\cdots\!\rightarrow} x', y \overset{a}{\cdots\!\not\rightarrow}}{x \mp y \overset{a}{\cdots\!\rightarrow} x'} \qquad \frac{x \overset{a}{\cdots\!\not\rightarrow}, y \overset{a}{\cdots\!\rightarrow} y'}{x \mp y \overset{a}{\cdots\!\rightarrow} y'} \qquad \frac{x \overset{a}{\cdots\!\rightarrow} x', y \overset{a}{\cdots\!\rightarrow} y'}{x \mp y \overset{a}{\cdots\!\rightarrow} x' \mp y'}$$

With this addition, we can give the deduction rules for the weak sequential composition (see [Ren99] for more explanation):

$$\frac{x \downarrow, y \downarrow}{x \circ y \downarrow} \qquad \frac{x \overset{a}{\cdots\!\rightarrow} x', y \overset{a}{\cdots\!\rightarrow} y'}{x \circ y \overset{a}{\cdots\!\rightarrow} x' \circ y'} \qquad \frac{x \overset{a}{\rightarrow} x', x \overset{a}{\cdots\!\not\rightarrow} \vee y \overset{a}{\not\rightarrow}}{x \circ y \overset{a}{\rightarrow} x' \circ y}$$

$$\frac{x \overset{a}{\not\rightarrow} x', x \overset{a}{\cdots\!\rightarrow} x', y \overset{a}{\rightarrow} y'}{x \circ y \overset{a}{\rightarrow} x' \circ y'} \qquad \frac{x \overset{a}{\rightarrow} x', x \overset{a}{\cdots\!\rightarrow} x'', y \overset{a}{\rightarrow} y'}{x \circ y \overset{a}{\rightarrow} x' \circ y \mp x'' \circ y'}$$

With these rules and the associativity of $\circ$ (which is proven in [Ren99]), we find that the process $action(i,a) \circ action(j,b) \circ action(i,c)$ can execute $action(i,a)$ and go over in $\epsilon \circ action(j,b) \circ action(i,c)$, and can execute $action(j,b)$ to go over in $action(i,a) \circ \epsilon \circ action(i,c)$, but cannot execute $action(i,c)$ because $action(i,a)$ does not permit $action(i,c)$.

For messages, there is something more to do. If we look at the MSC in Figure 4.9, the semantics as far as we have seen it now are $out(i,\_,j,m) \circ in(i,\_,j,m)$ (the $\_$ here shows the absence of gates, which are not dealt with in this thesis). Because $l(out(i,\_,j,m)) \neq l(in(i,\_,j,m))$, this process would be permitted to start with $in(i,\_,j,m)$, which is of course unwanted. Fiddling with the permission relation would not help, because this same MSC could also be split as $in(i,\_,j,m) \circ out(i,\_,j,m)$. Instead, the information is added to the weak sequencing operator, and the expression is written as $out(i,\_,j,m) \circ^{out(i,\_,j,m) \overset{0}{\mapsto} in(i,\_,j,m)} in(i,\_,j,m)$.

The condition $a \overset{n}{\mapsto} b$ means that before $b$ can be executed, first $a$ has to be executed, but that $b$ can still be executed $n$ times because $a$ has already been executed $n$ times more than $b$. There is a predicate $enabled(a,S)$ which is true if and only if $a$ is allowed by the set of conditions $S$, and an update function $upd(a,S)$, which gives the new set of conditions $S$ after $a$ has been executed. Their definitions are:

$$\begin{array}{lcl} enabled(a,S) & \Leftrightarrow & \forall_{b,c \in A, n \in N}\, b \overset{n}{\mapsto} c \in S \Rightarrow (c \not\equiv a \vee n > 0), \\ upd(a,S) & = & \{b \overset{n}{\mapsto} c | b \overset{n}{\mapsto} c \in S \wedge b \not\equiv a \wedge c \not\equiv a\} \\ & \cup & \{b \overset{n-1}{\mapsto} c | b \overset{n}{\mapsto} c \in S \wedge c \equiv a \wedge n > 0\} \\ & \cup & \{b \overset{n+1}{\mapsto} c | b \overset{n}{\mapsto} c \in S \wedge b \equiv a\} \end{array}$$

The deduction rules for $\circ^S$ are similar to those of $\circ$, but to execute an $a$ step, $enabled(a,S)$ has to be true, while doing so changes $S$ into $upd(a,S)$. This leads to:

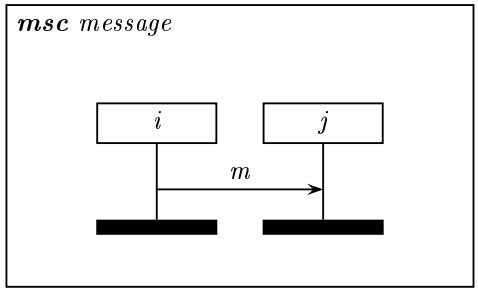



Figure 4.9: A simple MSC with a message

$$\frac{x\downarrow, y\downarrow}{x \circ^S y \downarrow} \qquad \frac{x \xrightarrow{a} x', x \overset{a}{\cdots}\!\!\!\!\rightarrow \vee y \xrightarrow{a}\!\!\!\!\!\!/\;\;, enabled(a,S)}{x \circ^S y \xrightarrow{a} x' \circ^{upd(a,S)} y}$$

$$\frac{x \overset{a}{\cdots}\!\!\!\!\rightarrow x', y \overset{a}{\cdots}\!\!\!\!\rightarrow y'}{x \circ^S y \overset{a}{\cdots}\!\!\!\!\rightarrow x' \circ^S y} \qquad \frac{x \xrightarrow{a}\!\!\!\!\!\!/\;\;, x \overset{a}{\cdots}\!\!\!\!\rightarrow x', y \xrightarrow{a} y', enabled(a,S)}{x \circ^S y \xrightarrow{a} x' \circ^{upd(a,S)} y'}$$

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

Then there is the parallel composition operator $\|$. $x \| y$ consists of all actions of $x$ and $y$ interleaved. Its semantics are:

$$\frac{x\downarrow, y\downarrow}{x \| y \downarrow} \qquad \frac{x \xrightarrow{a} x', y \xrightarrow{a}\!\!\!\!\!\!/\;\;}{x \| y \xrightarrow{a} x' \| y}$$

$$\frac{x \overset{a}{\cdots}\!\!\!\!\rightarrow x', y \overset{a}{\cdots}\!\!\!\!\rightarrow y'}{x \| y \overset{a}{\cdots}\!\!\!\!\rightarrow x' \| y'} \qquad \frac{x \xrightarrow{a}\!\!\!\!\!\!/\;\;, y \xrightarrow{a} y'}{x \| y \xrightarrow{a} x \| y'}$$

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

There are a few more operators, namely a version of the parallel composition with requirement $\|^S$ and repetitions $x^*$ and $x^{\inf}$, but those will not be dealt with in this thesis. Instead, we will refer the interested reader to [Ren99].

# Chapter 5

# MSC and Communication Models

## 5.1 Introduction

In MSC, a message can be sent and received at any time. In particular, a message can overtake another message, even if it is sent between the same pair of instances. Apparently, MSC uses the assumption that either the communication mediums or the buffers involved in a system can send messages through in any order. If there were for example only a FIFO buffer between each pair of instances, message overtaking would be impossible.

The assumptions about the buffering of messages in MSC, are in contrast with the situation in a specification language such as SDL [IT94], where every entity has its own FIFO input buffer. Since MSC and SDL are often used in conjunction, there is a need to clarify this seemingly contradictory situation.

When considering restricted communication mechanisms, it is very natural to identify subclasses of MSC which exactly satisfy such buffering properties. One can consider the class of *FIFO buffered* MSCs, the class of *synchronous* MSCs, etc. In fact, the Interworkings language [MvWW93, MR01] is the latter class.

When considering Interworkings simply as a subset of MSC, an obvious question to ask is: what exactly is the distinction between synchronous and asynchronous MSCs? Or, phrased a little bit differently, how can we formally characterise the class of synchronous MSCs? Finding an answer to this question is not too difficult. An MSC is synchronous if and only if in every execution trace of the MSC there are no events between every pair of corresponding send and receive events.

But, how about the question whether an MSC can be implemented using only one FIFO buffer. And what, if we are allowed to use a number of FIFO buffers? This gives rise to a more general question. Can a given MSC be implemented by means of a given communication model? This is the question which will be studied in this chapter.

Thereto, we define the notion of *communication model*, we present a formal se-

mantics of MSC based on partial orders, and we define criteria for an MSC being implementable in a given communication model. We will not study the complete range of all possible communication models, but we will single out a number of interesting options, which we systematically derive by looking at the *locality* of the buffers between the communicating entities. One can, e.g., assume one single FIFO buffer for the complete system, or a FIFO buffer between each pair of entities, etc. We will also take into account the difference between output buffers and input buffers, since in practice this distinction is often made.

Apart from studying the fundamental concepts behind the implementability of Message Sequence Charts, there are also more practical motivations for the research presented here. First of all, the formal relation between scenario specifications in MSC and complete system specifications in a Formal Description Technique, such as SDL [IT94], is an important issue in the software engineering process. Not only the derivation of MSC scenarios from a Formal Description Technique, but also the synthesis of a complete specification from a collection of MSC scenario specifications is considered of great importance by many authors and tool builders (see [SDV95, RKG97, SD97, KRBG98, LMR98, Fei99, KGSB99, AKB99, MZ99, HJ00]). This naturally leads to the question which MSCs can and which MSCs cannot be implemented in the given specification language.

One can also study the same question from a different perspective, namely, given an arbitrary MSC, how can we restrict (or extend) its semantics in such a way that it can be implemented in a given communication model. This question is partly studied by Alur et al. [AHP96], who also derived supporting tools. Our starting point, however, will be that we consider the standard MSC semantics.

This brings us to the variety of ways in which MSCs are used, some of which are essentially different. We mention the distinction between *hot* and *cold* MSCs (see [DH99]) where (parts of) MSCs must or may occur in the implementation and we mention the difference between positive and negative use of MSC (an MSC must occur or is not allowed to occur). Finally, some users apply MSC to specify one single trace, while others consider the complete set of traces generated by an MSC. This latter dichotomy is wide-spread and, therefore, we will study the main question from both perspectives: one trace of an MSC must be implementable (the *weak case*) or all traces of an MSC must be implementable (the *strong case*).

Since all implementation relations introduced in this chapter identify subclasses of the class of Message Sequence Charts, it is interesting to know how these classes relate. The answer to this question is formulated as a hierarchy of communication models for Message Sequence Charts.

We present our research in the following way. In Section 5.1.1 we introduce the subset of the MSC language called basic MSCs, and give a simple formal semantics based on partial orders. The communication models which we study are defined in Section 5.2.1. In order to be able to deal with two distinct buffers between two communicating entities, we will extend the standard partial order semantics in Section 5.2.2. The definition of implementability of a single trace with respect to a communication model is given in Section 5.2.3. In Section 5.3, we classify traces according to their implementability. This work is lifted to the level of MSCs in Section 5.4, where we first study the strong case (Section 5.4.1), and then the weak case (Section 5.4.2).

The overall picture combining the strong and the weak case is given in Section 5.4.3. We also give a number of characterisations of the implementability relations, which make it possible to determine the implementability of a given MSC algorithmically (see Section 5.5). Section 5.6 contains a comparison with related literature and in Section 5.7 we summarise our findings and discuss options for further research.

### 5.1.1 Basic Message Sequence Charts

In this chapter, we will not be looking at the complete MSC language. Rather, we take only a subset, consisting of just instances and messages. In particular, we will have no co-regions, no HMSCs and no inline expressions. Because we have no HMSCs, each description will consist of just a single MSC. We will also not use the official semantics, but use a much simpler semantics that is equivalent to it when used for these simple MSCs, but not strong enough to give the semantics of more complicated structures. Furthermore, we will be assuming that the MSCs are all semantically correct, that is that they do not contain deadlocks through cyclic dependencies.

The easiest way to express the semantics of such a simple MSC is by using a partial order on the events that are comprised in an MSC. Depending on the particular dialect of the MSC language, one can assign different classes of events to an MSC. For example, in Interworkings [MvWW93, MR01] every message is considered to be a single event. There is no buffering, and thus communication is synchronous.

In MSC [IT00], messages are divided into two events, the output and the input of the message. The output of message $m$ is denoted by $!m$ and the input by $?m$. The only assumption about the implementation of communication is that an output precedes its corresponding input. An MSC describes a partial order on output and input events.

**Definition 1 (basic MSC)** A basic MSC is a quintuple $\langle I, M, \mathsf{from}, \mathsf{to}, \{<_i\}_{i\in I}\rangle$, where $I$ is a finite set of instances, $M$ is a finite set of messages, $\mathsf{from}$ and $\mathsf{to}$ are functions from $M$ to $I$, and $\{<_i\}_{i\in I}$ is a family of orders. For each $i \in I$ it is required that $<_i$ is a total order on $\{!m \mid \mathsf{from}(m) = i\} \cup \{?m \mid \mathsf{to}(m) = i\}$. We use the shorthand $E_{\mathsf{msc}}(M)$ to denote the set $\{!m, ?m \mid m \in M\}$.

In the above definition, $\mathsf{from}(m)$ denotes the instance which sends message $m$. Likewise, $\mathsf{to}(m)$ denotes the instance which receives message $m$. Given an instance $i$, the ordering $<_i$ denotes in which order the events attached to instance $i$ occur.

The partial order denoting the semantics of an MSC $k$ is derived from two requirements. First, the ordering of the events per instance is respected, and second, a message can only be received after it has been sent. The first requirement is formalised by defining the *instancewise* partial order $<_k^{\mathsf{inst}}$ ($k$ being the MSC under discussion):

$$<_k^{\mathsf{inst}} = \bigcup_{i\in I} <_i,$$

and the second requirement is formalised by the *output-before-input* order $<_k^{\mathsf{oi}}$:

$$<_k^{\mathsf{oi}} = \{(!m, ?m) \mid m \in M\}.$$

Now, we define the partial order induced by the MSC as the transitive closure (denoted by $^+$) of the instancewise order and the output-before-input order. For an MSC $k$, we denote this order by $<_k^{\mathsf{msc}}$ or by $<^{\mathsf{msc}}$ if $k$ is known from the context.

**Definition 2** For a given MSC $k = \langle I, M, \mathsf{from}, \mathsf{to}, \{<_i\}_{i\in I}\rangle$, the relation $<_k^{\mathsf{msc}}$ is defined by $<_k^{\mathsf{msc}} = (<_k^{\mathsf{inst}} \cup <_k^{\mathsf{oi}})^+$.

From an operational point of view, one can say that an MSC describes a set of traces. Such a trace denotes the ordering of output and input events ($!m$ and $?m$).

**Definition 3 (Traces)** Given a set of messages $M$, a *trace* $t$ over $M$ is a total ordering $(e_1, e_2, \ldots, e_n)$ of the set $E_{\mathsf{msc}}(M)$. A trace $(e_1, e_2, \ldots, e_n)$ is denoted $e_1\, e_2\, \ldots\, e_n$.

We denote the $i$th element of a trace $t$ by $t_i$, and its length by $|\,t\,|$. As a consequence of the above definition we can associate with each trace $t$ an order $<_t^{\mathsf{trace}}$. This order is useful in expressing that a certain trace $t$ is actually a trace of an MSC $k$.

**Definition 4 (msc-trace)** A trace $t$ is said to be an msc-trace of the MSC $k$ if and only if it is defined over the messages $M$ of $k$, and $<_k^{\mathsf{msc}} \subseteq <_t^{\mathsf{trace}}$.

**Lemma 5** For an MSC $k$ over $M$, and events $e, e' \in E_{\mathsf{msc}}(M)$, we have $e <_t^{\mathsf{trace}} e'$ for all msc-traces $t$ of $k$ if and only if $e <_k^{\mathsf{msc}} e'$.

**Proof** The 'if'-part is trivial. For the 'only if'-part we use contraposition. Suppose that $e \not<_k^{\mathsf{msc}} e'$. Then the relation $<_k^{\mathsf{msc}} \cup \{(e', e)\}$ does not contain a cycle. Thus, it can be extended to a total order $<$. Because $<^{\mathsf{msc}} \subseteq <$, $<$ will be the trace-order $<_t^{\mathsf{trace}}$ of some msc-trace $t$ of $k$. In this msc-trace we will have $e' <_t^{\mathsf{trace}} e$, and thus $e \not<_t^{\mathsf{trace}} e'$. ■

## 5.2 Implementation Models

### 5.2.1 Implementation Models for Communication

In this section we discuss possible architectures for realising an MSC. We consider only implementation models consisting of FIFO buffers for the output and input of messages. For msc-traces, we define what it means to be implementable on some architecture.

The particular implementation models which we are interested in are constructed of entities that communicate with each other via FIFO buffers. We assume that the buffers have an unbounded capacity. We discern two uses of buffers, namely for the output and for the input of messages.

A second distinction can be made based on the locality of the buffer. From most global to most local we distinguish the following types:

- global: A global FIFO buffer: All messages from all instances pass this buffer.
- inst: A FIFO buffer, local to an instance: All messages sent (or received) by one single instance go through the same buffer.

- pair: A FIFO buffer, local to two instances: All messages that are sent from one specific instance to another specific instance go through this buffer.
- msg: A FIFO buffer, local to a message: There is one buffer for every message.

This last model, a buffer per message, is a specific architecture to catch up the cases in which the buffers do not behave like FIFO queues, but as random-access buffers. Taking into account the assumption that messages are unique, it can easily be seen that it is equivalent to a global random-access buffer. A communication model with only a random-access buffer represents the implied model of the MSC standard: the only assumption made about the implementation of communication is that output precedes input, no more, and no less.

Finally, we consider the following possibility:

- nobuf: There are no buffers; communication is synchronous.

We assume that all output buffers are of the same type, and similarly that all input buffers are of the same type. This results in four possibilities for the output as well as for the input. Adding the possibility of using no buffer at all, we have a total of 25 possible architectures, as shown in Figure 5.1. To denote the elements of this scheme, we use the notation $(X, Y)$, where $X$ denotes the type of output buffer, and $Y$ the type of input buffer.

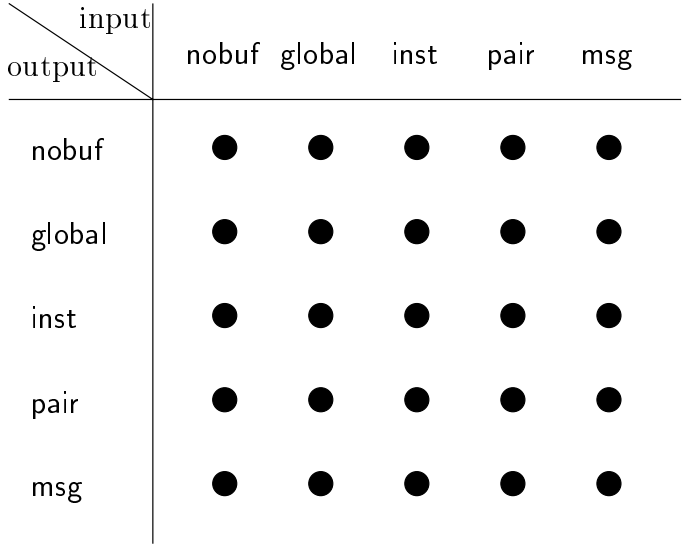



Figure 5.1: Implementation models.

In Figure 5.2 we give examples of a physical architecture of three communication models. A circle denotes an instance, an open rectangle denotes an output buffer, a filled rectangle denotes an input buffer, and an arrow denotes a communication channel. Each example contains three instances. The first example illustrates the

(nobuf,global) model. There is no output buffer, and one universal input buffer. As there is no output buffer, the messages go straight into the input buffer. This single buffer could be regarded as an output buffer as well, so this example is an illustration of (global,nobuf) too if we replace the input buffer by an output buffer. The second example shows the (global,inst) model. There is one general output buffer and every instance has a local input buffer. The third architecture is an example of the (pair,pair) model.

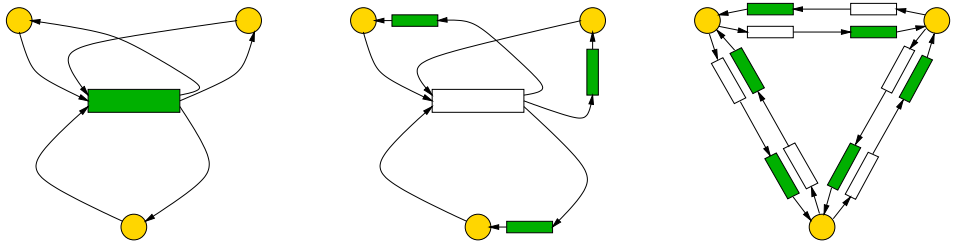


Figure 5.2: Some models: (nobuf,global), (global,inst) and (pair,pair).

Please note that not all models described in Figure 5.1 make sense to an equal degree. For example, the model (global,inst) (i.e., a shared medium for transmitting messages and an input buffer for each entity) is more natural than the exotic (global,pair) model.

Many of these architectures occur in practice as either the underlying communication architecture of a programming language or as a physical architecture. We give some examples of languages. The model (nobuf,nobuf) is typical for process algebraic formalisms based on synchronous communication, such as LOTOS [ISO88b] and ACP [BK84]. The specification language SDL [IT94, BHS91], which is closely related to MSC, has as a general communication model (pair,msg), but if we leave out the *save* construct we obtain (pair,inst) and if we also do not consider the possibility of delayed channels, we have (nobuf,inst). Some examples of physical architectures are: an asynchronous complete mesh has a (nobuf,pair) architecture, and an Ethernet connection with locally buffered input and output behaves like (inst,inst).

### 5.2.2 Extending the Semantics

In the previous section we have seen that we consider implementation models of communication in MSCs where each message passes at most two FIFO buffers. In order to reason about such implementation models we will extend the semantics of MSC in this section. In this extension of the semantics, a single communication of message $m$ will be modeled by three events. These are the events $!m$, $!!m$, and $?m$. The intuition here is, as expressed in Figure 5.3, that $!m$ denotes the putting of a message into an output buffer, $!!m$ is the transmission of the message from the output buffer to the appropriate input buffer, and $?m$ is the removal of the message from the input buffer. We assume these events to be instantaneous.

The intermediate transmit events $!!m$ play a crucial role in our description of the communication models. However, we have formulated the semantics of an MSC

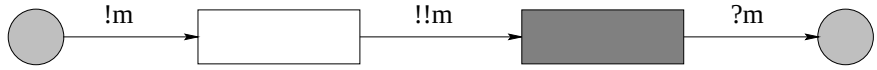



Figure 5.3: Events associated with a communication.

without using transmit events. In the remainder of this section we will define a semantics of MSC in which the transmit event occurs. The approach is similar to the previously defined semantics.

The order $<_i$ is lifted in the trivial way to the set $E_{\mathsf{impl}}(M) = \{!m, ?m, !!m \mid m \in M\}$.

We define the *output-before-transmit-before-input* order by

$$<_k^{\mathsf{oti}} = \{(!m, !!m), (!!m, ?m) \mid m \in M\},$$

and the relation $<_k^{\mathsf{impl}}$ by adding the instancewise ordering on the MSC.

**Definition 6** For a given MSC $k = \langle I, M, \mathsf{from}, \mathsf{to}, \{<_i\}_{i \in I}\rangle$, the ordering $<_k^{\mathsf{impl}}$ is defined by $<_k^{\mathsf{impl}} = (<_k^{\mathsf{inst}} \cup <_k^{\mathsf{oti}})^+$.

It is easy to see that $<^{\mathsf{msc}}$ is the restriction of $<^{\mathsf{impl}}$ to output and input events.

From an operational point of view, one can say that an MSC describes a set of traces. We distinguish msc-traces and impl-traces. An msc-trace denotes the ordering of output and input events ($!m$ and $?m$), an impl-trace those of transmit events ($!!m$) as well.

**Definition 7 (impl-traces)** An *impl-trace* is the same as an msc-trace (see Definition 4), except for the fact that it contains transmit events as well.

**Definition 8 (Trace order)** For a trace $t$ over a set of messages $M$ we define an order $<_t^{\mathsf{trace}}$ on $E_{\mathsf{impl}}(M)$, for all $1 \leq i \leq |t|$ and $1 \leq j \leq |t|$ by $t_i <_t^{\mathsf{trace}} t_j \Leftrightarrow i < j$.

**Definition 9 (MSC-trace)** A trace $t$ is said to be an impl-trace of the MSC $k$ if and only if it is defined over the messages $M$ of $k$, and $<_k^{\mathsf{impl}} \subseteq <_t^{\mathsf{trace}}$.

An impl-trace can be turned into an msc-trace by removing all transmit events ($!!m$). If, for an impl-trace $t$ this results in an msc-trace $t'$, then $t$ is said to be an extension of $t'$. It is not hard to see that an impl-trace $t$ is an MSC-trace of an MSC $k$ if and only if the trace of which it is an extension is a trace of the MSC and additionally the output-before-transmit-before-input order is respected: $<_k^{\mathsf{oti}} \subseteq <_t^{\mathsf{trace}}$.

The MSC from Figure 5.4 implies the following orderings: $!a <^{\mathsf{msc}} ?a$, $!b <^{\mathsf{msc}} ?b$, and $?a <^{\mathsf{msc}} ?b$. The first two are implied by the $<^{\mathsf{oi}}$-order, the third by the $<^{\mathsf{inst}}$-order. The MSC has exactly three msc-traces: $!a\,?a\,!b\,?b$, $!a\,!b\,?a\,?b$, and $!b\,!a\,?a\,?b$. These msc-traces can be extended to ten impl-traces, such as $!a\,!!a\,?a\,!b\,!!b\,?b$ and $!a\,!b\,!!b\,!!a\,?a\,?b$.

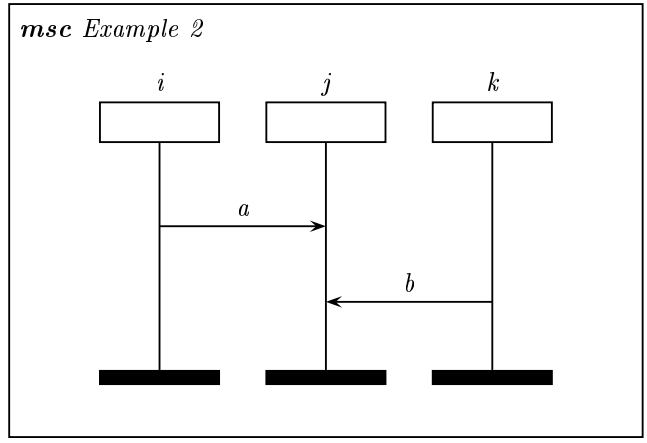



Figure 5.4: Example MSC.

### 5.2.3 Implementability

The main question of this chapter is, whether a system with a given implementation model can exhibit the behaviour described by a certain MSC. To answer this question, we first give a formal definition of what it means for a trace to have a certain implementability property. The definitions below can be seen as a formalisation of the notions introduced in Section 5.2.1.

**Definition 10 (Output-implementability)**

- nobuf-output: Every output event is directly followed by the corresponding transmit event. Thus, output and transmit events may be combined into one new event. An impl-trace $t$ is nobuf-output implementable if and only if

$$\underset{m \in M}{\forall} \neg \underset{e \in E_{\mathsf{impl}}(M)}{\exists} !m <_t^{\mathsf{trace}} e <_t^{\mathsf{trace}} !!m.$$

- global-output: The order of two output events is respected by the corresponding transmit events. An impl-trace $t$ is global-output implementable if and only if

$$\underset{m,m' \in M}{\forall} !m <_t^{\mathsf{trace}} !m' \Rightarrow !!m <_t^{\mathsf{trace}} !!m'.$$

- inst-output: The order of any two output events from the same instance is respected by the corresponding transmit events. An impl-trace $t$ is inst-output implementable if and only if

$$\underset{m,m' \in M}{\forall} \mathsf{from}(m) = \mathsf{from}(m') \Rightarrow (!m <_t^{\mathsf{trace}} !m' \Rightarrow !!m <_t^{\mathsf{trace}} !!m').$$

- pair-output: The order of two output events with the same source and the same destination, is respected by the corresponding transmit events. An impl-trace $t$ is pair-output implementable if and only if

$$\underset{m,m' \in M}{\forall} \begin{array}{l} \mathsf{from}(m) = \mathsf{from}(m') \wedge \mathsf{to}(m) = \mathsf{to}(m') \\ \Rightarrow (!m <_t^{\mathsf{trace}} !m' \Rightarrow !!m <_t^{\mathsf{trace}} !!m'). \end{array}$$

- msg-output: An impl-trace $t$ is always msg-output implementable.

For msg-output implementability we can remark that it can be put in line with the three definitions preceding it, by restating it as

$$\underset{m,m' \in M}{\forall} m = m' \Rightarrow (!m <_t^{\mathsf{trace}} !m' \Rightarrow !!m <_t^{\mathsf{trace}} !!m').$$

For nobuf-output implementability such a translation is not possible; this is qualitatively another definition. Also note that, because $<^{\mathsf{trace}}$ is a total order, $!m <_t^{\mathsf{trace}} !m' \Rightarrow !!m <_t^{\mathsf{trace}} !!m'$ is equivalent to both $!m <_t^{\mathsf{trace}} !m' \Leftrightarrow !!m <_t^{\mathsf{trace}} !!m'$ and $!m <_t^{\mathsf{trace}} !m' \Leftarrow !!m <_t^{\mathsf{trace}} !!m'$.

The input implementabilities are defined analogously.

**Definition 11 (Input-implementability)**

- nobuf-input: An impl-trace $t$ is nobuf-input implementable if and only if

$$\underset{m \in M}{\forall} \neg \underset{e \in E_{\mathsf{impl}}(M)}{\exists} !!m <_t^{\mathsf{trace}} e <_t^{\mathsf{trace}} ?m.$$

- global-input: An impl-trace $t$ is global-input implementable if and only if

$$\underset{m,m' \in M}{\forall} !!m <_t^{\mathsf{trace}} !!m' \Rightarrow ?m <_t^{\mathsf{trace}} ?m'.$$

- inst-input: An impl-trace $t$ is inst-input implementable if and only if

$$\underset{m,m' \in M}{\forall} \mathsf{to}(m) = \mathsf{to}(m') \Rightarrow (!!m <_t^{\mathsf{trace}} !!m' \Rightarrow ?m <_t^{\mathsf{trace}} ?m').$$

- pair-input: An impl-trace $t$ is pair-input implementable if and only if

$$\underset{m,m' \in M}{\forall} \begin{array}{l} \mathsf{from}(m) = \mathsf{from}(m') \wedge \mathsf{to}(m) = \mathsf{to}(m') \\ \Rightarrow (!!m <_t^{\mathsf{trace}} !!m' \Rightarrow ?m <_t^{\mathsf{trace}} ?m'). \end{array}$$

- msg-input: An impl-trace $t$ is always msg-input implementable.

Having defined formally the notions of output- and input-implementability, we now combine them and obtain our notion of communication model.

**Definition 12** An impl-trace is said to be $(X, Y)$-*implementable* (for $X, Y \in \{$nobuf, global, inst, pair, msg$\}$) if and only if it is $X$-output implementable and $Y$-input implementable. An msc-trace is said to be $(X, Y)$-*implementable* if and only if it can be extended (by adding $!!m$'s) to an impl-trace that is $(X, Y)$-implementable.

## 5.3 Classification of Implementability of Traces

To each of the implementation models defined in the previous section we can associate the set of all traces that are implementable in the model. Based on the subset relation on these sets of traces, we can order implementation models. We consider two models equivalent if they have the same set of implementable traces.

In Lemma 13 we give a classification of the notions of output-implementability. It states that a trace that is implementable on a certain architecture is also implementable on an architecture where these buffers are partitioned into buffers with a more restricted locality. For example, if a trace can be implemented on an architecture with one output buffer per instance, it can also be implemented on an architecture with an output buffer per pair of instances (provided the input buffers remain the same).

**Lemma 13 (Classification of output-implementability)**

- Every nobuf-output implementable trace is global-output implementable.
- Every global-output implementable trace is inst-output implementable.
- Every inst-output implementable trace is pair-output implementable.
- Every pair-output implementable trace is msg-output implementable.

**Proof** For impl-traces this follows directly from the definitions. For msc-traces this follows from the definition plus the fact that it holds for impl-traces. ■

The following lemmas give the orderings between the implementation models.

**Lemma 14**

- Every (inst,global)-implementable msc-trace is (inst,nobuf)-implementable.
- Every (global,global)-implementable msc-trace is (global,nobuf)-implementable.
- Every (pair,pair)-implementable msc-trace is (pair,nobuf)-implementable.
- Every (msg,msg)-implementable msc-trace is (msg,nobuf)-implementable.

**Proof** We show the proof for (inst,global). The other proofs are roughly analogous. Let $t$ be an msc-trace over the set of messages $M$, and let $t'$ be an impl-trace that is an (inst,global)-implementable extension of $t$. It suffices to construct an (inst,nobuf)-implementable extension $t''$ of $t$. We create $t''$, for which we will prove that it is (inst,nobuf)-implementable, in the following way: Starting from $t$, for each message $m \in M$ we add the transmit event $!!m$ just before the input event $?m$. This $t''$ is nobuf-input implementable by definition, so it suffices to prove that $t''$ is inst-output implementable. Thereto, let $m, m' \in M$ such that $\mathsf{from}(m) = \mathsf{from}(m')$. We have to prove that $!m <^{\mathsf{trace}}_{t''} !m' \Rightarrow !!m <^{\mathsf{trace}}_{t''} !!m'$.

Suppose that $!m <^{\mathsf{trace}}_{t''} !m'$. Then, since $t''$ is an extension of $t$, we have $!m <^{\mathsf{trace}}_{t} !m'$, and similarly, since $t'$ is an extension of $t$, $!m <^{\mathsf{trace}}_{t'} !m'$. Using that $t'$ is inst-output implementable and $\mathsf{from}(m) = \mathsf{from}(m')$ we have $!!m <^{\mathsf{trace}}_{t'} !!m'$. By using that

$t'$ is global-input implementable, we also have $?m <_{t'}^{\text{trace}} ?m'$. Since $t'$ is an extension of $t$ we have $?m <_{t}^{\text{trace}} ?m'$ and since $t''$ is an extension of $t$ also $?m <_{t''}^{\text{trace}} ?m'$. Since $t''$ is nobuf-input implementable, we obtain $!!m <_{t''}^{\text{trace}} !!m'$, which completes the proof. ■

**Lemma 15**

- Every nobuf-input implementable trace is global-input implementable.
- Every global-input implementable trace is inst-input implementable.
- Every inst-input implementable trace is pair-input implementable.
- Every pair-input implementable trace is msg-input implementable.
- Every (global,inst)-implementable msc-trace is (nobuf,inst)-implementable.
- Every (global,global)-implementable msc-trace is (nobuf,global)-implementable.
- Every (pair,pair)-implementable msc-trace is (nobuf,pair)-implementable.
- Every (msg,msg)-implementable msc-trace is (nobuf,msg)-implementable.

**Proof** Fully analogous to Lemmas 13 and 14. ■

Next, we describe how the above lemmas are useful in ordering the models. Lemma 13 provides us with a partial ordering on the various implementations: Any $(X, Y)$-implementable trace is implementable by all implementation models located to the right of or below $(X, Y)$ in Figure 5.1. Lemmas 13 to 15 give us the equivalences as expressed in Figure 5.5 by means of the clustering of implementation models.

For example, the models from the last column are equivalent. This can be seen as follows. Because of the analogue of Lemma 14, any (msg,msg)- implementable msc-trace is (nobuf,msg)-implementable, while Lemma 13 gives that any (nobuf,msg)-implementable msc-trace is ($X$,msg)-implementable, and every ($X$,msg)-implementable msc-trace is (msg,msg)-implementable.

Now we have reduced the number of implementation models to only seven different classes. Of course, some of these could still be equivalent for other reasons than the above lemmas. That this is not the case, will be seen in Corollary 20 below. We name the equivalence classes as follows: nobuf, global, inst_out, inst_in, inst2, pair, msg (see Figure 5.5).

Of these, the first two and last two will be clear immediately, inst_out means that there are instancewise output buffers and global or no input buffers, inst_in means that there are instancewise input buffers and global or no output buffers, and inst2 means that there are both an instancewise output buffer and an instancewise input buffer.

**Theorem 16** For traces, the seven implementation models are ordered as is shown in Figure 5.6.

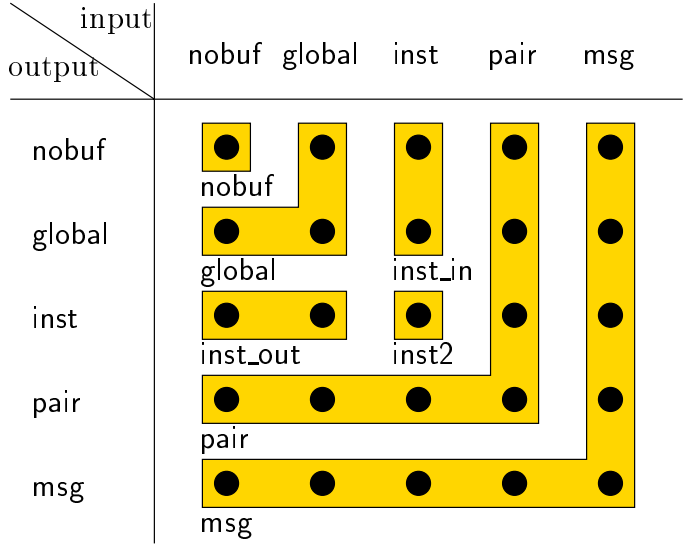



Figure 5.5: Equivalence of implementation models for traces.

**Proof** This follows from the Lemmas 13 to 15 as explained above. ■

Note that of these seven cases only inst2 is not of the form $(X, \mathsf{nobuf})$ or $(\mathsf{nobuf}, X)$. As these forms imply that there is respectively no input buffer or no output buffer, of these seven cases only the case inst2 needs two buffers, all other cases can be modelled such that each message goes through at most one buffer.

It will prove useful to have a characterisation of these implementabilities (except for inst2 of course) that does not use *transmit*s.

**Lemma 17** Let $t$ be an msc-trace over a set of messages $M$. Then:

- $t$ is nobuf-implementable if and only if

$$\bigwedge_{m \in M} \neg \bigvee_{e \in E_{\mathsf{msc}}(M)} !m <_t^{\mathsf{trace}} e <_t^{\mathsf{trace}} ?m;$$

- $t$ is global-implementable if and only if

$$\bigwedge_{m,m' \in M} !m <_t^{\mathsf{trace}} !m' \Rightarrow ?m <_t^{\mathsf{trace}} ?m';$$

- $t$ is inst_out-implementable if and only if

$$\bigwedge_{m,m' \in M} \mathsf{from}(m) = \mathsf{from}(m') \Rightarrow (!m <_t^{\mathsf{trace}} !m' \Rightarrow ?m <_t^{\mathsf{trace}} ?m');$$

- $t$ is inst_in-implementable if and only if

$$\bigwedge_{m,m' \in M} \mathsf{to}(m) = \mathsf{to}(m') \Rightarrow (!m <_t^{\mathsf{trace}} !m' \Rightarrow ?m <_t^{\mathsf{trace}} ?m');$$

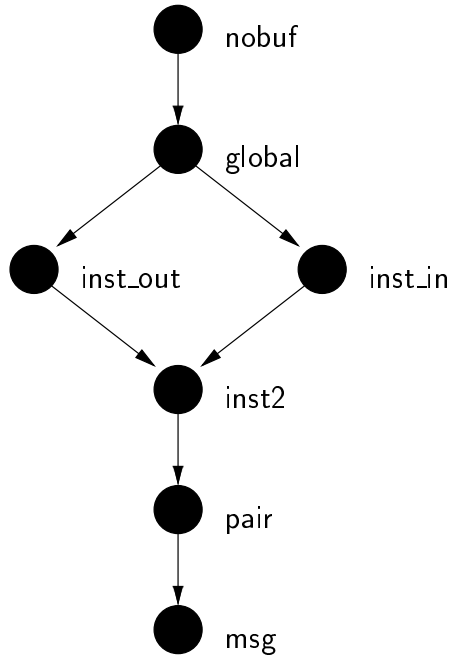



Figure 5.6: Ordering of the implementation models for traces.

- $t$ is pair-implementable if and only if

$$\underset{m,m' \in M}{\forall} \begin{array}{l} \mathsf{from}(m) = \mathsf{from}(m') \wedge \mathsf{to}(m) = \mathsf{to}(m') \\ \Rightarrow (!m <_t^{\mathsf{trace}} !m' \Rightarrow ?m <_t^{\mathsf{trace}} ?m'); \end{array}$$

- $t$ is always msg-implementable.

Again note that because $<_t^{\mathsf{trace}}$ is a total order, $\forall_{m,m' \in M} !m <_t^{\mathsf{trace}} !m' \Rightarrow ?m <_t^{\mathsf{trace}} ?m'$ can be replaced by $\forall_{m,m' \in M} !m <_t^{\mathsf{trace}} !m' \Leftrightarrow ?m <_t^{\mathsf{trace}} ?m'$ without loss of correctness.

**Proof** The proofs for this are easily found by realising that a msc-trace is ($X$,nobuf)-implementable exactly if the conditions for $X$-output implementability hold with $!!m$ everywhere replaced by $?m$. ■

## 5.4 Classification of MSCs

The use of MSCs in practice (and theory) is twofold. First, MSCs are often used to restrict the behaviour of communicating entities. In this use, it is the intention that the actual behaviour of the system is contained in the behaviour specified by the MSC. It does not mean that all behaviour of the MSC must be realised in the system. In this case only one of the traces of the MSC has to be implementable

in the given communication model. This notion of implementability is called *weak implementability*.

On the other hand, if the language MSC is used for the description of required behaviour (as for example in use cases), it is intended that each of the behaviours specified by the MSC is realised. In this case all traces of the MSC have to be implementable in the given communication model. This notion of implementability is called *strong implementability*.

We first focus on strong implementability, then on weak implementability. After this we consider the relation between classes from the strong and weak spectrum.

### 5.4.1 Strong Implementability

**Definition 18** An MSC $k$ is said to be *strongly X-implementable*, notation $X_s$-implementable, if and only if all msc-traces $t$ of $k$ are $X$-implementable.

From this definition it follows immediately that the ordering of the implementation models for traces as given in Figure 5.6 also holds for MSCs as far as strong implementability is concerned (see Figure 5.10). Next, we demonstrate that the implementation models, obtained by lifting them from the trace level to MSCs in the strong way, are indeed different. This is achieved by finding examples of MSCs that are in one class but not in another.

MSC 1 in Figure 5.7 shows an example that is $\mathsf{global}_s$-implementable, but not $\mathsf{nobuf}_s$-implementable. It is not $\mathsf{nobuf}_s$-implementable, because the trace $!a!b?a?b$ is not. The input events necessarily have to be ordered in the same way as the output events, so it is $\mathsf{global}_s$-implementable.

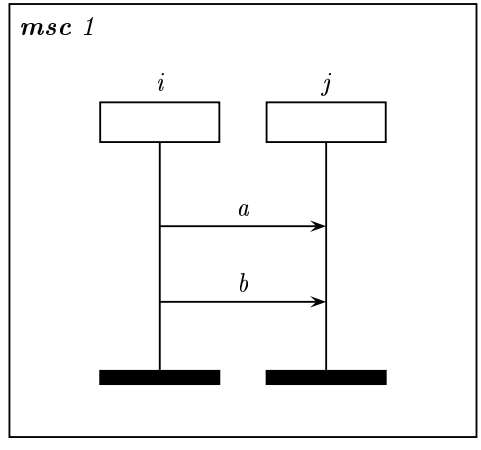



Figure 5.7: MSCs to distinguish the implementation models: strong case (1)

MSC $2a$ in Figure 5.8 is $\mathsf{inst\_out}_s$-implementable, but not $\mathsf{global}_s$-implementable due to the trace $!b!a?a?b$. That MSC $2a$ is $\mathsf{inst\_out}_s$-implementable can be seen as follows: All messages go through a different output buffer, so there is no problem

with the output buffers at all. Similarly, MSC $2b$ is $\mathsf{inst\_in_s}$-implementable, but not $\mathsf{global_s}$-implementable due to the trace $!a\,!b\,?b\,?a$.

MSCs $2a$ and $2b$ show the difference between $\mathsf{inst\_out_s}$ and $\mathsf{inst\_in_s}$. MSC $2a$ is $\mathsf{inst\_out_s}$-implementable, as mentioned before, but not $\mathsf{inst\_in_s}$-implementable. The trace $!b\,!a\,?a\,?b$ is not inst_in-implementable, because the input events of instance $j$ do not reach the input buffer in the order in which they are to be manipulated. For MSC $2b$ the reverse is the case: It is $\mathsf{inst\_in_s}$-implementable, but not $\mathsf{inst\_out_s}$-implementable. MSC $2a$ is $\mathsf{inst\_out_s}$-implementable and therefore also $\mathsf{inst2_s}$-implementable. We have already established that it is not $\mathsf{inst\_in_s}$-implementable. Similarly, MSC $2b$ is $\mathsf{inst\_in_s}$ and $\mathsf{inst2_s}$-implementable, but not $\mathsf{inst\_out_s}$-implementable. Together, these show that $\mathsf{inst\_out_s}$, $\mathsf{inst\_in_s}$ and $\mathsf{inst2_s}$ are all different.

One might suspect that the class of $\mathsf{inst2_s}$-implementable MSCs is simply equal to the intersection of the classes of $\mathsf{inst\_out_s}$-implementable and $\mathsf{inst\_in_s}$-implementable MSCs. This is not the case, as can easily be shown by combining the MSCs 2a and 2b into one MSC (see MSC 8 in Figure 5.16).

MSC 3 in Figure 5.9 is an example of an MSC that is $\mathsf{pair_s}$-implementable, but not $\mathsf{inst2_s}$-implementable. It is easy to see that it is $\mathsf{pair_s}$-implementable, because each message goes through a different buffer. Its only msc-trace is $!c\,!a\,?a\,!b\,?b\,?c$. If we try to extend this to an inst2-implementable impl-trace $t'$, we need to have $!!c <_t^{\mathsf{trace}} !!a <_t^{\mathsf{trace}} !!b <_t^{\mathsf{trace}} !!c$, which is impossible (the first $<_t^{\mathsf{trace}}$ is because of the inst-output implementability and $!c <_t^{\mathsf{trace}} !a$, the second is clearly true for every impl-trace of the MSC, and the third is because of the inst-input implementability together with $?b <_t^{\mathsf{trace}} ?c$).

Finally, MSC 4 shows the difference between $\mathsf{pair_s}$- and $\mathsf{msg_s}$-implementability. All other implementation models are also pairwise different. This result is obtained due to the transitive closure of the ordering as presented in Figure 5.10.

Together the examples used above show that if we look at strong implementability, the seven remaining implementation models are indeed different for MSCs, and thus that they are also different for msc-traces.

**Theorem 19** The implementation models for strong implementability of Figure 5.10 are different and these are ordered as expressed in Figure 5.10.

**Proof** In the above text we have demonstrated by means of counterexamples that the implementation models must be different. Also the ordering has been explained above. ■

**Corollary 20** The classes nobuf, global, inst_out, inst_in, inst2, pair, and msg are different for MSC-traces.

### 5.4.2 Weak Implementability

**Definition 21** An MSC $k$ is said to be *weakly X-implementable*, notation $X_{\mathrm{w}}$-implementable, if and only if there is an $X$-implementable msc-trace $t$ of $k$.

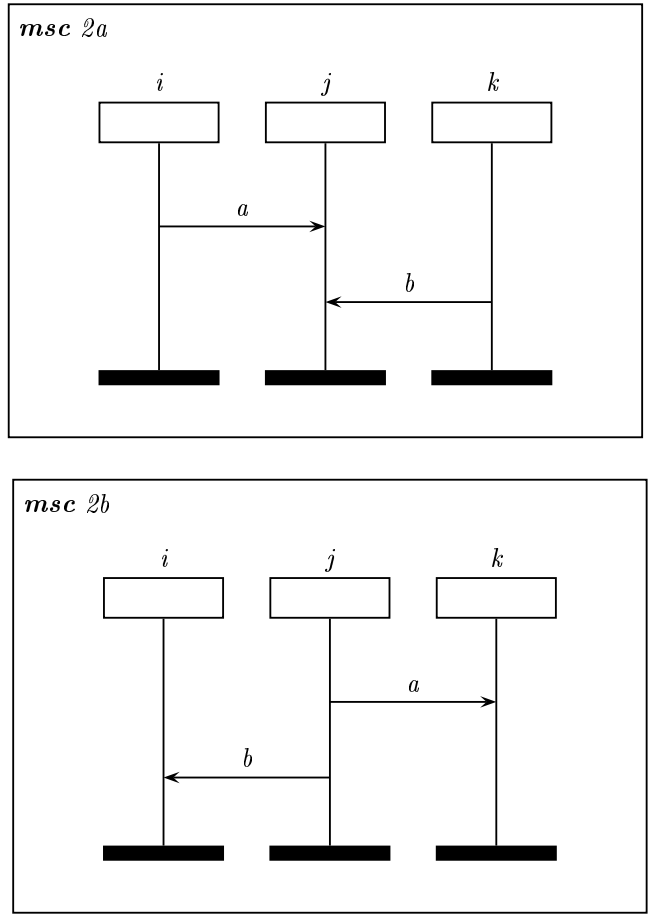
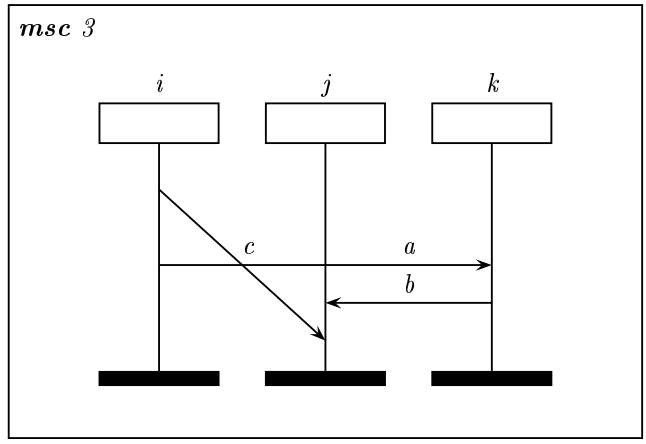



Figure 5.8: MSCs to distinguish the implementation models: strong case (2)



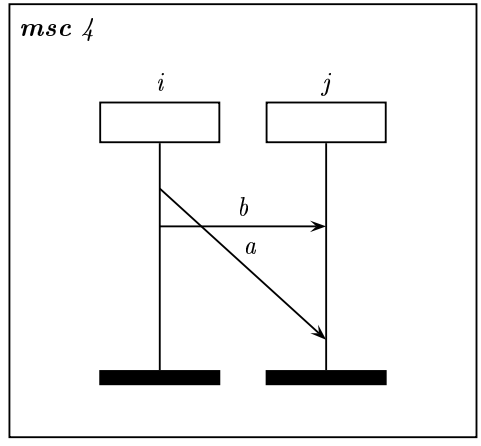




Figure 5.9: MSCs to distinguish the implementation models: strong case (3)

As was the case for strong implementability, for weak implementability we also have the ordering as expressed in Figure 5.6 as a starting point. However, using weak implementability, we do not have anymore that all implementation models differ. To see this, we first give an alternative way to characterise some of the implementations and prove that these are equivalent to the original definition.

We will use some new relations (to denote these relations we will use the same type of symbols as we have used to denote partial orders) to give this new definition. The idea is that these new relations give an ordering requirement that must be fulfilled by a trace so as to be inst_out-implementable, inst_in-implementable or inst2-implementable. For example, to be inst_out-implementable, each time two messages $m$ and $m'$ come from the same instance, they must be received in the same order as the order in which they were sent. Because they are on the same instance, there will be some $<^{\mathsf{msc}}$-order between $!m$ and $!m'$. To ensure that the trace has the receipts in the same

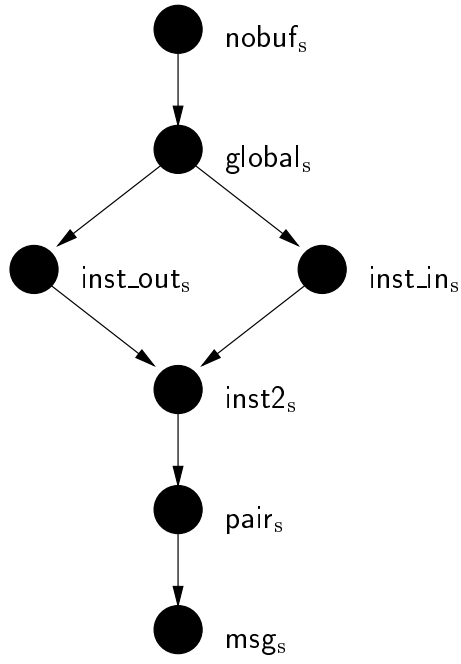



Figure 5.10: Ordering scheme for strong implementability.

order, we will have to add the equivalent order between $?m$ and $?m'$.

**Definition 22** Let $k$ be an MSC over the set of messages $M$. Then we define the relations $<_k^{\mathsf{io}}$ and $<_k^{\mathsf{ii}}$ on $E_{\mathsf{msc}}(M)$ and $<_k^{\mathsf{i2}}$ on $E_{\mathsf{impl}}(M)$ as follows:

$$<_k^{\mathsf{io}} = (<_k^{\mathsf{msc}} \cup \{(?m, ?m') \mid m, m' \in M \wedge \mathsf{from}(m) = \mathsf{from}(m') \wedge !m <_k^{\mathsf{msc}} !m'\})^+,$$

$$<_k^{\mathsf{ii}} = (<_k^{\mathsf{msc}} \cup \{(!m, !m') \mid m, m' \in M \wedge \mathsf{to}(m) = \mathsf{to}(m') \wedge ?m <_k^{\mathsf{msc}} ?m'\})^+,$$

$$\begin{aligned} <_k^{\mathsf{i2}} = (<_k^{\mathsf{impl}} \cup &\{(!!m, !!m') \mid m, m' \in M \wedge \mathsf{from}(m) = \mathsf{from}(m') \wedge !m <_k^{\mathsf{impl}} !m'\} \\ \cup &\{(!!m, !!m') \mid m, m' \in M \wedge \mathsf{to}(m) = \mathsf{to}(m') \wedge ?m <_k^{\mathsf{impl}} ?m'\})^+. \end{aligned}$$

A picture of these orderings can be seen in Figure 5.11. It shows an MSC together with its $<^{\mathsf{msc}}$, $<^{\mathsf{impl}}$, $<^{\mathsf{io}}$, $<^{\mathsf{ii}}$ and $<^{\mathsf{i2}}$ relations. For the last three, the orderings that have been added when compared to $<^{\mathsf{msc}}$ or $<^{\mathsf{impl}}$ have been dashed, while the orderings that caused these extra orderings have been drawn fat.

The inst_out-implementable traces of the MSC are also traces of the ordering $<_k^{\mathsf{io}}$ as they respect the requirements for inst_out-implementability by definition, and vice versa. Basically this is what is expressed in Lemma 23.

**Lemma 23** Let $t$ be an msc-trace of an MSC $k$. Then,

- $t$ is inst_out-implementable if and only if $<_k^{\mathsf{io}} \subseteq <_t^{\mathsf{trace}}$;

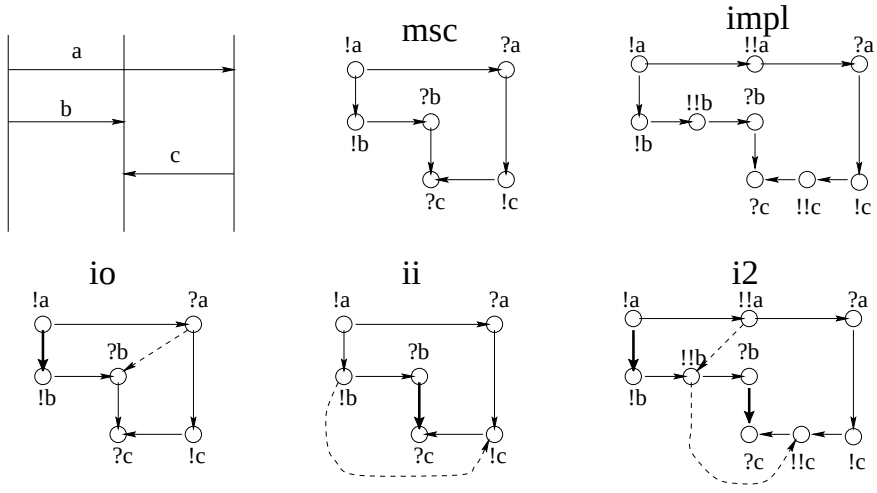



Figure 5.11: Explanation of the $<^{\mathsf{io}}$, $<^{\mathsf{ii}}$ and $<^{\mathsf{i2}}$ relations

- $t$ is inst_in-implementable if and only if $<_k^{\mathsf{ii}} \subseteq <_t^{\mathsf{trace}}$;
- $t$ is inst2-implementable if and only if there exists an extension $t'$ of $t$ such that $<_k^{\mathsf{i2}} \subseteq <_{t'}^{\mathsf{trace}}$.

**Proof** We only give the proof for the last proposition. The proofs for the first two propositions follow the same line.

First, suppose that $t$ is inst2-implementable. Then we must prove that $<_k^{\mathsf{i2}} \subseteq <_{t'}^{\mathsf{trace}}$ for some impl-trace $t'$ which is an extension of $t$. Let the impl-trace $t'$ be an arbitrary inst2-implementable extension of $t$ (the existence of such a trace follows trivially from Definition 12). Suppose that $e <_k^{\mathsf{i2}} e'$ for arbitrary events $e, e' \in E_{\mathsf{impl}}(M)$. Now it suffices to prove $e <_{t'}^{\mathsf{trace}} e'$. Since $e <_k^{\mathsf{i2}} e'$ we have the existence of events $e_1, \cdots, e_n$ such that $e \equiv e_1$, $e' \equiv e_n$ and for all $1 \leq i < n$ we have one of the following:

- $e_i <_k^{\mathsf{impl}} e_{i+1}$;
- $e_i \equiv !!m$ and $e_{i+1} \equiv !!m'$ for some $m, m' \in M$ such that $\mathsf{from}(m) = \mathsf{from}(m')$ and $!m <_k^{\mathsf{impl}} !m'$;
- $e_i \equiv !!m$ and $e_{i+1} \equiv !!m'$ for some $m, m' \in M$ such that $\mathsf{to}(m) = \mathsf{to}(m')$ and $?m <_k^{\mathsf{impl}} ?m'$.

In the first case we immediately have $e_i <_{t'}^{\mathsf{trace}} e_{i+1}$. Due to the fact that $t'$ is an inst2-implementable impl-trace, and thus both inst-output and inst-input implementable, we can conclude that $e_i <_{t'}^{\mathsf{trace}} e_{i+1}$ for the second and third case as well (see Definitions 10 and 11). Since $<_{t'}^{\mathsf{trace}}$ is transitive we have $e <_{t'}^{\mathsf{trace}} e'$, which completes this part of the proof.

Second, suppose that $<_k^{\mathsf{i2}} \subseteq <_{t'}^{\mathsf{trace}}$ for some impl-trace $t'$ which is an extension of $t$. We must prove that $t$ is (inst,inst)-implementable. Thereto, it suffices to show that $t'$ is (inst,inst)-implementable, i.e., that $t'$ is inst-output implementable and inst-input implementable. We prove that $t'$ is inst-output implementable, the proof that $t'$ is inst-input implementable is analogous. Let $m, m' \in M$ such that $\mathsf{from}(m) = \mathsf{from}(m')$. Then it suffices to show that $!m <_{t'}^{\mathsf{trace}} !m' \Rightarrow !!m <_{t'}^{\mathsf{trace}} !!m'$. Thus, suppose that $!m <_{t'}^{\mathsf{trace}} !m'$. Since $\mathsf{from}(m) = \mathsf{from}(m')$, we have $!m <_k^{\mathsf{msc}} !m'$. So $!!m <_k^{\mathsf{i2}} !!m'$. Because $<_k^{\mathsf{i2}} \subseteq <_{t'}^{\mathsf{trace}}$ we therefore have $!!m <_{t'}^{\mathsf{trace}} !!m'$. ■

Thus far, we have seen that the ordering $<_k^{\mathsf{io}}$ contains all inst_out-implementable traces of MSC $k$. An MSC $k$ is $\mathsf{inst\_out}_{\mathrm{w}}$-implementable if and only if it has a trace $t$ that is inst_out-implementable. Clearly, such a trace exists if and only if there is a trace for the ordering $<_k^{\mathsf{io}}$, in other words, if and only if $<_k^{\mathsf{io}}$ is cycle-free.

**Theorem 24** Let $k$ be an MSC. Then,

- $k$ is $\mathsf{inst\_out}_{\mathrm{w}}$-implementable if and only if $<_k^{\mathsf{io}}$ is cycle-free;
- $k$ is $\mathsf{inst\_in}_{\mathrm{w}}$-implementable if and only if $<_k^{\mathsf{ii}}$ is cycle-free;
- $k$ is $\mathsf{inst2}_{\mathrm{w}}$-implementable if and only if $<_k^{\mathsf{i2}}$ is cycle-free.

**Proof** Follows immediately from Lemma 23. ■

We use the alternative characterisations provided by Theorem 24 in the proof of the equivalence of the classes $\mathsf{inst\_out}_{\mathrm{w}}$, $\mathsf{inst\_in}_{\mathrm{w}}$, and $\mathsf{inst2}_{\mathrm{w}}$.

**Lemma 25** Let $k$ be an MSC over the set of messages $M$ and let $m, m' \in M$. If $?m <_k^{\mathsf{io}} ?m'$, then $!!m <_k^{\mathsf{i2}} !!m'$

**Proof** Suppose that $?m <_k^{\mathsf{io}} ?m'$. Then by the definition of $<_k^{\mathsf{io}}$ we have the existence of events $e_1, \cdots, e_n$ such that $e_1 \equiv ?m$, $e_n \equiv ?m'$, and for $1 \leq i < n$ we have one of the following:

- $e_i <_k^{\mathsf{msc}} e_{i+1}$;
- $e_i \equiv ?p$, $e_{i+1} \equiv ?p'$ for some $p, p' \in M$ such that $\mathsf{from}(p) = \mathsf{from}(p')$ and $!p <_k^{\mathsf{msc}} !p'$.

In the second case we have $!!p <_k^{\mathsf{i2}} !!p'$ directly from Definition 22. In the first case we have a sequence of events where the smallest steps are due to $<^{\mathsf{inst}}$ or due to $<^{\mathsf{oi}}$. In this sequence any subsequence of events which are defined on the same instance can be replaced by one single step. As a result we have the existence of messages $m_1, \cdots, m_{n'}$ such that

$$e_i \leq^{inst} !m_1 <^{\mathsf{oi}} ?m_1 <^{\mathsf{inst}} !m_2 <^{\mathsf{oi}} ?m_2 <^{\mathsf{inst}} \cdots <^{\mathsf{inst}} !m_{n'} <^{\mathsf{oi}} ?m_{n'} \leq^{inst} e_{i+1},$$

where $f \leq^{inst} f'$ is short for $f <^{\mathsf{inst}} f'$ or $f \equiv f'$. Now we observe that we only have the following three possibilities for $<^{\mathsf{inst}}$:

- $!q <^{\mathsf{inst}} !q'$ for some $q, q' \in M$ such that $\mathsf{from}(q) = \mathsf{from}(q')$. Then also $!!q <_k^{\mathsf{i2}} !!q'$ by the definition of $<^{\mathsf{i2}}$.
- $?q <^{\mathsf{inst}} ?q'$ for some $q, q' \in M$ such that $\mathsf{to}(q) = \mathsf{to}(q')$. Then also $!!q <_k^{\mathsf{i2}} !!q'$, again by the definition of $<^{\mathsf{i2}}$.
- $?q <^{\mathsf{inst}} !q'$ for some $q, q' \in M$ such that $\mathsf{to}(q) = \mathsf{from}(q')$. Then $!!q <_k^{\mathsf{impl}} ?q <_k^{\mathsf{impl}} !q' <_k^{\mathsf{impl}} !!q'$, so clearly $!!q <_k^{\mathsf{impl}} !!q'$ and $!!q <_k^{\mathsf{i2}} !!q'$ (since $<_k^{\mathsf{i2}} \subseteq <_k^{\mathsf{impl}}$).

Thus, we obtain $!!e_i <_k^{\mathsf{i2}} !!e_{i+1}$ for all $1 \leq i < n$. Therefore $!!m <_k^{\mathsf{i2}} !!m'$. ■

**Lemma 26** The implementation models $\mathsf{inst\_out_w}$, $\mathsf{inst\_in_w}$, and $\mathsf{inst2_w}$ are all equivalent.

**Proof** We show that each $\mathsf{inst2_w}$-implementable MSC is also $\mathsf{inst\_out_w}$-implementable. The reverse implication is trivial, and the proofs with $\mathsf{inst\_in_w}$ are analogous. From Lemma 24 we see that it suffices to prove that $<^{\mathsf{io}}$ is cycle-free if $<^{\mathsf{i2}}$ is cycle-free. We prove this using contraposition, so we assume that $<^{\mathsf{io}}$ has a cycle. Let $e_1 <^{\mathsf{io}} e_2 <^{\mathsf{io}} \ldots <^{\mathsf{io}} e_n <^{\mathsf{io}} e_1$ be an arbitrary cycle such that for every ordering in the cycle, say $e_i <^{\mathsf{io}} e_{i+1}$, either $e_i <^{\mathsf{msc}} e_{i+1}$, and hence $e_i <^{\mathsf{i2}} e_{i+1}$, or $e_i \equiv ?m$, $e_{i+1} \equiv ?m'$ for some $m, m' \in M$ such that $!m <^{\mathsf{msc}} !m'$ and $\mathsf{from}(m) = \mathsf{from}(m')$ (any cycle can be extended to some cycle of this form by the addition of events).

If the first is always the case, then we have a cycle in $<^{\mathsf{msc}}$, so certainly in $<^{\mathsf{i2}}$. Now assume we have the second at least once in the cycle. In that case we have at least two input events in the cycle, say $?m$ and $?m'$. Then $?m <^{\mathsf{io}} ?m'$ and $?m' <^{\mathsf{io}} ?m$. Lemma 25 gives that this implies that $!!m <^{\mathsf{i2}} !!m'$ and $!!m' <^{\mathsf{i2}} !!m$, so $<^{\mathsf{i2}}$ has a cycle. ■

Lemma 26 establishes that the classes $\mathsf{inst\_out_w}$, $\mathsf{inst\_in_w}$, and $\mathsf{inst2_w}$ are equivalent. In the remainder we denote this class by $\mathsf{inst_w}$. The remaining models are all different. MSC 3 and MSC 4 in Figure 5.9 show the difference between $\mathsf{inst_w}$ and $\mathsf{pair_w}$, and $\mathsf{pair_w}$ and $\mathsf{msg_w}$, respectively, in the weak case too (these MSCs have only one msc-trace, so their weak implementability equals their strong implementability). MSC 5 in Figure 5.12 is $\mathsf{global_w}$-implementable, but not $\mathsf{nobuf_w}$-implementable. The trace $!a\,!b\,?a\,?b$ is global-implementable, but because both output events must have been executed before any input event can be processed, there is no nobuf-implementable trace.

MSC 6 is $\mathsf{inst_w}$-implementable, but not $\mathsf{global_w}$-implementable. It is not $\mathsf{global_w}$-implementable, as can be seen thus: $!a <^{\mathsf{msc}} !b$, so if a trace $t$ of this msc is global-implementable, we must have $?a <_t^{\mathsf{trace}} ?b$. Because $!d <^{\mathsf{msc}} ?a$ and $?b <^{\mathsf{msc}} !c$, we get $!d <_t^{\mathsf{trace}} !c$. But we also have $?c <^{\mathsf{msc}} ?d$, and thus $?c <_t^{\mathsf{trace}} ?d$, from which it follows that $t$ cannot be global-implementable. On the other hand, the trace $!a\,!b\,!d\,?a\,?b\,!c\,?c\,?d$ is inst_out-implementable, so the MSC is $\mathsf{inst_w}$-implementable.

**Theorem 27** The implementation models for weak implementability of Figure 5.13 are all different and they are ordered as expressed in Figure 5.13.

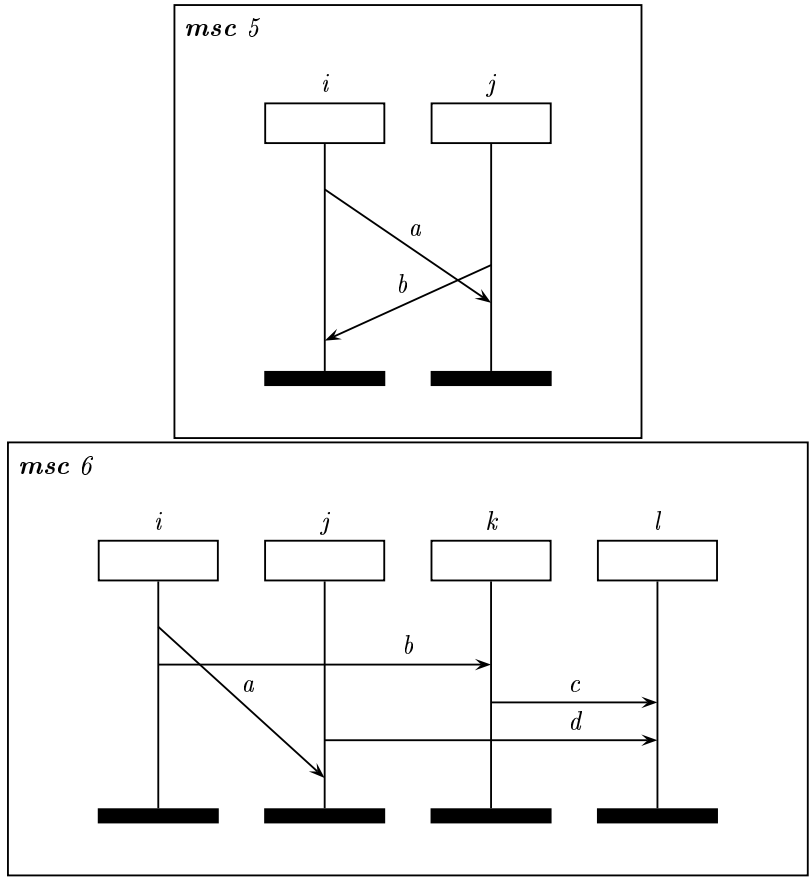



Figure 5.12: MSCs to distinguish the implementation models: weak case.

**Proof** The counterexamples that imply that the implementation models are different are given above. The ordering of the models is inherited from the ordering of the implementation models with respect to traces. Lemma 26 provides that the implementation models $\mathsf{inst\_out}_{\mathrm{w}}$, $\mathsf{inst\_in}_{\mathrm{w}}$, and $\mathsf{inst2}_{\mathrm{w}}$ are equivalent. ■

### 5.4.3 Combining the Strong and Weak Hierarchies

The relations between the classes in one of the two hierarchies have been studied extensively in the previous sections. We have 12 possible implementations left: $\mathsf{nobuf}_{\mathrm{s}}$, $\mathsf{global}_{\mathrm{s}}$, $\mathsf{inst\_out}_{\mathrm{s}}$, $\mathsf{inst\_in}_{\mathrm{s}}$, $\mathsf{inst2}_{\mathrm{s}}$, $\mathsf{pair}_{\mathrm{s}}$ and $\mathsf{msg}_{\mathrm{s}}$ in the strong case, and $\mathsf{nobuf}_{\mathrm{w}}$, $\mathsf{global}_{\mathrm{w}}$, $\mathsf{inst}_{\mathrm{w}}$, $\mathsf{pair}_{\mathrm{w}}$ and $\mathsf{msg}_{\mathrm{w}}$ in the weak case. From the definitions of strong and weak implementability it is clear that any $X_{\mathrm{s}}$-implementable MSC is also $X_{\mathrm{w}}$-implementable. The remaining classes are ordered as shown in Figure 5.14.

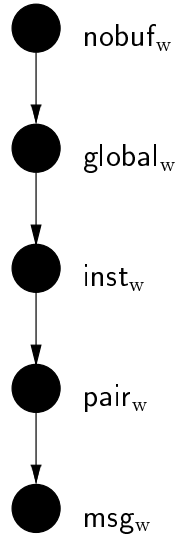



Figure 5.13: Ordering scheme for weak implementability.

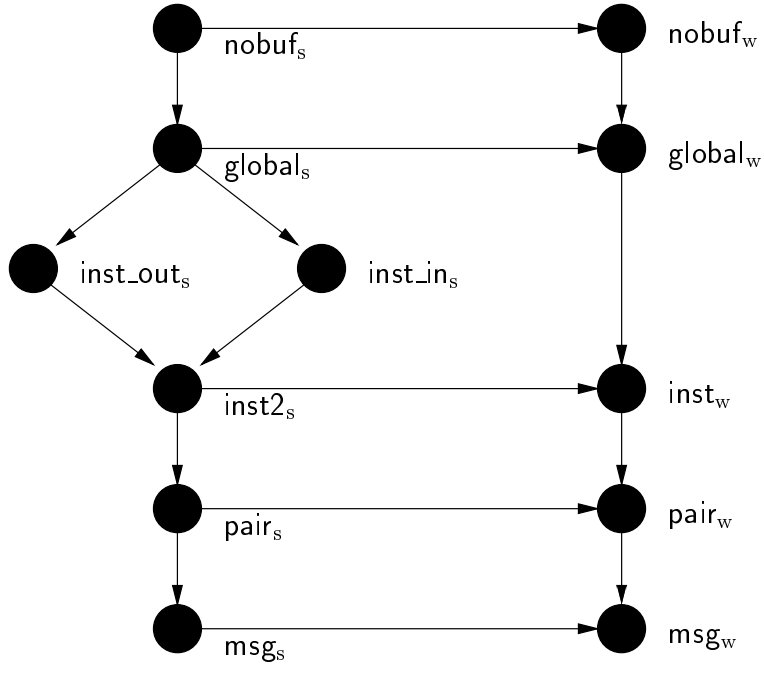



Figure 5.14: Incomplete hierarchy.

An arrow pointing from one of the classes to another means that all MSCs that are implementable in the communication model corresponding to the first class are also implementable in the communication model corresponding to the second class. Any superfluous arrows (those that can be inferred from the transitivity of the relation) have been removed.

These evident relationships between the two hierarchies have led us to the further investigation of such relationships. As it turns out there are more relationships between and identifications of the classes from the two hierarchies. First, we prove that some classes can be identified.

**Lemma 28** An MSC $k$ is $\mathsf{pair}_\mathrm{s}$-implementable if and only if it is $\mathsf{pair}_\mathrm{w}$-implementable.

**Proof** Clearly, any $\mathsf{pair}_\mathrm{s}$-implementable MSC is also $\mathsf{pair}_\mathrm{w}$-implementable. It remains to prove that any $\mathsf{pair}_\mathrm{w}$-implementable MSC is also $\mathsf{pair}_\mathrm{s}$-implementable. Let $k$ be a $\mathsf{pair}_\mathrm{w}$-implementable MSC. Let $t$ be an arbitrary msc-trace of $k$. Let $m, m' \in M$ such that $\mathsf{from}(m) = \mathsf{from}(m')$ and $\mathsf{to}(m) = \mathsf{to}(m')$. We want to prove that $!m <_t^{\mathsf{trace}} !m' \Rightarrow ?m <_t^{\mathsf{trace}} ?m'$, from which it follows that the (arbitrary) trace $t$ is pair-implementable.

Suppose that $!m <_t^{\mathsf{trace}} !m'$. Then, because $\mathsf{from}(m) = \mathsf{from}(m')$ and $!m <_t^{\mathsf{trace}} !m'$, we have $!m <_k^{\mathsf{msc}} !m'$ (when $\mathsf{from}(m) = \mathsf{from}(m')$, either $!m <_t^{\mathsf{msc}} !m'$ or $!m' <_t^{\mathsf{msc}} !m$ , and the second cannot be the case). Since $k$ is $\mathsf{pair}_\mathrm{w}$-implementable there exists a trace $t'$ that is pair-implementable. Since $!m <_k^{\mathsf{msc}} !m'$ we have $!m <_{t'}^{\mathsf{trace}} !m'$. Since $t'$ is pair-implementable we have by Lemma 17 that $?m <_{t'}^{\mathsf{trace}} ?m'$. Because $\mathsf{to}(m) = \mathsf{to}(m')$ we then have $?m <_k^{\mathsf{msc}} ?m'$. Therefore we have $?m <_t^{\mathsf{trace}} ?m'$, which completes the proof. ■

**Lemma 29** An MSC $k$ is $\mathsf{msg}_\mathrm{s}$-implementable if and only if it is $\mathsf{msg}_\mathrm{w}$-implementable.

**Proof** Trivial, because every impl-trace is msg-implementable, and thus each msc-trace is as well. ■

Lemmas 28 and 29 establish that the classes $\mathsf{pair}_\mathrm{s}$ and $\mathsf{pair}_\mathrm{w}$, and $\mathsf{msg}_\mathrm{s}$ and $\mathsf{msg}_\mathrm{w}$ are equivalent. In the remainder we denote these by pair and msg, respectively.

Next, we will prove that any $\mathsf{inst\_out}_\mathrm{s}$-implementable MSC is $\mathsf{global}_\mathrm{w}$-implementable and that any $\mathsf{inst\_in}_\mathrm{s}$-implementable MSC is $\mathsf{global}_\mathrm{w}$-implementable. To do this we first give some alternative characterisations for these implementations.

**Lemma 30** An MSC $k$ is $\mathsf{inst\_out}_\mathrm{s}$-implementable if and only if $<_k^{\mathsf{io}} = <_k^{\mathsf{msc}}$. An MSC $k$ is $\mathsf{inst\_in}_\mathrm{s}$-implementable if and only if $<_k^{\mathsf{ii}} = <_k^{\mathsf{msc}}$.

**Proof** We only give the proof for the first proposition. The proof of the second proposition follows the same lines.

First, suppose that MSC $k$ is $\mathsf{inst\_out}_\mathrm{s}$-implementable. Directly from the definition we know that $<_k^{\mathsf{msc}} \subseteq <_k^{\mathsf{io}}$, so it only remains to be proven that $<_k^{\mathsf{io}} \subseteq <_k^{\mathsf{msc}}$. Suppose that $e <_k^{\mathsf{io}} e'$ for arbitrary $e, e' \in E_{\mathsf{msc}}(M)$. Then we have the existence of $e_1, \cdots, e_n$ such that $e \equiv e_1$, $e' \equiv e_n$ and for all $1 \leq i < n$ we have one of the following:

- $e_i <_k^{\mathsf{msc}} e_{i+1}$;
- $e_i \equiv ?m$ and $e_{i+1} \equiv ?m'$ for some $m, m' \in M$ such that $\mathsf{from}(m) = \mathsf{from}(m')$ and $!m <_k^{\mathsf{msc}} !m'$.

In the second case we have, by Lemma 5, $!m <_t^{\mathsf{trace}} !m'$ for every msc-trace $t$ of $k$. Since $k$ is $\mathsf{inst\_out_s}$-implementable we have that every MSC trace of $k$ is inst_out-implementable. Thus, by Lemma 17 and the assumption that $\mathsf{from}(m) = \mathsf{from}(m')$ we have $?m <_t^{\mathsf{trace}} ?m'$ for every msc-trace $t$ of $k$. Then, again by Lemma 5, we have $?m <_k^{\mathsf{msc}} ?m'$. In the first case we already know that $e_i <_k^{\mathsf{msc}} e_{i+1}$, and taking all these steps together we have $e <_k^{\mathsf{msc}} e'$, from which it follows that $<_k^{\mathsf{io}} \subseteq <_k^{\mathsf{msc}}$.

Second, suppose that $<_k^{\mathsf{io}} = <_k^{\mathsf{msc}}$. Then we must prove that MSC $k$ is $\mathsf{inst\_out_s}$-implementable. Let $t$ be an msc-trace of $k$, and let $m, m' \in M$ such that $\mathsf{from}(m) = \mathsf{from}(m')$. Suppose $!m <_t^{\mathsf{trace}} !m'$. Then because of $\mathsf{from}(m) = \mathsf{from}(m')$, we have $!m <_k^{\mathsf{msc}} !m'$. By the definition of $<_k^{\mathsf{io}}$ we then have $?m <_k^{\mathsf{io}} ?m'$. By the assumption that $<_k^{\mathsf{io}} = <_k^{\mathsf{msc}}$, this implies $?m <_k^{\mathsf{msc}} ?m'$, and thus $?m <_t^{\mathsf{trace}} ?m'$. ■

For a similar characterisation of $\mathsf{global_w}$-implementability we define a relation $<_k^{\mathsf{g}}$.

**Definition 31** Let $k$ be an MSC. The relation $<_k^{\mathsf{g}}$ on $E_{\mathsf{msc}}(M)$ is defined as the smallest relation that satisfies:

1. $<_k^{\mathsf{msc}} \subseteq <_k^{\mathsf{g}}$;
2. $<_k^{\mathsf{g}}$ is transitive;
3. $!m <_k^{\mathsf{g}} !m' \Leftrightarrow ?m <_k^{\mathsf{g}} ?m'$ for all $m, m' \in M$.

**Lemma 32** An MSC $k$ is $\mathsf{global_w}$-implementable if and only if the relation $<_k^{\mathsf{g}}$ is cycle-free.

**Proof** First, suppose that MSC $k$ is $\mathsf{global_w}$-implementable. Let $t$ be a global-implementable trace of $k$. Then $<_t^{\mathsf{trace}}$ adheres to the restrictions in Definition 31, and thus $<_k^{\mathsf{g}} \subseteq <_t^{\mathsf{trace}}$, and $<_k^{\mathsf{g}}$ is cycle-free.

Second, suppose that the relation $<_k^{\mathsf{g}}$ is cycle-free. The idea of the proof is that we extend this relation until it is a total order. Then, if we can prove that the trace corresponding with this total order is global-implementable, we are done.

We extend the relation $<_k^{\mathsf{g}}$ to form an ordering $<$ by repeatedly choosing a smallest element that has not yet been chosen, and taking that as the next element of our total order, all the while ensuring that the preconditions of Definition 31 are still being met. More formally, we will use the following algorithm (with $S$ and $<$ as our variables):

1. $S := E_{\mathsf{msc}}(M)$, $<:= <_k^{\mathsf{g}}$
2. Let $e$ be any smallest element of $S$ with respect to $<$, that is, any element of $S$ for which there is no $e' \in S$ with $e' < e$.
3. $S := S \setminus \{e\}$
4. $<:= (< \cup \{(e, e') | e' \in S\})^+$

5. if $e \equiv !m$ for some $m \in M$, then $<:= (< \cup \{(?m, ?m') | !m' \in S\})^+$

6. Repeat steps 2 to 5 until $S = \emptyset$

We first remark that the following invariant holds: $?m <?m' \Rightarrow ?m <_k^{\mathsf{g}} ?m' \vee !m \notin S$ for all $m, m' \in M$. This clearly holds at the beginning, and only pairs $(?m, ?m')$ are added for which $!m \notin S$ since, otherwise, $?m$ would not be a smallest element of $S$. Also, after every execution of the body of the repetition (i.e. after step 5), $<$ is a total ordering on those events that are not contained in $S$.

Before we can make any arguments regarding the resulting ordering $<$, we have to prove that the algorithm is well-defined. In particular, for step 2 of the above algorithm it is necessary that $<$ is cycle-free. After step 1 $<$ is cycle-free because by the assumption $<_k^{\mathsf{g}}$ is cycle-free. There are two places where the relation $<$ is extended, namely step 4 and step 5. Step 4 maintains cycle-freeness of $<$. This can be seen as follows. Let $e$ be an arbitrary smallest element of $S$ with respect to $<$. Suppose that by adding the pairs $(e, e')$ for $e' \in S \setminus \{e\}$ to $<$ a cycle appears. Then $e' < e$ for some $e' \in S \setminus \{e\}$ which contradicts the assumption that $e$ is a smallest element of $S$ with respect to $<$.

Step 5 maintains cycle-freeness as-well. Suppose that $!m$ is a smallest element of $<$ with respect to $S$. Suppose that a cycle is introduced by step 5. This can only be the case if a pair $(?m, ?m')$ is added to $<$ for which we already had $?m' <?m$ and $!m' \in S$. By the previously mentioned invariant we have $?m' <_k^{\mathsf{g}} ?m$. By the definition of $<_k^{\mathsf{g}}$ then also $!m' <_k^{\mathsf{g}} !m$. As $!m' \in S$ this contradicts the assumption that $!m$ was a smallest element of $S$ with respect to $<$. Thus, we have established that step 2 of the algorithm is well-defined. The other steps cause no problems, so the algorithm is well-defined.

The algorithm is guaranteed to terminate as the number of elements of the finite set $S$ is decreased by one every time the body of the repetition is executed. Furthermore, because $<$ is a total order on those events that are not contained in $S$, and $S$ is empty when the algorithm ends,

Thus, upon termination of the algorithm, $<$ is a total order on $E_{\mathsf{msc}}(M)$. This total order corresponds to a trace of the MSC as $<_k^{\mathsf{msc}} \subseteq <_k^{\mathsf{g}} \subseteq <$.

All that remains to be proven is that $<$ corresponds to a global-implementable trace of $k$. Note that, after step 1, for all $m, m' \in M$ we have $!m <!m' \Rightarrow ?m <?m'$. If in step 4, an ordering $!m <!m'$ is added then in step 5 $?m <?m'$ is added. Thus, $!m <!m' \Rightarrow ?m <?m'$ is an invariant, from which it follows that the trace corresponding with $<$ is global-implementable. ■

**Lemma 33** If $e <_k^{\mathsf{g}} e'$, there is a sequence of events $e_1\, e_2, \ldots, e_n$, such that:

1. $e \equiv e_1$, $e' \equiv e_n$

2. Either $e_i <_k^{\mathsf{msc}} e_{i+1}$ or $e_i \equiv ?m$ and $e_{i+1} \equiv !m$ for a certain $m$ (for each $i \in \{1 \ldots n-1\}$)

3. The number of $e_i$'s for which $e_i \not<_k^{\mathsf{msc}} e_{i+1}$ (and thus $e_i \equiv ?m$ and $e_{i+1} \equiv !m$ hold) is less than or equal to the number of $e_i$'s for which $e_i \equiv !m$ and $e_{i+1} \equiv ?m$.

Thus, the sequence consists of $<^{\mathsf{msc}}$-orderings with additionally some messages that are passed 'in the wrong direction', but there are at least as many messages passed in the right as in the wrong direction.

As an example, look at MSC 2a in Figure 5.8. In this MSC, $!a <_k^g !b$. The sequence of events corresponding to the Lemma is $!a{\cdot}?a{\cdot}?b{\cdot}!b$. There is one message ($b$) that is passed from receipt to sending, and one message ($a$) that is passed from sending to receipt.

**Proof** In this proof we will denote the sequence $e \equiv e_1, \ldots, e_n \equiv e'$ for a given $e$ and $e'$ by $\overrightarrow{(e, e')}$. This sequence is of course in general not uniquely defined, but this does not matter for the proof.

First we note that $<_k^{\mathsf{g}}$ can be constructed by the following algorithm:

1. $<_k^{\mathsf{g}} := <_k^{\mathsf{msc}}$
2. $<_k^{\mathsf{g}} := <_k^{\mathsf{g}} \cup \{(?m, ?m') \mid !m <_k^{\mathsf{g}} !m'\}$
3. $<_k^{\mathsf{g}} := <_k^{\mathsf{g}} \cup \{(!m, !m') \mid ?m <_k^{\mathsf{g}} ?m'\}$
4. $<_k^{\mathsf{g}} := <_k^{\mathsf{g}} \cup \{(e, e') \mid \exists\, e'' : e <_k^{\mathsf{g}} e'' \wedge e'' <_k^{\mathsf{g}} e'\}$
5. Repeat steps 2 to 4 until no change occurs

We will prove that the lemma remains true throughout the running of this algorithm.

It is trivially true after step 1.

Suppose step 2 introduces a new pair into $<_k^{\mathsf{g}}$, $?m <_k^{\mathsf{g}} ?m'$. Then $!m <_k^{\mathsf{g}} !m'$ already is part of $<_k^{\mathsf{g}}$, so by induction hypothesis $\overrightarrow{(!m, !m')}$ exists. Then

$$\overrightarrow{(?m, ?m')} = (?m) {+\!\!+} \overrightarrow{(!m, !m')} {+\!\!+} (?m')$$

(where $(e_1, \ldots, e_n) {+\!\!+} (f_1, \ldots, f_n)$ is defined to be $(e_1, \ldots, e_n, f_1 \ldots, f_n)$) satisfies the requirements. There is one pair of the form $(?m, !m)$ added, but also one of the form $(!m', ?m')$, so this is ok.

Likewise, if step 3 introduces a new pair $!m <_k^{\mathsf{g}} !m'$, we can choose $\overrightarrow{(!m, !m')} = (!m) {+\!\!+} \overrightarrow{(?m, ?m')} {+\!\!+} (!m')$.

Finally, if step 4 introduces a new pair $e <_k^{\mathsf{g}} e'$, the lemma is preserved by making the choice $\overrightarrow{(e, e')} = \overrightarrow{(e, e'')} {+\!\!+} \overrightarrow{(e'', e')}$ (or rather, we should remove one of the now double $e''$ to get a correct sequence). ■

**Lemma 34** Every $\mathsf{inst\_out}_{\mathrm{s}}$-implementable MSC is $\mathsf{global}_{\mathrm{w}}$-implementable. Every $\mathsf{inst\_in}_{\mathrm{s}}$-implementable MSC is $\mathsf{global}_{\mathrm{w}}$-implementable.

**Proof** We prove this for an $\mathsf{inst\_out}_{\mathrm{s}}$-implementable MSC. The proof is completely analogous for a $\mathsf{inst\_in}_{\mathrm{s}}$-implementable MSC.

We prove this by contradiction, so we assume that $k$ is an $\mathsf{inst\_out}_{\mathrm{s}}$-implementable MSC that is not $\mathsf{global}_{\mathrm{w}}$-implementable. By Lemma 30 we have $<_k^{\mathsf{io}} = <_k^{\mathsf{msc}}$, and by Lemma 32 we have that $<_k^{\mathsf{g}}$ has a cycle.

Because $<_k^{\mathsf{g}}$ has a cycle, we can conclude from Lemma 33 that there is a cycle of steps which are either steps of $<_k^{\mathsf{msc}}$ or of the form $(?m, !m)$, where, furthermore, the number of steps of the form $(?m, !m)$ is not greater than the number of steps of the form $(!m, ?m)$. We call such a cycle a quasi-cycle of order $N$, where $N$ is the number of times that a step of the form $(?m, !m)$ occurs in the cycle.

We prove that this cycle can be changed into a quasi-cycle of order 0. Let the order be greater than 0. Because the quasi-cycle is a cycle, and contains at least one $(?m, !m)$-step and at least one $(!m, ?m)$-step, there will be at least one $(!m, ?m)$-step, such that after that $(!m, ?m)$-step a $(?m, !m)$-step will take place before the next $(?m, !m)$-step. Thus, the quasi-cycle contains a subsequence $(?m, !m, \cdots, !m', ?m')$, where there are no steps of the forms $(?m, !m)$ or $(!m, ?m)$ between $!m$ and $!m'$.

Because we have $!m <_k^{\mathsf{msc}} !m'$, by definition we get $?m <_k^{\mathsf{io}} ?m'$, from which we get $?m <_k^{\mathsf{msc}} ?m'$ from the assumption that $<_k^{\mathsf{msc}} = <_k^{\mathsf{io}}$. Thus, by removing all steps between $?m$ and $?m'$, and replacing them with a single step, we still have a cycle of $<_k^{\mathsf{msc}}$ and $(!m, ?m)$ steps, but with one less occurrence of both the type $(!m, ?m)$ and the type $(?m, !m)$. Thus, this is a quasi-cycle of order $N - 1$. Repeating this, we will finally obtain a quasi-cycle of order 0. However, a quasi-cycle of order zero is a cycle of only $<^{\mathsf{msc}}$-steps.

Thus, we see that, given the assumption, $<^{\mathsf{msc}}$ must have a cycle. This is impossible, so the assertions cannot simultaneously hold, so each $\mathsf{inst\_out}_{\mathrm{s}}$-implementable MSC is $\mathsf{global}_{\mathrm{w}}$-implementable. ■

In Figure 5.15 we give all communication models that remain after the identifications obtained until now. The arrows between these models follow also from the previous theorems and lemmas. Finally, we have to prove that the arrows between models from the strong and weak hierarchy are strict and that there are no additional arrows necessary. It suffices to show that the following arrows do not exist: $\mathsf{global}_{\mathrm{s}}$ to $\mathsf{nobuf}_{\mathrm{w}}$, $\mathsf{nobuf}_{\mathrm{w}}$ to $\mathsf{inst2}_{\mathrm{s}}$, and $\mathsf{inst2}_{\mathrm{s}}$ to $\mathsf{global}_{\mathrm{w}}$. The rest then follows because of transitivity. For example, the nonexistence of an arrow from $\mathsf{global}_{\mathrm{s}}$ to $\mathsf{nobuf}_{\mathrm{w}}$ implies the nonexistence of an arrow from $\mathsf{inst\_out}_{\mathrm{s}}$ to $\mathsf{nobuf}_{\mathrm{w}}$, because if the second arrow exists then, by transitivity, also the first must exist. Similarly we obtain the nonexistence of arrows from $\mathsf{inst\_in}_{\mathrm{s}}$ and $\mathsf{inst2}_{\mathrm{s}}$ to $\mathsf{nobuf}_{\mathrm{w}}$. We use the MSCs in Figure 5.16 to indicate that the first two arrows do not exist. MSC 7 is $\mathsf{global}_{\mathrm{s}}$-implementable, but not $\mathsf{nobuf}_{\mathrm{w}}$-implementable. It has one trace, $!a\,!b\,?a\,?b$, which is global-implementable, but not nobuf-implementable. We see that MSC 7 contains only one instance, so all messages are messages to the same instance that sent them. This is no coincidence, it can be shown that all possible counterexamples have such messages.

MSC 8 is $\mathsf{nobuf}_{\mathrm{w}}$-implementable, but not $\mathsf{inst2}_{\mathrm{s}}$-implementable. That it is $\mathsf{nobuf}_{\mathrm{w}}$-implementable can be seen from the picture, which shows that there is the trace $!a\,?a\,!b\,?b\,!c\,?c$, which is nobuf-implementable. However, the trace $!b\,!c\,?c\,!a\,?a\,?b$ is not inst2-implementable: Because $?b$ is after $?a$ in the trace, $!!b$ must be after $!!a$ to make the trace inst-input implementable, while, because $!b$ is before $!c$, $!!b$ must be before $!!c$ to make the trace inst-output implementable. However, $!!a$ must be after $!a$ and $!!c$ before $?c$, so $!!c$ will be before $!!a$ in any extension of this trace, which implies that $!!b$ cannot be both before $!!c$ and after $!!a$.

The non-existence of an arrow from $\mathsf{inst2}_{\mathrm{s}}$ to $\mathsf{global}_{\mathrm{w}}$ is taken care of by MSC

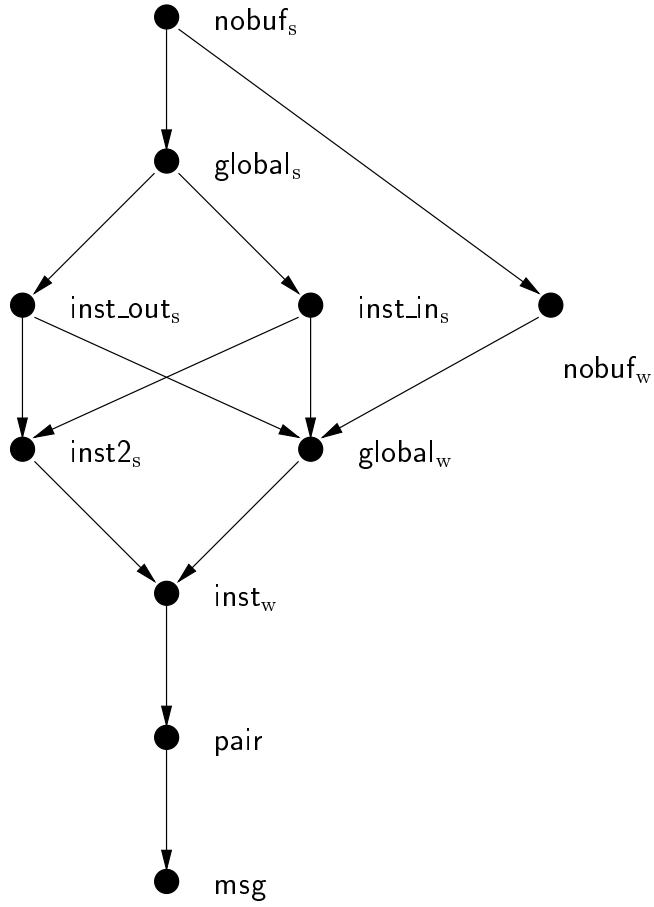



Figure 5.15: Final hierarchy.

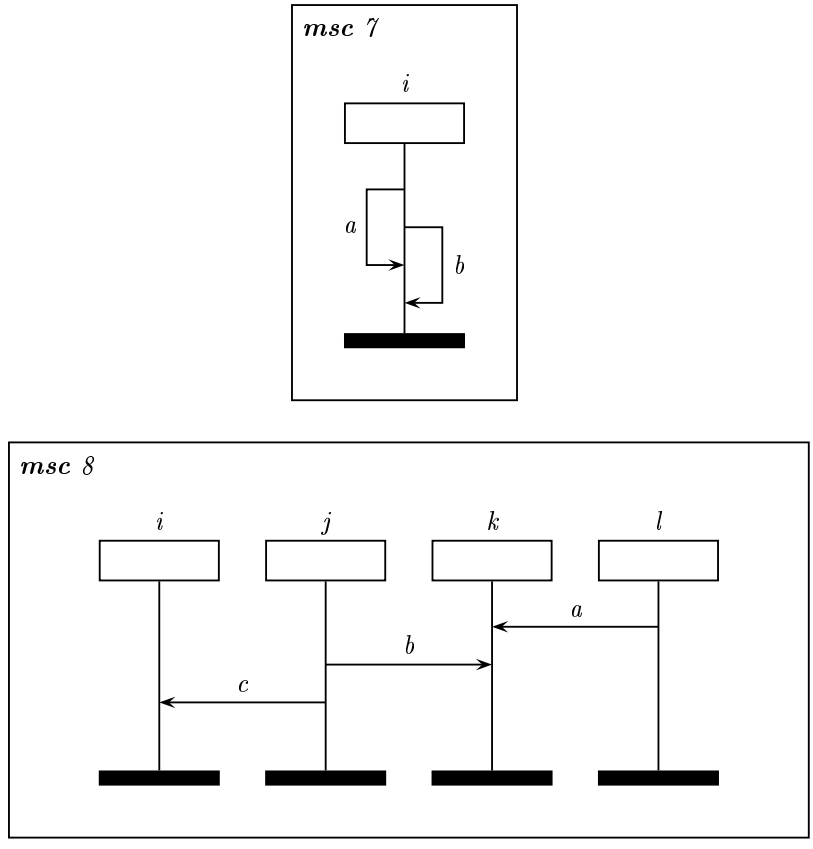



Figure 5.16: Distinguishing MSCs: comparing strong and weak.

6 in Figure 5.12. It has already been shown not to be $\mathsf{global}_{\mathrm{w}}$-implementable. It is $\mathsf{inst2}_{\mathrm{s}}$-implementable because every msc-trace of this MSC can be extended to an inst2-implementable impl-trace by adding $!!a$ and $!!b$ immediately after $!a$ and $!b$, and $!!c$ and $!!d$ immediately before $?c$ and $?d$.

**Theorem 35** The implementation models from Figure 5.15 are all different, and they are ordered as expressed in Figure 5.15.

**Proof** This has been explained in the text above. ■

## 5.5 Characterisations

Thus far, we have considered the notions of strong and weak implementability and we have ordered those in a hierarchy. In this section, we will consider how to determine

the implementability of a given MSC with respect to a given communication model. That is, we study the algorithmic aspects of the implementation models. The original definitions of the implementation models are hard to check automatically. To do so would require one to look at all traces, possibly even all impl-traces, of the MSC and check whether or not they are implementable with respect to the communication model. An MSC can have many traces, in fact their number is exponential in the number of events of the MSC.

In the previous sections, for the communication models $\mathsf{global}_{\mathrm{w}}$, $\mathsf{global}_{\mathrm{s}}$, $\mathsf{inst}_{\mathrm{w}}$, $\mathsf{inst\_out}_{\mathrm{s}}$, and $\mathsf{inst\_in}_{\mathrm{s}}$ characterisations have already been given that are easier to check. These are based on cycle-freeness of relations between the events, or the equality of two orderings. Both the creation of these relations and orderings, and checking for their cycle-freeness or equality can be done in polynomial time in the number of events. For the communication models $\mathsf{pair}$ and $\mathsf{msg}$ the fact that weak and strong implementability coincide leads directly to an easy to use characterisation: Because implementability of a single trace and implementability of all traces are equivalent, looking at one single trace suffices. Thus, we only need new characterisations for $\mathsf{nobuf}_{\mathrm{w}}$, $\mathsf{nobuf}_{\mathrm{s}}$, and $\mathsf{inst2}_{\mathrm{s}}$.

**Definition 36** Let $k$ be an MSC over the set of messages $M$. The relation $<_k^{\mathsf{w}}$ on $M$ is for all $m, m' \in M$ such that $m <_k^{\mathsf{w}} m'$ if and only if $!m <_k^{\mathsf{msc}} ?m'$ and $m \not\equiv m'$.

**Lemma 37** An MSC $k$ is $\mathsf{nobuf}_{\mathrm{w}}$-implementable if and only if the relation $<_k^{\mathsf{w}}$ is cycle-free.

**Proof** Let $k$ be an MSC over the set of messages $M$. First, suppose that $k$ is $\mathsf{nobuf}_{\mathrm{w}}$-implementable. Suppose furthermore that $<_k^{\mathsf{w}}$ has a cycle, say $m_1 <_k^{\mathsf{w}} m_2 <_k^{\mathsf{w}} \cdots <_k^{\mathsf{w}} m_n$ for some $m_1, m_2, \cdots, m_n \in M$ such that $m_1 \equiv m_n$. Then, from the definition of $<_k^{\mathsf{w}}$ and $<_k^{\mathsf{msc}}$, we obtain for all $1 \leq i < n$ that $!m_i <_k^{\mathsf{msc}} ?m_{i+1}$ and $!m_{i+1} <_k^{\mathsf{msc}} ?m_{i+1}$. Then, for every trace $t$ of $k$, we must have $!m_i <_t^{\mathsf{trace}} ?m_{i+1}$ and $!m_{i+1} <_t^{\mathsf{trace}} ?m_{i+1}$ for all $1 \leq i < n$. Since $k$ is $\mathsf{nobuf}_{\mathrm{w}}$-implementable, there is a $\mathsf{nobuf}$-implementable trace $t'$. In this trace, there can be no events between $!m_{i+1}$ and $?m_{i+1}$, so $!m_i <_{t'}^{\mathsf{trace}} ?m_{i+1}$ implies $!m_i <_{t'}^{\mathsf{trace}} !m_{i+1}$. Thus we get $!m_1 <_{t'}^{\mathsf{trace}} !m_2 <_{t'}^{\mathsf{trace}} \cdots <_{t'}^{\mathsf{trace}} !m_n$ and since $!m_1 \equiv !m_n$ we thus have a cycle of $<_{t'}^{\mathsf{trace}}$. Thus such a $\mathsf{nobuf}$-implementable trace $t'$ does not exist. This contradicts the assumption that $k$ is $\mathsf{nobuf}_{\mathrm{w}}$-implementable. Therefore, $<_k^{\mathsf{w}}$ is cycle-free.

Second, suppose that $<_k^{\mathsf{w}}$ is cycle-free. We extend $<_k^{\mathsf{w}}$ to a total order $<$, say $m_1 < m_2 < \cdots < m_n$ where $M = \{m_1, m_2, \cdots, m_n\}$. Then the trace

$$t \equiv !m_1\, ?m_1\, !m_2\, ?m_2 \cdots !m_n\, ?m_n$$

is clearly $\mathsf{nobuf}$-implementable. Thus, if suffices to prove that the trace $t$ is a trace of MSC $k$. Thereto, suppose that $e <_k^{\mathsf{msc}} e'$ for some $e, e' \in E_{\mathsf{msc}}(M)$. We distinguish four cases:

- $e \equiv !m$ and $e' \equiv !m'$ for some $m, m' \in M$. As $!m <_k^{\mathsf{msc}} !m'$ and $!m' <_k^{\mathsf{msc}} ?m'$, we also have $!m <_k^{\mathsf{msc}} ?m'$. Then, by the definition of $<_k^{\mathsf{w}}$, we have $m <_k^{\mathsf{w}} m'$, and therefore $!m <_t^{\mathsf{trace}} !m'$.

- $e \equiv !m$ and $e' \equiv ?m'$ for some $m, m' \in M$. If $m \equiv m'$, then trivially $!m <_t^{\mathsf{trace}} ?m'$. Otherwise, by the definition of $<_k^{\mathsf{w}}$, we have $m <_k^{\mathsf{w}} m'$, and therefore $!m <_t^{\mathsf{trace}} ?m'$.
- $e \equiv ?m$ and $e' \equiv !m'$ for some $m, m' \in M$. As $!m <_k^{\mathsf{msc}} ?m$, $?m <_k^{\mathsf{msc}} !m'$ and $!m' <_k^{\mathsf{msc}} ?m'$, we have $!m <_k^{\mathsf{msc}} ?m'$. Then, by the definition of $<_k^{\mathsf{w}}$, we have $m <_k^{\mathsf{w}} m'$, and therefore $?m <_t^{\mathsf{trace}} !m'$.
- $e \equiv ?m$ and $e' \equiv ?m'$ for some $m, m' \in M$. As $!m <_k^{\mathsf{msc}} ?m$ and $?m <_k^{\mathsf{msc}} ?m'$, we have $!m <_k^{\mathsf{msc}} ?m'$. Then, by the definition of $<_k^{\mathsf{w}}$, we have $m <_k^{\mathsf{w}} m'$, and therefore $?m <_t^{\mathsf{trace}} ?m'$.

In each of the four cases we have $e <_t^{\mathsf{trace}} e'$, which completes the proof. ■

**Lemma 38** If an MSC $k$ is $\mathsf{nobuf_s}$-implementable, then $<_k^{\mathsf{msc}}$ is a total order.

**Proof** Let $k$ be an MSC over the set of messages $M$. We use contraposition, so assuming that $<_k^{\mathsf{msc}}$ is not a total order, we prove that $k$ is not $\mathsf{nobuf_s}$-implementable. Let $t$ be an arbitrary msc-trace of the MSC. Because $<_k^{\mathsf{msc}}$ is not a total order, there are events $e, e' \in E_{\mathsf{msc}}(M)$ such that $e <_t^{\mathsf{trace}} e'$, but not $e <_k^{\mathsf{msc}} e'$. For any event $e'' \in E_{\mathsf{msc}}(M)$ with $e <_t^{\mathsf{trace}} e'' <_t^{\mathsf{trace}} e'$ we have either $e \not<_k^{\mathsf{msc}} e''$ or $e'' \not<_k^{\mathsf{msc}} e'$ as otherwise $e <_k^{\mathsf{msc}} e'$. So there also is a such a pair of events that are immediately after one another in the trace $t$. Then, interchanging these events would result in another trace $t'$ of the MSC. It cannot be the case that both $t$ and $t'$ are $\mathsf{nobuf}$-implementable. ■

**Lemma 39** Let $<$ be a partial order, such that $b \not< a$ and $d \not< c$, and let $<' = < \cup \{(a,b),(c,d)\}^+$ contain a cycle. Then it contains a simple cycle with both $(a,b)$ and $(c,d)$ part of this cycle.

**Proof** If $<'$ has a cycle, then so does $< \cup \{(a,b),(c,d)\}$. Look at an arbitrary simple cycle of $< \cup \{(a,b),(c,d)\}$. If this cycle did not contain $(a,b)$ or $(c,d)$, then this would also be a cycle of $<$. If it contained $(a,b)$ but not $(c,d)$, we would have $b < a$, and if it contained $(c,d)$ but not $(a,b)$, we would have $d < c$. Thus, the cycle must contain both $(a,b)$ and $(c,d)$. ■

**Lemma 40** An MSC $k$ is $\mathsf{global_s}$-implementable if and only if for all $m, m' \in M$, we either have both $!m <_k^{\mathsf{msc}} !m'$ and $?m <_k^{\mathsf{msc}} ?m'$, or we have both $!m' <_k^{\mathsf{msc}} !m$ and $?m' <_k^{\mathsf{msc}} ?m$.

**Proof** First, suppose that MSC $k$ is $\mathsf{global_s}$-implementable. Let $m, m' \in M$. Without loss of generality we may assume $!m' \not<_k^{\mathsf{msc}} !m$. Then it suffices to prove that $!m <_k^{\mathsf{msc}} !m'$ and $?m <_k^{\mathsf{msc}} ?m'$. Now we can distinguish two cases: $!m <_k^{\mathsf{msc}} !m'$ and $!m \not<_k^{\mathsf{msc}} !m'$.

Suppose that $!m <_k^{\mathsf{msc}} m'$. Then, by Lemma 5, $!m <_t^{\mathsf{trace}} !m'$ for every msc-trace $t$ of $k$. Since every msc-trace of $k$ is $\mathsf{global}$-implementable, we have by Lemma 17

that $?m <_t^{\mathsf{trace}} ?m'$ for every msc-trace $t$ of $k$. Then, again by Lemma 5, we have $?m <_k^{\mathsf{msc}} ?m'$, which completes this part of the proof.

Now, suppose that $!m \not<_k^{\mathsf{msc}} !m'$. A similar reasoning as above shows that $?m <_k^{\mathsf{msc}} ?m'$ implies $!m <_k^{\mathsf{msc}} !m'$ (remember that the single arrow in Lemma 17 is allowed to be read as a double arrow), so $?m \not<_k^{\mathsf{msc}} ?m'$, and analogously $?m' \not<_k^{\mathsf{msc}} ?m$. We will now show that there exists a msc-trace $t$ of $k$ such that $!m <_t^{\mathsf{trace}} !m'$ and $?m' <_t^{\mathsf{trace}} ?m$, thereby contradicting the assumption that $k$ is $\mathsf{global}_\mathsf{s}$-implementable.

We define the ordering $<$ as follows: $<= (<_k^{\mathsf{msc}} \cup\{(!m, !m'), (?m', ?m)\})^+$. We prove that $<$ is cycle-free, from which it immediately follows that there is a trace $t$ such that $!m <_t^{\mathsf{trace}} !m'$ and $?m' <_t^{\mathsf{trace}} ?m$ (just extend $<$ to a total order). Assume that $<$ is not cycle-free. From Lemma 39 we can conclude that there exists a simple cycle in $<$ with both $!m <!m'$ and $?m' <?m$. Because this is a simple cycle, this would imply that $!m <!m' <_k^{\mathsf{msc}} ?m' <?m <_k^{\mathsf{msc}} !m$. However, $?m <_k^{\mathsf{msc}} !m$ is impossible because $!m <_k^{\mathsf{msc}} ?m$ and $<_k^{\mathsf{msc}}$ is cycle-free. Thus, $<$ is cycle-free, which leads to a contradiction with the assumption that $k$ is $\mathsf{global}_\mathsf{s}$-implementable, so this possibility cannot occur.

Second, suppose that for all $m, m' \in M$ we have $!m <_k^{\mathsf{msc}} !m'$ and $?m <_k^{\mathsf{msc}} ?m'$, or $!m' <_k^{\mathsf{msc}} !m$ and $?m' <_k^{\mathsf{msc}} ?m$. We must prove that MSC $k$ is $\mathsf{global}_\mathsf{s}$-implementable. Let $t$ be an arbitrary msc-trace of MSC $k$. Let $m, m' \in M$. By Lemma 17 it suffices to prove that $!m <_t^{\mathsf{trace}} !m' \Rightarrow ?m <_t^{\mathsf{trace}} ?m'$. Suppose that $!m <_t^{\mathsf{trace}} !m'$. Then, by Lemma 5, $!m' \not<_k^{\mathsf{msc}} !m$. Therefore, by the assumption, $!m <_k^{\mathsf{msc}} !m'$ and $?m <_k^{\mathsf{msc}} ?m'$. So, by Lemma 5, we have $?m <_t^{\mathsf{trace}} ?m'$. ■

**Lemma 41** An MSC $k$ is $\mathsf{inst2}_\mathsf{s}$-implementable if and only if $<_k^{\mathsf{i2}} = <_k^{\mathsf{impl}}$.

**Proof** Let $k$ be an MSC over the set of messages $M$. First, suppose that $k$ is $\mathsf{inst2}_\mathsf{s}$-implementable. By definition, $<_k^{\mathsf{impl}} \subseteq <_k^{\mathsf{i2}}$, so it only remains to be proven that $<_k^{\mathsf{i2}} \subseteq <_k^{\mathsf{impl}}$. Suppose that $e <_k^{\mathsf{i2}} e'$ for some $e, e' \in E_{\mathsf{impl}}(M)$. Then we have the existence of $e_1, \cdots, e_n$ such that $e \equiv e_1$, $e' \equiv e_n$ and for all $1 \leq i < n$ we have one of the following:

- $e_i <_k^{\mathsf{impl}} e_{i+1}$;
- $e_i \equiv !!m$ and $e_{i+1} \equiv !!m'$ for some $m, m' \in M$ such that $\mathsf{from}(m) = \mathsf{from}(m')$ and $!m <_k^{\mathsf{impl}} !m'$;
- $e_i \equiv !!m$ and $e_{i+1} \equiv !!m'$ for some $m, m' \in M$ such that $\mathsf{to}(m) = \mathsf{to}(m')$ and $?m <_k^{\mathsf{impl}} ?m'$.

In the second case we use induction on the number of output events $!m''$ that can be in between $!m$ and $!m'$ to prove that $!!m <_k^{\mathsf{impl}} !!m'$.

- If there is no output event $!m''$ such that $!m <_k^{\mathsf{impl}} !m'' <_k^{\mathsf{impl}} !m'$, then either $!m <^{\mathsf{inst}} !m'$ or $?m <_k^{\mathsf{impl}} !m'$. In the first case, if $!!m <_k^{\mathsf{impl}} !!m'$ did not hold, $<_k^{\mathsf{impl}} \cup\{(!!m', !!m)\}$ would be cycle-free. Any extension of this relation to a total order would be $<_t^{\mathsf{trace}}$ for a trace $t$ that is not $\mathsf{inst}$ output-implementable,

and thus not inst2-implementable. In the second case we have $!!m <_k^{\text{impl}} ?m <_k^{\text{impl}} !m' <_k^{\text{impl}} !!m'$.

- If there is at least one output event $!m''$ such that $!m <_k^{\text{impl}} !m'' <_k^{\text{impl}} !m'$, then, using the induction hypothesis, we have $!!m <_k^{\text{impl}} !!m'' <_k^{\text{impl}} !!m'$.

For the third case a similar reasoning gives $e_i <_k^{\text{impl}} e_{i+1}$. Thus, in all cases we obtain $e_i <_k^{\text{impl}} e_{i+1}$ and therefore also $e <_k^{\text{impl}} e'$ which was to be proven.

Second, suppose that $<_k^{\text{i2}} = <_k^{\text{impl}}$. Let $t$ be an impl-trace of $k$, and let $m, m' \in M$ with $\text{from}(m) = \text{from}(m')$. We have to prove that $!m <_t^{\text{trace}} !m' \Rightarrow !!m <_t^{\text{trace}} !!m'$ for an arbitrary 3-trace $t$ of $k$.

$!m <_t^{\text{trace}} !m'$ implies $!m <_k^{\text{impl}} !m'$. Then, by the definition of $<^{\text{i2}}$, we have $!!m <_k^{\text{i2}} !!m'$. Since we assumed that $<_k^{\text{i2}} = <_k^{\text{impl}}$ we also have $!!m <_k^{\text{impl}} !!m'$, and therefore $!!m <_t^{\text{trace}} !!m'$.

The proof that $to(m) = to(m') \Rightarrow (?m <_k^{\text{impl}} ?m' \Leftrightarrow !!m <^{\text{impl}} !!m')$ is analogous, and taken together we can conclude that $k$ is $\text{inst2}_\text{s}$-implementable. ■

In the following theorem we list the characterisations for implementability we have given in this chapter and we add characterisations for the implementabilities not yet characterised. An overview is presented in Figure 5.17.

**Theorem 42**

1. An MSC $k$ is $\text{nobuf}_\text{w}$-implementable if and only if $<_k^{\text{w}}$ is cycle-free.
2. An MSC $k$ is $\text{nobuf}_\text{s}$-implementable if and only if it has exactly one trace, and that trace is nobuf-implementable.
3. An MSC $k$ is $\text{global}_\text{s}$-implementable if and only if for each pair of messages $m$ and $m'$ either both $!m <_k^{\text{msc}} !m'$ and $?m <_k^{\text{msc}} ?m'$, or both $!m' <_k^{\text{msc}} !m$ and $?m' <_k^{\text{msc}} ?m$ hold.
4. An MSC $k$ is $\text{global}_\text{w}$-implementable if and only if $<_k^{\text{g}}$ is cycle-free.
5. An MSC $k$ is $\text{inst\_out}_\text{s}$-implementable if and only if $<_k^{\text{io}} = <_k^{\text{msc}}$.
6. An MSC $k$ is $\text{inst\_in}_\text{s}$-implementable if and only if $<_k^{\text{ii}} = <_k^{\text{msc}}$.
7. An MSC $k$ is $\text{inst2}_\text{s}$-implementable if and only if $<_k^{\text{i2}} = <_k^{\text{impl}}$.
8. An MSC $k$ is $\text{inst}_\text{w}$-implementable if and only if $<_k^{\text{io}}$ is cycle-free.
9. For any trace $t$ of an MSC $k$, $k$ is pair-implementable if and only if $t$ is pair-implementable.
10. An MSC $k$ is always msg-implementable.

**Proof**

1. See Lemma 37.

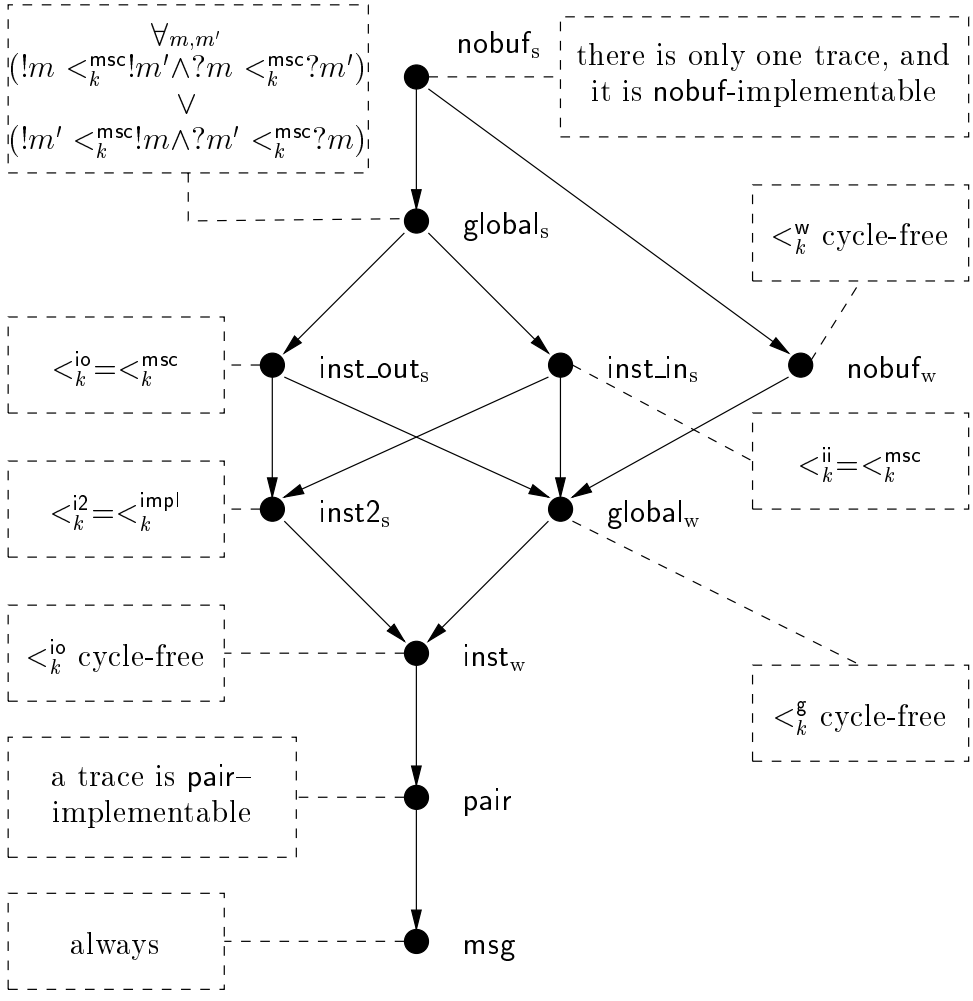



Figure 5.17: Overview.

2. If the MSC $k$ is $\mathsf{nobuf_s}$-implementable it has one trace because $<_k^{\mathsf{msc}}$ is a total order (Lemma 38).

3. See Lemma 40.

4. See Lemma 32.

5. See Lemma 30.

6. See Lemma 30.

7. See Lemma 41.

8. See Lemma 24.

9. First, if $k$ is pair-implementable, it is $\mathsf{pair_s}$-implementable and thus every trace $t$ of $k$ is pair-implementable. Second, if a randomly chosen trace $t$ is pair-implementable, then $k$ is $\mathsf{pair_w}$-implementable, and thus also $\mathsf{pair_s}$-implementable.

10. See Lemma 29.

■

## 5.6 Related Work

In this section we will compare our conclusions with those found in related literature.

In [CBMT96] Charron-Bost et al. discuss three different implementations for MSC-like diagrams: *RSC* (Realizable with Synchronous Communication), *CO* (Causally Ordered) and *FIFO*. They also define *A* (asynchronous), but this is (just like msg in our hierarchy) used to denote the set of all allowable diagrams, not some subset. They find that there is a strict ordering $RSC \subset CO \subset FIFO \subset A$. In the following, we will compare their ordering with our work.

**Theorem 43** The implementations that in [CBMT96] are named *RSC* and *FIFO* are equivalent to the implementations $\mathsf{nobuf_w}$, and pair. The implementation *CO* is strictly between the implementations $\mathsf{inst_w}$ and pair.

**Proof**

- *RSC*-$\mathsf{nobuf_w}$: Definition 3.6 in [CBMT96] states, after translating it into our terminology, that a computation is *RSC* if and only if there is a trace $t$ for which for each $m \in M$ we have that the set $\{x \in C \mid !m <_t^{\mathsf{trace}} x <_t^{\mathsf{trace}} ?m\}$ is empty, which is equivalent to the definition that is obtained by combining Lemma 17 and Definition 21.

- *FIFO*-pair: The definition for *FIFO* in [CBMT96] (Definition 3.3) translates to (by rewriting the terminology of Charron-Bost et al. in ours): $!m <_k^{\mathsf{msc}} !m' \wedge \mathsf{from}(m) = \mathsf{from}(m') \wedge \mathsf{to}(m) = \mathsf{to}(m') \Rightarrow ?m <_k^{\mathsf{msc}} ?m'$, or $\mathsf{from}(m) = \mathsf{from}(m') \wedge \mathsf{to}(m) = \mathsf{to}(m') \Rightarrow (!m <_k^{\mathsf{msc}} !m' \Rightarrow ?m <_k^{\mathsf{msc}} ?m')$, which is seen to

be equivalent to the definition in Lemma 17 once it is realised that (for the basic MSCs considered here) $\mathsf{to}(m) = \mathsf{to}(m') \Rightarrow (?m <_k^{\mathsf{msc}} ?m' \vee ?m' <_k^{\mathsf{msc}} ?m)$ and $\mathsf{from}(m) = \mathsf{from}(m') \Rightarrow (!m <_k^{\mathsf{msc}} !m' \vee\ !m' <_k^{\mathsf{msc}} !m)$.

- *CO*: That the class of pair-implementable MSCs is strictly greater than that of *CO*-implementable MSCs is shown in [CBMT96]. Remains to be shown that the class of *CO*-implementable MSCs is strictly greater than that of $\mathsf{inst}_\mathrm{w}$-implementable MSCs. The definition of *CO* as given in [CBMT96] (definition 3.4) can be translated to $\mathsf{to}(m) = \mathsf{to}(m') \wedge\ !m <_k^{\mathsf{msc}} !m' \Rightarrow ?m <_k^{\mathsf{msc}} ?m'$. An example of an MSC that is *CO*-implementable, but not $\mathsf{inst}_\mathrm{w}$-implementable, is the MSC *lobster* in Figure 5.18. It is *CO*-implementable, because there is no pair of messages with $\mathsf{to}(m) = \mathsf{to}(m')$ where $!m$ and $!m'$ are ordered, but it is not $\mathsf{inst}_\mathrm{w}$-implementable, as can for example be seen by the fact that $!a <^{\mathsf{msc}} !c$ and $!b <^{\mathsf{msc}} !d$, and thus $?a <^{\mathsf{io}} ?c$ and $?b <^{\mathsf{io}} ?d$, while at the same time we clearly have $?c <^{\mathsf{io}} ?b$ and $?d <^{\mathsf{io}} ?a$, so $<^{\mathsf{io}}$ contains a cycle.

  It remains to be proven that each $\mathsf{inst}_\mathrm{w}$-implementable MSC is *CO*-implementable. We do this using contraposition, so let $k$ be an MSC that is not *CO*-implementable. We then have that there are messages with $!m <_k^{\mathsf{msc}} !m'$, $?m \not<_k^{\mathsf{msc}} ?m'$ and $\mathsf{to}(m) = \mathsf{to}(m')$. From the last two we can derive that $?m' <^{\mathsf{msc}} ?m$, and thus we have both $?m <^{\mathsf{io}} ?m'$ (because $!m <_k^{\mathsf{msc}} !m'$) and $?m' <^{\mathsf{io}} ?m$ (because $?m' <^{\mathsf{msc}} ?m$), so the MSC $k$ is not $\mathsf{inst}_\mathrm{w}$-implementable.

■

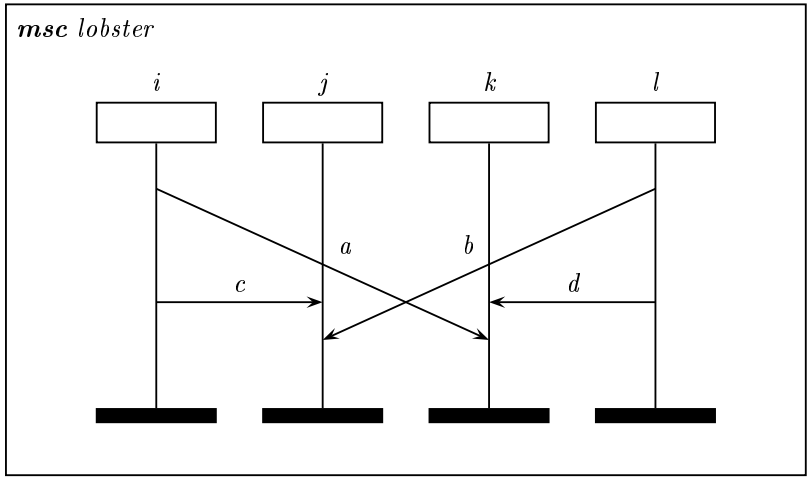



Figure 5.18: MSC to distinguish the implementation models: *CO*.

Another paper in which different communication models for MSC have been studied, is [AHP96]. Although some of their communication models are similar to some of ours, the works cannot be directly compared, because of the different focus. Whereas

our question is whether an MSC can be implemented in a given communication model, their question is whether an MSC will run as expected if it is implemented on a given communication model.

Their main point of focus is the problem of race conditions, in which two messages which might be supposed by the user to be received in the ordering prescribed by the MSC, might in reality arrive in the reverse order. The implementation model influences both which messages the user assumes to arrive in the correct order and which messages actually do.

This line of thought has been extended in [RKG97]. There a set of channels is assumed, which can be any implementation model between pair and msg (other possible models can be inserted easily, but were not looked at because of the specific subject of the paper, namely characterization of an MSC in SDL). Then, for each message it is checked which messages on the same channel that have to be dealt with later may have been received earlier.

## 5.7 Concluding Remarks and Future Research

We have considered implementation models for asynchronous communication in Message Sequence Chart. These models consist of FIFO buffers for the sending and reception of messages. By varying the locality of the buffers we have arrived, in a systematic way, at 25 models for communication. With respect to traces, consisting of putting a message into a buffer and removing a message from a buffer, there are seven different models.

By lifting this implementability notion from traces to Message Sequence Charts in two ways, strong and weak, we obtain fourteen models. After identification, ten essentially different models on the level of Message Sequence Charts remain.

For defining the models we have used the notion of impl-traces; these are a natural extension of normal MSC-traces if a message can pass two buffers on its way from source to destination.

In this chapter, we have only considered Basic Message Sequence Charts. An interesting question is how to transfer the notions and properties defined for this simple language to the complete language MSC. As many of our theorems rely on the fact that the events on an instance are totally ordered, an extension to MSC with more sophisticated ordering mechanisms (e.g., coregion and causal ordering) will imply a revision of the hierarchy. Another interesting question is whether the implementation properties are preserved under composition by means of the operators of MSC.

Furthermore, we have restricted ourselves to the treatment of architectures in which each message has exactly one possible communication path and where each such path contains at most two buffers. The extension to more flexible architectures is non-trivial and is expected to lead to an extension of the hierarchy.

An important assumption that we have made in this chapter, which is often not true in real-life examples, is the assumption of homogeneity, that is, the assumption that all instances have exactly the same type of buffers. In real life it may for example well be the case, that there is more than one channel between two instances, but some channels are still used for more than one message, thus creating an architecture

somewhere between our 'one buffer per pair' and 'one buffer per message'. This subject has been given some attention in [AHP96] and [RKG97].

Finally, our assumption of infinite FIFO buffers may be relaxed, allowing other types of buffers and buffers with finite capacity.

The results obtained in this chapter form a solid base for several applications. First, they allow us to discuss the relation between different variants of MSC, such as Interworkings [MvWW93]. Interworkings presuppose a synchronous communication mechanism. An Interworking can be considered as the restriction of the semantics of an MSC to only the nobuf-implementable traces. Thus, an MSC can be interpreted as an Interworking if and only if there is at least one such trace, i.e., the MSC is $\mathsf{nobuf}_{\mathrm{w}}$-implementable. This implies that using the theory in this chapter, a formal semantics of Interworkings can be derived in a systematic way from the semantics of MSC. We also envisage tool-oriented applications. One could for example consider a tool in which a user can select a communication model, draw an MSC and invoke an algorithm to check whether the MSC is implementable with respect to the selected model. Alternatively, the user can provide an MSC and use a tool to determine the minimal architecture, according to our hierarchy, which is needed for implementation.

Often, a user is interested in the question whether all traces of his MSC are implementable with respect to a certain architecture. We can also envisage two possible uses relying on the implementability of a single trace. First, MSCs are often used to display one single trace, for example if it is the result of a simulation run. In this case, the question is not whether the MSC is strongly or weakly implementable, but whether the implied trace is implementable (as defined in Section 5.3). Second, given an MSC, a user may want to know if at least one trace is implementable and if so, which trace that is. He is interested in a *witness*. Both applications can easily be derived from the results on weak implementability. The algorithms (see below) can easily be modified to check implementability of a given trace and to produce a witness.

A more involved application would be to use a selected communication model to reduce the set of traces defined by a given MSC to only those traces that are implementable on the given model. In this way, the semantics of an MSC would be relative to some selected model.

For most of these applications computer support would be useful. Based upon the definitions presented in this chapter, it is feasible to derive efficient algorithms. All models in the weak spectrum can be characterised in terms of the cycle-freeness of an extended ordering relation, see Theorem 42. An example of such a characterisation is given in Theorem 24. There it is stated that an MSC $k$ is $\mathsf{inst\_out}_{\mathrm{w}}$-implementable iff the ordering $<_k^{\mathsf{io}}$ (which is an extension of $<_k^{\mathsf{msc}}$) is cycle-free. Thus checking whether an MSC is $\mathsf{inst\_out}_{\mathrm{w}}$-implementable boils down to checking cycle-freeness of this relation. This immediately gives a wide range of efficient implementations for checking class-membership as many algorithms are known in literature for determining whether a given ordering is cycle-free. For the strong spectrum characterisations are given as well.

Note that the MSCs that distinguish between the different models are surprisingly simple. This indicates that the differences between the classes will appear not only in theory, but also in practice. Besides that, for these distinguishing MSCs, it is not easy

to indicate at a glance to which class they do or do not belong. This also supports our view that mechanical support for determining whether a given Message Sequence Chart belongs to a given class is necessary.

# Chapter 6

# Data in MSC

## 6.1 Introduction and History

When the new MSC2000 standard [IT00] was introduced, one of the additions to the language was the incorporation of data. In this chapter, we will look at the history of this aspect of MSC, and the way in which data languages and MSCs are actually combined, and we will try to give a semantics for data in MSC.

The first attempt to formally combine MSC with a data language was made by Grabowski, Hogrefe, Nussbaumer and Spichiger, who combined MSC and ASN.1 [GHNS95]. Baker and Jervis next presented a more generally usable data language [BJ97]. Starting from the work of Baker and Jervis, Feijs and Mauw opted for a more general approach. Instead of defining one standard language to go along with MSC, they introduced a framework in which a large range of languages could be described, thus for each of these giving the interface of the language with MSC [FM98]. This method of incorporating data has several advantages, which will be discussed in Section 6.2.

Although the MSC standardisation community agreed with this principle, there was a strong wish to extend the work of Feijs and Mauw, more specifically to introduce the possibility of dynamic data. In the original framework, variables could only be used as placeholders for an unknown value. The semantics would then be that any of the possible values for the variable could occur. There was a wish to have the possibility to assign values to a variable which then could be used later in the MSC. Feijs and Mauw, together with the current author, investigated the possibilities and problems of incorporating this possibility in a second paper [EFM99]. Partially based on this paper, in an ITU experts meeting at the Eindhoven University of Technology, the subject of data was then incorporated in the MSC2000 standard [IT00].

Another extension of the MSC language that is closely related to data, is the subject of guards. In the MSC'92 and MSC'96 standards, conditions were included, but did not have any formal meaning [IT98, Ren99], except for some static restrictions on HMSCs [Ren96]. In practice they were often used to combine MSCs – a final condition was given to some MSCs, an initial condition to others, and two MSCs could be combined (only) if the final condition of the first MSC corresponded to the initial

condition of the second [RGG96a]. This feature was originally intended to be included in the MSC language, but it was kept out of the semantics to the MSC'92 language because its meaning was felt to be insufficiently clear [MR95]. Nevertheless, some attempts have been made to provide a semantics for conditions [LL94, GHRW98]. The MSC standardisation community felt that it would be a good idea to standardise and extend this feature, by defining some conditions to act as guards, meaning that one can only pass the condition if their text is in some sense 'true'. Others would define a kind of state for the MSC, which could then be used by the guarding conditions. Obviously, the text in a guarding condition would often be some boolean expression in a data language, and thus this subject was seen as closely related to that of data.

In this chapter we will first give an overview of the problems and choices regarding data that were indicated in [EFM99], together with the choices that have been made on these points in the final MSC2000 standard, sometimes with an indication of the reasons for these choices. We will also show some of the problems that occurred especially on the subject of conditions and guards, and how these led to the actual static requirements on conditions that can be found in the standard. By showing which choices have been made, and why, the language hopefully becomes more transparent. Also, the problems that have occurred in this extension of the language may be similar to problems that occur later.

After this part, we will look at the interface as it was defined in [IT00], and make an attempt to define a formal semantics for MSC with data.

## 6.2 Reasons for Parameterisation

As was first argued by Feijs and Mauw [FM98], introducing a single data language in MSC has several adverse consequences, most or all of which can be overcome by parameterising MSC with a data language instead. Four such advantages of parameterisation can be distinguished:

1. Introducing a specific data language to MSC would inevitably be problematic for certain groups of users who are more used to different languages or different types of languages. If the data language is parameterised instead, all of these groups can use their own preferred language (as long as the language keeps to some general restrictions).

2. In a parameterised model, MSC and the data language both get their own, well described domains. Because of this, the data language does not influence the non-data parts of the MSC language, or vice versa.

3. If a specific data language is added to MSC, all its semantic problems will become problems of the MSC language as well. Although parametrisation probably does not completely solve this problem, it at least greatly diminishes the problem by looking only at the interface of the data language with MSC, and not considering the underlying aspects of the language itself.

4. If a specific data language were chosen, the MSC community would burden itself with the task of maintenance of whichever data language would be chosen. If

MSC does not couple itself to a specific data language, there is no reason for such an enterprise.

5. If through time, it was felt that another data language would have been a better choice, the language would have to be rewritten completely. This could also lead to a situation where several different versions of the language would exist, one for each data language. This situation for example happened with SDL, where there are now two versions, an original one using an algebraic specification languages, and a newer one that uses ASN.1 [Ste90], which is considered easier to use.

This parametrisation takes place through the definition of an interface, that contains all aspects of the data language that are of importance for MSC. Later in this chapter, we will show what interface has been defined in the standard [IT00], and in what way this could be used to provide a semantics for MSC with data. Before doing that, we will first discuss the choices that have been made.

## 6.3 Basic Principles

During the discussions on the new standard, the work has been guided by a number of principles. In some cases these principles clashed, and problems arose. Here, some difficult choices had to be made, as we will see later.

In the subject at hand, the following principles are of importance:

1. **Backward Compatibility**

   Old MSCs, made according to the MSC'96 or MSC'92 standard, should still be usable under the new MSC2000 standard. Within the standardisation committee a 'loose' definition of backward compatibility has been followed: it might be allowed to have the necessity to change a Message Sequence Chart made according to the MSC'96 standard, as long as there were only small, syntactic changes to be made. However, large changes, or changes that could not be made on a syntactic level were not considered acceptable.

2. **Correspondence to Existing Practice**

   Several of the subjects that are introduced, and this certainly holds for data, are already being used in existing practice. The standardisation committee wanted to link with this existing practice – the new MSC standard should include as much as possible of this existing practice. Where possible, what is already done unofficially should be made official.

3. **Semantic Clarity and Simplicity**

   When something is introduced, its semantics should be clear. That does not mean that the formal semantics should immediately be added (at the moment there are no plans known to us for a complete formal semantics of MSC2000), but it does mean that there should be a good and complete *intuition* of what those semantics should look like.

The semantics should not only be clear, they should also be simple. Creating the semantics should not be unduly hard, and it should be relatively straightforward, once the semantics have been defined, to decide what the semantics of a given MSC are. This is of importance both to people working with the language on a theoretical level and to tool builders, who want to include functionality such as the simulation of the behaviour of an MSC, which can only be done if the semantics are not too complex.

4. **Intuitive Semantics**

   In line with the second principle, the semantics should be intuitive for the user. Many MSCs, even though they include not yet officially implemented features, have a meaning that is intuitively clear. The semantics should conform to this intuitive meaning.

## 6.4 Choices

A number of choices were mentioned in [EFM99]. There, no actual choices were made, but rather, the advantages and disadvantages of the various choices were presented, to allow the standardisation committee to make a choice themselves. In this section, we will look at these choices once more with some discussion of which choice was finally made, and why.

### 6.4.1 Static vs. Dynamic Nature of a Variable

The first choice was whether the data would be static, dynamic, or somewhere in between. In [EFM99], four possibilities were distinguished, which, from most static to most dynamic, were called:

1. fully static variables
2. parameter variables
3. single time assignable variables
4. multiple times assignable variables

**Fully static variables:** In a fully static environment, the values of variables are either completely pre-determined, or not defined at all. In the latter case the semantics is taken to consist of *all* possible behaviours for *any* valuation of the various variables. This is the situation that was covered by [FM98]. This way of including data causes the fewest problems. However, the price that has to be paid is that it also provides the least expressive power.

**Parameter variables:** Parameter variables play a role within HMSC or MSC reference expressions, the idea being that one provides a value for one or more variables while calling the reference MSC. In calling the MSC, the values of some variables are given as parameters to the reference. An example of how this works is shown in Figure 6.1.

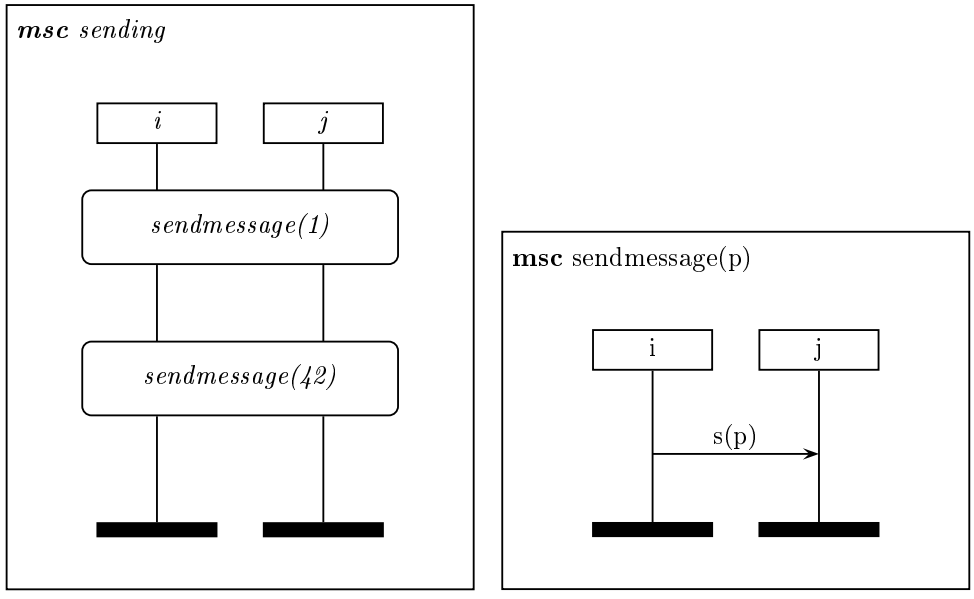



Figure 6.1: Parametric Data

The MSC *sending* calls the MSC *sendmessage* twice, but the first time with *sendmessage(1)* and the second time with *sendmessage(42)*. The effect of this, is that the MSC *sendmessage* is used with the value of 1 for $p$ the first time, and the value of 42 for $p$ the second time. Thus, first the message $s(1)$ is sent and received, then $s(42)$. Compared to the next two options with assignable variables, Parametric data is semantically less complicated, because it is easier to connect the occurences of variables with the place where they get their values. On the other hand, it is also less powerful. Changing the value of a variable, certainly where the new value depends on the old one, is difficult and counter-intuitive, if not downright impossible, when using parametric data. Of course introducing both options results in an even greater power of expression.

**Single time assignable variables:** Here a variable can be assigned at any place in the MSC, but once it is assigned, it cannot get a new value. So each time a variable is accessed, it will still have the same value. This will solve some of the problems with variables that are shown below, because these typically occur when the value of a variable is changed before it is used, or between several uses of the same variable. However, to actually make this solution work, one would also have to take steps to make the usage of a variable before its assignment impossible, otherwise one still has to deal with two different values – an indeterminate one before it is assigned, and a determinate one afterwards.

**Multiple times assignable variables:** Here, a variable can at any time be given a value, and this value can be changed later. This choice offers most possibilities for

the user, but at the same time also complicates the language more than other options. A first problem is shown in Figure 6.2.

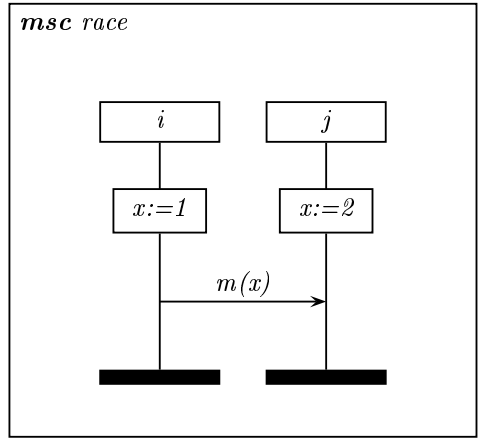



Figure 6.2: Problematic MSC (simple case)

What message is to be sent here? $m(1)$ or $m(2)$? Or does it depend on the order in which the assignments happen? And what if the change of the value of $x$ from 1 to 2 happens between sending and receiving of the message? Will we have to remember the old value of $x$?

Despite these problems, which were mentioned in [EFM99], it was nevertheless felt that multiple-times assignable variables were needed, because users would like to have them. To keep this type of 'race conditions' manageable, it was decided to have one specific instance should act as the 'owner' of a variable. The variable could only be changed on this instance.

The first idea was that the usage of such a variable would also be restricted to that same instance. No other instance could use the value of the variable. However, this proved too restrictive. From a usage point of view, it seemed necessary that an MSC like the one in Figure 6.3 could be drawn.

The idea here, is that at one point the value of $x$ is decided, which then through a series of messages is sent to another part of the system described by the MSC. To enable this, it was decided that the value of variables of one instance could be used at another instance, provided that the value had been sent to that instant through messages. Thus, the value of a variable on an instance other than the owner is changed only when this instance receives the value through some message. Not only does this make the semantics more intuitive than one in which a change in value of a variable is 'magically' copied to all instances of the MSC, it also is more in line with the basic semantic ideas behind MSC: any transfer of information from one instance to another should be explicitly shown. Introduction of a shared variable paradigm goes against the spirit of MSC.

Because their added value in expressiveness of the language seemed useful, while they did not add more problems than the ones that already had to be solved for the

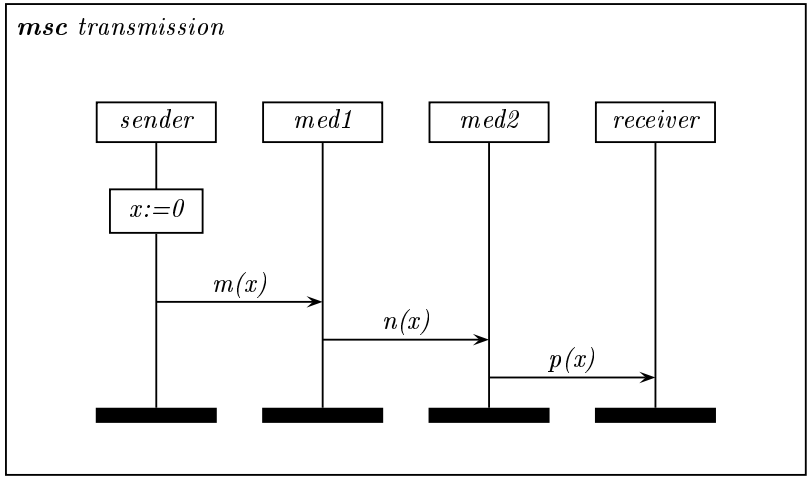



Figure 6.3: Transmission of variables

assignable variables, parametric data have also been included in the MSC language. The variables that are defined in this way are called 'static variables'. That is, static variables are variables that are given as a parameter to an MSC, and whose value is determined when the MSC is called.

### 6.4.2 Binding of Variables

Another matter was, how these variables could change their values. For static variables this is obvious: They receive their value because the MSC is called with a certain actual parameter list. For dynamic variables, we have to distinguish between owning and not-owning instances.

An instance that is not the owning instance of the variable receives a new value for the variable when a message arrives that contains that variable in its parameter list. The new value of the variable, as seen by that instance, will be the value the variable had on the sending instance at the time of sending.

For the owning instance, two major ways of giving values to variables were discussed in [EFM99]. Both have been included in the MSC language. The most obvious is through a direct binding. This occurs when a text like $x := 2$ is found inside an action box. A second way is the propagation of a message through a gate. This is best explained with an example, see Figure 6.4.

The incoming message here gives $x$ the value of 3. Of course this can only be done if $x$ is a variable of the receiver. In all of these cases, the outgoing message (the message in the MSC '`send`') should contain some expression as one of its parameters, while the incoming message (the same message in the MSC '`receive`') should contain a variable to be bound. This can also be done with MSC reference expressions, see Figure 6.5.

An instance that does not own the variable, changes its (local) value when it

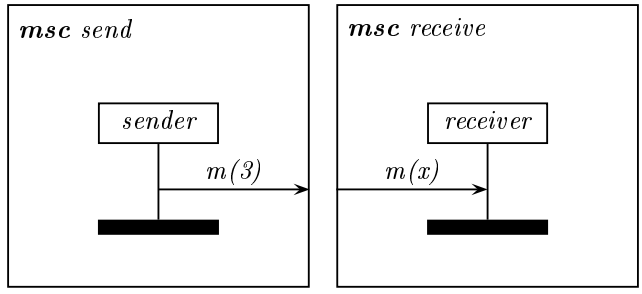



Figure 6.4: Assignment through gated messages

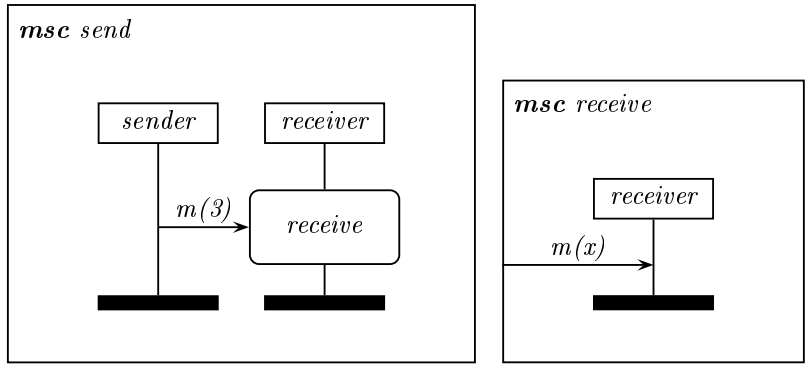



Figure 6.5: Variable assignment for MSC reference expressions

receives a message that has that variable in one of its parameter expressions. It is then changed to the value that this variable has on the sending instance. Although this has been designed to correspond to existing practice, it can still lead to results that some may regard counter-intuitive, as can be seen in Figure 6.6.

Assume that $x$ is a variable owned by instance $i$.

Intuitively, one might expect that $k$ always sends back $ack(2)$, because 2 is the latest value of $x$, and $k$ has already received that value. However, $k$ does not 'know' which is the latest value, so it will always assume that $x$ has the last value that it has received. Thus, if $n(1)$ arrives at $k$ after $m'(2)$, $ack(1)$ rather than $ack(2)$ will be sent.

### 6.4.3 Undefined Variables

Another problem raised in [EFM99] was what one had to do with variables being used before they have any value assigned to them. The main choices possible were:

1. Disallow this by static rules

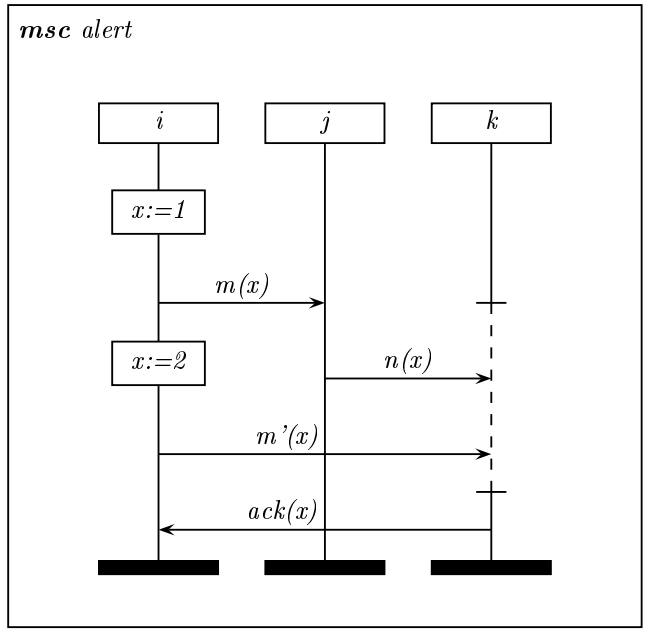



Figure 6.6: A counter-intuitive case

2. Regard it as a semantic deadlock
3. Regard all undefined variables as universally quantified
4. Use default values for each variable

All of these options have their drawbacks (the first may be hard, the second leads to non-intuitive semantics, the third leads to an explosion of the number of possible traces and the fourth extends the interface with the data language by a default value for each domain, see [EFM99] for a more extensive discussion). In MSC, the first option has been chosen (see [IT00], section 5.4: *In a defining MSC there must be no trace through an MSC in which a variable is referenced without being defined.*)

### 6.4.4 Scope of a Variable

A further point on which different choices could have been made, is in the definition of the scope of a variable. That is, once a variable has been declared, on which part of the MSC can it be used? This is the scope of that variable. Scopes might be nested, in which case the variables in the outer scope can also be used in the inner scope, unless a new declaration of the same variable has taken place. If a variable is used in two different scopes, then the two uses of the variable have nothing in common, and they should be regarded as two different variables that happen to share the same name.

We can distinguish two different dimensions to the scope: Block scope and architectural scope.

The block scope of a variable is a separated (framed) part of an MSC where the variable is defined. For most variables the block scope has been chosen to be the complete MSC document; however for static variables it consists only of a single MSC.

Apart from this there is also the architectural scope. This gives the locality with respect to the instances in an MSC. One could specify that variables exist on only one instance, or on all instances of the MSC. Possibilities in between, where a variable is defined on a number of instances (for example, the instances that reside on one processor), could also be considered, although it might be harder to find a syntax for that option.

Again, this has been done differently for dynamic and static variables. Dynamic variables, as mentioned before, have a local architectural scope, although 'copies' of the value of a variable may be present on other instances. On the other hand, static variables have global architectural scope.

## 6.5 Guards

The largest problems with data were encountered when trying to introduce guards. In the older versions of the MSC standard, MSC'92 [IT93] and MSC'96 [IT96], conditions had very little function. Semantically, they had no meaning at all, except for statically forbidding some HMSCs (in MSC'96), and thus they formally were no more than comments. However, in practice conditions were being used in a more functional manner, namely to create MSCs like those in Figures 6.7 through 6.9.

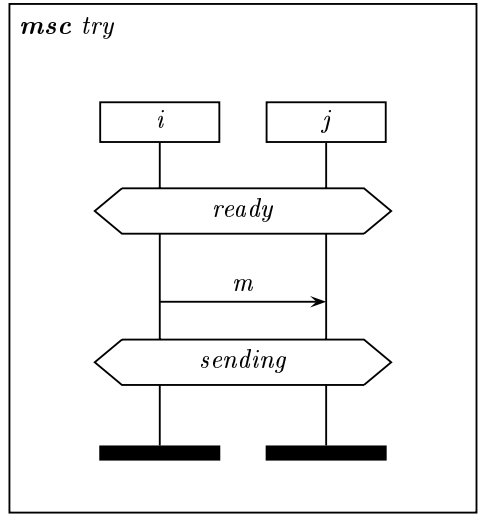



Figure 6.7: Conditions as a coupling mechanism (1)

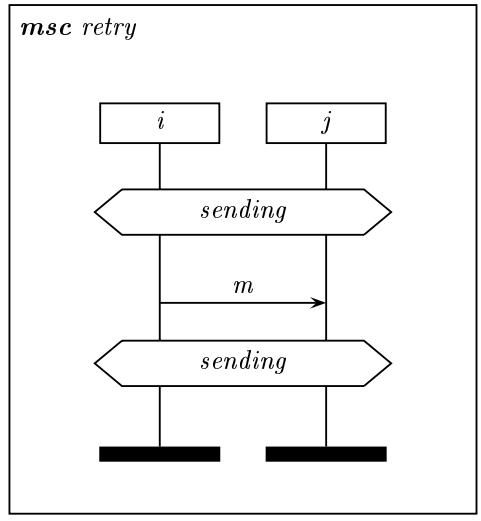



Figure 6.8: Conditions as a coupling mechanism (2)

The conditions here function as a method to decide in which order the MSCs are gone through. More precisely, one may go from one MSC to another if, and only if, the second MSC starts with the same condition as the first ends with. Thus, after finishing MSC *try*, one can go to *retry* or *success*, and likewise after MSC *retry*, while after MSC *success* one can only go to MSC *try*.

### 6.5.1 Including Guards in the Language

An attempt was made to formalise this method of using conditions. Certain conditions would act as guards, others as defining conditions. One can only pass through a guard if it is equal to the last defining condition that was encountered. The above MSCs would then become a correct MSC when included in an HMSC like the one in Figure 6.10.

Of course, the HMSC in Figure 6.10 is not very clear. There are a number of paths which seem to be present from the HMSC description, but are made impossible by the guards. One would prefer to use an HMSC like the one in Figure 6.11, which shows the order in which the parts of the HMSC are passed explicitly. This is done by adding a node for each state, and connecting the MSCs with the node corresponding to each of its start and final state. However, with the MSCs given, the HMSCs of Figures 6.10 and 6.11 are semantically equivalent – all additional connections in Figure 6.10 are without effect; they cannot actually be taken because the states do not coincide. To minimize an HMSC by removing superfluous edges, such as the transformation from Figure 6.10 into Figure 6.11, might be a useful task for tool support. The process is similar to that which is used in [Mei00] to create a so-called 'connectability diagram'.

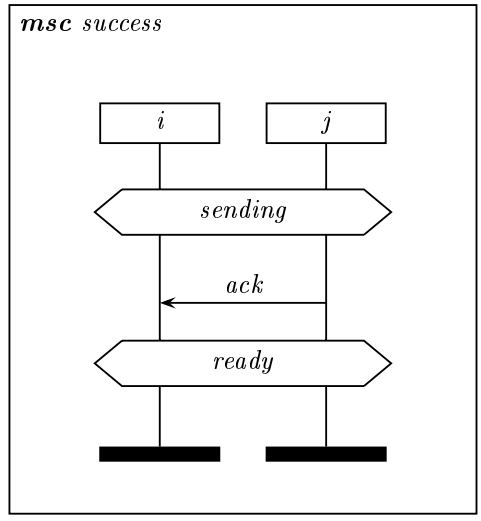



Figure 6.9: Conditions as a coupling mechanism (3)

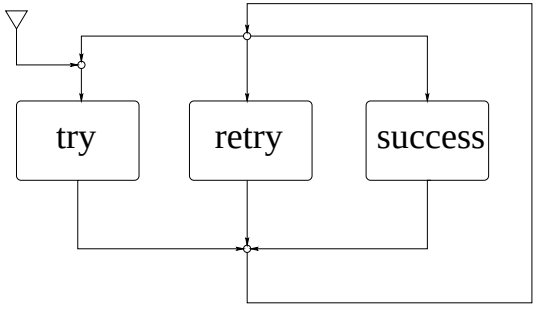



Figure 6.10: An HMSC for Figures 6.7 to 6.9

The conditions at the bottom of the MSCs should be defining conditions, and the conditions at the top should be guards. To avoid any confusion about which conditions are defining conditions and which are guards (confusion could for example arise when a condition only covers instances that have no events within the MSC), all guards have the keyword '**when**' added to them. Defining conditions have no additional keyword. Thus, in the above case, apart from adding the HMSC one should also change '*ready*' and '*sending*' into '**when** *ready*' and '**when** *sending*' in the conditions at the tops of the MSCs, while leaving the conditions at the bottoms as they are.

When combining these guarding conditions with data, the possibility arises to use data expressions as guards. More precisely, if a data expression can have boolean values (true or false), it could be used as a guard, which then can only be passed if it is true. In this way, behaviour that depends on the value of variables can be described.

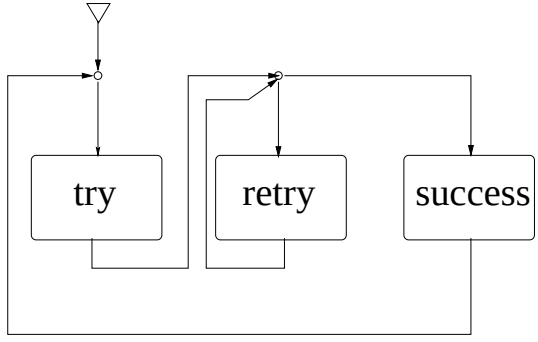



Figure 6.11: A more readable HMSC

An example is in Figure 6.12, where it is checked whether a certain variable is zero.

### 6.5.2 Semantic Proposals

Unfortunately, this option also resulted in some semantic problems. Central question was, at which time the guard would be evaluated.

Figure 6.13 shows the problem. Suppose that this MSC starts with $x := 1$, and after that the message $m$ is sent and received. Which of the messages $a$ and $b$ (if any) can then be sent? There existed two schools of thought, neither of which in the end prevailed. Instead, some kind of compromise was made, which will be described in Section 6.5.3

The first school of thought held to the principle that one should look at the value of the variable at the time the first instance is trying to pass the guard. In this case, because the first action after the guard is either the sending of $a$ or the sending of $b$, the deciding factor is the value of the variable $x$ at the time either $a$ or $b$ is sent. If one of the messages is sent before $x := 2$ is executed, it must be $a$. After $x := 2$ has been executed, only $b$ can be sent.

On the other hand, the second school of thought preferred a more syntactic look at the MSC. Because the $x := 2$ is above the guard, and on the same instance, it seems logical to regard it as happening before the guard, and thus influencing the choice. In this interpretation, only $b$ can be sent in this MSC.

The disagreement can be described as a disagreement on the time and place where the guard is evaluated. In the first proposal, the guard is evaluated by the first instance to do an action after the guard, the principle could be called 'first past the post'. In the second proposal, the guard is evaluated by the instance that owns the variables in the guard.

The arguments for the two proposals came from two of the principles mentioned before.

The first proposal was based on semantic simplicity. Adopting the second proposal would require that one first calculates the full semantics of an MSC without looking at the guard, then removes the traces that do not adhere correctly to the guards present. This would make it impossible to find out the next event in an MSC trace

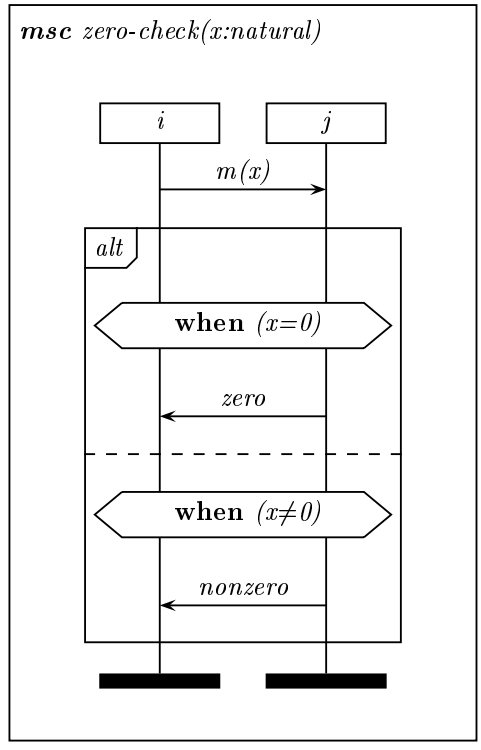



Figure 6.12: Usage of data in guards

without first calculating the complete semantics of the MSC. This makes any semantic calculation extremely hard. It is also different from the existing semantics of MSC, in which a deadlock does not have effects at earlier points in time. In one term, this objection could be called 'backward causality': What happens now is dependent on what happens or can happen at a later time.

The second proposal was based on intuition. This view, which is more whole-system-based, is closer to the intuition of the users. Thus, having a different semantics will lead to MSCs that mean something different from what they are thought to mean.

Apart from the lack of intuition (for the first proposal), and the complicated semantics (for the second proposal), there are also some more profound problems, certainly with the second proposal. For some MSCs this proposal provides no semantics at all, because guards have to be evaluated by instances that never pass them. An example of these problems is in the MSC in Figure 6.14.

If we first look at the semantics of this MSC without guards, we see that one possible trace is to start with $x := 1$, then send and receive $m$, then go through the $a$-loop indefinitely. Is this trace still a valid trace in the situation where guards are present? Under the second proposal we do not know: The guard should be evaluated when the second instance is at the relevant point in time – but in this trace the second

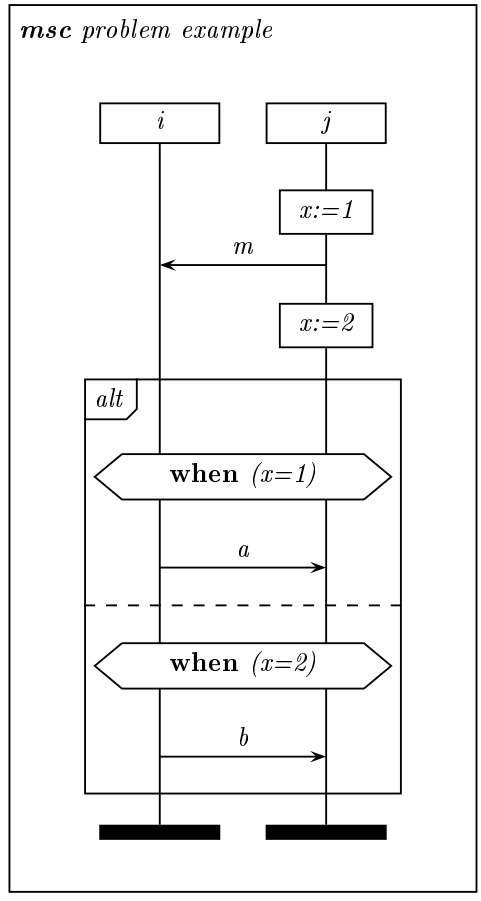



Figure 6.13: Problems with data in guards

instance never does.

Both proposed semantics also lead to strange behaviours for some MSCs. The first semantics gives for instance some unexpected ordering requirements for the MSC in Figure 6.15. Under the first semantics, instance $i$ must defer sending message $a$ until instance $j$ has done the action $x := 2$. This thus leads to a synchronisation, or at least to an extra ordering requirement, which is not clear in the syntax of the language, and counter-intuitive.

For the second proposal, causality can go strange ways, for example in Figure 6.16. In this MSC, instance $i$ can send message $a$ if and only if $j$ changes the value of $x$ to 1. If $i$ sends out a message before $j$ sets the value of $x$, $i$ decides what choice $j$ is going to make. Again, this is a connection (causal rather than ordering in this case) that is not obvious from the syntax of the MSC, as well as unwanted.

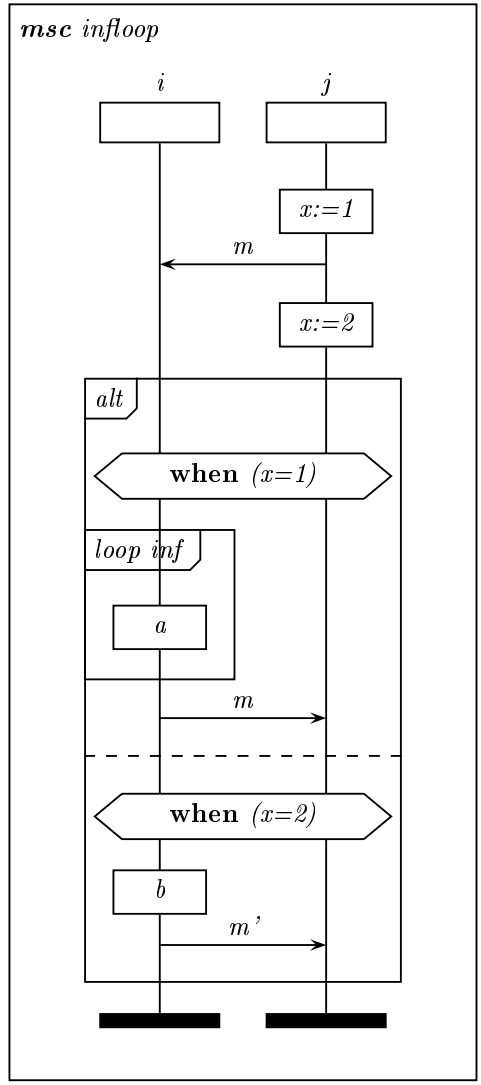



Figure 6.14: An infinite loop causing semantic unclarity

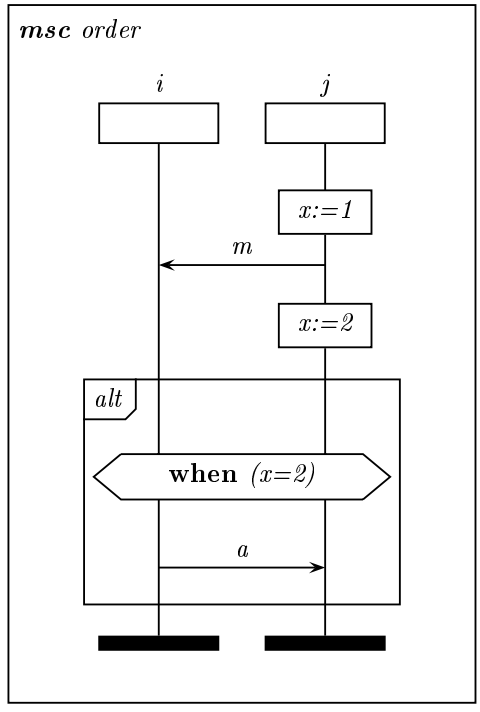



Figure 6.15: Unexpected ordering requirements

### 6.5.3 Static Requirements as a Solution

Neither option seemed very attractive. Instead, one would like a semantics that is intuitive from both the usage and the semantic point of view, rather than just one. Instead of choosing one of the alternatives, it was preferred to restrict the MSC language. Only those MSCs where both interpretations provided the same result would be allowed.

To see which MSCs these are, we have to go back to where we introduced the two proposed semantics. There it was said that their difference was in the time and place where the guard is evaluated. The first semantics evaluates the guard when the first instance passes the guard (by performing some action coming after the guard), the second semantics when the owning instance of the variables goes through it.

The two semantics will provide the same result if the value of the guard does not change, or if they both evaluate it at the same time and place. The evaluation of a guard does not change if it does not contain any dynamic variables. Thus, any such guard is unproblematic. If there are dynamic variables involved, the evaluation may be different at different points in time. Thus, in this case both semantics should evaluate the guard at the same time. This is only the case if the owning instance of the variables is the first to do an action after the guard. An instance which is able to do the first action after a guard is called a *ready* instance for that guard, and thus

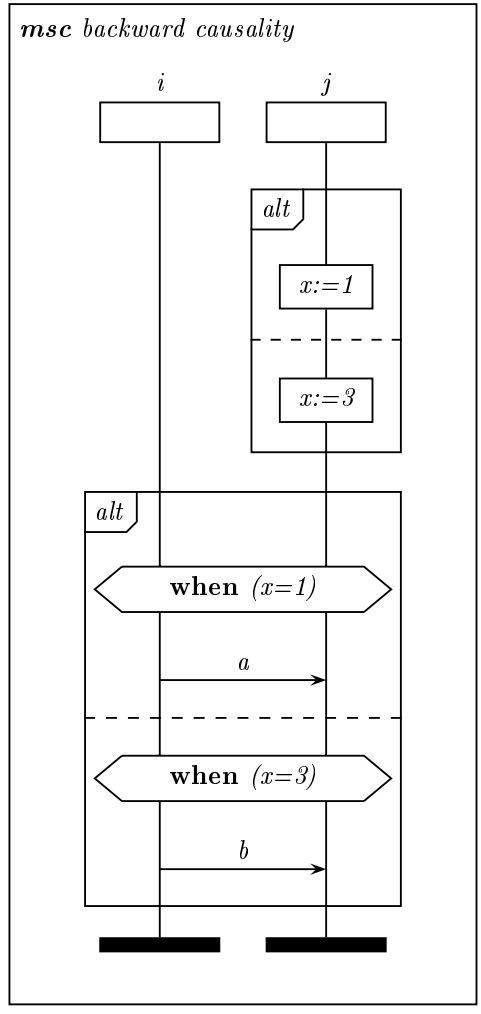



Figure 6.16: MSC with backward causality

the following static requirement was added to the language:

> If a guard contains a data expression, then this expression must be of type Boolean. If this expression furthermore contains dynamic variables, it may only cover a single instance, which thus must be the only ready instance of the scope.[1]

### 6.5.4 Non-Data Guards

Similar problems, where the truth value of a guard becomes indeterminate, can also occur for guards without data, which get their truth value from defining guards. The problem occurs in MSCs like the one in Figure 6.17.

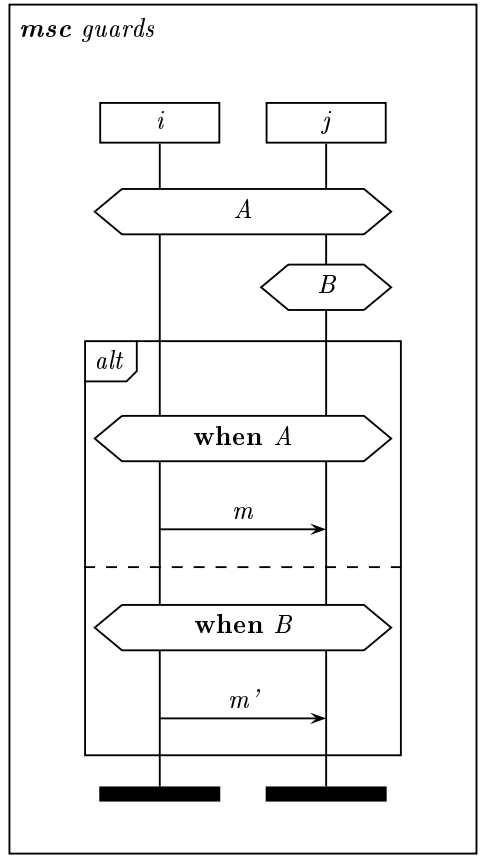



Figure 6.17: Why not all defining conditions affect a guard

[1] The text in the published version [IT00] is slightly different. The quoted text is taken from an official correction to the published version. In this correction it is also made clear that the restriction that all ready instances must be covered by the guard holds for all guards, not just for those with data.

If one simply would have one state for each instance, instance $i$ would enter the alt-expression in state $A$, while instance $j$ would do so in state $B$, so they disagree on which of the alternatives has to be chosen. This has been solved by allowing *only* conditions on exactly the same instances to be referred to by the guard. The setting of $B$ is thus not of importance, and both instances here agree that the first alternative has to be chosen.

## 6.6 The Interface

The current MSC standard [IT00] defines the interface between a data language and MSC. It consists of a number of functions which define the information that MSC needs from the data language to decide the semantics of the combined language. We will here give an overview of this interface, for two reasons. In the first place, the interface definition in [IT00] is often not very clear, which we hope will be done better in this chapter. In the second place, we have made some changes which increased the clarity while not removing any applicability. In Section 6.7, our new version of the interface will be used. Finally, while creating the semantics, we found that there was one function lacking in the interface. This one has been added in this chapter. All the non-trivial differences between the Interface defined here and the one from [IT00] can be found in 6.6.4.

The interface can be divided into two types of functions: the static functions, which are used to check whether a data expression is legal, and the dynamic functions, which are used to define the actual semantic meaning of a data expression.

### 6.6.1 Example Language

In this chapter, we will use a simplified data language to explain the various parts of the interface. Note that this is just a simple language used as an example, this nor any other data language is specifically adapted for use in MSCs.

Our language will consist of:

- The data types of Naturals and Booleans,
- Variables $x$, $y$, $z$ and $x_1, x_2, \ldots$,
- Constants $1, 2, 3, \ldots$, true and false, with the obvious meanings,
- Operators $+$, $\cdot$, $\wedge$, $\vee$, $\neg$ and $=$.

Furthermore, we will take the liberty of adding brackets where necessary, and removing them where possible.

### 6.6.2 Static Functions

Before going into each of the static functions in a bit more of detail, we will first give an overview in the table below.

In the rest of this chapter, we will use the following:

- $D \in \mathcal{D}$ is a data definition (that is, the data information part of an MSC document),
- Var is the set of all possible variables,
- $V \subset$ Var is the set of all actual variables (not including wildcards),
- $W \subset$ Var is the set of all actual wildcards,
- $t \in T$ is a data type,
- $\omega \subset \Omega$ is the set of all pairs of actual variables and wildcards with their data types (thus, $\Omega \subset \text{Var} \times T$, and $(x, t) \in \omega$ for exactly one $t$ if $x \in V \cup W$ and for no $t$ if $x \notin V \cup W$.
- $\Sigma$ is the set of all possible strings of text that can occur in an MSC.

A wildcard in MSC is used to denote a random value. The difference with a variable is, that a wildcard can have a different value every time it is used, even within one expression. For example, if _ is a wildcard with type natural, then _ is any natural, $2 \cdot$ _ any even natural, and _ + _ any natural.

| **Function** | **Type** | **Usage** |
|---|---|---|
| variable-check $c1(\sigma)$ | $\Sigma \to$ Bool | Checks whether $\sigma$ is a variable. |
| data-definition-check $c2(\sigma)$ | $\Sigma \to$ Bool | Checks whether $\sigma$ is a data definition. |
| typeref-check $c3_D(\sigma)$ | $\mathcal{D} \to \Sigma \to$ Bool | Checks whether $\sigma$ is a type reference. |
| expression-check $c4_{D,\omega}(\sigma, t)$ | $(\mathcal{D} \times \mathcal{P}(Omega)) \to (\Sigma \times T) \to$ Bool | Checks whether $\sigma$ is an expression of a given type. |
| variable-equivalence EqVar$(\sigma_1, \sigma_2)$ | $\Sigma \times \Sigma \to$ Bool | Checks whether two variables are the same |

To get a better idea of what these functions do, we will show what they look like for our example language. Note that our language is slightly overspecified: In MSC, the types of variables, and which variables and wildcards actually can exist, is defined in the MSC rather than as an intrinsic part of the data language. For the current chapter, we will change our language definition by only specifying that there are variables or wildcards of the forms $p$, $q$, $r$, $x$, $y$, $z$, _, $x_1, \ldots$ and $p_1, \ldots$, and that the division of these identifiers between variables and wildcards, as well as their types, are to be defined in the MSC document header.

- The variable-check predicate $c1$, which defines whether a string $\sigma$ is parsed correctly as a variable is true for any string of the abovementioned syntactic forms, false otherwise.
- Our language is relatively simple, and when used in combination with MSC will not require any auxiliary data definition. Because of this, the data-definition-check predicate $c2$ will be false for any string. In more complicated languages

it could be used for the definition of non-standard data types, extra operators, and other such additions.

- Only two strings will pass our type-reference-check c3: the existing types 'Natural' and 'Boolean'. Any other string cannot be a type reference. Note however that in more general cases, $c3$ depends on the data definition – the data definition could include the introduction of new types (or even the removal of existing ones).

- Next we get to $c4$. $c4_{D,\omega}(t, \sigma)$ is true if and only if $\sigma$ is a correct expression of type $t$. For our example language this function can be defined inductively as follows:

  - $c4_{D,\omega}(c, t)$ with $c$ a constant is true if and only if $t \equiv$ Natural and $c$ is a natural constant, or $t \equiv$ Boolean and $c$ is a boolean constant.
  - $c4_{D,\omega}(x, t)$, with $x$ a variable or wildcard, is true if and only if $(x, t) \in \omega$.
  - $c4_{D,\omega}(\sigma + \tau, t)$ and $c4_{D,\omega}(\sigma \cdot \tau, t)$ are true if and only if $t \equiv$ Natural and $c4_{D,\omega}(\sigma, \text{Natural}) = c4_{D,\omega}(\text{Natural}, \tau) = \text{True}$
  - $c4_{D,\omega}(\sigma \wedge \tau, t)$ and $c4_{D,\omega}(\sigma \vee \tau, t)$ are true if and only if $t \equiv$ Boolean and $c4_{D,\omega}(\sigma, \text{Boolean}) = c4_{D,\omega}(\text{Boolean}, \tau) = \text{True}$
  - $c4_{D,\omega}(\neg\sigma, t)$ is true if and only if $t \equiv$ Boolean and $c4_{D,\omega}(\sigma, \text{Boolean}) = \text{True}$
  - $c4_{D,\omega}(\sigma = \tau, t)$ is true if and only if $t \equiv$ Boolean and $c4_{D,\omega}(\sigma, \text{Natural}) = c4_{D,\omega}(\tau, \text{Natural}) = \text{True}$

  Note that normally in a language like our example language, one would have used brackets to disambiguate the various expressions; for reasons of simplicity, these have been omitted wherever this did not lead to ambiguity.

- Finally, EqVar for our language is such that that two variable names are equal if and only if they are identical as strings. This will be true for many languages, but not for all. EqVar might for example be used to specify that a language is case-sensitive.

### 6.6.3 Dynamic Functions

Four functions are required for the dynamic semantics of MSC with data. Three of these are introduced to be able to manipulate strings on a syntactic level, and are in the first place (but not only) meant for dealing with wildcards correctly. The last is a semantic evaluation function. $\sigma$ will from now on stand for a string that is supposed to be an expression. The set of such strings will be denoted $\Sigma$. Although not defined in the standard, working out the semantics we found that a fifth function is necessary.

Below, $x \in$ Var is a variable, $U$ is the semantic domain of the data language (defined below), $E : V \to U$ is a function that gives the current value of each variable, and $\mathcal{E}$ the set of all such functions.

| Function | Type | Usage |
|---|---|---|
| Variable counter $\mathrm{Vars}_D(\sigma)$ | $\mathcal{D} \to \Sigma \to \mathcal{P}(\mathrm{Var})$ | Gives the variables in a string $\sigma$ |
| Variable replacement $\mathrm{Rep}_D(\sigma, x, x')$ | $\mathcal{D} \to \Sigma \times \mathrm{Var} \times \mathrm{Var} \to \Sigma$ | Replaces a single occurrence of a variable in a string |
| New Variable $\mathrm{NewVar}_D(V)$ | $\mathcal{D} \to \mathcal{P}(\mathrm{Var}) \to \mathrm{Var}$ | Provides a fresh variable |
| Semantic Domain $U$ | (not applicable) | The set of all semantic objects |
| Semantic Evaluation $\mathrm{Eval}_{D,E}(\sigma)$ | $\mathcal{D} \times \mathcal{E} \to \Sigma \to U$ | Gives the semantic meaning of an expression |
| Semantic Range $\mathrm{Sem}_D(t)$ | $\mathcal{D} \to \mathcal{T} \to \mathcal{P}(U)$ | Gives the semantic range of a type |

We will now look at the various functions just defined, to give an explanation of what they are supposed to do, and an example of how they would or could look like in our example language.

- The function Vars checks which variables (or wildcards) appear in a given expression. For example, $\mathrm{Vars}_D((x+y)\cdot x) = \{x, y\}$.

  For our example language, Vars can easily be defined inductively as follows:

  - For a constant $c$, $\mathrm{Vars}_D(c) = \emptyset$.
  - For a variable $x$, $\mathrm{Vars}_D(x) = \{x\}$.
  - $\mathrm{Vars}_D(\sigma + \tau) = \mathrm{Vars}_D(\sigma \cdot \tau) = \mathrm{Vars}_D(\sigma \wedge \tau) = \mathrm{Vars}_D(\sigma \vee \tau) = \mathrm{Vars}_D(\sigma = \tau) = \mathrm{Vars}_D(\sigma) \cup \mathrm{Vars}_D(\tau)$
  - $\mathrm{Vars}_D(\neg\sigma) = \mathrm{Vars}_D(\sigma)$

  A definition of this type is possible for most languages, but in some cases more complicated functions are necessary – for example because some variables are 'hidden' by an abbreviation, or because something can be a variable if defined as such, but have another meaning in other cases.

- The next function, Rep, substitutes a single occurrence of a variable by a given other variable. For example, $\mathrm{Rep}_D((x+y)\cdot x, x, z)$ gives the result of replacing one $x$ in $(x+y)\cdot x$ by $z$, which, depending on the exact definition of Rep, might either be $(z+y)\cdot x$ or $(x+y)\cdot z$. Note that $(z+y)\cdot z$ would not be an allowable outcome for this substitution, because then two occurrences of $x$ would have been replaced.

  For our example language, we will choose to have Rep always replace the first occurence of a variable, which leads to:

  - $\mathrm{Rep}_D(c, x, y)$ for $c$ a constant is equal to $c$.
  - $\mathrm{Rep}_D(z, x, y)$ for $z$ a variable is equal to $y$ if $x \equiv z$, and $z$ otherwise.

  - $\mathrm{Rep}_D(\sigma + \tau, x, y)$ equals $\mathrm{Rep}_D(\sigma, x, y) + \tau$ if $x \in \mathrm{Vars}_D(\sigma)$, and $\sigma + \mathrm{Rep}_D(\tau, x, y)$ otherwise.
  - Similar rules hold for the other operators $\cdot$, $\wedge$, $\vee$, $\neg$ and $=$.

  The Rep function is used for two purposes: In the first place to handle wildcards properly, and in the second place to disambiguate a situation where two static variables by the same name have been defined.

- $\mathrm{NewVar}_D(V)$ basically provides a new variable from Var, not yet in $V$. Of course this is not always possible (just take $V = \mathrm{Var}$), but in actual usage $V$ will always have a finite number of variables, so NewVar needs only to be defined in that case. If Var itself is infinite, this is enough to ensure the possible well-definedness of NewVar.

  For our language, we can define $\mathrm{NewVar}_D(V)$ for finite sets $V$ to be $x_n$, with $n$ the smallest $n$ such that $x_n \notin V$.

- Finally, there is the actual semantic interface. It consists of some semantic domain $U$, and the function Eval, which gives the semantic meaning of an expression. This meaning depends on the current value of the variables, which is encoded in the function $E$. As a static semantic restriction one could specify that only those variables that actually occur in $\sigma$ (as defined through the function Vars) are allowed to influence the result. More formally, we should have $\mathrm{Eval}_{D,E}(\sigma) = \mathrm{Eval}_{D,E'}(\sigma)$ if $E(x) = E'(x)$ for each $x \in \mathrm{Vars}_D(\sigma)$.

  For our example language we have:

  - $\mathrm{Eval}_{D,E}(c)$ for $c$ a constant equals the 'natural' meaning of $c$.
  - $\mathrm{Eval}_{D,E}(x)$ for $x$ a variable equals $V(x)$.
  - $\mathrm{Eval}_{D,E}(\sigma+\tau)$ equals the sum of $\mathrm{Eval}_{D,E}(\sigma)$ and $\mathrm{Eval}_{D,E}(\tau)$, and similarly for the other operators.

  The Semantic Range function $\mathrm{Sem}_D(t)$ is necessary for the correct handling of wildcards. It gives the complete semantic range of a data type, that is, all values that can be taken by variables of a given type. As such, it specifies the values that a wildcard of that type can have. For our example language, this function is defined by $\mathrm{Sem}_D(\mathrm{Natural}) = \mathbb{N}$, $\mathrm{Sem}_D(\mathrm{Boolean}) = \{\mathrm{true}, \mathrm{false}\}$.

### 6.6.4 Changes in the Interface

The interface as it has been presented here, differs somewhat from the one in [IT00]. The differences, with justifications, will be mentioned below.

- Compared to [IT00], we have removed some static functions. The reason for that, is that some of the static functions defined above are the combination of more than one function from [IT00]. $c2$ and $c3$ both combine two functions from [IT00], where a well-formedness predicate is checked first to see whether a string *could* be a data definition or type reference, after which a type-check

predicate checks whether it actually is. These functionalities have been combined into a single function in our version. The function $c4$ even combines three functions, because [IT00] contains both a general and a type-dependent type-check. Only the latter has been kept – the former can be derived by stating that something is an expression if and only if for some type it is an expression of that type.

- In [IT00], Vars also counts the number of occurrences of each variable. For the semantics this information is actually unnecessary. The semantics does some extra work that would otherwise be unnecessary because of the removal of this information, but in general nevertheless looks better without it.
- In [IT00], Rep also contains among its arguments a number specifying which occurrence of a variable has to be changed. Because this information is not of relevance for the semantics, it has been removed.
- The semantic domain $U$ has been mentioned explicitly, whereas in [IT00] it is only defined implicitly, by the definition of Eval.
- The function Sem, which we found to be necessary for a correct semantic handling of wildcards, has been added.

## 6.7 Semantics for Data in MSC

In this section, we will give an indication how data and guards can be semantically added to MSC. To do so, we take the semantics for MSC'96 as found in the thesis of Reniers [Ren99] (see section 4.4 for a short introduction) as a starting point, and look how data can be added and where it might cause problems.

### 6.7.1 The State Variable $\Psi$

If we add data to the language, this is most easily done by adding a kind of 'state variable' $\Psi$, which keeps track of the relevant data information. In particular, the following information is kept:

- The data definition information $D$
- A set $V$ of all variables that have been defined
- A set $W$ of all wildcards that have been defined
- The function $d : V \to$ Bool which specifies whether a variable is dynamic
- The function $o : V \to I$ ($I$ being the set of instances), giving the owning instance for each dynamic variable
- The function $t : V \cup W \to T$ ($T$ being the set of types that are allowed), giving the type of each variable and wildcard

- The functions $\phi_i : V \to U \cup \{\perp\}$, giving the local value on the instance $i$ of a variable. The special value $\perp$ ($\perp \notin U$) is used if, as far as is 'known' to a given instance, no value has been given to a variable yet
- The function $s : I \times \mathcal{P}(I) \to \mathcal{P}(\Sigma) \cup \{\perp\}$, remembering the last defining condition on a given set of instances that a given instance has met. A set of strings $\in \mathcal{P}(\Sigma)$ rather than a single string is used because a number of possible states can be defined by a single condition.

The semantics of MSC, which is currently defined on process algebra expressions, must now be defined on the combination of these process algebra expressions and these state variables.

At the start of an MSC document, $D$, $V$, $W$, $o$ and $t$ are initialised in an obvious way in the document header. Furthermore, at this point $d(x) = \text{true}$ for all $x \in V$, $\phi_i(x) = \perp$ for all $x \in V, i \in I$, and $s(i, J) = \perp$ for each instance $i$ and set of instances $J$.

### 6.7.2 Local Actions

The basic use for data is simple. Every time an expression is encountered, it should be replaced by its meaning. We will formalise this, first for expressions $\sigma$ which do not contain wildcards. The more complicated subjects, such as wildcards and messages, will be dealt with later.

A restriction to such a usage is that each variable $x$ has to be defined on the instance on which the event takes place of which the expression is a part, that is $\phi_i(x) \neq \perp$. This will have to be checked dynamically, although it functions in the same way as a static restriction: MSCs for which this condition is not true are considered illegal. If the expression is not connected to a specific event, it is not allowed to contain any dynamic variables.

For each action $a$ that contains an expression without wildcards, if under the existing (MSC'96) semantics the step $x \xrightarrow{a} y$ (here $x$ and $y$ are process algebra expressions, and $a$ is the process algebra event corresponding to the action $a$) is possible, then under the semantics with data the step $(x, \Psi) \xrightarrow{\text{Eval}_{D,\phi_i}(a)} (y, \Psi)$ is possible, where $i$ is the instance on which $a$ takes place, and $\text{Eval}_{D,\phi_i}(a)$ is found by replacing each expression $\sigma$ in $a$ by $\text{Eval}_{D,\phi_i}(\sigma)$.

These issues get more complicated when wildcards are used. If an expression does contain one or more wildcards, it should be evaluated for any possible value of these wildcards. Furthermore, if the same wildcard is used several times, it should be possible to instantiate it with different values each time. For the latter purpose, we first make each occurrence of a wildcard unique, in the following way:

Let $\sigma$ be an expression, and let $T'$ be some (finite) set of variable-type pairs (we will see later what the function of the latter is). We define $\text{wf}(\sigma, T')$ inductively as follows (the definition is not complete, because it is not specified which wildcard has to be chosen at each step; however, the result is valid whichever choice is made).

- If $\sigma$ does not contain any wildcards (that is, $Vars_D(\sigma) \cup W = \emptyset$), then $\text{wf}(\sigma, T') = (\sigma, T')$

- Otherwise, choose any wildcard $x$ in $\sigma$ (that is, $x \in Vars_D(\sigma) \cap W$). Then $\mathrm{wf}(\sigma, T') = \mathrm{wf}(\sigma', T' \cup \{z, t\})$, where:
  - $z = \mathrm{NewVar}_D(V \cup W \cup W')$, where $W'$ consists of all first elements of pairs in $T'$
  - $t = t(x)$ (the type of $x$, and thus of $z$)
  - $\sigma' = \mathrm{Rep}_D(\sigma, x, z)$

Thus, $\mathrm{wf}(\sigma, \emptyset)$ consists of a rewriting of $\sigma$ into a form where every wildcard occurs only once, and a list of extra wildcards and their types that have to be created to do so. We will call this rewriting and this list $\mathrm{wf}(\sigma)$ and $T'$, respectively, while we define $W'$ to be the set of first elements of pairs in $T'$ and $\mathrm{Sem}(t(x))$ for $x \in W'$ to be the type $t$ such that $(x, t) \in T'$.

Provided that Rep and NewVar both work in the way that they are supposed to work (that is, Rep replaces exactly one occurence of a variable, and NewVar provides a new variable), the definition above will give a result after a finite number of steps, although the result may depend on choices that have been made.

We define a 'choice function' to be a function $c : W \cup W' \to U$ such that $c(x) \in t(x)$ for each wildcard $x$. It thus gives an arbitrary allowed value for each wildcard. With this information, we can finally add wildcards to the description of data in simple expression:

For each action $a$ that contains an expression, if under the existing MSC'96 semantics the step $x \xrightarrow{a} y$ is possible, then under the semantics with data the step $(x, \Psi) \xrightarrow{\mathrm{Eval}_i(a)} (y, \Psi)$ is possible, where $i$ is the instance on which $a$ takes place, and $\mathrm{Eval}_i(a)$ is defined by replacing each expression $\sigma$ in $a$ by $\mathrm{Eval}_{D,\phi_i \cup c}(\mathrm{wf}(\sigma, V \cup W))$ for any arbitrary choice function $c$ on the wildcards in $\mathrm{wf}(\sigma, V \cup W)$.

To simplify notation, we will define the predicate $c_i(\sigma, u)$ for an expression $\sigma$ and a semantic object $u \in U$ to be true if and only if there is some choice function $c$ such that $\mathrm{Eval}_{D,\phi_i \cup c}(\mathrm{wf}(\sigma, V \cup W)) = u$. The above then becomes: If $x \xrightarrow{\sigma} y$ is possible under the current semantics, then under the semantics with data $(x, \Psi) \xrightarrow{u} (y, \Psi)$ is possible, provided $c_i(\sigma, u)$ holds.

As a next complicating factor, a local action can contain a binding of the type $x := \sigma$ (with $x$ a variable and $\sigma$ an expression). In this case the following static requirements must be met:

- $\sigma$ and $x$ must be of the same type, that is $c4_{D,\overline{T}}(t(x), \sigma) = \mathrm{true}$.
- $x$ is a dynamic variable that is owned by the instance to which the local action is connected, that is, $d(x) = \mathrm{true}$ and $o(x) = i$

In this case not only the expression must be replaced, as done above (one can even imagine that one does not want to replace the expression, but that is a choice that I do not want to go into at the moment), but also the value of $x$ has to be changed. Thus, the rule now becomes that if under the current semantics $x \xrightarrow{\mathrm{action}(x:=\sigma)} y$, then under the semantics with data we have $(x, \Psi) \xrightarrow{\mathrm{action}(x:=u)} (y, \Psi')$, provided $c_i(\sigma, u)$ holds, where $\Psi'$ equals $\Psi$, except that in $\Psi'$, $\phi_i(x) = u$.

### 6.7.3 Simple Messages

Messages behave different from other actions in two ways:

1. The interpretation of an input event depends on the corresponding output event, rather than on the current value of the variables on the receiving instance itself

2. The message can have the effect of changing the value of a variable, or of communicating the value of a variable to another instance.

First, we look at a simple message – a message which contains an expression, but no bindings, and does not go through any gates. A message consists of two parts, the sending and the receipt of the message. We cannot handle them as two independent events for the reasons mentioned above. Instead, we will have to remember the information of the sending event when the receiving event happens.

The information that is needed, is the value of all variables that are in the expression. A logical place to do so, is in the ordering requirement that already exists to ensure that messages are sent before they are received, that is in the requirements part $S$ of $\circ^S$ and $\|^S$ (see chapter 4.4).

To extend the semantics to also be able to work with data, the information in these requirements should also contain the value of the variables. To handle wildcards in an easy way, we also need to put the actual value of the expression as a whole in here. We need a set of all values of variables in the expression of the contents of the message, and one such set for each occurrence of the sending of the message. Furthermore, these lists have to be considered in FIFO (First-In-First-Out) order. Thus, the numbers $n$ in $\overset{n}{\mapsto}$ are replaced by lists of sets $E = (e, x_1 = e_1, x_2 = e_2, \ldots, x_n = e_n)$, where $e$ is the value of the expression, $x_1, \ldots, x_n$ are the variables in the expression and $e_1, \ldots, e_n$ their valuations.

The new $enabled(a, S)$ predicate need not be more complicated than the old one. The rule that the number $n$ is larger than 0 for all orderings with $a$ on the right side of the ordering, is replaced by the rule that the list of lists $E$ is non-empty.

The new update function $upd(a, S)$ does get more complicated. For a message sending event $out(i, j, m)$, apart from the ordering $out(i, j, m) \mapsto in(i, j, m)$, we add the valuation of the message $m$ and a list of the variables in $m$ together with the value of $\phi_i(x)$ for each of these variables. For generalised orderings, the list of variables and values is necessarily empty.

When receiving a message, we need to change the receive action that is done, using the valuations as defined by the corresponding send action. If $x \overset{a}{\rightarrow} x'$ is allowed in the existing semantics for some receipt event $a = in(i, j, m)$, there will in this case be exactly one non-empty ordering requirement $b \overset{E}{\mapsto} a$ (to be exactly, this will be the case for $b = out(i, j, m)$). From this ordering requirement we get a set of variable valuations $E(x)$, as well as a value of the expression itself $E(m)$.

The step which now is allowed in the semantics with data, will be: $(x, \Psi) \overset{a'}{\rightarrow} (x, \Psi')$, where $a' = in(i, j, E(m))$ and $\Psi'$ has $\phi_j(x) = E(x)$ for all variables that occur in $m$ (which must necessarily be the same as the variables that occur in $E$) which have $c(x) = \text{true}$ and $o(x) \neq j$, and is equal to $\Psi$ elsewhere.

### 6.7.4 Bindings and Gates in Messages

Apart from expressions, a message $m$ can also contain bindings as one of its parameters. We could thus have a message $m(x+1, y := 7)$, which would mean that the value $x+1$ is sent with the message, and $y$ is given the value 7. The number of such parameters is indefinite.

If the variable that is given a value is a wildcard, everything works just like above. If it is a real variable, then this variable should be a dynamic variable, owned by the receiving instance. In this case, the new value for the variable will be set to (the evaluation during the send event of) the expression. Note that this is the only way that a message can change the value of an owned variable.

If a message is sent through a gate, the parameters given by the sending event and those given by the receiving event are not the same. Rather, the sending event is parameterised with expressions, while the receiving one is parameterised with variables (or wildcards). The semantics of this should be that the variables given as parameters on the receiving side should be given the values given as parameters on the sending side.

In the current semantics, there is already a function to couple the corresponding events on two sides of the gate. This function, which is described on page 141 of [Ren99], at the moment couples the events $out(i, G, \_, m)$ (which means, sending message $m$ from instance $i$ through gate $G$ to an unknown place ($\_$)) and $in(\_, G, j, m)$ to create the events $out(i, G, j, m)$ and $in(i, G, j, m)$. This function can easily be extended to also combine the data information in a correct way – that is, we have to combine $out(i, G, \_, m(e_1, e_2, \ldots, e_n))$ and $in(\_, G, j, x_1, x_2, \ldots, x_n)$ to $out(i, G, j, m(x_1 := e_1, x_2 := e_2, \ldots, x_n := e_n))$ and $in(i, G, j, m(x_1 := e_1, x_2 := e_2, \ldots, x_n := e_n))$. Apart from this, the semantics are exactly as described above. There is still a choice here whether any wildcards are actually put in the binding, or that they are regarded as signifying that expressions rather than bindings are to be added. That is, whether $out(i, G, \_, m(7, x+1))$ and $in(\_, G, j, m(y, \_))$ are combined to $out/in(i, G, j, m(y := 7, \_ := x+1))$ or to $out/in(i, G, j, m(y := 7, x+1))$. This does however not make an essential difference in the semantics.

### 6.7.5 Static Data

The next thing that has to be added to the semantics is the issue of static (parametric) data. When an MSC has a parameter, say $x$, it can only be called with an actual value for that parameter. All occurrences of the variable $x$ are then replaced by its actual value.

The best way to deal with this, seems to be to create a new variable (using NewVar) $x'$, add this to the set of variables $V$, and store its value. This new variable will have $d(x') =$ false, and $\phi_i$ equal to the defined value for each instance $i$. All events in the MSC are then labeled with an extra statement $x := x'$, which has the effect of replacing each occurrence of $x$ by an occurrence of $x'$ (by repeated application of Rep). This renaming is also applied in all MSCs that are called by the MSC itself, except if this other MSC also has $x$ as one of its parameters.

The reason that we choose to include an extra variable $x'$ rather than using the existing variable $x$, is that $x$ may be defined at more than one place. In such a case,

we have to have a mechanism to decide which value of $x$ is to be used, which is done through this new variable $x'$.

### 6.7.6 Guards

Passing a guard is preferably not considered an action. Rather, guards are conditions on the permissibility of actions that happen later. Luckily, in the semantics of MSC there exists a mechanism to add such a condition in a natural way, namely the permission relation $\cdots\rightarrow$.

We now turn a guard into a quasi-event, that is, it looks like an event, but it cannot be actually executed. More precisely, we add one such quasi-event for each instance covered by the guard. We denote these quasi-events by $guard(i, e)$, where $i$ is the instance on which (this part of) the guard is defined, and $e$ is the data expression connected to it (which of course as a static semantic requirement must be of the type Boolean). We will look at non-data guards later.

For guards, the normal permission rule holds as well:

$$\frac{l(a) \neq i}{(guard(i,e), \Psi) \stackrel{a}{\cdots\rightarrow} (guard(i,e), \Psi)}$$

But there is an extra rule here: Something happening on the same instance, after the guard, may also be executed – but only if the guard evaluates to true. Furthermore, once we have passed the guard this way, it will not hinder us later – at least not on this instance. That is:

$$\frac{l(a) = i, \mathrm{Eval}_i(e) = \mathrm{true}}{(guard(i,e), \Psi) \stackrel{a}{\cdots\rightarrow} (\epsilon, \Psi)}$$

If we look at non-data guards, the guards themselves work very similar to what is mentioned above. This time there will be three parameters: The instance, the set of instances on which the guard is defined, and the texts of the guard (a guard can have more than one string, it can then be passed if the system is (on the given instances) in any of the states defined by the guard). The same holds for defining conditions (if there is more than one string on a defining condition, the system can be in any of the states defined). We will denote them by $guard(i, I, \Sigma)$ and $cond(i, I, \Sigma)$, respectively, with $i$ the relevant instance and $I$ the total set of instances on which the guard or condition is defined. The guards work just like above, except that the check now is that there is a state in which the system can be which is allowed by the guard, rather than the old $\mathrm{Eval}_i(e) = \mathrm{true}$, and that passing a guard can restrict the number of states a system is in: If the system (for a given instance and set of instances) is in a state $\{A, B, C\}$, and passes a guard **when** $A, B, D$, the system state changes to $\{A, B\}$ – it cannot any more be in state $C$.

$$\frac{l(a) \neq i}{(guard(i,I,\Sigma), \Psi) \stackrel{a}{\cdots\rightarrow} (guard(i,I,\Sigma), \Psi)}$$

$$\frac{l(a) = i, \Sigma \cap s(i,I) \neq \emptyset}{(guard(i,I,\Sigma), \Psi) \stackrel{a}{\cdots\!\rightarrow} (\epsilon, \Psi')}$$

Here $\Psi'$ is equal to $\Psi$ except that the value of $s(i,I)$ is changed to that of $s(i,I) \cap \Sigma$.
A defining condition can always be passed, but if it is passed, it changes the state:

$$\frac{l(a) \neq i}{(cond(i,I,\Sigma), \Psi) \stackrel{a}{\cdots\!\rightarrow} (cond(i,I,\Sigma), \Psi)}$$

$$\frac{l(a) = i}{(cond(i,I,\Sigma), \Psi) \stackrel{a}{\cdots\!\rightarrow} (\epsilon, \Psi')}$$

Here $\Psi'$ is equal to $\Psi$ except that the value of $s(i,I)$ is changed to $\Sigma$.
A few special points have to be noted.

In the first place, under this semantics it can happen that the data state $\Psi$ is not uniquely defined any more. We could have an MSC like the one in Figure 6.18, where, after action $a$ has been done, we do not know whether we are in state $A$ or state $B$. We are however not in the compound state, since further actions can make it clear where we are without any guards.

The solution for this is to 'lift' the delayed choice operator ($\mp$) through the data part, that is, rather than just allowing pairs $(x, \Psi)$ with $x$ a process algebra expression and $\Psi$ a data state, we allow expressions of the form $(x_1, \Psi_1) \mp (x_2, \Psi_2) \ldots$. Then such a situation where different paths have the same observable actions, but different data consequences can be solved by changing the current SOS-rule [Ren99]:

$$\frac{x \stackrel{a}{\rightarrow} x', y \stackrel{a}{\rightarrow} y'}{x \mp y \stackrel{a}{\rightarrow} x' \mp y'}$$

into

$$\frac{(x, \Psi) \stackrel{a}{\rightarrow} (x', \Psi'), (y, \Psi) \stackrel{a}{\rightarrow} (y', \Psi'')}{(x \mp y, \Psi) \stackrel{a}{\rightarrow} (x', \Psi') \mp (y', \Psi'')}$$

Of course, some other extra rules are necessary as well to define the behaviour of the delayed-choice operator on algebra-data pairs, these are however all easily derived from the existing semantics.

A second point that needs to be noticed, is that under the semantics as defined above, a guard is evaluated when the first action *after* the guard is done, not at some earlier stage. This might have some unexpected consequences for situations involving parallel composition:

In Figure 6.19, the sending of $m$ cannot be done any more after the binding $x := 2$ is taking place. At that time the value of $x$ is not zero any more, and thus the guard evaluates to false. That the guard has been true at some previous time does not matter: in these semantics it is not possible to pass a guard at some time, but then wait before doing any actions. The guard is connected to the action it guards.

Another issue that has to be covered is the termination predicate $\downarrow$. When there are no actions to be done any more, just guards or conditions, successful termination

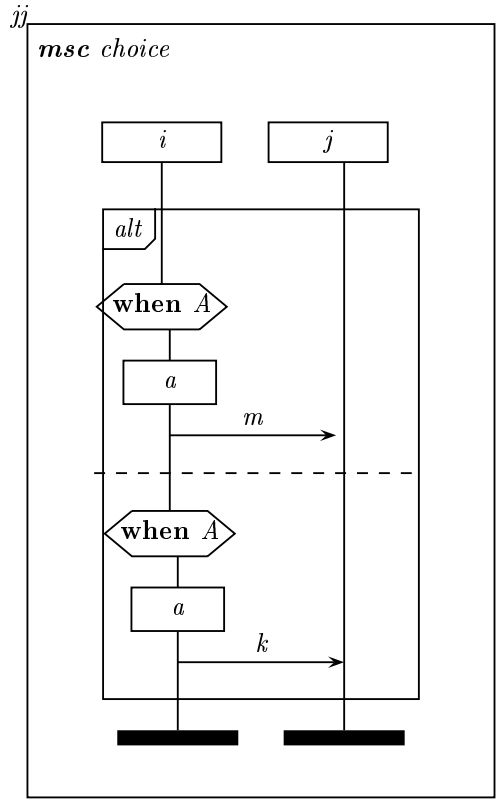



Figure 6.18: The data state is not always uniquely defined

is possible – provided all guards evaluate to true. As a first attempt, one might try to simply give guards the possibility to terminate, but a closer look shows that this will not be sufficient. What if there is a defining condition followed by a non-data guard on the same instances that is still to be passed? The guard should then be evaluated using the state as defined by the defining condition. To enable this, we will have to remember the state variable $\Psi$ also after a termination. Thus, termination will not any more be given by a simple predicate $\downarrow$, but by a predicate $\downarrow_\Psi$, meaning 'termination in data state $\Psi$'.

The existing SOS-rules are to be changed for this predicate. The basic case $((\epsilon, \Psi) \downarrow_\Psi)$ and the rules for delayed choice are easy. More complicated is the case of the merge. For this is good to think of what a guard actually does to $\Psi$: It reduces the number of states in which the system can be. Now, what will happen if the state is reduced in two different ways in two places? Then both reductions will happen. The endstate will be the one with both reductions, which is the same as getting one of the reductions from the end situation of the other reduction.

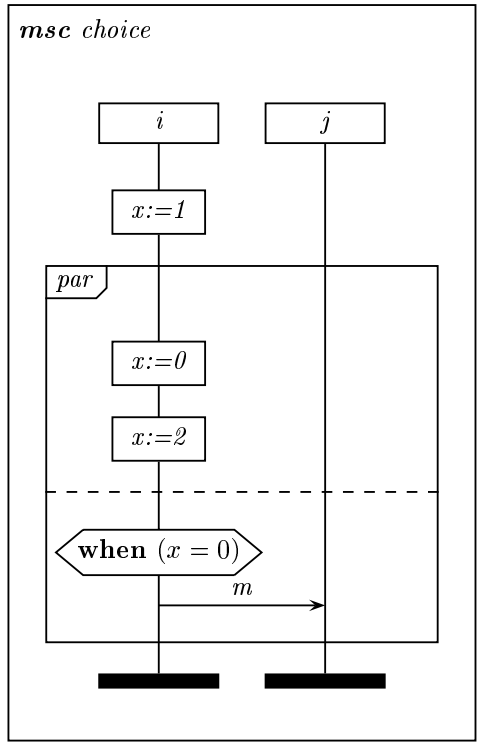



Figure 6.19: Possibly unexpected behaviour of parallel composition

$$\frac{}{(\epsilon, \Psi) \downarrow_{\Psi}}$$

$$\frac{(x, \Psi) \downarrow_{\Psi'}}{(x \mp y, \Psi) \downarrow_{\Psi'}}$$

$$\frac{(y, \Psi) \downarrow_{\Psi'}}{(x \mp y, \Psi) \downarrow_{\Psi'}}$$

$$\frac{(x, \Psi) \downarrow_{\Psi'}, (y, \Psi') \downarrow_{\Psi''}}{(x \parallel y, \Psi) \downarrow_{\Psi''}}$$

$$\frac{(y, \Psi) \downarrow_{\Psi'}, (x, \Psi') \downarrow_{\Psi''}}{(x \parallel y, \Psi) \downarrow_{\Psi'}}$$

$$\frac{(x, \Psi) \downarrow_{\Psi'}, (y, \Psi') \downarrow_{\Psi''}}{(x \circ y, \Psi) \downarrow_{\Psi''}}$$

There are some more rules regarding $\downarrow$ [Ren99], but all are easily translated into rules for $\downarrow_\Psi$.

Termination for conditions and guards can now be defined by:

$$\frac{}{(cond(i, I, \Sigma), \Psi) \downarrow_{\Psi'}}$$

Here $\Psi'$ equals $\Psi$ except that in $\Psi'$, $s(i, I) = \Sigma$.

$$\frac{\Sigma \cap s(i, I) \neq \emptyset}{(guard(i, I, \Sigma), \Psi) \downarrow_{\Psi'}}$$

Here $\Psi'$ equals $\Psi$ except that $s(i, I)$ has been changed to the previous value of $\Sigma \cap s(i, I)$.

$$\frac{\mathrm{Eval}_i(e) = \mathrm{true}}{(guard(i, e), \Psi) \downarrow_\Psi}$$

## 6.8 Conclusions

The addition of data to the MSC language was not an easy task. Much work has been done by a number of people. It was felt early that using a flexible interface would be better than to using a single pre-defined data language. However, there were still many choices to be made regarding the way in which a data language could be combined with MSC. Some of these choices were quite automatic once they had been identified [EFM99], but others have remained open for a large part of the process.

Even larger problems were found with the inclusion of guards in the MSC language. The various standards that are being used in extending the language clashed here – in particular, the intuitive meaning that certain MSCs have could not easily, and in some cases not at all, be translated into a semantics. Thus, there were two methods of interpreting guards, one corresponding with intuition, the other semantically 'clean'. As both had some disadvantages that were regarded decisive, neither was chosen, but rather, the language was restricted through static requirements, so as to only allow those cases where both interpretations resulted in the same semantics.

An overview of what the semantics for data and guards would look like is also given in this chapter. The semantics in this chapter are based on the existing semantics for MSC'96 [MR97b, IT98, Ren99]. By extending this semantics at various places with data concepts or guards, we can create a semantics for the subset of the MSC2000 language consisting of MSC'96 plus data features and MSC2000-style guards and defining conditions. Although the complete semantics is not actually given, the treatment in this chapter should be enough to make the creation of such a complete semantics relatively easy, providing a solution for all major stumbling blocks. Note that to be able to work with this semantics, the need was felt to extend the interface between MSC and the data language, as defined in the standard, with the function Sem. However, it seems likely that a similar function would be necessary for any reasonable semantics of this language.

It would be a good thing to have a semantics for the complete MSC2000 language; however, such a semantics might well be impossible. The current semantics for MSC are such that to see what the next action can be, one only has to look at the current state, not the future. Such a semantics we will call 'executable' – it is possible to 'walk through' the semantics without problems. The basic problem of the 'intuitive' solution regarding guards, was that it would not have this property any more. Rather, it would create a semantics in which a look-ahead is necessary, one would have to look at the future behaviour of the system to see whether a certain action would be possible in the current state. Such a semantics would be complicated from a theoretical point of view and hard or impossible for toolmakers to understand.

One of the other extensions that has been included in MSC2000 is time. Thus, a complete semantics for MSC2000 would also contain this extension. If we look at MSC with time, a look-ahead semantics seems to be the only reasonable solution. For example, the MSC in Figure 6.20. The notation $b$@3 here means '$b$ at time 3'. If in this MSC we would not allow look-ahead, then one possible trace could be to first do $b$ (at time 3), then do $a$ (at time 3 or some later time), and then deadlock. Although it would of course be possible to make a semantics this way, it can in no way be regarded intuitive.

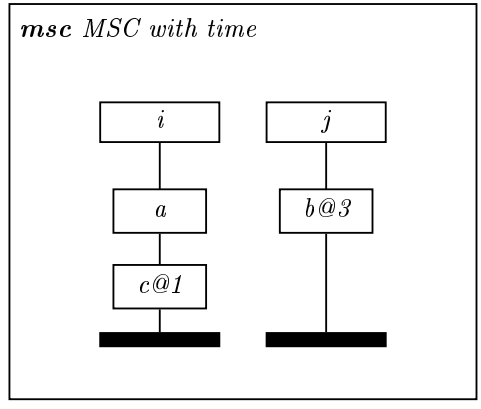



Figure 6.20: An MSC with (absolute) time

On the other hand, apart from disadvantages like the one already mentioned, if we add data (and guards) to MSC, backtracking semantics get even harder: Suppose we have the MSC like in Figure 6.21 (the inline expression here has to be passed zero or more times). Can this MSC, under a backtracking semantics, after setting $x$ equal to 1, start the loop with the action $x := f(x)$? This is possible if, and only if, there is some $n \geq 1$ such that $f^n(x) = 1$. And that is something that, even for quite simple languages, might well be undecidable. The problem is that the semantics ask an infinite amount of information from the data language.

Because this way time clashes with other parts of the language, it seems likely that a complete semantics of MSC2000 will not be developed in the near future (although the title of [JP01] seems to claim it is a semantics for MSC2000, it actually gives

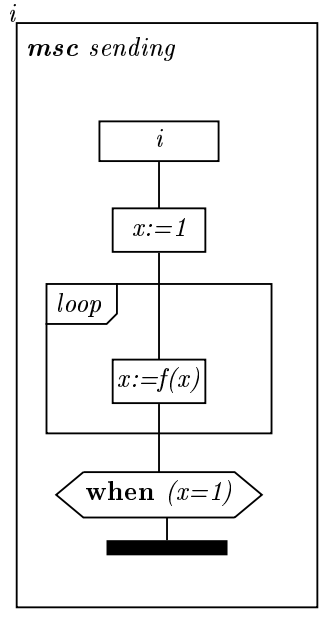



Figure 6.21: Another problem with backtracking semantics

a semantics only for basic MSCs plus inline expressions plus data, not for time) – even worse, that it is actually impossibile to make such a semantics. Such problems are likely to keep occurring wherever a language is developed much faster than its associated semantics. It would work better to develop both hand-in-hand.

This has been done for data and guards, leading to the problems and solutions identified in this chapter. For time on the other hand, the semantic checks seem to have been insufficient. There have certainly been ideas regarding the semantics of time in MSC [BAL97b, GDO98, Klu99, MH00], but all of these had elements that make an integration with the existing MSC'96 semantics hard.

# Chapter 7

# Message Refinement in MSC

## 7.1 Introduction

### 7.1.1 Motivation

One of the areas where MSC is most used, and the one for which the language was originally developed [GR89], is in the description of telecommunication protocols. Real life telecommunication protocols often have different levels of interpretation. Something that is regarded a single message at one level, can be a packet of messages at another, while on yet a lower level a number of regulation messages such as "are you ready to receive?" and "transmission successfully completed" might be added. At the lowest level, there are just a large number of bits being transferred in both directions.

As one single level is already quite complex by itself, one does not want to be concerned by what is going on at at the lower levels when specifying a higher one. However, in MSC this can currently only be done by dropping those lower levels altogether, which might also be undesirable. One might be interested in possible interactions between the various levels, or the computer system that is used to test the implementation might only be able to interpret the communication at a lower level.

Thus, one would like to adapt the formalism in such a way that it is possible to switch between different levels. That way one can design the system or protocol at one level while still being able to see the result at a lower level. In this paper we will introduce the concept of *message refinement*, in which one message can be used to denote a collection of events, as a construct that can be used to make such switches.

### 7.1.2 Composition and Refinement – A Historical Outline

Much discussion has been going on about the possibility of combining several MSCs to create one larger one. In many applications, MSCs tend to become unduly large, spanning several pages. One would like to break those up into smaller parts in order to gain a better overview.

In the oldest MSC-standard, MSC'92 [IT93], only one operation to break up or combine MSCs was defined, namely the so-called *instance refinement* [MR96]. Here one instance is used to show the behaviour of several instances. An example of instance refinement is given in Figures 7.1 to 7.3.

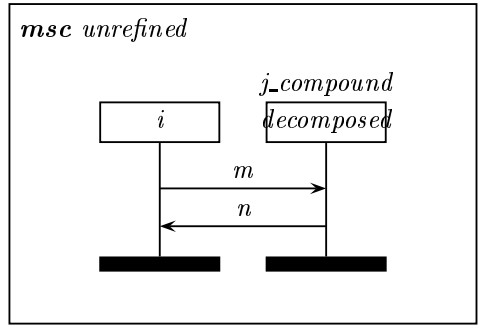



Figure 7.1: Instance refinement: Original MSC

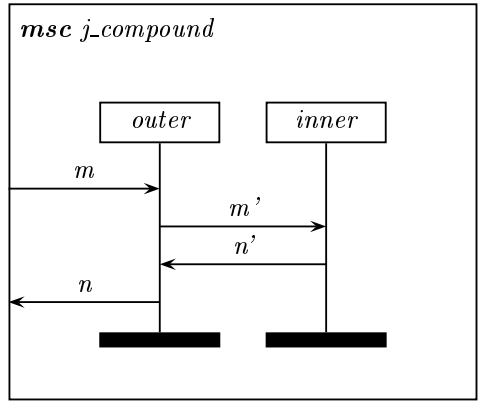



Figure 7.2: Instance refinement: Refining MSC

The instance *j_compound* in Figure 7.1 is *refined* by the MSC in Figure 7.2 That is, the middle MSC shows the internal behaviour of that instance, which appears to consist of two parts that communicate with each other as well as with their mutual environment. The external behaviour of the refining MSC should, of course, be equal to that of the instance to be refined – in this case, first receiving $m$, then sending $n$. Together the two MSCs shown here describe the same behaviour the single MSC in Figure 7.3 describes.

In MSC'96 [IT96], some more features were added to explicitly compose MSCs, namely MSC reference expressions and High-level MSCs (see Chapter 4.

The idea of refinement (using one entity to stand for several of them) could be extended. Two logical ways of doing this are action refinement, in which a local

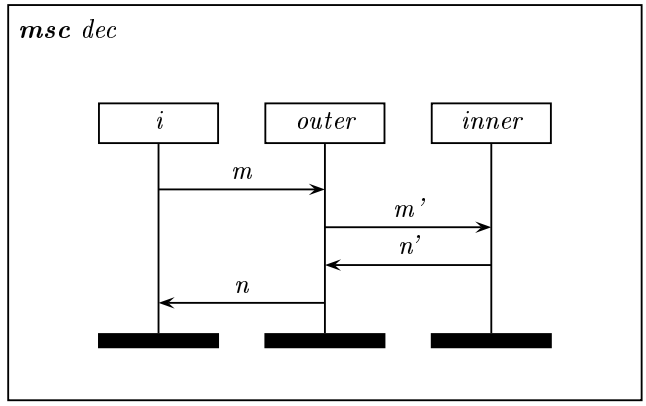



Figure 7.3: Instance refinement: Equivalent MSC

action stands for a number of actions, and message refinement, in which one message stands for a larger protocol consisting of several messages and other events. With the appearance of MSC'96, action refinement adds little, as it can easily be modelled by replacing the action by a one-instance reference MSC (see Chapter 4.3.4). Message refinement will be addressed in this chapter. We will also be proposing another addition to the language, namely synchronous communication.

## 7.2 Message Refinement

### 7.2.1 Protocol MSCs

The basic idea behind message refinement is to use a single message as the notation for some more complex behaviour. A separate MSC then defines this behaviour. In general, this behaviour will be some type of protocol, describing how the information exchange which is represented by the message will occur.

The idea behind message refinement is to have one message stand for an MSC of its own. This MSC, as it shows the protocol used to send the original message, we will call a *Protocol MSC*. What are the properties of such an MSC?

First, there will be two instances, the *sender* and the *receiver*, that are to take the roles of the instances sending and receiving the message to be refined in the unrefined MSC (that is, the MSC in which only the high-level message is shown, the MSC in which the message is 'replaced' by the protocol MSC will be termed the *refined MSC*). The protocol MSC may contain other instances as well. These describe (parts of) the medium between the communicating processes, or perhaps parts of the communicating processes themselves that specifically serve purposes in the input or output process only.

Furthermore, as there should be some sort of communication from the sender to the receiver, it is reasonable to assume there is some event at the sender that necessarily happens before some event at the receiver. When $e_1$ necessarily happens

before $e_2$, we will write $e_1 << e_2$. That is, $e_1 << e_2$ iff $e_1$ is before $e_2$ in every possible trace (allowed sequence of events) of the MSC.

A third point is that we want our MSC to reach neither a deadlock (in which the system has not successfully terminated and yet is unable to perform any actions) nor a lifelock (in which the system keeps on running in loops without ever terminating). If any of these two would be the case, the protocol MSC could not really be regarded as just a refinement of the original message, as it would add some other behaviour as well. Deadlock is forbidden in the MSC standard [Ren95], and an algorithm has been published to check for it [BAL97a].

Putting this all together we come to the definition set out below:

**Definition 44** A protocol MSC is an MSC with the following added requirements:

1. There are two different special instances, which are termed the sender and the receiver. The other instances (if present) are termed *internal instances*.
2. There are events $e_1$ at the sender and $e_2$ at the receiver such that $e_1 << e_2$.
3. The MSC is free of deadlocks, and every finite beginning of a trace of the MSC can be extended to a finite trace.

### 7.2.2 Message Refinement

Having defined what a Protocol MSC is, we next define what Message Refinement means. Thus, given an MSC and a message in that MSC, what is the result when we replace the message by a given Protocol MSC? To define an MSC, we need to specify its instances and events, and the orderings between these events.

If an MSC $k$ has a message $m$ that is to be refined by a protocol MSC $p$, we expect not to find $!m$ and $?m$ in the resulting MSC, as they have been replaced by $p$. All other events of $k$ will be there, and are as much as possible unchanged. Likewise, all events of $p$ are present. They too are as much as possible unchanged. All events of $p$ that are on the sender taken together replace the event $!m$ of $k$. Thus, apart from their own orderings in $p$ they also have to confirm to all orderings of $!m$ in $k$.

**Definition 45 (Message Refinement)** Let $k$ be an MSC, let $m$ be a message of $k$, that is, a message for which the sending $!m$ and the receipt $?m$ are events of $k$, and let $p$ be a protocol MSC. Then the message refinement of $m$ by $p$ in $k$ is the MSC with the following characteristics.

Its instances are all instances of $k$, and all internal instances of $p$.

Its events are all events of $k$ with the exception of $!m$ and $?m$, and all events of $p$. Those events which in $p$ are at the sender instead placed at the instance at which the event $!m$ takes place in $k$. Likewise, the events at the receiver are placed at the instance at which $?m$ takes place in $k$. The other events of $p$, and the remaining events on $k$ are not changed.

There is an ordering of a given sort $e << e'$ between two events $e$ and $e'$ (for example, an instance order or a causal order) iff one of the following is the case:

* $e$ and $e'$ are both events of $k$ and $e <<_k e'$.
* $e$ and $e'$ are both events of $p$ and $e <<_p e'$.
* $e$ is an event of $k$ with $e <<_k !m$ and $e'$ is an event at the sender of $p$.
* $e'$ is an event of $k$ with $?m <<_k e'$ and $e$ is an event at the receiver of $p$.

We will denote the message refinement of $m$ by $p$ in $k$ by $k[p/m]$. An example of message refinement we see in Figures 7.4 to 7.6.

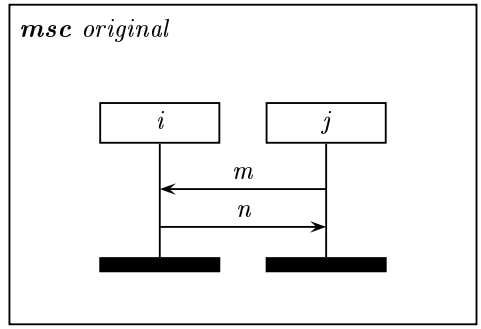



Figure 7.4: Message refinement: original MSC

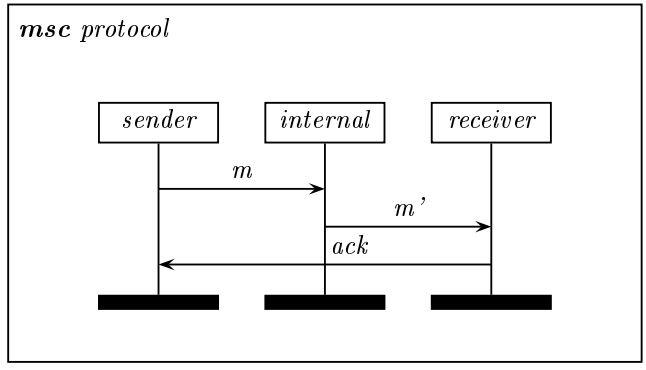



Figure 7.5: Message refinement: refining MSC

The MSC in Figure 7.4 is the original MSC, the MSC in Figure 7.5 the protocol MSC, and the one in Figure 7.6 the resulting MSC after message $m$ has been refined by the protocol MSC. For example, because $!m$ is before $?n$ and at the same instance $j$ in the original MSC, and $!m$ and $?ack$ are at the sender of the protocol MSC, they are also at instance $j$ and before $?n$ in the resulting MSC.

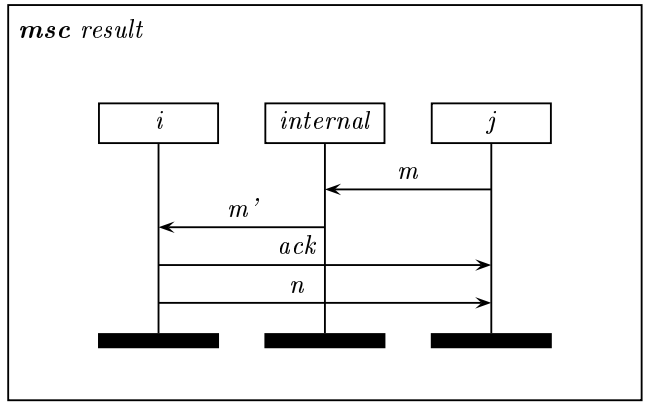



Figure 7.6: Message refinement: equivalent MSC

### 7.2.3 When is Message Refinement Allowed?

In Figures 7.7 to 7.9, a problem appears: the MSC in Figure 7.7 and the protocol MSC in Figure 7.8 are both perfectly valid MSCs. Yet, refining $m$ by the given protocol MSC, will result in the MSC in Figure 7.9, which contains a deadlock. After $m$ has been sent, all three instances are waiting for a message that will never arrive.

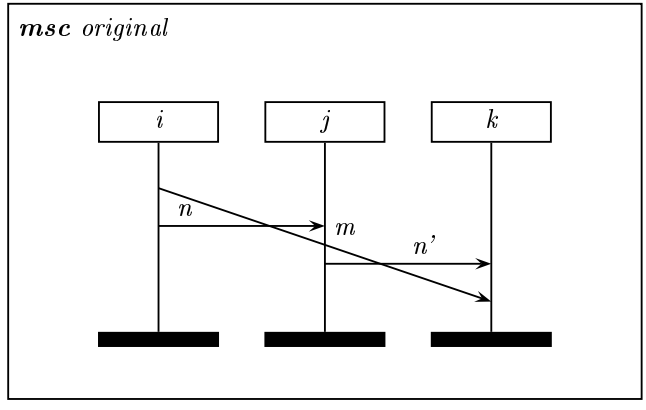



Figure 7.7: A problem: original MSC

Of course this is undesirable behaviour, so we would like to prevent it. However, to do so we need to know when such a situation might occur. We will see that for this purpose it is useful to distinguish between two types of protocol: *unidirectional* and *bidirectional* protocols. In a unidirectional protocol, information flows in just one direction. In a bidirectional protocol, interaction occurs:

**Definition 46** A protocol MSC is bidirectional if in each trace of the MSC there is

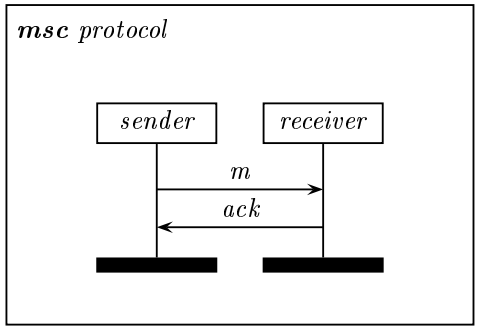



Figure 7.8: A problem: protocol MSC

an event $e$ at the receiver and an event $e'$ at the sender such that $e$ takes place before $e'$, and is unidirectional otherwise.

We first look at unidirectional protocol MSCs. They are very close to the intuition of a single message. No deadlocks are created by the refinement of messages with unidirectional protocols, as the following theorem shows:

**Theorem 47** Let $k$ be an MSC, $m$ a message of $k$, and $p$ be a unidirectional protocol MSC. Then, provided $k$ and $p$ have no deadlocks themselves, $k[p/m]$ has no deadlocks either.

**Proof** Suppose $k[p/m]$ contains a deadlock. Then there should be events $e$ and $e'$ such that $e << e'$ and $e' << e$ simultaneously hold. If there were no such pair in which $e$ is an event of $k$ and $e'$ one of $p$, then the pair would already have caused a deadlock in either $k$ or $p$, so we may assume that $e$ and $e'$ are events of $k$ and $p$, respectively.

$e << e'$ then implies that either $e <<_k !m$ ($<<_k$ of course being the $<<$-ordering of the original MSC $k$) and $e'' <<_p e'$ for some event $e''$ at the sender, or $e <<_k ?m$ and $e'' << e'$ for some event $e''$ of the receiver. Likewise, $e' << e$ implies that either $!m <<_k e$ and $e' <<_p e''$ for some event $e''$ of the sender, or $?m <<_k e$ and $e' <<_p e''$ for some event $e''$ at the receiver.

Because $!m <<_k ?m$, the only way in which $e <<_k !m$ or $e <<_k ?m$ can be combined with $!m <<_k e$ or $?m <<_k e$ without causing a deadlock in $k$ is when $!m <<_k e <<_k ?m$. Then it must be the case that $e <<_p e'$ for some $e''$ at the receiver and $e' <<_p e'''$ for some $e'''$ at the sender. However, in that case $e'' <<_p e'''$, which contradicts the unidirectionality of $p$. Thus we see there are no such $e$ and $e'$, so the refined MSC is free of deadlocks. ■

Bidirectional protocols are trickier. Here the anomaly shown in Figures 7.7 to 7.9 can occur. Luckily we can give the exact conditions under which it occurs. Intuitively one can say that the output and the input of the $m$ must be able to happen arbitrarily close to each other to avoid a deadlock.

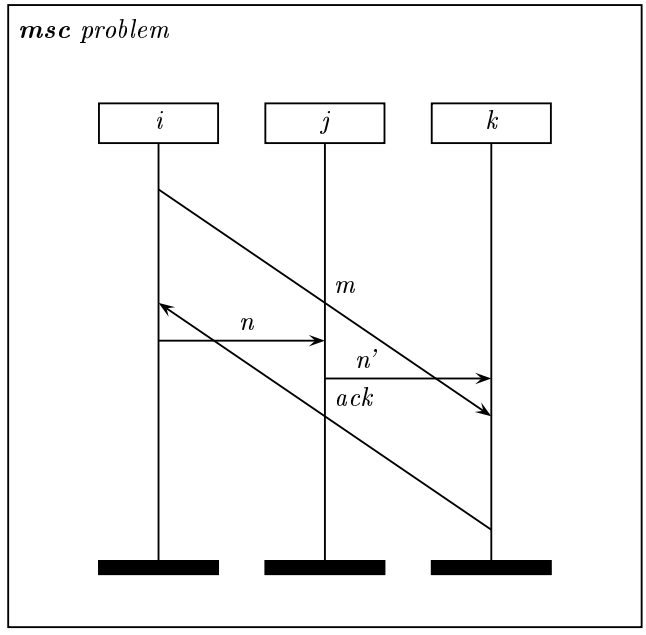



Figure 7.9: A problem: equivalent MSC

**Theorem 48** Let $k$ be an MSC, $p$ a bidirectional protocol MSC, and $m$ a message of $k$. Then $k[p/m]$ is free of cycles if and only if the following conditions hold:

1. $!m$ and $?m$ are not at the same instance in $k$
2. There is no event $a$ such that $!m << a <<?m$

**Proof** if: If the conditions are met, there is a trace where $!m$ and $?m$ follow each other immediately. A valid trace of the refined MSC can now be found by taking such a trace, and replacing $!m{\cdot}?m$ in this trace by any trace of $p$, renaming instances where needed.

only if: If $!m$ and $?m$ are at the same instance in $k$, then in the refined MSC, each event coming from the sender will come before each event coming from the receiver. This will obviously lead to a deadlock if the protocol is bidirectional.

Now suppose there is an event $!m << a <<?m$. There are events $e$ on the receiver and $e'$ on the sender such that $e << e'$ in $p$. However, in $k[p/m]$ we now have $a << e << e' << a$, and thus a deadlock. ■

### 7.2.4 Synchronous Communication

In the present context it would be desirable to have an extra construct in the language to show *synchronous communication*, that is, a message being sent which does not

take any time to go from the source to the destination directory. This looks like a useful extension in itself as well.

Such a synchronous communication can be implemented semantically in two ways: firstly as a single action that is shared by two instances, and secondly as two actions that have to be done without any other action between them. The first method of interpretation is probably preferable, because the second one is very hard to implement in process algebra – or in any of the other formalisms that have been used for proposed semantics for MSCs, for that matter. Anyway, any of the two representations can easily be translated into the other.

If the construct of synchronous communication would be present in the language, then avoiding deadlocks caused by message refinement can be done in the following way.

**Requirement 49** A normal message may only be refined by a unidirectional protocol. A synchronous message may only be refined by a bidirectional protocol.

## 7.3 Semantics

We will now try to give an operational semantics (in the style of [Ren99]) for message refinement. Here $k[p/m]$ is the refined version of $k$, with $p$ for $m$, while $k[p/m]^*$ is the same, but after $!m$, or in fact any of the events that replaces it, has already taken place. We will not explain these semantics any further, as we think there is a better option that will be given below. These semantics assume that $!m$ and $?m$ take place in $k$ exactly once, and the internal instances of $p$ are different from any instances in $k$.

$$\frac{k \xrightarrow{a} k', a \notin \{!m, ?m\}}{k[p/m] \xrightarrow{a} k'[p/m]} \qquad \frac{k \xrightarrow{?m} k', k'\downarrow, p\downarrow}{k[p/m]^*\downarrow}$$

$$\frac{p \xrightarrow{a} p', i(a) \notin \{\text{sender}, \text{receiver}\}}{k[p/m] \xrightarrow{a} k[p'/m]} \qquad \frac{k\downarrow, p\downarrow}{k[p/m]\downarrow}$$

$$\frac{k \xrightarrow{!m} k', p \xrightarrow{a} p', i(a) = \text{sender}}{k[p/m] \xrightarrow{a} k'[p'/m]^*} \qquad \frac{k\downarrow, p\downarrow}{k[p/m]^*\downarrow}$$

$$\frac{k \xrightarrow{!m} k', k' \xrightarrow{?m} k'', p \xrightarrow{a} p', i(a) = \text{receiver}}{k[p/m] \xrightarrow{a} p' \circ k''} \qquad \frac{k \cdots\!\xrightarrow{a} k'', p \cdots\!\xrightarrow{a} p''}{k[p/m] \cdots\!\xrightarrow{a} k''[p''/m]}$$

$$\frac{k \xrightarrow{a} k', a \neq ?m, p \cdots\!\xrightarrow{a} p''}{k[p/m]^* \xrightarrow{a} k'[p''/m]^*} \qquad \frac{k \cdots\!\xrightarrow{a} k'', p \cdots\!\xrightarrow{a} p''}{k[p/m]^* \cdots\!\xrightarrow{a} k''[p''/m]^*}$$

$$\frac{p \xrightarrow{a} p', i(a) \neq \text{receiver}}{k[p/m]^* \xrightarrow{a} k[p'/m]^*}$$

$$\frac{k \xrightarrow{?m} k', p \xrightarrow{a} p', i(a) = \text{receiver}}{k[p/m]^* \xrightarrow{a} p' \circ k'}$$

However, we prefer another way to include message refinement semantically. If we

let it be not an operation *in* but an operation *on* the language, the problems become much less. With this I mean that message refinement is regarded as another way of writing down the MSC where the message has already been defined. That is, to get the semantics of an MSC with refinement, one performs an operation like the one in Definition 45 (but more precisely defined) to get the refined MSC. The semantics of the MSC with refinement is then defined to be equal to that of this refined MSC. The advantage is that this way the actual semantics of MSC is not changed, and so no new problems can be introduced either.

Let thus $k$ be an MSC, $m$ a message of $k$ and $p$ a protocol MSC. The MSC $k[p/m]$ can then be found in the following way, using the textual syntax of $k$ and $p$ (let $i$ and $j$ be the sending and receiving instance of $m$, respectively):

1. In $p$, replace every occurence of 'sender' by the sending instance of $m$ in $k$, and every occurence of 'receiver' by the receiving instance of $m$ in $k$ (if these happen to be the same instance, one should keep track of what was originally on the sender and what on the receiver).

2. In the syntax of $k$, replace the event $i$ : out $m$ to $j$ by a series of events, consisting of all events originally on the sender in $p$, in the order in which they appear in $p$. Likewise, replace $j$ : in $m$ from $i$ by the series of all events originally on the receiver in $p$.

3. Add to $k$ instance declarations for all instances in $p$ except $i$ and $j$

4. Add to $k$, in the order in which they are in $p$, all events of $p$ which were not yet added in step 2.

Synchronous communication can be semantically included rather easily. A synchronous communication can simply be implemented as a single event that has a place in the instance ordering of two different instances. Such a construct does not seem to cause any major problems.

## 7.4 Conclusions

An important issue in MSC is the addition of various ways of composition, that is, combining a number of smaller MSCs into one large MSC. A new way has been put forward in this chapter, namely message refinement, in which a message can be replaced by a protocol consisting of a number of messages and possibly other events.

These protocols can be divided into two groups, namely unidirectional protocols and bidirectional protocols. Replacing a message by a unidirectional protocol causes no problems, but replacing it by a bidirectional protocol might cause deadlocks. One solution to this problem is the addition of synchronous communication, which might also be a useful addition to the language of itself. If we allow only synchronous messages to be replaced by bidirectional protocols, no deadlocks will occur.

To avoid problems in the semantics of MSC, it would be better to define protocol refinement, and other composition techniques also, not as an operator *in* the language, but as an operator *on* the language, describing a way in which MSCs can be changed

into other MSCs. This way, no complicated semantic constructs are necessary to implement them. For example, above we could do without a rather complicated set of rules by introducing a relatively simple algorithm to translate an MSC with message refinement into a 'standard' MSC.

# Chapter 8

# Interrupt and Disrupt in MSC

## 8.1 Introduction

Although extra features been added to the MSC language twice [IT96, IT00], there remains a wish for new features to be added. In this chapter, we will look into one of these proposed extensions, namely disrupt and interrupt. A disrupt means that the system starts executing one type of behaviour, but at a certain point is disrupted, and starts executing another behaviour instead. An interrupt is similar, but after an interrupt, the system returns to the previous behaviour, while after a disrupt, this does not happen.

Our opinion is that the semantics of a new construct should be well thought out before the construct is added to the language. For disrupt and interrupt this chapter attempts to make such a pre-introductory semantic overview. We will show the most important of the many choices that have to be made, and will show some of the problems that might occur if these operators are introduced.

## 8.2 Syntax

If disrupt and interrupt would be included in the language, there would not only be a need for a semantics, but for a syntax as well. These two subjects are not independent. Semantics that fit well with a certain syntax can be clumsy or counter-intuitive when combined with another syntax, and vice versa.

The main distinction here is between *local* and *global* interrupt (or disrupt). The difference here is the period during which the disrupt or interrupt can take place. In a local interrupt or disrupt, the disrupt or interrupt can only take place at a single point in time, while a global interrupt or disrupt can do so at any time during a given period.

We would like to stay as close as possible to the existing MSC syntax. In order to do so we will use inline expressions to describe the disrupt and interrupt.

In Figure 8.1, we see the proposed syntax for local interrupt. At the point where the interrupt is shown, the behaviour of the encompassing MSC could be interrupted by the behaviour of the interrupt inline expression.

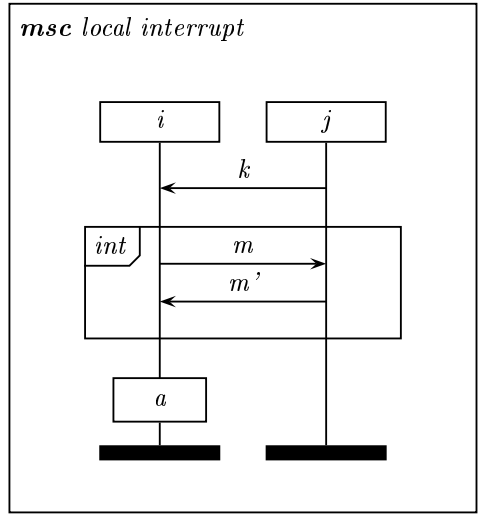



Figure 8.1: Proposed syntax for local interrupt

The MSC in Figure 8.2 shows our proposed syntax for global interrupt. As it is important here to specify at which times an interrupt is possible, we now have a two-component inline expression. The lower part is equal in function to the inline expression in the left example, giving the interrupting sequence. The interrupt can take place at any time when the system is in the upper part of the inline expression.

Let's look more precisely at what is done in the local case. The MSC can essentially have two different behaviours:

- Not doing the interrupt.
- Doing the interrupt, and doing it at exactly at the time given.

However, such a construct would not be an actual addition to the language. The `opt`-construct has exactly this same meaning – when something is placed in an 'optional' inline expression it can either be done at that precise moment, or not at all. Likewise a local disrupt could be replaced by an 'exception'.

Such an equality has advantages and disadvantages. The advantage is, that a semantics is easily found, and will cause no problems with the rest of the language, or at least no problems that were not already there. The problems are thus much smaller than they would be with most extensions. The disadvantage is, that adding such a local disrupt or interrupt will not make the language stronger while it would make it larger. That way some of the problems of language addition would be met without any useful extension even having been made.

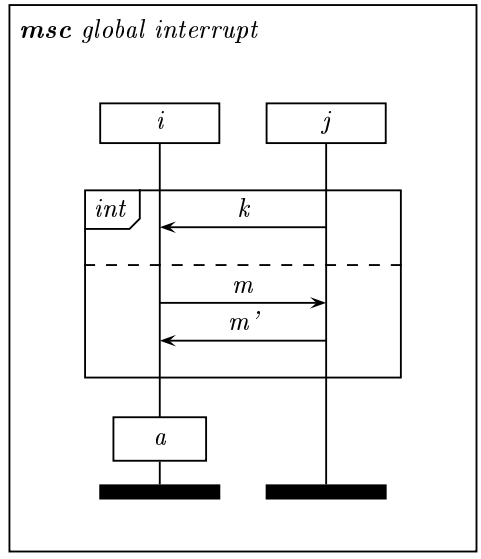



Figure 8.2: Proposed syntax for global interrupt

We feel that local interrupt and disrupt are not useful, while if they are, they are still semantically uninteresting. We will therefore in the rest of this chapter only discuss global interrupt and disrupt.

### 8.2.1 Semantic Choices

For global and interrupt a number of other decisions have to be made. Each of these will influence the resulting semantics. We see at least the following:

1. Can an interrupt take place only once, or any number of times?
2. In the second case, can the system be interrupted more than once between any two actions?
3. If yes, can one interrupt interrupt the other?
4. Can an interrupt or disrupt take place before the first and/or after the last action of the interrupted behaviour?
5. Are all instances interrupted or disrupted at the same time by an interrupt or disrupt, or is it enough that all instances are interrupted or disrupted at some time? This point will be explained in more detail below, as it is an important choice, which is not so obvious, and the most obvious answer might well not be the best one.

The last point above deserves some extra discussion. At first thought it might seem that the first interpretation is more natural – an interrupt or disrupt should

work on the whole system, or at least on all instances on which it is specified, at once. However, when we look at an example, this might not be as obvious. See for example Figure 8.3. It models a Telnet-protocol: The server sends two (packets of) messages to the client. The client can check whether the server is still alive by sending an *ayt*-signal ('are you there?'), to which the server answers by saying '*yes*'.

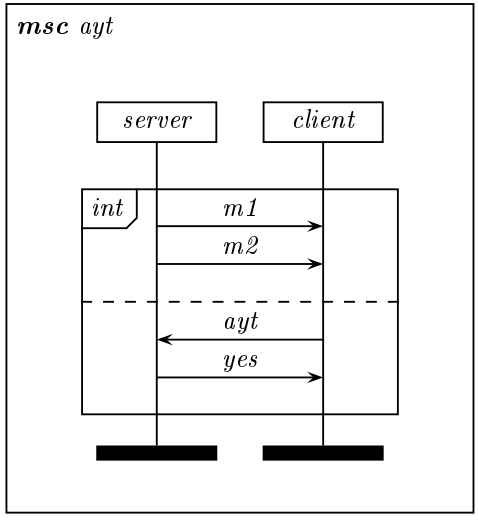



Figure 8.3: Example interrupt MSC: Are you there-protocol

Now if we regard the interrupt as interrupting all instances at once, then the sending of the *ayt*-message will block the action of the server. However, how is the server in a practical case to know that the *ayt* has been sent? It only notices this upon its receipt, so it is logical to assume it will not be blocked before that. Likewise, the server does not know when the 'yes' is received, only when it is sent. Thus letting the server be interrupted during all of the period leads to some possibly unwanted extra causalities. The more logical choice might be to let each process be interrupted separately, that is, each process has to do the interrupting actions at some point without doing anything else in between, but they do not have to do it all at the same time. Choosing to have all instances interrupted or disrupted at the same time goes contrary to the nature of MSC, where otherwise all communication is explicit and asynchronous.

When one would choose to have all instances interrupted or disrupted at the same time, one would in fact introduce synchronization points, which are contrary to the nature of MSC as it has been practiced until now.

For the other questions our preferred answers are as shown below. However, we do not feel very strongly about these questions, and if it appears that other choices would be closer to the wishes of the users these are the ones that have to prevail.

1. Any number of interrupts.

2. More than one interrupt between two events possible.

3. No interrupts interrupting interrupts.

4. Interrupts and disrupts possible at the given times.

## 8.3 Semantics

Based on the choices in the last section, we create an operational semantics for the disrupt and interrupt operators. There have been more attempts to make process algebraic description of the disrupt and interrupt operators (see for example [BBK86, Die94, BB00]), but the presence in particular of weak sequencing causes the semantics of these operators for MSC to be different (and more complicated) than in general process algebra.

We will define operators $\triangleleft$ and $\blacktriangleleft$ to denote the interrupt and disrupt respectively. That is, $x \triangleleft y$ is '$x$ (possibly) interrupted by $y$' and $x \blacktriangleleft y$ '$x$ (possibly) disrupted by $y$'.

When could $x \blacktriangleleft y$ do an action $a$? There are in fact two possibilities: Either $x$ does the action, or $y$ does it. In the first case, the resulting expression can still be disrupted. In the second case, all instances, except the one on which $a$ takes place do not have to have been disrupted yet. They have to be disrupted at some time, but that time can be somewhere in the future, and upto that point can still do actions of $x$. To describe this situation, we introduce the forced disrupt, $\underline{\blacktriangleleft}$. $x \underline{\blacktriangleleft} y$ can do $x$, but must at some time in the future be disrupted by $y$. However, this is not enough yet: The action of $y$ that has already been taken forbids any actions on the same instance of $x$ to be taken. That is, some events of $x$ may still happen before being disrupted, but others have already been disrupted. Therefore we add to the forced disrupt a set of instances $S \subset I$ that have already been disrupted. $x \underline{\blacktriangleleft}^S y$ can now do any action from $y$ (provided $y$ could do it), but actions from $x$ only if they happen on an instance not in $S$. We will be giving operational semantics for both $\blacktriangleleft$ and $\underline{\blacktriangleleft}$, although it would be possible to make the semantics without using $\blacktriangleleft$, as it can be eliminated through the equation $x \blacktriangleleft y = (x \underline{\blacktriangleleft}^{\emptyset} y) \mp x$.

For the interrupt $\triangleleft$ we also define a forced interrupt $\underline{\triangleleft}$, but in this case we cannot get away with re-defining the interrupt, as we need to keep the possibilities of further interrupts.

We have to check the behaviour of our operators for three operational modifiers: $x\downarrow$, which is true iff $x$ can terminate, $x \xrightarrow{a}$, which gives the result of doing an $a$ on $x$, and $x \xrightarrow{\cdots a}$, which gives the result of $x$ permitting $a$.

First, we consider termination. $x \blacktriangleleft y$ can terminate in two ways: Either $x$ terminates, or $y$ disrupts $x$ and then terminates without doing any action. For $x \underline{\blacktriangleleft}^S y$ to terminate it is necessary and sufficient for $y$ to terminate, while $x \triangleleft y$ can terminate by just $x$ terminating. Finally, for $x \underline{\triangleleft}^S y$ to terminate, both $x$ must be ready and $y$ must have no interrupting actions left, so $x \underline{\triangleleft}^S y$ only terminates if both $x$ and $y$ terminate. Thus:

$$\frac{x\downarrow}{x \blacktriangleleft y\downarrow} \qquad \frac{y\downarrow}{x \underline{\blacktriangleleft}^S y\downarrow} \qquad \frac{x\downarrow}{x \triangleleft y\downarrow} \qquad \frac{x\downarrow, y\downarrow}{x \underline{\triangleleft}^S y\downarrow}$$

$$\frac{y\downarrow}{x \blacktriangleleft y\downarrow}$$

Next we look at what happens for doing a step. When could an action $a$ be taken by $x \blacktriangleleft y$? There are in fact two possibilities: either $x$ took the step, after which a disrupt of course could still take place, or $y$ disrupted, and took the step. In the second case, we would get into a forced disrupt situation, where the instance on which $a$ took place (which is denoted $l(a)$) is already disrupted.

In a forced disrupt $x \underline{\blacktriangleleft}^S y$, $x$ could only take the action $a$ if the instance on which $a$ takes place was not already disrupted, that is, if $l(a) \notin S$. Actions of $y$ can always occur, and if one does then necessarily its instance must be disrupted as well.

With the interrupt $x \triangleleft y$ we again see two possibilities. Either the action can be done by $x$, and nothing shocking is happening, or it can be done by $y$. In the latter case we get into a forced interrupt. However, we will have to keep the 'old' interrupt as well, because the process could be interrupted a second time.

$x \underline{\triangleleft}^S y$ is the most difficult one in this aspect. $x$ can execute the step $a$ if $l(a) \notin S$, but it can also do it if $y$ allows $a$. This is, because in this case the instance on which $a$ takes place has done all it has to do, so it is not interrupted anymore, and can do steps from the main execution ($x$) again. Steps from $y$ work just like the former cases.

There is another complicating factor here: We can have just one possible step for a given action from a given expression, because the semantics of MSC are completely deterministic. Thus, we have to include a special case for the possibility that both $x$

and $y$ can do a given step. Taken together, this leads to:

$$\frac{x \xrightarrow{a} x', y \not\xrightarrow{a}}{x \blacktriangleleft y \xrightarrow{a} x' \blacktriangleleft y} \qquad \frac{x \xrightarrow{a} x', y \not\xrightarrow{a}, l(a) \notin S}{x \underline{\blacktriangleleft}^S y \xrightarrow{a} x' \underline{\blacktriangleleft}^S y}$$

$$\frac{y \xrightarrow{a} y', x \not\xrightarrow{a}}{x \blacktriangleleft y \xrightarrow{a} x \underline{\blacktriangleleft}^{\{l(a)\}} y'} \qquad \frac{y \xrightarrow{a} y', x \not\xrightarrow{a}}{x \underline{\blacktriangleleft}^S y \xrightarrow{a} x \underline{\blacktriangleleft}^{S \cup \{l(a)\}} y'}$$

$$\frac{x \xrightarrow{a} x', y \xrightarrow{a} y'}{x \blacktriangleleft y \xrightarrow{a} x' \blacktriangleleft y \mp x \underline{\blacktriangleleft}^{\{l(a)\}} y'} \qquad \frac{x \xrightarrow{a} x', y \xrightarrow{a} y', l(a) \notin S}{x \underline{\blacktriangleleft}^S y \xrightarrow{a} x' \underline{\blacktriangleleft}^S y \mp x \underline{\blacktriangleleft}^{S \cup \{l(a)\}} y'}$$

$$\frac{y \xrightarrow{a} y', l(a) \in S}{x \underline{\blacktriangleleft}^S y \xrightarrow{a} x \underline{\blacktriangleleft}^S y'}$$

$$\frac{x \xrightarrow{a} x', y \not\xrightarrow{a}}{x \triangleleft y \xrightarrow{a} x' \triangleleft y} \qquad \frac{x \xrightarrow{a} x', l(a) \notin S, y \not\xrightarrow{a}}{x \underline{\triangleleft}^S y \xrightarrow{a} x' \underline{\triangleleft}^S y}$$

$$\frac{x \xrightarrow{a} x', l(a) \in S, y \cdots\!\xrightarrow{a} y''}{x \underline{\triangleleft}^S y \xrightarrow{a} x' \underline{\triangleleft}^S y''}$$

$$\frac{y \xrightarrow{a} y', x \not\xrightarrow{a}}{x \triangleleft y \xrightarrow{a} (x \underline{\triangleleft}^{\{l(a)\}} y') \triangleleft y} \qquad \frac{y \xrightarrow{a} y', x \not\xrightarrow{a}}{x \underline{\triangleleft}^S y \xrightarrow{a} x \underline{\triangleleft}^{S \cup \{l(a)\}} y'}$$

$$\frac{x \xrightarrow{a} x', y \xrightarrow{a} y'}{x \triangleleft y \xrightarrow{a} x' \triangleleft y \mp (x \triangleleft y) \underline{\triangleleft}^{\{l(a)\}} y'} \qquad \frac{\begin{array}{c} x \xrightarrow{a} x', y \xrightarrow{a} y', l(a) \notin S \\ x \xrightarrow{a} x', y \xrightarrow{a} y', l(a) \in S, y \not\cdots\!\xrightarrow{a} \end{array}}{x \underline{\triangleleft}^S y \xrightarrow{a} x \underline{\triangleleft}^S y'}$$

$$\frac{x \xrightarrow{a} x', y \xrightarrow{a} y', l(a) \in S, y \cdots\!\xrightarrow{a} y''}{x \underline{\triangleleft}^S y \xrightarrow{a} x' \underline{\triangleleft}^S y'' \mp x \underline{\triangleleft}^S y'}$$

Finally, the permission relation. This relation has been introduced in the semantics of MSC to describe the possibility that in the expression $x \circ y$ events of $y$ can go before events of $x$. However, this can only be done if no events on $x$ are on the same instance as the event taking place, or are otherwise forced to go first. This can depend on choices that are made within $x$. In such a case those choices that would have made the event taking place impossible are subsequently disallowed. Thus we get the relation $x \cdots\!\xrightarrow{a} x'$, which denotes that $x$ by permitting an event from another (later) term is reduced to $x'$.

For $x \blacktriangleleft y$ this immediately leads to problems. There are two possibilities here:

Either $x$ has permitted the event, or $y$ has permitted it. However, in the second case, those events of $x$ that would have permitted it, can still take place. That is, $x$ may perhaps not take place in full, but it can still do those actions that are not forbidden by the event just having been allowed. This is not a simple removal of choices as it was with the permission relation for other MSC operators. Here those parts of $x$ that would normally be disallowed by the permission of the events can still take place upto the place where the permission would actually be impossible.

To see how we can deal with this, it is good to look at the forced disrupt $x \underline{\blacktriangleleft}^S y$. Here, in order for an event $a$ to be permitted, it necessarily has to be permitted by $y$. On the other hand, whether or not $x$ permits it is not interesting – any beginning of a trace in $x$ can happen as long as it does not contain any events that are disallowed by the permission of $a$, that is, as long as it does not contain any events on the same instance $l(a)$, independent of whether or not they are part of a complete trace that would have allowed $a$. Thus we see that, if $y$ permits an event $a$ to go over into $y''$, $x \underline{\blacktriangleleft}^S y$ permits that event, and goes over in $x \underline{\blacktriangleleft}^{S \cup \{l(a)\}} y''$. The strange thing of course is, that this is independent of whether or not $x$ permits $a$. The reason is that $x$ cannot terminate anyway, as it will be interrupted by $y$ at some time. Because of this it does not matter whether $x$, or even the trace taken, actually permits $a$, as long as that part of the trace that is actually taken does so. The SOS-rules for permission by $x \blacktriangleleft y$ now follow through the equality $x \blacktriangleleft y = x \mp x \underline{\blacktriangleleft}^{\emptyset} y$.

For $x \triangleleft y$ to permit $a$ it suffices that $x$ does so. If $y$ does not allow $a$, the process cannot be interrupted anymore, if it does it still can. $x \underline{\triangleleft}^S y$, finally, can permit an event only if both $x$ and $y$ do so. Note that it does not matter in this case whether or not $l(a)$ is added to $S$, as all events on $l(a)$ are 'sifted out' by the permission of $a$ anyway. This leads to the following:

$$\frac{x \overset{a}{\cdots\!\rightarrow} x'', y \overset{a}{\not\cdots\!\rightarrow}}{x \blacktriangleleft y \overset{a}{\cdots\!\rightarrow} x''}$$

$$\frac{x \overset{a}{\not\cdots\!\rightarrow}, y \overset{a}{\cdots\!\rightarrow} y''}{x \blacktriangleleft y \overset{a}{\cdots\!\rightarrow} x \underline{\blacktriangleleft}^{\{l(a)\}} y''} \qquad \frac{y \overset{a}{\cdots\!\rightarrow} y''}{x \underline{\blacktriangleleft}^S y \overset{a}{\cdots\!\rightarrow} x \underline{\blacktriangleleft}^{S \cup \{l(a)\}} y''}$$

$$\frac{x \overset{a}{\cdots\!\rightarrow} x'', y \overset{a}{\cdots\!\rightarrow} y''}{x \blacktriangleleft y \overset{a}{\cdots\!\rightarrow} x'' \mp x \underline{\blacktriangleleft}^{\{l(a)\}} y''}$$

$$\frac{x \overset{a}{\cdots\!\rightarrow} x'', y \overset{a}{\not\cdots\!\rightarrow}}{x \triangleleft y \overset{a}{\cdots\!\rightarrow} x''}$$

$$\frac{x \overset{a}{\cdots\!\rightarrow} x'', y \overset{a}{\cdots\!\rightarrow} y''}{x \triangleleft y \overset{a}{\cdots\!\rightarrow} x'' \triangleleft y''} \qquad \frac{x \overset{a}{\cdots\!\rightarrow} x'', y \overset{a}{\cdots\!\rightarrow} y''}{x \triangleleft^S y \overset{a}{\cdots\!\rightarrow} x'' \triangleleft^S y''}$$

## 8.4 Conclusions

Interrupt and disrupt can be introduced into MSC in various ways. These choices have to be made very carefully, because a language construct that is not understood in the same way by all users and other people concerned will cause many more problems than it solves. Restriction in the inclusion of new features is advisable from a more general point of view too.

If interrupt and disrupt are indeed to be included in the language, the first choice is whether a local or a global interrupt is taken. A local interrupt has the advantage of being semantically simple and easily understood, but on the other hand it does not really add anything to the language, so it is nothing but syntactic sugar. A global interrupt on the other hand is semantically quite complicated, which can lead to unclarities. There are also a number of additional choices to be made.

Although in this chapter a semantics for disrupt and interrupt in MSC has been defined, there are still some issues to be dealt with. In particular, the semantics as they are, are rather complicated. Also, a number of choices have been made before creating these semantics. Both factors increase the likelihood that, if these constructs were interrupted in the language, the official semantic meaning of an MSC containing these constructs might be different from the meaning intended by the user.

We feel that, in general, it is a bad thing to let the language grow too fast or too large. The MSC'96 language has not been thoroughly researched. It would be better, in our opinion, to have a solid, stabilized semantics for the existing language, and if possible also for the proposed extensions, before the language is extended. There are other reasons for restraint in the adoption of new features as well: If features are introduced too quickly, tool builders will have problems keeping up. Having too many features also runs the risk of groups of users using only subsets of the language, thus diminishing the advantage that using one single languages has. Another problem is that a large number of features greatly increases the chance that unforeseen interactions between them lead to unwanted or unexpected behaviour.

We do not intend to claim that additions to the language have to be avoided at all costs. Far from that, some additions are certainly useful, and not having any innovation whatsoever will be even more certain to kill the language's applicability than a too generous addition of new features would. However, new features should only be introduced when there is a wish for inclusion by a large number of users, and a well-defined semantics for it.

# Chapter 9

# Conclusions

Formal descriptions, and thus formal languages, can be useful in various parts of the development of a software system. In this thesis, we have looked at some uses of formal languages, as well as the process of defining these languages, and in particular their semantics.

In the first two chapters, the subject of testing has been discussed. In Chapter 2, we introduced a way to use formal methods created for one function (namely model checking) in a completely different area (namely test generation). Using existing methods and tools has the advantage that the amount of work that needs to be done to create tools is much smaller, and innovations in one area can be used in other areas as well.

In Chapter 3, a new language, *LOGAN*, has been developed for the analysis of log files. Although through circumstances that are not related to the work itself, it could not be applied at KPN as was originally intended, the language created seems to be both simple and expressive, and as such seems to be applicable in practice. Using a formal language certainly seems to be a great improvement over the current KPN practice of doing the checking of log files by hand.

The rest of the thesis discusses the language MSC, which is introduced in Chapter 4.

Chapter 5 is the last one about the applications of formal methods. The question of whether a system can be implemented with a given communication architecture is an important one, and the material in this chapter allows one to answer this question from the MSC description of the system using relatively simple algorithms.

The second subject of the thesis, about the development of formal languages and their semantics, also shows up in Chapter 3. Not only has *LOGAN* been developed and a semantics defined, but we also gave an algorithm to check the correspondence between the system described (in this case, a log file) and a description in the language (a pattern).

Chapters 6, 7 and 8 discuss various extensions of MSC, including discussions of their semantics. Much of the work in Chapter 6 has been part of the actual discussion of adding data to the language. Proposals that seem to be reasonable at first sight, might have hidden semantical problems, and the work in this Chapter has helped

the development process of MSC2000 by bringing these semantical problems to the surface, thus allowing them to be resolved before the actual standard was decided upon.

The Chapters 7 and 8 have a similar function for some extensions that have not yet been added to the language, but might be in the future. For message refinement, it seems that most semantical problems can be avoided by making refinement an operation on rather than in the language. For disrupt and interrupt on the other hand, a number of choices have to be made, and the resulting semantics are complicated. This complexity might be a reason to not introduce the constructs to the language, but that is a decision that falls outside the scope of this thesis.

# Bibliography


[Aal99] W.M.P. van der Aalst. Interorganizational workflows: An approach based on Message Sequence Charts and Petri Nets. *Systems Analysis – Modelling – Simulation*, 34(3):335–367, 1999.

[AB95] M. Andersson and J. Bergstrand. Formalizing use cases with Message Sequence Charts. Master's thesis, Department of Communication Systems, Lund Institute of Technology, 1995.

[ABM98] P. Ammann, P. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54, December 1998.

[AEY00] R. Alur, K. Etessami, and M. Yannakakis. Inference of Message Sequence Charts. In *ICSE 2000. Proceedings of the 22nd International Conference on Software Engineering*, pages 304–313, Limerick, Ireland, June 2000.

[AHP96] R. Alur, G.J. Holzmann, and D. Peled. An analyzer for Message Sequence Charts. *Software – Concepts and Tools*, 17(2):70–77, 1996.

[AKB99] M.M. Abdalla, F. Khendek, and G. Butler. New results on deriving SDL specifications from MSCs. In Dsoulli et al. [DvBL99].

[AY99] R. Alur and M. Yannakakis. Model checking of Message Sequence Charts. In J.C.M. Baeten and S. Mauw, editors, *CONCUR'99 Concurrency Theory. 10th International Conference. Eindhoven, The Netherlands, August 1999. Proceedings*, number 1664 in Lecture Notes in Computer Science, pages 114–129. Springer Verlag, 1999.

[Bak96] B.S. Baker. Parametrized pattern matching: Algorithms and applications. *Journal of Computer System Science*, 52(1):28–42, February 1996.

[BAL97a] H. Ben-Abdallah and S. Leue. Syntactic detection of process divergence and non-local choice in Message Sequence Charts. In Brinksma [Bri97], pages 259–274.

[BAL97b] H. Ben-Abdallah and S. Leue. Timing constraints in Message Sequence Chart specifications. In Mizuno et al. [MSHT97], pages 91–106.

[BB97] J. Bergstra and W. Bouma. Models for feature descriptions and interactions. In P. Dini, R. Boutaba, and L.M.S. Logrippo, editors, *Proceedings 4th International Workshop on Feature Interactions in Telecommunication Networks*, pages 31–45. IOS Press, June 1997.

[BB00] J.C.M. Baeten and J.A. Bergstra. Mode transfer in process algebra. Technical Report CSR 00-01, Eindhoven University of Technology, Department of Computer Science, January 2000.

[BBK86] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Syntax and defining equations for an interrupt mechanism in process algebra. *Fundamenta Informaticae*, IX(2):127–186, 1986.

[BC00] F. Bordeleau and D. Cameron. On the relationship between use-case maps and Message Sequence Charts. In Graf et al. [GJL00].

[BD87] S. Budkowski and P. Dembinski. An introduction to Estelle: A specification language for distributed systems. *Computer Networks and ISDN Systems*, 14(1):3–23, 1987.

[BH88] F. Belina and D. Hogrefe. The CCITT-specification and description language SDL. *Computer Networks and ISDN Systems*, 16(4):311–343, 1988.

[BHS91] F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from Protocol Specification*. The BCS Practitioners Series. Prentice-Hall International, 1991.

[BJ97] P. Baker and C. Jervis. Formal description of data. Temporary Document TD-L16, Experts Meeting ITU-T SG 10, Lutterworth, October 1997.

[BK84] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1/3):109–137, 1984.

[BM77] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[BM95] J.C.M. Baeten and S. Mauw. Delayed choice: an operator for joining message sequence charts. In Hogrefe and Leue [HL95], pages 340–354.

[BM99] V. Bos and S. Mauw. A LaTeX macro package for Message Sequence Charts. http://www.win.tue.nl/~sjouke/mscpackage.html, April 1999.

[BOY00] P.E. Black, V. Okun, and Y. Yesha. Mutation of model checker specifications for test generation and evaluation. In *Proceedings Mutation 2000*, October 2000.

[Bri97] E. Brinksma, editor. *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1217 in Lecture Notes in Computer Science. Springer Verlag, 1997.

[BRJ98] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1998.

[BRS01] P. Baker, E. Rudolph, and I. Schieferdecker. Graphical test specification – the graphical format of ttcn-3. In SDL01 [SDL01]. To appear.

[BS95] R. Bræk and A. Sarma, editors. *SDL'95 with MSC in CASE, Proceedings of the 7th SDL Forum*, Oslo, Norway, September 1995. Elsevier Science Publishers/North Holland.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[BW94] P.K. Bohacek and J.N. White. Service creation: The real key to intelligent network revenue. In *Proceedings Workshop Intelligent Networks '94*, Heidelberg, May 1994.

[CBMT96] B. Charron-Bost, F. Mattern, and G. Tel. Synchronous, asynchronous and causally ordered communication. *Distributed Computing*, 9(4):173–191, 1996.

[CE98] T. Cobben and A. Engels. Disrupt and interrupt in MSC: Possibilities and problems. In Lahav et al. [LWFH98], pages 75–83.

[CK94] E. Crabill and J. Kukla. Service processing systems for AT&T's intelligent network. *AT&T Techn. Journal*, 73(6):39–47, 1994.

[CS97] A. Cavalli and A. Sarma, editors. *SDL'97: Time for Testing – SDL, MSC and Trends, Proceedings of the Eighth SDL Forum*, Evry, France, September 1997. Elsevier Science Publishers/North Holland.

[CSE96] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using model-checking. In *Proceedings of the Second SPIN Workshop*, pages 129–146. Rutgers University, New Brunswick NJ, August 1996.

[CV93] E.J. Cameron and H. Velthuijsen. Feature interactions in telecommunications systems. *IEEE Communications Magazine*, pages 18–23, August 1993. Special Issue on Feature Interactions.

[DFG+97] G. Das, R. Fleischer, L. Gasieniec, D. Gunopulos, and J. Kärkkäinen. Episode matching. In A. Apostolico and J. Hein, editors, *Combinatorial Pattern Matching, 8th Annual Symposium*, number 1264 in Lecture Notes in Computer Science, pages 12–27, Aarhus, Denmark, June 1997. Springer Verlag.

[DH99] W. Damm and D. Harel. LSCs: Breathing life into Message Sequence Charts. In F. Cianconhi and R. Gorrieri, editors, *Proceedings 3rd IFIP Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99)*, pages 293–312, 1999.

[Die94] B. Diertens. New features in PSF I: Interrupts, disrupts, and priorities. Technical Report P9417, Programming Research Group, University of Amsterdam, 1994.

[dM93] J. de Man. Towards a formal semantics of Message Sequence Charts. In Færgemand and Sarma [FS93].

[DvBL99] R. Dsoulli, G. von Bochmann, and Y. Lahav, editors. *SDL'99: The Next Millennium, Proceedings of the 9th SDL Forum*, Montreal, Canada, June 1999. Elsevier Science Publishers/North Holland.

[EFM97] A. Engels, L.M.G. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In Brinksma [Bri97], pages 384–398.

[EFM99] A.G. Engels, L.M.G. Feijs, and S. Mauw. MSC and data: Dynamic variables. In Dsoulli et al. [DvBL99], pages 105–120.

[Ek93] A. Ek. Verifying Message Sequence Charts with the SDL validator. In Færgemand and Sarma [FS93], pages 237–249.

[EMR97a] A. Engels, S. Mauw, and M.A. Reniers. A hierarchy of communication models for message sequence charts. Technical Report CSR 97-11, Eindhoven University of Technology, Department of Computer Science, 1997.

[EMR97b] A. Engels, S. Mauw, and M.A. Reniers. A hierarchy of communication models for message sequence charts. In Mizuno et al. [MSHT97], pages 75–90.

[End00] I.D. van den Ende. Grammars compared: A study on determining a suitable grammar for parsing and generating natural language sentences in order to facilitate the translation of natural language and MSC use cases. Technical Report CSR 00-08, Eindhoven University of Technology, Department of Computer Science, March 2000.

[Eng98] A. Engels. Message refinement: Describing multi-level protocols in MSC. In Lahav et al. [LWFH98], pages 67–74.

[Eng00] A. Engels. Design decisions on data and guards in MSC2000. In Graf et al. [GJL00], pages 33–46.

[EVD89] P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors. *The Formal Description Technique LOTOS*. Elsevier Science Publishers/North Holland, 1989.

[Far96] F. Farnschädler. Kimba-MSC – Kimwitu-basiert annotierte Message Sequence Charts. Studienarbeit, Institut für Informatik, IMMD VII, Friedrich-Alexander Universität Erlangen-Nürnberg, October 1996.

[Fei99] L.M.G. Feijs. Generating FSMs from interworkings. *Distributed Computing*, 42(1):31–40, 1999.

[Fei00] L. Feijs. Natural language and message sequence chart representation of use cases. *Information and Software Technology*, 42(9):633–647, June 2000.

[FJ96] L.M.G. Feijs and M. Jumelet. A rigorous and practical approach to service testing. In H. Burkhardt and A. Giessler, editors, *Testing of Communicating Systems, IFIP TC6 Nineth International Workshop on Testing of Communicating Systems (IWTCS'96)*, pages 175–190. Chapman & Hall, 1996.

[FJJV97] J.C. Fernandez, C. Jard, T. Jéron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29(1/2):123–146, July 1997. Special issue on Industrially Relevant Applications of Formal Analysis Techniques.

[FM98] L.M.G. Feijs and S. Mauw. MSC and data. In Lahav et al. [LWFH98], pages 85–94.

[FR91] O. Færgemand and R. Reed, editors. *SDL'91: Evolving Methods, Proceedings fo the Fifth SDL Forum.* Elsevier Science Publishers/North Holland, 1991.

[FS93] O. Færgemand and A. Sarma, editors. *SDL'93 – Using Objects, Proceedings of the Sixth SDL Forum*, Darmstadt, October 1993. Elsevier Science Publishers/North Holland.

[GDO98] V. Grabowski, C. Dietz, and E.R. Olderog. Semantics for timed Message Sequence Charts via constraints diagrams. In Lahav et al. [LWFH98], pages 251–260.

[GGR01] J. Grabowski, P. Graubmann, and E. Rudolph. HyperMSCs with connectors for advanced visual system modelling and testing. In SDL01 [SDL01]. To appear.

[GH99] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proceedings of the Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on Foundations of Software Engineerings*, Toulouse, France, September 1999.

[GH00] J. Grabowski and D. Hogrefe. TTCN, SDL- and MSC-based specification and automated test case generation for INAP. In *Proceedings of the 8th International Conference on Telecommunication Systems (ICTS'2000) – Modeling and Analysis*, Nashville, March 2000.

[GHN93] J. Grabowski, D. Hogrefe, and R. Nahm. Test case generation with test purpose specification by MSCs. In Færgemand and Sarma [FS93], pages 253–265.

[GHNS95] J. Grabowski, D. Hogrefe, I. Nussbaumer, and A. Spichiger. Test case specification based on MSCs and ASN.1. In Bræk and Sarma [BS95], pages 307–322.

[GHRW98] T. Gehrke, M. Huhn, A. Rensink, and H. Wehrheim. An algebraic semantics for Message Sequence Chart documents. In S. Budkowski, A. Cavalli, and E. Najm, editors, *Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE/PSTV'98)*, pages 3–18. Kluwer Academic Publishers, 1998.

[GJL00] S. Graf, C. Jard, and Y. Lahav, editors. *SAM2000. 2nd Workshop on SDL and MSC*, Col de Porte, Grenoble, June 2000.

[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*, pages 3–10, Lisbon, 1989. Elsevier Science Publishers/North Holland.

[Gra90] J. Grabowski. Statische und dynamische Analysen für SDL-Spezifikationen auf der Basis von Petri-Netzen und Sequence Charts. Master's thesis, Universität Hamburg, FB Informatik, 1990.

[GRG91] P. Graubmann, E. Rudolph, and J. Grabowski. Telecommunications development based on Message Sequence Charts and SDL. Atmosphere briefing, ESPRIT project 2565, November 1991.

[GRG93] P. Graubmann, E. Rudolph, and J. Grabowski. Towards a Petri Net based semantics definition for Message Sequence Charts. In Færgemand and Sarma [FS93], pages 179–190.

[GSDH97] J. Grabowski, R. Scheurer, Z.R. Dai, and D. Hogrefe. Applying SaMsTaG to the B-ISDN protocol SSCOP – technical description and TTCN testsuite. Technical Report A-97-01, Medical University of Lübeck, Institut für Mathematik/Informatik, January 1997.

[GW96] J. Grabowski and T. Walter. Quality-of-service testing – specifying functional QoS testing requirements by using Message Sequence Charts and TTCN. In *Proceedings of the 6th GI/ITG Technical Meeting on 'Formal Description Techniques for Distributed Systems'*, volume 20:9 of *Arbeitsberichte des Instituts für Mathematische Maschinen- und Datenverarbeitung (Mathematik)*. Institut für Informatik, IMMD VII, Friedrich-Alexander Universität Erlangen-Nürnberg, June 1996.

[GW98] J. Grabowski and T. Walter. Visualisation of TTCN test cases by MSCs. In Lahav et al. [LWFH98], pages 161–170.

[Hau00] Ø. Haugen. MSC-2000: Interaction diagrams for the new millennium. *Computer Networks and ISDN Systems*, 2000.

[Hau01] Ø. Haugen. From MSC-2000 to UML 2.0 – the future of sequence diagrams. In SDL01 [SDL01]. To appear.

[Heg95] R. Hegi. Visualisiering von Testfällen. Master's thesis, University of Berne, Institute for Informatics and Applied Mathematics, 1995.

[Hey98] S. Heymer. A non-interleaving semantics for MSC. In Lahav et al. [LWFH98], pages 281–290.

[Hey00] S. Heymer. A semantics for MSC based on Petri net components. In Graf et al. [GJL00], pages 262–275.

[HJ00] L. Hélouët and C. Jard. Conditions for synthesis from Message Sequence Charts. In S. Gnesi, I. Schieferdecker, and A. Rennoch, editors, *5th International ERCIM Workshop on Formal Methods for Industrial Critical Systems. Proceedings of FMICS'2000*, Berlin, April 2000.

[HL95] D. Hogrefe and S. Leue, editors. *Formal Description Techniques VII. Proceedings of the Seventh IFIP WG 6.1 International Conference on Formal Description Techniques FORTE'94*, Berne, October 1994 1995. Chapman & Hall.

[HL97] Ø. Haugen and Y. Lahav. MSC/SDL new features. In *Tutorials of the Eighth SDL Forum*, Evry, France, September 1997.

[Hol96] G.J. Holzmann. Formal methods for early fault detection. In *Proceedings of Formal Techniques for Real-Time and Fault Tolerant Systems*, number 1135 in Lecture Notes in Computer Science, pages 40–54, Uppsala, Sweden, September 1996. Springer Verlag.

[ISO88a] ISO/EC. ESTELLE – a formal description technique based on an extended state transition model. Internation Standard 9074, ISO/EC, Geneva, September 1988.

[ISO88b] ISO/EC. LOTOS – a formal description technique based on the temporal ordering of observational behaviour. International Standard 8807, ISO/EC, Geneva, September 1988.

[IT93] ITU-TS. Message Sequence Chart (MSC). Recommendation Z.120, ITU-TS, Geneva, September 1993.

[IT94] ITU-TS. Specification and Description Language (SDL). Recommendation Z.100, ITU-TS, Geneva, June 1994.

[IT95] ITU-TS. Algebraic semantics of Message Sequence Charts. Recommendation Z.120 Annex B, ITU-TS, Geneva, April 1995.

[IT96] ITU-TS. Message Sequence Chart (MSC). Recommendation Z.120, ITU-TS, Geneva, October 1996.

[IT98] ITU-TS. Algebraic semantic of MSC'96. Recommendation Z.120 Annex B, ITU-TS, Geneva, 1998.

[IT00] ITU-TS. Message Sequence Chart (MSC). Recommendation Z.120, ITU-TS, Geneva, 2000.

[JP01] B. Jonsson and G. Padilla. An execution semantics form MSC-2000. In SDL01 [SDL01]. To appear.

[KGSB99] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to statecharts. In Franz J. Rammig, editor, *Distributed and Parallel Embedded Systems*, pages 61–71. Kluwer Academic Publishers, 1999.

[KL98] J.-P. Katoen and L. Lambert. Pomsets for Message Sequence Charts. In Lahav et al. [LWFH98], pages 291–300.

[Klu99] O. Kluge. Mapping of time extended Message Sequence Chart specification to a Stochastic Petri Net to derive temporal properties. In *Proceedings of the High Performance Computing Conference 1999 (HPC'99)*, San Diego, CA, USA, April 1999.

[KMP77] D.E. Knuth, J.H. Morris, jr., and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(1):323–350, 1977.

[Kos97] P. Kosiuczenko. Time in message sequence charts: A formal approach. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *EuroPar'97: Parallel Processing*, number 1300 in Lecture Notes in Computer Science. Springer Verlag, 1997.

[KRBG98] F. Khendek, G. Robert, G. Butler, and P. Grogono. Implementability of Message Sequence Charts. In Lahav et al. [LWFH98], pages 171–179.

[Kri91] F. Kristoffersen. Message Sequence Chart and SDL specification consistency check. In Færgemand and Reed [FR91].

[KW91] J. Kroon and A. Wiles. A tutorial on TTCN. In B. Jonsson, J. Parrow, and B. Pehrson, editors, *Protocol Specification, Testing and Verification. Proceedings IFIP WG 6.1 Eleventh International Symposium*, pages 40–92, Stockholm, 1991. Elsevier Science Publishers/North Holland.

[Lam78] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978.

[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 Confrerence on Formal Description Techniques*, pages 301–316, Boston, 1994. Elsevier Science Publishers/North Holland.

[LL95] P.B. Ladkin and S. Leue. Four issues concerning the semantics of Message Flow Graphs. In Hogrefe and Leue [HL95], pages 355–369.

[LMR98] S. Leue, L. Mehrmann, and M. Rezai. Synthesizing software architecture descriptions from Message Sequence Chart specifications. In *Proceedings 13th IEEE International Conference on Automated Software Engineering*, pages 192–195. IEEE Computer Society, October 1998.

[Loi96] S. Loidl. Interpretation und Werkzeugunterstützung von Message Sequence Charts (MSC'96). Master's thesis, Institut für Informatik, Ludwig-Maximilians-Universität München, 1996.

[LP97] V. Levin and D. Peled. Verification of Message Sequence Charts via template matching. In *TAPSOFT (FASE)'97, Theory and Practice of Software Development*, number 1214 in Lecture Notes in Computer Science, pages 652–666, Lille, France, 1997. Springer Verlag.

[LWFH98] Y. Lahav, A. Wolisz, J. Fischer, and E. Holz, editors. *Proceedings of the 1st Workshop of the SDL Forum Society on SDL and MSC*, number 104 in Informatik-Berichte, Berlin, Germany, June 1998. Humboldt-Universität zu Berlin.

[Man99] N. Mansurov. Automatic synthesis of SDL from MSC in forward and reverse engineering. In D. Bosnacki, S. Mauw, and T. Willemse, editors, *Proceedings of the first international symposium on Visual Formal Methods VFM'99*, number 99-08 in Computing Science Reports, pages 44–64, Eindhoven, August 1999. Eindhoven University of Technology, Department of Computer Science.

[Mei95] F.A.C. Meijs. Message Sequence Chart enhancements. Technical Report RWB-506-ir-95071, Information and Software Technology, Philips Research, November 1995.

[Mei00] F.A.C. Meijs. Connecting Message Sequence Charts. In Graf et al. [GJL00], pages 61–75.

[MH00] P. Le Maigat and L. Hélouët. A (max,+) approach for time in Message Sequence Charts. In *5th Workshop of Discrete Event Systems (WODES2000)*, Ghent, Belgium, August 2000.

[Mid94] C.A. Middelburg. A simple language for expressing properties of telecommunication services and features. Publication 94-PU-356, PTT Research, 1994.

[MP00] A. Muscholl and D. Peled. Analyzing message sequence charts. In Graf et al. [GJL00], pages 3–17.

[MPS98] A. Muscholl, D. Peled, and Z. Su. Deciding properties for Message Sequence Charts. In M. Nirat, editor, *Foundations of Software Science and Computation Structures, Proceedings of FoSSaCS'98*, number 1378 in Lecture Notes in Computer Science, pages 226–242, Lisbon, 1998. Springer Verlag.

[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 CSN 94-23, Eindhoven University of Technology, Department of Computer Science, 1994.

[MR95] S. Mauw and M.A. Reniers. Thoughts on the meaning of conditions. Temporary Document TD 9016, Experts Meeting ITU-T SG 10, St. Petersburg, April 1995.

[MR96] S. Mauw and M.A. Reniers. Refinement in interworkings. In *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 Cavalli and Sarma [CS97], pages 291–306.

[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. Institut national des télécommunications.

[MR01] S. Mauw and M.A. Reniers. A process algebra for Interworkings. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 19, pages 1269–1327. Elsevier Science Publishers/North Holland, 2001.

[MRW00] S. Mauw, M.A. Reniers, and T.A.C. Willemse. Message Sequence Charts in the software engineering process. In S.K. Chang, editor, *Handbook of Software Engineering and Knowledge Engineering*. World Scientific, 2000.

[MS93] N. Meng-Siew. Reasoning with timing constraints in Message Sequence Charts. Master's thesis, University of Stirling, August 1993.

[MSHT97] 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*, Osaka, 1997. Chapman & Hall.

[MT96] A. Mitschele-Thiel. Methodology and tools for the development of high performance parallel systems with SDL/MSCs. In I. Jelly, I. Gorton, and P. Croll, editors, *Software Engineering for Parallel and Distributed Systems*. Chapman & Hall, 1996.

[Mus99] M. Mussa. Automatic generation of SDL specifications from MSC. Master's thesis, Concordia University, November 1999.

[MvWW93] S. Mauw, M. van Wijk, and T. Winter. A formal semantics of synchronous interworkings. In Færgemand and Sarma [FS93], pages 167–178.

[MZ99] N. Mansurov and D. Zhukov. Automatic synthesis of SDL models in Use Case methodology. In Dsoulli et al. [DvBL99].

[Nah91] R. Nahm. Consistency analysis of Message Sequence Charts and SDL-systems. In Færgemand and Reed [FR91], pages 262–271.

[Nah94] R. Nahm. *Conformance Testing Based on Formal Description Techniques and Message Sequence Charts.* PhD thesis, University of Berne, Institute for Informatics and Applied Mathematics, 1994.

[Pel98] D.A. Peled. A toolset for Message Sequence Charts. In A.J. Hu and M.Y. Vardi, editors, *Computer Aided Verification. 10th International Conference, CAV'98. Proceedings*, pages 532–536, Vancouver, Canada, June 1998. Springer Verlag.

[RAB96] B. Regnell, M. Andersson, and J. Bergstrand. A hierarchical Use Case model with graphical representation. In *Proceedings of the Second International Symposium on Engineering Computer-Based Systems*, pages 270–277, Friedrichshafen, Germany, March 1996. IEEE Computer Society.

[Ren95] M.A. Reniers. Syntax requirements of Message Sequence Charts. In Bræk and Sarma [BS95], pages 63–74.

[Ren96] M.A. Reniers. Static semantics of Message Sequence Charts. Technical Report CSR 96-19, Eindhoven University of Technology, Department of Computer Science, 1996.

[Ren99] M.A. Reniers. *Message Sequence Chart: Syntax and Semantics.* PhD thesis, Eindhoven University of Technology, June 1999.

[RGG96a] 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.

[RGG96b] E. Rudolph, P. Graubmann, and J. Grabowski. Tutorial on Message Sequence Charts (MSC'96). In *Tutorials of the First Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing and Verification (FORTE/PSTV'96)*, Kaiserslautern, Germany, October 1996.

[RGG99] E. Rudolph, J. Grabowski, and P. Graubmann. Towards a harmonization of UML-Sequence Diagrams and MSC. In Dsoulli et al. [DvBL99].

[RKG97] G. Robert, F. Khendek, and P. Grogono. Deriving an SDL specification with a given architecture from a set of MSCs. In Cavalli and Sarma [CS97], pages 197–212.

[RSG00a] E. Rudolph, I. Schieferdecker, and J. Grabowski. Development of an MSC/UML test format. In J. Grabowski and S. Heymer, editors, *FBT'2000:Formale Beschreibungstechniken für verteilte Systeme*, Aachen, June 2000. Shaker Verlag.

[RSG00b] E. Rudolph, I. Schieferdecker, and J. Grabowski. HyperMSC – a graphical representation of TTCN. In Graf et al. [GJL00], pages 76–91.

[Rud95] E. Rudolph. MSC roadmaps. towards a synthesized solution. Temporary Document TD 90137, Experts Meeting ITU-T SG 10, Geneva, 1995.

[Rüf94] C. Rüfenacht. Extending MSCs with data information in order to specify test cases. Master's thesis, University of Berne, Institute for Informatics and Applied Mathematics, February 1994. In German.

[RV94] F.J. Redmill and A.R. Valdar. *SPC digital telephone exchanges, revised edition*. Number 21 in IEE Telecommunication series. Peter Peregrinus Ltd., 1994.

[Sch95] C. Schaffer. MSC/RT: A real-time extension of Message Sequence Charts (MSCs). Technical Report 129-95, Johannes Kepler University, Department of Systems Theory and Information Engineering, Linz, 1995.

[SD97] S. Somé and R. Dssouli. Using a logical approach for specification generation from Message Sequence Charts. Publication départementale 1064, Département IRO, Université de Montréal, April 1997.

[SDL01] *SDL'01: Meeting UML*, Lecture Notes in Computer Science, Copenhagen, June 2001. Springer Verlag. To appear.

[SDV95] S. Somé, R. Dssouli, and J. Vaucher. From scenarios to timed automata: Building specifications from user requirements. In *Proceedings 2nd Asia Pacific Software Engineering Conference*. IEEE Computer Society, December 1995.

[SEG+98] M. Schmitt, A. Ek, J. Grabowski, D. Hogrefe, and B. Koch. Autolink – Putting SDL-based test generation into practice. In A. Petrenko and N. Yevtuschenko, editors, *Testing of Communicating Systems*, volume 11, Tomsk, Russia, June 1998. Kluwer Academic Publishers.

[SG01] I. Schieferdecker and J. Grabowski. TTCN-3 and its MSC-based graphical presentation format. In *SDL'01 Tutorials*, Copenhagen, June 2001. To appear.

[Sil98] P.S.M. Silva. Extended Message Sequence Charts with time-interval semantics. In L. Khatib and R. Morris, editors, *Proceedings Fifth International Workshop on Temporal Representation and Reasoning*, pages 37–44, Sanibel Island, FL, USA, May 1998. IEEE Computer Society.

[SMC+96] B. Steffen, T. Margaria, A. Classen, V. Braun, and M. Reitenspiess. An environment for the creation of intelligent network services. In *Intelligent Networks: IN/AIN Technologies, Operations, Services, and Applications – A Comprehensive Report*, pages 287–300. IEC – International Engineering Consortium, 1996.

[SRM97] I. Schieferdecker, A. Rennoch, and O. Mertens. Timed MSCs – an extension to MSC'96. In A. Wolisz, I. Schieferdecker, and A. Rennoch, editors, *Formale Beschreibungstechniken für verteilte Systeme, 7. GI/ITG-Fachgespräch*, Berlin, June 1997.

[SRS89] R. Saracco, R. Reed, and J.R.W. Smith. *Telecommunications Systems Engineering Using SDL.* Elsevier Science Publishers/North Holland, 1989.

[Ste90] D. Steedman. *Abstract Syntax Notation One (ASN.1): The Tutorial and Reference.* Technology Appraisals Ltd., 1990.

[Tel95] Telelogic AB. SDT. In G. von Bochmann, R. Dssouli, and O. Rafiq, editors, *Participant's Proceedings of the 8th International Conference on Formal Description Techniques FORTE'95, List of tools for demonstrations*, page 455, 1995.

[Til91] P.A.J. Tilanus. A formalization of Message Sequence Charts. In Færgemand and Reed [FR91], pages 273–288.

[Ver96] Verilog. *ObjectGEODE MSC Editor – User's GUIDE.* Verilog SA, 1996.

[VGMF00] H. Vranken, T. Garcia Garcia, S. Mauw, and L. Feijs. IC validation using Message Sequence Charts. In *Proceedings 26th Euromicro Conference, Digital Systems Design DSD'2000*, pages 122–127, Maastricht, the Netherlands, September 2000. IEEE Computer Society.

[VT00] R.G. de Vries and J. Tretmans. On-the-fly conformance testing using SPIN. *Software Tools for Technology Transfer*, 2(4):382–393, March 2000.

[VWK95] G. Vermeer, M. Witteman, and J. Kroon. A framework for testing telecommunication services. In A. Cavalli and S. Budkowski, editors, *Proceedings of IWPTS '95*, pages 129–140, Evry, 1995.

[Wat95] B.W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms.* PhD thesis, Eindhoven University of Technology, Department of Computer Science, September 1995.

# Samenvatting

Wie een goed computersysteem wil bouwen, zal moeten beschrijven wat het doet of zou moeten doen. Natuurlijk talen, zoals Nederlands of Engels, zijn hiervoor niet zeer geschikt, omdat ze te weinig exact zijn. In plaats daarvan worden hiervoor zogenaamde 'formele talen' gebruikt. Dit zijn beschrijvingsmethoden met een exacte, wiskundige betekenis. Dit promotie-onderzoek heeft zich op zulke talen gericht, en meer specifiek op twee aspecten: hun betekenis ('semantiek'), en hun toepassingen voor het analyseren en testen van systemen.

In hoofdstuk 2 houden we ons bezig met testafleiding. Als een systeem eenmaal beschreven en gebouwd is, is het nuttig om te kunnen vaststellen of het systeem ook daadwerkelijk aan de beschrijving voldoet. Een manier om daar een uitspraak over te kunnen doen, is door het systeem te testen. In dit hoofdstuk worden testen afgeleid met behulp van 'model checking'. Model checking is een methode die ontworpen is om eigenschappen van een systeem af te leiden uit haar beschrijving: er kan worden vastgesteld of een systeem een toestand kan bereiken met bepaalde eigenschappen. Zo ja, dan wordt bovendien aangegeven op welke manier. In de methodologie die in dit hoofdstuk wordt beschreven, wordt deze laatste eigenschap van model checking gebruikt: door situaties te onderzoeken waarvan al bekend is dat ze bereikbaar zijn, wordt een pad naar deze situaties gevonden, dat vervolgens als test gebruikt kan worden.

Hoofdstuk 3 houdt zich ook bezig met testen. Het is gebaseerd op een praktijkprobleem: voor het testen van telefooncentrales wordt een groot aantal gesprekken gesimuleerd, die vervolgens met de hand worden gecontroleerd. Als een stap in de automatisering van dit proces, hebben we een taal ($LOGAN$) ontworpen waarmee op eenvoudige, geautomatiseerde wijze, afzonderlijke gesprekken uit de lijst met signalen die door de centrale zijn gegaan, kunnen worden gefilterd.

De volgende hoofdstukken hebben allen betrekking op de taal Message Sequence Charts (MSC). MSC wordt gebruikt voor de beschrijving van de communicatie binnen of tussen systemen. Het bestaat uit afbeeldingen zoals figuur 9.1. In dit plaatje zijn de verticale lijnen ($a$ en $b$) (delen van) computersystemen, terwijl de pijlen ('vraag' en 'antwoord') communicaties tussen die systemen zijn. De tijd loopt van boven naar onder in dit diagram. In dit plaatje stuurt dus eerst $a$ 'vraag' naar $b$, waarna $b$ 'antwoord' naar $a$ stuurt. In hoofdstuk 4 staat een meer uitgebreide uitleg, waarin ook diverse uitbreidingen van de taal beschreven zijn.

In hoofdstuk 5 bekijken we, hoe uit een beschrijving van een systeem in MSC kan worden afgeleid wat er nodig is om het communicatiegedrag mogelijk te maken. Soms

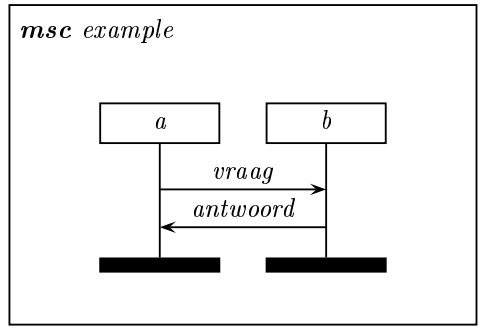



Figure 9.1: Een MSC

is het mogelijk alle berichten te ontvangen in de volgorde waarin ze zijn verzonden, in andere gevallen is dit onmogelijk. Er zullen dan meerdere buffers gebruikt moeten worden om de berichten in op te slaan, waar er in het eerste geval slechts een nodig is. In dit hoofdstuk valt te lezen hoe uit de MSC-beschrijving van een systeem valt af te leiden wat voor dat systeem het geval is.

Een van de recente uitbreidingen van MSC is de mogelijkheid om data toe te voegen. In hoofdstuk 6 wordt beschreven hoe dit is gedaan, waarom het zo is gedaan, en welke problemen daarbij overwonnen moesten worden. Het is gedeeltelijk gebaseerd op discussies binnen de standardisatiecommissie voor MSC. Daarnaast geeft het hoofdstuk ook aan hoe de officiële betekenis (semantiek) van MSC kan worden uitgebreid om ook dit aspect toe te voegen.

Hoofdstukken 7 en 8 behandelen twee mogelijke toekomstige uitbreidingen van MSC. In hoofdstuk 7 wordt een methode beschreven om een enkel bericht in MSC te gebruiken om een volledig communicatieprotocol te beschrijven, terwijl in hoofdstuk 8 de zogenaamde disruptie en interruptie behandeld worden. Met behulp hiervan kunnen systemen beschreven worden waarbij verschillende gedragingen elkaar kunnen onderbreken of stoppen. In deze hoofdstukken worden de mogelijkheden van deze uitbreidingen beschreven, hun problemen, en de semantiek die hen meegegeven zou kunnen worden.

# Titles in the IPA Dissertation Series

**J.O. Blanco**. *The State Operator in Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1996-1

**A.M. Geerling**. *Transformational Development of Data-Parallel Algorithms.* Faculty of Mathematics and Computer Science, KUN. 1996-2

**P.M. Achten**. *Interactive Functional Programs: Models, Methods, and Implementation.* Faculty of Mathematics and Computer Science, KUN. 1996-3

**M.G.A. Verhoeven**. *Parallel Local Search.* Faculty of Mathematics and Computing Science, TUE. 1996-4

**M.H.G.K. Kesseler**. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory.* Faculty of Mathematics and Computer Science, KUN. 1996-5

**D. Alstein**. *Distributed Algorithms for Hard Real-Time Systems.* Faculty of Mathematics and Computing Science, TUE. 1996-6

**J.H. Hoepman**. *Communication, Synchronization, and Fault-Tolerance.* Faculty of Mathematics and Computer Science, UvA. 1996-7

**H. Doornbos**. *Reductivity Arguments and Program Construction.* Faculty of Mathematics and Computing Science, TUE. 1996-8

**D. Turi**. *Functorial Operational Semantics and its Denotational Dual.* Faculty of Mathematics and Computer Science, VUA. 1996-9

**A.M.G. Peeters**. *Single-Rail Handshake Circuits.* Faculty of Mathematics and Computing Science, TUE. 1996-10

**N.W.A. Arends**. *A Systems Engineering Specification Formalism.* Faculty of Mechanical Engineering, TUE. 1996-11

**P. Severi de Santiago**. *Normalisation in Lambda Calculus and its Relation to Type Inference.* Faculty of Mathematics and Computing Science, TUE. 1996-12

**D.R. Dams**. *Abstract Interpretation and Partition Refinement for Model Checking.* Faculty of Mathematics and Computing Science, TUE. 1996-13

**M.M. Bonsangue**. *Topological Dualities in Semantics.* Faculty of Mathematics and Computer Science, VUA. 1996-14

**B.L.E. de Fluiter**. *Algorithms for Graphs of Small Treewidth.* Faculty of Mathematics and Computer Science, UU. 1997-01

**W.T.M. Kars**. *Process-algebraic Transformations in Context.* Faculty of Computer Science, UT. 1997-02

**P.F. Hoogendijk**. *A Generic Theory of Data Types.* Faculty of Mathematics and Computing Science, TUE. 1997-03

**T.D.L. Laan**. *The Evolution of Type Theory in Logic and Mathematics.* Faculty of Mathematics and Computing Science, TUE. 1997-04

**C.J. Bloo**. *Preservation of Termination for Explicit Substitution.* Faculty of Mathematics and Computing Science, TUE. 1997-05

**J.J. Vereijken**. *Discrete-Time Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1997-06

**F.A.M. van den Beuken**. *A Functional Approach to Syntax and Typing.* Faculty of Mathematics and Informatics, KUN. 1997-07

**A.W. Heerink**. *Ins and Outs in Refusal Testing.* Faculty of Computer Science, UT. 1998-01

**G. Naumoski and W. Alberts**. *A Discrete-Event Simulator for Systems Engineering.* Faculty of Mechanical Engineering, TUE. 1998-02

**J. Verriet**. *Scheduling with Communication for Multiprocessor Computation.* Faculty of Mathematics and Computer Science, UU. 1998-03

**J.S.H. van Gageldonk**. *An Asynchronous Low-Power 80C51 Microcontroller.* Faculty of Mathematics and Computing Science, TUE. 1998-04

**A.A. Basten**. *In Terms of Nets: System Design with Petri Nets and Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1998-05

**E. Voermans**. *Inductive Datatypes with Laws and Subtyping – A Relational Model.* Faculty of Mathematics and Computing Science, TUE. 1999-01

**H. ter Doest**. *Towards Probabilistic Unification-based Parsing.* Faculty of Computer Science, UT. 1999-02

**J.P.L. Segers**. *Algorithms for the Simulation of Surface Processes.* Faculty of Mathematics and Computing Science, TUE. 1999-03

**C.H.M. van Kemenade**. *Recombinative Evolutionary Search.* Faculty of Mathematics and Natural Sciences, Univ. Leiden. 1999-04

**E.I. Barakova**. *Learning Reliability: a Study on Indecisiveness in Sample Selection.* Faculty of Mathematics and Natural Sciences, RUG. 1999-05

**M.P. Bodlaender**. *Schedulere Optimization in Real-Time Distributed Databases.* Faculty of Mathematics and Computing Science, TUE. 1999-06

**M.A. Reniers**. *Message Sequence Chart: Syntax and Semantics.* Faculty of Mathematics and Computing Science, TUE. 1999-07

**J.P. Warners**. *Nonlinear approaches to satisfiability problems.* Faculty of Mathematics and Computing Science, TUE. 1999-08

**J.M.T. Romijn**. *Analysing Industrial Protocols with Formal Methods.* Faculty of Computer Science, UT. 1999-09

**P.R. D'Argenio**. *Algebras and Automata for Timed and Stochastic Systems.* Faculty of Computer Science, UT. 1999-10

**G. Fábián**. *A Language and Simulator for Hybrid Systems.* Faculty of Mechanical Engineering, TUE. 1999-11

**J. Zwanenburg**. *Object-Oriented Concepts and Proof Rules.* Faculty of Mathematics and Computing Science, TUE. 1999-12

**R.S. Venema**. *Aspects of an Integrated Neural Prediction System.* Faculty of Mathematics and Natural Sciences, RUG. 1999-13

**J. Saraiva**. *A Purely Functional Implementation of Attribute Grammars.* Faculty of Mathematics and Computer Science, UU. 1999-14

**R. Schiefer**. *Viper, A Visualisation Tool for Parallel Progam Construction.* Faculty of Mathematics and Computing Science, TUE. 1999-15

**K.M.M. de Leeuw**. *Cryptology and Statecraft in the Dutch Republic.* Faculty of Mathematics and Computer Science, UvA. 2000-01

**T.E.J. Vos**. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms.* Faculty of Mathematics and Computer Science, UU. 2000-02

**W. Mallon**. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes.* Faculty of Mathematics and Natural Sciences, RUG. 2000-03

**W.O.D. Griffioen**. *Studies in Computer Aided Verification of Protocols.* Faculty of Science, KUN. 2000-04

**P.H.F.M. Verhoeven**. *The Design of the MathSpad Editor.* Faculty of Mathematics and Computing Science, TUE. 2000-05

**J. Fey**. *Design of a Fruit Juice Blending and Packaging Plant.* Faculty of Mechanical Engineering, TUE. 2000-06

**M. Franssen**. *Cocktail: A Tool for Deriving Correct Programs.* Faculty of Mathematics and Computing Science, TUE. 2000-07

**P.A. Olivier**. *A Framework for Debugging Heterogeneous Applications.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

**E. Saaman**. *Another Formal Specification Language.* Faculty of Mathematics and Natural Sciences, RUG. 2000-10

**M. Jelasity**. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure.* Faculty of Mathematics and Natural Sciences, UL. 2001-01

**R. Ahn**. *Agents, Objects and Events a computational approach to knowledge, observation and communication.* Faculty of Mathematics and Computing Science, TU/e. 2001-02

**M. Huisman**. *Reasoning about Java programs in higher order logic using PVS and Isabelle.* Faculty of Science, KUN. 2001-03

**I.M.M.J. Reymen**. *Improving Design Processes through Structured Reflection.* Faculty of Mathematics and Computing Science, TU/e. 2001-04

**S.C.C. Blom**. *Term Graph Rewriting: syntax and semantics.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

**R. van Liere**. *Studies in Interactive Visualization.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

**A.G. Engels**. *Languages for Analysis and Testing of Event Sequences.* Faculty of Mathematics and Computing Science, TU/e. 2001-07