Validation and Refinement of Timed MSC Specifications

Tong Zheng

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfilment of the Requirements
for the Degree of Doctor of Philosophy at
Concordia University
Montreal, Quebec, Canada

February 2004

© Tong Zheng, 2004

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services

Acquisisitons et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-90407-5
Our file Notre référence
ISBN: 0-612-90407-5

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this dissertation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de ce manuscrit.

While these forms may be included in the document page count, their removal does not represent any loss of content from the dissertation.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: **Tong Zheng**

Entitled: **Validation and Refinement of Timed MSC Specifications**

and submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY (Electrical and Computer Engineering)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. S. Mudur — Chair

Dr. S. Graf — External Examiner

Dr. G. Butler — External to Program

Dr. M. Mehmet-Ali — Examiner

Dr. S. Tahar — Examiner

Dr. F. Khendek — Thesis Supervisor

Approved by

Dr. M. Mehmet-Ali, Graduate Program Director

APR 0 2 2004 2004

Dr. N. Esmail, Dean
Faculty of Engineering & Computer Science

ABSTRACT

Validation and Refinement of Timed MSC Specifications

Tong Zheng, Ph.D.
Concordia University, 2004

This thesis addresses the validation and the refinement of MSC (Message Sequence Charts) specifications at the requirement and the design phases in a software development process. The validation is necessary to ensure that an MSC specification does not contain semantic errors. The refinement provides a systematic approach to develop MSC specifications. The focus of this thesis is on timed MSC specifications, which may contain absolute and relative time constraints for specifying quantified timing requirements.

To provide a foundation for analysis of MSC specifications, we develop a formal semantics for timed MSCs based on labeled partially ordered sets (lposets). We equip an lposet with two timing functions for expressing absolute and relative time constraints. The semantics of an MSC is represented by a set of lposets. The set can be obtained compositionally from the semantics of constructs contained in the MSC.

Time constraints in an MSC specification may lead to inconsistencies. In such a case, the specification contains semantic errors. We study the time consistency of MSC specifications. We define the time consistency and develop sufficient and necessary conditions for the consistency. According to these conditions, algorithms are designed for checking the consistency. We also study the time consistency of high level MSCs and identify a subset of high level MSCs such that their consistency can be checked efficiently.

We propose a refinement approach where we refine not only behaviors, but also time constraints specified in an MSC specification. Refining time constraints makes constraints on a system stronger, and assumptions on the environment weaker. We define refinement relations and develop algorithms to check the satisfaction of these relations. To reduce the

complexity in the case of high level MSCs, we constrain the refinement rules.

At last, as an outcome of our investigation of timed MSCs, we propose a new time construct as an extension of timed MSC in order to specify more timing requirements.

Most of the algorithms presented in this thesis have been implemented and integrated to our set of tools MSC2SDL.

ACKNOWLEDGEMENTS

I wish to express my sincerest gratitude to my thesis supervisor, Dr. Ferhat Khendek, for his guidance and assistance. He has supported me academically and financially through the years. I am most impressed by his insight and ability to develop new ideas. I am especially grateful for his patience and encouragement throughout this research. This thesis could not be possible without the substantial time and effort he has devoted.

I would especially like to thank Dr. Susanne Graf from VERIMAG for agreeing to serve as external examiner of my thesis. Many thanks go to Dr. Gregory Butler, Dr. Mustafa K. Mehmet Ali and Dr. Sofiene Tahar for serving on my thesis committee.

I am grateful to all those persons who helped me in my studies. Dr. Loïc Hélouët from IRISA gave his comments promptly on the semantics of MSC. Dr. Ludovic Apvrille proofread this thesis carefully and provided helpful suggestions to improve the thesis.

I would like to thank Lixin Wang, who has implemented several algorithms in this thesis. I would also like to thank all of my friends and fellow graduate students in the telesoft research group, Umer Waqar, Stephan Bourduas, Xiaojun Zhang, Yang Liu, and Cedric Besse visiting from France. I had a lot of fun with them during the years I have spent at Concordia.

At last, my special thanks go to my wife Na. I owe a great deal to her for her endless love and understanding under any situations. I am indebted to her for giving me comfort and taking care of our baby so that I can continue my thesis. I dedicate this thesis to her and our daughter Ashley.

Table of Contents

List of Figures xii

1 Introduction 1

1.1 Formal Methods for Developing Reactive Systems 1

1.2 Specification and Development Using MSC and SDL 3

1.3 Motivation 5

1.3.1 Validation of MSC Specifications 6

1.3.2 Refinement of MSC Specifications 6

1.4 Contributions of Thesis 7

1.5 Thesis Organization 8

2 MSC Language 11

2.1 Introduction 11

2.2 MSC-96 12

2.2.1 Basic Behavioral Constructs 13

2.2.2 Other Behavioral Constructs 15

2.2.3 Compositional Constructs 19

2.3 New Constructs in MSC-2000 24

2.3.1 Data 24

2.3.2 Time 25

2.3.3 Guard Condition 27

2.3.4 Control Flow 27

2.3.5 Object Orientation in MSC Documents 28

2.4 MSC Dialects 29

2.4.1 LSC 29

2.4.2 PMSC 31

2.4.3 YAMS 31

2.4.4 HyperMSC 32

2.4.5 UML Sequence Diagram 33

2.4.6 Interworkings 34

2.5 Summary 35

3 Semantics of MSC **37**

3.1 Introduction 37

3.2 A Partial Order Model for MSC 39

3.2.1 Timed lposets 41

3.2.2 Operations on lposets 44

3.3 Partial Order Semantics for Timed MSC 47

3.3.1 Orderable Events 48

3.3.2 MSCs Containing only Orderable Events 49

3.3.3 Coregion 50

3.3.4 MSC Reference 51

3.3.5 Inline Expressions and MSC Expressions 52

3.3.6 High Level MSC (HMSC) 53

3.4 Related Work 56

3.5 Conclusion 58

4 Time Consistency of MSC Specifications **60**

4.1 Introduction 60

4.2 Time Consistency of bMSCs 63

4.3 Time Consistency of HMSCs 65

4.4 Algorithms for Checking Time Consistency in General 71

4.5 Algorithms for Checking Time Consistency in Specific Cases 75

4.6 Related work 84

4.7 Conclusion 85

5 **Refining MSC Specifications** **87**

5.1 Introduction 87

5.2 Refining bMSCs 88

5.2.1 Refinement Approach 88

5.2.2 Refinement Relation of bMSCs 93

5.2.3 Checking Conformance between bMSCs 97

5.3 Refining HMSCs 99

5.3.1 Refinement Relation for HMSCs 99

5.3.2 Checking Conformance of HMSCs 101

5.4 Example: Refining a Basic Call Specification 104

5.5 Related Works 107

5.6 Conclusion 108

6 **An Extension to MSC** **111**

6.1 Introduction 111

6.2 Instance Delay 112

6.3 Semantics of Instance Delay 114

6.4 An Application 116

6.5 Related Works 119

6.6 Conclusion 119

7 Conclusion **120**

7.1 Summary 120

7.2 Future Work 122

7.2.1 Syntactic and Semantic Extensions to MSC 122

7.2.2 Implementability of MSC Specifications 123

7.2.3 Translation to SDL Specifications 123

7.2.4 Tool Support and Case Studies 123

Bibliography **125**

A Simplified Textual Syntax of MSC **137**

B Proofs **139**

B.1 Proposition 3.1 139

B.1.1 Identity 139

B.1.2 Commutative property 140

B.1.3 Associative property 140

B.1.4 Distributive property 142

B.2 Theorem 4.1 142

B.3 Theorem 4.2 144

B.4 Proposition 4.1 145

B.5 Theorem 4.3 146

B.6 Proposition 4.2 148

B.7 Proposition 5.1 148

B.8 Theorem 5.1 149

List of Figures

1.1 An MSC-centered development process 5

2.1 A simple MSC describing a beverage machine 13

2.2 An MSC with action 15

2.3 MSC with timers 16

2.4 An MSC with condition 17

2.5 MSC with coregion and general ordering 17

2.6 Instances creation and termination (a), and Incomplete messages (b) 18

2.7 Alt inline expression(a), and Loop inline expression (b) 20

2.8 MSC references and gates 21

2.9 An HMSC 23

2.10 An MSC with data 25

2.11 An MSC with time constraint and measurement 26

2.12 An MSC with guard condition 27

2.13 Asynchronous method call (a), and Synchronous method call (b) 28

2.14 MSC documents 29

3.1 Graphical and textual notations of MSC 38

3.2 An MSC with coregion 50

3.3 An MSC with a reference 51

3.4 An MSC with inline expressions 53

3.5 An HMSC and referred MSCs 54

3.6 Transforming an HMSC to an expression 55

4.1 Time inconsistency in bMSCs 61

4.2 Time inconsistency in an HMSC 61

4.3 A directed constraint graph and its distance graph 64

4.4 Iteration and absolute time constraints 66

4.5 An inconsistent HMSC with a consistent simple path 68

4.6 Strong consistency of an HMSC 70

4.7 bMSCs in an inconsistent path PMN 75

4.8 Weak and strong consistency for time-disjoint HMSCs 80

4.9 Relation between lower bounds 81

5.1 Refining a door controller 90

5.2 Refining a door controller further 92

5.3 Refining time constraints 94

5.4 Comparing time constraints 94

5.5 Event Order Tables 99

5.6 An ATM specification before refinement 100

5.7 An ATM specification after refinement 100

5.8 Basic Call 105

5.9 Refining bMSC dial 106

6.1 Time constraints on events and MSC 113

6.2 Usage of instance delays 114

6.3 Specification of measurement process using instance delays 117

Chapter 1

Introduction

1.1 Formal Methods for Developing Reactive Systems

Methodologies for developing software systems depend on the nature of the systems under consideration. Generally, a software system can be classified as a reactive system or a transformational system [73]. A transformational system can be modeled as a function that transforms inputs into desired outputs. Once the outputs are produced, the system stops. Examples of transformational systems are compilers, and calculators. We are interested in reactive systems. Unlike transformational systems, a reactive system maintains interactions with its environment. It responds to inputs from the environment, which may arrive continuously and in unexpected sequences. Examples of reactive systems include telephone system, traffic-light controller, and air traffic control system.

In general, reactive systems have to satisfy certain timing requirements. For example, a telephone system has to send a dial tone within a certain time delay after the handset is picked up. So most reactive systems can be seen as real-time systems. Another characteristic of reactive systems is concurrency. Different inputs may occur simultaneously and without any fixed order. To handle these inputs, a reactive system often consists of several processes.

Moreover, these processes may be distributed over several processors and physical locations.

Because of these characteristics, developing a reactive system requires a lot of efforts, and it is often hard to assure the correctness of the system. To reduce faults in the system, formal methods can be used in the development process, such as in requirement gathering and analysis, design, or testing. Formal methods are mathematically based techniques [107]. When a formal method is applied, the system is described at an abstract level using a formal language first. The result is a formal specification. The specification is then analyzed to discover contradictions or errors in requirements or design. It can also be used to generate test cases for testing implementations.

As discussed in [91], usually two types of specifications are needed for reactive systems: requirement specifications and system specifications. These specifications are artifacts from different stages of a development process. In the requirement phase, the requirement specification is used to describe what are the required behaviors of the system, or what are the properties the system should exhibit. In the design phase, the system specification is used to describe an abstract implementation of the system. A system specification usually consists of the high level architecture of the system, the interface between the components of the system, and the behavior of each component.

Given these two specifications, we need to consider their correctness first. Each specification needs to be validated to ensure that there are not inconsistencies in requirements, or errors (such as deadlocks, non-reachable states) in design. Next, we need to ensure that the system specification conforms to the requirement specification. One way is to verify the system specification against the requirement specification using verification techniques. In many cases, these specifications are written in different formal languages. For example, temporal logics can be used for requirement specifications, and automata or program-like languages can be used for system specifications. To handle real-time requirements, real-time temporal logics can be used, such as Metric Temporal Logic [58, 60, 74], Explicit Clock Temporal Logic [37, 74, 89]. The automata can be timed automata [4]. Model

checking techniques are used for the verification [16].

On the other hand, both requirement and system specifications can be written in the same language. A requirement specification can be transformed into a system specification by a stepwise refinement, where the conformance can be assured by design. This refinement is based on the equivalence relation between the specifications, or sometimes it can be relaxed to the inclusion relation. For example, in [92], the temporal logic is used for both requirement and system specifications. Equivalences between specifications can be proven formally.

1.2 Specification and Development Using MSC and SDL

The Message Sequence Charts (MSC) [52] and the Specification and Description Language (SDL) [49] are two specification languages developed, standardized and maintained by ITU-T (International Telecommunication Union, Telecommunication Standardization Sector). They have been used in telecommunication software engineering for years. MSC concentrates on interactions between components in a system, while the internal behaviors of the components are not of concern. For example, a basic MSC contains only instances that represent components, and messages that describe asynchronous communications between these components. Timing requirements can be specified using timers, or time constraints as introduced in MSC-2000 [52]. Moreover, compositional constructs, such as high-level MSC (HMSC), are provided to combine several MSCs[1] as one specification. An introduction to different constructs in MSC is given in Chapter 2.

On the other hand, SDL describes the architecture of a system and behaviors of each component in the system. A system can be decomposed into a set of blocks. These blocks are connected by channels to carry signals. A block consists of several processes. Contrarily to MSC, SDL describes the internal behaviors of processes explicitly. Each process is modeled

[1] We use MSC to refer to the language, MSCs to refer to concrete charts.

as an extended finite state machine.

Since MSC and SDL supplement each other, they are often used together. In fact, MSC can be used for requirement specification. Using MSCs, requirements are captured as use cases or desired scenarios describing interactions between a system and its environment [3, 56]. On another hand, a system specification can be described using SDL and MSC. An SDL specification describes the architecture of the system and the abstract implementation of each process. An MSC specification describes the dynamic interface between processes.

If the SDL specification has been developed manually, it has to be verified against the MSC requirement specification formally or informally. This is the approach in the SDL-oriented Object Modeling Technique (SOMT) [21], for instance. An alternative approach is to derive SDL specifications automatically from MSC specifications. In this approach, the correctness of the SDL specifications is guaranteed by translation. Tools for the translation have been developed, such as the MSC2SDL by the telesoft research group at Concordia [81, 97].

Since the SDL specification can be obtained automatically, MSC specifications become the key model at the requirement and the design phases. In different phases, MSC specifications may serve different purposes and exhibit different levels of details. To ensure the conformance, we can, for instance, verify the MSC system specification against the MSC requirement specification. Or, we can enrich the MSC requirement specification by adding details to components in a system to form a system specification. Thus, a development process could begin with an abstract MSC specification, which is refined step by step towards a more detailed MSC specification. After that, given a target architecture, an SDL specification can be generated from the detailed MSC specification. During the testing phase, the MSC specification can also be used as test purposes for generating test cases automatically. In fact, MSC specifications take a central role in the development process, and their correctness is crucial.

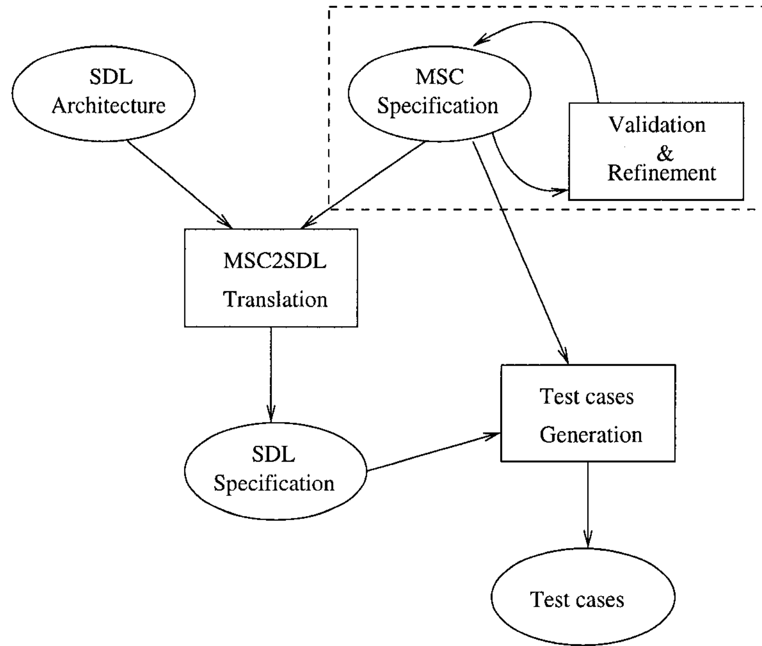


Figure 1.1: An MSC-centered development process

1.3 Motivation

Motivated by the usage of MSC specifications in a software development process, this thesis addresses mainly two issues that are essential for MSC specifications. The first one is the validation of MSC specifications. To be useful in a development process, MSC specifications must not contain semantic errors. They have to be validated. The other issue is the refinement of MSC specifications. The refinement provides an approach to develop MSC specifications systematically. In the refinement, the properties of MSC specifications should be preserved. Moreover, if an MSC specification is valid, then its refinement should remain valid. Together with translation to SDL and generation of test cases, we can obtain an MSC centered development methodology as shown in Figure 1.1.

1.3.1 Validation of MSC Specifications

Because of its graphical form, an MSC specification is intuitive and easy to understand. However, the intuitive understanding may be different from the formal semantics of MSCs. Furthermore, an MSC specification may contain semantic errors or logical inconsistencies. So its correctness has to be checked. Several criteria for the correctness, such as the absence of race conditions, process divergences, confluence and inferences [5, 7, 11, 41, 83], have been established and techniques for checking these criteria have been developed. However, these criteria are mainly for MSC specifications without timing requirements.

In this thesis, we investigate the validation of timed MSC specifications. The introduction of time constraints in MSC-2000 raises the time consistency issue of the MSC specifications. Only a few works have considered this issue partially [7, 10, 25]. Moreover, in the MSC-2000 standard, time concepts are introduced and time constraints are defined differently from the existing work. One of our goals is to develop techniques and algorithms to validate the time consistency of MSC-2000 specifications.

1.3.2 Refinement of MSC Specifications

The refinements of un-timed MSC and similar notations have been proposed in [56, 61, 75, 103]. However, the usage of timed MSC in the development of systems with real-time requirements has not been explored yet. Another goal of this thesis is to develop a practical and sound design approach to refine timed MSC specifications.

A refinement of specifications could be based on equivalence relation or inclusion relation. Many proposals for refining real-time systems, such as [72, 80], use equivalence relations between specifications. Time constraints are kept unchanged during the refinements. As argued in [70], the equivalence refinement may not be very useful for real-time systems. In our opinion, refinement in the design phase should give more flexibility to designers. We

develop an approach that refines both behaviors and time constraints based on inclusion relations.

1.4 Contributions of Thesis

Since MSC is a formal language, it is essential that we define its formal semantics, which provides a foundation for analyzing MSC specifications. Unlike existing proposals for the semantics of MSC, we treat events in an MSC as basic units. This allows us to define the semantics compositionally for both basic MSCs and MSCs with compositional constructs. Moreover, we follow the time concepts as defined in MSC-2000. This semantics is applicable to both timed MSC and un-timed MSC.

Based on this semantics, we validate timed MSC specifications. Specifically, we consider the consistency of time constraints. We develop sufficient and necessary conditions for an HMSC to be consistent. According to these conditions, we design algorithms to check the consistency. These algorithms have been implemented [105]. A special class of HMSCs is identified such that their consistencies can be checked more efficiently.

After its validation, an MSC can be developed further by adding more concrete information. We propose a refinement approach where not only behaviors of the system, but also timing requirements can be refined. Timing requirements are changed following rules that take into account the relationship between the system and its environment.

In our refinement approach, an MSC specification is changed according to a few rules. The rules do not assure the conformance automatically. We verify the conformance relation between specifications. Our intention is to give designers more choices during the refinement. However, the challenge is to check the conformance relation efficiently. To check the relation, we need to compare MSCs, which is generally complex in computation, and in some cases it is undecidable. For instance, the pattern matching problem is NP-complete as shown in [85].

The problem of deciding if two HMSCs specify the same event orders is undecidable [8, 84, 85]. We have to trade off between the restriction of refinement rules and the efficiency of conformance checking.

During the study of timed MSC, we have found that some timing requirements cannot be specified using the constructs in the current MSC standard. We propose an extension to timed MSCs to enhance its expressiveness. This extension is being considered in the standardization of the language.

The contributions of this thesis are summarized as follows.

- We develop a semantics for timed MSC based on partial orders.
- We define the consistency of time constraints in MSCs and develop algorithms to check the consistency.
- We propose a refinement approach for timed MSCs, and develop algorithms to check the conformance between MSC specifications.
- We propose an extension to timed MSC to enhance its expressiveness.

1.5 Thesis Organization

We introduce the MSC language in Chapter 2. We present MSC constructs in MSC-96, such as messages, timers, coregions, inline expressions and HMSCs. Then new constructs in MSC-2000 are introduced. These new constructs relate to timing, data and object-oriented concepts. Among them, time constraints are our main concern. We also survey some dialects of the standard MSC language.

After introducing the syntax of MSC in Chapter 2, we define its semantics in Chapter 3. We develop a denotational semantics for timed MSC based on timed *labeled partially ordered*

sets (lposets). A timed MSC is mapped to a set of lposets. We define the semantics for most of constructs in MSC-96. However, for the new constructs in MSC-2000, we focus on time constraints. We can apply this semantics for un-timed MSCs also, which can be seen as special cases of timed MSCs. The semantics forms a foundation for the validation and refinement of MSCs in the following chapters. This chapter has been published in [111].

When writing an MSC specification, we need to validate it before it can be used further in the process. We investigate under what conditions an MSC is consistent in Chapter 4. We define the consistency of basic MSCs and apply the theory of temporal constraint networks to check it. For HMSCs, we define two kinds of consistencies, namely strong consistency and weak consistency. We determine the sufficient and necessary conditions for these consistencies, and algorithms for checking the consistencies according to these conditions. At last, we characterize specific HMSCs for which the strong consistency can be checked efficiently. This chapter has been published in [109].

Given a valid MSC specification, we can refine it to a more concrete one, but always conform to the original specification. In Chapter 5, we propose a refinement approach to develop timed MSC specifications. We refine not only the behaviors expressed by MSCs, but also the time constraints in MSCs. A refinement procedure consists of decomposing processes, adding messages, and changing or adding time constraints. The resulting MSC specification is required to keep the properties of the original specification, such as event orders. Moreover, if the original specification is time consistent, the resulting MSC specification should remain time consistent also. We define refinement relations between MSCs formally, and develop algorithms for the verification of these relations. This chapter has been published in [110].

During our investigation of MSCs, we have came across the need to extend the language. For example, when using MSC to specify a repeated scenario, we may need to specify the periodicity, or the time interval between two repetitions. The current MSC standard has limitations for expressing this. Therefore, in Chapter 6, we propose a new construct, called

instance delay, as an extension to timed MSC. Using this construct, we can specify the periodicity of instances. We define formally the semantics of instance delay and illustrate its need and usage in the specification of the Radio Resource Control (RRC) protocol [1]. This chapter has been published in [108].

In Chapter 7, we summarize the contributions in this thesis, and discuss future work.

Chapter 2

MSC Language

2.1 Introduction

The MSC language appeared initially as a complement of the Specification and Description Language (SDL) [49]. SDL describes the behaviors of a process in a system as an extended finite state machine. However, the interactions between processes cannot be observed explicitly in SDL. So sequence charts were used as auxiliary diagrams for describing the interactions [32]. In 1990, the CCITT (the predecessor of ITU-T) decided to standardize the sequence charts as a new language, called Message Sequence Charts. In 1993, the first recommendation for MSC, recommendation Z.120 [50], was approved.

The first recommendation of MSC contained graphical and textual syntax definitions, but a formal semantics was missing. A formal semantics is crucial for a specification language in order to describe a system unambiguously. So in the second recommendation (MSC-96) [51], a formal semantics based on process algebra theory was defined, which benefited from the development of the Interworking language [76, 95]. Moreover, composition constructs were introduced into MSC-96, which were also influenced greatly by the Interworking. These constructs enhanced the expressiveness of MSC greatly, and made it possible to write a

complete specification for a large, complex system using MSC. Actually MSC-96 had become a language used independently to SDL.

Since then, MSC has been used more and more widely for software development. Most often it is used to describe scenarios and use cases of a system [3]. Properties of a system can be specified using MSC instead of temporal logic [18, 61, 82, 85]. It can also be used in verification [8, 20], simulation [39, 104], and testing [29, 30, 86]. CASE tools are also available, such as Tau [101], uBet [7], Mesa [12] and MSC2SDL [81, 97].

Meanwhile, the MSC language continues to evolve and further extensions have been proposed to handle data [24, 26], timing and performance [7, 10, 25, 69] in MSC. Data and timing concepts were officially introduced in MSC-2000 [52], which made it possible to address systems with quantified timing constraints.

In this chapter, we give an introduction to the MSC notations. In Section 2.2, we introduce constructs in MSC-96. For each construct, we provide its syntax and explain its semantics informally. Then in Section 2.3, we present new constructs in MSC-2000. We focus on time constraints. In Section 2.4, we introduce several extensions and similar notations to the standard MSC. We summarize the chapter in Section 2.5.

2.2 MSC-96

MSC provides a lot of constructs for specifying distributed systems. These constructs can be classified as behavioral constructs for describing the behaviors of the system, or compositional constructs for composing several MSCs as one complete specification. The MSC standard defines both graphical and textual syntax for each construct. For the purpose of better communications between developers and customers, the graphical syntax is often used in the specification. However, CASE tools are often built on the textual syntax. We focus on the graphical syntax for the purpose of presentation.

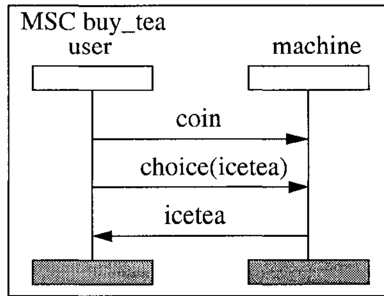


Figure 2.1: A simple MSC describing a beverage machine

The MSC-96 standard defines a formal semantics based on process algebra. We introduce the semantics informally when each construct is presented.

2.2.1 Basic Behavioral Constructs

We first show a simple MSC *buy_tea* in Figure 2.1. The MSC describes the interactions between a user and a beverage machine. The user inserts coins and selects the type of beverages. Then the machine delivers the beverage to the customer.

The MSC in Figure 2.1 consists of three kinds of constructs: instance (the vertical lines), message (the horizontal lines), and frame (the box around the MSC). These are the most basic and often used constructs. An MSC containing only these constructs is called a basic MSC (bMSC). We introduce these constructs below in turn.

Instance

An MSC contains a finite set of instances, each of which usually represents a component in a system. An instance is described by a vertical axis with an instance head symbol and an instance end symbol. The instance head and the instance end do not imply the creation and termination of the instance. They only define a segment of progress for an instance. In Figure 2.1, two instances are specified, which are *user* and *machine*.

Along an instance axis, events such as sending or receiving messages are ordered from the top to the bottom. However, the delay between two consecutive events is not specified by the instance axis. For example, in Figure 2.1, the *user* can make a choice any time after inserting coins. Furthermore, each instance runs independently with each other. The *user* can make a choice no matter whether the *coin* is received by the *machine* or not.

Message

The most essential behavior described by an MSC is the message exchange between instances. A message is represented by an arrow which is directed from the sending instance to the receiving instance. A message between instances is associated with two events: the sending (the output of the message) and the receiving (the input of the message). The MSC in Figure 2.1 contains three messages. The message *choice* carries a parameter *icetea*.

In MSC, message passing is asynchronous. The sending event has to precede the receiving event. They cannot occur at the same time. Moreover, the MSC standard does not restrict the method of transmitting messages. Basically it is implied that every message is transmitted in its own channel. So it is possible that the order of receiving messages and the order of sending the messages are not the same. In [23], several other communication models for transmitting messages are considered.

MSC Frame

The MSC frame represents the environment of the system specified in the MSC. Interactions between the system and the environment are described by messages sent to the environment, and messages received from the environment. Message events in the environment are not ordered.

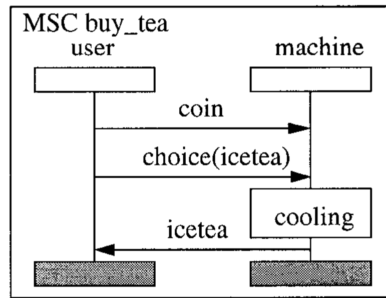


Figure 2.2: An MSC with action

2.2.2 Other Behavioral Constructs

We introduce briefly other behavioral constructs that are not contained in a bMSC in the following.

Action

The internal behavior of an instance can be described in an action box. In semantics, an action is considered as a local event in the instance. In Figure 2.2, the instance *machine* performs an action *cooling* before offering the *icetea.*

Timer

MSC-96 uses timers to specify timing requirements. A timer is a clock that can be set for a duration and started, be stopped before its expiration, or expire by itself. These actions are associated with three timer events: set, reset and time-out. These events are considered to occur in the instance to which a timer is attached. They are ordered with other events in the same instance.

The usage of timers is shown in Figure 2.3. In the MSC *return_coin*, setting a timer and the expiration of the timer are used together. The *machine* sets a timer after getting the *coin.* If the *user* does not choose a beverage in the duration set by the timer, then the

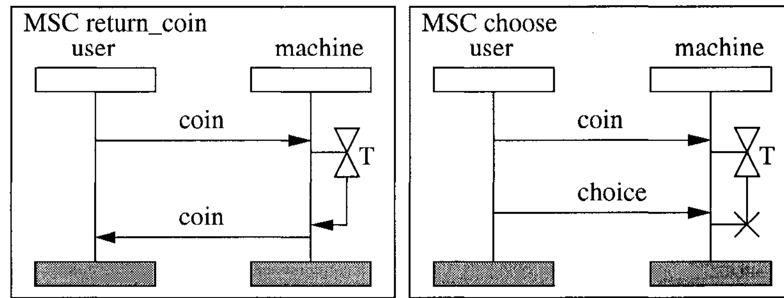


Figure 2.3: MSC with timers

coin is returned to the *user*. On the other hand, in the MSC *choose*, setting a timer and resetting the timer are used together. If the *user* chooses a beverage in the duration, the timer is stopped.

Since MSC-96 does not support data languages, a duration associated with a timer set event is just symbolic. It does not have a meaning in semantics. This is changed in MSC-2000 because the data concept is introduced into it. Moreover, quantitative time constraints can be specified in MSC-2000. We introduce them in Section 2.3.

Condition

Usually a condition is used to describe a certain status of one or more instances in an MSC. A condition could be global, non-global, or local. A global condition covers all the instances in an MSC; a non-global condition covers some of instances only; and a local condition covers exactly one instance. The local conditions in Figure 2.4 show the status of the *machine* before and after the *user* buys a beverage.

In the semantics of MSC-96, conditions are considered as comments. In practice, conditions can be used as a glue to connect MSCs, or as guards to restrict the execution of MSCs. MSC-2000 defines formally the usage of conditions as guards. We introduce guard conditions in Section 2.3.

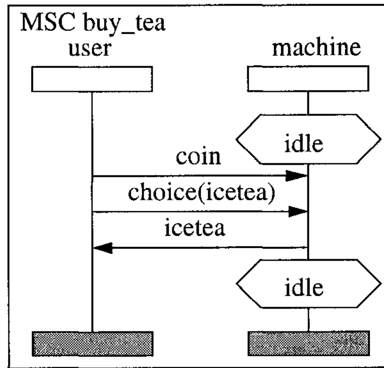
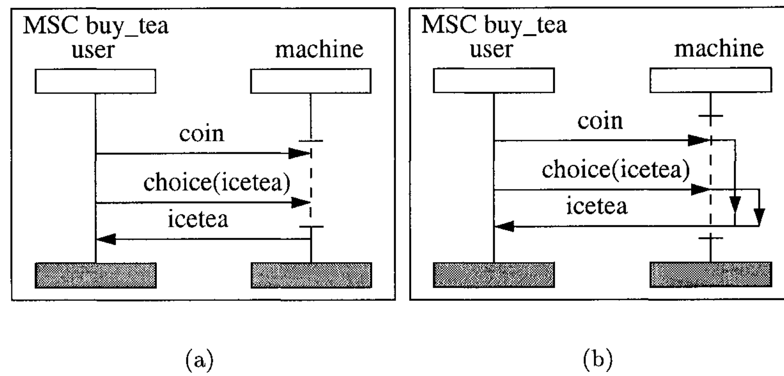


Figure 2.4: An MSC with condition



(a) (b)

Figure 2.5: MSC with coregion and general ordering

Coregion

As mentioned before, all the events in an instance are ordered from the top to the bottom. However, sometimes it is necessary to specify unordered events in an instance. The coregion construct allows for specifying a region in an instance in which events are not ordered. A coregion is represented by a dashed line. For example, in Figure 2.5(a), within the coregion in *machine*, either receiving *coin* or receiving *choice* could occur first. This is possible if these two (*coin* and *choice*) are transmitted separately in their own channels.

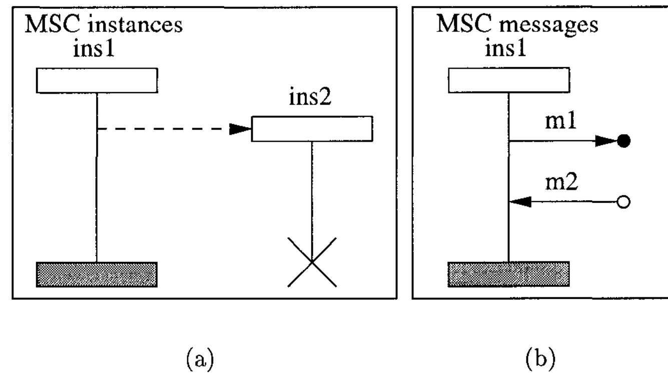


Figure 2.6: Instances creation and termination (a), and Incomplete messages (b)

General Ordering

Besides the ordering of events in an instance and the ordering of events by message exchanges, the general ordering provides another way to order events in the same instance or in different instances. However, the general ordering is mainly used in a coregion. It provides additional orders between some events in the coregion. A general ordering is represented by a solid line with an arrowhead in the middle. For example, in Figure 2.5(b), all the events of the instance *machine* are within a coregion. Two general orderings are used to specify that both receiving the *coin* and receiving the *choice* occur before offering the *icetea*.

Instance Creation and Termination

The creation of an instance by another instance, and the termination of an instance are denoted by a dashed arrow and a cross respectively. For example, in Figure 2.6(a), two instances *ins1* and *ins2* exist in the MSC. The instance *ins2* is created by *ins1*, and then terminates itself.

Incomplete Message

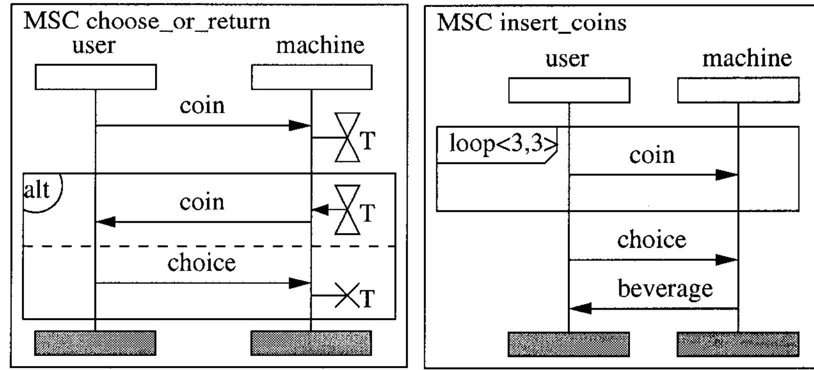
MSC allows for incomplete messages, which are associated with only a sending instance, or a receiving instance. A message is a lost message if there is no receiving instance. It is a found message if there is no sending instance. In Figure 2.6(b), the message *m1* is a lost message. The message *m2* is a found message.

2.2.3 Compositional Constructs

Inline Expression

Inline expressions are used to compose MSCs, whose names are omitted, inside an MSC. Usually these MSCs do not contain many constructs so that they can be described inline. The operators used in inline expressions are *alt*, *par*, *opt*, *exc*, and *loop*.

- An *alt* inline expression composes two or more MSCs alternatively. The choice between these MSCs is not made until it is not avoidable. It is called delayed choice.
- A *par* inline expression composes two or more MSCs in parallel. It means that events in different MSCs are interleaved.
- An *opt* inline expression contains one MSC that may or may not be executed. It can be seen as an *alt* inline expression with one empty MSC.
- An *exc* inline expression contains one MSC that describes an exception scenario. It means either the MSC or the part following the expression is executed. It can also be seen as a variant of the *alt* inline expression.
- A *loop* inline expression executes an MSC repeatedly. The number of times that the MSC is executed is defined by the loop boundary. The loop boundary contains a lower bound and an upper bound, which indicate the minimal and the maximal number of



(a) (b)

Figure 2.7: Alt inline expression(a), and Loop inline expression (b)

times that the MSC is executed. The bounds have to be natural numbers, and the upper bound could be infinite.

Inline expressions can be nested where the inner inline expression is contained in an MSC that is contained in the outer inline expression.

An *alt* inline expression and a *loop* inline expression are shown in Figure 2.7. The MSC in Figure 2.7(a) specifies an alternation between the two MSCs in Figure 2.3. Either a choice is made before the timer expires, or the coin is returned to the user. The MSC in Figure 2.7(b) specifies that the *user* inserts three coins to buy a beverage.

MSC Reference and Reference Expression

An MSC can refer to another MSCs as its component by means of MSC references. For example, the MSC *buy* in Figure 2.8 refers to two other MSCs *prepare* and *offer*. By using references, some parts of an MSC are hidden to make it more succinct and readable.

A reference can also refer to an MSC expression. An MSC expression consists of the names of several MSCs as operands. The operators include all those used in inline expres-

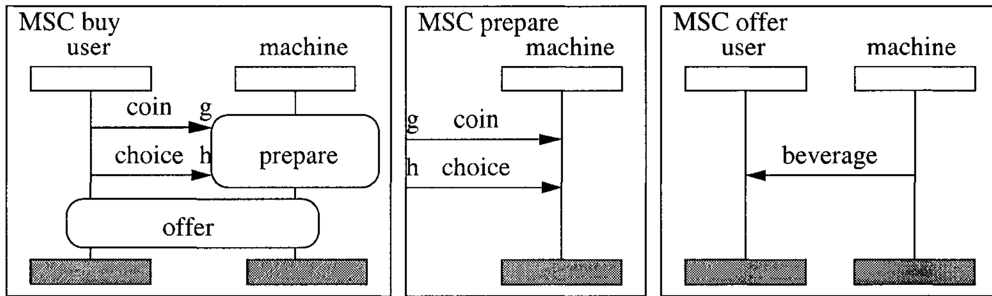


Figure 2.8: MSC references and gates

sions and a sequential composition operator *seq*. The *seq* operator composes two MSCs sequentially. Specifically, two MSCs are connected instance by instance. Thus, the events in the second MSC could occur before all the events in the first MSC have occurred. For instance, if the second MSC contains an instance that is not contained in the first MSC, then the events in the instance may occur before all the events in the first MSC. This kind of sequential composition is called weak sequencing.

Gate

A gate is a part of the environment of an MSC. The MSC standard defines two kinds of gates, message gates and order gates. A message gate represents an entry point in the environment through which an instance sends or receives a message. An order gate represents a point through which an event is ordered with another event. Gates can be used together with MSC reference expressions, and inline expressions. For example, in Figure 2.8, message gates g and h in the MSCs *buy* and *prepare* are used to connect the messages outside the MSC *prepare* with the messages inside. The use of gates facilitates the decomposition of large specifications.

High level MSC(HMSC)

An HMSC describes the composition of MSCs in a directed graph. It is a road map that presents the execution flow of MSCs. An HMSC consists of nodes and directed edges. A node can be of the following types.

- Start node. There is only one start node in an HMSC. It is the start of an HMSC. Each node has to be reachable from the start node.
- End node. An end node represents an exit point of an HMSC. An HMSC may not contain any end nodes.
- MSC reference node. An MSC reference node refers to an MSC, an MSC reference expression, or another HMSC. It is not allowed to refer the HMSC itself directly or indirectly.
- Condition node. A condition node describes the status of instances before entering an MSC reference node.
- Parallel frame. A parallel frame contains several HMSCs that are executed in parallel. The HMSCs in a parallel frame cannot be the HMSC containing the parallel frame.
- Connection point. Connection points are used to make the layout of a large HMSC more readable.

Except MSC reference nodes and parallel frames, other nodes in HMSCs are only syntax symbols and do not have semantics defined in the MSC standard. The semantics of an MSC reference node is the semantics of MSCs referred by the node. The semantics of an parallel frame is the semantics of the parallel composition (*par*) of HMSCs in the frame.

Nodes in an HMSC are connected by directed edges. Along paths formed by directed edges, MSCs referred by the MSC reference nodes are composed sequentially (corresponding

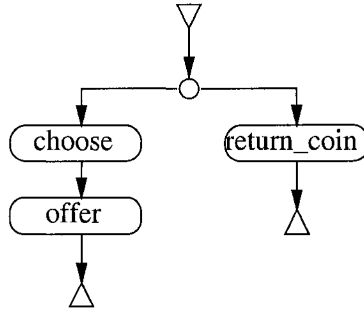


Figure 2.9: An HMSC

to the *seq* operator). The branches among the paths represent the alternative composition (corresponding to the *alt* operator). Cycles represent the repeated behaviors (corresponding to the *loop* operator).

An HMSC is shown in Figure 2.9. The symbols $\bigtriangledown$, $\bigtriangleup$ and $\bigcirc$ in the figure represent a start node, an end node and a connection point, respectively. The referred MSCs *return_coin* and *choose* are shown at Figure 2.3 (page 16), and the MSC *offer* is shown at Figure 2.8. The HMSC describes that the machine offers beverage if a choice is made, otherwise it returns the coin.

Instance Decomposition

The instance decomposition is a mechanism for describing a system at different abstract levels. By instance decomposition, an instance at a higher level of abstraction can be represented by an MSC containing a set of instances at a lower level of abstraction. The message inputs and outputs at the decomposed instance have to be preserved in the MSC. Moreover, orders between these inputs and outputs are also preserved. The instance decomposition actually refines the instance.

2.3 New Constructs in MSC-2000

The latest version of MSC, MSC-2000 [52], adds new concepts to enhance the expressiveness of MSC. These new concepts concern time, data, object orientation on MSC documents, control flow, and guard conditions [38]. We discuss them in the following subsections.

2.3.1 Data

One enhancement in MSC-2000 is the support of data. Instead of defining a specific data language for MSC, the actual data language is considered as a parameter [24, 26]. So the user can choose which language is used, for example, C or ASN.1 [48]. All data are given under the form of character strings in MSC.

Data can be used in different places, such as in action boxes, in loop boundaries, in time constraints, or in messages as parameters. The MSC language assumes the existence of data types in three places:

- Boolean valued expressions in guarding conditions,
- Natural number expressions in loop boundaries, and
- Time expressions in time constraints.

There are two kinds of data, static data or dynamic data. Static data appear as formal parameters of an MSC and cannot be modified. Dynamic data refer to MSC variables that can be assigned values multiple times. These variables are local variables attaching to individual instances.

Assigning a value to a variable in MSC is realized by bindings. A data expression is bound to a data pattern. A data pattern consists of either a variable, or a wildcard that represents some anonymous variable. A wildcard is more useful in data expressions, in

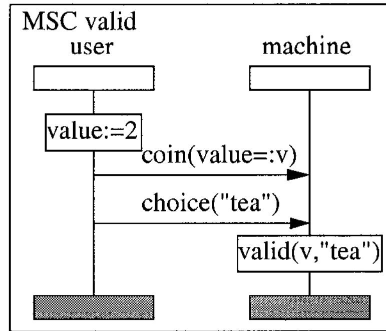


Figure 2.10: An MSC with data

which it represents any possible value. For instance, if a wildcard "*" is bound to an integer variable, then "*" can be any integer number. Because of wildcards, bindings are more general than assignments in programming languages.

An MSC with data is shown in Figure 2.10. In the action box in *user*, the number (2) is bound to the variable *value*. By the passing of message *coin(value=:v)*, the value of the variable *value* is bound to the variable v belonging to the instance *machine*. Then the variable v has the value 2. Notice that there are two kinds of bindings in MSC, left binding (":=") and right binding ("=:"). In a left binding, a data pattern appears in the left side, while in a right binding, it appears in the right side.

2.3.2 Time

To describe real-time systems with quantified timing requirements, MSC-2000 has introduced time concepts. Time in MSC can be discrete or continuous. It is assumed that time begins from an origin and progresses forever without stagnating. Thus the time domain must be a total order with a least element. It must be closed under an addition operation. The time progress is equal for all instances in an MSC. All events are instantaneous. They do not consume time.

MSC defines two kinds of time constraints to describe timing requirements: absolute

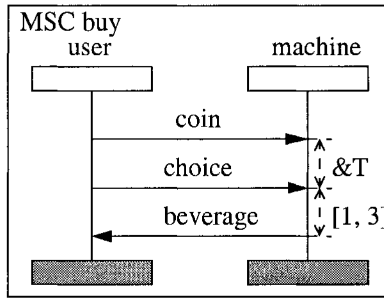


Figure 2.11: An MSC with time constraint and measurement

time constraints and relative time constraints. To distinguish absolute and relative time constraints, a "@" sign is used before absolute time constraints. An absolute time constraint specifies the occurrence time of an event, while a relative time constraint specifies the delay between any two events. A relative time constraint can also be associated with MSC references and inline expressions. In such a case, it constrains the first and the last events in the reference and the inline expression. It is worth to note that the first and the last events are determined dynamically.

A time constraint is an interval with minimal and maximal bounds in a time domain. The delay between two events (in the case of relative time constraints) or the occurrence time of an event (in the case of absolute time constraints) has to be one of the value within the interval. The minimal and maximal bounds can be equal. In that case, the time constraint becomes a time point instead of a time interval.

An example of using absolute time constraints is in the specification of the Wake up call service, where one has to specify the time at which the call to the service subscriber has to be made. This time could be an exact time or an interval. Relative time constraints can be used, for instance, to specify the delay between the receiving of a request and the sending of a response in a process. For example, in Figure 2.11, the delay between receiving *choice* and offering *beverage* is constrained by a relative time constraint [1, 3], which means the delay is at least 1 and at most 3. If the delay is exactly 3, we can specify it as [3].

A delay between two events or the occurrence time of an event can be observed using

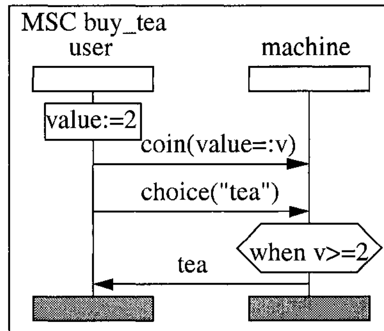


Figure 2.12: An MSC with guard condition

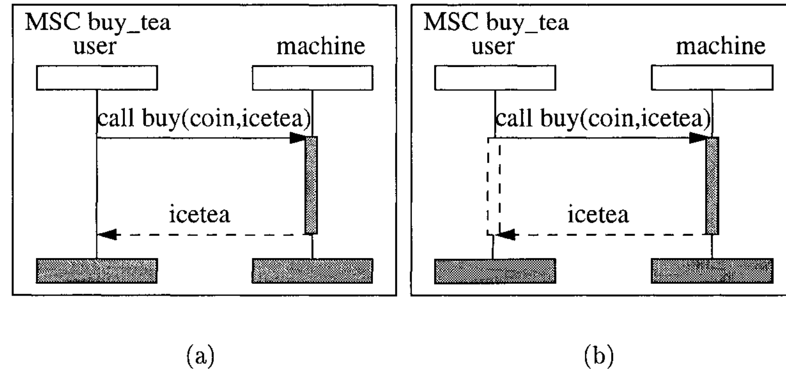
the measurement construct. For each measurement, a time variable has to be declared. To distinguish a time constraint with a measurement, a "&" sign is used before a measurement. In Figure 2.11, the delay between receiving *coin* and *choice* by *machine* is measured and the value of the delay is stored in a variable T.

2.3.3 Guard Condition

Before MSC-2000, conditions in MSC are treated as labels only. It describes the current state of one or several instances. With the data concept introduced in MSC, conditions can be used as guards also. To distinguish guard conditions and normal conditions, a key word *when* is used in guard conditions. Depending on its evaluation, a guard condition restricts the behavior of an MSC by allowing the execution of events in the scope of the condition. For example, in Figure 2.12, the *machine* offers *tea* only when the value of the *coin* is equal to or more than 2 (which could be the price of the *tea*).

2.3.4 Control Flow

Besides message exchanges, MSC-2000 also uses method calls and method replies to describe control flows. A method is a unit of behavior inside an instance. A method call may be asynchronous or synchronous. After an asynchronous call, the calling instance may continue



(a) (b)

Figure 2.13: Asynchronous method call (a), and Synchronous method call (b)

without waiting the reply. On the other hand, the calling instance have to suspend until the reply after a synchronous call. In the suspension region, no events could occur for the calling instance. Message calls and replies are similar to sequence diagrams in the UML.

The MSCs in Figure 2.13 show the scenario of buying tea using method calls and replies. In Figure 2.13(a), the *user* can perform other actions before the method reply *icetea*, while in Figure 2.13(a), the *user* has to wait for the reply.

2.3.5 Object Orientation in MSC Documents

An MSC document defines a system, which may contain several instances, and contains a collection of MSCs describing the system. It consists of a defining part and a utility part. The defining part actually defines MSCs while the utility part contains MSCs reused by the MSCs in the defining part. The system or instances in an MSC document may be inherited or overridden in another MSC document.

For example, we define a beverage machine system (*BMSystem*) in an MSC document in Figure 2.14(a). The system contains two instances: *user* and *machine*. The MSCs *buy*, *choose* and *return_coin* are defined in the defining part. The utility part contains MSCs *offer* and *prepare* that are used by the MSC *buy*. In Figure 2.14(b), we define another

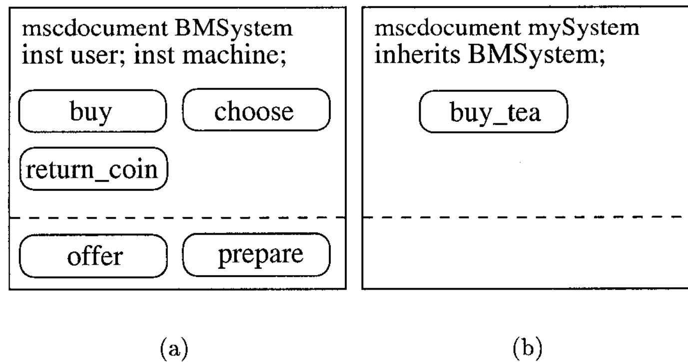


Figure 2.14: MSC documents

MSC document *mySystem*, which inherits the *BMSystem*. All the instances and MSCs in *BMSystem* are inherited by *mySystem*. In addition, it defines an MSC *buy_tea*.

2.4 MSC Dialects

Since MSC-96 has been released, many extensions have been considered to enhance the expressiveness of the MSC language. We discuss briefly some of them here, which are Live Sequence Charts (LSC), Performance Message Sequence Charts (PMSC), YAMS, and HyperMSC. While the LSC extends the MSC-96 to specify liveness, PMSC includes performance aspects into the language. YAMS uses a syntax identical to the MSC-96, but defines a different semantics. HyperMSC improves the presentation of HMSCs without affecting the semantics. We also discuss Sequence diagrams in UML, which is often used in object oriented analysis and design, and Interworkings.

2.4.1 LSC

Live Sequence Charts (LSC) [18] extends the standard MSC language to distinguish possible and necessary behaviors. The reason of making this distinction is that the interpretation of scenarios described by MSCs evolves in different stages of the software development

process. At the specification stage, MSCs are used to capture sample scenarios that *can* happen, while at the implementation stage, the system *must* adhere to the scenarios.

The distinction between possible and necessary behaviors can be made at different constructs as follows.

- An entire chart. One LSC is either universal (represented by solid-line frame around it) or existential (represented by dashed-line frame around it). The behavior in a universal LSC must be exhibited by any system run. On the other hand, an existential LSC is only required to be satisfied by at least one system run.

- Messages. For a message, a solid arrow describes mandatory behavior, that is, the sent message must be received. A dashed arrow describes provisional behavior, which means the receiving of the message is not guaranteed. Moreover, differently with the standard MSC, a message passing could be asynchronous or synchronous.

- Conditions. A condition could be mandatory or provisional also. A mandatory condition must be evaluated to true by a system run, otherwise the run halts and the system is not implemented correctly. However, if a provisional condition is evaluated to false, then the run skips the behaviors following the condition in the enclosing chart. This is similar to the guard condition in MSC-2000.

- Locations, which refer to the progress over time in instances. A solid vertical line segment indicates that the instance must run beyond it. In contrast, a dashed vertical line segment indicates that instance need not move beyond.

LSC does not consider HMSCs. It uses sub-charts for composing scenarios. However, constructs for branching and iteration are not detailed further in [18].

2.4.2 PMSC

Performance Message Sequence Charts (PMSC) [25] is an extension of MSC-96 to support performance engineering. In the design and implementation of a system, the following performance issues have to be considered:

- performance requirements, such as the response time of the system, the system throughput;
- implementation alternatives;
- resource requirements, which specifies the amount of physical resources needed depending on different implementation alternatives;
- available resources, such as the number of CPUs, channels, and the capacity of memory; and
- implementation decisions for selecting an implementation alternative.

PMSC uses the MSC-96 language for the functional specification, and describes the performance requirements, resource requirements and available resources in the comments. Compared to MSC-2000, PMSC addresses more performance issues. However, techniques for performance evaluation and validation are not considered in [25].

2.4.3 YAMS

YAMS [61] (Yet Another MSC Semantics) uses a syntax almost identical to the MSC-96, but with different semantics. First of all, a message in YAMS is not associated with a sending and a receiving event as defined in MSC. Instead, it is associated with a time point at which the message is sent through a channel. It is assumed that a global clock exists in a system.

Secondly, in MSC-96, the weak sequential semantics is used for the composition of two MSCs, where events in the second MSC can be executed before all the events in the first MSC have been executed. The sequential composition in YAMS is strong sequencing and does not allow for the interleaving of events in sequential MSCs.

YAMS also does not model delayed choice. In MSC-96, a choice between two alternative MSCs (or MSC expressions) are made as late as the moment when two alternatives differ. In YAMS, the choice is made from the beginning of an execution.

Moreover, YAMS defines more constructs than MSC-96. They are described in the follows.

- Guarded condition. It is similar to the guard condition in MSC-2000.
- Unbounded finite repetition. It represents any finite number of repetition. In MSC-2000, this can be described using a wildcard in the loop boundary [38].
- Join operator. It combines two MSCs as one where common messages are merged.
- Trigger composition operator. It associates two MSCs, where the execution of the first MSC triggers the execution of the second MSC.
- Preemption. It allows a message to interrupt an MSC and switch to another MSC immediately.

2.4.4 HyperMSC

HyperMSC [31] presents HMSCs in a hypertext-like manner. A HyperMSC allows an MSC reference or a path consisting of several MSC references in an HMSC to be expanded in detail within the HyperMSC, while the other MSC references remain hidden. In this way, an HMSC can be presented in different views. It is assumed that tools are available to support HyperMSCs for viewing different parts of an HMSC and switching between different views.

HyperMSC does not affect the semantics of HMSC, just changing the graphical layout of an HMSC for a compact and transparent representation.

In [31], a new construct called MSC connector is also introduced for abstracting communications between MSC references or inline expressions. It generalizes the gate construct. Using MSC connectors in HyperMSC provides a mechanism that allows for folding and unfolding instances to MSC references. Meanwhile, messages associated with the instances can be abstracted as one MSC connector. An MSC connector can be expanded as a group of messages also. This facilitates the compositional specification of complex systems.

MSC connectors not only bundle messages, but also specify communication mechanisms between MSC references. A few of MSC connector types are introduced in [31], such as FIFO (first in first out) connectors, LIFO (last in first out) connectors, or unrestricted connectors. However, formal semantics of the MSC connectors are not presented in [31].

2.4.5 UML Sequence Diagram

UML sequence diagram (SD) [88] can be seen as an object oriented variant of MSC-96. SD and MSC are used for different applications. While MSCs describe distributed systems with asynchronous communications, SDs describe the flow of control passing through different objects in a program [35].

However, SD and MSC have many similar constructs. Actually, one predecessor of the SD is the Object Message Sequence Chart notation, which is a modification of the MSC. Meanwhile, some constructs in MSC-2000, such as method call and reply, come from SD. There are proposals to combine MSC and SD as one language with rich constructs and a formal semantics [98].

We introduce the constructs in SD in the follows.

- *Object or actor axes* are life lines of objects or actors. They are same as instance axes

in MSC.

- *Object activations* show the period during which the object is performing a procedure.
- *Object creation and destruction* are same as instance creation and termination in MSC.
- *Messages* describe method calls or replies between objects/actors. A message could be asynchronous or synchronous. It can be recursive also. The repetition of a message can be expressed by the iterative message. A message guarded by a condition is called conditional message. Several conditional messages can originate from one single point to represent branches. In MSC, this can be represented using alternative inline expressions and guard conditions.
- *Conditional branchings* describe alternatives between object activations.
- *Iteration boxes* and *loops with conditions to exit* enclose a part of SD indicating that it can occur multiple times. This corresponds to the loop inline expression in MSC.
- *Time constraints* specify the time of transmitting messages, or duration of a part of object activations. They are similar to the relative time constraints in MSC-2000.
- *pseudo code* supplements the graphical representation of SD.

SD does not provide composition constructs that are similar to inline expressions and HMSCs in MSC. Moreover, there is no referencing mechanism in SD. So, it is limited to describe small scenarios or specifications.

2.4.6 Interworkings

Interworkings are the predecessor of the MSC-96 language [51, 95]. It focuses completely on communications between system components and omits their internal behaviors. In syntax, it contains only axes for entities (corresponding to instances in MSC) and arrows for messages. Moreover, message exchanges in Interworkings are synchronous.

Interworkings can be composed together vertically or horizontally using the sequencing operator and the merge operator. However, there are no operators for alternative and loop in Interworkings.

- The sequencing operation concatenates one Interworking below another one, by linking common entities. If two Interworkings do not contain common entities, the sequencing operation actually results in a parallel composition.
- The merge operation identifies common entities and their communications in two Interworkings. The communication behaviors between common entities have to be exactly same so that the two Interworking can be merged. If two Interworkings do not contain common entities, the merge operation also results in a parallel composition.

A semantics based on process algebra is defined for Interworking in [75], which includes a refinement relation. Informally, refining an Interworking is to replace one entity in the Interworking with a set of entities. Then the internal behaviors of the entity are represented by the collective behaviors of the set of entities. However, the external behaviors of the entity have to be kept same. This refinement is similar to the decomposition of instances in MSC.

2.5 Summary

In this chapter, we introduced the MSC language and its several variants. We classified constructs in MSC-96 as behavioral constructs and compositional constructs. An MSC other than HMSC is often called a plain MSC. Moreover, we call an MSC as a basic MSC if it contains only instances, messages and MSC frame.

MSC-2000 includes all the constructs in MSC-96, and some new constructs concerning data, time, object orientation, control flow and guard condition. It is worth to note that

timers can also be used to specify some timing requirements in an instance. However, time constraints in MSC-2000 are more general because they can be used to specify timing requirements involving events in different instances. Time constraints are more convenient to use than timers. For instance, to specify a delay between two events, a timer has to be set immediately after the first event, then a time-out event has to be observed immediately before the second event. Using time constraints, the delay can be specified by a relative time constraint between these two events. Actually, time constraints in an instance can be seen as high level requirements, while timers can be seen as low level implementations of the requirements.

While the new concepts in MSC-2000 enhance its expressiveness further, they also make the language more complex. It is more difficult to define a formal semantics for the whole language, based on which CASE tools could be built to support its usage. In this thesis, we do not intend to address all the features of MSC-2000. We focus on its time aspect and define its semantics in Chapter 3.

The MSC standard also defines instance decompositions as a refinement mechanism. To make it more useful, we generalize it to both basic MSCs and HMSCs. We also consider the refinement of time constraints. We will discuss the refinement issues in Chapter 5.

Chapter 3

Semantics of MSC

3.1 Introduction

A formal semantics is essential in several aspects during the usage of MSC. First, although the MSC language provides graphical syntax that makes a specification easy to understand, people may have different intuitive interpretations of an MSC. A formal semantics defines the MSC language precisely and avoids potential ambiguities in the specification. It also forms a foundation for analyzing MSC specifications and building simulation and verification tools. Moreover, by developing a formal semantics, we can detect ambiguities, omissions and contradictions in the definition of the language itself [59].

In the MSC-96 standard, a process algebra approach has been used to define the formal semantics [95]. The MSC-2000 language contains more constructs as introduced in Chapter 2. These constructs make it more complex to define a formal semantics. As pointed out in [22], it is very difficult, if not impossible, to develop a semantics for the whole language. A more feasible way is to define the semantics concentrating on one aspect of the language. For example, the definition for the semantics of data in MSC-2000 has been proposed [22, 24, 26].

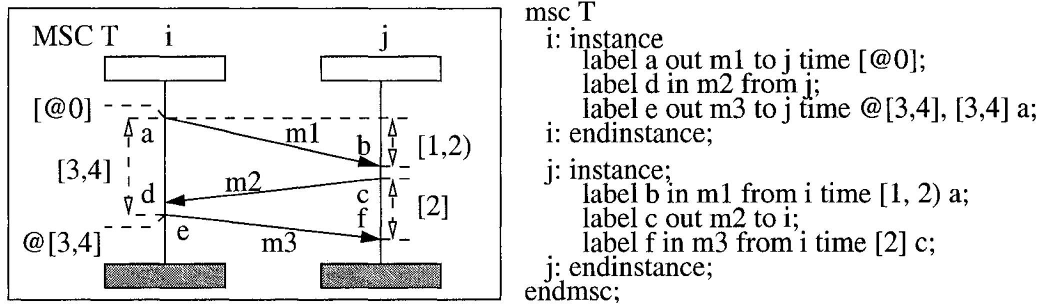


Figure 3.1: Graphical and textual notations of MSC

In this chapter, we define the semantics of MSC-2000 containing behavioral constructs, compositional constructs and timing constraints, which are another major extension to MSC-96. We consider both absolute and relative time constraints. Since we do not cover the data aspect, there are no data variables involved in time constraints. The lower and upper bounds of all the time constraints are concrete values in a time domain. The measurement of time is not considered also because it has to be related to a variable.

Examples of time constraints used in this chapter are shown in Figure 3.1. Both graphical syntax and textual syntax are shown. While the graphical form is easy to understand for human, the textual form is easy to handle when defining the semantics. We label the events in instances as *a, b, c, ...* in the figure. This MSC specifies the exchanges of messages *m1, m2* and *m3* between instances i and j. The relative time constraints between events a and b, a and e, c and f are specified. The absolute time constraints of a and e are also specified.

In the MSC-2000 standard, the timed semantics of an MSC is mentioned briefly as event traces with special time events between normal events. For example, a trace for the MSC in Figure 1 is $\{a, t_1, b, t_2, c, t_3, d, t_4, e, t_5, f\}$, where t_i represents time events. Actually time events are delays between normal events. If there is no time event between two normal events, it means they occur simultaneously. A trace always begins with a normal event.

This semantics is described informally in the standard and it is not mentioned how

an MSC corresponds to traces. Moreover, time events in traces can only express relative time constraints, while the informations specified by absolute time constraints are lost. For example, according to the relative time constraints in Figure 3.1, the following relations have to be satisfied: $1 \leq t_1 < 2$, $t_3 + t_4 + t_5 = 2$, and $3 \leq t_1 + t_2 + t_3 + t_4 \leq 4$. However, we cannot express that event a occurs at time 0, or event e occurs between 3 and 4 in the trace[1].

In this chapter, we develop a semantics for timed MSC based on partial orders. An MSC defines partial orders between events. Time constraints quantify the orders. Specifically, we define timed *labelled partially ordered sets* (lposets), which specify a set of traces of timed events. The denotational semantics of MSC is developed in a compositional way. We first define the semantics of events as timed lposets. Then the semantics of MSCs are obtained using the operations defined on timed lposets.

The rest of this chapter is organized as follows. In Section 3.2 we introduce timed lposets as a semantic model of timed MSC. In Section 3.3 we define the semantics of MSC in a compositional manner. In Section 3.4, we discuss existing work on the semantics of MSC. We conclude in Section 3.5.

3.2 A Partial Order Model for MSC

To define a formal semantics of MSC, we need an underlying model for describing the system that is specified by MSC. As introduced in Chapter 1, we are interested in distributed, concurrent systems. There are two kinds of models for concurrency, namely interleaving models and non-interleaving models. In an interleaving model, behaviors of a system are observed as totally ordered sequences of events that occur in different processes of the system. The concurrency between events are viewed as non-determinism between different

[1]To express absolute time constraints, we have to change the definition of trace, for example, adding a special event as the beginning of a trace.

interleavings of these events.

On the other hand, a non-interleaving model distinguishes between concurrency and non-determinism of interleavings. We prefer non-interleaving models since they are more truthful for modeling concurrent systems, especially real time ones. When we take time into account, the concurrency and the interleavings have to be differentiated. Concurrent events can occur at the same time, while interleaved events occur at different times. A non-interleaving model captures this difference because sequences of events in the model are only partially ordered. Therefore it is also referred to as a partial order model or a true concurrency model.

Well known examples of partial order models are Petri nets [94], event structures [87], Mazurkiewicz traces [78], pomsets (partial ordered multisets) [93], lposets (labelled partial order sets) [96], and asynchronous transition systems [100]. There are also timed extensions of partial order models, such as interval event structures [79], pomsets with delays [15], real time extended bundle event structures [54]. However, these timed partial order models cannot be used for timed MSC. In interval event structures, for instance, each event is associated with a duration. In MSC, events are instantaneous. Delays in pomsets and real time event structures are specified between two causally dependent actions. In MSC, time delays (relative time constraints) can be specified between any pair of events.

Therefore, instead of using these models, we extend lposets to timed lposets to meet the requirements of timed MSC. We choose lposets because it is straightforward to represent MSCs using lposets. An lposet defines causal orders between events. Events are labeled to indicate their types. Since there are two kinds of time constraints in MSC, we equip an lposet with two timing functions to describe these two types of constraints. In the following, we define formally timed lposets.

3.2.1 Timed lposets

We use *Time* to represent a time domain, which may be a set of non negative real or integer numbers. $P(Time)$ is a set of time intervals. An interval could be open or closed.

Definition 3.1 *A timed lposet is a 7-tuple* $(I, A, E, \leq, l, D, T)$*, where*

- *I is a set of processes (or instances).*
- *A is a set of labels.*
- *E is a set of events.*
- *$\leq$: $\subseteq E \times E$ is a partial order (which is a reflexive, anti-symmetric and transitive relation) on E. It specifies causal orders between events.*
- *l: $E \rightarrow A$ is a labeling function, which associates an event to a label. It can be a partial function.*
- *D: $E \rightarrow P(Time)$ is a function that associates an event to a time interval. It defines a range within which an event could occur.*
- *T: $E \times E \rightarrow P(Time)$ is a function that associates a pair of events to a time interval. It defines possible delays between two events.*

The set of labels A defines the types of events. An event in MSC could be: *message output, message input, internal action, start timer, stop timer,* or *timeout.* So a label is defined as one of the followings.

- *send(i, j, m_k)*: process i sends a message m_k to process j,
- *receive(i, j, m_k)*: process i receives a message m_k from process j,
- *action(i, a)*: process i does an internal action a,

- *starttimer(i, T, n)*: process i sets a timer T with a time-out period n,
- *stoptimer(i, T)*: process i cancels the timer T,
- *timeout(i, T)*: the timer T in instance i expires.

In the MSC standard, a *message output* or *message input* event can be associated with a *message instance name* to ensure that the textual notation corresponds to the graphical notation. The message instance name makes messages contained in an MSC differentiated. Thus we can associate every event in an MSC with a unique label. If a process i sends a message twice to another process j, we label the associated events as *send(i, j, m_1)*, *send(i, j, m_2)*, or *receive(i, j, m_1)*, *receive(i, j, m_2)*, where m is the message name, and the subscript of m represents the message instance name. The message instance name can be omitted if the messages can be differentiated from the message names.

The function l can be a partial function, which labels some events only. We allow this because in the MSC-2000, an event could have a relative time constraint with another event outside the current MSC. In such a case, we do not know the type of that event when considering the current MSC.

We use E^i to represent the set of events that occur at process i, such that $\bigcup_i E^i = E$, $i \in I$. An event e is a minimal element in E according to $\leq$ when there is no event $e' \in E$ such that $e' \neq e$ and $e' \leq e$. An event e is a maximal element in E according to $\leq$ when there is no event $e' \in E$ such that $e' \neq e$ and $e \leq e'$. There may be several minimal or maximal elements in E because of partial orders.

Using ϕ to represent the empty set, we define an empty timed lposet $\varepsilon = (I, A, E, \leq, l, D, T)$ in which I, A, E, $\leq$, l, D and T are ϕ.

Definition 3.2 *A trace of a timed lposet is defined as a (probably infinite) sequence of timed events* $(e_1, t_1), (e_2, t_2) \ldots (e_n, t_n)$ *with* $\bigcup_{i=1}^{n} \{e_i\} = E$, $t_i \in Time$ *such that for all* i *and* j, $0 < i \leq n$, $0 < j \leq n$:

- *if* $e_i \leq e_j$*, then* $t_i < t_j$*,*

- $t_i \in D(e_i)$*,*

- $|t_i - t_j| \in T(e_i, e_j)$*,*

- t_i *must grow over all bounds in an infinite sequence of timed events.*

A trace can be considered as an execution of a timed lposet. Informally, i and j represent positions of events in a trace. The first constraint means that two events in a trace have to satisfy their causal order, while the second and the third conditions enforce the time constraints of each event. The last condition excludes zeno behaviors, which allow infinite number of events to occur in a finite period of time.

In a trace, it is possible that $t_i > t_j$ if $i < j$ and $(e_i, e_j) \notin \leq$. In such a case, the trace is called ill-timed trace, otherwise it is called time-consistent trace [2, 54]. For each ill-timed trace, we can swap those unordered timed events to obtain a time-consistent trace [54].

Definition 3.3 *The set of traces Tr(lp) of a timed lposet lp is* $\{w \mid w$ *is a trace of lp*$\}$*.*

We can use timed lposets to describe the behavior and the time constraints of a system. For example, let us consider a process that performs two actions: collecting data and then computing. Collecting data occurs between time 1 and 5, and computing has to occur between time 3 and 6. Moreover, the delay between these two actions is at least 1 and at most 2. To model this process, we can use a timed lposet that contains one process (I = $\{i\}$), and two events (E = $\{a,\ b\}$) in the process. These two events are actions (A = $\{$*action*(i, *collect*), *action* (i, *compute*)$\}$). Event a is an action of collecting data ($l(a)$ = *action* (i, *collect*)), and b is an action of computing ($l(b)$ = *action*(i, *compute*)). The causal order between event a and b can be represented using $a \leq b$. It means a occurs causally before b. Their occurrence time can be represented as $D(a) = [1, 5]$, $D(b) = [3, 6]$, and the

delay between them can be represented as $T(a,\ b) = [1,\ 2]$. This timed lposet has many traces: $(e_1,\ 1)(e_2,\ 3)$, $(e_1,\ 2)(e_2,\ 3)$, $(e_1,\ 3)(e_2,\ 4)$ and so on.

3.2.2 Operations on lposets

A timed lposet describes a set of events or a part of an MSC. To define a compositional semantics based on lposets, we define sequential, parallel, and alternative operations on lposets.

For two lposets p and q, events in the lposets may be located at different sets of instances. We define the following requirements that have to be satisfied by the sequential composition.

- The orders and the time constraints in p and q are preserved.
- If one event corresponds to the sending of a message, and another event corresponds to its receiving, add a new order between these two events,
- If one event corresponds to the starting of a timer with a time-out period, and another event corresponds to its termination or expiration, add a relative time constraint between them,
- If two events are located at the same instance, add a new order between these two events.

Formally, let $p = (I_p, A_p, E_p, \leq_p, l_p, D_p, T_p)$ and $q = (I_q, A_q, E_q, \leq_q, l_q, D_q, T_q)$ be two timed lposets. For a relation S, we use S^+ to represent the transitive closure of S.

Definition 3.4 *The sequential composition $(\cdot)$ of p and q $(E_p \cap E_q = \phi)$ is defined as follows:*

$p \cdot q = (I_p \cup I_q, A_p \cup A_q, E_p \cup E_q, (\leq_p \cup \leq_q \cup \leq_{msg} \cup \leq_{ins})^+, l_p \cup l_q, D_p \cup D_q, T_p \cup T_q \cup T_{tim})$, *in which*

- $\leq_{msg} = \leq^p_q \cup \leq^q_p$, $\leq^p_q = \{(e, e') \in E_p \times E_q \mid l_p(e) = send(i,\ j,\ m_k) \wedge l_q(e') = receive(j,\ i,\ m_k)\}$, $\leq^q_p = \{(e, e') \in E_q \times E_p \mid l_q(e) = send(i,\ j,\ m_k) \wedge l_q(e') = receive(j,\ i,\ m_k)\}$,

- $\leq_{ins} = \bigcup_i (E^i_p \times E^i_q)$, *$E^i_p$ and E^i_q are the sets of events that occur at instance i*, $E^i_p \subseteq E_p$, $E^i_q \subseteq E_q$,

- $T_{tim} = T_1 \cup T_2 = \{((e, e'), [n]) \mid e \leq_{ins} e',\ \nexists f$ *(f is a timer event and* $e \leq_{ins} f \leq_{ins} e'$*)*, $l_p(e) = starttimer(i,\ T,\ n),\ l_q(e') = timeout(i,\ T)\} \cup \{((e, e'), (0, n)) \mid e \leq_{ins} e',\ \nexists f$ *(f is a timer event and* $e \leq_{ins} f \leq_{ins} e'$*)*, $l_p(e) = starttimer(i,\ T,\ n),\ l_q(e') = stoptimer(i,\ T))\}$.

In some cases, $(\leq_p \cup \leq_q \cup \leq_{msg} \cup \leq_{ins})^+$ may be not a partial order [55]. For example, if in two timed lposets p and q, $\leq_p = \{a \leq b\}$, $l_p = \{(a,\ receive(i,\ j,\ m)),\ (b,\ send(i,\ j,\ n))\}$, and $\leq_q = \{c \leq d\}$, $l_q = \{(c,\ receive(j,\ i,\ n)),\ (d,\ send(j,\ i,\ m))\}$, then the sequential composition of p and q contains relations $\{a \leq b,\ c \leq d,\ d \leq a,\ b \leq c\}$. We obtain $a \leq d$ and $d \leq a$, which violates the anti-symmetry property of the partial order. In MSC, this problem is avoided by a static requirement in the syntax, which does not allow a *message output* to depend causally on its *message input* via other messages or general orderings. Drawing rules are defined in the standard for this static requirement. So the sequential composition of two timed lposets is still a timed lposet in MSC.

Definition 3.5 *The parallel composition* ($\|$) *of p and q ($E_p \cap E_q = \phi$) is defined as a timed lposet:*

$$p \parallel q = (I_p \cup I_q, A_p \cup A_q, E_p \cup E_q, \leq_p \cup \leq_q, l_p \cup l_q, D_p \cup D_q, T_p \cup T_q).$$

In the sequential and parallel composition, we require that E_p and E_q are disjoint. Then A_p and A_q are disjoint also since every event is associated with a unique label by l_p and l_q. This ensures that in the lposet obtained from the composition, every event is associated with a unique label also.

Definition 3.6 *The alternative composition (#) of two lposets p and q is a set of lposets:*

$$p\#q = \{p\} \cup \{q\}.$$

While the result of a sequential or parallel composition is a timed lposet, the outcome of an alternative composition between p and q is a set of lposets. The alternative composition is different with the delay choice in the MSC standard. In a delayed choice ($\mp$), if p and q contain some common events, the choice is delayed until the events are different. So those common events appear only once in $p \mp q$. In the alternative composition, the choice is made before the execution of any event in p or q. In $p \# q$, the common events appear in both alternations.

We generalize the definition of these operations on sets of lposets.

Definition 3.7 *For two sets of lposets* $P = \{p_1, p_2, \dots p_n\}$ *and* $Q = \{q_1, q_2, \dots q_k\}$,

- $P \cdot Q = \{p_i \cdot q_j \mid p_i \in P, q_j \in Q, 1 \le i \le n, 1 \le j \le k\}$.
- $P\#Q = P \cup Q$.
- $P \parallel Q = \{p_i \parallel q_j \mid p_i \in P, q_j \in Q, 1 \le i \le n, 1 \le j \le k\}$.

The sequential, parallel and alternative compositions have the following properties.

Proposition 3.1 *Let p, q and r be lposets, P, Q and R be sets of lposets, and ε be the empty timed lposet.*

- *Identity.*
  - $\varepsilon \cdot p = p \cdot \varepsilon = p$.
  - $\varepsilon \parallel p = p \parallel \varepsilon = p$.

- *Commutative property.*

  - $p \parallel q = q \parallel p$, *and* $P \parallel Q = Q \parallel P$.

  - $p\#q = q\#p$, *and* $P\#Q = Q\#P$.

- *Associative property.*

  - $p \cdot (q \cdot r) = (p \cdot q) \cdot r$, *and* $P \cdot (Q \cdot R) = (P \cdot Q) \cdot R$.

  - $p \parallel (q \parallel r) = (p \parallel q) \parallel r$, *and* $P \parallel (Q \parallel R) = (P \parallel Q) \parallel R$.

  - $\{p\}\#(q\#r) = (p\#q)\#\{r\}$, *and* $P\#(Q\#R) = (P\#Q)\#R$.

- *Distributive property.*

  - $\{p\} \cdot (q\#r) = (p \cdot q)\#(p \cdot r)$, *and* $P \cdot (Q\#R) = (P \cdot Q)\#(P \cdot R)$.

  - $\{p\} \parallel (q\#r) = (p \parallel q)\#(p \parallel r)$, *and* $P \parallel (Q\#R) = (P \parallel Q)\#(P \parallel R)$.

PROOF. This proposition is proven in Appendix B.1. □

3.3 Partial Order Semantics for Timed MSC

As mentioned in Chapter 2, MSCs can be classified as plain MSCs and HMSCs. A plain MSC consists of orderable events and non-orderable events in the standard syntax. We use a simplified syntax as described in Appendix A. In this syntax, orderable events include message events, actions, and timer events. Non-orderable events include coregions, inline expressions and MSC references. Inline expressions and MSC references are actually the compositions of MSCs.

We define the semantics of an MSC as a set of lposets. We first define the semantics of orderable events as lposets. By composing them together, we obtain the semantics of a plain MSC containing only these events, which is a set including only one lposet. Then

we define the semantics of non-orderable events, and the semantics of MSCs containing them by composing orderable and non-orderable events. At last we define the semantics of HMSCs by composing MSCs.

The set of traces allowed by an MSC is defined as follows. We use $\mathcal{M}$ to represent a semantic mapping that maps events or MSCs to lposets or sets of lposets.

Definition 3.8 *For an MSC H represented by a set of lposets* $\mathcal{M}[H]$*, the set of traces of H is* $Tr(H) = \bigcup_i Tr(lp_i)$*,* $lp_i \in \mathcal{M}[H]$*.*

3.3.1 Orderable Events

The semantics of an orderable event is a timed lposet. For example, the message output event e in Figure 3.1 (page 38) is located in instance i. It has an absolute time constraint and a relative time constraint that specifies the delay between event e and event a. It is mapped to a timed lposet $\mathcal{M}[e] = (I,\ A,\ E,\ \leq,\ l,\ D,\ T)$, in which

- $I = \{i\}$,
- $A = \{send(i, j, m3)\}$,
- $E = \{e, a\}$ (event a also appears because it is associated with e by the time constraint),
- $\leq = \{(e, e), (a, a)\}$,
- $l(e) = send(i, j, m3)$ (here l is a partial function),
- $D(e) = [3, 4]$,
- $T(e, a) = [3, 4]$.

In the same way, other orderable events can be represented using lposets too.

3.3.2 MSCs Containing only Orderable Events

The semantics of an MSC is a set of lposets. If the MSC contains only orderable events, then the set contains only one timed lposet. The lposet is obtained by composing sequentially the lposets representing the events. Therefore, it contains all the events and specifies the orders between them. The orders are determined by message exchanges and instance axes. Along each instance axis, events are ordered from top to bottom. Between different instances, a *message output* event must appear before the corresponding *message input* event. As mentioned in Section 3.2.1, we need to label the events in a way such that every event is unique.

For example, the MSC T in Figure 3.1 contains events a, b, c, d, e and f. Its semantics is the sequential compositions of these events along the instances i and j. $\mathcal{M}[T] = \{\mathcal{M}[a] \cdot \mathcal{M}[d] \cdot \mathcal{M}[e] \cdot \mathcal{M}[b] \cdot \mathcal{M}[c] \cdot \mathcal{M}[f]\} = \{(I, A, E, \leq, l, D, T)\}$, where:

- *I:* $\{i,\ j\}$,
- *A:* $\{$*send(i, j, m1), receive(j, i, m1), send(j, i, m2), receive(i, j, m2), send(i, j, m3), receive(j, i, m3)*$\}$.
- *E:* $\{a,\ b,\ c,\ d,\ e,\ f\}$.
- $\leq$: $\{(a,b)(c,d)(e,f)(d,e)(b,c)\}^+$. (For the sake of simplicity, we omit those reflexive pairs such as $(a,\ a)$, $(b,\ b)$ $\ldots$).
- *l: l(a) = send(i, j, m1), l(b) = receive(j, i, m1), l(c) = send(j, i, m2), l(d) = receive(i, j, m2), l(e) = send(i, j, m3), l(f) = receive(j, i, m3).*
- D: $D(a) = 0$, $D(e) = [3,4]$.
- T: $T(a,b) = [1,2)$, $T(a,e) = [3,4]$, $T(c,f) = [2]$.

In the MSC T, not all the events are constrained by absolute or relative time constraints. We assume the time constraints for these events to be the whole time domain. For example,

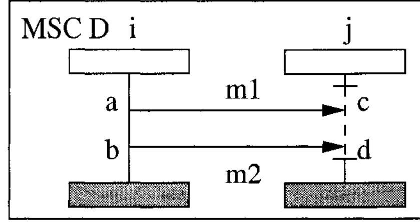


Figure 3.2: An MSC with coregion

if the time domain is the domain of non-negative real numbers, then the time constraints are $[0, \infty)$.

The set of traces for the MSC in Figure 3.1 includes *(a, 0)(b, 1)(c, 1.6)(d, 2)(e, 3.5)(f, 3.6)*, *(a, 0)(b, 1.5)(c, 2)(d, 2.5)(e, 3)(f, 4)*, and many others that satisfy the definition of trace.

3.3.3 Coregion

In an instance, a coregion contains a number of events that can be executed in any order. The semantics of a coregion is an lposet obtained by composing in parallel all the lposets representing the events in the coregion.

$$\mathcal{M}[coregion] = \mathcal{M}[e_1] \parallel \mathcal{M}[e_2] \parallel \ldots \parallel \mathcal{M}[e_n],\ e_i \in coregion.$$

For example, in Figure 3.2 (page 50), a coregion (the dash line) in instance j contains event c and d. It means either c or d can occur first, although a (sending m1) occurs before b (sending m2). The semantics of the coregion is $\mathcal{M}[coregion] = \mathcal{M}[c] \parallel \mathcal{M}[d]$, in which the causal order relation is $\{(c, c), (d, d)\}$. There is no order between c and d.

The semantics of MSC D is $\mathcal{M}[D] = \{\mathcal{M}[a] \cdot \mathcal{M}[b] \cdot \mathcal{M}[coregion]\} = \{(I, A, E, \leq, l, D, T)\}$, in which $\leq$ is $\{(a, b), (a, c), (b, d)\}^+$ (reflexive pairs are omitted).

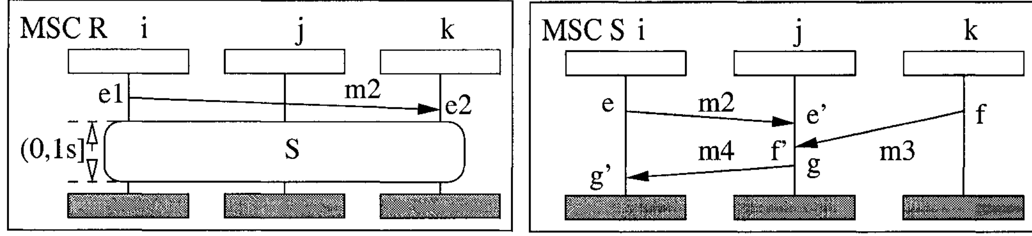


Figure 3.3: An MSC with a reference

3.3.4 MSC Reference

An MSC reference is used to refer to a single MSC, or an MSC expression. An MSC reference itself can be constrained by a relative time constraint. In such a case, the relative time constraint specifies the duration between the first and the last event in the MSC reference. If there are several first or last events in an MSC, then all of them should be constrained. Therefore, the semantics of an MSC reference is the semantics of the referred MSC with the relative time constraint.

For example, in Figure 3.3, MSC R has a reference which refers to MSC S with a relative time constraint (0, 1]. In MSC S, either e or f could be the first event. So the time constraint of this MSC reference specifies the time between e and g', and between f and g' also.

Assume the lposet for MSC S is $(I, A, E, \leq, l, D, T)$. We map the reference S with the relative time constraint to $\mathcal{M}[S] = \{(I, A, E, \leq, l, D, T')\}$, where

- E: $\{e, e', f, f', g, g'\}$
- $\leq$: $\{(e, g'), (e', f'), (f', g), (e, e'), (f, f'), (g, g')\}^+$, (reflexive orders are omitted.)
- l: $\{l(e) = send(i, j, m2),\ l(e') = receive(j, i, m2),\ l(f) = send(k, j, m3),\ l(f') = receive(j, k, m3),\ l(g) = send(j, i, m4),\ l(g') = receive(i, j, m4)\}$
- $T'(e, g') = (0, 1]$, $T'(f, g') = (0, 1]$.

Then the semantics of the MSC R in Figure 3.3 is $\mathcal{M}[R] = \{\mathcal{M}[e_1]\} \cdot \{\mathcal{M}[e2]\} \cdot \mathcal{M}[S]$.

A reference could refer to an MSC expression also. We consider MSC expressions in the next section.

3.3.5 Inline Expressions and MSC Expressions

Several operators have been defined in the MSC standard to combine MSCs. We consider the following operators: sequence (*seq*), alternation (*alt*), parallel (*par*), and iteration (*loop*). As defined in the MSC standard, the sequence operation is weak sequencing. Two MSCs are connected instance by instance only. It means that the next MSC may start before the previous MSC finishes its behavior. Our sequential composition models this property.

We map these operations to the compositions of lposets. For two MSCs or inline operands A and B, we have

- $\mathcal{M}[A \; seq \; B] = \mathcal{M}[A] \cdot \mathcal{M}[B]$,
- $\mathcal{M}[A \; alt \; B] = \mathcal{M}[A] \; \# \; \mathcal{M}[B]$,
- $\mathcal{M}[A \; par \; B] = \mathcal{M}[A] \; \| \; \mathcal{M}[B]$,
- $\mathcal{M}[loop\langle i,j\rangle A] = \mathcal{M}[A^i] \# \mathcal{M}[A^{i+1}] \# \ldots \# \mathcal{M}[A^j]$, where $\mathcal{M}[A^k] = \mathcal{M}[A]{\cdot}\mathcal{M}[A^{k-1}]$ for $k > 0$, and $\mathcal{M}[A^0] = \varepsilon$.

When calculating $\mathcal{M}[A^k]$, we need to relabel message events in MSC A so that they are unique in the iteration. If a loop is infinite, the set representing its semantics may contain infinite number of lposets.

Let us consider an alternative inline expression as shown in Figure 3.4. In the MSC, instance i sends *m1* or *m2* first, then instance j returns *m3*.

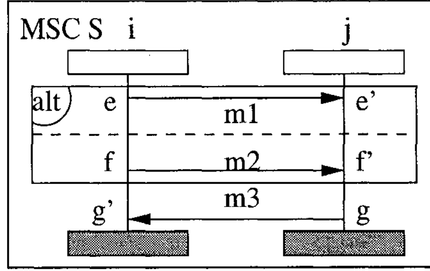


Figure 3.4: An MSC with inline expressions

For the alternative expression, the lposet of its first operand is $A_1 = (I, A, E, \leq, l, D, T)$, in which $\leq$ is $\{(e, e')\}$, $l(e) = send(i, j, m1)$, $l(e') = receive(j,\ i,\ m1)$. We omit the reflexive pairs and do not list elements in *I, A, E, D* and *T* for the sake of simplicity. Similarly, the second operand corresponds to a lposet A_2 in which $\leq$ is $\{(f, f')\}$, $l(f) = send(i, j, m2)$, $l(f') = receive(j, i, m2)$. So the alternative expression can be represented by $\{A_1, A_2\}$.

Then MSC S corresponds to a set of lposets $P = \{A1, A2\} \cdot \{\mathcal{M}[g']\} \cdot \{\mathcal{M}[g]\} = \{(I_1, A_1, E_1, \leq_1, l_1, D_1, T_1), (I_2, A_2, E_2, \leq_2, l_2, D_2, T_2)\}$, in which

- $\leq_1 = \{(e, e'), (e', g), (g, g')\}^+$ (reflexive pairs are omitted.)
- $\leq_2 = \{(f, f'), (f', g), (g, g')\}^+$ (reflexive pairs are omitted.)

3.3.6 High Level MSC (HMSC)

An HMSC is a directed graph in which MSCs are represented by nodes, and lines connect the nodes to indicate possible execution sequences among the nodes.

Definition 3.9 *An HMSC is a directed graph (S, E, L), where S is a finite set of nodes, $E \subseteq S \times S$ is the set of directed edges, L is a function that maps each node in S to an MSC.*

An HMSC can be transformed to an MSC expression consisting of the *seq, alt, par* and *loop* operators. This is similar to the transformation from a finite state machine to a

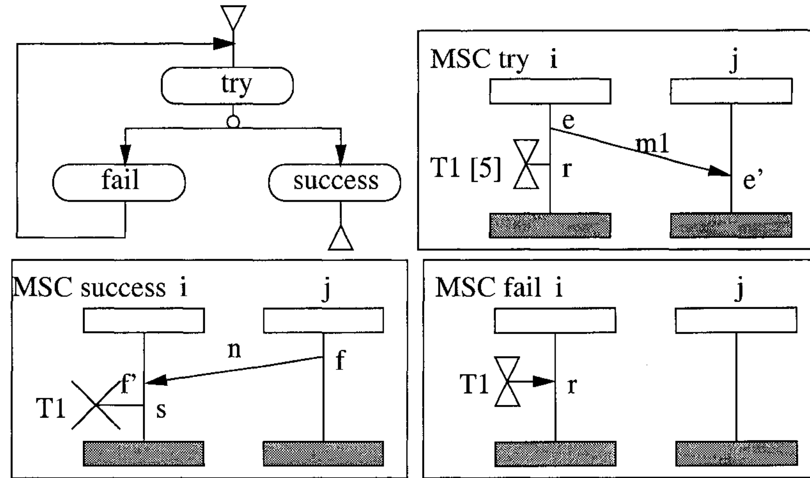


Figure 3.5: An HMSC and referred MSCs

regular expression [47]. In [95], rewriting rules are defined for transforming an HMSC to an expression. Here we only show the transformation through an example.

For instance, in Figure 3.5, the HMSC contains three MSCs: *try, fail* and *success*. After MSC *try*, MSC *fail* or *success* can be executed. They are alternative.

This HMSC is transformed to an expression in the following steps.

- First it is transformed to an automaton as shown in Figure 3.6(a). We take the start node, end nodes and connection points as states (0, 1, 2), and MSC references as transitions. We add an extra initial state (I) and an extra final state (F), which connect to the states representing the start node and end nodes by ε-transitions.
- We remove states other than the initial state and the final state one by one, and replace transitions. In Figure 3.6(a), State 2 has one incoming edge labeled with *success* from state 1 and one outgoing edge labeled with ε to state F. When state 2 is removed, we add an edge labeled with *success* $\cdot\varepsilon =$ *success* from state 1 to state F. The result is shown in Figure 3.6(b).
- In Figure 3.6(b), state 1 has one incoming edge from state 0 and two outgoing edges *fail* and *success* to state F and state 0 respectively. When removing state 1, we add

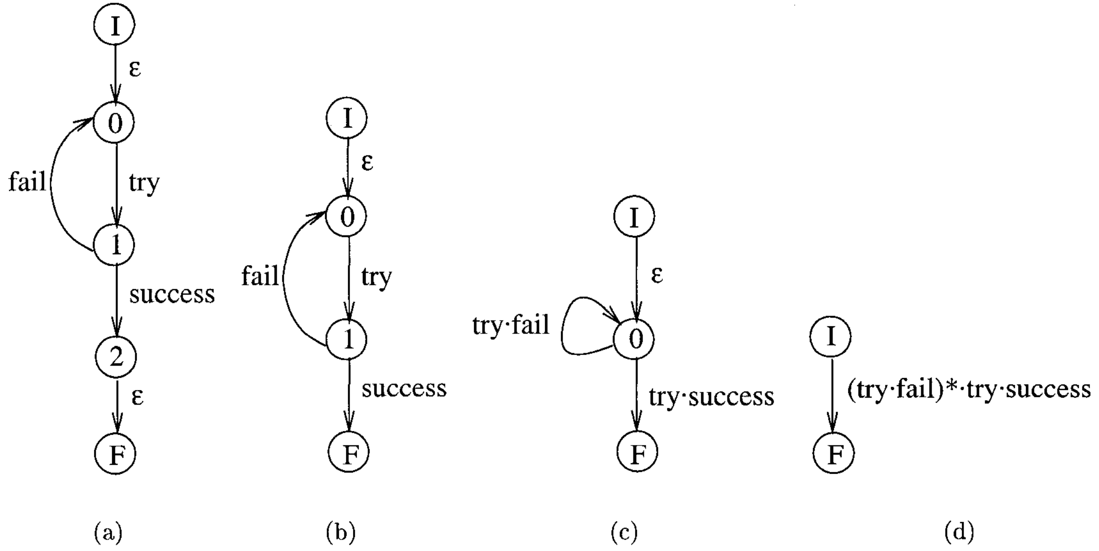


Figure 3.6: Transforming an HMSC to an expression

an edge labeled with $try \cdot success$ from state 0 to state F, and an edge labeled with $try \cdot fail$ from state 0 to state 0. The result is shown in Figure 3.6(c).

- In Figure 3.6(c), state 0 has one incoming edge from state I, one outgoing edge to state F, and one self-loop. When state 0 is removed, we add an edge from state I to state F labeled with $\varepsilon \cdot (try \cdot fail)^* \cdot try \cdot success = (try \cdot fail)^* \cdot try \cdot success$. The final result is shown in Figure 3.6(d).

After these steps, the transition from the initial state to the final state represents the expression equivalent to the automaton. During the transformation, states can be removed in any order. Different orders result in different look but equivalent expressions. Using operators in MSC, the expression can be written as:

$$(loop\langle 0, \infty\rangle(try\ seq\ fail))\ seq\ try\ seq\ success$$

Given the expression, we first define the semantics for individual MSCs.

- $\mathcal{M}[try] = \{(I, A, E, \leq, l, D, T)\}$ where $E = \{e, r, e'\}$, $\leq = \{(e,\ e'),\ (e,\ r)\}$ (re-

flexive pairs are omitted), $l(e) = send(i, j, m)$, $l(e') = receive(j, i, m)$, and $l(r) = starttimer(i, T1, 5)$.

- $\mathcal{M}[success] = \{(I, A, E, \leq, l, D, T)\}$ where $E = \{f, s, f'\}$, $\leq = \{(f, f'), (f', s)\}$ (reflexive pairs are omitted), $l(f) = send(j, i, n)$, $l(f') = receive(i, j, n)$, $l(s) = stoptimer(i, T1)$.

- $\mathcal{M}[fail] = \{(I, A, E, \leq, l, D, T)\}$ where $E = \{t\}$, $\leq = \{(t, t)\}$, $l(t) = timeout(i, T)$.

If we use TF to represent $\mathcal{M}[try\ seq\ fail]$, and use TS to represent $\mathcal{M}[try\ seq\ success]$, then

- $TF = \mathcal{M}[try] \cdot \mathcal{M}[fail] = \{(A_{TF}, \{e, e', r, t\}, \{(e, e'), (e, r), (r, t)\}^+, \{l(e), l(e'), l(r), l(t)\}, D_{TF}, T_{TF})\}$ in which $T_{TF}(r, t) = [5]$. (reflexive pairs and concrete labels are omitted for the sake of simplicity.)

- $\mathcal{M}[loop\langle 0, \infty\rangle(try\ seq\ fail)] = \{\varepsilon, TF, TF \cdot TF, TF \cdot TF \cdot TF, \ldots\}$.

- $TS = \mathcal{M}[try] \cdot \mathcal{M}[success] = \{(A_{TS}, \{e, e', f, f', r, s\}, \{(e, e'), (e, r), (f, f'), (f', s), (r, f'), (e', f)\}^+, \{l(e), l(e'), l(f), l(f'), l(r), l(s)\}, D_{TS}, T_{TS})\}$ in which $T_{TF}(r, s) = (0, 5)$. (reflexive pairs and concrete labels are omitted for the sake of simplicity.)

So this HMSC corresponds to a set of lposets: $\{TS, TF{\cdot}TS, TF{\cdot}TF{\cdot}TS, TF{\cdot}TF{\cdot}TF{\cdot}TS, \ldots\}$.

3.4 Related Work

The semantics in the MSC-96 standard is based on process algebra [76, 77]. First, an MSC is transformed into a process expression. Compositions in MSC are represented by appropriate process algebra operators. Then an operational semantics is defined by taking the process expression as an initial state of the MSC, and applying inference rules on it. This semantics

is extended in [27] to handle conditions as compositional constructs. The approach in [95] is slightly different, although it is also based on process algebra. In [95], a denotational semantics is given by defining mappings that transform an MSC to a process expression. In our semantics, we map an MSC to a set of lposets. The alternative composition defined in this chapter is not the delayed choice used in [76, 77, 95]. In the alternative composition, a choice is made at the beginning of an execution instead of the latest moment. This alternative composition is more suitable for specifications at a higher level of abstraction, while the delayed choice is more related to lower level specifications and implementations. However, from the trace point of view, they are equivalent [95].

A partial order semantics for un-timed MSC has been proposed in [7, 45, 55]. The authors of [7] only consider the semantics of MSCs containing message events. A more complete semantics is defined in [55]. In this semantics, an MSC is mapped to a family of pomsets. Various operations are defined on pomsets for composing MSCs. In [45], an MSC is translated into terms in a process algebraic language. Then two semantics, an interleaving one and a non-interleaving one are defined for the process algebraic language based on families of lposets. The semantics in this chapter extends these semantics to timed MSCs. Our sequential composition can be seen as the combination of the joining and the local concatenation in [55].

The semantics of MSC can be defined using other approaches also, such as Petri net [33], automata [62, 63, 64]. In [33], the MSC-92 language is translated to labeled occurrence nets. However, it may be difficult to translate the MSC-96 language because of its richer constructs and the ability to specify infinite behaviors in MSC-96. In [62, 63, 64], an MSC is translated into a graph with signal edges for communication and next-event edges for ordering in an instance. Then the graph is translated into a global state transition graph. Based on this graph, Büchi automata can be defined given acceptance criteria. This semantics is suitable for a subset of MSC only, since the MSC language is not a regular language [42, 43, 44]. For example, the non local choices in MSCs may require unbounded

history variables, which result in an infinite number of global states [11].

The semantics of MSC with timing constructs has been considered in [7, 10, 71]. In [7], a timed MSC is interpreted as partial orders with timing functions that associate each pair of events in the partial order to a time interval. In [10], timing delay intervals and timer events are used to express timing constraints. An MSC is interpreted as traces that are consistent with the partial order of events. A timing assignment is used to assign a time stamp to each event in a trace. In [71], each event and communication in MSC is associated with a duration. An MSC is translated to an order automaton. These semantics consider the time in MSC differently than MSC-2000. They only allow the time constraints defined between two ordered events. In MSC-2000, time constraints can be specified between any two events. Moreover, absolute time constraints can be used to specify the occurrence time of events. We define the semantics following the time concept in MSC-2000.

3.5 Conclusion

Time constraints are new in MSC-2000. To support the usage of time constraints in specification and validation of real-time systems, we define a denotational semantics for timed MSC based on timed lposets. We first map orderable events, such as message events, actions, or timer events, to timed lposets. An MSC containing only these orderable events can be seen as their sequential composition. An MSC containing inline expressions or MSC references is the (sequential, alternative, or parallel) composition of orderable events and MSCs, while an HMSC is the composition of MSCs. Thus the semantics is built compositionally using lposets. The semantics is also applicable to un-timed MSCs where time constraints can be seen as the whole time domain.

Given a (finite) set of lposets, we can define an operational semantics for MSCs also. First, we consider the set as the initial state of the MSC. Then we remove one minimal event with regard to the partial order to get a new set of lposets, which represents a new

state. We can repeat this procedure until all the events are removed.

Some constructs in the MSC standard are not taken into account, such as general orderings, instance decompositions, gates and conditions. General orderings specify some additional orders, which can be easily represented in our semantics. Instance decompositions and gates are considered as transformations in syntax and do not affect the dynamic semantics of MSCs. Guard conditions can be associated with predicates that restrict the traces of lposets. Moreover, the *opt* and *exc* operators in the inline expressions and reference expressions are not considered also. These operators can be seen as special cases of the *alt* operator.

The semantics in this chapter provides a foundation for analyzing MSC specifications with time constraints. Time constraints add extra requirements on a system. However, these requirements may be not consistent with the behaviors specified in MSC. For example, a loop may specify that an event is executed infinitely while an absolute time constraint may restrict it to be executed several times. In the next chapter, we consider the consistency issue in detail.

Chapter 4

Time Consistency of MSC Specifications

4.1 Introduction

An MSC specification has to be validated to ensure that it does not contain semantic errors or logical inconsistencies, such as race conditions [7], process divergence [11] and inferences [5]. These errors or inconsistencies make the MSC specification un-implementable, or result in undesired implementations. In the case of timed MSC, time constraints may also cause inconsistencies. One of our goals in this thesis is the validation of the time consistency of an MSC specification.

Time inconsistencies are results of different causes. In an MSC, time constraints specify temporal orders between events. These orders have to be consistent with the causal orders of events. For example, in Figure 4.1(a), event a occurs before event b. However, the absolute time constraint for event a and b are @[5, 6] and @[2, 3] respectively. The order defined by time constraints contradicts the causal order between event a and b. Moreover, time constraints have to be consistent with each other. In Figure 4.1(b), the absolute time

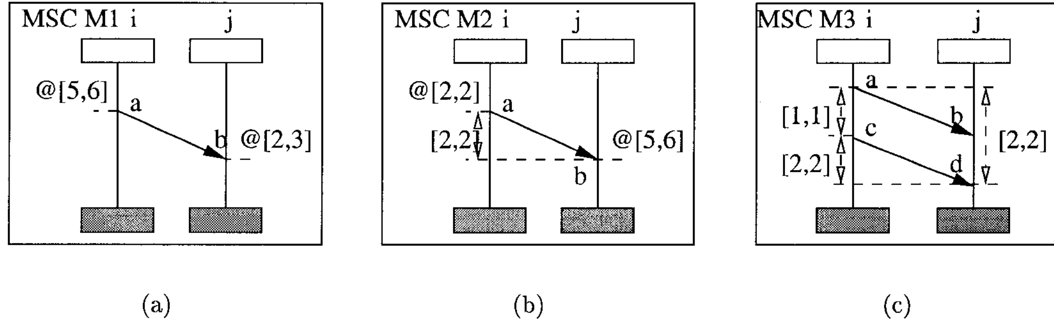


Figure 4.1: Time inconsistency in bMSCs

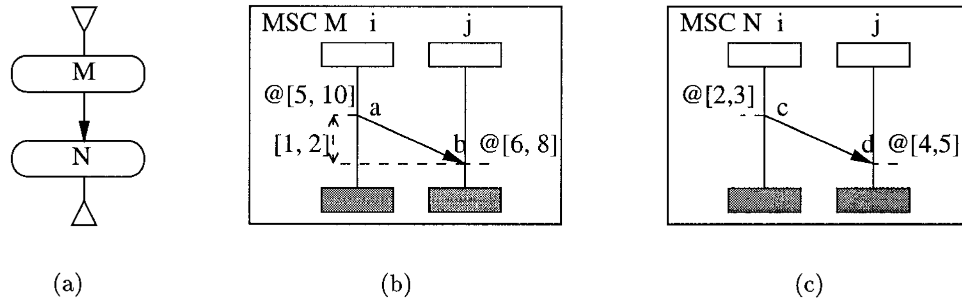


Figure 4.2: Time inconsistency in an HMSC

constraint of event a (@[2, 2]) and the relative time constraint between a and b ([2, 2]) determine that the event b should occur at 4. It contradicts with the specified absolute time constraint (@[5, 6]). An example of inconsistency between relative time constraints is shown in Figure 4.1(c). Events a, c and d occur in that order. The delay between events a and c is 1, and the delay between c and d is 2. This implicitly sets the delay between a and d to 3, which is conflicting with the relative time constraint specified in the MSC.

In the case of an HMSC, the consistency of the component MSCs does not guarantee the consistency of the HMSC. For instance, the HMSC in Figure 4.2(a) consists of a sequential composition of MSC M and MSC N. Time constraints in M and N, respectively, are consistent. However, when M and N are composed sequentially, time constraints of events a and c contradict their causal order. Therefore, this HMSC is not time consistent.

In this chapter, we define the time consistency of MSCs and develop algorithms to

check the consistencies. Instead of considering the full MSC language, we focus on the most important and used part. We consider bMSCs containing only processes and message exchanges, and HMSCs that are composed of bMSCs only. We choose the time domain as $[0, \infty)$, which can be real numbers or integers. We take into account both relative and absolute time constraints as defined the MSC-2000 standard, but with the following restrictions.

First, we restrict that relative time constraints appear only between causally ordered events. An MSC containing relative time constraints between unrelated events cannot be implemented in a distributed architecture, although it is reasonable to define such an MSC. For instance, in Figure 4.1(c), either event b or c could occur first. If we add a relative time constraint $[1, 2]$ between them, once one event occurs, the other one has to wait at least 1 and at most 2. Actually process i does not know what happens in process j, and vice versa. This time requirement is at a high level of abstraction and cannot be satisfied without refining the MSC and adding other message exchanges between the processes to relate these events. We do not take into account this kind of time constraints.

Furthermore, in an HMSC, we restrict that relative time constraints appear only between events in the same bMSC. Apparently, relative time constraints between events in two alternative bMSCs are not meaningful. For those between events in sequential bMSCs, we consider they are $(0, \infty)$[1]. In Section 4.7, we discuss briefly how to extend the results in this chapter to handle the situation where they are not $(0, \infty)$.

The rest of this chapter is organized as follows. We first define time consistency for bMSCs in Section 4.2. Then we investigate the consistency for HMSCs in Section 4.3. We distinguish between strong and weak consistency for HMSCs. Sufficient and necessary conditions are devised for both consistencies. We describe and discuss algorithms for checking these consistencies for HMSCs in general in Section 4.4, and for some HMSCs with certain specific properties in Section 4.5. We discuss related work in Section 4.6 and conclude in

[1]Since we allow for relative time constraints between causally ordered events only, the relative time constraint cannot be 0.

Section 4.7.

4.2 Time Consistency of bMSCs

As introduced in Section 4.1, time inconsistencies can be caused by inconsistencies between time constraints, or between the causal order and the temporal order specified by time constraints. Intuitively, an MSC is time consistent if all its events can be executed with the satisfaction of their (absolute and relative) time constraints and causal orders. This is the case when the events in the MSC can form a trace. If a bMSC is time consistent, then it should have at least one execution represented by a trace. Therefore, we define the consistency of a bMSC in terms of traces as follows.

Definition 4.1 *A timed bMSC is time consistent if and only if it has a trace.*

Deciding if a bMSC has at least one trace is equivalent to solving the simple temporal problem in directed constraint graph [19]. As defined in Chapter 3, the semantics of an MSC is a set of timed lposets. In the case of bMSCs, the set contains only one lposet. We can model a timed lposet as a directed constraint graph, where nodes are events and there exists an edge $e_i \rightarrow e_j$ if $e_i \leq e_j$. Edges are labeled by relative time constraints between events. (As mentioned before, we limit that relative time constraints exist between causally ordered events only.) We add a special event (node) e_0 in the graph; it occurs causally before all the events in a bMSC and it occurs at time zero. So absolute time constraints can be translated as relative time constraints between events and e_0.

A directed constraint graph can be associated further with a distance graph, which has the same node set. For an edge $e_i \rightarrow e_j$ in the directed constraint graph, there are two edges in the distance graph, one is $e_i \rightarrow e_j$ labeled by the upper bound of the time constraint between e_i and e_j, another one is $e_j \rightarrow e_i$ labeled by the negative of the lower bound of the

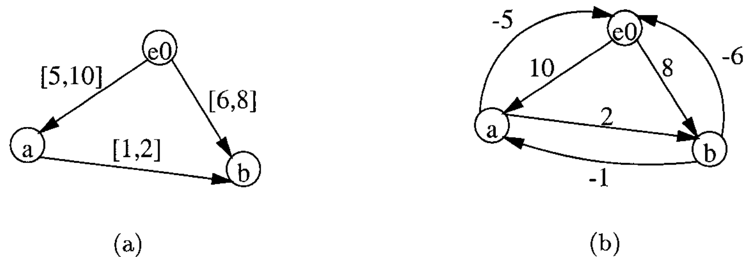


Figure 4.3: A directed constraint graph and its distance graph

time constraint. However, if the upper bound is infinity, the edge $e_i \rightarrow e_j$ is usually omitted in the distance graph. A directed constraint graph and a distance graph corresponding to the MSC in Figure 4.2(b) are shown in Figure 4.3(a) and Figure 4.3(b).

The simple temporal problem is to decide if each node in a directed constraint graph can be assigned to a value, such that the constraints between nodes are all satisfied. If such assignments exist, we can constitute a trace using the time values decided by these assignments. Thus the bMSC associated with the graph is consistent. As studied in [19], the consistency can be decided by using the Floyd-Warshall algorithm to compute all pairs shortest paths in a corresponding distance graph. If there are no cycles with a negative cost in the distance graph, then the bMSC is consistent.

If a bMSC is consistent, according to Corollary 3.4 and 3.5 in [19], we can obtain a unique interval for each node e, in which each value can be an assignment to e such that the graph is consistent. We define such an interval as a reduced absolute time constraint.

Definition 4.2 *Given a consistent bMSC and its distance graph, a reduced absolute time constraint of an event e is a maximal interval within its original absolute time constraint, such that the upper bound is the shortest distance from the node e_0 to e, and the lower bound is the negative of the shortest distance from e to the event e_0.*

For example, in Figure 4.3, the reduced absolute time constraints for events a and b are @[5, 7] and @[6, 8] respectively.

Another solution for the simple temporal problem is to achieve directional path consistency directly in the constraint graph. It can be done in time $O(nw^2)$ using the DPC algorithm in [19], where n is the number of nodes in a constraint graph, and w is the number of parents of a node. Usually w is much lower than n. So DPC is more efficient than the Floyd-Warshall algorithm, which has the complexity $O(n^3)$. However, unlike the latter, achieving directional path consistency does not result in reduced absolute time constraints. To obtain reduced absolute time constraints, we can compute shortest paths between the special event e_0 and any other event in the distance graph. This can be done in time $O(nE)$, where E is the number of edges in the graph [17].

4.3 Time Consistency of HMSCs

In this thesis, we view a timed HMSC as a set of timed bMSCs composed together. Time constraints are still specified within bMSCs. An instance of inconsistent HMSC is shown in Figure 4.2. Moreover, loops in an HMSC may cause more inconsistencies if events in a loop are constrained by absolute time constraints.

An example of HMSC with a loop is given in Figure 4.4. When the loop is repeated, the MSC P is composed sequentially with itself. According to the definition of sequential composition, all the occurrences of events in P are constrained by their time constraints. In other words, the absolute constraint of an event and the relative constraint between two events are not changed in the iteration of the loop. When MSC P is repeated, all the events of sending the message *m1* should occur between 2 and 10, and all the receptions of the message should occur between 3 and 11. Between each pair of sending and reception events, there is a relative time constraint [1, 2]. If the time domain is dense, then P can be repeated infinitely and time constraints are still satisfied. However, this results in zeno behavior. If the time domain is discrete, then P cannot even be repeated infinitely without violating the time constraints. We are more interested in discrete time domain, because it

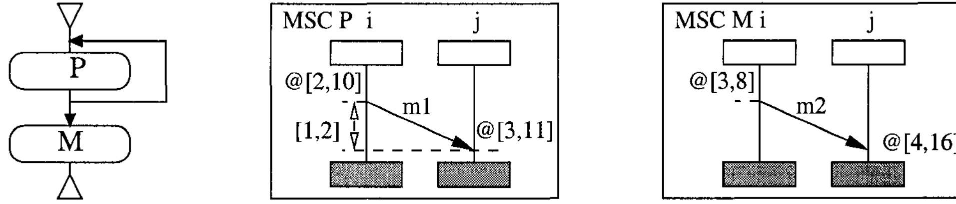


Figure 4.4: Iteration and absolute time constraints

is more practical and it is the cause of more inconsistencies. In the following, we choose non-negative integers as time domain. A time constraint can be written in one of the two formats, [a, b] or [a, ∞). For example, (0, ∞) can be written as [1, ∞) in our time domain.

The semantics of an HMSC is a set of lposets. We could define the time consistency of the HMSC in terms of the consistency of these lposets directly. However, to facilitate the development of algorithms, we define it based on the consistency of paths in the HMSC. As defined in Chapter 3, an HMSC is a directed graph (*S, E, L*). A path in an HMSC is a finite or infinite sequence of nodes $s_0 s_1 \ldots s_n \ldots$, in which s_0 is the start node, $(s_i, s_{i+1}) \in E$, $i \geq 0$. Since an HMSC can be translated to an automaton as shown in Chapter 3, a path represents a string (or a prefix of the string) accepted by the automaton. On the other hand, we transform the automaton to a regular expression, and obtain the set of lposets of the HMSC by enumerating all the strings defined by the expression. Therefore, each path corresponds to an lposet, which is the sequential composition of bMSCs referred by nodes along the path. An lposet may correspond to more than one path.

Definition 4.3 *A path is consistent if and only if the corresponding lposet, obtained by composing sequentially all the bMSCs in the path, has a trace.*

We distinguish between two notions of time consistency for HMSCs, the strong consistency and the weak consistency. An MSC specifies a set of scenarios (paths) to be implemented in the design specification. Sometimes, all the scenarios are seen as mandatory and have to be implemented in the design specification. Sometimes, the MSC specification is

seen as a set of possible behaviors to be implemented and implementing only one of them is sufficient. In the first case, the design specification has to be equivalent to the MSC specification in terms of behaviors, while in the second case it only represents a subset of what is allowed in the MSC. When all the paths specified in the MSC have to be implemented, we have to make sure that all of them are consistent, and therefore the MSC is required to be strongly consistent. In the second case, checking for weak consistency before design is sufficient.

Definition 4.4 *An HMSC H is strongly consistent if and only if every path in H is time consistent.*

Definition 4.5 *An HMSC H is weakly consistent if and only if at least one path in H is time consistent.*

Of course, strong consistency implies weak consistency and not the other way around. In the case of bMSCs, the concept of weak consistency coincides with strong consistency. We say an HMSC is inconsistent if and only if it is not weakly consistent.

In an HMSC, a loop generates infinite number of paths. Absolute time constraints in the loop affect the consistency of these paths. For example, in Figure 4.4, if MSC P does not contain absolute time constraints, then process i can send message *m1* an infinite number of times and the HMSC describes an infinite number of paths *PM, PPM, PPPM*, etc. However, the absolute time constraint @[2, 10] restricts process i to send message *m1* only 9 times (because our time domain is fixed to non-negative integers). Then all the paths $P^iM(i > 9)$ are not consistent. Moreover, the consistency of such a path may be affected also by absolute time constraints outside the loop. For example, due to the absolute time constraint in MSC M in Figure 4.4, if the message *m2* is sent, it must be sent within time [3, 8]. Since sending message *m1* has to occur before sending *m2*, *m1* can only be sent 6 times at most. So, all the paths $P^iM(i > 6)$ are not consistent. Thus, the HMSC is only

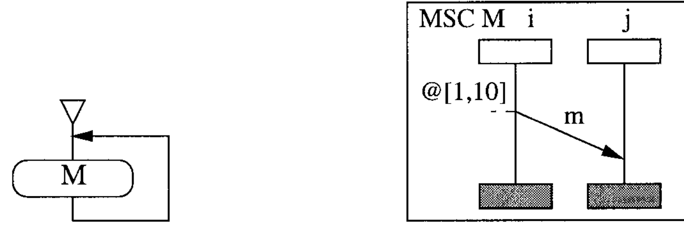


Figure 4.5: An inconsistent HMSC with a consistent simple path

weakly consistent.

Since an HMSC may contain an infinite number of paths, it is impossible to check all the paths one by one. We check the (strong, weak) consistency of an HMSC by checking its simple paths. A simple path is a sequence of nodes that begins from the start node and ends with an end node or a loop. It is actually a prefix of a path. There are no identical nodes in a simple path. The definition of simple path is similar to the sequential component defined in [10].

Definition 4.6 *Let H = (S, E, L) be an HMSC, a simple path is a sequence of nodes* s_0, $\dots s_n$, *(*$n > 0$*) in which*

- s_0 *is the start node,* $s_i \neq s_j$ *if* $i \neq j$, *and* $(s_i, s_{i+1}) \in E$,
- *either* s_n *is an end node (the simple path ending with an end node), or there is a node* s_j *such that* $(s_n, s_j) \in E$, $s_j \in \{s_0, \dots s_n\}$ *(the simple path ending with a loop).*

In an HMSC, if all the simple paths are inconsistent, then the HMSC is inconsistent. Otherwise, there is a consistent path whose prefix is an inconsistent simple path. However, if an HMSC is inconsistent, we cannot conclude that there are no consistent simple paths in the HMSC. For example, the HMSC in Figure 4.5 has only one path, which repeats MSC L infinitely. This HMSC is inconsistent. After MSC M is executed 10 times, the absolute time constraint of sending message m will be violated. However, the simple path M is consistent.

The following theorem states the necessary and sufficient conditions for an HMSC to be weakly consistent.

Theorem 4.1 *An HMSC is weakly consistent, if and only if*

- *there is at least one consistent simple path ending with an end node, or*
- *there is at least one consistent simple path ending with a loop, in which the upper bounds of absolute time constraints are infinite for all the events.*

PROOF. This theorem is proven in Appendix B.2. □

Intuitively, if the upper bounds of all the absolute time constraints are infinite in a consistent bMSC, then events in the bMSC can be repeated infinitely without violating their absolute and relative time constraint. For example, the simple path in the HMSC in Figure 4.5 is consistent but ends with a loop. In this loop, the upper bound of the sending event is not infinite. The second condition of Theorem 4.1 is not satisfied. So the HMSC is not weakly consistent.

For an HMSC to be strongly consistent, all the events in every loop must have infinite upper bounds in their absolute time constraints, so that there always exist time points for the events when the loop is repeated. Moreover, in a bMSC that appears after a loop, if an event e occurs causally after an event f in the loop, then the occurrence time of e has to be later than the occurrence time of f. So the upper bound of the absolute time constraint for e has to be infinite also. Due to the propagation of time constraints, this requirement affects the time constraints of other events. For example, there may exist another event e' causally ordered before e in the same bMSC (e' does not have an order with f). If the absolute time constraint of e' and the relative time constraint between e' and e do not have infinite upper bounds, then we may not find an occurrence time of e' for any occurrence time of e. Taking

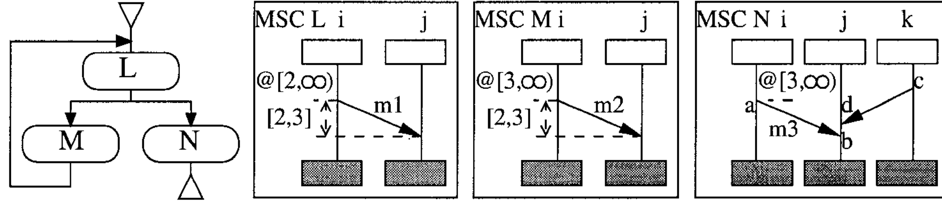


Figure 4.6: Strong consistency of an HMSC

this into account, we have to require that the reduced absolute time constraint of e has an infinite upper bound. This sets the requirement for other events implicitly. The following proposition states the conditions for strong consistency.

Theorem 4.2 *An HMSC is strongly consistent, if and only if*

- *all the simple paths are consistent, and*
- *every event in a loop or causally ordered after an event in a loop has an infinite upper bound in its reduced absolute time constraint.*

PROOF. This theorem is proven in Appendix B.3. □

The second condition of Theorem 4.2 ensures that a loop can be repeated infinitely and consistently. For example, in Figure 4.6, the HMSC has a loop *LM*, in which all the events have an infinite upper bound in their absolute time constraints. In any pass of the loop *LM*, for each event, we can always find an occurrence time within its absolute time constraint, which is larger than its occurrence time in the previous passes. Therefore, the loop can be repeated infinitely.

Moreover, the upper bounds of the reduced time constraints of events a, b, and d in MSC *N* have to be infinite to keep the strong consistency of the HMSC. For instance, assume the absolute time constraint of event a is [3, 5] instead of $[3, \infty)$. In the path *LMLMLN*, when

the MSC L is executed the third time, sending message *m1* occurs at time 6 at earliest, but the event a has to occur before time 6. This causes the inconsistency of the path *LMLMLN*. Furthermore, to keep the upper bound infinite for the reduced absolute time constraint of event d, either the upper bound has to be infinite for the absolute time constraint of c, or the relative time constraint between c and d must have an infinite upper bound.

4.4 Algorithms for Checking Time Consistency in General

To check the consistency of an HMSC according to Theorem 4.1 and 4.2, we need to check the consistency of simple paths, and upper bounds of absolute time constraints. This can be achieved in the following two steps.

- Step 1. Traverse the HMSC to determine all the simple paths. For each simple path, we compose all bMSCs in the simple path sequentially to obtain one bMSC (one lposet). Then, we check the consistency of the bMSC. If a simple path ends with a loop, we check if all the reduced absolute time constraints in the loop have infinite upper bounds. At the end of this step, we can decide if the HMSC is inconsistent or weakly consistent. However, to decide if it is strongly consistent, we need to have all the simple paths consistent and proceed with the next step.
- Step 2. Determine all the strongly connected components of the HMSC. We consider only those components including more than one nodes, or one node with a self-loop. Such a component forms a loop. Then for each component C that forms a loop, we check all the nodes that can be reached from C according to the second condition of Proposition 4.2 (Notice that, absolute time constraints inside a loop have already been checked in step 1). If the condition is satisfied, then the HMSC is strongly consistent.

The detailed algorithm is as follows. In this algorithm, we use *ubr* to represent the upper bound of the reduced absolute time constraint of an event.

Algorithm: CheckHMSC

```
result = checkHMSC_step1();
if result == weak_or_strong then
   use depth-first-search to find all the strongly connected components;
   for each component C that forms a loop do
      if checkHMSC_step2(C) == False then
         return(Weak_consistency);
      end if
   end for
   return(Strong_consistency);
else
   return(result);
end if
```

Algorithm: checkHMSC_step1

```
curr_path = s0; {s0 is the start node}
while curr_path is not empty do
   s = the last node in curr_path;
   if s does not have a new child node then
      if s is an end node then
         if curr_path is consistent then
            cons_path = Exist;
         else
            incons_path = Exist;
         end if
      end if
      delete s from curr_path;
   else
```

$s' =$ a child node of s;

if s' is not in curr_path **then**

append s' to curr_path;

else if curr_path is consistent and there are no reduced absolute time constraints with finite upper bounds in nodes from s' to the last node in curr_path **then**

cons_path = Exist;

else

incons_path = Exist;

end if

end if

end while

if (incons_path == Exist) && (cons_path == Exist) **then**

return(Weak_consistency);

else if incons_path == Exist **then**

return(Inconsistent);

else

return(weak_or_strong);

end if

Algorithm: checkHMSC_step2(C)

for each node s' such that $s' \notin C$ and there is an edge $s \to s'$ $(s \in C)$ **do**

for each event e_i in s' such that e_i is the first event in a process that also appears in C **do**

if $ubr_i \neq \infty$ **then**

return(False);

else

for each event e_j such that $e_i \leq e_j$ **do**

if $ubr_j \neq \infty$ **then**

```
            return(False);
          end if
        end for
      end if
    end for
    add s' to C;
  end for
  return(True);
```

In the aforementioned algorithm, *checkHMSC_step2* checks the time constraints in each node that can be reached from a strongly connected component. In each node, in the worst case, every event has to be compared with other events to check if they are ordered. Therefore, the complexity of this algorithm is $O(nm^2)$, where n is the number of bMSCs and m is the number of events in a bMSC. Since the number of strongly connected components is not larger than the number of nodes in an HMSC, *checkHMSC_step2* is repeated at most n times.

In the algorithm *checkHMSC_step1*, we check the consistency of all the simple paths, and absolute time constraints in each loop. However, in the worst case, the numbers of simple paths and loops are exponential with respect to the number of nodes. Moreover, even if two paths have some common bMSCs, the consistency of these bMSCs obtained from checking the first path is not reusable in checking the consistency of the second path because of the absolute time constraints. For example, if we already know that *MN* and *PM* are consistent, we cannot conclude that *PMN* is consistent. This is the case for instance for the bMSCs *P*, *M* and *N* given in Figure 4.7. The path *PMN* does not have a trace because of the absolute time constraints of events a, b and c. So when we traverse an HMSC, simple paths have to be checked one by one, and some nodes are visited several times if they appear in several simple paths. This causes the exponential complexity of the algorithm.

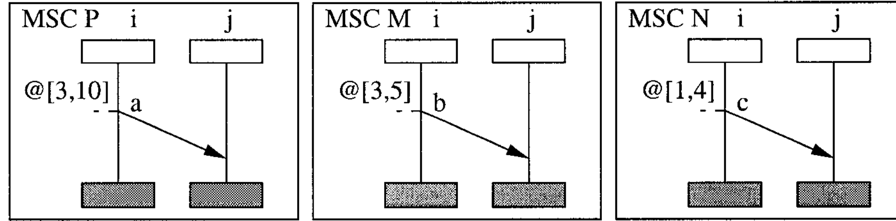


Figure 4.7: bMSCs in an inconsistent path PMN

4.5 Algorithms for Checking Time Consistency in Specific Cases

In this section, we investigate specific cases of HMSCs on which the consistency can be determined with less complex algorithms. For that specific set of HMSCs, we require that all bMSCs have the same set of processes, and there is at least one event in each process in each bMSC. We refer to this kind of HMSCs as Fixed Set of Processes HMSCs, or FSPHMSCs.

We identify subsets of FSPHMSCs by specifying relations between absolute time constraints in different bMSCs included in the FSPHMSCs. Absolute time constraints are time intervals, which could be disjoint or overlapping with each other. We define time-disjoint HMSCs, in which absolute time constraints in the same process but in different bMSCs along a path are disjoint. However, this requirement is so strong that a large number of HMSCs are excluded. By allowing overlapping, we define lower-bound-later and upper-bound-later HMSCs to include more HMSCs.

To define these specific HMSCs formally and determine their consistency in an efficient manner, we first establish in the following propositions the conditions under which the sequential composition of two bMSCs M and N is consistent, and reduced absolute time constraints in M are not affected by those in N and vice versa.

Proposition 4.1 *For two bMSCs M and N, let $[a_i, b_i]$ be the reduced absolute time con-*

straint of the last event in a process p_i in M, and $[c_i, d_i]$ be the reduced absolute time constraint of the first event in the same process p_i in N. The sequential composition $M \cdot N$ is consistent if and only if M and N are consistent, and $a_i < d_i$.

PROOF. This proposition is proven in Appendix B.4. □

Theorem 4.3 *For two consistent bMSCs M and N, let $[a_i, b_i]$ be the reduced absolute time constraint of the last event in a process p_i in M, and $[c_i, d_i]$ be the reduced absolute time constraint of the first event in the same process p_i in N,*

- *$M \cdot N$ is consistent, and the lower bounds of reduced absolute time constraints in M and N are not changed in $M \cdot N$ if and only if $a_i < c_i$.*
- *$M \cdot N$ is consistent, and the upper bounds of reduced absolute time constraints in M and N are not changed in $M \cdot N$ if and only if $b_i < d_i$, or b_i and d_i are both infinity.*

PROOF. The proof of this theorem is given in Appendix B.5. □

For example, the two bMSCs in Figure 4.2 are consistent. The reduced absolute time constraints for events a and b in MSC M are @[5, 7] and @[6, 8] respectively. The reduced absolute time constraints for events c and d in MSC N are @[2, 3] and @[4, 5] respectively. According to Proposition 4.1, $N \cdot M$ is consistent, but not $M \cdot N$. Moreover, the reduced absolute time constraints satisfy the conditions in Theorem 4.3. So in $N \cdot M$, the reduced absolute time constraints of these events are still @[5, 7], @[6, 8], @[2, 3] and @[4, 5].

Now we define special subsets of FSPHMSCs in which bMSCs satisfy the conditions in Theorem 4.3. For that, we first define the later-than relation between bMSCs and then time-disjoint HMSCs.

Definition 4.7 *A bMSC N is later than a bMSC M if in each common process p of M and N, the lower bound of the absolute time constraint for the first event in N is larger than the upper bound of the absolute time constraint for the last event in M.*

In Figure 4.2, the bMSC M is later than the bMSC N. Actually given any consistent bMSCs P and Q, if Q is later than P, then reduced absolute time constraints in P and Q must satisfy the conditions in Theorem 4.3, since reduced absolute time constraints are intervals within the original absolute time constraints. So $P \cdot Q$ must be consistent, and reduced absolute time constraints for events in $P \cdot Q$ are the same as reduced absolute time constraints in P and Q.

Definition 4.8 *An FSPHMSC is time-disjoint if in each simple path* $s_0 \ldots s_n$*,* s_{i+1} *is later than* s_i*,* $0 \leq i < n$*.*

In a simple path of a time-disjoint FSPHMSC, if s_0 and s_1 are consistent, then $s_0 \cdot s_1$ is consistent because s_1 is later than s_0. Moreover, the reduced absolute time constraints in $s_0 \cdot s_1$ are same as those in s_0 and s_1. Thus, if s_2 is consistent, then $(s_0 \cdot s_1) \cdot s_2$ is also consistent because s_2 is later than s_1, and so forth. Therefore, we can decide about the consistency of a simple path from the consistency of bMSCs in the simple path. The consistency of all the simple paths can be decided from the consistency of each bMSC as stated in the following proposition.

Proposition 4.2 *All the simple paths in a time-disjoint FSPHMSC are consistent if and only if all the bMSCs contained in the FSPHMSC are consistent.*

PROOF. This proposition is proven in Appendix B.6. □

It is worth to note that in general, for consistent bMSCs P, Q and O such that Q is

later than P, requiring the later-than relation only between O and Q cannot guarantee that $P \cdot Q \cdot O$ is consistent. The reason is that O and P may have some common processes that are not contained by Q. We have to require O to be later than both P and Q. However, in an FSPHMSC, all the bMSCs have the same set of processes. So we only need O to be later than Q.

Based on Proposition 4.2, we can develop an algorithm to check if a time-disjoint FSPHMSC is strongly consistent.

Algorithm: checkStrongConsistency

for each bMSC s_i in a time-disjoint FSPHMSC **do**
  if s_i is not consistent **then**
    return(Not_Strong_Consistency);
  end if
end for
use depth-first-search to find all the strongly connected components;
for each component C that forms a loop **do**
  if there is a reduced absolute time constraint with finite upper bound in C **then**
    return(Not_Strong_Consistency);
    if checkHMSC_step2(C) == False **then**
      return(Not_Strong_Consistency);
    end if
  end if
end for
return(Strong_Consistency)

The first loop in the algorithm *checkStrongConsistency* checks the consistency for each bMSCs, which can be done in polynomial time as discussed in Section 4.2. In the second loop, we check the absolute time constraints of events in each strongly connected component

C that forms a loop, and those of events that are causally ordered after events in C. This can be done in polynomial time also as discussed in Section 4.4. So the strong consistency of a time-disjoint FSPHMSC can be checked in polynomial time.

In the algorithm *checkStrongConsistency*, due to the properties of the time-disjoint FSPHMSC, we do not need to check the consistency of each simple path one by one. Unfortunately, the properties do not help when checking the weak consistency. To check the weak consistency, for each simple path ending with a loop, we have to check if all the absolute time constraints in the loop have infinite upper bounds according to the second condition of Theorem 4.1. It is not enough to check the maximal loops only. For example, the HMSC in Figure 4.8(a) has two simple paths, OQ and OPQ, which are also loops. Assume all the absolute time constraints in O and Q have infinite upper bounds, and there is an absolute time constraint in P which has a finite upper bound. Then OQ could be repeated infinitely, but not OPQ. We have to check all the absolute time constraints in both OQ and OPQ although OQ is included in OPQ. This requires us to go through each simple path so that each loop can be checked.

The conditions of time-disjoint HMSC are strong conditions and exclude a large number of HMSCs. In most cases, a time-disjoint HMSC will not be strongly consistent. For instance, the HMSC in Figure 4.8(b), in which the bMSCs M and N are from Figure 4.2, is a time-disjoint FSPHMSC. Since M has to be later than N, absolute time constraints in N cannot be infinite. Then the HMSC cannot be strongly consistent according to Proposition 4.2.

We observe that in a time-disjoint HMSC, two bMSCs in a simple path satisfy both conditions in Theorem 4.3. Actually we can define a kind of HMSCs that only satisfy one of the condition. We define the lower-bound-later relation and the upper-bound-later relation between bMSCs first.

Definition 4.9 *A bMSC N is lower-bound-later than a bMSC M if in each common process*

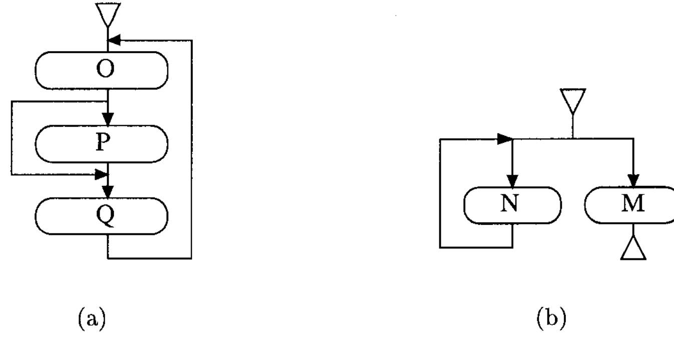


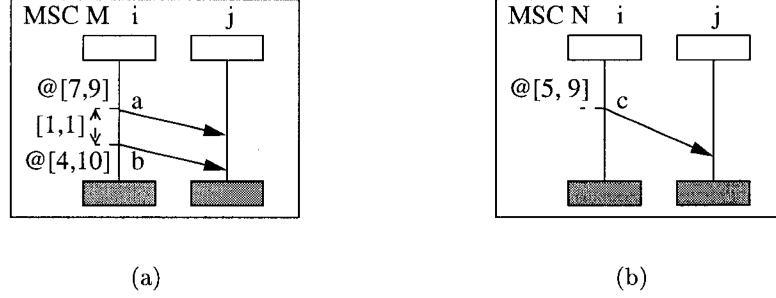
Figure 4.8: Weak and strong consistency for time-disjoint HMSCs

p of M and N, the lower bound of the reduced absolute time constraint for the first event in N is larger than the lower bound of the reduced absolute time constraint for the last event in M.

A bMSC N is upper-bound-later than a bMSC M if in each common process p of M and N, the upper bound of the reduced absolute time constraint for the first event in N is larger than the upper bound of the reduced absolute time constraint for the last event in M, or both upper bounds are infinity.

In contrast with the later-than relation, we have to define these two relations on reduced absolute time constraints so that one of the conditions in Theorem 4.3 can be satisfied. So the lower-bound-later and the upper-bound-later relation are meaningful for consistent bMSCs only. The later-than relation between two consistent bMSCs is a lower-bound-later and an upper-bound-later relation. If the lower bound of an absolute time constraint is larger than the lower bound of another absolute time constraint, their reduced absolute time constraints may not satisfy this relation. For example, in Figure 4.9, the lower bound of the absolute time constraint for event c is larger than the lower bound of the absolute time constraints for event b. However, we cannot say N is lower-bound-later than M, because the reduced absolute time constraint for event b is @[8, 10].

Definition 4.10 *An FSPHMSC is lower-bound-later (upper-bound-later) if for each simple*



(a) (b)

Figure 4.9: Relation between lower bounds

path, in its maximal prefix $s_0 \dots s_n$ *such that all the bMSCs are consistent,* s_{i+1} *is lower-bound-later (upper-bound-later) than* s_i $(0 \leq i < n)$.

The HMSC in Figure 4.4 is a lower-bound-later FSPHMSC. The HMSC in Figure 4.6 is an upper-bound-later FSPHMSC if the MSC N did not contain the process k.

Similarly to a time-disjoint FSPHMSC, in a lower-bound-later (or upper-bound-later) FSPHMSC, if a bMSC Q is lower-bound-later (or upper-bound-later) than a bMSC P, and another bMSC O is lower-bound-later (or upper-bound-later) than Q, then $P \cdot Q \cdot O$ is consistent according to Theorem 4.3. The following proposition is valid and the algorithm *checkStrongConsistency* can still be used on lower-bound-later and upper-bound-later FSPHMSCs.

Proposition 4.3 *All the simple paths in a lower-bound-later or upper-bound-later FSPHMSC are consistent if and only if all the bMSCs contained in the FSPHMSC are consistent.*

PROOF. Same as the proof of Proposition 4.2, except we replace the later-than relation with the lower-bound-later and upper-bound-later relation. □

The last question is how to decide if an FSPHMSC is time-disjoint, lower-bound-later or upper-bound-later. According to the definitions, we only need to check if two adjacent

bMSCs in simple paths satisfy these relations. The algorithm for checking if an FSPHMSC is time-disjoint is given below. It performs a depth-first-search [17] on an FSPHMSC. An array *color* is used to indicate the status of each node. A white node means that the node has not been visited. A gray node means it is being visited. A black node means it has been visited. If two adjacent bMSCs in a simple path do not satisfy the later-than relation, we set the *mark* as NS.

Algorithm: decideTDHMSC

curr_path = s_0;

mark = S;

for each node s_i in the HMSC **do**

  color[s_i] = white;

end for

visitProc(s_0, *later-than*);

if mark == NS **then**

  return(no);

else

  return(yes);

end if

Algorithm: visitProc(s, Relation)

color[s] = gray;

for each child node t of s **do**

  if color[t] == white **then**

    append t to curr_path;

    visitProc(t, *Relation*);

  else if color[t] == black **then**

    if s does not have the *Relation* with t **then**

      mark = NS;

end if

else

if s has more than one incoming edge and s does not have the *Relation* with t **then**

mark = NS;

end if

end if

end for

color[s] = black;

delete s from curr_path;

r = the last node in curr_path;

if r does not have the *Relation* with s **then**

mark = NS;

end if

The algorithms for deciding if an HMSC is lower-bound-later or upper-bound-later are similar to *decideTDHMSC*, except that only consistent bMSCs need to be checked.

Algorithm: decideBLHMSC

for each node s_i in the HMSC **do**

if the bMSC referred by s_i is inconsistent **then**

delete s_i;

else

color[s_i] = white;

end if

end for

if all the nodes are deleted **then**

mark = NS;

else

curr_path = s_0;

mark = S;

visitProc(s_0, *lower-bound-later*); {or visitProc(s_0, *upper-bound-later*);}

end if

if mark == NS **then**

return(no);

else

return(yes);

end if

4.6 Related work

Time consistency of MSCs has been partially investigated in [7, 10, 68, 69]. In [7], the authors use time constraints on pairs of events. Three types of design problems in timed MSCs are defined. They are timing inconsistency, visual conflicts, and timing conflicts. Timing inconsistency and timing conflicts correspond to the inconsistency between time constraints in our case. Visual conflicts correspond to the inconsistency between the causal order and the temporal order. These problems are checked by computing negative cost cycles and shortest distances in weighted graphs translated from MSCs. The consistency of HMSCs is not considered in [7].

In [10], besides the timer events in MSC, timing delay intervals are also used to express time constraints. To check the consistency of a bMSC, a temporal constraint graph is constructed from the bMSC and then checked if it has negative cost cycles. For HMSCs, timing consistency and partially timing consistency are defined. However, only sufficient conditions for timing consistency are given. To check the consistency of an HMSC, all the simple paths need to be found first, then checked one by one.

In [68, 69], one more construct, which is timing marks, is used for describing more general time requirements. A timing mark is a boolean expression in the form $a \leq \sum_i c_i(t_i - t'_i) \leq b$,

where t_i and t'_i are occurrence times of two events e_i and e'_i, a, b and c_i are real numbers. Linear programming techniques are used to check the consistency of bMSCs. In HMSCs, the strong sequencing semantics is used to connect bMSCs, which means that all the events in the second bMSC have to occur after all the events in the first bMSC. An algorithm is developed to check the consistency of HMSCs by going through all the simple paths.

We apply the solution of the simple temporal problem to check the consistency of bMSCs. The algorithm is also used in [7, 10]. In HMSCs. we use the weak sequencing semantics, which makes the problem more challenging. We develop the sufficient and necessary conditions for the consistency of HMSCs. We identify a subset of HMSCs in which consistency can be checked efficiently.

4.7 Conclusion

As a specification language, MSC can be used to specify real-time systems with quantified time requirements. In this chapter, we considered an important aspect of MSC-2000 specifications, namely the time consistency. Both relative and absolute time constraints introduced in MSC-2000 have been taken into account. Based on our semantics for timed MSC, we defined the time consistency for bMSCs, and the strong and weak consistency for HMSCs. We have also developed and discussed algorithms to check the consistency of HMSCs in the general case as well as for a special set of HMSCs.

We have made some assumptions about time constraints. We excluded relative time constraints between causally independent events. The considerations were the level of abstraction of such constraints and the implementability of such MSCs. Another important consideration is the complexity of checking consistency of MSCs with such constraints. If there is a relative time constraint between two causally independent events in a bMSC for instance, checking its consistency becomes a general temporal constraint satisfaction problem as defined in [19], which is NP-hard.

We assumed that in an HMSC, relative time constraints between events occurring in two sequential bMSCs are $(0, \infty)$ by default. The results in this chapter can be extended to handle the situation where these relative time constraints are not $(0, \infty)$. Actually absolute time constraints have the same nature as these relative time constraints. An absolute time constraint of an event e can be considered as a relative time constraint between e and a special event that occurs at the beginning of the HMSC. So we can handle these relative time constraints similarly to the absolute time constraints. Specifically, for weak consistency, we require further that in a simple path ending with a loop, all the relative time constraints between events outside and inside a loop have infinite upper bounds. For strong consistency, we require the all the relative time constraints between events outside a loop and events inside the loop (or causally ordered after events inside the loop) have infinite upper bounds. Of course these conditions need to be proved.

Another implicit assumption we have made is that the environment behaves correctly. If a process expects to receive a message from the environment within a time interval, the environment is assumed to send it and make the time constraint satisfied. If the environment is considered as uncontrollable, we can remove all the time constraints related to the environment, replace them with $(0, \infty)$ and check the consistency in the same way.

Chapter 5

Refining MSC Specifications

5.1 Introduction

In traditional development processes, designers take as input a requirement specification and develop in one big step a design specification. This design specification is then validated against the requirement specification. When the design specification does not satisfy the requirements, it is reworked out. The design and validation activities are repeated until the design satisfies the requirements.

An alternative to this approach is stepwise refinement. Indeed, the requirement specification can be taken as input and enriched step by step. The specification can be replaced by a refined version such that the system described by the specification still fits its environment. The refinement approach relates specifications in different levels of detail. The refinement relation could be equivalences, such as bisimulation equivalences in [9, 72, 75, 80, 106], or different preorders as in [56, 61, 70, 99]. In this approach, small design steps instead of a complete design activity are validated.

In this chapter, we develop a refinement approach for timed MSC specifications. When

refining a timed MSC, we add more details about the behaviors as well as the architecture of the system. Moreover, time constraints can also be refined. Most of existing refinements of real time systems keep the time constraints unchanged [9, 70, 72, 80, 99, 106]. In our opinion, a refinement approach should allow for strengthening time constraints on the system and relaxing assumptions on the environment. In this way, the behavior of the system can still be accepted by the environment, or even other environments with weaker assumptions.

During the refinement, the resulting MSCs should conform to the original MSCs. Specifically, we require events and their causal orders to be preserved. Moreover, if the original MSC specification is time consistent, the resulting specification should be consistent also. We define this relation formally as a preorder between lposets. This relation generalizes the one in [56]. In the chapter, we still focus on bMSCs, and HMSCs that contain bMSCs only. However, the relation can be extended to more general MSCs.

The rest of this chapter is organized as follows. In Section 5.2 and Section 5.3, we introduce approaches of refining bMSCs and HMSCs respectively. We also introduce refinement relations between MSCs and algorithms for checking these relations. In Section 5.4, we give an example to demonstrate the refinement of HMSCs. In Section 5.5, we discuss related work prior to concluding in Section 5.6.

5.2 Refining bMSCs

5.2.1 Refinement Approach

A bMSC describes communications between processes. To refine it into a detailed level, we can

- decompose one or several processes according to the architecture of the system,
- add new messages to refine the behaviors of processes, and

- add or change time constraints to refine the performance constraints.

These refinements can be performed alternatively many times, until the desired level of details is reached. We refer to the first two refinements as vertical refinement and horizontal refinement, respectively. They are introduced in [56]. We focus on the third refinement, where we allow time constraints to be added first, and then refined.

Adding Time Constraints

In a certain phase of design, designers need to add performance requirements to the functional description of a system. Examples of these requirements include the delay between two events, or the time when a process must send a message. We can use absolute and relative time constraints in MSC to specify these requirements.

For example, the bMSC in Figure 5.1(a) describes a use case of a door controller [38]. A user inserts his card and enters a PIN, then the Door Controller (DC) opens the door and returns the card. In a vertical refinement, DC is decomposed into an Access Point (AP) and an Authorizer (*auth*) as shown in Figure 5.1(b). In a following horizontal refinement, we specify the interactions between AP and *auth*. The AP sends the card ID and PIN to *auth* for authorization. If they are valid, *auth* sends back *approval*. Notice that in the bMSC *access2* we are only considering the partial behavior of authorization. After the horizontal refinement, we add timing requirements, such as AP has to return the card within 10 time units after the user enters a PIN, or *auth* has to response AP with a delay between 1 and 8 time units. They are represented by the relative time constraints shown in Figure 5.1(b).

In our refinement approach, we allow relative time constraints to be added between causally ordered events only. As already explained in Chapter 4, relative time constraints between un-ordered events cannot be guaranteed in a distributed system without further enrichments of the specification. In Figure 5.1(b), if we add a relative time constraint [1, 2] between event a (returning card) in AP or event b (opening door) in *door*, then this

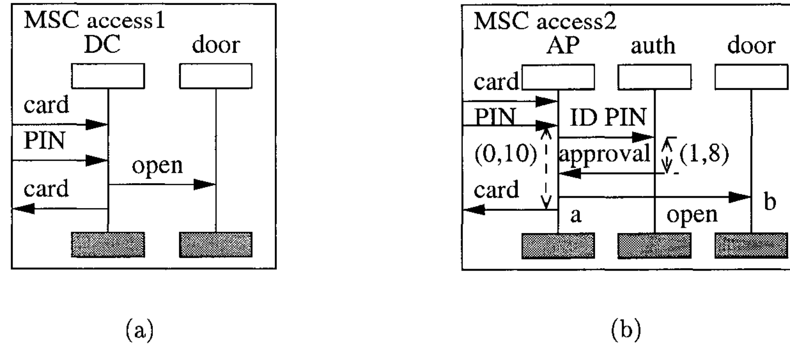


Figure 5.1: Refining a door controller

time requirement cannot be satisfied without adding other message exchanges between the processes to relate these events. We can check the existence of this kind of time constraints by checking if events related by the time constraints are causally ordered in the lposet representing the MSC. If this kind of time constraints appears, we notify designers to remove them or to make events causally ordered by adding messages.

Designers may not add time constraints for all the events in a bMSC in one refinement. So some events may not be constrained by any time constraints. In the semantics, we consider these events are associated with default time constraints that are the whole time domain minus the least element. For example, if the time domain is nonnegative real numbers, then the default time constraints are $(0, \infty)$. We reserve the absolute time constraint @[0] for a special event (the beginning of the world). A relative time constraint cannot be 0 because we allow for relative time constraints to appear between causally ordered events only, and these ordered events cannot occur at the same time. Semantically, adding time constraints can be considered as changing these default time constraints to others.

Refining Time Constraints

When adding for the first time a time constraint, a designer may not know how tight the constraint should be. He/She should be allowed to change this time constraint later on.

For example, for the bMSC in Figure 5.1(b), the designer may find that the delay before AP returns the card (the relative time constraint (0, 10)) is too long. The designer may reduce this time constraint to (0, 8), for instance.

To define rules of changing time constraints, we first consider the effects of time constraints on a system and its environment. In our point of view, time constraints specify requirements/constraints on a process in the system and assumptions on its environment. The environment of a process consists of all the processes in communication with the process within and outside the system under consideration. Adding an absolute time constraint on a sending event, adds a time requirement/constraint to the process executing the event. For instance, if the time constraint is @[3, 5], then the process has to send the corresponding message at any time between time 3 and 5. If the event is a reception event, the time constraint not only adds a requirement/constraint to the process, but also implies an assumption on the environment. For example, if a time constraint is @[3, 5] for a reception event, then the process has to consume a message from its channels at any time between 3 and 5. To ensure this, the environment has to send this message before time 5.

Similarly, relative time constraints also specify requirements and assumptions. We consider that a relative time constraint between two ordered events is associated with the second event. It constrains the occurrence of the second event, because it specifies how long after the occurrence of the first event, the second event can occur. If the second event is a sending event, the relative time constraint states a requirement for a delay on the process executing the event. The process has to send a message within the delay specified by the time constraint. If the second event is a reception event, the relative time constraint specifies a delay that the process has to wait for a message. In another word, the environment is expected to deliver the message within the time constraint. So relative time constraints on reception events specify assumptions on the environment.

For the refinement of time constraints, we require that constraints on a process become stronger, while assumptions on the environment become weaker. Specifically, we define the

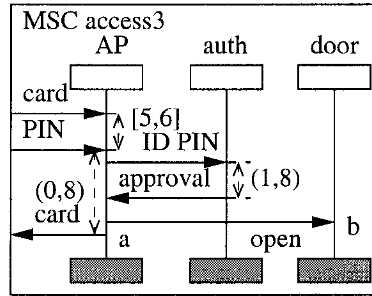


Figure 5.2: Refining a door controller further

following rules.

- If an event is a sending event, the range of the absolute or relative time constraint associated to it can be reduced. The process executing the event is more constrained.
- If an event is a reception event, the range of the absolute or relative time constraint associated with it can be increased. This makes the assumptions on its environment weaker. (Note that the time constraint could be enlarged to $(0, \infty)$, which means the event is not constrained any more.)

With such rules, a refined process can still fit in the same environment. For example, in Figure 5.1(b), the delay before AP returns the card is (0, 10). The user expects to get his card within 10 time units after entering PIN. If we refine the time constraint to (0, 8) as shown in Figure 5.2, the user may get the card sooner and his expectation is still met. A time refinement adds more constraints to the system, and reduces its non-determinism. On the other hand, assumptions on the environment can be relaxed by a refinement. For example, in Figure 5.2, the relative time constraint [5, 6] specifies a delay between receiving card and PIN in AP. The user has to enter PIN within the time constraint. We may change it to [4, 7] in a refinement to relax the assumptions on the user.

Refining time constraints is to reduce or enlarge time constraints that are not $(0, \infty)$. We distinguish between adding and refining time constraints, because the former always adds more constraints to a system and more assumptions to its environment, while the later may

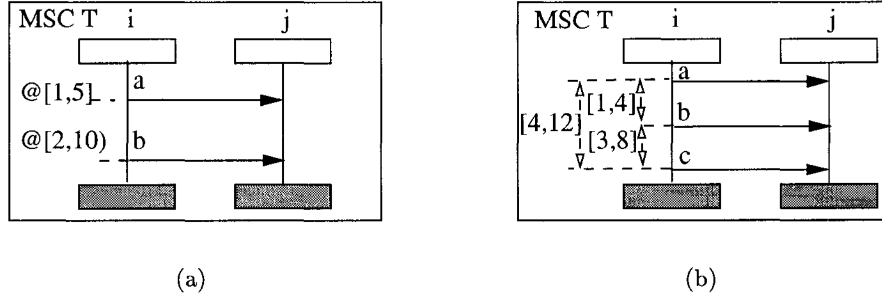
relax the assumptions to the environment.

5.2.2 Refinement Relation of bMSCs

Given the refinement approach, we need to define formally what is the refinement of an MSC. In other words, we need to define a relation between MSCs. If two MSCs satisfy this relation, then we say one is a refinement of another, or one conforms to another. We use refinement and conformance interchangeably. Although we have defined three kinds of refinement notations (vertical refinement, horizontal refinement, and time constraint refinement), they actually represent three aspects (architecture, behavior and performance) of a complete refinement notation. So we define only one refinement relation including all these aspects.

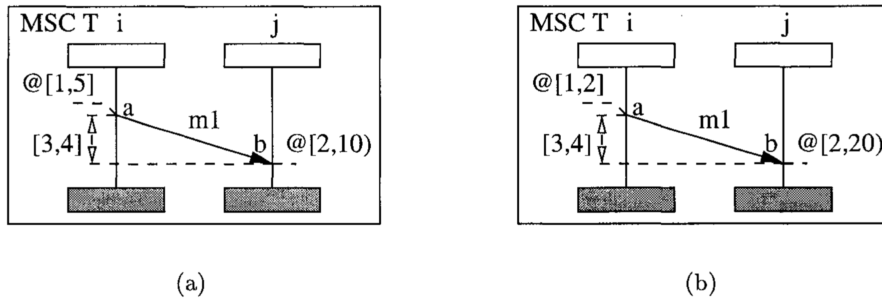
After an MSC is refined, the resulting MSC contains more processes and more events than the previous MSC. Apparently, this relation can be represented using set inclusions. However, we need to take additional considerations to time constraints. First, we refine time consistent bMSCs only. Inconsistent bMSCs contain semantics errors. They cannot be used for developing a system. We do not consider refining them further.

Furthermore, when adding or changing time constraints in a consistent bMSC according to the rules in Section 5.2.1, the time consistency is not guaranteed automatically in the resulting bMSC. The resulting bMSC may be inconsistent. For example, if we add a relative time constraint [10, 20] between event a and b in the MSC in Figure 5.3(a), then the MSC is inconsistent. Moreover, reducing time constraints associated with sending events may cause conflicts with causal orders, or inconsistencies with implicit time constraints determined by other time constraints. For the MSC in Figure 5.3(a), if we change the absolute time constraints of sending events a (@[1, 5]) and b (@[2, 10)) to @[4, 5] and @[2, 3] respectively, then the time constraints conflict with the causal order between event a and b. Changing relative time constraints may also cause inconsistency. The MSC in



(a) (b)

Figure 5.3: Refining time constraints



(a) (b)

Figure 5.4: Comparing time constraints

Figure 5.3(b) is consistent. However, if we change the relative time constraints [1, 4], [3, 8] and [4, 12] to [2, 4], [3, 4] and [10, 12] respectively, then the resulting MSC is not consistent.

On another hand, after a refinement, we need to compare time constraints to check if the rules defined in Section 5.2.1 are satisfied. We should compare reduced time constraints because they represent actual occurrence time or delays. For example, in Figure 5.4(a), although the absolute time constraint for event b is @[2, 10), event b cannot occur outside its reduced absolute time constraint @[4, 9]. If we change the MSC to the one in Figure 5.4(b), it seems all the time constraints in the two MSCs satisfy the relation discussed in Section 5.2.1. However, in Figure 5.4(b), event b can only occur within @[4, 6]. So the time constraint for event b is actually reduced. It does not satisfy the relation. We have to require reduced time constraints to satisfy the relation.

However, when time constraints are not consistent in the resulting MSC, we cannot check

if rules in Section 5.2.1 are satisfied, because reduced time constraints of the resulting MSC do not exist. Thus, we consider the time consistency as a condition of conformance between MSCs.

We already defined reduced absolute time constraints in Chapter 4. Here we define reduced relative time constraints similarly.

Definition 5.1 *Given a consistent bMSC and its distance graph, a reduced relative time constraint between events e and f ($e \leq f$) is a maximal interval within the original relative time constraint, such that the upper bound is the shortest distance from e to f, and the lower bound is the negative of the shortest distance from f to e.*

Within the reduced relative time constraints, each value is a delay between two events when they occur within their reduced absolute time constraints. Since the Floyd-Warshall algorithm calculates all pairs shortest paths, both reduced absolute and relative time constraints can be obtained by applying the algorithm.

In Figure 5.4(b), the relative time constraint between a and b has to be enlarged also to make the reduced time constraints satisfy the relation.

Since a bMSC can be represented by one lposet, we first define a refinement relation between lposets.

Definition 5.2 *Let $p_1 = (I_1, A_1, E_1, \leq_1, l_1, D_1, T_1)$ and $p_2 = (I_2, A_2, E_2, \leq_2, l_2, D_2, T_2)$ be two timed lposets. If p_1 is time consistent, then p_2 is a refinement of p_1 if and only if p_2 is time consistent, and there are two injective mappings $m_e : E_1 \rightarrow E_2$, and $m_a : A_1 \rightarrow A_2$ such that the following conditions are satisfied:*

- *For events $e, f \in E_1$, if $e \leq_1 f$, then $m_e(e) \leq_2 m_e(f)$.*
- *For an event $e \in E_1$, $m_a(l_1(e)) = l_2(m_e(e))$.*

- *Let D'_i (i = 1, 2) be reduced absolute time constraints. For an event $e \in E_1$ such that $D'_1(e)$ is not (0, ∞), if e is a sending event, then $D'_1(e) \supseteq D'_2(m_e(e))$; if e is a reception event, then $D'_1(e) \subseteq D'_2(m_e(e))$.*

- *Let T'_i (i = 1, 2) be reduced relative time constraints. For events $e, f \in E_1$ such that $e \leq_1 f$ and $T'_1(e, f)$ is not (0, ∞), if f is a sending event, then $T'_1(e, f) \supseteq T'_2(m_e(e), m_e(f))$; if f is a reception event, then $T'_1(e, f) \subseteq T'_2(m_e(e), m_e(f))$.*

The first condition in Definition 5.2 means the causal order between two events has to be preserved, and the second condition specifies the relation between m_e and m_a. The last two conditions specify the relation between time constraints. If a time constraint is (0, ∞), it means the event associated to the constraint is not constrained in fact. Changing a time constraint that is (0, ∞) corresponds to adding a time constraint in an MSC specification. However, once a time constraint is not (0, ∞), changing it has to obey to the rules specified in Section 5.2.1.

The refinement relation between bMSCs is defined as follows. Notice that in Definition 5.2, the set of events in an lposet could be infinite, while in Definition 5.3, a bMSC contains a finite number of events.

Definition 5.3 *A bMSC M_2 is a refinement of (or conforms to) a consistent bMSC M_1 if and only if the lposet representing M_2 is a refinement of the lposet representing M_1.*

Definition 5.3 is applicable to both timed and un-timed bMSCs, because an un-timed bMSC can also be represented by a timed lposet, in which all the time constraints are (0, ∞). An un-timed bMSC is always time consistent.

For example, the timed bMSC *access2* in Figure 5.1(b) conforms to the un-timed bMSC *access1* in Figure 5.1(a) because events and orders in *access1* are preserved in *access2* and *access2* is time consistent. The bMSC *access2* is refined further into the bMSC *access3*

in Figure 5.2. Events and orders are same in these two bMSCs and the time constraints satisfy the relation in Definition 5.2. The bMSC *access3* is also time consistent. So *access3* conforms to *access2*. However, if we have changed the time constraint (0, 10) in Figure 5.1 to (0, 15), the constraint on AP for returning the card becomes weaker. With such a system, the user may get his card back 13 units of time after entering PIN, which was not allowed previously in *access2*. The resulting bMSC would not conform to *access2*.

Our refinement relation extends the matching (conformance) relations for un-timed MSCs defined in [56, 75, 85]. The refinement relation is reflexive, but it is not transitive in general. Let us assume an absolute time constraint @[2, 5] on a reception event. Following the rules, we can enlarge it to @(0, ∞). It means that the constraint is relaxed to the maximum. In other words, there is no constraint any more for this event. Later on, a designer may see this event as an un-timed event and decide to add a constraint @[3, 4]. The time constraint @[3, 4] does not satisfy the relation with the original time constraint @[2, 5] as stated in Definition 5.2, and we cannot obtain the refinement relation with the original MSC. To ensure transitivity, we do not allow for a time constraint associated with a reception event to be enlarged to (0, ∞).

Proposition 5.1 *The refinement relation of bMSCs is reflexive. The refinement relation of bMSCs is transitive if time constraints associated with reception events are not enlarged to (0, ∞) when they are not (0, ∞).*

PROOF. This proposition is proven in Appendix B.7. □

5.2.3 Checking Conformance between bMSCs

To check if two bMSCs satisfy the refinement relation (or if one bMSC conforms to another), we check the conditions in Definition 5.2. Let a bMSC M_1 be represented by (I_1, A_1, E_1,

$\leq_1$, l_1, D_1, T_1), and another bMSC M_2 be represented by (I_2, A_2, E_2, $\leq_2$, l_2, D_2, T_2). The algorithm of checking the conformance includes the following steps.

We first check the time consistency of M_1 and M_2. As discussed before, this can be done using the Floyd-Warshall algorithm in time $O(n^3)$, where n is the number of events. At the same time, we obtain reduced time constraints.

Then we need to decide the mapping m_a between the label sets A_1 and A_2. To achieve this, we have to know which process in I_1 is vertically refined into which processes in I_2. That is, we need to have a surjective function $m_p : I_2 \rightarrow I_1$. Designers decide this mapping according to the architecture of the system. Then it is used as an input of our algorithm. For every label $a_1 = send(i_1, j_1, m)$ (or $a_1 = receive(i_1, j_1, m)$), $a_1 \in A_1$, if there is a label $a_2 = send(i_2, j_2, m)$ (or $a_2 = receive(i_2, j_2, m)$), $a_2 \in A_2$, such that $i_1 = m_p(i_2)$ and $j_1 = m_p(j_2)$, we say a_1 is mapped to a_2. If we can not find a mapping for a label in A_1, then two lposets do not conform. Since we need to compare each label A_1 with a label in A_2, the complexity of this step is $O(n^2)$.

According to m_a, we can get m_e between the event sets E_1 and E_2 through $l_1 : E_1 \rightarrow A_1$ and $l_2 : E_2 \rightarrow A_2$. Since every event is associated with a unique label, l_1 and l_2 are bijective functions. So we can always find m_e according to m_a. For every event $e_1 \in E_1$, we can find an event $e_2 \in E_2$, such that $m_a(l_1(e_1)) = l_2(e_2)$. Then we obtain the mapping m_e between E_1 and E_2. This step can be done in the complexity of $O(n)$.

After obtaining m_e, we check if the orders of events in $\leq_1$ are still preserved in $\leq_2$. We use Event Order Tables (EOT) [81] to represent orders $\leq_1$ and $\leq_2$. If events *e* and *f* have an order, then the cell (*e*, *f*) in the EOT is marked as True, otherwise it is marked as False. We extend EOTs to include time constraints in cells. If two events *e* and *f* are constrained by a relative time constraint, we write the reduced relative time constraint in the cell (*e*, *f*). We write the reduced absolute time constraint of an event *e* in the cell (*e*, *e*). For example, the EOTs of two bMSCs in Figure 5.4 are shown in Figure 5.5.

events	*a*	*b*
a	[1, 5]	True, [3, 4]
b	False	[4, 9]

(a) EOT1

events	*a*	*b*
a	[1, 2]	True, [3, 4]
b	False	[4, 6]

(b) EOT2

Figure 5.5: Event Order Tables

For every cell (*e*, *f*) in EOT1, if it is marked as True, we check if the cell $(m_e(e), m_e(f))$ in EOT2 is still marked as True. For example, the cell (*a*, *b*) in both EOTs in Figure 5.5 is marked as True. We also check if the time constraint associated with the cell (*e*, *f*) and the time constraint associated with the cell $(m_e(e), m_e(f))$ satisfy the relations defined in Definition 5.2. In Figure 5.5, the time constraints in cells (*b*, *b*) of two EOTs do not satisfy the relation since event *b* is a receiving event. Thus the MSC in Figure 5.4(b) is not a refinement of the MSC in Figure 5.4(a).

Since there are n^2 cells in an EOT, the complexity of comparing two EOTs is $O(n^2)$. Thus the complexity of the whole algorithm is $O(n^3)$.

5.3 Refining HMSCs

5.3.1 Refinement Relation for HMSCs

A bMSC specifies only one scenario of a system. A more complete specification of the system is often given as an HMSC, in which different scenarios are combined. Similarly to the refinement relation between bMSCs, we can also define the refinement relation between HMSCs as a preorder. In the semantics, an HMSC is represented by a set of lposets. After a refinement, each existing lposet should be a refinement of the original one. Moreover, the set could contain new lposets. This is the case when we refine each bMSC referred in the HMSC, and add some new bMSCs.

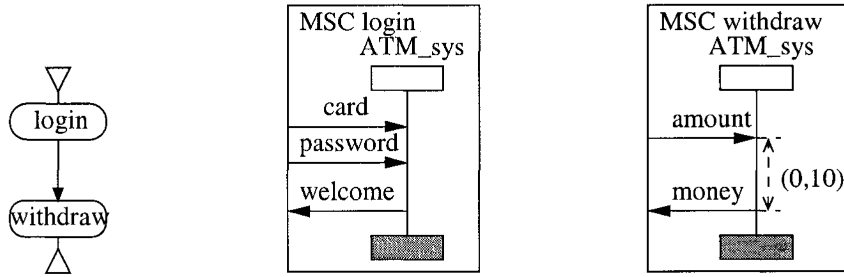


Figure 5.6: An ATM specification before refinement

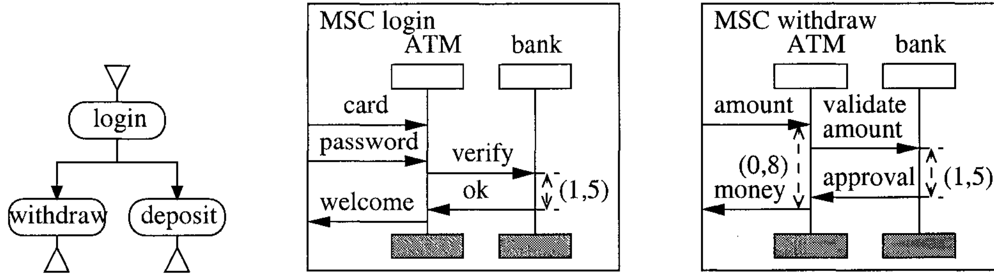


Figure 5.7: An ATM specification after refinement

For an illustration purpose, let us consider an HMSC specification of an Automatic Teller Machine (ATM) in Figure 5.6. The HMSC contains two bMSCs *login* and *withdraw*. We can refine them as described in Section 5.2, and add a bMSC *deposit* to enrich the specification. The resulting HMSC is shown in Figure 5.7. The process *ATM_sys* is decomposed into two processes *ATM* and *bank* in every bMSC, and messages exchanged between them are added as shown in Figure 5.7. Before refinement, a relative time constraint in bMSC *withdraw* specifies the delay between obtaining the amount and delivering the money. It is reduced after refinement to constrain more the *ATM*. Moreover, the response delay of *bank* is added in both bMSCs in Figure 5.7.

Similar to the refinement of bMSCs, the consistency of an HMSC should also be preserved when it is refined. In Chapter 4, we have defined the consistency of HMSCs based on the consistency of paths. We define the refinement relation based on the conformance of paths also.

Definition 5.4 *A path p_2 in an HMSC conforms to a consistent path p_1 in another HMSC*

if and only if the lposet representing p_2 is a refinement of the lposet representing p_1.

Notice that a path could be infinite. In such a case, the lposet contains an infinite number of events. From Definition 5.4, we define the refinement relation between HMSCs as follows.

Definition 5.5 *An HMSC H_2 is a refinement of another HMSC H_1 if and only if for every consistent path p_1 in H_1, there exists a path p_2 in H_2 such that p_2 conforms to p_1.*

The definition of the refinement of an HMSC does not limit the HMSC to be strongly consistent or weakly consistent. If the HMSC is strongly consistent, then its refinement has to be strongly consistent also. If it is weakly consistent, then its refinement can be strongly consistent or weakly consistent.

5.3.2 Checking Conformance of HMSCs

To check the refinement relation of HMSCs according to Definition 5.5, we need to check for a consistent path p in H_1, whether a path exists in H_2 such that it conforms to p. Similar to the matching problem in un-timed MSC [85], this problem is NP-complete.

Proposition 5.2 *The problem of finding a path in an HMSC H_2 that conforms to a given (finite) path in H_1 is NP-complete.*

PROOF. For a given finite path in H_1, we can guess a path in H_2 which conforms to it. So the problem is NP. The Proposition 3.5 in [85] shows that matching an un-timed bMSC with an un-timed HMSC is NP-complete. We reduce the matching problem to our problem since a finite path can be seen as a bMSC. We add absolute and relative time constraints $(0, \infty)$ to events in an un-timed bMSC M and an un-timed HMSC H. Then they become a

timed bMSC M' and a timed HMSC H'. M' conforms to H' if and only if M conforms to H. Then the problem is NP-complete. □

This result suggests it is very complex in computation to check the conformance between HMSCs in general, if it is decidable. However, an efficient algorithm is required since the conformance is checked after each refinement procedure. To solve this problem, we consider a subset of HMSCs, and restrict the methods of changing the HMSCs.

First, we consider the refinement of upper-bound-later FSPHMSCs only, which are defined in Section 4.5. We require that all the paths in an HMSC have to be implemented. So the HMSC is strongly consistent. We do not consider the refinement of weakly consistent HMSCs.

Furthermore, we have the following requirements when changing an HMSC.

- The road map of the HMSC is kept unchanged.
- In vertical refinement, all bMSCs are refined at the same time so that each bMSC has the same set of processes.
- In horizontal refinement, each bMSC is refined such that there is at least one event in each process.
- Time constraints are added or changed within each bMSC.

The first three requirements ensure that the resulting HMSC is still an FSPHMSC. Since the road map is not changed, the set of lposets representing the resulting HMSC contains the same number of lposets as the set prior to the refinement. Refinements occur in each lposet only. The number of bMSCs referred in the HMSC has to be decided at the beginning, but the bMSCs can be at a very abstract level.

The last requirement restricts the usage of relative time constraints within bMSCs. In other words, we assume relative time constraints specified between events in different bMSCs are $(0, \infty)$. However, a delay between events in different bMSCs in the same path of an HMSC can still be set implicitly by absolute time constraints on these events.

Given these restrictions, we investigate conditions under which an HMSC is a refinement of another HMSC.

Theorem 5.1 *An HMSC H_2 is a refinement of a strongly consistent upper-bound-later FSPHMSC H_1 if the following conditions are satisfied,*

- *H_2 is strongly consistent and upper-bound-later,*
- *each bMSC in H_2 conforms to its corresponding bMSC in H_1,*
- *the lower bounds of reduced (absolute and relative) time constraints in each bMSC in H_1 are not changed in H_2, and*
- *in H_2, for each pair of bMSCs M_1' and M_2', such that $M_1' \cdot M_2'$ in H_2, and for each set of axes $\{A_{11}, \ldots, A_{1n}\}$ in M_1' and M_2' resulting from the decomposition of A_1 (in M_1 and M_2 in H_1), the order between the last event of A_1 in M_1 and the first event of A_1 in M_2 is preserved.*

PROOF. This theorem is proven in Appendix B.8. □

This theorem extends the Theorem 1 in [56] for timed MSC. Same as for Theorem 1 in [56], the conditions in Theorem 5.1 are sufficient conditions only. If they are not satisfied, we cannot conclude that H_2 does not conform to H_1. For example, it is not necessary to keep the lower bounds of reduced time constraints unchanged. However, requiring this

condition ensures that the relation between time constraints in corresponding bMSCs is preserved in H_2.

The HMSC in Figure 5.6 (page 100) is an upper-bound-later FSPHMSC, and it is strongly consistent. If we keep its road map unchanged, and refine the bMSCs *login* and *withdraw* as shown in Figure 5.7 (page 100), then the resulting HMSC is also upper-bound-later FSPHMSC. Apparently the HMSC satisfies the conditions defined Theorem 5.1. Thus it is a refinement of the HMSC in Figure 5.6.

When refining a strongly consistent upper-bound-later HMSC, we enforce those conditions in Theorem 5.1 to be satisfied in the resulting HMSC to ensure the conformance. These conditions can be checked efficiently. First, as discussed in Chapter 4, we can check in polynomial time if an FSPHMSC is upper-bound-later, and if it is strongly consistent. For the second condition, the conformance of bMSCs can be checked in time $O(n^3)$ as discussed in Section 5.2, and reduced time constraints are calculated at the same time. The third condition can be checked in linear time by checking each reduced time constraint. For the fourth condition, we need to check each pair of bMSCs and each process in a bMSC in the worst case. This can be done in time $O(m^2p)$, where m is the number of bMSCs and p is the number of processes. Therefore, checking all the conditions in Theorem 5.1 only requires polynomial time.

5.4 Example: Refining a Basic Call Specification

In this section, we specify a basic call in a telephone system. Then we refine the specification to demonstrate our refinement approach for HMSCs.

The MSCs in Figure 5.8 describe the basic call at a high level. The telephone system is considered as a black box and represented by one instance. Users and their telephone handsets are represented by the system environment. At the beginning, a user dials a

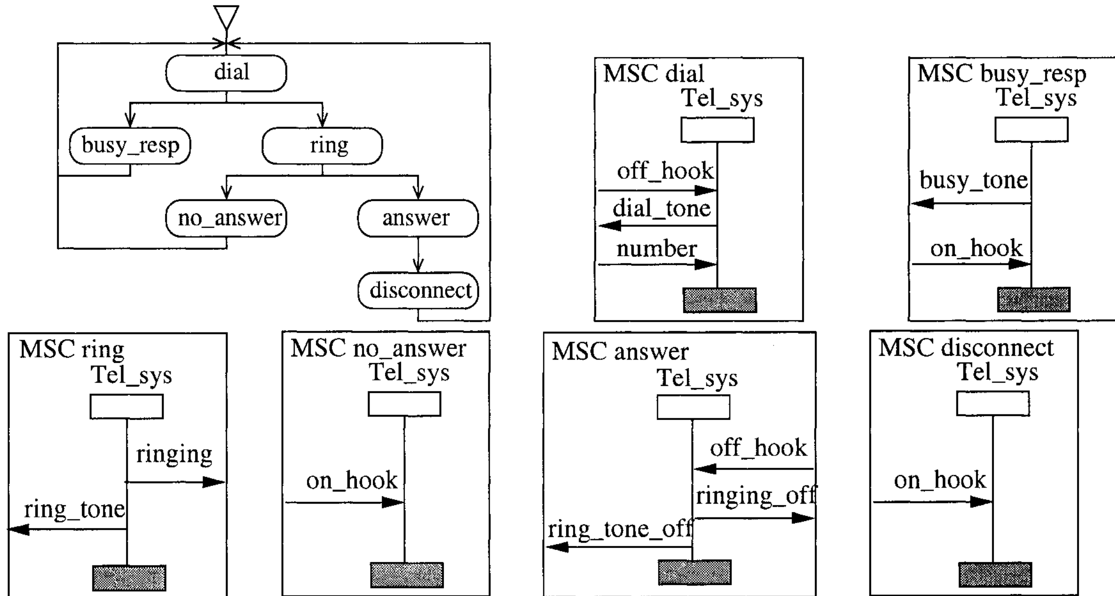


Figure 5.8: Basic Call

number (bMSC *dial*). If the callee is busy, the caller gets a busy response (bMSC *busy_resp*). Otherwise the caller gets a ring tone (bMSC *ring*). The caller may hang up the phone while waiting the callee to answer the call (bMSC *no_answer*). The call is connected if the callee picks up the phone (bMSC *answer*). The call is disconnected if caller or callee hangs up the phone (bMSC *disconnect*).

The MSC specification in Figure 5.8 does not contain any timing requirements. To refine it, we add two timing requirements. The first one is that the telephone system should send out a dial tone within 500 milliseconds after receiving an off_hook signal. The other one is that the system should receive a number within 10 seconds after sending out the dial tone. We choose a discrete time domain and take 100 milliseconds as one time unit. These two requirements can be specified using two relative time constraints in bMSC *dial* as shown in Figure 5.9(a).

The HMSC in Figure 5.8 is an FSPHMSC, and does not contain any explicit time constraints. It is upper-bound-later and strongly consistent because default time constraints are $(0, \infty)$. After those two time constraints are added, the resulting HMSC is still upper-

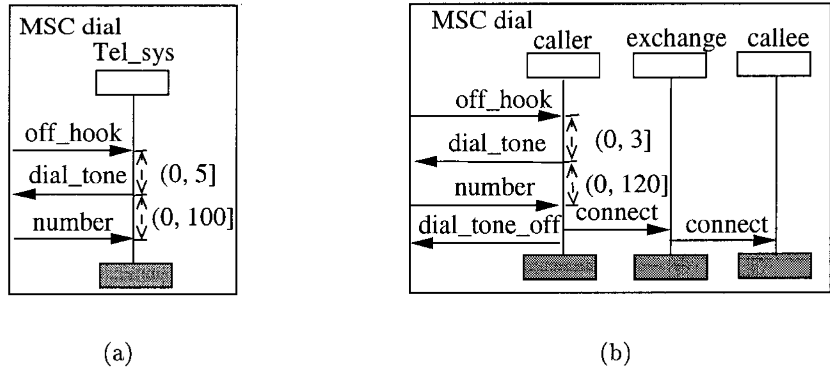


Figure 5.9: Refining bMSC dial

bound-later and strongly consistent. Since added time constraints are (0, 5] and (0, 100], the lower bounds of the default time constraints are not changed. We can check that the conditions in Theorem 5.1 are satisfied.

We can refine the MSC specification further by performing vertical refinements and horizontal refinements. We keep the road map of the HMSC unchanged and refine each bMSC in Figure 5.8. A telephone system can be decomposed as an exchange and controllers for the caller and the callee. In all the bMSCs in Figure 5.8, the instance *Tel_sys* needs to be decomposed as *caller*, *exchange* and *callee*. Messages can be added between these instances. For an illustration purpose, we show the refinement of bMSC *dial* only. In Figure 5.9(b), we add three messages. After receiving the *number*, the *caller* sends the *connect* message to the *exchange*, which transfers the message to the *callee*. The *caller* also stops the dial tone by sending the *dial_tone_off* message to the environment. The event orders in the original bMSC *dial* in Figure 5.9(a) are preserved in Figure 5.9(b).

Moreover, we can refine time constraints as well. We decrease the delay between receiving the *off_hook* and sending the *dial_tone* to (0, 3] as shown in Figure 5.9(b). It requires the system to send out a dial tone within 300 milliseconds instead of 500 milliseconds. We increase the delay between sending the *dial_tone* and receiving the *number* to (0, 120]. It means the system can wait longer for the inputs from the user. We change the upper bounds

of these time constraints only and keep the lower bounds. We can check that the bMSC in Figure 5.9(b) conforms to the bMSC in Figure 5.9(a).

Other bMSCs in Figure 5.8 can be refined similarly. The conformance can be ensured between MSC specifications if the conditions in Theorem 5.1 are satisfied.

5.5 Related Works

The refinement in this chapter generalizes the approach for un-timed MSC specifications in [56]. The refinement in [56] distinguishes between horizontal and vertical refinements. They can be seen as structural and behavioral refinements. In a vertical refinement, a designer decomposes a given instance into multiple instances according to the SDL [49] target architecture. In a horizontal refinement, the MSC specification is enriched with messages. A process alternating between vertical and horizontal refinement has been defined. In order to ensure the preservation of the properties of the MSCs during the refinement, for instance the orders between the events, a conformance relation is defined, which must hold between MSC at stage i and MSC at stage $i+1$ of the refinement process.

Message refinements in un-timed MSC are also discussed in [22]. A message can be replaced by an MSC, named protocol MSC. A protocol MSC can be unidirectional or bidirectional, according to the message flow in the MSC. Whether or not a message refinement can result in deadlocks in these two cases is investigated. In our refinement approach, we preserve the original messages and add new messages.

In [61], four refinement notations are defined, which are binding of references, property refinement, message refinement and structural refinement. The binding of references associates a reference node in an HMSC to an MSC. The property refinement restricts the behavior of an MSC, such as removing alternatives, removing interleaving and strengthening guards. The message refinement replaces a message with an MSC. The structural

refinement replaces a process with a set of processes.

Refinement of interworkings is proposed in [75]. An interworking is similar to a bMSC, but the communication between processes is synchronous. When an interworking is refined, a process can be decomposed into constituents and internal messages can be added between these constituents. Using an operational semantics, a refinement relation is defined based on bisimulation. Our refinement relation can be seen as an inclusion of the refinement relation in [75] and the matching relation in [85].

Refinements of real time specifications based on true concurrency have been proposed, for instance, in [72, 80]. In [80], interval event structures are used as a model. An event can be refined into an event structure. Various notions of equivalence of interval event structures have been defined, which allow for a refinement operation. The authors of [72] extend the refinement to timed bundle event structures. In [72, 80], an event has a duration, which is different with the time concept in MSC. Moreover, these approaches do not allow for refinement of time constraints.

Checking the refinement relation is similar to the matching problem, except it is considered for un-timed MSCs [28, 67, 82, 85]. An MSC graph M or-matches another MSC graph N if a path in M matches a path in N. The graph M and-matches N if all the paths in M matches one path in N. Our refinement relation requires each path in M matches a path (not necessarily the same path) in N. If M and-matches N, then N is a refinement of M. However, the reverse does not hold.

5.6 Conclusion

The specification of a system often begins with a description of functionalities at a high level of abstraction. This specification can be refined step by step into a design specification by adding details about the internal behavior, architecture, time constraints, etc. In this

chapter, we generalized the refinement approach in [56] to timed MSCs to take into account the new time features in MSC. Our methodology allows for the refinement of an un-timed MSC into a timed MSC. The later can be refined further through the refinement of time constraints. Refinement relations have been defined, and algorithms for checking these relations have been developed and discussed.

The refinement relation defined in this chapter satisfies the transitivity under some conditions. This is due to the time constraints associated with receiving events. In an un-timed MSC, we assume that a default time constraint $(0, \infty)$ is associated with a receiving event. Adding an explicit time constraint and then refining it actually reduces the default time constraint and then enlarges it. It may be enlarged to $(0, \infty)$ and reduced again by adding another explicit time constraint. At this time the transitivity may be broken. A potential solution is to assume that the default absolute time constraint of a receiving event is the time constraint of the corresponding sending event. The time constraint can only be enlarged so that the transitivity is assured. This approach has to be explored further by taking into account its effects to the semantics and the consistency of MSCs.

For the refinement of HMSCs, we restrict their modifications and determine sufficient conditions for the refinement relation to hold. This way, we avoid a highly complex solution for checking the refinement relation between HMSCs. The complexity for general HMSCs comes from at least two sources. First, from the formal language point of view, the language represented by HMSC is not a regular language, not even a context-free language [42, 43, 44, 90]. Thus many decision problems are not decidable for MSC, such as the intersection of two HMSCs [85], model checking and verification of HMSCs [6, 8], detecting race conditions and confluence in HMSCs [83], and deciding if an HMSC represents a regular language [42]. Some other problems are decidable but highly complex in computation. For example, the matching problem is NP-complete [82, 85]. The race problem and confluence problem for a subset of HMSCs is EXPSPACE-complete [83]. The model checking for bounded HMSCs is EXPSPACE-complete [8].

Moreover, time constraints in an HMSC can affect each other. Changing the absolute time constraint of an event or a relative time constraint may affect all the absolute time constraints of events causally ordered after it. Even worse, these events may be contained in different bMSCs. This makes the relation between time constraints much harder to check during the refinement, and defeats in most cases the attempts of checking the refinement of an HMSC by checking only individual bMSCs.

In our refinement, we restrict the usage of relative time constraints. A relative time constraint can only be added between causally ordered events to ensure that the specification can be implemented. Alternatively, we may allow for relative time constraints between unrelated events during design. However, this kind of time constraints is at a high level of abstraction and in order to be implementable, the final specification should not contain these relative time constraints. Moreover, we may allow a relative time constraint to be split between different bMSCs in an HMSC specification. Allowing for these kinds of relative time constraints adds more complexity to conformance checking.

Chapter 6

An Extension to MSC

6.1 Introduction

Since the first standardization of MSC, many features have been added to the language to enhance its expressiveness. Compositional structures, such as inline expressions and HMSCs, have been added in MSC-96. Time constraints and data have been added in MSC-2000. With these new concepts, MSC can be used in more application domains. On the other hand, during the usage of MSC, we may discover needs to extend the language further.

During the investigation of time constraints, we found that some timing requirements on repeated behaviors cannot be specified with the time constraints in MSC-2000. For a repeated scenario, we may need to specify how long the scenario takes and the interval between the repetitions. The MSC standard defines relative time constraints between two different events. To specify how long a scenario takes, we may specify the delay between the first and the last event in the scenario using a relative time constraint. Or we may put a relative time constraint on the MSC reference representing the scenario. However, relative time constraints cannot specify the interval between the repetitions. To specify such an interval, we have to specify the delay between the last event in the current repetition and

the first event of the next repetition. Using relative time constraints cannot distinguish different repetitions.

Therefore we propose an extension of time constraints. We introduces a new construct called instance delay. An instance delay specifies a duration such that the executions of all the events in the same process have to be delayed. Using this construct, we can specify the periodicity of processes. The periodicity of events in a process can be specified implicitly with this construct. We define formally the semantics of instance delay and show its usage in a specification of the Radio Resource Control (RRC) protocol [1].

This chapter is organized as follows. In Section 6.2, we discuss the need for the extension, and introduce the syntax of instance delay. Then, in Section 6.3, we define formally its semantics by extending the semantics in Chapter 3. An application of the extension is given in Section 6.4 using the RRC protocol. In Section 6.5, we discuss related work, before we conclude in Section 6.6.

6.2 Instance Delay

To introduce the concept of instance delay, we first look at an example in a client-server system. A server in the system has to respond continuously to the requests from clients. We require that the server has to respond between 1 and 2 seconds after receiving a request. We also require that a client has to wait 2 seconds to send another request after receiving the response for the previous request. In a bMSC, the first requirement can be specified as a time constraint between two events, as shown in MSC *Transaction* in Figure 6.1. For the second requirement, however, we cannot specify the delay between a response and the next request after it in a bMSC.

The second requirement actually defines a delay between the executions of MSCs. If we consider this at the HMSC level, only the whole execution time of an MSC can be specified,

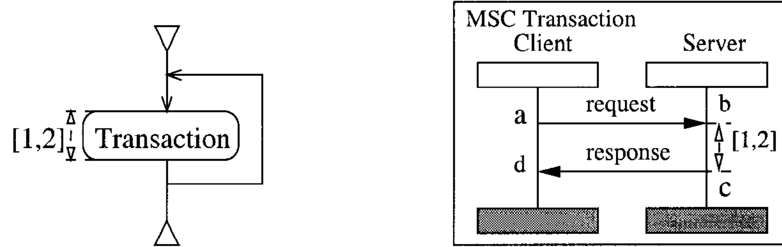


Figure 6.1: Time constraints on events and MSC

such as the constraint on the MSC *Transaction* in Figure 6.1. We cannot specify the delay between the first and the second execution of the MSC *Transaction* in the loop.

We considered the usage of the time offset as defined in the standard to specify the delay between two MSCs indirectly. A time offset is an offset to all absolute time values within an MSC. Since the scope of a time offset is the whole MSC, it cannot be used to specify some more complex timing requirements. For example, if another client is required to wait 3 seconds instead of 2 seconds to send a new request after receiving a response, the time offset cannot express the timing requirement for the two clients in an MSC, because the clients need different offsets.

To solve this problem, we introduce a new concept called instance delay, which defines the delay between two MSCs in the manner of weak sequence. By the weak sequencing operation, two MSCs are connected instance by instance. An instance delay affects the occurrence time of events only at one instance, instead of the whole MSC.

For example, in Figure 6.1, if we define the instance delay is 2 for the instance Client in the MSC *Transaction*, and assume the Client receives the first response at time t_1, then it can send the second request at time $t_1 + 2$. If it receives the second response at time t_2, it can send the third request at time $t_2 + 2$, and so on. If the event of sending a request itself is constrained by an absolute time constraint that is a range of time values, then all the occurrences of sending the request should be within that range.

We define the syntax of instance delays as follows:

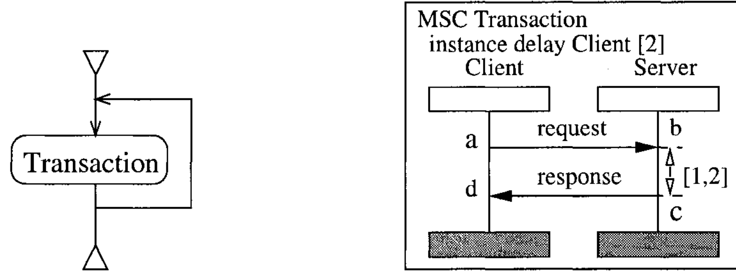


Figure 6.2: Usage of instance delays

⟨instance delay list⟩ ::= *instance delay* ⟨instance delay⟩

⟨instance delay⟩ ::= ⟨instance name⟩ ⟨time range⟩ [, ⟨instance delay⟩]

The ⟨time range⟩ is a range in the time domain with lower and upper bounds. Similarly to the time constraints, if an instance delay is between 1 and 2 time units, we write it as [1, 2]. If it is exactly 2 time units, we write it as [2]. We state instance delays after the MSC name in a bMSC. If the instance delay of an instance is not specified, it can be any time value. Using an instance delay, the two requirements mentioned above can be specified as shown in Figure 6.2.

Since instance delays define the delay between two MSCs, they affect the occurrence time of events in the second and later execution of the MSC only. When the MSC *Transaction* in Figure 6.2 is executed for the first time, events a and d are not affected by the instance delay. Later on, their occurrences will be delayed. This is different from the time offset, which changes the occurrence time of events even in the first execution of the MSC.

6.3 Semantics of Instance Delay

We extend the semantics in Chapter 3 to handle instance delays. We first extend the definition of timed lposet as follows. We use $P(Time)$ to represent a set of time intervals.

Definition 6.1 *An extended timed lposet is a 8-tuple* $(I, A, E, \leq, l, o, D, T)$*, where* I*,* A*,*

E, $\leq$, l, D, and T are same as Definition 3.1, and $o : I \to P(Time)$ is a function that associates a process to a time interval.

The sequential, alternative and parallel compositions of lposets also need to be extended to take instance delays into account. Let $p = (I_p, A_p, E_p, \leq_p, l_p, o_p, D_p, T_p)$ and $q = (I_p, A_q, E_q, \leq_q, l_q, o_q, D_q, T_q)$ be two extended lposets. For two time intervals $U = [x_1, y_1]$ and $V = [x_2, y_2]$, $U + V = [x_1 + x_2, y_1 + y_2]$. If U or V is left (right) open, then $U + V$ is also left (right) open.

Definition 6.2 *The sequential composition ($\cdot$) of p and q is defined as:*
$p \cdot q = (I_p \cup I_q, A_p \cup A_q, E_p \cup E_q, (\leq_p \cup \leq_q \cup \leq_{msg} \cup \leq_{ins})^+, l_p \cup l_q, o_p \cup o'_q, D_p \cup D_q, T_p \cup T_q \cup T_{tim} \cup T_{ins})$, *in which*

- $\leq_{msg} = \leq^p_q \cup \leq^q_p$, $\leq^p_q = \{(e, e') \in E_p \times E_q \mid l_p(e) = send(i, j, m_k) \wedge l_q(e') = receive(j, i, m_k)\}$, $\leq^q_p = \{(e, e') \in E_q \times E_p \mid l_q(e) = send(i, j, m_k) \wedge l_q(e') = receive(j, i, m_k)\}$,
- $\leq_{ins} = \bigcup_i (E^i_p \times E^i_q)$, *$E^i_p$ and E^i_q are the sets of events that occur at instance i,* $E^i_p \subseteq E_p$, $E^i_q \subseteq E_q$,
- $o'_q : I'_q \to P(Time)$, *where* $I'_q \subseteq I_q$, $I'_q \cap I_p = \phi$, *and* $o'_q(i) = o_q(i)$.
- $T_{tim} = T_1 \cup T_2 = \{((e, e'), [n]) \mid e \leq_{ins} e', \nexists f$ *(f is a timer event and $e \leq_{ins} f \leq_{ins} e'$),* $l_p(e) = starttimer(i, T, n), l_q(e') = timeout(i, T)\} \cup \{((e, e'), (0, n)) \mid e \leq_{ins} e', \nexists f$ *(f is a timer event and $e \leq_{ins} f \leq_{ins} e'$),* $l_p(e) = starttimer(i, T, n), l_q(e') = stoptimer(i, T))\}$.
- $T_{ins} = \{((e, f), o_q(i) + T(e, f)) \mid e \in E^i_p$ *and it is the maximal event in the instance* i, $f \in E^i_q$ *and it is the minimal event in the instance* i$\}$.

The sequential composition $p \cdot q$ in Definition 6.2 handles the instance delay as follows. Instance delays in p are kept in $p \cdot q$. Instance delays for processes in q but not in p are also

preserved. The other instance delays in q add delays to relative time constraints between the maximal event in p and the minimal event in q at the same instance.

The alternative composition and the parallel composition are similar to the definitions in Chapter 3, except we use extended time lposets.

The semantics of a bMSC with instance delays is represented by an extended lposet. The semantics of an HMSC is a set of extended lposets. For example, the semantics of the MSC *Transaction* in Figure 6.2 is given by an extended lposet $p = (I, A, E, \leq, l, o, D, T)$, where $E = \{\text{a, b, c, d}\}$, $T(\text{b, c}) = [1, 2]$, and $o(\text{Client}) = [2]$. The semantics of the HMSC in Figure 6.2 is $\{p, p^2, p^3, \ldots\}$. In p^2, for example, $E = \{a_1, b_1, c_1, d_1, a_2, b_2, c_2, d_2\}$, a_1 (a_2) and b_1 (b_2) correspond to the sending and the reception of the first (second) request message, c_1 (c_2) and d_1 (d_2) correspond to the sending and the reception of the first (second) response message. $T(b_1, c_1) = [1, 2]$, $T(b_2, c_2) = [1, 2]$, $T(d_1, a_2) = [0, \infty) + [2] = [2, \infty]$.

6.4 An Application

To demonstrate the need and the usage of the extension, we consider the measurement process in the Radio Resource Control (RRC) protocol [1]. In the WCDMA wireless communication network, a User Equipment (UE) keeps measuring the power of radio signals received from Base Stations (BS). On the request of a BS, the measurement results can be sent back to the BS periodically. Specifically, a BS sends a Measurement Control message to indicate how often the UE should report the results. Then the UE sends Measurement Report messages periodically to the BS. Consider there are two kinds of UEs. One needs to report the result every 12 seconds, and another needs to report every 24 seconds. Using instance delays, we specify the measurement process of two UEs with different periods in Figure 6.3. For the sake of simplicity, we only show the measurement control and report messages. We also add absolute time constraints on the events of receiving the measurement control (@[2]) and sending the measurement reports on UE1 and UE2 (@[2, 50] and @[2,

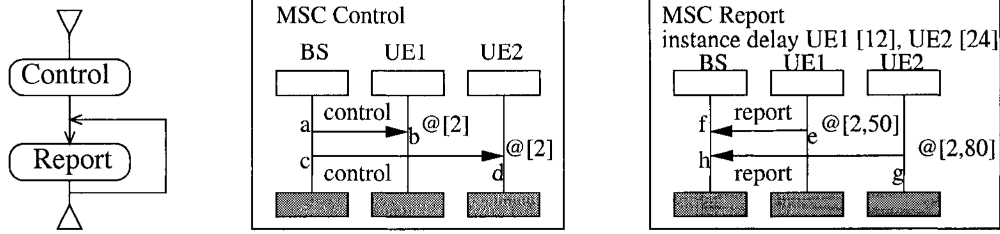


Figure 6.3: Specification of measurement process using instance delays

80] respectively).

The MSC *Control* in Figure 6.3 can be represented by a timed lposet $M_1 = (I_1, A_1, E_1, \leq_1, l_1, o_1, D_1, T_1)$ in which

- I_1 = {BS, UE1, UE2},
- A_1 = {send(BS, UE1, control), receive(UE1, BS, control), send(BS, UE2, control), receive(UE2, BS, control)},
- $E_1 = \{a, b, c, d\}$,
- l_1 = {(a, send(BS, UE1, control)), (b, receive(UE1, BS, control)), (c, send(BS, UE2, control)), (d, receive(UE2, BS, control))},
- $\leq_1 = \{(a,b),(c,d),(a,c)\}^+$, those reflexive pairs such as (a, a), (b, b) are omitted.
- o_1 = {(BS, $[0,\infty)$), (UE1, $[0,\infty)$), (UE2, $[0,\infty)$)}, the instance delays are not specified in the MSC, so we consider that they can be any time value.
- $D_1(b) = [2]$, $D_1(d) = [2]$, the absolute time constraint for event a and c are not specified in the MSC, we consider them as $[0,\infty)$.
- For the function T_1, all the relative time constraints between events are not specified in the MSC, we consider their default values as $[0,\infty)$.

Similarly, we can use a timed lposet $M_2 = (I_2, A_2, E_2, \leq_2, l_2, o_2, D_2, T_2)$ to represent the MSC *Report* in Figure 6.3, in which

- l_2 = {(e, send(UE1, BS, report)), (f, receive(BS, UE1, report)), (g, send(UE2, BS, report)), (h, receive(BS, UE2, report))},

- $\leq_2 = \{(e, f), (g, h), (f, h)\}^+$, those reflexive pairs are omitted.

- o_2 = {(BS, [0, ∞)), (UE1, [12]), (UE2, [24])},

- $D_2(e)$ = [2, 50], $D_2(g)$ = [2, 80].

The semantics of the HMSC in Figure 6.3 can be represented by a set of lposets:

$$\{M_1 \cdot M_2, M_1 \cdot M_2 \cdot M_2, M_1 \cdot M_2 \cdot M_2 \cdot M_2, \ldots\}$$

In M_1, event b and d are the maximal elements in UE1 and UE2 respectively. They are constrained by absolute time constraints (@[2]). When we calculate $M_1 \cdot M_2$, the occurrence time of event e (sending a report at UE1) should satisfy its absolute time constraint @[2, 50] and the relative time constraint between e and b, which is [12, ∞) obtained by taking the instance delay of UE1 into account. So event e occurs at [14, 50] ([2, 50] $\cap$ ([2] + [12, ∞)) = [14, 50]). Similarly, the occurrence time of event g (sending a report at UE2) is [2, 80] $\cap$ ([2] + [24, ∞)) = [26, 80]. For $M_1 \cdot M_2$, event e and g are the maximal elements in UE1 and UE2 respectively. So when we calculate $M_1 \cdot M_2 \cdot M_2$, the occurrence time of the event sending the second measurement report at UE1 will be [2, 50] $\cap$ ([14, 50] + [12, ∞)) = [26, 50], and the occurrence time of the event sending the second measurement report at UE2 will be [2, 80] $\cap$ ([26, 80] + [24, ∞)) = [50, 80]. It is worth to note that when we calculate $M_1 \cdot M_2 \cdot M_2 \cdot M_2 \cdot M_2$, the occurrence time of the event sending the fourth report in UE2 will be [2, 80] $\cap$ ([74, 80] + [24, ∞)) = ϕ. It means that M_2 can only be executed three times actually. So the HMSC is only weakly consistent.

6.5 Related Works

Some extensions to MSC have been introduced in Chapter 2, such as HyperMSC [31], LSC [18], PMSC [25], and YAMS [61]. Moreover, the needs for broadcast messages and timer tables are mentioned in [40]. In the Interval project [46, 102], the authors proposed a new symbol to express periodic occurrence of repetitive events. The symbol is associated with an individual event. However, it is considered as a syntax extension only.

6.6 Conclusion

To specify periodical behaviors more precisely, we introduced instance delay as an extension to MSC-2000. Using this extension, we can specify the delay between two MSCs in an HMSC, and the periodicity can be specified at the instance level. The concept of instance delay is consistent with the weak sequence composition. To define the semantics of this extension, we extended our partial order semantics of timed MSC. With the application to the RRC protocol, we demonstrated the need and the usage of this extension.

This extension has triggered more considerations on time constraints in a loop. An instance delay actually defines the delay between a maximal event and a minimal event in two consecutive bMSCs. It is being considered in the MSC standard organization to define a construct that can specify the delay between any two events in any two iterations of a loop, for example, an event e in the second iteration and an event f in the fifth iteration. While this potential extension enhances the expressiveness of MSC, it brings more difficulties to the analysis of an MSC specification.

Chapter 7

Conclusion

7.1 Summary

As a specification language, MSC is widely used in telecommunication software engineering for specifying behavioral scenarios. Recently, the time concept has been introduced into MSC to handle specifications with real-time constraints. This thesis has addressed two aspects of timed MSC specifications in a development process, validation and refinement. An MSC specification can be used only if it has been validated. The refinement is necessary for developing MSC specifications systematically.

To provide a foundation for the validation and the refinement of timed MSC specifications, we first define a formal semantics for timed MSC. We use timed lposets as a semantic model. A timed lposet includes two timing functions for expressing absolute and relative time constraints respectively. Based on this model, we define all major constructs in MSC-96 and time constraints in MSC-2000. An event in a timed MSC is mapped to a timed lposet. A timed MSC is mapped to a set of lposets. The semantics is compositional. Using the sequential, parallel and alternative operations on lposets, we can obtain the semantics of an MSC from the semantics of contained events, inline expressions or references.

In our semantics, we did not require an MSC to be time consistent. In other words, whether it is consistent or not, an MSC is mapped to a set of lposets. The lposets may be inconsistent. In such a case, the MSC contains errors. We have defined the consistency of an lposet based on the existence of its traces. To validate a bMSC, we have transformed it to a directed constraint graph. Its consistency can be checked in polynomial time. For HMSCs, we have defined strong consistency and weak consistency, and developed sufficient and necessary conditions for these consistencies. However, checking these conditions is complex in computation for HMSCs in general. We have identified a class of HMSCs (FSPHMSCs) such that the strong consistency can be checked in polynomial time.

To develop MSC specifications, we have proposed a refinement approach. A major difference between our approach and other refinement approaches is that we allow for changing time constraints. Time constraints specify constraints on a system and assumptions on its environment. They can be changed during the refinement to make constraints on the system stronger, and assumptions on the environment weaker. However, the consistency between time constraints has to be preserved after the change. It means that if the original MSC is valid, the resulting MSC should also remain valid. We have defined refinement relations for bMSCs and HMSCs. Again, checking the relation between HMSCs is complex in computation. To reduce the complexity, we have restricted the way of refining HMSCs, and developed sufficient conditions for FSPHMSCs to satisfy the relation. These conditions can be checked in polynomial time. During the refinement, they are enforced to ensure the refinement relation.

During the investigation of the semantics and the use of timed MSC, we have found needs to extend the standard language for specifying some time requirements. In the current standard, a relative time constraint can specify the delay between two events. However, it is not possible to specify the delay between two occurrences of the same event in a loop. We have proposed a new construct called instance delay as an extension to enhance further its expressiveness. We have extended our semantics to handle this new construct.

At last, we have implemented the algorithms for checking the consistency of bMSCs and HMSCs.

7.2 Future Work

In the previous chapters, we have provided foundations for using timed MSCs in a development process. We discuss future work about the usage of timed MSC in the following.[1]

7.2.1 Syntactic and Semantic Extensions to MSC

We have proposed a new construct as an extension to timed MSC. More extensions for expressing time requirements in loops are needed and being discussed in the MSC community. For example, a construct is needed to express a delay between two events (or two occurrences of the same event) in different iterations of a loop, or a delay between one occurrence of an event inside a loop and an event outside the loop. A construct expressing a jitter of one iteration inside a loop may be necessary also. The difficulty of making extensions is to find an intuitive graphical syntax for the new constructs.

One future work on the semantics is to incorporate the data concept as defined in MSC-2000. Such a semantics provides a foundation for analyzing an MSC specification containing data variables and time constraints. We may also define a consistent semantics, where the set of lposets representing an MSC contains only consistent lposets. For example, according to the semantics in Chapter 3, a loop expression *loop* $\langle 2,3 \rangle$ M can be represented by $\{p \cdot p, p \cdot p \cdot p\}$, where p is the lposet representing the MSC M. If a time constraint in p causes $p \cdot p \cdot p$ inconsistent, then the consistent semantics of the loop expression is $\{p \cdot p\}$. Under this semantics, time constraints are allowed to change the functional specification. However, this is a challenging task.

[1]Some topics are being investigated by the telesoft research group at Concordia.

7.2.2 Implementability of MSC Specifications

An MSC specification specifies requirements of a system. However, it does not contain information about the architecture of the system. Not all MSC specifications can be implemented under a given architecture. We have to decide if an MSC specification is implementable or not. The implementability of un-timed MSCs has been considered in [23, 57]. Time constraints raise new implementability issues. To be implementable, an MSC has to be consistent. However, consistency does not assure the implementability of an MSC specification. For example, a relative time constraint between two un-ordered events cannot be assured in a distributed architecture. We need to investigate if the other kinds of time constraints can be guaranteed under a given architecture.

7.2.3 Translation to SDL Specifications

If a timed MSC specification is implementable, we can translate it to an SDL specification. Time constraints in MSCs can be implemented using the *now* construct in SDL. However, several new constructs for modeling non-functional aspects, such as communication delay, execution time, scheduling and local time, have been proposed as extensions to SDL [34]. It is interesting to investigate the relations between these constructs and time constraints in MSC.

7.2.4 Tool Support and Case Studies

Tools are needed to support the use of timed MSC. In this thesis, we have provided algorithms for checking the consistency and the refinement relations. The algorithms for checking the consistency have been implemented. It is not difficult to implement other algorithms. These implementations will be integrated with a set of tools, such as tools for translating to SDL and generating test cases, to support the use of MSC in the whole

development process.

To assess our refinement approach, we need more case studies of medium or large size. Through these case studies, we can decide if the refinement rules for HMSCs need to be relaxed, or made more restrictive to ease the checking of conformance relations.

Bibliography

[1] 3GPP: TS 25.331 Radio Resource Control (RRC) Protocol Specification. http://www.3gpp.org.

[2] L. Aceto and D. Murphy: On the Ill-Timed but Well-Caused. Proceeding of CONCUR'93, International Conference on Concurrency Theory, LNCS **715**, Springer-Verlag, 1993.

[3] M. Andersson and J. Bergstrand: Formalizing Use Cases with Message Sequence Charts, Master thesis, Lund Institute of Technology, 1997.

[4] R. Alur and D. L. Dill: A theory of timed automata, Theoretical Computer Science **126** (1994) 183–235.

[5] R. Alur, K. Etessami and M. Yannakakis: Inference of message sequence charts, 22nd International Conference on Software Engineering, 2000 304–313.

[6] R. Alur, K. Etessami and M. Yannakakis: Realizability and verification of MSC graphs, 28th International Colloquium on Automata, Languages and Programming, 2001.

[7] R. Alur, G. J. Holzmann and D. Peled: An analyzer for Message Sequence Charts, in: Proceedings of 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96), LNCS **1055** (1996) 35–48.

[8] R. Alur and M. Yannakakis: Model Checking of Message Sequence Charts, in: the 10th International Conference on Concurrency Theory, LNCS **1664** (1999) 114–129.

[9] J. C. M. Baeten and J. A. Bergstra: Real time process algebra, Journal of Formal Aspects of Computing Science, **3(2)**:142–188, 1991.

[10] H. Ben-Abdallah and S. Leue: Expressing and analyzing timing constraints in Message Sequence Chart specifications, Technical Report **97-04**, Department of Electrical and Computer Engineering, University of Waterloo, 1997.

[11] H. Ben-Abdallah and S. Leue: Syntactic Detection of Process Divergence and non-Local Choice in Message Sequence Charts, in: E. Brinksma (ed.), Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems TACAS'97, LNCS **1217** (1997) 259–274.

[12] H. Ben-Abdallah and S. Leue: MESA: Support for Scenario-Based Design of Concurrent Systems, in: B. Steffen (ed.), Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98, LNCS **1384** (1998) 118–135.

[13] H. Bowman, M. W. A. Steen, E. A. Boiten and J. Derrick: A formal framework for viewpoint Consistency, Formal Methods in System Design, **21** (2002) 111–166.

[14] M. Broy and I. Kruger: Interaction Interfaces - Towards a scientific foundation of a methodological usage of Message Sequence Charts. In: J. Staples, M.G.Hinchey and S. Liu (Eds.), Formal Engineering Methods (ICFEM'98), 1998, 2–15.

[15] R. T. Casley: On the specification of concurrent systems, Ph.D Thesis, Stanford University, 1991.

[16] E. M. Clarke, O. Grumberg and D. A. Peled: Model Checking, the MIT Press, 2000.

[17] T. H. Cormen, C. E. Leiserson and R. L. Rivest: Introduction to Algorithms, second edition, MIT Press, 2001.

[18] W. Damm and D. Harel: LSCs: Breathing life into Message Sequence Charts, Formal Methods in System Design, **19-1** (2001) 45–80.

[19] R. Dechter, I. Meiri and J. Pearl: Temporal constraint networks, Artificial Intelligence **49** (1991) 61–95.

[20] A. Ek: Verifying Message Sequence Charts with the SDT validator, in: O. Færgemand, A. Sarma (Eds.), SDL'93 – Using Objects, Proceedings of the Sixth SDL Forum, 1993, 237–249.

[21] A. Ek: The SOMT Method, technical paper, Telelogic, http://tau.telelogic.com/download/papers, 1995.

[22] A. G. Engels: Languages for Analysis and Testing of Event Sequences, Ph.D. thesis, Eindhoven University of Technology, 2001.

[23] A. Engels, S. Mauw and M. A. Reniers: A hierarchy of communication models for message sequence charts, in: T. Mizuno, N. Shiratori, T. Higashino, A. Togashi (Eds.), Formal Description Techniques and Protocol Specification, Testing and Verification. Proceedings of FORTE X and PSTV XVII'97, Chapman & Hall, 1997.

[24] A. G. Engels, L. M. G. Feijs and S. Mauw: MSC and data: dynamic variables, In R. Dssouli, G. von Bochmann, and Y. Lahav, editors, SDL'99, Proceedings of the Ninth SDL Forum, Elsevier Science Publishers, June 1999.

[25] N. Faltin, L. Lambert, A. Mitschele-Thiel and F. Slomka: An Annotational Extension of Message Sequence Charts to Support Performance Engineering, SDL'97: Time for Testing - SDL, MSC and Trends, Proc. Eighth SDL Forum, A. Cavalli, A. Sarma (Ed.), Elsevier, 1997.

[26] L. M. G. Feijs and S. Mauw: MSC and data, In Y. Lahav, A. Wolisz, J. Fischer, and E. Holz, editors, SAM98 - 1st Workshop on SDL and MSC, Proceedings of the SDL Forum Society on SDL and MSC, number 104 in Informatikberichte, pages 85-96. Humboldt-Universität Berlin, 1998.

[27] T. Gehrke, M. Huhn, A. Rensink and H. Wehrheim: An Algebraic Semantics for Message Sequence Chart Documents, Proceedings of Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE/PSTV'98), 1998, 3–18.

[28] B. Genest and A. Muscholl: Pattern matching and membership for hierarchical Message Sequence Charts, in: Proceedings of LATIN'02, LNCS **2286**, 2002.

[29] J. Grabowski: Test case generation and test case specification with message sequence charts, Ph.D thesis, Universität Bern, 1994.

[30] J. Grabowski, D. Hogrefe and R. Nahm: Test case generation with test purpose specification by MSCs, in: O. Færgemand, A. Sarma (Eds.) SDL'93 - Using Objects, Proceedings of the Sixth SDL Forum, North-Holland, 1993, 253–265.

[31] J. Grabowski, P. Graubmann and E. Rudolph: HyperMSCs with Connectors for Advanced Visual System Modelling and Testing. Proceedings of 10th International SDL Forum, LNCS **2078** (2001) 129–147

[32] J. Grabowski and E. Rudolph: Putting extended sequence charts to practice, in: O. Færgemand, M.M.Marques, (Eds.), SDL'89: The Language at Work, Elsevier Science Publisher, 1989, 3–10.

[33] P. Graubmann, E. Rudolph and J. Grabowski: Towards a Petri Net Based Semantics Definition for Message Sequence Charts, in: Proceedings of the 6th SDL Forum (SDL'93), 1993.

[34] S. Graf: Expression of time and duration constraints in SDL, in: Proceedings of 3rd SDL and MSC Workshop(SAM'02), Aberystwyth, UK, June 2002.

[35] P. Graubmann and E. Rudolph: HyperMSCs and Sequence Diagrams for Use Case Modelling and Testing, UML2000 – The Unified Modeling Language, LNCS **1939** (2000) 32–47.

[36] D. Harel and H. Kugler: Synthesizing State-Based Object Systems from LSC Specifications, in: S. Yu, A. Păun (Eds.), Implementation and Application of Automata, CIAA 2000, LNCS **2088** 2000.

[37] E. Harel, O. Lichtenstein and A. Pnueli: Explicit clock temporal logic, 5th IEEE Symposium on Logic in Computer Science, 1990 402–413.

[38] Ø. Haugen: MSC-2000 Interaction Diagrams for the new Millennium, Computer Networks, bfseries 35 (2001) 721–732.

[39] L. Hélouët: A simulation model for Message Sequence Charts, in: Proceedings of the Ninth SDL Forum, Elsevier Science, 1999, 473–488.

[40] L. Hélouët: Distributed System Modeling with Scenarios: The Example of the RMTP2 Protocol. Concordia Prestigious Workshop on Communication Software Engineering. September 2001

[41] L. Hélouët: Some Pathological Message Sequence Charts and How to Detect them, in: R. Reed (Ed.), 10th SDL Forum, Meeting UML, LNCS **2078**, 2001, 348–364.

[42] J. G. Henriksen, M. Mukund, K. N. Kumar and P. S. Thiagarajan: Towards a theory of regular MSC languages, BRICS Report **RS-99-52**, University of Aarhus, Denmark, 1999.

[43] J. G. Henriksen, M. Mukund, K. N. Kumar and P. S. Thiagarajan: On message sequence graphs and finitely generated regular MSC languages, ICALP'2000, LNCS **1853**, Springer-Verlag, 2000.

[44] J. G. Henriksen, M. Mukund, K. N. Kumar and P. S. Thiagarajan: Regular Collections of Message Sequence Charts, in: Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science (MFCS'2000), LNCS **1893**, Springer-Verlag, 2000.

[45] S. Heymer: A Non-Interleaving Semantics for MSC. The 1st Workshop of the SDL Forum Society on SDL and MSC (SAM'98). 1998

[46] D. Hogrefe, B. Koch and H. Neukirchen: Some Implications of MSC, SDL and TTCN Time Extensions for Computer-Aided Test Generation. Proceedings of 10th International SDL Forum. LNCS **2078** (2001) 168–181

[47] J. E. Hopcroft and J. D. Ullman: Introduction to Automata Theory, Languages, and Computation, 2nd edition, Addison-Wesley, 2000.

[48] ITU-T: Abstract Syntax Notation One - ASN.1, ITU-T Recommendation X.680 - X.693, July 2002.

[49] ITU-T: Specification and Description Language - SDL-2000. ITU-T Recommendation Z.100. November 1999.

[50] ITU-T: Message Sequence Charts, ITU-T Recommendation Z.120, 1993.

[51] ITU-T: Message Sequence Charts, ITU-T Recommendation Z.120, 1996

[52] ITU-T: Message Sequence Charts - MSC-2000, ITU-T Recommendation Z.120, 1999

[53] B. Jonsson and G. Padilla: An Execution Semantics for MSC-2000. Proceedings of 10th International SDL Forum. LNCS **2078** (2001) 365–378

[54] J. P. Katoen: Quantitative and qualitative extensions of event structures, Ph.D Thesis, University of Twente, 1996.

[55] J. P. Katoen and L. Lambert: Pomsets for Message Sequence Charts. The 1st Workshop of the SDL Forum Society on SDL and MSC (SAM'98). 1998

[56] F. Khendek, S. Bourduas and D. Vincent: Stepwise Design with Message Sequence Charts, Proceedings of FORTE'2001, Cheju Island, Korea, August 2001.

[57] F. Khendek, G. Robert, G. Butler and P. Grogono: *Implementability of Message Sequence Charts*, SAM'98, Berlin, Germany, June 1998.

[58] R. Koymans: Specifying real-time properties with metric temporal logic, Real-time Systems, **2(4)** (1990) 255–299.

[59] R. Koymans: Specifying Message Passing and Time-Critical Systems with Temporal Logic, LNCS **651**, Springer-Verlag, 1992.

[60] R. Koymans, J. Vytopyl and W.-P. de Roever: Real-time programming and asynchronous message passing, 2nd ACM Symposium on Principles of Distributed Computing, 1983, 187–197.

[61] I. H. Kruger: Distributed System Design with Message Sequence Charts, Ph.D thesis, Technische Universität München, 2000.

[62] P. B. Ladkin and S. Leue: What do Message Sequence Charts mean? in: R.L. Tenney, P.D.Amer, M.Ü.Uyar, (Eds.), Formal Description Techniques VI, IFIP Transactions C, Proceedings of the Sixth International Conference on Formal Description Techniques, North-Holland, 1994, 301–316.

[63] P. B. Ladkin and S. Leue: Four issues concerning the semantics of Message Flow Graphs, in: D.Hogrefe, S.Leue, (Eds.), Formal Description Techniques VII, Proceedings of the Seventh IFIP WG 6.1 International Conference on Formal Description Techniques (FORTE'94), Chapman &Hall, 1995, 355–369.

[64] P. B. Ladkin and S. Leue: Interpreting Message Flow Graphs. Formal Aspects of Computing **7(5)** (1995) 473–509.

[65] S. Leue and P.B. Ladkin: Implementing and Verifying Scenario-Based Specifications Using Promela/XSpin, Proceedings of the Second SPIN Workshop, 1996.

[66] S. Leue, L. Mehrmann and M. Rezai: Synthesizing ROOM Models from Message Sequence Chart Specifications, the 13th IEEE Conference on Automated Software Engineering, 1998.

[67] V. Levin and D. Peled: Verification of message sequence charts via template matching, in: Theory and Practics of Software Development, TAPSOFT(FASE)'97, LNCS **1214** (1997) 652–666.

[68] X. Li and J. Lilius: Timing analysis of UML sequence diagrams, in: Proceedings of UML'99 - The Unified Modeling Language, Beyond the Standard, LNCS **1723** (1999) 430–445.

[69] X. Li and J.Lilius: Timing analysis of Message Sequence Charts, TUCS Technical Report **255**, Turku Centre for Computer Science, 1999.

[70] N. Lynch and F. Vaandrager: Action Transducers and Timed Automata, Formal Aspects of Computing **8(5)** (1996) 499–538.

[71] P. L. Maigat and L. Helouët: A (MAX, +) Approach for Time in Message Sequence Charts. 5th Workshop on Discrete Event Systems. August 2000

[72] M. Majster-Cederbaum and J. Wu: Action Refinement for True Concurrent Real Time, Seventh International Conference on Engineering of Complex Computer Systems, Sweden, 2001.

[73] Z. Manna and A. Pnueli: The Temporal Logic of Reactive and Concurrent Systems: Specification, Springer-Verlag, 1992.

[74] Z. Manna and A. Pnueli: Models for Reactivity, Acta Informatica, **30** (1993) 609–678.

[75] S. Mauw and M. A. Reniers: Refinement in Interworkings, Proceedings of CONCUR'96, LNCS **1119** (1996) 671–686.

[76] S. Mauw and M. A. Reniers: High-level Message Sequence Charts. SDL'97: Time for Testing - SDL, MSC and Trends. September 1997

[77] S. Mauw and M. A. Reniers: Operational Semantics for MSC'96. Computer Networks and ISDN Systems **31(17)** (1999) 1785–1799

[78] A. Mazurkiewicz: Basic notions of trace theory, in: J.W. de Bakker, W.P. de Roever, G. Rozenberg, (Eds.) Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, LNCS **354**, Springer Verlag, 1989.

[79] D. Murphy: Time, causality and concurrency, Ph.D Thesis, University of Surrey, 1990.

[80] D. Murphy and D. Pitt: Real-timed Concurrent Refineable Behaviours, Proceedings of the 2nd International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, LNCS **571** (1992) 529–545.

[81] M. M. Musa, F. Khendek and G. Butler: New results on deriving SDL specification from MSCs, in: R.Dssouli, G.v.Bochmann, Y.Lahav (Eds.), Proceedings of SDL Forum'99, Elsevier Science B.V., June 1999.

[82] A. Muscholl: Matching specifications for message sequence charts, Proceedings of FoSSaCS'99, LNCS **1578** (1999).

[83] A. Muscholl and D. Peled: Message sequence graphs and decision problems on Mazurkiewicz traces, in: Proceedings of MFCS'99, LNCS **1672** (1999) 81–91.

[84] A. Muscholl and D. Peled: Analyzing Message Sequence Charts, in: Proceedings of SAM'00, 2000.

[85] A. Muscholl, D. Peled and Z. Su: Deciding Properties for Message Sequence Charts, Proceedings of the 1st International Conference on Foundations of Software Science and Computation Structures, LNCS **1378** (1998).

[86] R. Nahm: Conformance Testing Based on Formal Description Techniques and Message Sequence Charts, Ph.D thesis, Universität Bern, 1994.

[87] M. Nielsen, G. D. Plotkin and G. Winskel: Petri nets, event structures and domains, part 1, Theoretical Computer Science, **13(1)**, 85–108, 1981.

[88] Object Management Group: OMG Unified Modeling Language Specification, Version 1.5, March 2003, http://www.omg.org.

[89] J. S. Ostroff: Temporal Logic of Real-time Systems, Advanced Software Development Seires, Research Studies Press (John Wiley & Sons), England, 1990.

[90] D. Peled: Specification and verification using Message Sequence Charts, in: B.Caillaud, A.Muscholl (Eds.), Validation and Implementation of Scenario-Based Specifications (VISS'02), ENTCS **65:7** 2002.

[91] A. Pnueli: Specification and Development of Reactive Systems, in: H.-J. Kugler, editor, Information Processing 86, IFIP, North-Holland, 1986, 845–858.

[92] A. Pnueli: System Specification and Refinement in Temporal Logic, in: R.K. Shyamasundar, editor, Foundations of Software Technology and Theoretical Computer Science, LNCS **652** (1992) 1–38.

[93] V. R. Pratt: Modeling concurrency with partial orders, International Journal of Parallel Programming, **15(1)** 33–71, 1986

[94] W. Reisig: Petri Nets, An Introduction, in: W.Brauer, G.Rozenberg, A.Salomaa (Eds.), EATCS, Monographs on Theoretical Computer Science, Springer Verlag, 1985.

[95] M. A. Reniers: Message Sequence Chart: Syntax and Semantics, Ph.D thesis, Eindhoven University of Technology, 1999.

[96] A. Rensink: Models and methods for action refinement, Ph.D Thesis, University of Twente, 1993.

[97] G. Robert, F. Khendek, and P. Grogono: Deriving an SDL specification with a given architecture from a set of MSCs, in: A.Cavalli, A.Sarma (Eds.), Proceedings of SDL Forum'97, Elsevier Science B.V., September 1997.

[98] E. Rudolph, J. Grabowski and P.Graubmann: Towards a Harmonization of UML-Sequence Diagrams and MSC, in: Y.Lahav, R.Dssouli (Eds.), SDL'99, The Next Millennium, Proceedings of the 9th SDL Forum, North Holland, June 1999.

[99] D. J. Scholefield: A Refinement Calculus for Real-time Systems, Ph.D thesis, University of York, UK, 1992.

[100] M. W. Shields: Concurrent machines, The Computer Journal, **28(5)** 449–465, 1985.

[101] Telelogic Tau, http://www.telelogic.com.

[102] The Interval Project, http://www-interval.imag.fr.

[103] The ESPRIT project: CREWS - Cooperative Requirements Engineering with Scenarios, http://sunsite.informatik.rwth-aachen.de/crews.

[104] D. Toggweiler, J. Grabowski and D. Hogrefe: Partial Order simulation of SDL specifications, in: R. Bræk, A. Sarma (Eds.) SDL'95 - with MSC in CASE, Proceedings of the Seventh SDL Forum, 1995, 293–306.

[105] L. Wang: Implementation of Time Consistency of MSC-2000 Specifications, Internal Report, Department of Electrical and Computer Engineering, Concordia, 2003.

[106] Y. Wang: Real-time behavior of asynchronous agents, in: J.C.M.Baeten and J.W.Klop (Eds.), Proceedings of CONCUR 90, LNCS **458**, 1990, 502–520.

[107] J. M. Wing: A specifier's introduction to formal methods, IEEE Computer, **23**(9):8–24, 1990.

[108] T. Zheng and F. Khendek: An Extension for MSC-2000 and Its Application, in: E.Sherratt (Ed.), Telecommunications and beyond: The Broader Applicability of SDL and MSC, Third International Workshop, SAM 2002, LNCS **2599**, 2002.

[109] T. Zheng and F. Khendek: Time Consistency of MSC-2000 Specifications, Computer Networks, **42:3** (2003) 283–417.

[110] T. Zheng, F. Khendek and B. Parreaux: Refining Timed MSCs, in: R. Reed, J. Reed (Eds.), SDL 2003: System Design, 11th SDL forum, LNCS **2708**, 2003. [Best Paper Award]

[111] T. Zheng, F. Khendek and L. Hélouët: A semantics for timed MSC, in: B.Caillaud, A.Muscholl (Eds.), Validation and Implementation of Scenario-Based Specifications (VISS'02), ENTCS **65:7** 2002.

Appendix A

Simplified Textual Syntax of MSC

⟨msc⟩ ::= ⟨msc statement⟩*

⟨msc statement⟩ ::= ⟨instance name⟩ : ⟨instance event list⟩

⟨instance event list⟩ ::= ⟨instance event⟩$^+$

⟨instance event⟩ ::= ⟨orderable event⟩ | ⟨non-orderable event⟩

⟨orderable event⟩ ::= {⟨event name⟩ ⟨message event⟩ |
    ⟨action⟩ |
    ⟨timer statement⟩} [*time* ⟨time dest list⟩]

⟨message event⟩ ::= *out* ⟨message name⟩ *to* {⟨instance name⟩ | *env*} |
    in ⟨message name⟩ *from* {⟨instance name⟩ | *env*}

⟨action⟩ ::= *action* ⟨action name⟩

⟨timer statement⟩ ::= *starttimer* ⟨timer name⟩ [⟨duration⟩] |
stoptimer ⟨timer name⟩ |
timeout ⟨timer name⟩

⟨time dest list⟩ ::= ⟨time interval⟩ [⟨event name⟩] [, ⟨time dest list⟩]

⟨time interval⟩ ::= ⟨singular time⟩ | ⟨bounded time⟩

⟨singular time⟩ ::= '['⟨time point⟩']'

⟨time point⟩ ::= [@]⟨time value⟩

⟨bounded time⟩ ::= [@]{'[' | '('}[⟨time point⟩], [⟨time point⟩] {']' | ')'}

⟨non-orderable event⟩ ::= ⟨shared msc reference⟩ | ⟨shared inline expr⟩ | ⟨coregion⟩

⟨shared msc reference⟩ ::= *reference* ⟨msc reference name⟩ [time ⟨time interval⟩]

⟨shared inline expr⟩ ::= {*loop* [⟨loop boundary⟩] *begin* ⟨instance event list⟩
loop end [⟨time interval⟩] |
alt begin ⟨instance event list⟩
{*alt* ⟨instance event list⟩*}
alt end [⟨time interval⟩] |
par begin ⟨instance event list⟩
{*par* ⟨instance event list⟩*}
par end [⟨time interval⟩]}

⟨coregion⟩ ::= *concurrent* ⟨orderable event⟩* *endconcurrent*

Appendix B

Proofs

B.1 Proposition 3.1

PROOF.

B.1.1 Identity

I, A, E, $\leq$, l, D, and T in ε are empty set ϕ.

- According to Definition 3.4, $\varepsilon \cdot p = (\phi \cup I_p, \phi \cup A_p, \phi \cup E_p, (\phi \cup \leq_p \cup \leq_{msg} \cup \leq_{ins})^+, \phi \cup l_p, \phi \cup D_p, \phi \cup T_p \cup T_{tim})$, where $\leq_{msg}$, $\leq_{ins}$ and T_{tim} must be empty also. So $\varepsilon \cdot p = (I_p, A_p, E_p, \leq_p, l_p, D_p, T_p) = p$. Similarly, $p \cdot \varepsilon = (I_p \cup \phi, A_p \cup \phi, E_p \cup \phi, (\leq_p \cup \phi \cup \leq_{msg} \cup \leq_{ins})^+, l_p \cup \phi, D_p \cup \phi, T_p \cup \phi \cup T_{tim}) = (I_p \cup \phi, A_p \cup \phi, E_p \cup \phi, (\leq_p \cup \phi \cup \phi \cup \phi)^+, l_p \cup \phi, D_p \cup \phi, T_p \cup \phi \cup \phi) = p$.

- According to Definition 3.5, $\varepsilon \parallel p = (\phi \cup I_p, \phi \cup A_p, \phi \cup E_p, \phi \cup \leq_p, \phi \cup l_p, \phi \cup D_p, \phi \cup T_q) = p$. $p \parallel \varepsilon = (I_p \cup \phi, A_p \cup \phi, E_p \cup \phi, \leq_p \cup \phi, l_p \cup \phi, D_p \cup \phi, T_p \cup \phi) = p$.

B.1.2 Commutative property

- According to Definition 3.5, $p \parallel q = (I_p \cup I_q, A_p \cup A_q, E_p \cup E_q, \leq_p \cup \leq_q, l_p \cup l_q, D_p \cup D_q, T_p \cup T_q) = (I_q \cup I_p, A_q \cup A_p, E_q \cup E_p, \leq_q \cup \leq_p, l_q \cup l_p, D_q \cup D_p, T_q \cup T_p) = q \parallel p$. According to Definition 3.7, $P \parallel Q = \{p_i \parallel q_j \mid p_i \in P, q_j \in Q, 1 \leq i \leq n, 1 \leq j \leq k\}$. Since $p_i \parallel q_j = q_j \parallel p_i$, $P \parallel Q = Q \parallel P$.

- According to Definition 3.6, $p\#q = \{p\} \cup \{q\} = \{q\} \cup \{p\} = q\#p$. According to Definition 3.7, $P\#Q = P \cup Q = Q \cup P = Q\#P$.

B.1.3 Associative property

- According to Definition 3.4, $p \cdot (q \cdot r) = (I_p \cup I_q \cup I_r, A_p \cup A_q \cup A_r, E_p \cup E_q \cup E_r, \leq_{p(qr)}$ $, l_p \cup l_q \cup l_r, D_p \cup D_q \cup D_r, T_{p(qr)})$. On the other hand, $(p \cdot q) \cdot r = (I_p \cup I_q \cup I_r, A_p \cup A_q \cup A_r, E_p \cup E_q \cup E_r, \leq_{(pq)r}, l_p \cup l_q \cup l_r, D_p \cup D_q \cup D_r, T_{(pq)r})$. We need to prove $\leq_{p(qr)}=\leq_{(pq)r}$.

  $\leq_{p(qr)}= (\leq_p \cup \leq_{qr} \cup \leq_p^{qr} \cup \leq_{qr}^{p} \cup \bigcup_i (E_p^i \times E_{qr}^i))^+$, where $\leq_{qr}= (\leq_q \cup \leq_r \cup \leq_q^r \cup \leq_r^q \cup \bigcup_i (E_q^i \times E_r^i))^+$, and $\leq_{(pq)r}= (\leq_{pq} \cup \leq_r \cup \leq_{pq}^r \cup \leq_r^{pq} \cup \bigcup_i (E_{pq}^i \times E_r^i))^+$, where $\leq_{pq}= (\leq_p \cup \leq_q \cup \leq_q^p \cup \leq_p^q \cup \bigcup_i (E_p^i \times E_q^i))^+$. For (e,e') such that $(e,e') \in \leq_{p(qr)}$, there are the following cases:

  - $(e,e') \in \leq_p$. We obtain $(e,e') \in \leq_{pq} \in \leq_{(pq)r}$.
  - $(e,e') \in \leq_{qr}$. If $(e,e') \in \leq_q$, then $(e,e') \in \leq_{pq} \in \leq_{(pq)r}$. If $(e,e') \in \leq_r$, then $(e,e') \in \leq_{(pq)r}$. If $(e,e') \in \leq_q^r$, then $(e,e') \in \leq_{pq}^r \in \leq_{(pq)r}$. If $(e,e') \in \leq_r^q$, then $(e,e') \in \leq_r^{pq} \in \leq_{(pq)r}$. If $(e,e') \in \bigcup_i (E_q^i \times E_r^i)$, then $(e,e') \in \bigcup_i (E_{pq}^i \times E_r^i) \in \leq_{(pq)r}$.
  - $(e,e') \in \leq_p^{qr}$. If $e \in E_r$, then $(e,e') \in \leq_{pq}^r \in \leq_{(pq)r}$. If $e \in E_q$, then $(e,e') \in \leq_p^q \in \leq_{pq} \in \leq_{(pq)r}$.
  - $(e,e') \in \leq_{qr}^p$. If $e' \in E_q$, then $(e,e') \in \leq_q^p \in \leq_{pq} \in \leq_{(pq)r}$. If $e' \in E_r$, then $(e,e') \in \leq_r^{pq} \in \leq_{(pq)r}$.

- $(e,e') \in \bigcup_i(E_p^i \times E_{qr}^i)$. If $e' \in E_q$, then $(e,e') \in \bigcup_i(E_p^i \times E_q^i) \in \leq_{pq} \in \leq_{(pq)r}$. If $e' \in E_r$, then $(e,e') \in \bigcup_i(E_{pq}^i \times E_r^i) \in \leq_{(pq)r}$.

So, if $(e,e') \in \leq_{p(qr)}$, then $(e,e') \in \leq_{(pq)r)}$. Similarly to the proof above, if $(e,e') \in \leq_{(pq)r}$, then $(e,e') \in \leq_{p(qr)}$. Thus $\leq_{p(qr)} = \leq_{(pq)r}$.

We also need to prove $T_{p(qr)} = T_{(pq)r}$. $T_{p(qr)} = T_p \cup T_{qr} \cup T_{tim}^{p(qr)}$ where $T_{qr} = T_q \cup T_r \cup T_{tim}^{qr}$, and $T_{(pq)r} = T_{pq} \cup T_r \cup T_{tim}^{(pq)r}$, where $T_{pq} = T_p \cup T_q \cup T_{tim}^{pq}$. For $((e,e'),t) \in T_{p(qr)}$, there are the following cases:

- $((e,e'),t) \in T_p$. We obtain $((e,e'),t) \in T_{pq} \in T_{(pq)r}$.
- $((e,e'),t) \in T_{qr}$. If $((e,e'),t) \in T_q$, then $((e,e'),t) \in T_{pq} \in T_{(pq)r}$. If $((e,e'),t) \in T_r$, then $((e,e'),t) \in T_{(pq)r}$. If $((e,e'),t) \in T_{tim}^{qr}$, then $((e,e'),t) \in T_{tim}^{(pq)r} \in T_{(pq)r}$.
- $((e,e'),t) \in T_{tim}^{p(qr)}$. If $e' \in E_q$, then $((e,e'),t) \in T_{tim}^{pq} \in T_{(pq)r}$. If $e' \in E_r$, then $((e,e'),t) \in T_{tim}^{(pq)r} \in T_{(pq)r}$.

So if $((e,e'),t) \in T_{p(qr)}$, then $((e,e'),t) \in T_{(pq)r}$. Similarly, if $((e,e'),t) \in T_{(pq)r}$, then $((e,e'),t) \in T_{p(qr)}$. Thus, $T_{p(qr)} = T_{(pq)r}$. We can conclude $p \cdot (q \cdot r) = (p \cdot q) \cdot r$.

According to Definition 3.7, $P \cdot (Q \cdot R) = P \cdot \{q_i \cdot r_j \mid q_i \in Q, r_j \in R\} = \{p_k \cdot (q_i \cdot r_j) \mid p_k \in P, q_i \in Q, r_j \in R\}$. Because $p_k \cdot (q_i \cdot r_j) = (p_k \cdot q_i) \cdot r_j$, $P \cdot (Q \cdot R) = (P \cdot Q) \cdot R$.

- According to Definition 3.5, $p \parallel (q \parallel r) = p \parallel (I_q \cup I_r, A_q \cup A_r, E_q \cup E_r, \leq_q \cup \leq_r, l_q \cup l_r, D_q \cup D_r, T_q \cup T_r) = (I_p \cup I_q \cup I_r, A_p \cup A_q \cup A_r, E_p \cup E_q \cup E_r, \leq_p \cup \leq_q \cup \leq_r, l_p \cup l_q \cup l_r, D_p \cup D_q \cup D_r, T_p \cup T_q \cup T_r) = (p \parallel q) \parallel r$. According to Definition 3.7, $P \parallel (Q \parallel R) = P \parallel \{q_i \parallel r_j \mid q_i \in Q, r_j \in R\} = \{p_k \parallel (q_i \parallel r_j) \mid p_k \in P, q_i \in Q, r_j \in R\}$. Because $p_k \parallel (q_i \parallel r_j) = (p_k \parallel q_i) \parallel r_j$, $P \parallel (Q \parallel R) = (P \parallel Q) \parallel R$.

- According to Definition 3.6 and Definition 3.7, $\{p\}\#(q\#r) = \{p\}\#(\{q\} \cup \{r\}) = \{p\} \cup (\{q\} \cup \{r\}) = (\{p\} \cup \{q\}) \cup \{r\} = (p\#q)\#\{r\}$. $P\#(Q\#R) = P \cup (Q \cup R) = (P \cup Q) \cup R) = (P\#Q)\#R$.

B.1.4 Distributive property

- According to Definition 3.6 and Definition 3.7, $\{p\} \cdot (q\#r) = \{p\} \cdot (\{q\} \cup \{r\}) = \{p \cdot q, p \cdot r\}$. $(p \cdot q)\#(p \cdot r) = \{p \cdot q\} \cup \{p \cdot r\} = \{p \cdot q, p \cdot r\}$. So $\{p\} \cdot (q\#r) = (p \cdot q)\#(p \cdot r)$. $P \cdot (Q\#R) = P \cdot (Q \cup R) = \{p_i \cdot q_j, p_i \cdot r_k \mid p_i \in P, q_j \in Q, r_k \in R\}$. $(P \cdot Q)\#(P \cdot R) = \{p_i \cdot q_j \mid p_i \in P, q_j \in Q\} \cup \{p_i \cdot r_k \mid p_i \in P, r_k \in R\} = \{p_i \cdot q_j, p_i \cdot r_k \mid p_i \in P, q_j \in Q, r_k \in R\}$. So $P \cdot (Q\#R) = (P \cdot Q)\#(P \cdot R)$.

- According to Definition 3.6 and Definition 3.7, $\{p\} \parallel (q\#r) = \{p\} \parallel (\{q\} \cup \{r\}) = \{p \parallel q, p \parallel r\}$. $(p \parallel q)\#(p \parallel r) = \{p \parallel q\} \cup \{p \parallel r\} = \{p \parallel q, p \parallel r\}$. So $\{p\} \parallel (q\#r) = (p \parallel q)\#(p \parallel r)$. $P \parallel (Q\#R) = P \parallel (Q \cup R) = \{p_i \parallel q_j, p_i \parallel r_k \mid p_i \in P, q_j \in Q, r_k \in R\}$. $(P \parallel Q)\#(P \parallel R) = \{p_i \parallel q_j \mid p_i \in P, q_j \in Q\} \cup \{p_i \parallel r_k \mid p_i \in P, r_k \in R\} = \{p_i \parallel q_j, p_i \parallel r_k \mid p_i \in P, q_j \in Q, r_k \in R\}$. So $P \parallel (Q\#R) = (P \parallel Q)\#(P \parallel R)$.

□

B.2 Theorem 4.1

PROOF. *If part.* For the first condition, since a simple path ending with an end node is a path of the HMSC, if it is consistent, then the HMSC is weakly consistent.

For the second condition, we assume that a simple path $s_0 \dots s_{i-1} s_i \dots s_j$ is consistent and $s_i \dots s_j$ is a loop. Consider the path $s_0 \dots s_{i-1}(s_i \dots s_j)^\omega$, in which the loop is repeated infinitely. We use $s_{n,i}$ to represent the nth times that s_i is repeated. In the follows we prove by induction that this path is consistent.

Base. The simple path $s_0 \dots s_{i-1} s_i \dots s_j$ is consistent.

Induction. Assume $s_0 \dots s_{i-1}(s_i \dots s_j)^n$ is consistent. Then it has a trace Tr. We need to find a trace for $s_1 \dots s_{i-1}(s_i \dots s_j)^{n+1}$, equal to $s_1 \dots s_{i-1}(s_i \dots s_j)^n \times (s_{n+1,i} \dots s_{n+1,j})$.

Because events in the node $s_{n+1,k}$ has the same absolute and relative time constraints as events in the node $s_{1,k}$, and there exists a trace for $s_i \dots s_j$ (otherwise $s_0 \dots s_{i-1} s_i \dots s_j$ can not be consistent), we can get a trace Tr' for $s_{n+1,i} \dots s_{n+1,j}$. Since the absolute time constraints for all the events in $s_{n+1,i} \dots s_{n+1,j}$ have infinite upper bounds, we can increase time values in Tr' by n such that they are larger than all the time points in Tr, and still keep the consistency. Then TrTr' constitutes a trace for $s_1 \dots s_{i-1}(s_i \dots s_j)^{n+1}$. So it is also consistent.

Then we can conclude that the path $s_1 \dots s_{i-1}(s_i \dots s_j)^{\omega}$ is consistent.

Only if part. We only need to prove that if an HMSC is weakly consistent and the first condition of the proposition is not satisfied, then the second condition must be satisfied. We prove it by contradiction. There are two cases.

- Assume all the simple paths ending with a loop are not consistent. In such a case, all the simple paths are not consistent because it is assumed that all the simple paths ending with an end node are not consistent also. Then the HMSC is neither weakly nor strongly consistent. Contradiction.
- Assume in each consistent simple path ending with a loop, there is an event contained in the loop, whose absolute time constraint is [lb, ub] where ub is not infinity. In such a case, each loop can only repeat finite times consistently. So if there is a consistent path p, then p is a finite path ending with an end node. Since p cannot be a simple path, p contains some identical nodes caused by loops. Assume p goes through one loop. Then p can be represented by $s_1 \dots s_{i-1}(s_i \dots s_j)^n s_{j+1} \dots s_n$. In a trace Tr of p, if we remove all the timed events in $(s_{k,i} \dots s_{k,j})$ $1 < k \leq n$, the remaining part is still a trace, which corresponds to the simple path $s_1 \dots s_{i-1} s_i \dots s_j s_{j+1} \dots s_n$. So $s_1 \dots s_{i-1} s_i \dots s_j s_{j+1} \dots s_n$ is consistent. It contradicts with the assumption that all the simple paths ending with an end node are not consistent. Same contradiction can be obtained when p goes through more than one loop.

□

B.3 Theorem 4.2

PROOF. *If part.* Assume the HMSC is not strongly consistent. Then there is an inconsistent path p. Since all the simple paths are consistent, p can not be a simple path. Assume p is $s_1 \dots s_i s_{i+1} \dots$, in which nodes $s_1 \dots s_i$ are not included in any loop and s_{i+1} is included in a loop l. So all the nodes after s_{i+1} can be reached from the loop l. In the follows, we show that there is a trace for p.

Since all the nodes in $s_1 \dots s_i s_{i+1}$ are not identical, $s_1 \dots s_i s_{i+1}$ must be a prefix of a simple path, which is consistent. So there is a trace Tr1 for $s_1 \dots s_i s_{i+1}$. For the node s_{i+2}, since it must appear in some simple paths and all the simple paths are consistent, there exists a trace for s_{i+2}. Let it be Tr2 = $(e_1, t_1)\ (e_2, t_2)\ \dots (e_n, t_n)$. According to Corollary 3.2 in [19], t_i can be the upper bound of the reduced time constraint of e_i. If e_i is in the loop or causally after events in the loop, the upper bound of e_i is infinity. We can replace it with a value that is larger than all the time values of events e_j in Tr1 and Tr2 such that $e_j \leq e_i$, and time constraints are still satisfied. So Tr1Tr2 is a trace for $s_1 \dots s_i s_{i+1} s_{i+2}$.

Inductively, we can build traces for $s_1 \dots s_i s_{i+1} \dots s_k$ $(k \geq i+2)$ until $s_1 \dots s_i \times s_{i+1} \dots s_k$ is p. Then we get contradiction with the assumption that p is not consistent. So the HMSC is strongly consistent.

Only if part. We prove that the two conditions hold as follows.

- A simple path is a prefix of a path in the HMSC. So if all the paths in the HMSC are consistent, all the simple paths are consistent also.

- Assume the second condition of the proposition does not hold. Let e be an event that occurs in a loop or is causally ordered after events in the loop. The reduced time constraint of e does not have an infinite upper bound. Since the loop can be repeated infinitely, there exists a path in which the occurrence time of e will be larger than the upper bound of its reduced time constraint. However, if the occurrence time is outside of the reduce time constraint, then there does not exist a trace including this occurrence time according to the definition of reduced time constraints. So the path is not consistent. It contradicts with the strong consistency.

□

B.4 Proposition 4.1

PROOF. *If part.* Since M and N are consistent, they have traces. According to Corollary 3.2 in [19], we can find a trace Tr1 for M, in which every t_i is the lower bound of the reduced time constraint of e_i. We can also find a trace Tr2 for N, in which every t_i is the upper bound of the reduced time constraint of e_i. If the upper bound of the reduced time constraint of e_i is infinity, we can replace it with a value that is larger than any t_j such that $e_j \leq e_i$. Because of the second condition of the proposition, for the last event e at process i in M and the first event f at i in N, the occurrence of e in Tr1 is earlier than the occurrence of f in Tr2. Then Tr1Tr2 is a trace for $M \cdot N$. So $M \cdot N$ is consistent.

Only if part. We prove that the two conditions hold as follows.

- In a trace of $M \cdot N$, if we remove all the events in M(or N), the remaining part is a trace for N(or M). So M and N are consistent.
- Assume the second condition in the proposition does not hold. Then for the last event e at a process in M with reduced absolute time constraint $[a,\ b]$, the earliest time it

can occur is at a. For the first event f at the same process in N with reduced absolute time constraint $[c,\ d]$, the latest time it can occur is at d. Since it is assumed that $d \leq a$, f can only occur before e. However, because $M \cdot N$ is consistent and $e \leq f$, e should occur before f in any trace of $M \cdot N$. Contradiction.

□

B.5 Theorem 4.3

PROOF. According to Proposition 4.1, we can get $M \cdot N$ is consistent immediately.

Given consistent bMSCs M and N and their distance graphs $G_1 = (V_1 \cup \{e_0\}, E_1)$ and $G_2 = (V_2 \cup \{e_0\}, E_2)$, we construct a graph G for $M \cdot N$. G $= (V_1 \cup V_2 \cup \{e_0\}, E_1 \cup E_2 \cup E)$, in which $E = \{(g,\ h) \mid g$ is the first event in a process in N, and h is the last event in the same process in M$\}$. Since the relative time constraint between g and h is $[1,\ \infty)$, there is no edge from h to g, and the edge from g to h has the weight -1. For a node e in V_1 with reduced absolute time constraint $[a', b']$, the shortest distance from e_0 to e is b', and the shortest distance from e to e_0 is $-a'$. Since in G there are only edges from nodes in V_2 to nodes in V_1, there are no new paths from e to e_0 in G, which are not in E_1. So the shortest distance from e to e_0 is still $-a'$. The lower bound of the reduced absolute time constraint for e is not changed. Similarly for a node f in V_2, there are no new paths from e_0 to f in G, which are not in E_2. So the upper bound of the reduced absolute time constraint for f is not changed.

In the following, we prove that the upper bound of the reduced absolute time constraint for events in G_1 and the lower bound of the reduced absolute time constraint for events in G_2 are not changed if and only if the two conditions in the proposition are satisfied.

If part. Let a node e be in V_1 with reduced absolute time constraint $[a', b']$. For all the paths from e_0 to e,

- if all the nodes in a path are in V_1, then the length of the path is larger than b'.
- assume a path includes nodes in V_1 and V_2. Then there must be two events $g \in V_2$ and $h \in V_1$ in the path, and g is the first event in a process in G_2 with reduced absolute time constraint $[c,\ d]$ and h is the last event in the same process in G_1 with reduced time constraint $[a,\ b]$. The path can be written as $e_0 \ldots g\ h \ldots e$. Using D(e, f) to represent the distance between e and f, the length of the path is p = D(e_0, g) - 1 + D(h, e). Since the length of the shortest path from e_0 to g is d, $p \geq$ D(h, e) - 1 + d.
  - If d is infinity, we can get D(e_0, g) is infinity and p is also infinity immediately. Then $p \geq b'$. The shortest distance from e_0 to e is still b'.
  - If b and d are not infinity, since $b < d$ according to the condition, we get $d - 1 \geq b$ (our time domain is integer). So $p \geq$ D(h, e) + b. Since b is the (shortest) distance from e_0 to h, and h and e are both in G_1, D(h, e) + $b \geq b'$. So $p \geq b'$. Then the shortest distance from e_0 to e is still b'.

So the reduced absolute time constraints of nodes in V_1 are not changed. If we reverse the direction of all the edges in G, using the similar proof as above we can prove that for nodes in V_2, the condition $a < c$ can assure that their reduced absolute time constraints are not changed.

Only if part. Assume h is the last event in the a process in G_1 with reduced absolute time constraint $[a,\ b]$. If the reduced absolute time constraint of h is not changed in G, then the length of all the paths from e_0 to h is not less than b. Assume a path is $e_0\ g\ h$, in which g is the first event in the same process in G_2 with reduced absolute time constraint $[c,\ d]$. The length of the path is D(e_0, g) - 1. Since b is not larger than any value of D(e_0, g) - 1, if b is infinity, then d has to be infinity also. If b is not infinity, and we choose D(e_0, g) = d, then $b < d$. Similarly, we can prove $a < c$. □

B.6 Proposition 4.2

PROOF. *if part.* In each simple path $s_0 \ldots s_n$ of a time-disjoint FSPHMSC, s_0 and s_1 satisfy the conditions in Proposition 4.3 because s_1 is later than s_0, and s_0, s_1 are consistent. So $s_0 \cdot s_1$ is consistent. Since reduced absolute time constraints in s_1 are not changed in $s_0 \cdot s_1$, and s_2 is later than s_1, s_2 is later than $s_0 \cdot s_1$ also. Then $s_0 \cdot s_1 \cdot s_2$ is consistent. By induction, we can obtain that $s_0 \ldots s_n$ is consistent. So all the simple paths are consistent since all the bMSCs are consistent.

Only if part. For a consistent simple path $s_0 \ldots s_n$, there is a corresponding consistent lposet $(s_0 \cdot s_1 \ldots s_{n-1}) \cdot s_n$. According to Proposition 4.3, $s_0 \cdot s_1 \ldots s_{n-1}$ and s_n are consistent. By induction, every bMSC s_i in the simple path is consistent. Since all the simple paths cover all the bMSCs, all the bMSCs are consistent. □

B.7 Proposition 5.1

PROOF. For a bMSC represented by $(I, A, E, \leq, l, D, T)$, we can build two mappings $m_e : E \to E$, and $m_a : A \to A$ such that Definition 5.3 is satisfied. So a bMSC conforms to itself.

For transitivity, if $M_2 = (I_2, A_2, E_2, \leq_2, l_2, D_2, T_2)$ conforms to $M_1 = (I_1, A_1, E_1, \leq_1, l_1, D_1, T_1)$, then there are two injective mappings $m_{e12} : E_1 \to E_2$, and $m_{a12} : A_1 \to A_2$ such that Definition 5.3 is satisfied. Similarly, for M_2 and M_3, if M_3 conforms to M_2, then there exist $m_{e23} : E_2 \to E_3$, and $m_{a23} : A_2 \to A_3$ such that Definition 5.3 is satisfied. We construct two functions, $m_e : E_1 \to E_3$ and $m_a : A_1 \to A_3$, which are the composition of m_{e23} and m_{e12}, m_{a23} and m_{a12} respectively. That is, $m_e = m_{e23} \cdot m_{e12}$, $m_a = m_{a23} \cdot m_{a12}$. Then for two events e and f in M_1, we have:

- If $e \leq_1 f$, since M_2 conforms to M_1, we get $m_{e12}(e) \leq_2 m_{e12}(f)$. Furthermore, because M_3 conforms to M_2, we get $m_{e23}(m_{e12}(e)) \leq_3 m_{e23}(m_{e12}(f))$. So $m_e(e) \leq_3 m_e(f)$.

- $m_a(l_1(e)) = m_{a23}(m_{a12}(l_1(e))) = m_{a23}(l_2(m_{e12}(e))) = l_3(m_{e23}(m_{e12}(e))) = l_3(m_e(e))$.

- If e is a sending event, and $D_1(e)$ is not $(0, \infty)$, we can get $D_1(e) \supseteq D_2(m_{e12}(e))$. Since $m_{e12}(e)$ is still a sending event, we get $D_2(m_{e12}(e)) \supseteq D_3(m_{e23}(m_{e12}(e))) = D_3(m_e(e))$. So $D_1(e) \supseteq D_3(m_e(e))$.

- If e is a receiving event, and $D_1(e)$ is not $(0, \infty)$, we can get $D_1(e) \subseteq D_2(m_{e12}(e))$. Since $D_2(m_{e12}(e))$ is also not $(0, \infty)$ according to the condition of the proposition, we get $D_2(m_{e12}(e)) \subseteq D_3(m_{e23}(m_{e12}(e))) = D_3(m_e(e))$. So $D_1(e) \subseteq D_3(m_e(e))$.

- Similarly to absolute time constraints, we can get $T_1(e, f) \supseteq T_2(m_e(e), m_e(f))$ if f is a sending event, and $T_1(e, f) \subseteq T_2(m_e(e), m_e(f))$ if f is a receiving event.

So M_3 conforms to M_1. □

B.8 Theorem 5.1

PROOF. Since we keep the road map unchanged, for each path p in H_1, there is a corresponding path p' in H_2. We prove p' conforms to p in the follows.

- Since H_2 is strongly consistent, p' is consistent.

- Due to the second and the third conditions of the proposition, according to Theorem 1 in [56], all the events and orders in p are preserved in p'.

- For a reduced absolute time constraint t in a bMSC in p, since each bMSC in p' conforms to its corresponding bMSC in p, the corresponding reduced absolute time

constraint t' in the bMSC in p' satisfies the relation in Definition 5.2 with t. Because the HMSC is upper-bound-later, the upper bounds of t and t' are not changed in the lposets representing p and p' according to Theorem 4.3 (page 76). Since we do not change the lower bounds of reduced absolute and relative time constraints, the lower bounds of t and t' are same in the lposets representing p and p'. So reduced absolute time constraints in p and p' still satisfy the defined relation.

- For a reduced relative time constraint between events within a bMSC in p, since each bMSC in p' conforms to its corresponding bMSC in p, the corresponding reduced relative time constraint t' in a bMSC in p' satisfies the defined relation with t. Because the events are within one bMSC, the relation is still kept in the lposets representing p and p'.

- For a reduced relative time constraint t between events in different bMSCs (t is obtained implicitly from relative time constraints in the same bMSC and the delay between two bMSCs), since the delay between two bMSCs is assumed as $(0, \infty)$, the upper bound of t is always ∞. Because we do not change the lower bound of reduced relative time constraints when refining an HMSC, the lower bound of t is not changed. So t is not changed after refinement.

Thus p' conforms to p. □